zrec — формат метаданных репозитория

Алексей Турбин

17 сентября 2018 г.

Аннотация

Новый формат сжатия записей обеспечивает высокий коэффициент сжатия и в то же время сохраняет возможность произвольного доступа к записям. Для этого поток записей разбивается на куски (chunking). Поддерживается синхронизация / частичное скачивание через HTTP range requests.

1 Введение

Перед обновлением из репозитория скачивается большой файл с *записями пакетов* (records); в `apt-rpm` это файлы `pkglist.xz` и `srclist.xz`, содержащие урезанные rpm-заголовки (в `repomd`-репозиториях записями можно считать сегменты XML-файла). Файл `pkglist.xz` хорошо сжат, но распаковка его занимает несколько секунд, и далее он хранится в `/var/lib/apt` в разжатом виде, занимая довольно много места (200–300 Мб; здесь не столько жалко места на диске, сколь возникают «тормоза» при доступе к файлу). Пережать же в более «легкий» формат `pkglist` нельзя, поскольку `apt` требует произвольный доступ к записям (команда `apt-cache show` делает `lseek(2)` и считывает запись).

В 2017 г. автор предпринял попытку разработать «легкий» формат `zpkglist` для сжатия записей после скачивания [1]. За основу была взята библиотека сжатия `LZ4`; для нее была адаптирована техника сжатия со словарем, реализованная в библиотеке `zstd`. Перед сжатием записи группировались по 4 штуки, что значительно улучшало коэффициент сжатия, однако произвольный доступ уже требовал распаковки группы из четырех записей.

В мае 2018 г. Джонатан Дайетер (Jonathan Dieter) анонсировал похожий проект `zchunk` [2] (формат файла и библиотека сжатия). Поддержка `zchunk` добавлена в `dnf`, планируется к использованию в Fedora 29 либо Fedora 30. `zchunk` основывается на алгоритме `zstd`, который способен, в отличие от `LZ4`, обеспечить значительно более высокий, «релизный» уровень сжатия (хотя и несколько не дотягивает до `xz`). Поэтому интерес автора стал смещаться в сторону использования нового формата для сжатия на сервере (сжатый файл не требует распаковки на клиенте).

Файл в формате `zchunk` можно понимать просто как сжатый поток байтов. При сжатии поток нарезается на куски (chunking) и каждый кусок сжимается хотя и отдельно, но *со словарем*, хранящимся в том же файле (что значительно улучшает

коэффициент сжатия). Естественно, при сжатии записей нарезка идет по границам записей, но в одном куске может содержаться и несколько записей. В начале же файла хранится *индекс* кусков: их размеры и хеш-суммы. Наличие индекса делает возможным *синхронизацию*: клиент сначала скачивает индекс и потом — через HTTP range requests — недостающие куски, перестраивая старый файл в новый.

Формат `zchunk` не устраивает автора лишь в деталях. Далее рассмотрены некоторые специальные особенности сжатия и синхронизации, которые приводят к созданию альтернативного формата — `zrec` (сокр. от compressed records).

2 О пользе сортировки

Главным просчетом в формате `zchunk` является его общность: он сжимает какие-то куски, которые распаковываются во что угодно. Формат `zrec` вместо этого постулирует, что сжимаемые записи *упорядочены* каким-либо образом (напр., естественным образом — по имени пакета, но возможна и группировка по `src.rpm`). Этот постулат имеет далеко идущие последствия.

Во-первых, в отсортированном списке у записей возникает *локальное сходство*, которое «не улавливается» словарем. Например, пакеты `perl-*` имеют между собой сходство в части зависимостей и т. п. Значит, группировка соседних записей при нарезке должна привести к значительному улучшению сжатия. А библиотека тогда может реализовать процедуру сжатия с автоматической группировкой.

Во-вторых, сортировка меняет требования к сопоставлению кусков. Это даже две разные математические задачи: 1) чтобы уникально идентифицировать каждый кусок среди всех остальных, требуется 96-битный хеш (при числе кусков порядка 10^5 и вероятности коллизии порядка 10^{-18}); 2) чтобы определить, изменился ли кусок, требуется 60 битов хеш-материала (т. к. $2^{-60} \approx 10^{-18}$). Поэтому можно считать, что когда куски «расставлены по местам», то для идентификации куска в «локальной окрестности» достаточно 64-битного хеша.

3 Группировка записей

Группировка записей может заметно, хотя и не радикально, улучшить сжатие. Возьмем файл `srclist.xz` (1.86 Мб) — в разжатом виде 10 Мб. «Солидное» сжатие `zstd -19` дает 1.97 Мб. Если сжимать каждую запись по отдельности, получается 5.8 Мб — мало избыточности! Радикальное улучшение дает сжатие записей со словарем — 2.44 Мб (словарь 512 Кб, в сжатом виде 177 Кб). Если теперь еще группировать записи парами, получается 2.26 Мб, по три — 2.21 Мб, по четыре — 2.18 Мб и далее очень медленно. В общем, группировка по 2-3 записи улучшает сжатие на 8-10%, а также уменьшает в 2-3 раза размер индекса.

Записи однако нельзя группировать в ровных количествах, это приведет к *сдвигу границ* и сделает невозможным синхронизацию. Так, если из потока записей, сгруппированных парами: `AB CD EF...` будет удалена запись `B`, то границы пар сдвинутся: `AC DE F...` и все куски перестанут совпадать. Классический подход к решению этой проблемы состоит в том, что нужно нарезать куски переменной

длины псевдослучайным образом, основываясь на данных внутри кусков (content-defined chunking). А именно, данные сканируются скользящим окном и нарезаются, когда контрольная сумма в окне принимает определенное значение. Упрощенная реализация такого подхода известна как rsyncable gzip. Об оптимальной нарезке см. статью 2005 г. [3] и свежую работу [4].

Наша идея *ультракороткой оптимальной нарезки* состоит в следующем: записи можно сравнивать по их хеш-коду. Тогда если напр. $\underline{A < B < C} > \underline{D < E}$, то упорядоченная подпоследовательность и образует группу, а нарушение порядка дает разрез. Нарушение порядка можно также вынужденно допустить в первых двух элементах группы. Это приводит к следующей **Стратегии**: 1) Поместить в очередь A, B и C. Если $B > C$, отрезать AB (и тогда C становится новым A). 2) Иначе добавить в очередь D. Если $C > D$, отрезать ABC. 3) Иначе отрезать $ABCD$.

Отметим, что хотя на шагах 1) и 2) сравнение одно и то же, вероятность его разная: $\Pr(B < C) = 1/2$, а $\Pr(C < D) < 1/2$. И поскольку условие одно и то же, то для преодоления рассинхронизации дается более одной возможности «зацепиться за нужное место», и в то же время рассинхронизация быстро пресекается.

Отметим также важную *практическую модификацию* процедуры нарезки: для использования в качестве $A, B \ldots$ желательно хешировать не всю запись, а лишь имя пакета (или имя `src.rpm` пакета без версии при группировке по `src.rpm`). Тогда изменение версии у пакета не приводит к изменению нарезки: изменение оказывается полностью локализованным внутри куска; к рассинхронизации может привести только удаление и добавление новых пакетов.

Мы взяли файл `srclist.xz` от 1 июня и 1 июля 2018 г. Без группировки записей для регенерации нового файла недостает 923 куска общим объемом 271 Кб, а с описанной группировкой — 809 кусков объемом 568 Кб. При этом группировка также уменьшает размер индекса примерно с 80 до 30 Кб, однако в данном случае увеличившийся объем кусков в несколько раз перекрывает уменьшение индекса. Видно, что при нерегулярных обновлениях группировка оказывается невыгодной.

4 Хеширование и расстановка

При любом изменения репозитория индекс будет скачиваться полностью, что может составлять значительную часть скачанного. Поэтому представляет интерес дальнейшее уменьшение объема хеш-материала внутри индекса.

Выделим подзадачу: пусть каждый кусок наделяется 32-битным хешем; нужно установить соответствие между кусками в старом и новом файле. Ясно, что соответствие определяется совпадением хеша, но при этом совпадения, нарушающие «общий порядок», считаются коллизиями и отсеиваются.

В общем виде эта задача известна как Longest common subsequence problem, и одним из ее решений является алгоритм Майерса [5], используемый в `diff(1)`. Однако алгоритм Майерса выполняется, вообще говоря, квадратичное время, как и все остальные решения этой задачи в наиболее общем виде или в худшем случае. Несколько точнее, сложность алгоритма Майерса — $O(nd)$, где n — общее число элементов в двух последовательностях, а d — число отличающихся элементов. Увы, на больших файлах `diff(1)` хорошо работает, только если изменения в них

оказываются небольшими. В нашем же случае при нерегулярных обновлениях число отличающихся кусков может быть большим. Однако существует другой алгоритм — алгоритм Ханта-Шиманского со сложностью $O((r+n)\log n)$, где r — общее число всевозможных совпадений в двух последовательностях. Этот алгоритм деградирует, когда число неоднозначных совпадений велико (например, из-за малой мощности алфавита, или когда в текстовых файлах много одинаковых строк). Но в нашем случае при общем количестве кусков порядка 2^{16} неоднозначными совпадениями (коллизиями 32-битного хеша) можно пренебречь, и задача расстановки решается за $O(n \log n)$. Позже были открыты другие алгоритмы с похожей асимптотикой, которые используются в биоинформатике [6, с. 291] (в биоинформатике задача расстановки называется global alignment). См. тж. обзоры [7, 8].

Естественно, 32-битного хеша маловато для надежного сопоставления. Поэтому формат вводит хеширование второго уровня: пусть $h(A)$ и $h(B)$ — хеши идущих подряд кусков, тогда в дополнение к ним вычисляется $H(A+B)$ — 32-битный «проверочный» хеш. Важно, что функции h и H отличаются, т. е. H предоставляет *дополнительную* информацию о каждом куске. Тогда каждый кусок оказывается наделенным 64-битным хешем (поскольку с точки зрения каждого куска другой кусок можно считать параметром). А составная конструкция содержит 96 битов хеш-материала, что уже достаточно для уникальной идентификации. Проверочный хеш используется на второй стадии синхронизации: если $h(A)$ и $h(B)$ совпадают, а $H(A+B)$ не совпадает, значит, куски A и B надо скачать повторно.

Теперь остается только применить тот же алгоритм группировки кусков, который применяется для группировки записей: для наделения кусков составным хешем их нужно нарезать по 2-4 штуки. Совпадение составных кусков разбивает задачу расстановки на мелкие подзадачи.

5 О надежности конструкции

Таким образом, мы получили, что для достаточно надежного сопоставления в индексе требуется всего около 5.5 байтов хеш-материала на кусок, или же около 2 байтов на запись. Конечно, такую конструкцию нельзя считать криптографически безопасной. Криптографическую проверку мы просто делегируем на уровень выше: когда файл перестроен, клиент проверяет криптографическую сумму перестроенного файла. В худшем случае, когда сумма не совпадает, остается только скачать весь файл заново.

Более вероятным представляется несовпадение из-за того, что клиент скачал разные версии `pkglist` и `release` файлов. Решением является включение в `release` не только суммы всего `pkglist.zrec` файла, но и отдельно суммы его индекса, чтобы такое несовпадение можно было бы сразу же обнаружить.

Список литературы

[1] Alexey Tourbin. `zpkglist` — Compressed file format
`https://github.com/svpv/zpkglist`

[2] Jonathan Dieter. What is `zchunk`?
`https://www.jdieter.net/posts/2018/05/31/what-is-zchunk/`

[3] Kave Eshghi, Hsiu Khuern Tang (2005). A Framework for Analyzing and Improving Content-Based Chunking Algorithms

[4] Wen Xia et al. (2016). FastCDC: a Fast and Efficient Content-Defined Chunking Approach for Data Deduplication

[5] Eugene W. Myers (1986). An O(ND) Difference Algorithm and Its Variations

[6] Dan Gusfield (1997). Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology

[7] L. Bergroth et al. (2000). A Survey of Longest Common Subsequence Algorithms

[8] Mike Paterson, Vlado Dancik (1994). Longest Common Subsequences