Предложения по развитию системы управления пакетами RPM

Алексей Турбин <at@altlinux.org>

12 апреля 2012 г.

Содержание

Введение

Ян Мёрдок (основатель проекта Debian) назвал управление пакетами «самым выдающимся достижением, которое Linux привнес в IT-отрасль».[1] Мёрдок отмечает, в частности, возможность унифицированного обновления ПО по сети, а также модель распределенной разработки, которая стала возможна благодаря разбиению операционной системы на компоненты.

Нужно заметить, однако, что возможность обновления пакетов по сети реализуется в виде надстройки (apt, yum) над базовым уровнем управления пакетами (rpm, deb), тогда как модель распределенной разработки является не столько техническим, сколько социальным аспектом разработки свободного ПО. В то же время функции, реализуемые на базовом уровне управления пакетами, подозрительно похожи на функции архиватора типа tar, который существовал в UNIX всегда. Действительно, rpm-пакеты можно рассматривать как «тарболлы», а rpm — как некоторое подобие tar (по крайней мере, в части установки пакетов). Более того, в некоторых системах «тарболл» и является основным форматом пакетов. Возникает вопрос: каковы главные особенности системы управления пакетами на базовом уровне, делающие ее принципиально отличной от архиватора? Что можно считать «достижением» на базовом уровне?

На первый взгляд, главным отличием rpm от архиватора является возможность обновления и удаления пакетов. В отличие от обычного архиватора, rpm поддерживает базу пакетов `rpmdb`, в которой хранится информация об установленных пакетах и о файлах, ассоциированных с пакетами. При обновлении пакетов эта информация позволяет корректно выполнить замещение файлов и т. п. Впрочем, нужно понимать, что некоторая сложность, связанная с корректным замещением файлов, возникает вследствие использования традиционной иерархии файловой системы, при которой файлы в зависимости от их назначения располагаются в разных системных каталогах. Некоторые разработчики считают традиционную иерархию файловой системы слишком сложной. Так, в дистрибутиве GoboLinux выполнена полная реорганизация файловой системы: каждому пакету отводится собственный подкаталог в системном каталоге `/Programs`. Это радикально упрощает обновление пакетов, сводя его к удалению старого каталога и распаковке нового архива.

На наш взгляд, принципиальное отличие управления пакетами от «управления тарболлами» состоит в другом. Архиватору безразлично содержимое файлов в архиве, распаковка — это его единственная функция; сохранность байтов — это всё, что его в принципе интересует. В то же время перед rpm стоит более сложная задача: поддержка операционной системы в целостном, согласованном состоянии. Фактически, базовый уровень управления пакетами должен гарантировать работоспособность операционной системы (в той степени, в которой это вообще возможно) и обеспечивать совместимость установленных компонентов ПО. Возможность обновления ПО по сети, отмеченную Мёрдоком, можно интерпретировать именно так — как поддержку ОС в актуальном рабочем состоянии.

Таким образом, можно сказать, что базовый уровень управления пакетами является «достижением» настолько, насколько он *не сводится* к функциям архиватора.

[1] *The single biggest advancement Linux has brought to the industry.* `http://ianmurdock.com/solaris/how-package-management-changed-everything/`

В частности, в какой-то момент система управления пакетами должна весьма сильно интересоваться содержимым файлов, входящих в пакет, т. к. работоспособность операционной системы и совместимость компонентов ПО во многих случаях могут быть объяснены в терминах содержимого тех или иных файлов. Если же не интересоваться содержимым файлов, то гарантировать их согласованную совместную работу, по-видимому, никак нельзя.

В rpm основным способом описания работоспособности и совместимости пакетов являются *зависимости*. При сборке пакетов rpm изучает содержимое файлов, входящих в пакет, и формирует зависимости пакета. Зависимости Requires описывают требования, которые предъявляются к другим пакетам. Зависимости Provides в свою очередь описывают «предоставляемые» свойства пакета, которые могут быть «затребованы» через Requires в других пакетах. Соответственно, в rpm имеются вспомогательные программы `find-requires` и `find-provides`, которые выполняют поиск Requires и Provides зависимостей. Эти две программы составляют основу *подсистемы автоматического поиска зависимостей* (эта подсистема также может быть использована для автоматического поиска зависимостей типа `Reqires(post)` в %post-скриптах, о чем подробнее в разделе 9). Большая часть данного документа посвящена усовершенствованию этой подсистемы. Как будет видно в дальнейшем, многие требования работоспособности и совместимости пакетов могут быть описаны гораздо более точно. Так, в разделе 7.2 описан способ компактного представления множества символов, с помощью которого можно в значительной степени решить проблему т. н. «dll hell» — проблему версионирования и неполной совместимости разделяемых библиотек.

Интересно отметить «асимметричный» характер системы управления пакетами на базовом уровне: содержимое файлов изучается только во время сборки пакетов для формирования зависимостей. На стадии же установки rpm руководствуется зависимостями, полученными при сборке; а содержимое файлов его снова не интересует. В терминах дихотомии Эрика Реймонда[2] можно сказать, что при установке пакетов работает базовый *механизм*, тогда как *политика* управления пакетами на самом деле осуществляется во время сборки.

Политика сборки пакетов не обязательно должна ограничиваться анализом файлов для формирования зависимостей. Можно реализовать более агрессивную политику сборки, направленную, во-первых, на автоматическую коррекцию содержимого пакетов, а во-вторых, на автоматическую верификацию, которая должна быть выполнена, чтобы пакет успешно собрался. Процедуры верификации могут выполняться, в частности, на стадии brp (buildroot policy). Примером коррекции содержимого пакетов является процедура `brp-debuginfo`, которая используется для подготовки пакетов с отладочной информацией (см. раздел 11). Некоторые процедуры автоматической верификации рассмотрены в разделе 12.

Однако не все проверки стоит выполнять при сборке пакета. Некоторые проверки являются специфическими для дистрибутива и поэтому не могут быть частью базовой политики сборки пакета (например, сюда можно отнести формальное требование указывать в %changelog «официальный» e-mail адрес, связанный с проектом). Другие проверки не требуют доступа к содержимому файлов — например,

[2] *Rule of Separation: Separate policy from mechanism; separate interfaces from engines.* `http://www.faqs.org/docs/artu/ch01s06.html`

проверка путей на соответствие FHS (filesystem hierarchy standard). Другой пример: архитектурно-независимые пакеты (noarch) не должны содержать архитектурно-зависимых путей (файлов в каталоге `/usr/lib64` и подкаталогах). Такие проверки можно оформить в виде отдельной программы типа rpmlint.

Между прочим, последняя проверка является во многих отношениях недостаточной: в noarch пакетах не только не должно быть архитектурно-зависимых путей, но и вообще при сборке на любой архитектуре должны получаться *идентичные* noarch пакеты. Для выполнения такой проверки требуется синхронная сборка пакета на двух (или более) архитектурах, т.е. эта проверка не может быть реализована на уровне rpm, а только на уровне сборочной системы (если сборочная система синхронизирует сборку для нескольких архитектур). В сборочной системе одного российского дистрибутива реализована именно такая — достаточно строгая — политика верификации noarch пакетов.

Мы завершаем введение кратким обзором сборочной системы.[3] Приоритетной задачей сборочной системы является автоматическое тестирование и верификация пакетов во время и после сборки, т. е. политика сборки, направленная на поддержку целостности репозитория пакетов (в то время как, например, наличие веб-интерфейса для нас не представляет интереса). Сборочная система характеризуется относительно высоким процентом отказов в приеме пакетов (при обнаружении дефектов в пакетах). В то же время сборочная система работает полностью автоматически, и пакеты без дефектов поступают в репозиторий сразу же (проверка вручную и дополнительное тестирование могут быть организованы перед публикацией репозитория на официальном сервере).

Политику сборочной системы можно описать в терминах *формальной модели*: поступление новых пакетов моделируется как переход репозитория из состояния A_0 в состояние A_1. При этом, в идеале, в любом состоянии должно быть выполнено условие, которое можно назвать *инвариантом перехода*: все пакеты должны успешно проходить тестовую пересборку, и все пакеты должны успешно (т. е. без нарушения зависимостей и т. д.) устанавливаться в базовую среду (basesystem). В реальности контролировать инвариант перехода удается лишь частично, т. к. тестовая пересборка всех пакетов требует довольно много вычислительных ресурсов. Как следствие, нереализованной остается принципиально важный компонент сборочной системы — *метарепозиторий*, который позволил бы наиболее точно учитывать взаимное влияние между пакетами и, в частности, контролировать изменение свойств пакета после тестовой пересборки (а не только сам факт прохождения пересборки).

Интерфейс сборочной системы позволяет формировать *задания* на сборку. Задание может содержать несколько пакетов, которые собираются в строгой последовательности (без изменения основного репозитория). После этого моделируется переход $A_0 \to A_1$ и, если переход разрешен, задание *транзакционно* применяется к репозиторию. Таким образом, сборочная система позволяет проводить несовместимые изменения — такие, как изменение имени библиотеки — при условии синхронной пересборки (в этом же задании) всех несовместимых/зависимых пакетов.

К недостаткам сборочной системы можно отнести некоторые ограничения на возможность параллельной сборки, т. к. параллельные переходы не моделируются.

[3] Подробнее см. *Сборочная система git.alt* в сборнике `http://www.altlinux.ru/media/book-thesis-Protva-2008-5.pdf`

1 Раскрытие макросов в спекфайле

Макросы в rpm являются базовым механизмом параметризации. С помощью макросов можно задать как простейшие текстовые подстановки, так и более сложные (в том числе рекурсивные) подстановки с аргументами. В спекфайле «вызов» макроса выполняется с помощью конструкций `%{name}`, `%{name args}` или (в сокращенной записи) `%name`, `%name args`.

Однако использование макросов сопряжено с некоторой опасностью: когда rpm встречает неизвестный макрос (т. е. одну из конструкций указанного вида, в которой макрос с именем `name` не был заранее определен), rpm оставляет макрос «нераскрытым», т. е. сохраняет всю конструкцию в неизменном виде, как если бы она не считалась «вызовом» макроса. При этом rpm не выводит предупреждений, с помощью которых можно было бы обнаружить нераскрытые макросы. Это порождает целый класс ошибок в пакетах.

Например, в пакете `gnumeric-1.10.17-1-mdv2012.0.x86_64.rpm` содержится следующий %preun-скрипт:

```
SCHEMAS=""
for SCHEMA in %schemas ; do
  SCHEMAS="$SCHEMAS /etc/gconf/schemas/$SCHEMA.schemas"
done
GCONF_CONFIG_SOURCE=‘/usr/bin/gconftool-2 --get-default-source‘ \
/usr/bin/gconftool-2 --makefile-uninstall-rule $SCHEMAS >/dev/null || true
```

Макрос `%schemas` в этом скрипте остался нераскрытым. Т. к. %preun-скрипт выполняется при удалении пакета, то при удалении будет выполнен вызов `gconftool-2` с неправильными аргументами, который должен завершиться с ошибкой (однако эта ошибка игнорируется с помощью конструкции `>/dev/null || true`).

Другой пример: в пакете `stardict-3.0.1-10-mdv2011.0.x86_64.rpm` имеется %preun-скрипт

```
%preun_install_gconf_schemas stardict
```

При удалении такого пакета возникнет ошибка вида `fg: no job control` или `fg: no such job`, т. к. интерпретатор `/bin/sh` считает такую синтаксическую конструкцию обращением к фоновому заданию.

Ещё один пример: пакет `squid-3.1.16-1-mdv2012.0.x86_64.rpm` содержит %postun-скрипт

```
/usr/share/rpm-helper/del-user squid $1 squid
%post cachemgr
%postun cachemgr
```

Природа ошибки в данном случае более тонкая — эта ошибка не сводится к простому «человеческому фактору», к которому можно отнести неправильные названия макросов и опечатки в названиях макросов.

Таким образом, чтобы обнаружить целый класс ошибок и повысить надежность сборки пакетов, мы считаем принципиально важным диагностировать (в том или

ином виде) нераскрытые макросы во время сборки. В дальнейшем также будет показано, что определения многих макросов, специфичных для некоторой группы пакетов, могут быть вынесены в отдельный файл и находиться в отдельном пакете (см. раздел 3). Это открывает еще одну возможность для появления нераскрытых макросов: нераскрытые макросы могут появиться из-за нарушения зависимостей или вследствие недостаточных `BuildRequires` зависимостей у пакетов. Поэтому, прежде чем выносить определения макросов в отдельные файлы, необходимо реализовать защиту от ошибок, к которым это может привести.

Некоторая сложность в реализации проверки на нераскрытые макросы связана с тем, что rpm не делает специального различия между «ключевыми словами» и сокращенной записью макросов (например, rpm позволяет определить макрос с именем `build`, в результате чего будет нарушена структура спекфайла — исчезнет секция `%build`). Кроме того, некоторые «ключевые слова» являются специфичными для отдельных секций спекфайла (например, атрибут `%ghost` используется только в секции `%files`). Поэтому в текущей реализации проверка выполняется немного по-разному в зависимости от секции спекфайла (проверка реализована на уровне функции `parseSpec`).

Как уже было сказано, если определения макросов, используемых в спекфайле, содержатся в отдельном пакете, то в спекфайл должна быть добавлена зависимость `BuildRequires` на этот пакет. Однако, чтобы извлечь эту зависимость, нужно распарсить спекфайл, который содержит нераскрытые макросы. Поэтому желательно реализовать два режима проверки: строгий и мягкий. В *строгом* режиме работа rpm завершается с ошибкой, если нераскрытые макросы обнаружены по крайней мере в некоторых секциях спекфайла (таких как %post-скрипты). Строгий режим используется по умолчанию при сборке пакета. В *мягком* режиме ошибки заменяются на предупреждения, что позволяет распарсить спекфал и извлечь зависимости.

2 Оптимизация зависимостей

Большая часть предложений в данном документе направлена на совершенствование зависимостей у пакетов; в том числе предполагается реализация новых типов зависимостей. Зависимостей станет много. Это не только увеличивает размер пакетов, но и повышает нагрузку на сборочную систему, увеличивает время, необходимое для проверки зависимостей при установке пакетов и т. п. Поэтому, прежде чем добавлять новые — полезные — зависимости, следует рассмотреть способы оптимизации лишних зависимостей — зависимостей, которые можно считать бесполезным.

Прежде всего, можно реализовать *слияние* «похожих» зависимостей внутри отдельно взятого пакета. Например, зависимость с версией `Provides: foo = 1.0` делает ненужной зависимость без версии `Provides: foo`, а из двух зависимостей с версиями `Requires: bar >= 1.0` и `Requires: bar >= 2.0` можно оставить только одну — в данном случае с наибольшей версией. В общем виде, с учетом всевозможных атрибутов зависимостей, алгоритм слияния может быть довольно сложным.

Однако главным источником лишних зависимостей являются автоматически сгенерированные зависимости `Requires`, которые могут быть разрешены внутри своего пакета в соответствующие зависимости `Provides`. Это связано с тем, что

скрипты генерации зависимостей `Requires` обычно не учитывают то, что некоторые из этих зависимостей предоставляются тем же самым пакетом (и поэтому всегда будут удовлетворены). Вместо того, чтобы модифицировать скрипты генерации зависимостей, лучше реализовать глобальную оптимизацию: исключать из пакета зависимости Requires, которые удовлетворяются зависимостями Provides в этом же пакете. Эта оптимизация, однако, не может быть распространена на зависимости с некоторыми флагами типа `Requires(pre)` — опять же, корректная реализация должна учитывать всевозможные специальные случаи.

В качестве примера рассмотрим зависимости пакета `perl-XML-SAX-0.960.0-2-mdv2011.0.noarch.rpm`.

```
Requires: perl(XML::SAX::PurePerl::DTDDecls)
Requires: perl(XML::SAX::PurePerl::DocType)
Requires: perl(XML::SAX::PurePerl::EncodingDetect)
Requires: perl(XML::SAX::PurePerl::Productions)
...
Provides: perl(XML::SAX::PurePerl::DTDDecls)
Provides: perl(XML::SAX::PurePerl::DebugHandler)
Provides: perl(XML::SAX::PurePerl::DocType)
Provides: perl(XML::SAX::PurePerl::EncodingDetect)
Provides: perl(XML::SAX::PurePerl::Exception)
Provides: perl(XML::SAX::PurePerl::Productions)
...
```

Большую часть `Requires` зависимостей в этом пакете следует отнести к лишним зависимостям — а именно, почти все зависимости вида `perl(XML::SAX::...)`.

Оптимизация зависимостей может не ограничиваться зависимостями в пределах одного пакета: оптимизацию можно распространить на подпакеты, собираемые из одного исходного пакета, в том случае, если подпакеты связаны строгой зависимостью. *Строгой зависимостью* мы называем зависимость вида `Requires: %name = %version-%release`, которая требует базовый подпакет с указанием не только версии, но и релиза (номера сборки). Тогда, если сборочная система не принимает пакеты без последовательного увеличения либо версии, либо релиза, то можно считать, что строгая зависимость уникально идентифицирует базовый подпакет.

Идея оптимизации состоит в том, что наличие строгой зависимости позволяет оптимизировать некоторые другие — нестрогие — зависимости. В качестве пример рассмотрим зависимости пакета `bzip2-1.0.6-3-mdv2011.0.x86_64.rpm`.

```
Requires: lib64bzip2_1 = 1.0.6-3
Requires: libbz2.so.1()(64bit)
Requires: libc.so.6()(64bit)
```

Здесь первая зависимость является строгой — она уникально идентифицирует базовый подпакет с библиотекой сжатия. В таком случае можно упразднить вторую зависимость — на имя (soname) библиотеки, т. к. эта зависимость предоставляется базовым подпакетом, на который уже имеется строгая зависимость.

Вообще, если пакет `%name-foo` содержит строгую зависимость на некоторый подпакет `%name-base`, то это позволяет удалить из пакета `%name-foo` два рода `Requires` зависимостей:

- Зависимости, которые *представляются* пакетом `%name-base`; или же, несколько точнее, зависимости, которые могут быть удовлетворены `Provides` зависимостями пакета `%name-base`.

- Зависимосте, которые *уже требуются* пакетом `%name-base`; или же, несколько точнее, зависимости, которые могут быть подчинены `Requires` зависимостям пакета `%name-base` в смысле возможности слияния.

В примере с `bzip2` зависимость `Requires: libc.so.6()(64bit)` относится к зависимостям второго рода — базовый подпакет `lib64bzip2_1` содержит такую же зависимость.

Некоторая сложность при оптимизации зависимостей второго рода связана с возможностью строгих *циклических* зависимостей. *Наивная* оптимизация зависимостей второго рода может привести к полному — «циклическому» — удалению зависимостей у подпакетов, которые образуют цикл.

Строгие зависимости между подпакетами можно наделить *транзитивностью*: если подпакет `foo` содержит строгую зависимость на подпакет `bar`, а подпакет `bar` содержит строгую зависимость на подпакет `baz`, то можно считать, что подпакет `foo` содержит строгую зависимость на пакет `baz` (и оптимизировать зависимости пакета `foo` не только относительно пакета `bar`, но и `baz`).

Может возникнуть вопрос: в какой степени строгие зависимости действительно являются строгими? Не стоит ли реализовать еще более строгие зависимости типа `build-id`, которые идентифицируют пакет по его содержимому? Можно указать два сценария, при которых строгие зависимости оказываются недостаточно строгими:

- Какой-либо посторонний пакет предоставляет зависимость вида `Provides: %name = %version-%release` и, таким образом, удовлетворяет строгую зависимость Requires.

- Сборка пакета выполняется без увеличения релиза (возможно, для другого дистрибутива), и возникает два набора пакетов. Строгие зависимости не предотвращают «перемешивания» пакетов из обоих наборов.

В обоих случаях оптимизация зависимостей может оказаться некорректной, т. к. оптимизация выполняется в предположении, что строгая зависимость уникально идентифицирует базовый пакет и его зависимости (а оказывается, что базовый пакет можно «подменить»).

На наш взгляд, возможность подмены базового пакета является скорее гипотетической, и на практике строгие зависимости можно считать достаточно строгими. Существует, однако, один дефект в реализации rpm, который может сделать строгие зависимости менее строгими: сравнение версий у зависимостей выполняется таким образом, что зависимость вида `Provides: foo = %version` (с версией, но без релиза) удовлетворяет любую зависимость вида `Requires: foo = %version-%release` (с такой же версией и с релизом). По-видимому, алгоритм сопоставления зависимостей в rpm должен быть скорректирован.

3 Модульная система поиска зависимостей

На последних стадиях сборки пакета выполняется автоматический поиск зависимостей: rpm запускает скрипты `find-requires` и `find-provides`, которые анализируют содержимое файлов в пакете и формируют соответственно `Requires` и `Provides` зависимости.

Почти во всех дистрибутивах скрипты `find-requires` и `find-provides` являются монолитными программами, т. е. добавление новых типов зависимостей требует непосредственной модификации этих скриптов. Задача *модульной системы поиска зависимостей*[4] состоит в том, чтобы для каждого типа зависимостей выделить соответствующие программы поиска зависимостей и сделать возможным независимое добавление зависимостей новых типов. В скриптах `find-requires` и `find-provides` должен остаться только вспомогательный код диспетчеризации.

Рассмотрим фрагмент монолитного скрипта `find-requires`:

```
# --- Grab the file manifest and classify files.
exelist=‘echo $filelist | xargs -r file | \
        grep -Ev ":.* (commands|script)[, ]" | \
        grep ":.*executable" | cut -d: -f1‘
scriptlist=‘echo $filelist | xargs -r file | \
        grep -E ":.* (commands|script)[, ]" | cut -d: -f1‘

# --- Executable dependency sonames.
  for f in $exelist; do
    [ -r $f -a -x $f ] || continue
    lib64=‘if file -L $f 2>/dev/null | \
        grep "ELF 64-bit" >/dev/null; then echo "$mark64"; fi‘
    ldd $f | awk ’/=>/ {
        if ($1 !~ /libNoVersion.so/ && $1 !~ /4[um]lib.so/) {
            gsub(/’\’’"/,"\\&",$1);
            printf "%s’$lib64’\n", $1
        }
    }’
  done | xargs -r -n 1 basename | sort -u
```

Данную схему поиска зависимостей можно описать так: сначала rpm классифицирует все имеющиеся файлы и формирует разные категории файлов — *отбирает* файлы для поиска зависимостей, причем основным инструментом классификации является программа `file(1)`. Затем для каждой группы выполняется анализ файлов и формируются соответствующие зависимости (в приведенном фрагменте отбираются исполняемые программы и далее с помощью `ldd(1)` определяются библиотеки, необходимые для запуска этих программ).

Скрипт `find-requires` можно сделать модульным, если для каждого типа зависимостей выделить две программы: программу *отбора* файлов и программу *анализа*

[4] См. также *Автоматический поиск зависимостей в rpm-пакетах* в сборнике `http://www.altlinux.ru/media/protva4.pdf`

файлов. Тогда после реорганизации работу скрипта `find-requires` можно пояснить следующим образом:

- Утилита `file(1)` запускается только один раз — в скрипте `find-requires`. Таким образом, скрипт `find-requires` формирует список файлов и их «типов». Дальнейшая работа `find-requires` сводится к вызову программ поиска зависимостей. Для определенности будем рассматривать зависимости на разделяемые библиотеки.

- Первая программа — с именем `lib.req.files` — по списку файлов и их «типов» отбирает файлы, которые содержат зависимости на разделяемые библиотеки. Список отобранных файлов возвращается в `find-requires`.

- Затем `find-requires` запускает вторую программу — с именем `lib.req` — которая выполняет анализ отобранных файлов и формирует список зависимостей на разделяемые библиотеки.

Модульность данной схемы состоит в том, что `find-requires` запускаются все имеющиеся программы поиска зависимостей (по шаблону `*.req.files` и `*.req`) из каталога `/usr/lib/rpm`. Скрипт `find-provides` работает аналогична (запускаются программы с суффиксом `.prov` вместо `.req`). Таким образом, для добавления нового типа зависимостей нужно написать четыре программы: программу отбора файлов и программу анализа файлов, соответственно для Requires и Provides.

Другим аспектом модульности является возможность, в дополнение к новому типу зависимостей, реализовать макросы, предназначенные для некоторой группы пакетов. Как правило, такие макросы используются для того, чтобы упростить сборку и унифицировать некоторые конструкции в спекфайлах (см. макросы для perl-пакетов в разделе 6.2). Макросы также могут использоваться для того, чтобы управлять некоторыми параметрами сборки.

Макросы могут располагаться в отдельных файлах и подключаться по такому же модульному принципу. Рассмотрим два вида файлов с макросами — файлы с суффиксом `.def` и файлы с суффиксом `.env`. Файлы с суффиксом `.def` являются традиционными файлами определения макросов. Например, в файле `perl.def` определен следующий макрос, который позволяет в «императивном стиле» задать дополнительные каталоги расположения perl-модулей:

```
%add_perl_lib_path()  %global _perl_lib_path %{?_perl_lib_path} %*
```

У файлов с суффиксом `.def` другое назначение — они позволяют «вклиниваться» в некоторые стадии сборки пакетов с достаточно произвольным кодом, который генерируется с учетом текущего макроконтекста в спекфайле. Например, в файле `perl.env` имеется следующая строка:

```
%{?_perl_lib_path:export RPM_PERL_LIB_PATH="%_perl_lib_path"}
```

Таким образом, если в спекфайле был задан макрос `_perl_lib_path`, то его значение экспортируется в виде переменной окружения `RPM_PERL_LIB_PATH`. В дальнейшем значение этой переменной учитывается в скриптах поиска зависимостей `perl.req` и `perl.prov`.

4 Файловые зависимости и процедура find-package

Файловыми зависимостями мы называем зависимости вида `Requires: /usr/foo`, в которых в качестве имени зависимости используется путь к файлу (или каталогу). В rpm реализована полная поддержка файловых зависимостей. А именно, файловая зависимость может быть удовлетворена двумя способами:

- если пакет предоставляет соответствующую зависимость `Provides: /usr/foo`;
- если в пакете запакован файл (или каталог) `/usr/foo`.

Таким образом, преимуществом файловых зависимостей является то, что их не нужно явно «предоставлять» — они предоставляются «по факту» наличия файлов в пакете. Исключением являются «альтернативы», то есть символические ссылки, создаваемые при установке программой `update-alternatives` (такие пути должны быть предоставлены через `Provides`).

Другим преимуществом файловых зависимостей является то, что во многих случаях они являются наиболее точным выражением `Requires` зависимостей. Например, если в каком-либо perl-скрипте имеется shebang-инструкция `#!/usr/bin/perl`, то работоспособность такого скрипта напрямую зависит от наличия интерпретатора `/usr/bin/perl`. Вместо зависимости на `/usr/bin/perl` можно было бы указать зависимость на пакет `perl-base` (или `perl`, если в дистрибутиве нет пакета `perl-base`), но такая зависимость оказалась бы менее точной, т. к. она не гарантирует напрямую наличие интерпретатора `/usr/bin/perl` (кроме того, как уже видно, не всегда можно установить однозначное соответствие между интерпретатором и именем пакета).

Хуже того, при сборке некоторых пакетов генерируются shebang-инструкция с указанием версии интерпретатора: `#!/usr/bin/perl5.14.2`. Если вместо файловой зависимости для такой инструкции сгенерировать зависимость на `perl-base`, то эта зависимость окажется не только неточной, но и очень хрупкой. Зададимся вопросом: что будет при обновлении perl до новой версии? Если была сгенерирована файловая зависимость, то при обновлении у пакетов с такой инструкцией появится неудовлетворенная зависимость (и пакеты нужно будет пересобрать). Если же сгенерировать зависимость на имя пакета, то обновление пройдет без нарушения зависимостей, но пакеты утратят работоспособность — скрипты перестанут запускаться ввиду отсутствия интерпретатора! А это довольно неприятное последствие — получается, что зависимости, которые должны давать хотя бы минимальную гарантию работоспособности программы, на самом деле не дают и минимальной гарантии.

Таким образом, мы считаем, что файловые зависимости использовать не только уместно, но и необходимо во всех случаях, когда зависимость может быть однозначно выражена в виде пути к файлу.

Хотя rpm полностью поддерживает файловые зависимости на базовом уровне (на уровне проверки зависимостей), некоторые системы на основе rpm не поддерживают файловые зависимости как следует — не учитывают того, что файловые зависимости могут разрешаться через обычные файлы, а не только через `Provides`. Дело в том, что при работе с репозиторием пакетов общий список файлов во всех пакетах может оказаться довольно большим, и его обработка представляет некоторую сложность. Рассмотрим два характерных случая:

- Система доступа к репозиторию вообще не учитывает файлов, запакованных в пакете, а учитывает только явные `Provides`. В системах такого рода нужно либо добавить «распознавание» файлов, либо вместо этого добавлять некоторые файловые пути прямо в `Provides`. Однако, например, `apt-rpm` изначально поддерживает файловые зависимости, так что для корректной работы не требуется модификации клиентских программ.

- Файловые зависимости не учитываются при «индексировании» репозитория, т. е. список файлов игнорируется при создании метаинформации. Как ни странно, этот случай характерен для `apt-rpm`, и в одном российском дистрибутиве пришлось довольно серьезно переделать процедуру генерации репозитория. В таких системах порождение метаинформации должно выполняться в два прохода: на первом проходе надо искать файловые зависимости, чтобы на втором проходе сохранить частичные списки файлов.

В связи с генерацией репозитория нужно учитывать еще один важный случай:

- Последовательная оверлейная сборка пакетов. Пусть мы собираем пакеты на репозитории с корректной, но частичной поддержкой файловых зависимостей. Собранные пакеты помещаются в оверлейный репозиторий, который также используется при сборке. Тогда при сборке пакета A может образоваться файловая зависимость, для разрешения которой не хватает информации в основном репозитории. После этого не может быть выполнена сборка пакета B, который требует для сборки пакет A, т. к. пакет A не может быть установлен. В общем случае поддержка оверлейной сборки (двух и более пакетов) требует, чтобы основной репозиторий был перегенирирован с полной информацией о файлах (именно такой подход используется в одном российском дистрибутиве).

Итак, сборочная система и система доступа к репозиторию должны как следует поддерживать файловые зависимости.

Файловые зависимости генерируются не только для shebang-инструкций (см. раздел 8.4), но и, например, при обработке символических ссылок (см. раздел 8.6). Для порождения файловых зависимостей разработана вспомогательная библиотека шелл-функций `/usr/lib/rpm/find-package`. Изначально библиотека была предназначена для анализа зависимостей в шелл-скриптах, поэтому основная процедура `FindPackage` используется как для порождения файловых зависимостей, так и зависимостей для *команд*, т. е. для программ, вызываемых по имени (без указания полного пути программы).

При порождении файловых зависимостей в `find-package` выполняется *каноникализация* пути. При этом должны быть учтены некоторые особенности сопоставления путей в каталоге `%buildroot` и в хост-системе. Например, во многих дистрибутивах путь `/etc/init.d` является символической ссылкой, указывающей на каталог `/etc/rc.d/init.d`. Тогда, если в каком-либо скрипте загружается файл `/etc/init.d/functions`, то с учетом символической ссылки `/etc/init.d` должна быть сгенерирована файловая зависимость на `/etc/rc.d/init.d/functions`. В то же время содержимое каталога `%buildroot` должно иметь приоритет над содержимым корневой файловой системы.

Генерация зависимостей для команд рассмотрена в следующем разделе.

5 Зависимости шелл-скриптов

Подробнее остановимся на анализаторе зависимостей шелл-скриптов `shell.req`, т. к. при его реализации возникают вопросы, в разной степени характерные для многих других методов поиска зависимостей.

Первичный анализ скрипта выполняется интерпретатором `/bin/sh` с опцией `--rpm-requires`, которая задействует специальный режим анализа кода, производный от режима проверки синтаксиса `sh -n`. Таким образом, дополнительным преимуществом `shell.req` следует считать синтаксическую проверку кода, выполняемую при поиске зависимостей. Обнаружение синтаксических ошибок в скриптах можно отнести к дополнительному тестированию, выполняемому при сборке (если `shell.req` обнаруживает синтаксическую ошибку, то сборка пакета завершается с ошибкой). Проверка синтаксиса особенно важна для %post-скриптов (см. раздел 9).

Особенностью шелл-скриптов является отсутствие специального механизма для указания требований (такого, как подключение модулей в других языках программирования) — фактически, каждая *команда* может порождать некоторую зависимость. Вместе с отсутствием специального механизма исключений, который мог бы использоваться для условной загрузки кода, это обостряет проблему порождения *условных зависимостей*. Например, для кода

```
if [ -f /etc/foo.conf ]; then
        . /etc/foo.conf
fi
if [ -f /usr/local/foo.conf ]; then
        . /usr/local/foo.conf
fi
```

будут порождены файловые зависимости на `/etc/foo.conf` и `/usr/local/foo.conf`, при том, что обе зависимости являются условными, а вторая — недопустимой (т. к. иерархия `/usr/local` не должна использоваться в пакетах).

Существует несколько способов, с помощью которых можно избежать порождения условных зависимостей:

- Можно переписать код таким образом, чтобы зависимость не порождалась. Для этого можно «спрятать» команду или значение аргумента в переменную

  ```
  foo=/usr/local/foo.conf
  if [ -f $foo ]; then
          . $foo
  fi
  ```

  либо дополнить команду специальным присваиванием, которое распознается анализатором как инструкция игнорировать команду:

  ```
  if [ -f /usr/local/foo.conf ]; then
          a= . /usr/local/foo.conf
  fi
  ```

Заметим, что в силу особенностей реализации интерпретатора зависимости в *подстановках команд*, т. е. зависимости в конструкциях вида `$(cmd ...)` также игнорируются.

- Можно отфильтровать ненужные зависимости, используя специальные макросы в спекфайле (см. раздел 10).

- Для пакетов категории `contrib` (содержащих скрипты, которые не предназначены для непосредственного запуска пользователем) поиск шелл-зависимостей можно отключить: `AutoReq: yes, noshell`. Однако для пакетов, не относящихся к категории `contrib`, полностью отключать `shell.req` не следует, т. к. желательно сохранить синтаксическую проверку. Вероятно, для такого класса пакетов нужно реализовать специальный режим `AutoReq: yes, shell=syntax-check-only`.

Итак, синтаксический анализ позволяет обнаружить *команды*, используемые в скрипте (аргументы некоторых стандартных команд, как в примере с `foo.conf`, также считаются командами). Если команда имеет вид *пути к файлу*, то для такой команды `find-package` порождает файловую зависимость; оставшиеся *простые* команды подразделяются анализатором на два вида: шелл-функции и исполняемые файлы.

Для простых команд выполняется *элиминация зависимостей на функции*: формируется глобальный список шелл-функций, которые определены в каком-либо файле в пределах пакета. В дальнейшем команды с названием этих функций исключаются из поиска зависимостей. Дело в том, что интерпретатор не всегда может заранее определить, является ли вызываемая команда функцией или исполняемым файлом. Для реализации этой стадии требуется модифицировать режим `--rpm-requires` таким образом, чтобы в качестве зависимостей выводились не только вызываемые, но и все определяемые функции.

Оставшиеся команды считаются исполняемыми файлами, которые должны быть расположены в стандартных каталогах `PATH`. При формировании зависимостей для таких команд возникает следующее противоречие: с одной стороны, команды могут перемещаться между каталогами (например, программу можно переместить из каталога `/usr/bin` в каталог `/bin`), и тогда в качестве зависимости лучше использовать имя пакета. В то же время файлы могут перемещаться между подпакетами (например, программы из базового подпакета могут быть перенесены в дополнительный подпакет), и тогда зависимость на имя пакета не гарантирует наличия команды. Можно было бы использовать отдельное пространство имен для исполняемых команд — `command(foo)`, однако при этом теряется различие между `sbin`-командами, доступными только пользователю `root`, и `bin`-командами, доступными всем пользователям. Поэтому сейчас реализован компромиссный вариант, в котором по умолчанию генерируется зависимость на имя пакета.

Однако команда не всегда может быть однозначно сопоставлена с именем пакета. Если приоритетный путь `/usr/bin/foo` предоставляется двумя пакетами с разным именем (обычно такой путь является «альтернативой», хотя возможны и конфликтующие реализации), то нужно сгенерировать файловую зависимости на `/usr/bin/foo`. Если же два разных пути находятся в одном пакете (например,

/bin/foo и /usr/bin/foo — обычно один из путей в таком случае является символической ссылкой), то лучше сгенерировать зависимость на имя пакета. При этом *приоритет путей* может быть разным: для скриптов, расположенных в каталогах /sbin, /usr/sbin и в некоторых других системных каталогах поиск команд выполняется в порядке /sbin, /bin, /usr/sbin, /usr/bin, а для остальных скриптов — в порядке /bin, /sbin, /usr/bin, /usr/sbin. Как видно, различие между sbin-скриптами и bin-скриптами не является строгим — каталоги /sbin и /usr/sbin используются в обоих случаях.

Понятно, что результат поиска команды может зависеть не только от пакетов, установленных при сборке. В частности, по установленным пакетам нельзя определить, для каких команд имеются конфликтующие реализации. Поэтому при сборке пакета сборочная система формирует *глобальный индекс команд* для всех пакетов в репозитории, который мы называем contents_index_bin. Алгоритм find-package, как правило, использует этот индекс в приоритетном порядке (чтобы сформировать достаточно общие зависимости для команд, имеющих альтернативные и конфликтующие реализации).

Однако существует несколько реальных случаев, как в следующем примере, когда у команды отличается как путь, так и название пакета.

```
/usr/bin/arpsend        arpsend
/usr/sbin/arpsend       vzctl
```

В таких случаях с помощью contents_index_bin уже нельзя сформировать зависимость достаточно общего вида — придется делать выбор в пользу одной из конкретных реализаций. Поэтому в таких случаях приоритет при поиске снова отдается установленным пакетам, т. к. нужная реализация, вероятно, установлена и используется при сборке пакета.

Необходимо заметить, что использование глобального индекса нарушает принцип *воспроизводимости* сборки: результат сборки пактов должен зависеть только от исходного кода и от пакетов, используемых при сборке (т. е. от пакетов, установленных в сборочный чрут для «чистовой» сборки). При использовании глобального индекса команд получается, что зависимости пакета могут в некоторой степени зависеть и от других пакетов в репозитории. Поэтому, во-первых, политика дистрибутива должна ограничивать размещение «левых» команд в стандартных каталогах PATH. Во-вторых, если сборочная система строго контролирует инвариант перехода (см. введение) и, в частности, фиксирует любые изменения зависимостей у пакетов после тестовой пересборки, то нужно реализовать дополнительных механизм контроля зависимостей, которые могут измениться только из-за содержимого contents_index_bin.

В последнее время появились предложения по объединению каталогов /bin и /sbin с каталогами в иерархии /usr. В связи с этим некоторые подходы, описанные в данном разделе, могут быть пересмотрены.

6 Сборка perl-пакетов — rpm-build-perl

Модульный подход, обозначенный в разделе 3, позволяет независимо разрабатывать компоненты, дополняющие базовую политику сборки. Одним из таких компонентов является пакет `rpm-build-perl`. В этот пакет вынесены скрипты поиска зависимостей и еще несколько файлов, относящихся к языку perl или направленных на поддержку сборки perl-пакетов. При этом в пакет `rpm-build` добавлена зависимость на `rpm-build-perl` — таким образом, поддержка perl-зависимостей сохранена для всех пакетов. Некоторые другие расширения, однако, не могут быть внесены в базовую сборочную среду; для таких расширений должна быть продумана специальная схема поддержки зависимостей (см. `rpm-build-mono` в разделе 8.7).

Пакет `rpm-build-perl` содержит следующие компоненты:

- Скрипт формирования Requires зависимостей `perl.req` и соответствующий ему скрипт отбора файлов `perl.req.files`.
- Скрипт формирования Provides зависимостей `perl.prov` и соответствующий ему скрипт отбора файлов `perl.prov.files`.
- Скрипт `perl.clean` для очистики каталога `%buildroot` на стадии brp.
- Файл `perl.def` с макросами, которые используются при сборке perl-пакетов.
- Файл `perl.env` для управления параметрами сборки (см. раздел 3).

По сравнению с традиционной реализацией было изменено представление зависимостей для модулей: вместо `perl(File::Spec)` используются зависимости вида `perl(File/Spec.pm)`, т. е. вместо имени модуля используется соответствующие компоненты имени файла. Таким образом, выбрано унифицированное представление зависимостей для модулей и зависимостей вида `perl(getopts.pl)` и `perl(sys/ioctl.ph)`.

Кратко остановимся на некоторых особенностях сборки perl-пакетов. Базовая совместимость модулей с интерпретатором выражается через зависимости на каталоги (см. раздел 8.5), при этом не используется версионирование каталогов (механизм `inc_version_list` отключен при сборке интерпретатора). Вообще, структура каталогов для модулей подвергнута максимальному упрощению. Так, все модули из noarch пакетов располагаются в каталоге `/usr/share/perl5`, так что результат сборки таких пакетов, как правило, не зависит от версии интерпретатора. Однако для компилируемых модулей зависимости на каталог `/usr/lib/perl5` или `/usr/lib64/perl5` будет недостаточно. Процедура сборки perl-расширений модифицирована таким образом, чтобы загружаемые объекты `.so` компоновались с библиотекой libperl. А библиотека libperl в свою очередь наделяется именем (soname) вида `libperl-5.14.so`. Таким образом, бинарная совместимость модулей и интерпретатора контролируется стандартным способом — через зависимости на имя библиотеки, при этом также используются set-версии (см. раздел 7.2). Всё это позволяет сформировать более *точные* и менее *жесткие* зависимости, которые позволяют контролировать не только *обратную*, но и *прямую* совместимость (например, бинарную совместимость модуля, собранного с новой версией интерпретатора 5.14.2, с более старой версией интерпретатора 5.14.1).

6.1 Анализатор зависимостей perl.req

Загрузка модулей в языке perl выполняется с помощью функции `require` или инструкции `use` (которая сводится к вызову функции `require`). В традиционной реализации скрипт `perl.req` с помощью регулярного выражения ищет строки, которые начинаются со слов `require` или `use`, и выполняет дальнейший разбор этих строк.

Рассмотрим фрагмент традиционной реализации `perl.req`.

```
if (m/^(\s*)         # we hope the inclusion starts the line
    (require|use)\s+(?!\{)     # do not want 'do {' loops
    # quotes around name are always legal
    ['"]?([^; '"\t#]+)['"]?[\t; ]
    # the syntax for 'use' allows version requirements
    # the latter part is for "use base qw(Foo)" and friends special case
    \s*($modver_re|(qw\s*[(\/'"]\s*|['"])[^)\/"'\$]*?\s*[)\/"'])?
    /x)
) {
my ($whitespace, $statement, $module, $version) = ($1, $2, $3, $4);

# we only consider require statements that are flushed against
# the left edge. any other require statements give too many
# false positives, as they are usually inside of an if statement
# as a fallback module or a rarely used option
($whitespace ne "" && $statement eq "require") && next;

# skip if the phrase was "use of" -- shows up in gimp-perl, et al.
next if $module eq 'of';
```

Как видно, регулярное выражение нельзя назвать простым. Кроме того, регулярного выражения оказывается недостаточно. Язык perl характеризуется очень сложным синтаксисом — неоднородной системой кавычек и ограничителей, многострочными литералами here-documents, наконец, встроенной документацией pod. Не случайно последняя проверка в приведенном фрагменте «отсеивает» типичное вхождение слова «use» в предложения английского языка (в дальнейшем выполняется еще несколько похожих проверок). Обращает на себя внимание и первая проверка: она оставляет только строки без отступа (а строки с отступом считаются условными конструкциями). Однако отступы в языке perl не являются значащими. Кроме того, уровень вложенности не всегда означает условную конструкцию, как в следующем типичном примере:

```
BEGIN {
    use Exporter;
    @EXPORT = qw(foo bar);
}
```

Следует также отметить, что инструкции `use` выполняются всегда (на стадии загрузки кода), и поэтому понятие условных конструкций к ним не относится.

Таким образом, разбор кода на языке perl с помощью регулярных выражений нельзя считать надежным подходом. Вообще, как говорят хакеры, «Only perl can parse perl», т. е. надежный разбор кода на языке perl может быть выполнен только с привлечением средств самого интерпретатора perl.

К счастью, интерпретатор предоставляет доступ к внутреннему представлению кода. При загрузке кода интерпретатор формирует дерево опкодов (которое обладает как признакми синтаксического дерева, так и признаками байткода); в дальнейшем дерево «выполняется» с помощью отдельной процедуры (которую можно считать прототипом виртуальной машины). Модуль `B`, иногда называемый «компилятором», предоставляет доступ к дереву опкодов, а также позволяет отключить основную стадию исполнения кода (активизируя режим проверки синтаксиса `perl -c`). Модуль `O` иногда называют «интерфейсом компилятора» — он упрощает загрузку дополнительных модулей, работающих в «режиме компилятора».

Модуль `B::Concise` позволяет отобразить дерево опкодов:

```
$ perl -MO=Concise -e 'require Foo; print 1'
9  <@> leave[1 ref] vKP/REFC ->(end)
1     <0> enter ->2
2     <;> nextstate(main 1 -e:1) v:{ ->3
4     <1> require sK/1 ->5
3        <$> const[PV "Foo.pm"] s/BARE ->4
5     <;> nextstate(main 1 -e:1) v:{ ->6
8     <@> print vK ->9
6        <0> pushmark s ->7
7        <$> const[IV 1] s ->8
-e syntax OK
```

Дерево опкодов сохраняет всю важную информацию об исходном коде. Модуль `B::Deparse` позволяет реконструировать исходный код:

```
$ perl -MO=Deparse -e 'require Foo; print 1'
require Foo;
print 1;
-e syntax OK
```

Идея «нетрадиционной» реализации `perl.req` состоит в том, чтобы вместо регулярного выражения использовать для извлечения зависимостей вспомогательный модуль `B::PerlReq`, разработанный специально для этой цели:

```
$ perl -MO=PerlReq -e 'require Foo; print 1'
perl(Foo.pm)
-e syntax OK
```

Основным источником зависимостей в дереве опкодов являются опкоды `require`, которые соответствуют непосредственным вызовам функции `require` и инструкциям `use`. Однако к образованию зависимостей могут приводить и некоторые другие конструкции. Например, версионирование зависимостей осуществляется с помощью вызова статического метода `VERSION` — инструкция вида

```
use Module 1.0 qw(list);
```

на самом деле «раскрывается» в код

```
BEGIN {
    require Module;
    Module->VERSION(1.0);
    Module->import(list);
}
```

Поэтому в модуле `B::PerlReq` в качестве общего случая реализован анализ вызовов функций и статических методов (в том числе выполняется реконструкция константных аргументов). Это позволяет обнаруживать отложенную загрузку модулей вида

```
use autouse Foo => qw(bar baz);
```

за счет обработки вызова статического метода `autouse->import`:

```
$ perl -MO=PerlReq -e 'use autouse Foo => qw(bar baz)'
perl(autouse.pm)
perl(Foo.pm)
-e syntax OK
```

Анализ вызова функций и статических методов позволяет обнаружить довольно сложные способы образования зависимостей. Например, в следующих двух случаях

```
open $fh, ">", \$var;
IO::File->new(\$var, "w");
```

при открытии файла вместо имени файла передается ссылка на переменную `$var`. В таких случаях доступ к файлу на диске заменяется доступом к содержимому переменной `$var` (т. н. in-memory files). Примитивы доступа к таким псевдофайлам реализованы в модуле `PerlIO::scalar`, который загружается интерпретатором по мере необходимости. Соответственно, при обнаружении подобных конструкций модуль `B::PerlReq` порождает зависимость на `perl(PerlIO/scalar.pm)`.

Подробнее рассмотрим проблему условных зависимостей. Функция `require`, которая выполняет загрузку модулей, всегда возвращает истинное значение. Если же модуль загрузить не удается, то функция `require` генерирует исключение, которое, как правило, приводит к аварийному завершению программы. Единственный способ перехватить исключение — заключить вызов `require` в блок `eval`. Это определяет основное правило обработки условных зависимостей, принятое в модуле `B::PerlReq`: условными считаются только зависимости, которые возникают внутри блоков `eval` (такие зависимости игнорируются). В то же время не игнорируются, например, зависимости на модули, загружаемые внутри функций, если вызовы `require` в функциях не «защищены» блоком `eval`.

Одна из проблем при таком подходе, однако, связана с тем, что для условных зависимостей часто характерна «обратная логика». Рассмотрим следующий пример:

```
eval {
    require Foo;
};
if ($@) {
    # Foo not available, fall back to Bar
    require Bar;
}
```

В этом примере предпринимается попытка загрузить модуль `Foo`, а модуль `Bar` загружается только в случае, если модуль `Foo` загрузить не удалось. В соответствии с логикой обработки условных зависимостей будет сгенерирована зависимость `perl(Bar.pm)`. Однако «по смыслу» более предпочтительной была бы зависимость `perl(Foo.pm)` — если модуль `Foo` существует и его удается загрузить.

Понятно, что в общем случае проблему «обратной логики» условных зависимостей решить довольно сложно — потребовалось бы реализовать предсказание переходов. Однако в некоторых случаях, продолжая аналогию «Only perl can parse perl», можно рассчитывать на исполнение кода самим интерпретатором. Например, поскольку блоки `BEGIN` выполняются только один раз и в безусловном порядке, то при обработке блоков `BEGIN` можно просто проверять (через `%INC`), какие модули были фактически загружены интерпретатором. Аналогичная возможность может быть распространена на «основной код» модулей (код инициализации), если реализовать специальный режим загрузки модулей `modexec`, при котором не только загружается текст, но и выполняется код инициализации (режим `modexec` пока не реализован, однако трудности реализации не являются принципиальными). В то же время режим исполнения кода не может быть распространен на скрипты, т. к. выполнение скриптов обычно связано с побочными эффектами.

6.2 Макросы и автоматизация сборки

В данном разделе рассматривается сборка perl-модулей, распространяемых через CPAN. Сборка пакетов выполняется по одинаковой схеме:

```
%prep
%setup -q -n Foo-%version

%build
perl Makefile.PL INSTALLDIRS=vendor
make

%check
make test

%install
make install DESTDIR=%buildroot

%files
%perl_vendorlib/Foo*
```

В некоторых пакетах, однако, вместо `Makefile.PL` используется `Build.PL`; в таких пакетах сборка идет по другой схеме: вместо `Makefile` создается скрипт `Build`, который используется в качестве самостоятельной программы вместо `make(1)`.

Можно реализовать вспомогательные макросы, в которых выбор нужной схемы сборки будет выполняться автоматически. Тогда соответствующие секции спекфайла будут выглядеть следующим образом:

```
%build
%perl_build

%check
%perl_check

%install
%perl_install
```

В результате мы получаем не только максимальное *упрощение*, но и максимальную *унификацию* сборки perl-пакетов. А унификация важна сама по себе: лишние степени свободы далеко не всегда могут быть использованы для улучшений, зато часто являются источником «разнобоя». Последовательность команд для сборки, даже правильная, всегда может стать предметом для проверки. А если сборка полностью унифицирована, то в этом отношении в пакетах попросту «нечего исправлять».

Кроме того, что полученная конструкция является унифицированной, ее можно считать достаточно *надежной*: почти все пакеты содержат тесты, которые выполняются при сборке; зависимости должны обеспечить работоспособность пакета при установке; а также поиск зависимостей подразумевает синтаксическую проверку кода (режим `modexec` должен дополнительно гарантировать возможность загрузки модулей). Таким образом, получить в результате использования этой конструкции неработоспособный пакет довольно сложно.

Дальнейшая автоматизация сборки связана с возможностями, предоставляемыми CPAN. CPAN экспортирует информацию о модулях в хорошо структурированном виде, что позволяет вычислить список модулей, которые должны быть обновлены:

```
$ perl -MCPAN -e 'CPAN::Shell->r'
Package namespace  installed  latest  in CPAN file
AnyEvent                6.02    6.12  MLEHMANN/AnyEvent-6.12.tar.gz
Apache::DBI             1.10    1.11  PHRED/Apache-DBI-1.11.tar.gz
Archive::Extract        0.56    0.58  BINGOS/Archive-Extract-0.58.tar.gz
...
```

Следует отметить, что в любом современном дистрибутиве должно быть порядка 1000 perl-пакетов (пакеты, на которые существует «спрос»). Понятно, что автоматизация обновления perl-пакетов в таком случае становится довольно актуальной задачей. Однако системы автоматизированного обновления perl-пакетов, которая целиком бы нас устраивала, пока не создано. Дальнейшее обсуждение автоматизации сборки выходит за рамки данного документа. Заметим только, что при автоматизации должна быть сохранена возможность проверки пакета человеком — перед тем, как пакет будет направлен в репозиторий. Таким образом, *автоматизированное* обновление не означает *слепое* обновление.

7 Зависимости на ELF-библиотеки, set-версии

Исполняемые файлы и разделяемые библиотеки в формате ELF являются основной реализации пользовательского пространства ОС. Формат ELF также является основой спецификации *System V Application Binary Interface*, которая является составной частью UNIX и Linux стандартов. Следует отметить, что формат ELF реализует низкоуровневые возможности и не предоставляет механизма обработки ошибок и исключительных ситуаций: в нештатной ситуации — например, при обнаружении несовместимости — работа приложения аварийно завершается. Поэтому поддержку бинарной совместимости следует считать приоритетной задачей для системы управления пакетами. Перефразируя *тезис Силина*,[5] можно сказать, что расходы, затраченные на поддержку бинарной совместимости, окупаются.

В современных дистрибутивах Linux программы *динамически* компонуются с разделяемыми библиотеками. Запуск таких программ выполняется с помощью вспомогательной программы — загрузчика `ld.so`, который конструирует исполняемый образ программы (полный путь к загрузчику содержится в сегменте `PT_INTERP` программы). При этом происходит загрузка необходимых разделяемых библиотек (список библиотек содержится в таблице `DT_NEEDED` программы). Разделяемые библиотеки в свою очередь могут требовать другие разделяемые библиотеки. После того, как все требуемые библиотеки загружены, загрузчик выполняет проверку версионированных интерфейсов — наличие интерфейсов типа `GLIBC_2.4` (информация о которых содержится в записях `DT_VERDEF` и `DT_VERNEED`). После этого выполняется подготовка к запуску, и загрузчик передает управление в стартовую процедуру программы. *Разрешение символов* обычно выполняется во время работы программы: поиск функции по имени выполняется при первом обращении к функции.

Таким образом, чтобы обеспечить бинарную совместимость динамически компонуемых программ и разделяемых библиотек, rpm должен фактически следовать логике загрузчика `ld.so`. А именно, rpm должен контролировать следующие аспекты совместимости:

- Возможность загрузки необходимых разделяемых библиотек.
- Наличие необходимых версионированных интерфейсов.
- Разрешимость символов во время работы программы.

Традиционно rpm контролирует только первые два аспекта совместимости (особенности реализации обсуждаются далее в разделе 7.1). Последний аспект является новым: поддержка зависимостей, в которых учитывается информация о символах, была реализована относительно недавно (реализация рассмотрена в разделе 7.2).

7.1 Пространство имен библиотек

В традиционной реализации для всех библиотек используется глобальное пространство имен, независимо от каталога, в котором расположена библиотека, и без учета назначения библиотеки. Это порождает много ненужных Provides зависимостей.

[5] *Расходы, затраченные на управление виртуальной памятью, окупаются.* Игорь Силин — разработчик ОС «Дубна» для БЭСМ-6.

Например, пакет `perl-XML-LibXSLT-1.700.0-4-mdv2011.0.x86_64.rpm` предоставляет зависимость `Provides: LibXSLT.so()(64bit)`, которая соответствует файлу `/usr/lib/perl5/.../auto/XML/LibXSLT/LibXSLT.so`. Однако этот файл не является «библиотекой» общего назначения в том смысле, что он не предназначен для динамической компоновки и не будет загружаться через `DT_NEEDED`. Это файл является «плагином» для интерпретатора perl и будет загружаться через `dlopen(3)`. Поэтому он расположен в специальном каталоге, в котором его будет искать интерпретатор.

Эта проблема лишних зависимостей уже обратила на себя внимание. Так, в проекте Fedora реализован макрос для фильтрации зависимостей `%perl_default_filter`, который предлагается добавлять в спекфайлы perl-пакетов.

Проблему лишних зависимостей, однако, не следует понимать исключительно как проблему оптимизации. Зависимости имен библиотек неявно образуют пространство имен. Это пространство имен наделяется определенной семантикой: если какой-либо пакет требует зависимость вида `libfoo.so.1`, то он рассчитывает, что динамический загрузчик сможет загрузить эту библиотеку в конфигурации по умолчанию (т. е. библиотека должна быть найдена в стандартном системном каталоге); соответственно, если какой-либо пакет предоставляет зависимость `libfoo.so.1`, то библиотека `libfoo.so.1` должна стать доступна в конфигурации по умолчанию, т. е. располагаться в стандартном системном каталоге. С другой стороны, какой-либо пакет может содержать копию библиотеки `libfoo.so.1` в своем «приватном» каталоге (такая практика не поощряется, но в принципе пакет «имеет право» содержать в приватном каталоге любые библиотеки). Такие приватные библиотеки не должны влиять на разрешение системных зависимостей.

Таким образом, мы предлагаем использовать пространство имен вида `libfoo.so.1` только для библиотек, доступных в конфигурации по умолчанию, т. е. для библиотек, расположенных в каталогах `/lib`, `/usr/lib` (на архитектуре i686) и в каталогах `/lib64`, `/usr/lib64` (на архитектуре x86-64).

В некоторых случаях, однако, динамический загрузчик может загружать библиотеки из нестандартных каталогов — например, при использовании в исполняемом файле пути к библиотекам `RPATH`. Для указания зависимостей на такие библиотеки можно использовать файловые зависимости — зависимости вида `Requires: /usr/foo/libfoo.so.1` (см. раздел 4). Преимуществом таких зависимостей является то, что для их разрешения не требуется соответствующих Provides зависимостей — достаточно фактического наличия нужного файла в каталоге `/usr/foo`.

Недостатком же файловых зависимостей является отсутствие версионирования — в предложенной схеме с использованием простых файловых зависимостей невозможно выразить требования на версионированные интерфейсы и на символы. Поэтому в некоторых случаях файловые зависимости могут «наращиваться» соответствующими Provides зависимостями, сформированными на основе имени файла, т. е. зависимостями вида `Provides: /usr/foo/libfoo.so.1(FOO_1.0)`. При этом префикс `/usr/foo` можно также трактовать как пространство имен, которое отделяет зависимость `libfoo.so.1(FOO_1.0)` от стандартного пространства имен.

Для «наращивания» зависимостей в спекфайле с помощью макроса должен быть задан каталог с приватными библиотеками: `%add_findprov_lib_path /usr/foo`.

Если приватные библиотеки используются только внутри пакета (или внутри подпакетов одного пакета), то «наращивания» зависимостей не требуется: будет

сформирована файловая зависимость, которая в дальнейшем должна быть оптимизирована (в том числе и в подпакетах, которые связаны строгими зависимостями — см. раздел 2). Наращивание может быть полезным только в случаях, когда библиотеки, расположенные в нестандартном каталоге, активно используются другими пакетами. Однако такие случаи редки, и обычно такие библиотеки следует переносить в системный каталог.

К спорным особенностям реализации зависимостей можно отнести возможность использования дополнительных «стандартных» каталогов. С одной стороны, загрузчик `ld.so` предоставляет стандартную возможность добавления каталогов через файлы конфигурации `/etc/ld.so.conf` и `/etc/ld.so.conf.d/*`. С другой стороны, использование этой возможности требует дополнительного контроля конфигурации `ld.so`. Фактически требуется контролировать содержимое конфигурационных файлов `ld.so.conf` — чтобы в одном из них всегда был указан нужный каталог. Однако rpm допускает модификацию конфигурационных файлов.

7.2 Версионирование интерфейса, set-версии

При разработке ПО большое внимание уделяется обратной совместимости. *Обратная совместимость* означает, что новая версия разработанного компонента ПО будет работать в старом окружении — сохраняет совместимость. Однако обратная совместимость не может обеспечить все *реальные* требования совместимости. Вообще, при запуске программы не существует «обратной» совместимости, а существует *просто совместимость*. Если компоненты программы совместимы, то она будет работать. А в противном случае компоненты программы несовместимы, и программу следует считать неработоспособной.

Ситуации, когда обратной совместимости оказывается недостаточно, часто возникают при использовании разделяемых библиотек. Это связано с тем, что версии пакетов могут обновляться асинхронно и использоваться в разных комбинациях, не предусмотренных хронологией разработки. Поясним это на следующем примере. Разработчик библиотеки, добавляя новые функции в библиотеку, считает, что он сохраняет обратную совместимость — существующие программы будут работать с новой версией библиотеки. Однако пользователь репозитория пакетов обычно хочет обновить интересную ему программу. Тогда в комбинации «новая программа со старой библиотекой» могут образоваться неразрешимые символы (при обнаружении неразрешимых символов программа аварийно завершается).

Таким образом, требования обратной совместимости часто должны быть дополнены требованием «минимальной версии» библиотеки — такой версии, в которой реализованы все функции, используемые в программе. С помощью *версионированных интерфейсов* можно наделить набор новых функций специальной меткой вида `FOO_1.0`, которая в дальнейшем будет учитываться в зависимостях (т. е. появится зависимость на `libfoo.so.1(FOO_1.0)`, если используется какая-либо функция из интерфейса `FOO_1.0`). Однако следует отметить, что версионированные интерфейсы являются, *во-первых*, очень хрупкой конструкцией. Например, версионирование не очень строго учитывается при разрешении символов: загрузчик `ld.so` допускает не только разрешение неверсионированной ссылки в версионированный символ, но и наоборот, разрешение версионированной ссылки в неверсионированный символ

(чтобы сохранить возможность переопределения версионированных символов через `LD_PRELOAD`). Кроме того, версионированные интерфейсы не должны заполняться постепенно, а должны появляться «одновременно» с набором новых функций, что плохо согласуется с инкрементальной разработкой. Если же добавлять функции в интерфейс постепенно, то мы возвращаемся к прежней проблеме: наличие интерфейса оказывается недостаточным для того, чтобы гарантировать наличие нужных функций. Конечно, можно попытаться запретить использовать «нестабильные» версии библиотек с частично заполненными интерфейсами, но это не решает проблему гарантий на уровне зависимостей. Для использования промежуточных версий библиотек на самом деле требуется бóльшая *гранулярность* версионирования — в пределе, каждый новый символ библиотеки нужно снабжать отдельным интерфейсом (тогда наличие интерфейса будет напрямую гарантировать наличие символа).

Во-вторых, использование версионированных интерфейсов не получило широкого распространения (например, оно не используется в библиотеках GNOME и KDE). Отчасти это связано с тем, что информацию о версионированных интерфейсах нужно поддерживать «вручную», отчасти — с ограниченной совместимостью (версионирование поддерживается только в загрузчике `ld.so` из `glibc`). В одном российском дистрибутиве мы предприняли попытку самостоятельно поддерживать версионированные интерфейсы в пакетах с библиотеками. Однако такой подход приводит к тому, что программы, собранные в одном российском дистрибутиве, перестают запускаться в других дистрибутивах (т. к. требуются отсутствующие версионированные интерфейсы). Кроме того, как уже отмечено, поддержка версионированных интерфейсов не может быть полностью автоматизирована, и каждый раз при сборке новой версии пакета с библиотекой требуется работа по обновлению и квалифицированная проверка. В дальнейшем мы отказались от использования «самодельных» версионированных интерфейсов.

Рассмотренная проблема «минимальной версии» не является единственной проблемой, которая приводит к нарушению бинарной совместимости. Часто авторы библиотек сознательно допускают нарушение обратной совместимости, удаляя устаревшие функции в новых версиях библиотек. Авторы также могут считать, что они вправе изменять недокументированные функции и т. п. Как бы там ни было, в исследовании, выполненном в ИПС РАН,[6] удаление функций и глобальных переменных названо главной проблемой бинарной совместимости. Некоторые другие аспекты бинарной совместимости, такие, как изменение типа данных параметра в языке C++ или добавление спецификатора `static`, тоже связаны с разрешением символов.

Все эти соображения привели к разработке новой модели зависимостей,[7] направленной на «просто совместимость», т. е. такой модели, в которой напрямую гарантируется наличие нужных символов. В этой модели интерфейс, предоставляемый библиотекой, формализуется как множество символов P — множество функций и глобальных переменных библиотеки; а требуемая версия библиотеки формализуется как множество библиотечных символов R, используемых в программе. Программа совместима с библиотекой, если $R \subseteq P$. Несколько точнее, пусть имеется несколько

[6] См. *Автоматизированный анализ обратной совместимости Linux-библиотек* в сборнике `http://docs.altlinux.org/archive/conference/trubezh2010.pdf`

[7] См. *Комплементарное хеширование подмножеств* там же.

версий библиотеки с разновидностями интерфейса $P_0, P_1, \ldots$ и несколько версий программы, в которых используются наборы символов $R_0, R_1, \ldots$ Тогда версия библиотеки P_i совместима с версией программы R_j, если $R_j \subseteq P_i$.

В данной модели символы ассоциируются с библиотеками, то есть считается установленным локальное соответствие между библиотечными символами, используемыми в программе, и библиотеками. Формат ELF, вообще говоря, не требует такого соответствия: в программе содержится только список используемых библиотек и список неопределенных символов, а поиск символов в библиотеках не ограничивается. Другими словами, формат ELF допускает «перемешивание» символов между библиотеками. Таким образом, данная модель несколько ограничивает то, что изначально считается допустимым. Однако мы не считаем это серьезной проблемой. Во-первых, «перемешивание» символов между библиотеками еще не встречалось нам на практике. Во-вторых, есть основания думать, что символы в действительности *должны* быть ассоциированы с библиотеками, т. к. каждый символ является частью интерфейса библиотеки. Так, в *System V Application Binary Interface* в разделе 6 функции «приписываются» к библиотекам. В OpenSolaris реализована процедура разрешения символов «Direct binding», при которой символы разрешаются сразу в нужные библиотеки (эту процедуру можно трактовать и как оптимизацию, и как защиту от пересечения с пользовательскими символами).

Возникает также вопрос, как установить соответствие между символами и библиотеками. Для этого мы используем вспомогательную программу, основанную на `ldd(1)`, в которой фактически выполняется запуск загрузчика `ld.so` в специальном «отладочном» режиме. Загрузчик в этом режиме загружает разделяемые библиотеки, полностью выполняет разрешение символов и выводит «отладочную» информацию о загрузке библиотек и разрешении символов, но не передает управление в стартовую процедуру программы. Таким образом, как и в случае с perl-зависимостям, используется «эмпирический» подход: чтобы узнать, как «в действительности» разрешаются символы, нужно использовать «родной механизм». Использование «родного механизма» также связано с нашим пониманием гарантий, которые могут быть предоставлены системой управления пакетами. Дело в том, что при сборке пакета выполняется согласование кода пакета с окружением, в котором собирается пакет. Тогда саму сборку пакета можно рассматривать как момент *наибольшего согласования*. Действительно, во время сборки программа обычно хорошо совместима с теми библиотеками, с которыми она собирается; несовместимость появляется только при работе в других конфигурациях (с другими версиями библиотек). Поэтому использование «родного механизма» для разрешения символов можно считать частью *переноса гарантий*, достигнутых в момент наибольшего согласования, на другие конфигурации. А именно, зависимости будут гарантировать, что при запуске программы символы будут разрешаться в некотором смысле *не хуже*, чем во время сборки пакета.

Понятно, что предложенная модель является слишком громоздкой — количество символов исчисляется сотнями даже в простых случаях, а названия символов в языке C++ бывают длинные. Поэтому напрямую символы не могут быть внесены в зависимости. Но строго говоря, нам и не нужно хранить полный список символов, а нужно лишь уметь проверять, являются ли требуемые символы подмножеством предоставляемых. Возникает вопрос, нельзя ли

придумать такую процедуру хеширования двух множеств R и P, при которой сохраняется возможность проверить вложение $R \subseteq P$? Тогда для хранения информации о символах можно реализовать специальные *версии* rpm-зависимостей — т. н. set-версии, которые представляют собой захешированные множества символов: пакет с разделяемой библиотекой предоставляет зависимость вида `Provides: libfoo.so.1 = set:7f0252c3...`, а пакет, который использует библиотеку, требует зависимость `Requires: libfoo.so.1 >= set:3f5b289c...`

Пусть заданы множества R и P, и определен предикат $R \subseteq P$. *Схемой комплементарного хеширования* мы называем тройку $\langle H_R, H_P, \subseteq^* \rangle$, состоящую из двух функций хеширования $H_R(R) \to R^*$, $H_P(P) \to P^*$ и предиката $R^* \subseteq^* P^*$. При этом если $R \subseteq P$, то должно всегда выполняться и $R^* \subseteq^* P^*$. А если $R \not\subseteq P$, то $R^* \not\subseteq^* P^*$ выполняется с вероятностью $1 - \varepsilon$, где параметр ε задает одностороннюю ошибку, обусловленную потерей информации при хешировании. Односторонний характер ошибки (false positive) означает, что проверка зависимостей никогда не будет давать ложных срабатываний, но может пропускать некоторые «настоящие» ошибки. Другими словами, в худшем случае проверка не сработает.

Оказывается, комплементарное хеширование в чистом виде невозможно: а именно, для $\varepsilon \ll \frac{1}{2}$ нельзя придумать хеш фиксированной длины. Поскольку размер «хеша» растет пропорционально числу элементов множества, то можно кодировать каждый элемент отдельно, используя 16–32-битный хеш; а затем рассмотреть процедуру более эффективной упаковки элементов. Таким образом, мы приходим к следующему представлению множеств:

```
P: a b c d e f g h i j k l m n o p q r s t u v w ...
R:   b     f     j   l   o   q r   t u     ...
```

Строчными буквами здесь обозначены «хеши» символов — числовые значения, полученные при хешировании. Эти значения располагаются в возрастающем порядке, поэтому методом *слияния* проверку $R \subseteq P$ можно выполнить достаточно быстро — за линейное время.

Рассмотрим вопросы вероятности и коллизий. Если множество P состоит из n элементов, то при хешировании можно использовать $\lceil \log_2 n \rceil + 10$ битов на элемент (как для элементов P, так и для элементов R). Например, если $n = 1024$, то можно использовать 20-битный хеш; будем считать такую конфигурацию типичной. Тогда вероятность *простой коллизии*, обусловленной ошибкой ε, составит $2^{10}/2^{20} = 2^{-10} \approx 0.1\%$. Другими словами, проверка может не сработать, если требуемый символ отсутствует, но его хеш совпадает с каким-либо другим имеющимся символом. Вероятность такой коллизии — это число «занятых» элементов n по отношению к общему числу элементов универсума.

Существует еще одна возможность коллизий — коллизии внутри самого множества P, обусловленные парадоксом «дней рождения». При вставке элементов в m-битный хеш (т. е. по принципу $\bmod 2^m$) первая коллизия появляется в среднем через $2^{(m+1)/2}$ вставок, а матожидание общего числа попарных коллизий после n вставок равно[8]

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{ij}\right] = \frac{n(n-1)}{2^{m+1}} \approx \frac{n^2}{2^{m+1}}.$$

[8] Cormen et al., *Introduction to Algorithms* (3rd ed.), p. 133.

Другими словами, этот тип коллизий возникает из-за того, что два предоставляемых символа могут иметь одинаковый хеш. В типичной конфигурации таких символов должно быть немного: для $n = 2^{10}$ и $m = 20$ получаем $\mathrm{E}[X] = 1/2$. Но для «большого» множества символов $n = 2^{16}$ и $m = 26$ получаем уже $\mathrm{E}[X] = 32$ коллизии. Однако в пересчете на один символ вероятность коллизии составляет

$$\frac{\mathrm{E}[X]}{n} \approx \frac{n}{2^{m+1}} = \frac{n}{2^{\lceil \log_2 n \rceil + 11}} \approx 2^{-11} \approx 0.05\%.$$

Таким образом, несмотря на то, что число коллизий растет, в пересчете на символ вероятность коллизии «дней рождения» остается невысокой: она «подчинена» вероятности простой коллизии.

Рассмотрим теперь вопросы реализации. Реализация должна быть *практичной*, что в нашем понимании означает две вещи:

- Нужно получить наиболее короткое представление set-версий. Другими словами, нужна эффективная процедура упаковки и сериализации массива из n m-битных чисел, причем эффективность прежде всего оценивается по длине полученной строки: длина строки должна быть близка к информационно-теоретическому минимуму.

- Распаковка строк и восстановление массива чисел должны выполняться очень быстро. Другими словами, требуется эффективная обратная процедура, причем эффективность в данном случае оценивается по времени.

Рассмотрим возможность упаковки элементов в типичной конфигурации: набор из 1024 20-битных хешей можно рассматривать как выбор 2^{10} элементов из 2^{20} элементов, или же как выбор 1 сочетания из $\binom{2^{20}}{2^{10}}$ сочетаний. Чтобы сделать такой выбор, требуется $\log_2 \binom{2^{20}}{2^{10}} \approx 11710$ битов информации (если считать все сочетания равновероятными). Таким образом, оптимальный способ упаковки элементов может значительно снизить размера хеша — вместо 20 битов на элемент в упакованном виде потребуется примерно 11.44 бита на элемент. В нашей реализации требуется примерно 1.95 алфавитно-цифровых буквы, или же примерно 11.6 бита на элемент (разреженные R-версии требуют несколько больше — например, для упаковки 32 20-битных хешей нужно уже 16.5 битов на элемент, но экономия все еще заметна).

Упаковка выполняется в несколько этапов:

- Массив чисел сортируется, дубликаты удаляются.

- Выполняется *дельта-кодирование*: возрастающая последовательность чисел заменяется последовательными разностями чисел.

- К полученной последовательности «маленьких чисел» применяется *код Голомба–Райса*: младшие биты чисел кодируются в обычном бинарном коде, а старшие биты — в унарном коде переменной длины. Поскольку большинство чисел являются «маленькими», то их унарные части оказываются короткими. В результате получается оптимальная[9] по длине битовая последовательность, кодирующая массив разностей.

[9] Putze et al., *Cache-, Hash- and Space-Efficient Bloom Filters*. `http://algo2.iti.uni-karlsruhe.de/singler/publications/cacheefficientbloomfilters-wea2007.pdf`

- Битовая последовательность сериализуется в кодировке «Base62», то есть, наконец, образуется строка, состоящая из символов набора `0-9A-Za-z`.

Несмотря на то, что при распаковке не используется дорогих операций (таких, как деление в цикле), первоначальная «наивная» реализация распаковки работала недостаточно быстро. Чтобы ускорить распаковку, были использованы следующие техники оптимизации:

- Реализована комбинированная стадия распаковки «Base62» и Голомба–Райса. При распаковке строки битовая последовательность не восстанавливается полностью; вместо этого распакованные биты поступают в конвейер, который сразу формирует массив разностей.

- На каждом шаге распаковываются сразу две буквы. Для этого «кусочки строки» приводятся к типу `unsigned short`, и далее используется вспомогательная таблица, которая упрощает обработку «двухбуквенных значений» (таблица содержит 65536 элементов, но из них используются только $62 \cdot 62 = 3844$, так что таблица хорошо умещается в кэше процессора).

- Для Provides-версий реализован LRU-кэш. Кэш содержит 160 слотов и занимает 1.3–2 Мбайт, коэффициент попаданий — 67%.

В результате проверка всех версий в репозитории занимает менее 1 с. user time (проверка всех версий выполняется, например, при обновлении базы `apt`, когда все требуемые версии зависимостей сопоставляются с предоставляемыми версиями).

Отметим еще одну важную особенность реализации. При обновлении библиотеки число символов может перешагнуть очередную отметку 2^m, в результате чего изменится количество битов на символ: $m_1 = \lceil \log_2 |P_1| \rceil + 10 > m_0 = \lceil \log_2 |P_0| \rceil + 10$. При этом у пакетов в репозитории зависимости на библиотеку сформированы со старым значением m_0. Однако благодаря тому, что хеш вычисляется по принципу $\bmod 2^m$ (т. е. фактически берутся младшие биты более длинного хеша), а не по принципу mod prime (как принято в некоторых других случаях), сравнение версий с несовпадающим количество битов все равно можно выполнить (правда, с некоторой потерей точности). Для этого значения в P_1 нужно «подрезать», сохранив только m_0 битов, и выполнить повторную сортировку. При сборке пакетов с новой версией библиотеки P_1 формирование зависимостей уже будет выполняться с параметром m_1. Таким образом, можно считать, что «выравнивание точности» зависимостей происходит автоматически, но с некоторой задержкой.

Попробуем подытожить наши рассуждения. Мы решили не полагаться на обратную совместимость, а напрямую контролировать наличие нужных символов в нужных библиотеках. Для этого мы реализовали вероятностную схему проверки вложения множеств $R \subseteq P$. Задав достаточно низкую «посимвольную» вероятность ошибки $\varepsilon \approx 0.1\%$, мы получили относительно короткое представление set-версий — примерно 2 буквы на символ. Конечно, данное решение является компромиссным, и поэтому уязвимо для критики: с одной стороны, вероятность ошибки не настолько низкая, чтобы считать, что зависимости дают полную гарантию; с другой стороны, версии получаются не слишком-то короткими — для 1024 символов длина строки будет почти 2 Кбайта. Вместе с тем, реализация является во многих отношениях оптимальной, так что серьезно улучшить этот компромисс, по-видимому, невозможно.

8 Другие методы поиска зависимостей

Стандартная политика сборки включает в себя более 20 методов поиска зависимостей; кроме того, имеется более 20 дополнительных методов поиска зависимостей, которые используются только при сборке определенной группы пакетов.

Основная задача автоматического поиска зависимостей — обеспечить работоспособность компонентов ПО. Для программ условия работоспособности очевидны — программа должна запускаться и т. п. В некоторых других случаях, однако, понятие работоспособности нуждается в уточнениях. В таких случаях лучше говорить о *факторах работоспособности*. Рассмотрим в качестве примера библиотечные `devel` пакеты — пакеты со вспомогательными файлами для компиляции и компоновки. Т. к. эти пакеты обычно не нужны для работы программ, а только для сборки, то возникает вопрос: какие зависимости должны быть у `devel` пакетов? Можно выделить следующие факторы работоспособности `devel` пакетов:

- Возможность включения заголовочных файлов. Если заголовочные файлы в свою очередь включают другие заголовочные файлы, то включение внешних по отношению к пакету заголовочных файлов должно быть поддержано соответствующими зависимостями.

- Возможность компоновки с библиотекой. Файл `libfoo.so`, который используется при компоновке с опцией `-lfoo`, является символической ссылкой, причем ссылка обычно указывает не на имя библиотеки `libfoo.so.0`, а на файл с `libtool`-версией `libfoo.so.0.0.0`. Поэтому обычно `devel` пакет должен содержать строгую зависимость на пакет с библиотекой. Однако в некоторых случаях строгая зависимость нежелательна, и, вообще говоря, возможность компоновки должна быть поддержана зависимостью, которая будет гарантировать, что символическая ссылка `libfoo.so` указывает на существующий файл.

- Если в пакете имеется `pkgconfig`-файл `foo.pc`, то зависимости на другие компоненты подсистемы `pkgconfig`, указанные в поле `Requires` этого файла, должны быть перенесены в зависимости `devel` пакета.

- Кроме того, если `pkgconfig`-файл `foo.pc` содержит поле `Libs`, то указанные в этом поле опции компоновки с внешними библиотеками `-lbar -lbaz` должны быть поддержаны зависимостями на файлы `libbar.so` и `libbaz.so`.

Из рассмотренного примера ясно, что работоспособность компонентов ПО можно в общем случае понимать как возможность использовать компоненты по прямому назначению, при этом даже в минимальной конфигурации не должно возникать ошибок, говорящих о том, что для полноценного использования не хватает каких-либо других компонентов ПО.

Нежелательными, однако, являются *транзитивные зависимости*: так, в зависимостях нужно учитывать только те заголовочные файлы, которые включаются напрямую, но не учитывать последующие файлы, которые включаются «в свою очередь». Если бы транзитивные зависимости были допустимы, то реализация сводилась бы к построению *транзитивного замыкания* с помощью *трассировки*: полный список требуемых файлов можно получить в результате фактической операции включения.

8.1 cpp.req

Итак, можно реализовать довольно смелую идею: формировать зависимости, которые обеспечивают возможность включения заголовочных файлов, расположенных в каталоге `/usr/include` и подкаталогах, за счет анализа директив `#include`. Поскольку заголовочные файлы часто содержат условные конструкции и т. п., то для анализа нужно использовать «родной механизм» — препроцессор `cpp(1)`. За счет выполнения фактической операции включения (препроцессирование можно рассматривать как включение на «нулевом уровне») препроцессор позволяет определить список используемых файлов. Однако, как уже отмечено, к зависимостям следует относить только те файлы из списка, которые были включены непосредственно (на первом уровне). К счастью, в директивах для компилятора, которые выводит препроцессор, содержатся не только имена файлов и номера строк, но и дополнительная информация,[10] которая позволяет реконструировать стек включения файлов. Тогда в зависимостях нужно учитывать только файлы, которые включаются при пустом стеке.

Существуют, однако, заголовочные файлы, которые не предназначены для непосредственного включения и, более того, содержат защиту от непосредственного включения. Например, файл `/usr/include/gtk-2.0/gtk/gtkaccessible.h` содержит следующую конструкцию:

```
#if !defined (__GTK_H_INSIDE__)
#error "Only <gtk/gtk.h> can be included directly."
#endif
#include <atk/atk.h>
#include <gtk/gtkwidget.h>
```

Этот файл является «составной частью» файла `<gtk/gtk.h>`, а при попытке его непосредственного анализа работа `cpp(1)` завершится с ошибкой. В то же время при анализе `<gtk/gtk.h>` будут учтены только непосредственно включаемые файлы. Таким образом, зависимость на `<atk/atk.h>` будет потеряна.

Каким образом можно избежать потери зависимостей в таких случаях? С одной стороны, можно попытаться подавить директиву `#error`. С другой стороны, директива `#error` не срабатывает, если включение происходит в файле `<gtk/gtk.h>`. Так что лучше будет расширить анализ зависимостей `<gtk/gtk.h>` на стадии реконструкции стека включений файлов. А именно, если файл, включенный на первом уровне, является «внутренним», то нужно также учитывать файлы на втором уровне и т. д., вплоть до первого файла на «внешнем» уровне. В нашем примере рассмотренный файл будет внутренним по отношению к `<gtk/gtk.h>`, а `<atk/atk.h>` — первым «внешним» файлом, который и должен учитываться в зависимостях. Вообще, к «внутренним» можно отнести файлы, которые при сборке находятся в `%buildroot`, хотя более корректная реализация должна учитывать различие между подпакетами.

Итак, с помощью препроцессора удается установить, что директива `#include <atk/atk.h>` приводит к включению файла `/usr/include/atk-1.0/atk/atk.h`. Использовать файловую зависимость в данном случае не очень хорошо, потому путь содержит подкаталог с версией, причем подкаталог задан в другом файле — `atk.pc`. Поэтому формируется зависимость на имя пакета `libatk-devel`.

[10] См. `info cpp 'Preprocessor Output'`

8.2 pkgconfig.req

Рассмотрим другие зависимости, характерные для `devel` пакетов: зависимости подсистемы `pkgconfig`. Эта подсистема часто используется в скриптах `configure` для определения расположения заголовочных файлов и т. п. Во многих дистрибутивах уже используются такие зависимости — зависимости вида `pkgconfig(foo)`, поэтому остановимся только на некоторых особенностях реализации.

В подсистеме `pkgconfig` существуют два типа зависимостей: `Requires` и `Requires.private`. Смысл зависимостей `Requires` примерно соответствует зависимостям `Requires` в `rpm`, а зависимости `Requires.private` изначально предназначались для указания компонентов, которые используются в реализации, но не влияют интерфейс (эти компоненты могут потребоваться для статической компоновки).

В дальнейшем различие между `Requires` и `Requires.private` привело к грандиозной неразберихе. Дело в том, что зависимости `Requires` влияют как на включение заголовочных файлов при компиляции (параметр `Cflags`), так и на подключение библиотек при компоновке (параметр `Libs`). Во многих случаях, однако, требуется только включение файлов (для использования макросов и т. п), а компоновка с библиотекой приводит к порождению лишней зависимости. Поэтому появилась идея использовать для таких случаев зависимости `Requires.private`. Идея получила некоторое распространение и была узаконена в одной из последних версий `pkg-config`: в режиме `--cflags` наличие зависимостей `Requires.private` стало обязательным. Поэтому в некоторых дистрибутивах стали также учитывать `Requires.private` зависимости — фактически, на уровне межпакетных зависимостей различие между `Requires` и `Requires.private` стерлось.

Мы предлагаем следующее решение этой проблемы:

- Вопросы компоновки с библиотеками рассмотрены далее в разделе 13.4.
- Учитывать нужно только зависимости типа `Requires`.
- В тех случаях, когда зависимости `Requires.private` используются для подключения заголовочных файлов, зависимости на заголовочные файлы будут сформированы с помощью `cpp.req`.
- Нужно модифицировать `pkg-config` таким образом, чтобы в режиме `--cflags` зависимости `Requires.private` считались необязательными: имеющиеся компоненты должны учитываться, а отсутствующие компоненты не должны приводить к ошибке.

8.3 pkgconfiglib.req

Этот метод поиска зависимостей анализирует параметр `Libs` в файлах `pkgconfig`. Для каждой опции компоновки `-lfoo` выполняется поиск библиотеки `libfoo.so` и формируется соответствующая зависимость. При этом файловая зависимость оказывается не всегда желательной, т. к. путь к библиотеке, вообще говоря, контролируется через опцию `-L`. Поэтому, как и в случае с `cpp.req`, формируется зависимость на имя пакета с библиотекой.

Таким образом, зависимости у `devel` пакетов контролируются двумя способами: зависимости подсистемы `pkgconfig` отображаются в зависимости вида `pkgconfig(foo)`, а *фактические* зависимости, связанные с включением заголовочных файлов и компоновкой с библиотеками, представлены именем пакета `libfoo-devel`. Обычно эти два типа зависимостей должны соответствовать друг другу. Однако в некоторых пакетах для зависимости `pkgconfig(foo)` не находится соответствующий зависимости `libfoo-devel`. Как правило, это означает, что в `pkgconfig` файле зависимость на `foo` является излишней или ошибочной.

Отметим еще одну особенность реализации: если для получения поля `Libs` использовать `pkg-config(1)` с опцией `--libs`, то вывод будет «рекурсивно» дополнен флагами `Libs` компонентов, указанных в `Requires`. Таким образом, напрямую этот вывод использовать нельзя, т. к. это приведет к образованию транзитивных зависимостей. Можно было бы не использовать `pkg-config(1)`, а попытаться извлечь поле `Libs` другим способом; однако эта задача усложняется из-за того, что поле `Libs` может содержать подстановки переменных. Поэтому мы предлагаем еще раз модифицировать `pkg-config`: добавить опцию `--disable-reqcursion`, которая отключает «рекурсивное» дополнение флагов.

8.4 shebang.req

Рассмотрим еще несколько уже более общих методов поиска зависимостей. Метод `shebang.req` является общим для всех *скриптов*: исполняемых файлов, содержащих в первой строке инструкцию вида `#!/usr/bin/foo args`. Для таких файлов порождается зависимость на интерпретатор `/usr/bin/foo`.

Однако в некоторых случаях, когда точный путь к интерпретатору неизвестен, интерпретатор можно вызвать через программу `env(1)`, т. е. с помощью инструкции вида `#!/usr/bin/env foo`. В таких случаях дополнительно выполняется поиск команды `foo` с помощью того же механизма, который используется при формировании зависимостей для команд в шелл-скриптах (см. раздел 5).

Кроме того, `shebang.req` выполняет проверку инструкций указанного вида и позволяет обнаружить некоторые типичные ошибки. Во-первых, ошибочным можно считать использование окончаний строк `\r\n`, т. к. символ `\r` в первой строке не будет удален при запуске. Более того, если инструкция вызова интерпретатора не содержит аргументов, то скрипт даже не сможет запуститься, т. к. файла `/usr/bin/foo\r` не существует. В таких случаях работа `shebang.req` завершается с ошибкой, что также должно привести к ошибочному завершению сборки пакета.

Во-вторых, при вызове интерпретатора с аргументами `#!/usr/bin/foo args` на самом деле не выполняется полного разбиения аргументов `args`, т. е. `args` передается в виде одного аргумента, даже если в нем содержатся пробелы. Некоторые интерпретаторы «знают» о том, при запуске скрипта нужно выполнять дополнительное разбиение аргументов. Так, `perl(1)` прекрасно справляется с инструкцией `#!/usr/bin/perl -w -T`. Однако команда `env(1)` не выполняет дополнительного разбиения аргументов. Поэтому инструкции вида `#!/usr/bin/env perl -w` являются ошибочными: такие скрипты нельзя будет запустить (в данном случае работа скрипта завершится с ошибкой `/usr/bin/env: perl -w: No such file or directory`). В таких случаях работа `shebang.req` также завершается с ошибкой.

8.5 files.req

Этот метод поиска зависимостей позволяет формировать зависимости на каталоги, в которых расположены файлы.

Рассмотрим для примера зависимости perl-модулей. Пакет, который содержит модуль `Foo.pm`, должен предоставлять зависимость `Provides: perl(Foo.pm)`. Однако эта предоставляемая зависимость напрямую связана с файлом `/usr/share/perl5/Foo.pm`, расположенном в иерархии `/usr/share/perl5`. Если интерпретатор не поддерживает иерархию `/usr/share/perl5` (не выполняет поиск модулей в этом каталоге), то он не сможет загрузить этот модуль. В таком случае мы не имеем права предоставлять «логическую» зависимость `perl(Foo.pm)`, не обеспечив требования «физической» согласованности с интерпретатором.

Требования согласованности виртуальных зависимостей с поддерживаемыми иерархиями каталогов можно довольно удачно выразить с помощью файловых зависимостей на каталоги: в рассмотренном примере пакет должен быть дополнен зависимостью `Requires: /usr/share/perl5`. Для разрешения такой зависимости достаточно, чтобы каталог `/usr/share/perl5` «принадлежал» интерпретатору, т. е. чтобы этот каталог был запакован в базовом пакете `perl-base`. Если же интерпретатор не поддерживает каталог `/usr/share/perl5`, то для установки модуля потребуется обновление интерпретатора.

Однако такая схема зависимостей является довольно хрупкой: если какой-либо другой пакет по ошибке «завладеет» каталогом `/usr/share/perl5`, то проверка согласованности через зависимость на каталог потеряет смысл. Поэтому нужно дополнительно контролировать эксклюзивную принадлежность каталогов, образующих иерархии, к своим базовым пакетам.

Метод `files.req` позволяет решить обе задачи: формировать зависимости на каталоги и контролировать эксклюзивную принадлежность каталогов. Для этого используются конфигурационные файлы `/usr/lib/rpm/*-files.req.list`, содержащие пары ⟨*каталог*, *базовый-пакет*⟩. Так, файл `/usr/lib/rpm/perl-files.req.list` содержит строки ⟨`/usr/lib64/perl5`, `perl-base`⟩, ⟨`/usr/share/perl5`, `perl-base`⟩. Если какой-либо файл в пакете располагается в одном из этих каталогов (или их подкаталогах), то `files.req` формирует зависимость на каталог. Кроме того, `files.req` выдает предупреждение, если пакет содержит сам каталог, и при этом пакет называется не `perl-base` (после сборки пакета другая программа выполняет более строгую проверку, которая может завершиться с ошибкой).

8.6 symlinks.req

Этот метод формирует зависимости для символических ссылок. Каждая символическая ссылка в пакете должна указывать на существующий файл (либо на другую символическую ссылку, которая в свою очередь должна указывать на существующий файл и т. д.). Значение символической ссылки считывается, и выполняется один шаг разрешения пути: в качестве зависимости выступает путь, на который указывает ссылка. При этом каноникализация путей выполняется средствами процедуры `find-package` (см. раздел 4). Следует также отметить, что бóльшая часть полученных зависимостей впоследствии будет оптимизирована (см. раздел 2) — будут сохранены только зависимости для ссылок на «внешние» файлы.

8.7 rpm-build-mono

В пакет `rpm-build-mono` реализованы дополнительные методы поиска зависимостей, используемые при сборке пакетов Mono — свободной реализации .Net. Скрипты `mono.req` и `mono.prov` формируют зависимости на `.dll` библиотеки (аналогичные зависимости реализованы в других дистрибутивах). Кроме того, реализован метод `monolib.req`, который формирует зависимости на «нативные» ELF-библиотеки, подключаемые при загрузке `.dll` библиотек. Поскольку `.dll` библиотеки напрямую используют символы из подключаемых «нативных» библиотек, то существует также возможность ввести для этих зависимостей set-версии (однако реализация усложняется из-за того, что для `.dll` библиотек нельзя использовать `ld.so`).

Особенностью `rpm-build-mono` является то, что он используется только для сборки специфической группы пакетов. Кроме того, в реализации `rpm-build-mono` используется программа `monodis(1)` из базового комплекта Mono. Поэтому пакет `rpm-build-mono` нельзя внести в базовую сборочную среду. В то же время неправильно было бы требовать, чтобы в каждом Mono пакете была добавлена зависимость `BuildRequires: rpm-build-mono`. Нужна общая схема, при которой сборка всех Mono пакетов выполняется с поддержкой Mono зависимостей. Чтобы реализовать такую схему, мы предлагаем добавить зависимость на `rpm-build-mono` в пакет с компилятором `mcs(1)`. Тогда любой пакет, при сборке которого используется компилятор `mcs(1)`, автоматически получит поддержку Mono зависимостей. В то же время пакет `rpm-build-mono` не потребуется для использования собранных компонентов и для запуска программ, т. к. компилятор `mcs(1)` обычно не входит в базовую Mono среду «времени выполнения», а находится в отдельном пакете.

8.8 buildreq

Данный раздел является отступлением. До сих пор мы рассматривали автоматическое формирование зависимостей `Requires` и `Provides` на основе содержимого файлов, входящих пакет. В следующем разделе будет также рассмотрено формирование зависимостей типа `Requires(post)` для %post-скриптов, выполняемых при установке пакета. В этом разделе рассматривается автоматическое формирование `BuildRequires` зависимостей. Автоматическое формирование `BuildRequires` зависимостей представляет интерес, потому что в таком случае *все* виды зависимостей могут формироваться автоматически. Указание зависимостей вручную часто приводит к ошибкам или неточностям. Кроме того, вручную указанные зависимости могут устареть или потерять актуальность, так что каждый раз при подготовке новой версии пакета их нужно проверять заново. Если же продумать схему, при которой все виды зависимостей могут быть сформированы автоматически, и при этом получен приемлемый результат, то это позволяет решить целый класс проблем.

Программа `buildreq` использует `strace(1)` для трассировки доступа к файлам в процессе сборки пакета. На первом этапе составляется список файлов, использованных при сборке пакета. В качестве доступа к файлам обычно нужно учитывать только системные вызовы `open(2)` и `execve(2)`, т. к. системные вызовы типа `stat(2)` не используют содержимого файлов. Кроме того, если аргументом `open(2)` или `execve(2)` является символическая ссылка, то к списку файлов, используемых при сборке, добавляются также файл, на который указывает ссылка.

На следующем этапе по списку файлов, использованных при сборке, `buildreq` формируется список пакетов, к которым относится эти файлы. Однако предварительно должна быть выполнена фильтрация списка. Дело в том, что существуют случаи, при которых использование файлов при сборке происходит по «глобальному шаблону». Например, при работе программы `autoconf(1)` используются все имеющиеся файлы `/usr/share/aclocal/*.m4`. Это происходит не потому, что все эти эти файлы действительно необходимы для сборки, а просто потому, что `autoconf(1)` подключает все имеющиеся макроопределения. Поэтому при формировании списка требуемых пакетов файлы такого рода учитывать не следует.

Полученный список пакетов является очень большим и может включать в себя сотни пакетов (в дистрибутиве SuSE для такого списка в спекфайле было введено поле `usedforbuild`). Недостатком такого большого списка является избыточность и малая информативность. Так, если при сборке были использованы пакеты `libfoo` и `libfoo-devel`, то в списке следовало бы оставить только `libfoo-devel` (но при этом желательно убедиться, что он требует `libfoo`). Однако оптимизацию, основанную на именах пакетов, нельзя считать серьезным подходом. Нужна общая процедура оптимизации списка пакетов, которая выделяет минимальное подмножество, которое «вытягивает» по зависимостям все остальные пакеты.

Процедура оптимизации работает следующим образом:

- По списку пакетов на входе выстраиваются пары пакетов с непосредственными зависимостями `A->B` — пакет A требует пакет B, а также пары с виртуальными зависимостями `A->v->B`, где `v` — виртуальный пакет, предоставляемый пакетом B и требуемый пакетом A.

- Полученное множество пар зада частичный порядок. С помощью программы топологической сортировки `tsort(1)` формируется линейный список, в котором пакеты упорядочены по взаимным зависимостям между ними.

- Последний проход алгоритма реализует идею т. н. «решета Эратосфена». Пакеты, расположенные в начале упорядоченного списка, содержат зависимости на другие пакеты, которые расположены в списке ближе к концу. Оптимизация состоит в «вычеркивании» всех «зависимых» пакетов.

После оптимизации `buildreq` добавляет список пакетов в спекфайл в качестве зависимостей `BuildRequires`.

К недостаткам описанного подхода относится уже отмеченный недостаток, характерный для всех методов трассировки: непосредственные зависимости оказываются смешаны с транзитивными зависимостями. Оптимизация в таком случае может приводить к тому, что вместе с транзитивными зависимостями удаляются и некоторые прямые зависимости. В некоторых случаях это оправдано. Например, если для сборки требуются пакеты `gtk2-devel` и `glib2-devel`, то в `BuildRequires` достаточно будет оставить `gtk2-devel`, т. к. на уровне заголовочных файлов интерфейс `gtk2` напрямую связан с интерфейсом `glib2`. В некоторых других случаях, однако, оптимизация выглядит сомнительной. Так, если для сборки требуются пакеты `cairo-devel` и `libpng-devel`, то оптимизировать `libpng-devel` не следовало бы, т. к. библиотека `libpng` не используется в интерфейсе `cairo`. Однако же в файле `cairo-png.pc` указана зависимость на `libpng` (этот файл генерируется автоматически).

9 Зависимости %post-скриптов

В результате усовершенствования методов поиска зависимостей оказалось, что автоматически сформированные зависимости `Requires` и `Provides` практически всегда дают приемлемый результат, так что вручную писать зависимости в спекфайле стало не только бесполезно, но и нежелательно (т. к. при сборке новых версий такие зависимости требуют проверки). Автоматическое добавление в спекфайл `BuildRequires` зависимостей тоже было реализовано довольно давно. В какой-то момент единственным видом зависимостей, которые нужно было писать в спекфайле вручную, остались зависимости %post-скриптов в спекфайле. Несколько точнее, к %post-скриптам мы относим скрипты, находящиеся в секциях спекфайла `%pre`, `%preun`, `%post` и `%postun`. Для обеспечения работоспособности этих скриптов должны быть сформированы зависимости соответственно следующих типов: `Requires(pre)`, `Requires(preun)`, `Requires(post)` и `Requires(postun)`.

Поскольку %post-скрипты выполняются во время установки и удаления пакетов, то ошибки в их работе могут привести к нарушению целостности базы `rpmdb`: если скрипт завершился с ошибкой, то `rpm` не может считать пакет полностью установленным или обновленным (или, соответственно, удаленным). Обычно после таких ошибок требуется ручная работа по восстановлению целостности базы пакетов. Поэтому обеспечение работоспособности %post-скриптов можно отнести к первоочередным задачам управления пакетами. Поиск зависимостей позволяет не только обеспечить наличие нужных программ, используемых в %post-скриптах, но также выполняет синтаксическую проверку скрипта.

Идея автоматического поиска зависимостей в %post-скриптах довольно проста: нужно сохранить скрипт во временном файле и далее использовать общий механизм поиска зависимостей, как если бы скрипт являлся файлом в пакете. Однако полученные зависимости нельзя приравнять к обычным зависимостям `Requires`. Поэтому реализация требует внесения модификаций в библиотеку `librpmbuild`: наряду со стадиями поиска зависимостей `find-requires` и `find-provides` должна быть предусмотрена стадия `find-scriptlet-requires`.

У зависимостей %post-скриптов имеется и некоторая специфика. Что происходит, если в %post-скрипте пакета запускается программа, расположенная в самом пакете? Хотя rpm и выполняет топологическую сортировку пакетов, он не дает строгой гарантии, что на момент запуска программы все `Requires` зависимости пакета удовлетворены. Поэтому желательно продублировать часть `Requires` зависимостей, связанных с работоспособностью программы, в зависимости `Requires(post)`. Однако реализация оказывается довольно сложной: если для работы программы требуются другие файлы в этом же пакете, то для них в свою очередь снова нужен перенос зависимостей, и т. д. С другой стороны, если программа запускается в скрипте `%pre`, тогда вообще не следует считать, что запускается программа, расположенная в самом пакете (возможно, запускается программа из старой версии пакета, и зависимость является условной).

На следующем этапе развития репозитория были реализованы файлтриггеры (см. раздел 13.3), и потребность в %post-скриптах частично отпала. Если %post-скрипт дублируется в пакетах, а его действие является глобальным и может быть отложено, то вместо «пакетного» действия лучше реализовать «системное» действие.

10 Фильтрация зависимостей, weak provides

Автоматический поиск зависимостей наилучшим образом работает для файлов с простой и достаточно «жесткой» структурой. К таким файлам относятся, например, ELF-программы и библиотеки. Им можно противопоставить интерпретируемый код: поскольку для интерпретируемого кода существует проблема условных зависимостей, то неправильные или «лишние» зависимости время от времени все же будут возникать. Конечно, мы уделяем большое внимание корректному разрешению условных зависимостей, и допускаем возникновение неправильных зависимостей только в качестве исключения. Тем не менее, нужно иметь готовый механизм «фильтрации» или, вообще, коррекции автоматических зависимостей.

Как уже отмечено в разделе 5, для пакетов с шелл-скриптами категории `contrib` иногда имеет смысл вообще отключить поиск шелл-зависимостей: `AutoReq: yes, noshell`. Аналогичная конструкция может быть использована и для других методов. В данном разделе рассматривается более избирательная фильтрация зависимостей.

В качестве примера рассмотрим пакет `perl-CGI`. В этом пакете содержится основной модуль `CGI.pm`, а также несколько дополнительных модулей, в том числе модуль `CGI/Fast.pm`, который является подклассом `CGI.pm` и упрощает интеграцию с внешним модулем `FCGI.pm`, вследствие чего образуется зависимость на `perl-FCGI`. Эту зависимость следует считать нежелательной, т. к. `perl-CGI` относится к базовым пакетам и в остальном не требует специфических модулей. Возможное решение — выделить модуль `CGI/Fast.pm` в отдельный подпакет `perl-CGI-Fast`. Однако на самом деле модуль `CGI/Fast.pm` не используется ни в одном пакете в репозитории (по крайней мере, по умолчанию), т. к. использование FastCGI требует предварительной настройки. В таком случае возникает вопрос: стоит ли выделять отдельный подпакет ради одного неиспользуемого файла? Возможно другое решение: признать модуль `CGI/Fast.pm` несущественной часть пакета и целиком игнорировать его зависимости. Для этого нужен специальный макрос, который позволяет отключить поиск зависимостей на уровне файлов: `%add_findreq_skiplist */CGI/Fast.pm`.

Аналогичный макрос может пригодиться для Provides зависимостей. Например, в пакете `perl-DateTime-Locale` информация о «локалях» располагается в отдельных модулях типа `DateTime/Locale/ru_UA.pm`, которые используются только в реализации основного модуля `DateTime/Locale.pm`. Чтобы не создавать большого числа лишних Provides зависимостей, нужно исключить эти модули из процедуры `find-provides`: `%add_findprov_skiplist */DateTime/Locale/[a-z]*.pm`.

Фильтрация на уровне файлов, однако, не всегда позволяет решить проблему условных зависимостей: в некоторых случаях требуется фильтровать сами зависимости. Вернемся к модулю `CGI.pm`. Поддержка `mod_perl` влияет на внутреннюю логику работы модуля, поэтому выбор режима работы и подключение `mod_perl` происходит на стадии инициализации модуля:

```
if ($ENV{MOD_PERL}) {
    if ($ENV{MOD_PERL_API_VERSION} == 2) {
        $MOD_PERL = 2;
        require Apache2::Response;
        require APR::Pool;
    } else {
```

```
        $MOD_PERL = 1;
        require Apache;
    }
}
```

Понятно, что зависимости на модули `Apache`, `Apache2` и `APR` являются условными: эти модули являются частью `mod_perl` и должны быть доступны по факту работы модуля в режиме `mod_perl`. Поскольку приведенный код относится к коду инициализации модуля, то в режиме `modexec` условные зависимости будут игнорироваться автоматически (см. раздел 6.1). Однако режим `modexec` пока не реализован, поэтому приходится фильтровать зависимости с помощью макросов:

```
%filter_from_requires /^perl.APR/d
%filter_from_requires /^perl.Apache/d
```

В качестве параметров этот макрос использует `sed`-выражения, которые построчно применяются к зависимостям, полученным на стадии `find-requires`.

С помощью `sed`-выражений можно не только удалять ненужные зависимости, но и выполнять довольно сложную коррекцию зависимостей. В качестве примера рассмотрим библиотеку `libexpat`. С выходом версии 2.0 в 2006 г. библиотека изменила имя с `libexpat.so.0` на `libexpat.so.1`, при этом изменения в интерфейсе были незначительными. Для сохранения совместимости в пакет была добавлена символическая ссылка `libexpat.so.0`. Однако для этой символической не генерировалсь Provides зависимость для старого имени библиотеки, так что пришлось вручную написать в спекфайле `Provides: libexpat.so.0`. С введением set-версий этого оказалось бы недостаточно: фактически нам нужно взять зависимость с set-версией и на ее основе создать еще одну зависимость, скорректировав имя зависимости. Теперь этого можно добиться с помощью следующего макроса в спекфайле:

```
%filter_from_provides /^libexpat\.so\.1/{p;s/1/0/}
```

Как уже отмечено, некоторые Provides зависимости предоставлять нежелательно. Кроме оптимизации, иногда это помогает предотвратить недопустимые Requires зависимости. Например, при сборке `glibc` из Provides исключаются интерфейсы `GLIBC_PRIVATE`, т. к. они не должны использоваться во внешнем коде. Однако эти интерфейсы используются внутри `glibc`, в том числе между подпакетами. Тогда `GLIBC_PRIVATE` одновременно нужно удалять и в Requires. Возможен более тонкий подход: некоторые Provides зависимости можно «ослабить»: они будут учитываться как обычные Provides зависимости, в том числе при оптимизации зависимостей между подпакетами, но в самом конце будут удалены из пакета. Макрос для `gllibc` будет таким:

```
%weaken_provides lib*.so*(GLIBC_PRIVATE)*
```

В общем случае фильтрацию зависимостей следовало бы рассматривать по трем параметрам: $\langle$*файл*, *метод*, *зависимость*$\rangle$. Однако фильтрация используется относительно редко, и на практике хватает возможности фильтрации на уровне отдельных файлов и на уровне отдельных зависимостей.

11 Пакеты с отладочной информацией

Когда-то давно существовало два режима сборки пакетов: обычный режим и режим сборки с отладочной информацией. В обычном режиме на стадии `brp` выполнялось «обрезание» ELF-программ и библиотек с помощью программы `strip(1)`; в отладочном режиме «обрезание» было отключено, а при компиляции генерировалась дополнительная отладочная информация (с ключом компилятора `-g`). Таким образом, чтобы заняться отладкой пакета, нужно было пересобрать его в отладочном режиме и переустановить в системе. После переустановки часть ошибок воспроизвести уже не удавалось.

Затем программы типа `gdb(1)` и `valgrind(1)` научились загружать отладочную информацию из отдельных файлов: при отладке программы `/bin/foo` отладочная информация должна располагаться в файле `/usr/lib/debug/bin/foo.debug`. Так появилась идея создания пакетов с отладочной информацией: компиляция должна всегда выполняться с ключом `-g`, а на стадии `brp` вместо «обрезания» отладочная информация перемещается в `.debug` файлы в каталоге `/usr/lib/debug`. Кроме того, отладочная информация содержит ссылки на исходный код. Утилита `debugedit` позволяет модифицировать в `.debug` файлах ссылки на исходный код: заменить каталог типа `~/RPM/BUILD`, используемый для сборки пакетов, на системный каталог `/usr/src/debug`. Таким образом, файлы с отладочной информацией и с исходным кодом можно поместить в отдельные пакеты, доступные для установки по мере появления необходимости.

В традиционной реализации `deubginfo` пакетов можно выделить две стадии:

- Создание содержимого для `debuginfo` пакета: создание файлов с отладочной информацией в каталоге `/usr/lib/debug`; создание символических ссылок на `.debug` файлы, имитирующих ссылки в основном дереве пакета, а также дополнительных символических ссылок в каталоге `/usr/lib/debug/.build-id`; редактирование ссылок на исходный код и копирование исходных файлов в каталог `/usr/src/debug`.

- Создание отдельного подпакета на уровне rpm: в спекфайл автоматически включается шаблон создания подпакета `%name-debuginfo`, где `%name` — имя `src.rpm` пакета; список файлов для пакета передается через файл, созданный на первой стадии.

Традиционной реализации присущи два крупных недостатка:

- Создается только один пакет с отладочной информацией — `%name-debuginfo`, при этом никак не учитывается разделение основного пакета на подпакеты. Например, для компилятора `gcc` будет создан пакет `gcc-debuginfo`, в котором будет содержаться как отладочная информация для базовой библиотеки `libgcc`, реализующей некоторые «встроенные» функции, так и для программ компиляции `cc1`, `cc1plus`, но больше всего в пакете занимает отладочная информация для языка Java.

- Отсутствие зависимостей между пакетами с отладочной информацией. Часто при работе программы ошибки проявляются в библиотеках, и поэтому для

полноценной отладки требуется отладочная информация для всех стековых фреймов вызова функций — пакета с отладочной информацией для одной только программы оказывается недостаточно. В дистрибутиве Fedora имеется специальный скрипт `debuginfo-install`, который облегчает поиск и установку отладочной информации для требуемых библиотек. Тем не менее, замкнутость и вообще доступность отладочной информации в репозитории не контролируется на уровне зависимостей.

В одном российском дистрибутиве мы реализовали альтернативный подход. Изменения в наибольшей степени коснулись второй стадии — создания пакетов, тогда как подготовка отладочной информации (процедура `brp-debuginfo`) выполняется сходным образом. При этом можно выделить два главных отличия:

- Для каждого подпакета `foo` создается отдельный пакет с отладочной информацией `foo-debuginfo` (если список файлов в для пакета `foo-debuginfo` оказывается непустым). Таким образом, пакеты с отладочной информацией естественным образом дополняют основное дерево пакетов.

- Между `.debug` файлами для программ и библиотек автоматически формируются зависимости вида `debug(libfoo.so.0)`, которые отражают связи между самими программами и библиотеками. Таким образом, наличие отладочной информации для программы означает также доступность отладочной информации для всех используемых библиотек.

Логика создания `debuginfo` пакетов довольно прямолинейна: для каждого файла `f` из пакета `foo` в пакет `foo-debuginfo` должен быть включен файл `/usr/lib/debug ${f}.debug`, если такой файл существует в %buildroot (т. е. если такой файл был создан на стадии brp). Кроме того, если в этом файле содержатся ссылки на исходный код, то соответствующие файлы в каталоге `/usr/src/debug` также должны быть включены в пакет `foo-debuginfo`. Однако существует одно отступление от общего правила: в нашей реализации выполняется *перегруппировка символических ссылок*. Как уже сказано, в каталоге `/usr/lib/debug` создаются символические ссылки, имитирующие основное дерево программ и библиотек: например, если в основном дереве ссылка `/usr/lib64/libfoo.so` указывает на файл библиотеки `/usr/lib64/libfoo.so.0`, то для пакета с отладочной информацией будет создана ссылка `/usr/lib/debug/usr/lib64/libfoo.so.debug`, указывающая на файл с отладочной информацией библиотеки `/usr/lib/debug/usr/lib64/libfoo.so.0.debug`. Вместе с тем, ссылки типа `libfoo.so` обычно предназначены для компоновки и находятся в пакетах типа `libfoo-devel`. Если строго следовать правилу соответствия между основными пакетами и пакетам с отладочной информацией, то нужно было бы создать пакет `libfoo-devel-debuginfo`, все содержимое которого представляет собой ссылку `libfoo.so.debug`. Поэтому, чтобы не создавать «фиктивных» пакетов, содержащих одну-единственную ссылку, выполняется перегруппировка ссылок: `.debug` ссылки приписываются к тому пакету, в котором находится файл, на который они указывают. Таким образом, ссылка `libfoo.so.debug` помещается в пакет `libfoo-debuginfo`, а пакет `libfoo-devel-debuginfo` создан не будет.

Однако у этого подхода есть и недостаток: перегруппировка ссылок может привести к файловым конфликтам, которые удается избегать в основных пакетах.

Это может быть связано с наличием нескольких версий библиотеки: пусть, например, имеются пакеты `libfoo1` и `libfoo2`, которые могут быть установлены одновременно, а также пакеты `libfoo1-devel` и `libfoo2-devel`, которые конфликтуют из-за символической ссылки `/usr/lib64/libfoo.so`, так что для сборки может быть установлен только один из них. Перегруппировка ссылок в приводит к тому, что пакеты `libfoo1-debuginfo` и `libfoo2-debuginfo` будут конфликтовать из-за ссылки `/usr/lib/debug/usr/lib64/libfoo.so.debug`. Таким образом, существуют допустимые конфигурации, в которых отладочная информация оказывается недоступна. Тем не менее, мы не считаем это серьезным недостатком. Во-первых, такая же проблема характерна и для традиционной реализации (при которой создание всего одного пакета можно считать глобальной перегруппировкой). Во-вторых, в нормальном состоянии репозитория конфликтующие конфигурации должны быть строго ограничены: две версии библиотеки не должны использоваться в одном и том же приложении. На самом деле перегруппировка ссылок позволяет обнаруживать именно такие конфигурации. Как уже было сказано во введении, для поддержки целостности репозитория сборочная система должна контролировать инвариант перехода, причем одним из условий является возможность установки любого пакета в базовую среду (basesystem). Эту проверку можно распространить на пакеты с отладочной информацией, и тогда получаем следующее: если пакет с отладочной информацией для приложения не удается установить (вследствие файлового конфликта), то это свидетельствует о том, что в приложении используется несколько версий библиотеки.

Рассмотрим еще несколько особенностей реализации. Создание нескольких подпакетов с отладочной информацией приводит к тому, что файлы с исходным кодом могут дублироваться (обычно заголовочные файлы используются во всех подпакетах). Возникает вопрос: нельзя ли избавиться от дублирования исходных файлов? Например, если подпакет с программой `foo-debuginfo` требует подпакет с библиотекой `libfoo-debuginfo` через зависимость `debug(libfoo.so.0)`, то из пакета `foo-debuginfo` можно было бы удалить пересекающиеся исходники. Однако реализация такого подхода требует очень глубокой и сложной модификации библиотеки `librpmbuild`. К счастью, реализацию можно значительно упростить, если отталкиваться от идеи строгих зависимостей между подпакетами (см. раздел 2): если пакет `foo-debuginfo` содержит строгую зависимость на `libfoo-debuginfo`, то из него можно удалить дублирующиеся исходники. Но для этого сначала надо наделить пакеты с отладочной информацией строгими зависимостями. Для наделения пакета `foo-debuginfo` строгой зависимостью на пакет `libfoo-debuginfo` должно быть выполнено два условия:

- Пакет `foo-debuginfo` должен содержать какую-либо зависимость, которая разрешается в пакет `libfoo-debuginfo`; подразумевается `debug(libfoo.so.0)`.
- Строгая зависимость должна существовать и между базовыми подпакетами: пакет `foo` должен содержать строгую зависимость на пакет `libfoo`.

Наделение пакетов с отладочной информацией строгими зависимостями выполняется на ранней стадии, и в дальнейшем они подпадают под общие правила оптимизации пакетов со строгими зависимостями. В частности, в приведенном примере «нестрогая» зависимость `debug(libfoo.so.0)` будет удалена.

12 Проверки во время сборки пакетов

Как уже сказано во введении, при сборке пакета выполняется ряд проверок, причем проверки рассосредоточены на нескольких уровнях: часть проверок выполняется прямо во время сборки пакета (на стадии brp); некоторые проверки вынесены в отдельную программу; еще часть проверок со сложными условиями (синхронная сборка для нескольких архитектур и т. п.) выполняется сборочной системой. Поэтому в данном разделе не ставится цель дать исчерпывающее описание проверок, выполняемых во время сборки — вместо этого следовало бы обсудить общую архитектуру системы проверки пакетов и распределение проверок по уровням с точки зрения сборочной системы. В то же время проверки, выполняемые на стадии brp, имеют свою специфику. Поэтому в данном разделе мы подробно рассмотрим только одну характерную проверку — проверку ELF-программ и библиотек `brp-verify_elf`.

Процедура `brp-verify_elf` выполняется для всех программ и библиотек, найденных в каталоге `%buildroot`. Несколько точнее, процедура `brp-verify_elf` выполняется после процедуры `brp-debuginfo` (см. раздел 11), и каталог `/usr/lib/debug` исключается из поиска. Процедура контролирует несколько разных аспектов соответствия. Рассмотрим их подробнее.

ARCH В noarch пакетах не должно содержаться ELF файлов. Однако существуют исключения: в каталоге `/lib/firmware` располагаются прошивки для устройств, причем некоторые прошивки используют формат ELF. Т. к. прошивки не зависят от архитектуры (i686 или x86-64), но они могут находится в noarch пакетах. Данная проверка не учитывает noarch подпакетов, т. к. на стадии brp доступно только «глобальное» значение `BuildArch`. Сборочная система впоследствии выполняет строгую верификацию noarch пакетов (см. введение), но для этого требуется синхронная сборка для двух или более архитектур.

FHS В соответствии с *Filesystem Hierarchy Standard*, ELF файлы не должны располагаться в каталогах `/usr/share`, `/etc` и подкаталогах.

LINT Разнообразные проверки, выполняемые программой `elflint(1)`.

RPATH Проверка путей поиска библиотек `DT_RPATH`. Недопустимыми являются пути, которые содержат текущий каталог «`.`», в том числе в неявной форме — `:/usr/foo` (первый путь является пустым и трактуется как текущий каталог). Кроме того, к недопустимым относится каталог сборки пакета `~/RPM/BUILD`, а также некоторые пути типа `/lib/../lib64`, обусловленные погрешностями в сборочных скриптах. В некоторых дистрибутивах взят курс на полный или почти полный отказ от `RPATH`, см. `http://fedoraproject.org/wiki/RPath_Packaging_Draft`

STACK Программы и библиотеки не должны выполнять кода на стеке (это позволяет выполнять код с дополнительной защитой — с отключенной возможностью исполнения на стеке). Если файл не требует исполняемого стека, то он должен содержать заголовок `GNU_STACK` со значением `off` (проверка проходит успешно). Как правило, компилятор добавляет такой заголовок автоматически

(т. к. компилятор не генерирует код, который требовал бы исполнения на стеке). Однако заголовок не может быть сформирован автоматически для файлов с исходным кодом на ассемблере. В таком случае в файл с ассемблерным кодом секция `.note.GNU-stack` должна быть добавлена явно.

TEXTREL В библиотеках не должно содержаться text relocations — участков кода с абсолютными адресами, т. к. библиотеки обычно не могут быть загружены по номинальному адресу, и загрузчику `ld.so` придется редактировать текстовый сегмент библиотеки. Text relocations обычно возникают, если какой-либо объектный файл, входящий в библиотеку, скомпилирован без опции `-fPIC` (position-independent code). Однако в некоторых случаях источником text relocations являются ассемблерные вставки, написанные без учета требований для перемещаемого кода.

UNRESOLVED Программы и библиотеки в глобальном пространстве имен не должны содержать неразрешимых символов (см. далее).

Проверки могут выполняться строго или нестрого, т. е. либо вызывать ошибку сборки, либо ограничиваться предупреждением. Несколько точнее, для каждого аспекта проверки существует три режима строгости: `noraml` (по умолчанию), `strict` и `relaxed`. В режиме `strict` все нарушения вызывают ошибку, а в режиме `relaxed` — только предупреждения. В режиме `normal` некоторые проверки вызывают ошибку, а некоторые ограничиваются предупреждениями (при менее тяжких нарушениях). Режим проверки можно контролировать специальным макросом в спекфайле. Например, если text relocations появляются вследствие ассемблерных вставок, которые не поддаются легкому исправлению, то проверку `TEXTREL` можно ослабить с помощью конструкции `%set_verify_elf_method textrel=relaxed`. Если же пакет содержит закрытые драйверы, распространяемые в бинарном виде, то можно ослабить сразу все проверки: `%set_verify_elf_method relaxed`.

Вернемся к вопросу разрешимости символов. Зависимости с set-версиями (см. раздел 7.2) контролируют только наличие символов, которые удалось разрешить на стадии поиска зависимостей. Проверка `UNRESOLVED` в свою очередь запрещает неразрешимые символы в программах и «публичных» библиотеках. Однако существуют еще один класс разделяемых библиотек — это плагины, загружаемые через `dlopen(3)`. Для плагинов неразрешимые символы допустимы, т. к. эти символы могут разрешаться, например, в саму программу, которая загружает плагин.

Таким образом, существует класс символов, который не учитывается в зависимостях и не имеет строгой разрешимости. Чтобы покрыть этот класс, сборочная система выполняет *глобальную проверку разрешимости символов*:[11] составляется глобальный список неопределенных символов и глобальный список определенных символов (для всех пакетов в репозитории). Для этого ко всем программам и библиотекам применяется программа `nm(1)`, которая выводит символы и их типы. Символы типа `U` относятся к неопределенным, а функции `T`, переменные `D`, `B` и еще некоторые типы символов относятся к определенным. Стоит также отметить, что вывод `nm(1)` можно кешировать, так что распаковка потребуется всего один раз, и при

[11] См. *Анализ бинарной совместимости репозитория rpm-пакетов* в сборнике `http://www.altlinux.ru/media/3rdconference_theses.pdf`

повторных запусках проверка будет выполняться довольно быстро — менее минуты. Неопределенные символы, которые не удалось «разрешить» по глобальному списку определенных символов (процедура `bad_elf_symbols`), считаются недопустимыми.

Понятно, что такая «глобальная» проверка не является достаточной, потому что в ней не учитываются связи между файлами. Дело, однако, именно в том, что неразрешимые символы в плагинах возникают как раз из-за отсутствия связей. Поясним сказанное на примере (это довольно сложный пример из реальной жизни). Сервер баз данных PostgreSQL позволяет реализовать расширения для сервера — такие расширения используют как библиотечные функции, так и внутренние функции самого сервера. Например, расширение `/usr/lib/pgsql/plproxy.so` из пакета `plproxy` использует функции `DirectFunctionCall1` и `FunctionCall2`, предоставляемые основной программой `/usr/bin/postgres`, находящейся в пакете `postgresql9.0-server`. В новой версии `postgresql9.1-server` эти функции уже не предоставляются, т. к. этот внутренний интерфейс подвергся переработке. В то же время на уровне обычных зависимостей пакет `plproxy` считается совместимым с новой версией сервера, т. к. библиотечные интерфейсы сохранили совместимость. Проверка `bad_elf_symbols` также не срабатывает, если в репозитории находятся сразу оба пакета `postgresql9.0-server` и `postgresql9.1-server`. Несовместимость обнаруживается только при последующей попытке удалить старую версию `postgresql9.0-server` — тогда в файле `plproxy.so` обнаруживаются недопустимые символы `DirectFunctionCall1` и `FunctionCall2`.

Приведенный пример показывает, что глобальная проверка может не сработать, если в репозитории имеется несколько версий программы. Тем не менее, данная проверка хорошо справляется с такими ошибками, как опечатки в названиях функций, а также нераскрытые макросы препроцессора (см. `CONVERT_LIST` ниже).

Вот некоторые недопустимые символы, обнаруженные у пакетов в каталоге `mandriva/devel/cooker/x86_64/media/main/release`:

```
drakxtools-backend-13.51-15-mdv2012.0.x86_64.rpm
    /usr/lib/libDrakX/auto/c/stuff/stuff.so
        get_pci_description
lib64gnome-vfs2_0-2.24.4-4-mdv2012.0.x86_64.rpm
    /usr/lib64/gnome-vfs-2.0/modules/libsftp.so
        g_thread_supported
    /usr/lib64/gnome-vfs-2.0/modules/libsmb.so
        smbc_remove_unused_server
libgda4.0-4.2.11-1-mdv2012.0.x86_64.rpm
    /usr/lib64/libgda-4.0/providers/libgda-sqlcipher.so
        _gda_server_operation_new_from_string
        _split_identifier_string
proftpd-mod_time-1.3.3e-1-mdv2011.0.x86_64.rpm
    /usr/lib64/proftpd/mod_time.so
        log_pri
tomoe-gtk-python-0.6.0-12-mdv2011.0.x86_64.rpm
    /usr/lib64/python2.7/site-packages/gtk-2.0/tomoegtk.so
        CONVERT_LIST
        tomoe_char_table_layout_get_type
```

13 Разное

13.1 RPM4 vs RPM5

На наш взгляд, вариант rpm4 (rpm.org), разрабатываемый в Red Hat, является во многих отношениях более удачным и, в частности, лучше подходит для реализации наших предложений, чем вариант rpm5, разрабатываемый Джонсоном.

Большой вклад Джонсона в развитие rpm невозможно отрицать, но, тем не менее, стоит начать с того, что Джонсон не является автором оригинального rpm. В законченном виде rpm 2.x был разработан Троуном и Юингом в 1995-97 годах; вскоре авторы разбогатели и отошли от разработки, передав свой проект Джонсону. Активная работа Джонсона началась в июле 1998 года, а начало самостоятельной разработки rpm 4.0 относится к апрелю 1999 года.

Стиль программирования Джонсона на языке Си не представляется нам особенно удачным. Например, Джонсон ввел ряд двухбуквенных сокращений, которые не кажутся нам удобными или очевидными: ts означает «transaction», ds означает «dependency set», fc означает «file coloring», al означает «available list» и т. п. Помимо собственно сокращений, эти нововведения часто привносят и неудачные абстракции: в них смешиваются сущности, контейнеры и операции. Например, интерфейс rpmal означает «имеющийся список» пакетов, т. е. он реализует список с операциями «добавить», «удалить» и т. д.; однако настоящим назначением интерфейса rpmal является операция разрешения зависимостей, т. к. в этом списке поддерживается индекс Provides зависимостей. Но напрямую доступ к индексу не предоставляется.

Вообще, в коде на Си Джонсон слишком усердно использует «сокрытие данных» (opaque data structures), т. е. доступ к данным только через функции. Но это имеет смысл делать только для публичной части API, а на внутреннем уровне сокрытие данных часто приводит к усложнению кода. Например, если какая-либо структура данных `s` содержит массив элементов, то наиболее естественно было бы предоставить доступ к массиву напрямую через `s->n` и `s->v[]` (в стиле `argc` и `argv`). Однако Джонсон считает необходимым использовать для доступа «итератор». Например, доступ к элементам rpmds выполняется через вызов `rpmdsNext(ds)`. При такой реализации «итератор» оказывается связан с самим массивом данных, в результате чего невозможно реализовать вложенный цикл.

Пострадала и ясность кода. Чтобы разобраться в работе rpm, нам часто приходилось изучать исходный код rpm 3.0.

Вопросы кода как реализации, однако, не являются главными. Гораздо хуже то, что Джонсон попросту вносит в код *необдуманные изменения*: Джонсон идет на поводу у своего желания вносить *улучшения*, не подвергая их *критическому осмыслению*. Приведем несколько конкретных примеров необдуманных изменений.

В 2003 году, еще на стадии разработки rpm 4.x, Джонсон реализовал в rpmsq примитивы работы с тредами (через стандартный интерфейс pthread). Предполагалось, в частности, что это позволит параллельно выполнять файловые операции fsm. Однако параллельная распаковка файлов при установке пакетов невозможна по другой причине: распаковка LZMA-архива пакета выполняется довольно медленно и не может быть распараллелена. В результате нововведение оказывается практически бесполезным. В 2010 году в rpm 4.9 примитивы работы с тредами были удалены.

Незадолго до открытия проекта rpm5, в версии 4.4.9 Джонсон реализовал «tagged indices» — оптимизацию при поиске файлов в базе `rpmdb`. Однако эта оптимизация работает неверно: в некоторых случаях нужные файлы будут не найдены. (Эта проблема связана с тем, что файлы в rpm идентифицируются не по их полному пути, а более сложным способом, т. к. rpm до некоторой степени допускает «перемещение» каталогов — замену промежуточного каталога на символическую ссылку). В дальнейшем были внесены и другие изменения, которые могут нарушить логику разрешения зависимостей в rpm. При поиске файлов в базе иногда необходимо найти *все* файлы, а оптимизация с использованием `FILEPATHS` приводит к тому, что могут быть найдены только *некоторые* файлы. (Для удовлетворения файловой зависимостей достаточно найти некоторые файлы, однако проверка конфликтов должна выполняться для всех файлов.)

В августе 2009 года в алгоритм топологической сортировки `rpmtsOrder` было внесено изменение с пометкой fix (исправление), которое в ряде случаев значительно ухудшает порядок установки пакетов. По-видимому, последствия этого изменения Джоносоном осознаны не были. В январе 2010 года Джонсон заимствовал новый алгоритм сортировки пакетов, разработанный в Red Hat.

Почти все попытки Джонсона улучшить стандартную систему зависимостей выглядят сомнительно. В стандартной системе зависимость Requires может быть удовлетворена только с помощью зависимости Provides. Исключением являются зависимости вида `rpmlib(Foo)`, для удовлетворения которых в библиотеке `librpm` должна быть реализована поддержка `Foo`. Джонсон хочет расширить класс специальных случаев за счет введения «пространств имен» rpmns, для которых выполняется runtime probe (проверка времени выполнения). Например, для зависимости вида `Requires: envvar(FOO)`, по мысли Джонсона, должно проверяться наличие переменной окружения `FOO`. Однако эта проверка выполняется во время установки пакета, а во время работы программы переменной `FOO` может и не оказаться.

В конце концов, частое внесение необдуманных изменений и «улучшений» привело к разрастанию кода: по грубой оценке, в rpm4 имеется 2.3 Мбайт кода на языке Си, а в rpm5 — 8.5 Мбайт (по несколько более точной оценке, размер кода rpm5 в 3.2 раза больше). Возникает вопрос: где сосредоточен новый код, и какую новую функцию он выполняет? Оказывается, одним из самых больших файлов в rpm5 является `rpmsql.c` — 156 Кб. В этом файле реализован SQL-движок и «виртуальные таблицы». Одна из виртуальных таблиц предоставляет доступ к файловой системе `/proc`, другая — к базе пользователей `/etc/passwd`. Еще одна таблица предоставляет доступ к переменным окружения, так что с ее помощью выполняется проверка уже упомянутых зависимостей вида `envvar(FOO)`. Трудно удержаться от того, чтобы назвать всю эту конструкцию наслоением глупостей. Однако Джонсон понимает таким образом *реализацию в общем виде* — чтобы объединить и представить в общем виде разнородные понятия, нужно в первую очередь реализовать SQL-движок.

Все это возвращает нас к первоначальному вопросу: что вообще в этом деле можно считать «улучшением» или «достижением»? Для Джонсона любое «улучшение» является ценным само по себе, т. к. это позволяет заявить о проекте rpm5 и противопоставить его rpm4. Разработчики дистрибутивов соблазнятся тем, что Джонсон позволяет им легко вносить изменения. Менеджеры разработчиков тоже довольны, если им удается убедить начальство, что переход с rpm4 на rpm5 улучшает

дистрибутив на 25%. Получается, что правда у каждого своя, и, более того, вроде бы все довольны. Что здесь не так?

Существуют две парадигмы разработки ПО: эволюционная и инженерная. Свободное ПО тяготеет к эволюционной парадигме, с чем связаны некоторые его достоинства и многие недостатки. В соответствии с эволюционной парадигмой, частое внесение изменений является необходимым условием развития, а аккумуляция изменений непременно приводит к совершенствованию ПО. Возможность «развития» ПО трактуется по аналогии с биологической эволюцией, но при этом часто забывают о жестком отборе изменений — ведь большинство мутаций вредны или фатальны. Вообще, биологическая эволюция основывается на рекомбинации генетического материала и на сложных механизмах саморегуляции; если подробно рассматривать процесс мейоза или механизмы гомеостаза, то аналогия между биологической эволюцией и развитием ПО станет совсем поверхностной. Более того, по современным представлениям видообразование не является непрерывным процессом, а происходит «скачкообразно» (punctuated equilibrium). Понимая слабость аналогии, апологеты эволюционной парадигмы дополняют ее социальными аспектами, пытаются придать ей социальную значимость (см. «Собор и Базар» Эрика Рэймонда).

Эволюционной парадигме можно противопоставить инженерную парадигму: требования к программе должны быть определены заранее; программа должна быть спроектирована, реализована и протестирована. При этом главной задачей в проектировании является *декомпозиция*, а к главным принципам реализации следует отнести *минимализм* — высокая сложность конструкции является не преимуществом, а недостатком. Если же программа удовлетворяет всем требованиям, то она не нуждается в улучшениях — во всяком случае, пока новые требования не будут обоснованы, критически осмыслены, внесены коррективы на стадии проектирования и т. п. Конечно, на практике редко следуют формальному процессу, и формальный процесс не интересует нас сам по себе. Если вместо биологической эволюции проводить аналогию со строительством капитальных сооружений, то нас интересуют характеристики полученной конструкции. В связи с этим следует рассмотреть архитектуру rpm — в конструкции этой системы можно выделить 3 части:

- `rpm` — программа установки и обновления пакетов;
- `rpmbuild` — программа сборки пакетов;
- *политика сборки* — программы формирования зависимостей и т. п.

Для программы `rpm` в наибольшей степени характерны функции архиватора; кроме того, программа rpm выполняет проверку зависимостей. Из всех наших предложений на уровне программы `rpm` должны быть реализованы только set-версии (см. раздел 7.2), а также добавлена поддержка файлтриггеров (см. далее раздел 13.3). В остальном программа `rpm` не нуждается в каких-либо усовершенствованиях. Бесконечное развитие и выпуск принципиально новых версий архиватора следует признать навязчивой идеей.

Программа `rpmbuild` занимается формированием пакетов установленного формата; для нее также характерны функции разбора спекфайла и запуска внешних стадий сборки пакета. Из всех наших предложений к этому уровню относятся

только оптимизация зависимостей (см. раздел 2) и формирование `debuginfo` пакетов (см. раздел 11). Кроме того, требуется добавить дополнительную стадию поиска зависимостей в %post-скриптах (см. раздел 9). Таким образом, этот уровень подвергается изменениям в большей степени, однако все предложенные изменения напрямую связаны с функциональностью программы — они напрямую влияют на формирование пакетов. Когда сформированные пакеты будут нас устраивать, изменения в программе `rpmbuild` нужно будет ограничить.

Наконец, большая часть наших предложений относится к *политике сборки* пакетов, причем особое внимание мы уделяем формированию зависимостей, поскольку зависимости выражают условия работоспособности и совместимости ПО. Чем точнее мы сможем выразить эти условия, тем более предсказуемые конфигурации и более надежный репозиторий пакетов мы сможем получить. Но в то же время политика сборки пакетов слабо связана с программой `rpmbuild` — формирование зависимостей происходит на внешних стадиях сборки пакета. Кроме того, политика сборки является специфичной для дистрибутива, и вообще является наиболее «подвижной» и наименее «портабельной» частью конструкции. Желая добиться максимальной переносимости rpm5, понятой однобоко, Джонсон концентрируется на переносимости кода `rpm` и `rpmbuild`, в то время как политика сборки в rpm5 представлена в зачаточном виде. Таким образом, реальные возможности усовершенствования репозитория мало связаны с той разработкой rpm, которой занимается Джонсон.

Для rpm4 характерна более консервативная политика, которая, на наш взгляд, лучше согласуется с инженерным подходом к разработке. Вот что пишет один из разработчиков rpm4 в комментарии к статье *Who maintains RPM? (2011 edition)*[12]

> ... the releases do neither focus on new features nor on transforming RPM into something else. Instead lots of work has been put in solving scalability issues and improving the code base. We believe that RPM — beside some ugly and awkward implementation details — does the right thing and the basic design — created one and a half decades ago — is still valid. We also believe that features added now should be prepared to survive a similar time span — especially when considering the long life time of today's enterprise distributions.
>
> The other reason why the development of RPM does not look that spectacular is that there are several ten thousands of packages out there that need to continue to work — both building and installing. So every change has to be considered thoroughly. Packaging is a complicated topic and the implications are not always easy to foresee. Even simple looking bug fixes have hit packages relying on the broken behavior.

Итак, в чем же правда? Как уже было сказано, правда у каждого своя. Несмотря на то, что наши доводы представляются нам довольно убедительными, и мы готовы отстаивать нашу точку зрения, в нашем взгляде имеются и субъективные стороны. В связи с этим попробуем пояснить субъективные аспекты нашей мотивации.

Главным вопросом для любого дистрибутива, отличного от Red Hat, по большому счету является вопрос о праве не существование. Что интересного может предложить

[12] `http://lwn.net/Articles/441085/`

альтернативный дистрибутив, кроме «нескучных обоев» и т. п.? Безусловно сильной стороной Red Hat является обладание ресурсом, в том числе квалифицированным человеческим ресурсом (3700 сотрудников), а также лояльной базой пользователей и бета-тестеров. Напрямую конкурировать с Red Hat в этой области невозможно. Альтернативный подход может состоять в том, чтобы в условиях ограниченных ресурсов максимально автоматизировать разработку и тестирование дистрибутива. Именно поэтому мы уделяем такое большое внимание формированию зависимостей, ранней диагностике ошибок и автоматическому тестированию.

У Red Hat есть и слабое место — в условиях максимального разделения труда появляется фрагментация ответственности. Так, «выделенные» сотрудники, которые занимаются разработкой rpm, вряд ли будут участвовать в создании сборочной системы. В результате конструкция имеет «зазоры», за которые никто не отвечает. Сократив эти зазоры, можно получить некоторое преимущество. Собственно, данный документ можно считать попыткой сформировать более целостный взгляд на сборку пакетов. Субъективные аспекты нашей точки зрения становятся совершенно очевидны, если учесть, что вполне успешных альтернативных дистрибутивов на данный момент не существует.

13.2 Multiarch

Если управление пакетами, как сказано во введении, можно считать большим достижением, то самой некрасивой страницей в истории этого достижения следует считать multiarch — возможность одновременной установки пакетов разных архитектур, в частности, возможность установки пакетов i686 на x86-64. Возможность установки разнородных пакетов ломает традиционную модель пакетов и зависимостей, и часто приводит к неразберихе. В традиционной модели имя пакета, как правило, идентифицирует установленный пакет в системе (несколько точнее, rpm в принципе допускает *установку* нескольких одноименных пакетов при отсутствии файловых конфликтов, но в дальнейшем возникают сложности с *обновлением* таких пакетов). Кроме того, в традиционной модели пакеты, как правило, не должны содержать пересекающиеся файлы, т. к. это приводит к образованию файловых конфликтов (кроме случаев, когда пересекающиеся файлы идентичны). Поддержка multiarch нарушает оба этих принципа: имя пакета уже нельзя использовать в качестве ссылки на пакет, а пакеты содержат большое количество пересекающихся файлов, нередко конфликтующих. Для разрешения файловых конфликтов используется механизм *file coloring*, плохо документированный, впрочем как и не обладающий никакими другими достоинствами. Все это заставляет обратить на «проблему multiarch» особое внимание.

Можно выделить несколько основных вариантов поддержки multiarch.

- Вариант Fedora — двойной комплект пакетов с одинаковыми именами.
- Вариант Mandriva — имена пакетов с библиотеками и `devel` пакетов в зависимости от архитектуры имеют префикс `lib64` либо `lib`.
- Вариант, реализованный в одном российском дистрибутиве — выполняется перепаковка пакетов.

Таким образом, в варианте Mandriva предпринимается попытка сохранить уникальное наименование пакетов. Однако эта попытка приводит к дальнейшей неразберихе: имена пакетов становятся архитектурно-зависимыми, так что их нельзя использовать в спекфайле в зависимостях `BuildRequires` (т. к. зависимость `BuildRequires: lib64foo-devel` не может быть удовлетворена на архитектуре i686, и наоборот). Чтобы смягчить эту проблему, пакет `lib64foo-devel` может предоставлять зависимость `Provides: libfoo-devel = %version` (тогда в `BuildRequires` можно будет использовать архитектурно-нейтральное имя `libfoo-devel`). Однако эту зависимость требуется добавить в спекфайл вручную, так что нельзя рассчитывать на единообразное пространство имен. Кроме того, зависимость `BuildRequires: libfoo-devel` будет в приоритетном порядке удовлетворена пакетом неправильной архитектуры.

На наш взгляд, главным аспектом проблемы multiarch является неопределенность относительно *объема*, в котором вообще требуется поддержка multiarch. Изначально считалось, что поддержка multiarch должна быть *минимальной*, то есть достаточной лишь для запуска унаследованных приложений. Именно в таком объеме поддержка multiarch предусмотрена в FHS: стандарт вводит раздельные каталоги для библиотек `lib` и `lib64`, однако в остальном поддержка multiarch не затрагивает иерархии файловой системы. Вероятно, наилучшим вариантом минимальной поддержки multiarch является перепаковка пакетов. В этом варианте набор пакетов с системным библиотеками определяется заранее (например, это может быть набор, достаточный для запуска Skype). Конвертация состоит в том, что имя пакета с i686 библиотекой снабжается префиксом i686 (так, пакет `glibc` после перепаковки получит имя `i686-glibc`), при этом из пакета исключаются пересекающиеся файлы (так, из пакета `i686-glibc` будет удален файл `/sbin/ldconfig`). Таким образом, реализуя принцип уникального наименования пакетов и напрямую исключая файловые конфликты, конвертация следует традиционной модели.

Вернемся к варианту Mandriva. Несмотря на попытку сохранить уникальное наименование пакетов с библиотеками, Mandriva в то же время пытается реализовать поддержку multiarch *в наиболее полном объеме*: обеспечить как возможность равноправного сосуществования пакетов разной архитектуры, так и сборку пакетов под обе архитектуры в гибридной среде. Чтобы реализовать эти возможности в полной мере, требуется введение архитектурно-зависимых подкаталогов `/usr/bin`, не предусмотренных стандартом FHS. На наш взгляд, поддержка multiarch в полном объеме является слишком трудной и довольно авантюрной задачей: в конечном счете она потребует полного разделения файловой системы на иерархии `/32` и `/64`.

Проблема multiarch требует дальнейшего рассмотрения и обсуждения.

13.3 Файлтриггеры

Файлтриггеры (filetriggers) используются для выполнения «системных» действий над файлами после установки или удаления пакетов. Реализация файлтриггеров, используемая в rpm5, была разработана в Mandriva несколько лет назад (rpm.org не поддерживает файлтриггеры). Предлагаемая нами реализация основана на ранней реализации Mandriva, но в значительной степени отличается от нее. Рассмотрим кратко особенности реализаций.

В процессе выполнения транзакции (т. е. в процессе установки и удаления пакетов) rpm формирует список файлов, затронутых транзакцией. В реализации Mandriva файлы разделяются на «удаленные» (отмечены знаком «−») и «добавленные» (отмечены знаком «+»). Однако такое разделение часто не имеет смысла, поскольку при обновлении пакета одни и те же файлы будут считаться сначала удаленными, а потом добавленными. Поэтому в нашей реализации поддерживается единый список файлов, затронутых транзакцией. Перед запуском файлтриггеров список сортируется, и дубликаты в списке удаляются, благодаря чему последующий отбор файлов выполняется быстрее. В тех случаях, когда файлтриггеру нужно знать, был ли файл удален или добавлен, можно использовать фактический доступ к файловой системе: если файл отсутствует, то, очевидно, он удален; в противном же случае файл был установлен или обновлен.

В реализации Mandriva каждый файлтриггер состоит из двух частей: файл `foo.filter` содержит регулярное выражение, которое является условием срабатывания триггера; rpm считывает регулярное выражение и сопоставляет его со списком файлов, затронутых транзакцией. Если подходящие файлы найдены, то выполняется скрипт `foo.trigger`; список подходящих файлов подается на вход скрипта. В нашей реализации каждый триггер представлен единственным скриптом `foo.filetrigger`; при этом запускаются все имеющиеся триггеры с полным списком файлов на входе. Триггеры должны сами выполнять отбор файлов — для этого можно использовать команду `grep(1)` или встроенные средства интерпретатора (`while read line`). Тем не менее, часто такая реализация оказывается быстрее. Дело в том, что для многих триггеров требуется найти только первый подходящий файл, и тогда оставшуюся часть списка можно не обрабатывать (именно так работает команда `grep -q PATTERN`). Если же поиск подходящих файлов выполняется средствами rpm, то, по условию, rpm должен составить полный список подходящих файлов для каждого триггера.

В целом, наша реализация заметно проще и экономнее в средствах: мы сразу отказались от того, чтобы добавлять в rpm низкоуровневый код поиска по регулярному выражению. Несмотря на использование внешних средств, реализация не проигрывает в эффективности.

13.4 Компоновка с библиотеками

Во время компоновки программы или разделяемой библиотеки компоновщик `ld(1)` получает опции командной строки `-lfoo`, которые указывают на подключение внешних библиотек. При этом логика работы компоновщика в принципе может быть *тупой* (dumb) или *умной* (smart). *Тупая* логика состоит в том, чтобы в точности удовлетворить запрос командной строки — компоновщик ограничивается преобразованием аргумента `-lfoo` в путь и затем в имя (soname) библиотеки. В *умном* же режиме компоновщик занимается *интерпретацией* аргументов в рамках некоторой модели данных, т. е. выполняет дополнительный анализ, с помощью которого можно установить, какие из указанных библиотек действительно необходимо подключить, а какие был указаны «на всякий случай»; и, кроме того, все ли необходимые библиотеки были указаны.

По умолчанию компоновка выполняется в режиме, который можно считать

«тупым»: компоновщик подключает все указанные библиотеки и не выполняет анализа недостающих библиотек. «Тупой» режим имеет право на существование и обладает тем неоспоримым преимуществом, что в нем в точности выполняется запрос пользователя — ведь нельзя сказать, что программа работает неправильно, если она делает в точности то, что от нее требуют. Но на самом деле режим компоновки по умолчанию не является полностью «тупым» — возможность использования версионированных интерфейсов типа `GLIBC_2.4` означает, что компоновщик *должен* заниматься интерпретацией аргументов. А именно, компоновщик должен выполнить процедуру разрешения символов, чтобы ассоциировать неопределенный символ `foo` с имеющийся библиотечной функцией `foo@GLIBC_2.4`. Процедуру разрешения символов можно считать главным признаком «умного» компоновщика. Поскольку «умная» компоновка уже необходима для использования версионированных интерфейсов, то нет смысла в остальном поддерживать видимость «тупой» компоновки.

Главный принцип «умной» компоновки должен состоять в следующем: при компоновке должны быть подключены все те и только те библиотеки, которые используются для разрешения неопределенных символов компонуемой программы (или библиотеки). Если библиотека не используется для разрешения какого-либо символа, то компоновщик должен счесть ее избыточной и не добавлять в список `DT_NEEDED`. В то же время все неопределенные символы должны быть разрешены. Именно такой подход естественным образом работает при статической компоновке. Проводя аналогию между статической и динамической компоновкой, можно придти к заключению, что «точное выполнение запроса пользователя» не следует трактовать слишком буквально; в большей степени нас интересует собственно результат компоновки.

На практике использование «тупого» режима компоновки приводит к подключению большого числа лишних библиотек. Как описано в разделе 8.3, `pkg-confg(1)` рекурсивно дополняет список библиотек `Libs`, хотя в таком дополнении обычно нет необходимости. В других случаях лишние библиотеки указывают для того, чтобы обеспечить максимальную переносимость сборочных скриптов и т. п. Проблема компоновки с лишними библиотеками имеет несколько аспектов. С точки зрения rpm-пакетов компоновка с лишними библиотеками приводит к образованию лишних зависимостей. Внедрение set-версий сделает этот аспект проблемы более очевидным: библиотеки, которые требуются для разрешения символов, будут иметь версионировнные зависимости `Requires: libfoo.so.0 >= set:...`, а «лишние» библиотеки получат зависимости без версий `Requires: libbar.so.0`. Однако еще более важным аспектом проблемы является собственно лишняя нагрузка на `ld.so(1)` во время выполнения. На самом деле проблема лишних библиотек не очень часто приводит к *загрузке* лишних библиотек, так как неиспользуемые напрямую библиотеки обычно все же требуются косвенно. Но в то же время проблема лишних библиотек может значительно замедлить процедуру разрешения символов, т. к. процедура разрешения символов основана на рекурсивном поиске «в ширину»: поиск символов выполняется в строгой последовательности сначала в непосредственно подключенных библиотеках, затем на первом уровне косвенности и т. д. Поскольку значительная часть используемых символов должна быть разрешена на раннем этапе работы программы, проблема лишних библиотек на практике может заметно замедлить запуск больших приложений.

Стандартный компоновщик `ld(1)` из комплекта `binutils` предоставляет две опции, которые реализуют «умный» режим. С опцией `--copy-dt-needed-entries` компоновщик выполняет поиск недостающих библиотек, которые «забыли» указать в командной строке. Поиск выполняется в дереве уже подключенных библиотек. Например, если программа компонуется с библиотекой `-lgtk-x11-2.0`, но при этом дополнительно использует функции библиотеки `libglib-2.0`, то библиотека `libglib-2.0` будет подключена автоматически, т. к. она используется в библиотеке `libgtk-x11-2.0`. Опция `--as-needed` в свою очередь «отсеивает» библиотеки, которые «не пригодились» для разрешения неопределенных символов.

Мы предлагаем активировать опции «умной» компоновки на уровне gcc, т. е. модифицировать спецификацию компоновки (см. `gcc -dumpspecs`) таким образом, чтобы `gcc(1)` запускал `ld(1)` с опциями «умной» компоновки. Тогда, учитывая то, что `ld(1)` редко используется напрямую, режим «умной» компоновки будет доступен по умолчанию для всех пакетов в репозитории. Таким образом, для исправления проблемы компоновки с лишними библиотеками потребуется просто пересобрать пакеты. Однако режим «умной» компоновки предъявляет некоторые дополнительные требования: опции подключения библиотек `-lfoo` должны быть указаны в командной строке *после* объектных файлов, использующих библиотеки. Аналогичное требование предъявляется при статической компоновке. Дело в том, что компоновщик обрабатывает аргументы командной строки последовательно и только один раз; если библиотека, подключаемая на текущем шаге, оказывается неиспользуемой, то она отбрасывается и в дальнейшем не рассматривается. В результате компоновка с неправильным порядком аргументов (например, `gcc -lfoo main.o -o prog`) перестанет работать, что может привести к нарушению собираемости некоторых пакетов. Впрочем, поскольку «умная» компоновка уже используется в других дистрибутивах, неисправленных пакетов почти не осталось. В тех редких случаях, когда «умный» режим компоновки оказывается нежелательным, можно восстановить «тупой» режим, передав компилятору флаг `-Wl,--no-as-needed`.

Возможность добавления флагов в спецификацию gcc оказывается довольно привлекательной в том отношении, что она позволяет строго проводить системную политику сборки пакетов: некоторые флаги будут использованы всегда, или, во всяком случае, по умолчанию, а не будут оставлены на усмотрение пакетов. Такой подход оправдывает себя не только в случае с компоновкой, но и в случае компиляции, когда речь идет о безопасности кода. Так, в одном российском дистрибутиве в спецификации gcc по умолчанию добавлен флаг `-fstack-protector`, который добавляет защиту от переполнения буфера на стеке, а также флаг `-D_FORTIFY_SOURCE=2`, который добавляет проверку переполнения буфера для стандартных функций `strcpy`, `memmove` и др.