

Chương 5

Lập trình & Kiểm thử

Chương này nêu một số đoạn mã trong quá trình lập trình và kiểm thử, cùng với ảnh chụp màn hình của kết quả cuối cùng.

5.1 Lập trình

5.1.1 Coroutine

Về mặt lập trình, trong quá trình viết ứng dụng, một trong các khó khăn chủ yếu là tăng tốc ứng dụng bằng coroutine. Trang sau trình bày một trong những đoạn mã dùng coroutine trong ca sử dụng tìm kiếm, có sử dụng cả LiveData:

Đoạn mã đó trả về một đối tượng `LiveData<List<List<Metadata>>>`:

- **Metadata:** Đối tượng chứa một kết quả tìm kiếm, có thể là `Comic`, `Series`,...
- **List<Metadata>:** Chứa kết quả tìm kiếm của một DAO, các `Metadata` trong danh sách này thuộc cùng một kiểu.
- **List<List<Metadata>>:** Danh sách kết quả của một DAO, nếu có kết quả (không rỗng) sẽ được thêm vào danh sách tổng hợp. Danh sách tổng hợp chính là danh sách kết quả tìm kiếm *chưa làm phẳng* (xem lại ca sử dụng tìm kiếm về cách làm phẳng).
- **LiveData:** Dùng cho tính năng data binding với View.

```

suspend fun search(query: QueryMultipleTypes, limit: Int) :
    LiveData<List<List<Metadata>>> = liveData(timeoutInMs = 3000) {
    val results2D = mutableListOf<List<Metadata>>()
    var latch = daos.size

    withContext(Dispatchers.IO) {
        daos.parallelForEach { dao ->
            val results = dao.search(query.query, limit)
            var shouldEmit = false

            synchronized(results2D) {
                if (results.isNotEmpty()) {
                    results2D.add(results)
                    shouldEmit = true
                }

                if (--latch == 0 && results2D.size == 0) {
                    shouldEmit = true
                }
            }

            if (shouldEmit) emit(results2D)
        }
    }
}

```

Đoạn mã làm những việc sau:

- Tạo ra 6 coroutine cho 6 DAO, tương ứng với 6 bảng, để tìm metadata.

daos.parallelForEach

Cụ thể hơn, đây là một hàm tự viết, có chữ kí như sau:

```

suspend fun <A> Iterable<A>.parallelForEach
    (f: suspend (A) -> Unit): Unit =
        coroutineScope { forEach { launch { f(it) } } }

```

Hàm này “mở rộng” giao diện `Iterable<A>` (trong trường hợp này là `daos`, có kiểu là `List<Dao>`). Nó nhận một hàm `f`, tạo một coroutine (`launch`) ứng với mỗi phần tử trong `Iterable<A>`, và chạy hàm `f` trong coroutine riêng đó.

Bản thân hàm `f` nhận vào một phần tử `A` (trong trường hợp này là `Dao`), và thực hiện tính toán trên coroutine riêng.

- Mỗi khi có kết quả (danh sách kết quả không rỗng), thêm vào danh sách tổng hợp, và thông báo cho `View` để hiển thị.

Biến `shouldEmit` kiểm soát xem có cần thông báo (`emit`) cho `View` không. Nếu danh sách kết quả từ một `Dao` không rỗng, nó được đưa vào danh sách tổng hợp.

6 `Dao` có tối đa 6 lần thông báo, giúp Màn hình Tìm kiếm được hiển thị từ từ theo quá trình tìm kiếm, đúng với yêu cầu phi chúc năng về việc liên tục cập nhật.

- Thông báo cho view ít nhất một lần để báo kết thúc tìm mà không có kết quả.

Trong trường hợp không có kết quả từ cả 6 `Dao`, thì cần thông báo danh sách tổng hợp rỗng cho `View`. Việc này được thực hiện bằng một `CountDownLatch`.

Mỗi khi `Dao` tìm xong, `latch` sẽ giảm đi 1 (`countDown()`). Khi `latch` bằng không, tức đã xong cả 6 `Dao`, mà danh sách tổng hợp vẫn trống (không tìm thấy gì), thì đặt `shouldEmit` để báo một lần cho `View`.

Do coroutine có thể chạy trên nhiều luồng khác nhau, nên cần một `CountDownLatch` để đếm (chứ không thể dùng một biến đếm thông thường) và môi trường `synchronized` để chạy. Nói cách khác, đoạn mã này là một “critical section” trong lập trình đa luồng.

- Toàn bộ 6 coroutine được chạy trên (các) luồng IO.

`withContext(Dispatchers.IO)`

Đây là mấu chốt để ứng dụng có cảm giác mượt mà khi tìm kiếm.

Việc tạo ra 6 coroutine có thể so sánh với cùng lúc tìm kiếm song song trên 6 luồng (thread) khác nhau, giúp giảm thời gian tìm kiếm. Tuy nhiên, nếu trong 6 luồng này lại có một luồng là *luồng giao diện* (UI thread), thì ứng dụng sẽ bị treo, không phản ứng cho đến khi `Dao` trong luồng đó tìm xong.

Để tránh việc này, coroutine cần được đảm bảo không chạy trên luồng giao diện. Trong trường hợp này, 6 coroutine được chạy trên các luồng IO - một thread pool dành riêng cho nhập/xuất - phù hợp với bản chất công việc là đọc cơ sở dữ liệu.

Đoạn mã trên thể hiện hai tác dụng quan trọng của coroutine:

Tính năng	Vấn đề giải quyết
Chạy song song	Tăng tốc ứng dụng
Không chạy trên luồng giao diện	Giúp ứng dụng luôn phản hồi (responsive)

Mỗi khi ứng dụng cần đọc ảnh từ tệp nén, và khi truy cập cơ sở dữ liệu, hai vấn đề trên xảy ra và cần coroutine để giải quyết.

5.1.2 CBZParser

Chương 4 - Thiết kế đã giới thiệu sơ lược về cách CBZParser hoạt động, là lưu ý hai phần đầu-cuối tệp tin. Cụ thể hơn, mã nguồn của thư viện Apache Commons Compress được phân tích để tìm mẫu đọc (read pattern), và cache những phần được đọc nhiều. Sau đó, CBZParser tạo ra một luồng đọc ngẫu nhiên - là đầu vào yêu cầu của thư viện - nhưng bên dưới là `InputStream` thông thường và dữ liệu cache.

Thư viện Apache Commons Compress được chọn vì những lí do sau:

- Viết bằng Java: Tiêu chí này loại được zlib, dù là thư viện mạnh nhất nhưng viết bằng C++ và có API phức tạp; hơn nữa chẳng nào mọi thư viện đọc tệp ZIP đều chỉ là “vỏ”, “lõi” thực chất đều là zlib.
- Mở rộng: Tiêu chí này loại được thư viện ZIP tích hợp trong Java, vì nếu hỗ trợ Apache Commons Compress thì sau này có thể thêm một số định dạng nén được thư viện này hỗ trợ.

Mẫu đọc của API dùng chế độ tệp (xem lại Mục 2.5.1), không phải chế độ luồng (phù hợp với `InputStream` nhưng chậm và không thể nhảy cóc), có thể tóm tắt như sau:

1. Tìm “mục lục” - Central Directory

(a) Tìm đuôi mục lục

“Đuôi” mục lục được đánh dấu bằng chuỗi `0x06054b50`. Chuỗi này được dò ngược từ cuối tệp ZIP, và có thể dò tối đa 65557 byte.

(b) Đặt vị trí đọc ở đầu mục lục

Sau khi tìm được đuôi mục lục, ta tìm được vị trí trong tệp (offset) của mục lục, và cần đặt vị trí đọc ở đó.

2. Đọc mục lục

- (a) Đọc một đoạn nhỏ 4 byte ở đầu để kiểm tra có lỗi không
- (b) Đọc mục lục để có offset của từng “đè mục” - File Entry.

3. Đọc đè mục

Đọc đè mục từ đầu đến cuối để kiểm tra một số thông tin.

Dựa theo mẫu đọc trên, `CBZParser` cache - hay trong trường hợp này từ chính xác hơn là `buffer` - và đọc dữ liệu như sau:

- Một luồng đọc `InputStream` đọc đến hết tệp, buffer lại 128KB đầu tiên và 1MB cuối cùng.

Theo kinh nghiệm, 1MB là đủ để chứa mục lục. 1MB này sẽ giúp cho việc dò ngược và đọc mục lục không cần tạo mới luồng nhập, miễn là truy cập trong vòng 1MB cuối.

128KB ở đầu giúp cho một số kiểm tra chạy được.

Đến đây, luồng đọc đầu tiên có thể được bỏ đi. Nếu truy cập ra ngoài phạm vi đã lưu, ném ngoại lệ và ứng dụng thử lại với bộ đệm lớn hơn.

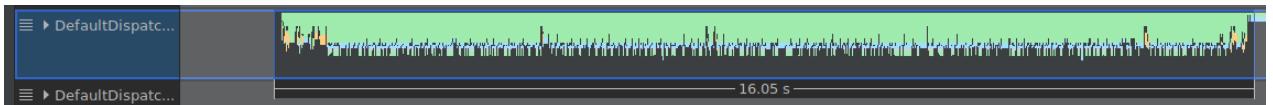
- Tạo mới luồng đọc thứ hai: chuẩn bị cho một loạt hoạt động đọc tuần tự các File Entry.

Quá trình sử dụng các thao tác trên trong `CBZParser` như sau:

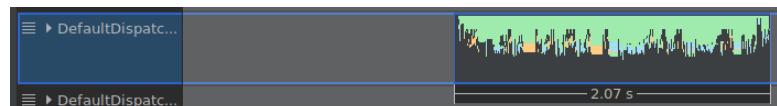
- Toàn bộ các thao tác trên được cài đặt trong `ZipBuffer` là một lớp cài đặt giao diện `SeekableByteChannel` - một luồng đọc ngẫu nhiên.
- `ZipBuffer` được truyền vào bộ đọc `ZipFile` của thư viện.
- Sau khi `ZipFile` khởi tạo xong, tức đọc xong mục lục, `CBZParser` lấy ra từ `ZipFile` danh sách tệp lẻ (File Entry) cùng với offset của nó.

Benchmark

Sử dụng Profiler tích hợp trong Android Studio, ta đo thời gian quét thư viện. Thời gian quét thư viện thể hiện được sự cải thiện tốc độ của `CBZParser` vì khi quét thư viện, từng tệp truyện phải được quét để tìm metadata. Baseline của so sánh này là cách cơ bản thứ hai trong Mục 4.3.2.



Hình 5.1: Baseline



Hình 5.2: Tối ưu

Ta có thể thấy với cách tối ưu, CBZParser có tốc độ nhanh hơn 8 lần so với cách đơn giản, vốn đọc và giải nén thừa rất nhiều.

5.2 Kiểm thử

5.2.1 ZipBuffer

Lớp này được kiểm thử riêng, do có quan hệ chặt chẽ với thư viện Apache Commons Compress. Cách kiểm thử là dùng lại nguyên các unit test cho ZipFile (đi kèm mã nguồn), nhưng thay vì truyền vào SeekableByteChannel thì truyền vào một ZipBuffer.

Các unit test này đều chạy thành công, và thông kê được một số trường hợp ngách (chủ yếu là khi dung lượng tệp nén lớn nhưng nhiều tệp nhỏ) cần 3 lần tạo mới luồng đọc, thay vì 2 như dự kiến.

5.2.2 Unit test

Các unit test của yacv tập trung vào phần cơ sở dữ liệu Room. Việc test cơ sở dữ liệu diễn ra theo bản mẫu sau:

```
/**
 * Tao moi va dua du lieu gia
 */
@Before fun createDb() = runBlocking {
    val context = InstrumentationRegistry.getInstrumentation().targetContext
    database = Room.inMemoryDatabaseBuilder(context, AppDatabase::class.java).build()

    database.comicDao().insertAll(testComics)
}
```

```

/**
 * Test thêm truyện trong folder mới hoàn toàn
 */
@Test fun test_addComic_newFolder() {
    database.comicDao().insert(testComic)

    assertEquals(database.folderDao().last().Uri, testComic.FolderUri)
}

```

5.3 Sản phẩm

5.3.1 Màn hình Quyền đọc

Màn hình xuất hiện khi mở ứng dụng mà không có quyền đọc dữ liệu (lần đầu dùng, người dùng chủ động bỏ quyền đọc trong Settings,...)



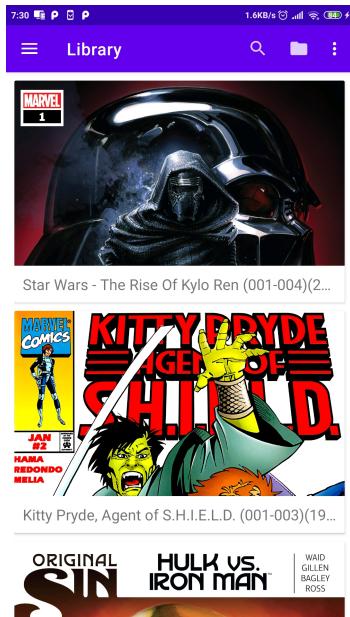
Hình 5.3: Màn hình Quyền đọc

5.3.2 Màn hình Duyệt truyện

Hai màn hình - Màn hình Thư viện và Màn hình Thư mục dùng để duyệt danh sách truyện. Màn hình Thư viện là màn hiển thị mặc định. Từ nó, khi ấn vào một thư mục, ta truy cập được Màn hình Thư mục tương ứng.

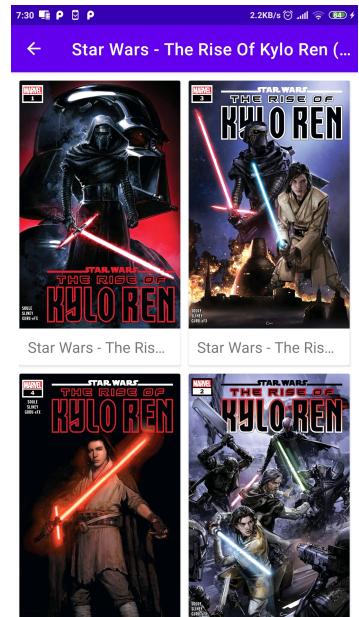


No folder selected, so yacy can't scan for comics. Click the [Folder icon](#) to select one!



(a) Màn hình Thư viện trống

(b) Màn hình Thư viện

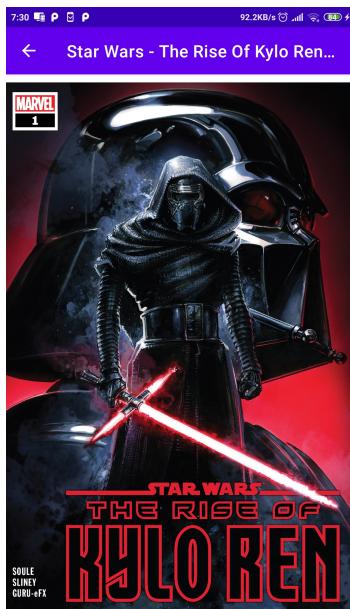


(c) Màn hình Thư mục

Hình 5.4: Hai Màn hình Duyệt truyện

5.3.3 Màn hình Đọc truyện

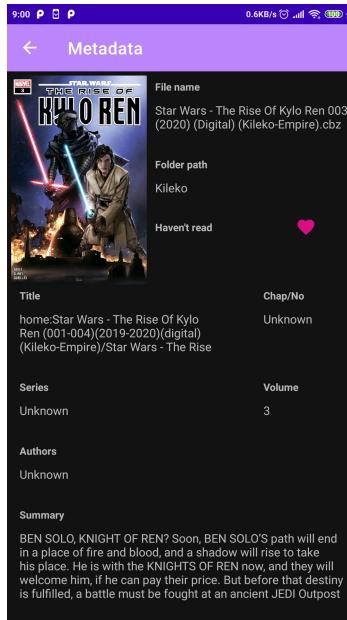
Màn hình Đọc truyện dùng để xem các trang truyện, được truy cập bằng cách ấn vào một truyện trong Màn hình Thư mục.



Hình 5.5: Màn hình Đọc truyện

5.3.4 Màn hình Metadata

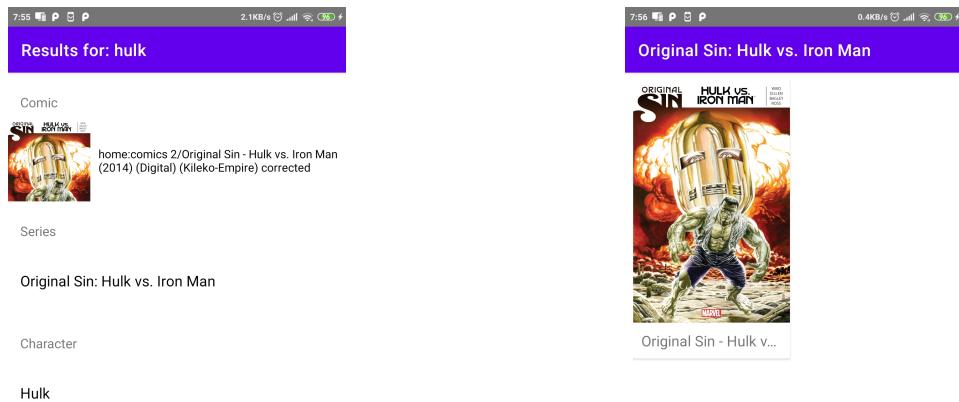
Màn hình này hiển thị metadata đi kèm, được truy cập từ Màn hình Đọc truyện.



Hình 5.6: Màn hình Metadata

5.3.5 Màn hình Tìm kiếm

Màn hình Tìm kiếm gồm hai màn hình con gần giống nhau, dùng để rút gọn số lượng kết quả hiển thị. Cả hai màn hình đều có đích đến là Màn hình Đọc truyện.



(a) Màn hình Tìm kiếm Tổng quan

(b) Màn hình Tìm kiếm Chi tiết

Hình 5.7: Hai Màn hình Tìm kiếm