

МЕТОД УНІФІКАЦІЇ РОБОТИ ІЗ ФАЙЛОВИМИ ІЄРАРХІЯМИ ШЛЯХОМ ЗАСТОСУВАННЯ ДЛЯ ЇХ ОБРОБКИ ПАРАМЕТРИЗОВАНИХ ОПЕРАЦІЙ НАД ГРАФАМИ

При традиційному підході до обробки ієрархій файлів, для кожної конкретної області застосування розробляється власне програмне забезпечення. У статті запропоновано метод обробки, що забезпечує більш гнучкий і універсальний підхід. Розглянуто окремі питання дизайну. Наведена ємнісна та обчислювальна складності. Намічено подальші шляхи оптимізації.

Traditional approach to file hierarchies processing requires development of specific software for every use case. In this article, it is proposed the processing method, which provides more flexible and universal approach. Some design problems are considered. Analysis of capacity and computing complexity has been made. Further optimization ways are touched.

Вступ

Оскільки найпоширенішою структурою файлових систем є ієрархічна, то її можна представити у вигляді дерева, вершинами якого є файли та директорії. В межах статті, буде використовуватись прийнятий у *nix-системах підхід, коли директорії трактуються як особливий тип файлів.

Пропонується представити файлову ієрархію у оперативній пам'яті у вигляді структури об'єктів і працювати із структурою, а не з файловою системою на диску. Позбавившись таким чином необхідності мати доступ до файлової системи, ми отримуємо можливість працювати із представленням локальних та віддалених файлових систем, (Дерево може бути отримане шляхом сканування файлової системи або може бути завантажено із попередньо згенерованого файлу опису). Потенційно, це дає нам можливість порівнювати файлові ієрархії на віддалених комп'ютерах, стани однієї й тієї ж файлової системи у різні періоди часу, виділяти змінені файли для збереження до інкрементальної резервної копії тощо.

Постає питання ефективної організації роботи із файловими ієрархіями. Можна застосувати традиційний підхід, коли для кожної моделі застосування пишеться свій програмний комплекс, котрий реалізує жорстко закодований алгоритм: синхронізації, контролю версій, резервного копіювання. Однак такий підхід не забезпечує достатньої гнучкості і завжди існує вірогідність того, що конкретному користувачу не вистачатиме

якоїсь функції і доведеться писати свій комплекс практично з нуля.

З іншого боку, оскільки файлова ієрархія представлена деревом, тобто графом, то природно було б застосовувати для роботи з нею **бінарні операції над графами**. Детальний аналіз такого методу обробки файлових ієрархій і наведений у статті.

Операції над графами, застосовні для файлових ієрархій

Кожна операція виконується шляхом рекурсивного симетричного обходу обох графів і побудови графу-результату.

Об'єднання $G_1 \sqcup G_2$

У результуючому графі створюється посилення на кожен елемент з обох дерев. Однакові елементи зберігаються у одному екземплярі.

Перетин $G_1 \cap G_2$

У результуючому дереві зберігаються тільки елементи, спільні для обох графів, причому у одному екземплярі.

Різниця $G_1 \setminus G_2$

У результуючому дереві зберігаються тільки елементи, присутні у дереві G_1 і відсутні у G_2 .

Введемо допоміжну операцію:

Виключення $-G_1$

Із результуючого дерева видаляються елементи, що задовольняють певній умові.

Комбінуючи вказані операції, можна описати операції вибору файлів для синхронізації, резервного копіювання, системи контролю версій (на рівні файлів) тощо.

Постає питання, на базі чого виконувати **порівняння елементів**. Для забезпечення достатньої гнучкості та універсальності, не достатньо використовувати тільки розмір файлу чи контрольну суму. Тому для кожної операції слід ввести параметр, з урахуванням якого вона буде здійснюватись.

Структура запису про файл

Для однозначного представлення файлу у пам'яті нам потрібно мати достатній набір характеристик файлів:

- Ім'я
- Розмір
- Файлові атрибути
- Час створення
- Час зміни
- Чи це файл, чи тека

Крім того, необхідно мати засіб, що дозволить розрізнити файли однакового розміру та з однаковими іменами, не потребуючи завантаження до оперативної пам'яті вмісту файлів. З цією метою зазвичай застосовуються контрольні суми (хеші). Суттєвий недолік лише один – теоретична ймовірність колізій. Але для використовуваних нині алгоритмів вірогідність виникнення колізій вкрай низька: наприклад, для 160-бітового SHA-1 вона оцінюється як 1 співпадіння на 2^{80} наборів вхідних даних [1]. Якщо навіть цього не достатньо, завжди можна ввести комбінацію двох чи більше контрольних сум, обчислених за різними алгоритмами (такий підхід використовується зокрема у протоколі Secure Sockets Layer (SSL)). Оскільки на сучасних процесорах хеш-функції обчислюються набагато швидше, ніж йде зчитування з жорсткого диску, то суттєвого зниження швидкодії не відбудеться, зате опірність до колізій буде як мінімум не гірша за таку для найякіснішого із застосованих алгоритмів.

Реалізація методу

Система, реалізована на базі описаного методу, працює подібно до бази даних: у систему завантажуються дані про файлові ієрархії, а потім надсилаються спеціально складені **запити**. Система виконує операції, описані у запитах, застосовуючи при цьому кешування та різноманітні оптимізації. Тобто різноманітна функціональність (наприклад, резервне копіювання або синхронізація) реалізується написанням порівняно простого скрипту без повторної компіляції програми.

Причому скрипт може бути згенерований автоматично іншою програмою.

Описану вище функціональність можна також об'єднати у набір бібліотек – каркас (framework). Це дасть можливість повторного використання коду у інших проектах.

Синтаксис запитів

Запити можуть описуватись, наприклад, таким чином:

Операції

& – об'єднання

^ – перетин

\ – різниця

-- виключення

F_x – файлова ієрархія з номером x, задається шляхом до файлової ієрархії на диску, шляхом до файлу опису або присвоєння результату виконання операцій.

– Рядковий коментар

Приклади окремих запитів:

1. F₃ = F₁& F₂ : hash;

Об'єднання файлових ієрархій F₁ та F₂ з виключенням дублікатів, що мають однаковий hash і знаходяться у паралельних рівнях ієрархії. Результат присвоюється F₃.

2. F₃ = -F₂ : size > 100mb;

Присвоєння F₃ дерева F₂, окрім тих елементів, що мають розмір, більший за 100 мегабайт.

3. F₃ = -F₂: name = “*.tmp”

Виключити з F₂ всі файли, ім'я яких закінчується на *.tmp, і присвоїти отриману файлову ієрархію F₃. (застосовуються регулярні вирази).

Як і у випадку баз даних, швидкість виконання запиту у великій мірі залежить від оптимальності його складання.

Приклад скрипту

```
# Скрипт для складного інкрементного резервного копіювання.
F1 = /media/data/;
F2 = /media/backups/bak44;
F3 = /media/backups/bak44/tmp/;
F4 = /media/backups/bak44/mail/;
# файли для відправлення по e-mail
F5 = /media/backups/bak44/index.fil;
F6 = /media/backups/bak43/index.fil;
# файл опису файлової ієрархії
# вибираємо файли, яких не було у попередній резервній копії
# Це робиться з урахуванням хешу, тобто вибираються і змінені файли.
F7 = F1 \ F6 : hash, name;
# вибираємо всі тимчасові файли до окремого дерева
F8 = -F7 : name != “*.tmp|*.bak|*.~” ;
# вибираємо не тимчасові файли
```

```

F9 = F7 \ F8 : hash , name;
# Створюємо резервну копію даних
COPY F9 F2 ;
# Тимчасові файли переміщуємо до іншої
резервної копії
MOVE F8 F3;
# Генерація файлу опису
LIST F5 ;
# Вибираємо особливо важливі файли,
які слід надіслати по
# e-mail для віддаленого зберігання
F10 = -F9 : name != "*-important.doc";
COPY F10 F4;

```

Наведений вище скрипт може генеруватися із шаблону автоматично за допомогою допоміжних засобів (наприклад, у *nix-системах – це утиліта awk). При цьому іме-

нем резервної копії може бути поточна дата тощо. Але детальний розгляд таких дій виходить за межі даної статті.

Зауваження щодо дизайну системи

Вхідний скрипт може бути розібраний за допомогою табличного парсера. Запропонований синтаксис розроблявся з урахуванням такого використання.

Пропонується об'єкти типу File зберігати у пам'яті тільки у одному екземплярі, а у деревах зберігати тільки посилання на них (шаблон проектування flyweight [2]).

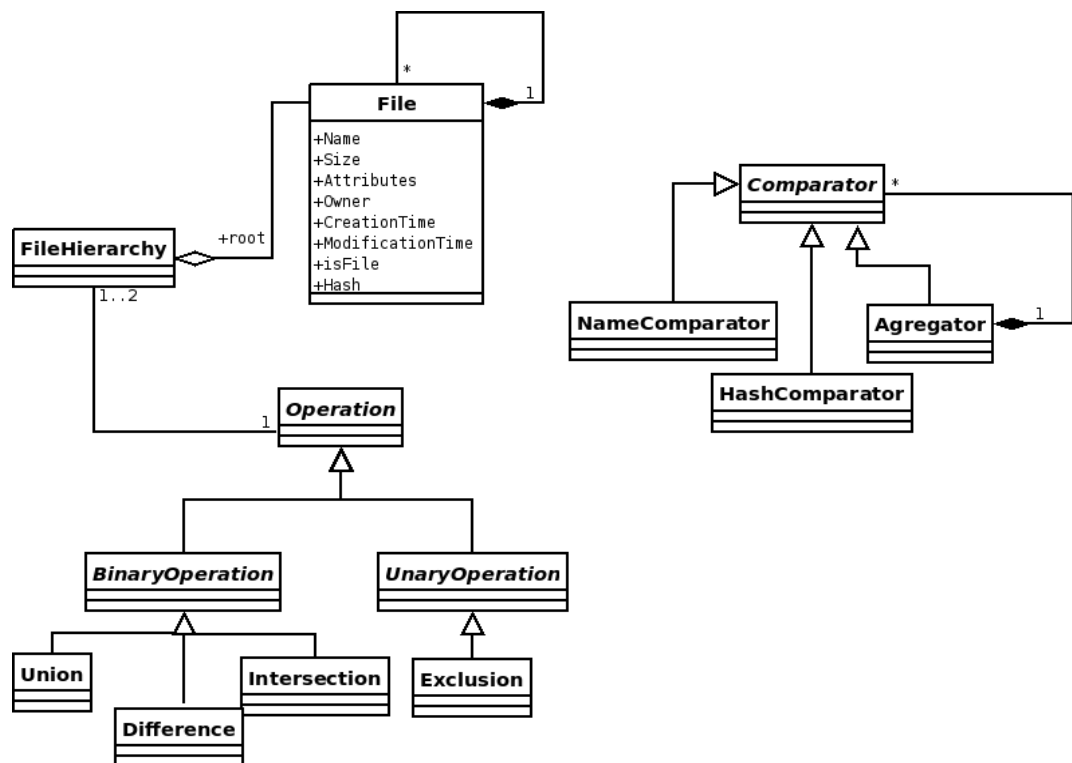


Рис. 1 Запропонований дизайн системи

Під час виконання операцій над деревами використовується лише операція перевірки рівності записів про файли за певним критерієм. Тому, підвищуючи рівень абстракції, можна ввести спеціальний абстрактний об'єкт-порівнювач: Comparator. Операції, що виконуються над деревами, параметризуються конкретними нащадками цього об'єкта. Таким чином, можна досягнути високого ступеня повторного використання коду і при цьому зберегти легкість розширення можливостей системи. Наприклад, щоб додати можливість роботи з файлами на основі їх власника, досить додати відповідне поле до об'єктів, що представляють файли, та оголосити нового нащадка класу Comparator. Якийсь Comparator може бути списком інших Comparator'ів, таким чином можна реалізувати обробку кількох операцій порівняння за раз (рис.1).

лізувати обробку кількох операцій порівняння за раз (рис.1).

Оцінка вимог до обчислювальних ресурсів

Можна обчислити обсяг пам'яті, необхідний для зберігання одного запису про файл. Зазвичай, використовуваними сьогодні операційними системами накладається обмеження на сумарну довжину шляху до файла та імені файла близько 512 байт [3]. Також, слід урахувати пам'ять, виділену під зберігання хешу (припустимо, 20 байтів), розміру файла (4 або 8 байтів), атрибутів, часу створення та модифікації, власника файла, тощо. Сумарно на один запис про файл у навіть у граничному випадку буде достатньо 550-600 байтів. Тобто для зберігання у пам'яті мільйону записів потрібно до 600мб RAM.

Дерево вимагає ще порядку десятків мегабайт оперативної пам'яті на зберігання посилань на записи про файли, а також зв'язки між елементами (4 або 8 байтів на одне посилання для 32-х та 64-х розрядних систем відповідно).

Обчислювальна складність залежить квадратично від середньої кількості файлів у кожному каталозі (зростає кількість порівнянь) і росте лінійно пропорційно зростанню кількості файлів. Тобто обчислювальна складність поліноміальна.

Як бачимо, для обробки навіть великої файлової ієрархії достатньо можливостей персонального комп'ютера.

Можливі оптимізації

Під час виконання бінарної операції із якимось параметром вперше, слід одночасно генерувати для поточного параметра дерево перетину та дерева різниць, якщо очікується повторна операція з цим самим параметром. Потенційне прискорення роботи від такої операції переважає накладні витрати пам'яті. Оскільки скрипт завантажується увесь одразу, то неважко передбачити, чи потрібні нам будуть ці дані.

Для операцій над файлами, можна ввести журналізацію на рівні операцій над цілим деревом. Журналізація використовується у базах даних та файлових системах типу ext3/ext4. І дозволяє у випадку раптового збою/вимкнення живлення уникнути втрати даних. Реалізується шляхом об'єднання виконуваних операцій у атомарні транзакції,

кожна з яких або виконується цілком, або не виконується взагалі і реалізується за принципами **ACID** (*Atomicity, Consistency, Isolation, Durability*) – (Атомарність, узгодженість, ізоляція, стійкість). [4]

Користування програмою можна спростити, написавши графічну оболонку. У ній можна передбачити спрощений файловий менеджер для перегляду вмісту файлових ієрархій.

Підсумок

У статті запропоновано універсальний метод обробки файлових ієрархій на противагу розробленим для конкретних потреб, використовуваним сьогодні. Він базується на представленні файлових ієрархій як графів та виконанні над ними бінарних операцій над графами за заданими параметрами. Метод не накладає обмежень на апаратне забезпечення, оскільки його коректна реалізація матиме поліноміальну обчислювальну складність та вимоги до обсягу оперативної пам'яті, адекватні кількості оброблюваних файлів. Також розглянуто окремі питання проектування, оптимізації та практичного використання описаного методу.

Реалізація запропонованого методу обробки файлових ієрархій у вигляді окремої бібліотеки або каркасу (framework) дасть можливість повторного використання коду завдяки застосуванню готових відтестованих та оптимізованих алгоритмів та спростить написання програм, що інтенсивно працюють з файловими ієрархіями.

Список посилань

1. Bruce Schneier, Cryptanalysis of SHA-1 http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html
2. Гамма Э., Хелм Р., Джонсон Р., Влссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001. – 368 с
3. <http://en.wikipedia.org/wiki/File:Filename>
4. Gray, Jim; and Reuter, Andreas; Distributed Transaction Processing: Concepts and Techniques, Morgan Kaufman, 1993 (ISBN 1558601902)

Поступила в редакцію 17.12.2010