Комплементарное хеширование подмножеств

Алексей Турбин <at@altlinux.org>

16 июля 2010 г.

Аннотация

Обсуждается разрешимость символов в программах, динамически компонуемых с разделяемыми библиотеками. Предложен новый вид зависимостей для rpm-пакетов. Рассмотрена возможность *комплементарного хеширования* — способа хеширования двух множеств R и P, который сохраняет возможность проверки $R \subset P$.

1 Зависимости на разделяемые библиотеки

Для запуска программы загрузчик `ld.so` конструирует исполняемый образ программы. При этом происходит загрузка необходимых разделяемых библиотек. Разделяемая библиотека характеризуется своим *именем* (soname). Кроме того, каждая разделяемая библиотека содержит экспортируемые *символы* (т. е. функции и глобальные переменные библиотеки). Для работоспособности программы существенной является не только возможность загрузки всех необходимых разделяемых библиотек по их имени, но и *разрешимость* всех символов, используемых в программе.

Зависимости rpm-пакетов напрямую отражают связи между программами и разделяемыми библиотеками, используя имена библиотек: пакет с разделяемой библиотекой предоставляет зависимость вида `Provides: libfoo.so.1`, а пакет, который использует библиотеку, требует зависимость `Requires: libfoo.so.1`. Однако информация о символах напрямую в зависимостях не используется.

Ясно, что зависимость на имя библиотеки может оказаться недостаточной — например, это можно быть связано с добавлением новых символов в библиотеку. 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).

Один из подходов, который позволяет в некоторой степени решить проблему обратной совместимости — это *версионирование символов* (набор новых символов библиотеки снабжается специальной меткой). Такой подход используется, в частности, в библиотеке `glibc` [DSO howto]. Однако этот подход нельзя автоматизировать, а из-за ограниченной совместимости он не получил широкого распространения.

2 Модель зависимостей с учетом символов

Символы ассоциируются с библиотеками, то есть считается установленным локальное соответствие между каждым экспортируемым символом и соответствующей ему библиотекой. (Формат ELF, вообще говоря, не требует такого соответствия. Однако подобная возможность реализована, например, в OpenSolaris [Direct Binding]).

Тогда пакет с разделяемой библиотекой предоставляет зависимость `Provides: libfoo.so.1` и множество экспортируемых символов этой библиотеки `foo1, foo2, foo3`. А пакет, который использует библиотеку, требует зависимость `Requires: libfoo.so.1` и множество используемых символов `foo1, foo2, ...`

Зависимость `Requires` считается удовлетворенной, если множество используемых символов является *подмножеством* экспортируемых символов (т. е. если все требуемые символы предоставлены).

Данная модель является слишком громоздкой. Но вообще-то 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 $R \subset P$?

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...`

3 Комплементарное хеширование

Пусть заданы множества R и P, а также определен предикат $R \subset P$. *Схемой комплементарного хеширования* мы называем тройку $\langle H_R, H_P, \subset^* \rangle$, состоящую из двух функций хеширования $H_R(R) \to R^*$, $H_P(P) \to P^*$ и предиката $R^* \subset^* P^*$. При этом если $R \subset P$, то должно всегда выполняться и $R^* \subset^* P^*$. А если $R \not\subset P$, то $R^* \not\subset^* P^*$ выполняется с вероятностью $1 - \varepsilon$, где параметр ε задает одностороннюю ошибку, обусловленную потерей информации при хешировании. Односторонний характер ошибки 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. Будем требовать, чтобы для различных множеств R их хеш отличался (или «почти всегда» отличался). Тогда для представления такого хеша требуется хотя бы один бит на каждый всевозможный элемент (т. к. каждый элемент либо входит, либо не входит во всевозможные множества R). Другими словами, хеш не может быть короче битовой шкалы (которая, в свою очередь, не содержит избыточной информации). Кроме того, универсальной битовой шкалы не существует.

Таким образом, в название доклада вынесен негативный результат.

4 Компактное кодирование множества строк

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.

Приведем некоторые численные оценки. Пусть для представления множества P, состоящего из $1024 = 2^{10}$ символов, используется 20-битный хеш на элемент. 20-битный хеш позволяет адресовать 2^{20} элементов. Тогда вероятность коллизии с некоторым «случайным» символом, не входящим во множество P, составит $2^{10}/2^{20} = 2^{-10} \approx 0.1\%$.

Вычислим энтропию множества P. Множество P можно рассматривать как выбор 2^{10} элементов из 2^{20} элементов. Тогда энтропия $log_2|P| = log_2\binom{2^{20}}{2^{10}} \approx 11710$ битов. Таким образом, оптимальный способ упаковки элементов может значительно снизить размера хеша (вместо 20 битов на элемент в упакованном виде потребуется примерно 11.5 битов на элемент).

В качестве процедуры упаковки элементов можно использовать, например, дельта-кодирование (предварительно выполнив сортировку элементов). Другая процедура кодирования, близкого к оптимальному, предложена Д. В. Чистиковым [Кодирование сочетаний]. Используется представление множества целых чисел в виде дерева, причем каждый узел дерева кодирует как значение элемента, так и (дополнительно) диапазоны значений «потомков».

5 Тестовая реализация

На момент публикации реализованы основные алгоритмы, необходимые для поддержки set-версий в зависимостях на разделяемые библиотеки, и выполнена тестовая пересборка репозитория. Нереализованной остается процедура упаковки элементов (set-версии представлены в упрощенном виде).

По результатам тестовой переборки удалось оценить «размерность» задачи. На архитектуре x86-64 репозиторий предоставляет 6740 разделяемых библиотек, причем каждая библиотека экспортирует в среднем 776 символов. Кроме того, 9947 пакетов в репозитории содержат зависимости на разделяемые библиотеки; каждый из них требует в среднем 9 библиотек для разрешения 269 символов. Таким образом, всего в репозитории нужно учитывать около 8 млн. символов. Тогда, если считать, что после упаковки каждый символ будет занимать примерно 12 битов, то для хранения информации о символах потребуется 12 Мб (на данный момент размер файла `pkglist.classic.bz2` — примерно 4 Мб).

Список литературы

[DSO howto] Ulrich Drepper. *How To Write Shared Libraries.*
`http://people.redhat.com/drepper/dsohowto.pdf`

[Direct Binding] *Relocation Processing, Symbol Lookup, Direct Binding.*
`http://docs.sun.com/app/docs/doc/817-3677/chapter3-2`

[Кодирование сочетаний] Д. В. Чистиков. *Кодирование сочетаний* (личное сообщение, ноябрь 2009 г.)