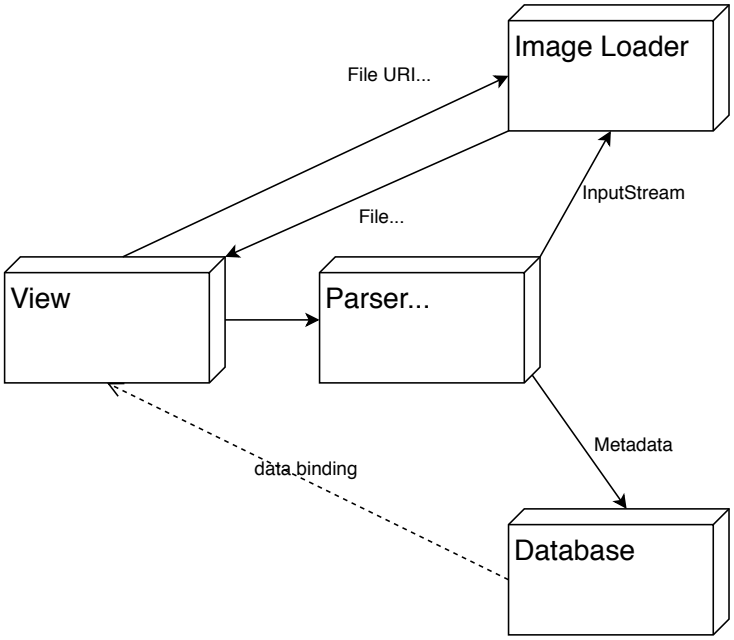
Chương 1

Thiết kế

Chương này tập trung vào thiết kế của ứng dụng, là triển khai cụ thể của **??**.



Hình 1.1: Kiến trúc tổng quan của ứng dụng

Kiến trúc tổng quan của yacv rất đơn giản, gồm 4 module như hình:

1. yacv *quét* metadata tệp truyện bằng module **Parser & Scanner** và lưu kết quả quét vào *cơ sở dữ liệu*, tức module **Database**
2. Các *Màn hình* trong module **View** hiển thị dữ liệu cho người dùng
3. Khi người dùng đọc truyện, yacv trích xuất và lưu đệm tệp ảnh bằng module **Image Loader**

4. Khi người dùng xem metadata, yacv trích xuất và hiển thị thông tin tệp truyện lên Màn hình bằng data binding với module **Database**

Ở **??** - **??**, trong phần về MVVM, yacv được giới thiệu là có sử dụng kiến trúc này. Tuy nhiên, MVVM chỉ là một phần nhỏ của ứng dụng, chủ yếu liên quan đến việc hiển thị dữ liệu nên chỉ có ý nghĩa khi xét đến các thành phần trong module **View**.

Ở **??**, yêu cầu về tránh trói buộc người dùng được nên lên đầu tiên trong các yêu cầu phi chức năng. Ở chương này, yêu cầu đó được hiện thực hóa bằng việc *yacv dùng phân vùng bộ nhớ chung*.

- Nếu dùng phân vùng bộ nhớ riêng, yacv phải chép các tệp truyện vào đó (tốn thời gian và dung lượng), hơn nữa các ứng dụng đọc truyện khác không thể thấy được tệp truyện ở khu vực này. Tuy nhiên, dữ liệu lưu ở phân vùng này có thể truy cập bằng API File của Java, giúp đơn giản đáng kể thiết kế ứng dụng.
- Dùng phân vùng bộ nhớ chung tránh được mọi điều trên, không buộc người dùng vào ứng dụng, tuy nhiên lại bị giới hạn chỉ được dùng API SAF.

yacv chỉ thiết kế cho *một người dùng*, do đó có rất ít tương tác, dẫn đến kiến trúc tối giản và rời rạc như trên. Các tiểu mục sau sẽ đi sâu vào các module này.

1.1 Module Database

Thông thường mục này được tách riêng ra, xếp vào mục *Thiết kế cơ sở dữ liệu*, ngang hàng với mục Thiết kế hướng đối tượng. Tuy nhiên, yacv còn cần xử lí dữ liệu khác quan trọng không kém là dữ liệu ảnh. Do không còn có vai trò trung tâm, duy nhất, phần cơ sở dữ liệu chỉ được coi là một module trong thiết kế hướng đối tượng của ứng dụng.

yacv chọn SQLite vì đây là một cơ sở dữ liệu gọn nhẹ nhúng sẵn trong Android. SQLite sử dụng mô hình quan hệ, do đó thiết kế bảng cần đảm bảo được chuẩn hóa (normalization).

Do không cần quản lí người dùng, cơ sở dữ liệu của yacv chỉ dùng để *lưu thông tin metadata*, cho phép ứng dụng quét dữ liệu ít lần hơn và tìm kiếm truyện. Theo như yêu cầu về metadata ở hai Phụ lục, và sau khi chuẩn hóa, ta có lược đồ cơ sở dữ liệu như sau:

Các bảng thực thể gồm:

- `Comic`: lưu thông tin *tập truyện lẻ*, là bảng trung tâm
- `Series`: lưu thông tin *bộ truyện*
- `Author`: lưu tên tác giả

- `Role`: lưu vai trò của tác giả trong một tập truyện
- `Character`: lưu tên nhân vật
- `Genre`: lưu tên thể loại truyện

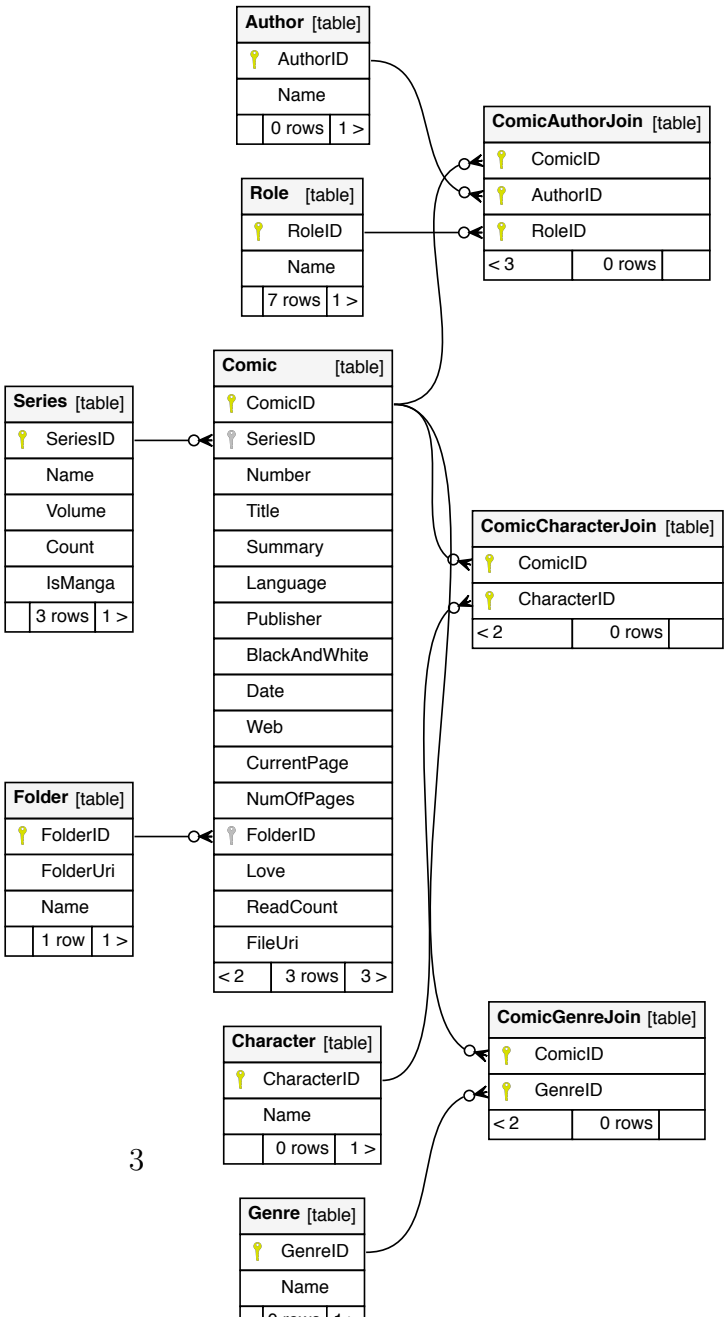
Một hạn chế quan trọng của các bảng `Character` và `Author` là chúng chỉ lưu thông tin tên, và chỉ phân biệt với nhau bằng tên. Nếu có hai tác giả/nhân vật trùng tên, yacv không thể phát hiện và hiển thị riêng.

Xét bảng trung tâm `Comic`. Bảng này có một số trường không phải metadata mà dùng để lưu thông tin của riêng ứng dụng, gồm:

- `CurrentPage`: lưu trang đang đọc
- `Love`: lưu trạng thái Yêu thích
- `ReadCount`: lưu số lần đọc

Trong lược đồ, có nhiều trường nhìn qua không cần thiết nhưng thực tế có ích, do thư viện SAF đã mô tả ở **??**:

- Trường `FileUri` trong `Comic`: Lưu đường dẫn của tệp truyện ở dạng URI.
- Trường `FolderUri` trong `Folder`: Lưu đường dẫn của thư mục ở dạng URI.
- Trường `Name` trong `Folder`: Tên thư mục. Thông thường nếu có đường dẫn, có thể tìm ra tên thư mục rất nhanh, tuy nhiên cũng do SAF mà việc này trở nên khó khăn, nên cần lưu riêng trường này.



Các trường URI đều cần có ràng buộc `UNIQUE`, do mỗi URI trỏ đích danh đến một đối tượng.

Ta xem xét đến các bảng nối:

- `ComicCharacterJoin`:
  - Mỗi tập truyện có thể có nhiều nhân vật và ngược lại, do đó `Comic` và `Character` có quan hệ Nhiều - Nhiều.
  - Chú ý rằng các nhân vật có quan hệ với tập truyện chứ không phải bộ truyện, vì có nhân vật phụ (không xuất hiện trong mọi tập truyện).
- `ComicAuthorJoin`:
  - Mỗi tập truyện có thể có nhiều tác giả và ngược lại, do đó `Comic` và `Author` có quan hệ Nhiều - Nhiều.
  - Chú ý rằng các tác giả có quan hệ với tập truyện chứ không phải bộ truyện, vì mô hình xuất bản nhiều truyện tranh là nhà xuất bản sở hữu nhân vật và thuê người viết.
  - Đồng thời, một tác giả có thể giữ vai trò khác nhau trong các bộ truyện khác nhau, do đó bảng này còn nối với bảng `Role`.
- `ComicGenreJoin`: Mỗi tập truyện có thể có nhiều thể loại khác nhau và ngược lại, do đó `Comic` và `Genre` có quan hệ Nhiều - Nhiều.

Do dùng Room, mỗi bảng ứng với một lớp. Các truy vấn với bảng cần đóng gói dữ liệu vào các lớp này, trước khi gửi đến hoặc nhận về từ *DAO* (Data Access Object). Mỗi câu lệnh lại được chuyển thành một hàm trong DAO.

1.2 Module View

Phần này tập trung vào các Màn hình, và phân tích chúng theo hướng MVVM.

Trong phần này có dùng nhiều biểu đồ tuần tự (sequence diagram) để minh họa tương tác của ba thành phần MVVM (cùng với một số thành phần liên quan) trong các ca sử dụng. Có một số điểm chung về các biểu đồ này:

- Trừ khi cần thiết, thành phần View sẽ được lược bỏ cho ngắn gọn.

- Đường thẳng nét đứt thể hiện tính năng data binding (tự động cập nhật View), và thường trỏ về ViewModel. Đáng ra, mũi tên này phải trỏ về View, nhưng do View bị ẩn đi, nên nó trỏ về ViewModel. Mặc dù không được đề cập đến trong phần giới thiệu về MVVM, đây thực ra là một chi tiết đúng về mặt kĩ thuật: ViewModel hoàn toàn đọc được luồng dữ liệu gửi đến View (hoặc ít nhất là đúng trong cách viết ứng dụng Android thông thường).

Các biểu đồ trạng thái cũng có một số chi tiết chung:

- Trừ khi nêu rõ, mọi trạng thái đều có thể là trạng thái bắt đầu (trạng thái khi mở ứng dụng) hoặc kết thúc (khi đóng ứng dụng).
- Mũi tên chuyển trạng thái tương ứng với *tương tác của người dùng*, do vậy thường được ánh xạ đến một phương thức trong View.
- Kí hiệu hình tròn đen chỉ dùng để tả trạng thái đầu *khi lần đầu dùng* yacv.

Biểu đồ lớp thường có một phương thức chung là "Get InputStream from ID". Cách truy cập để lấy `InputStream` của ảnh từ `ComicID` có thể tham khảo từ Màn hình Đọc truyện.

1.2.1 Nguồn dữ liệu - Repository - DAO - `ComicParser`

Như đã đề cập ở **??**, yacv sử dụng Kiến trúc Google khuyên dùng, vốn dựa trên MVVM. Phần này nêu rõ hơn cách triển khai MVVM của yacv trong phần nguồn dữ liệu (Model/Repository).

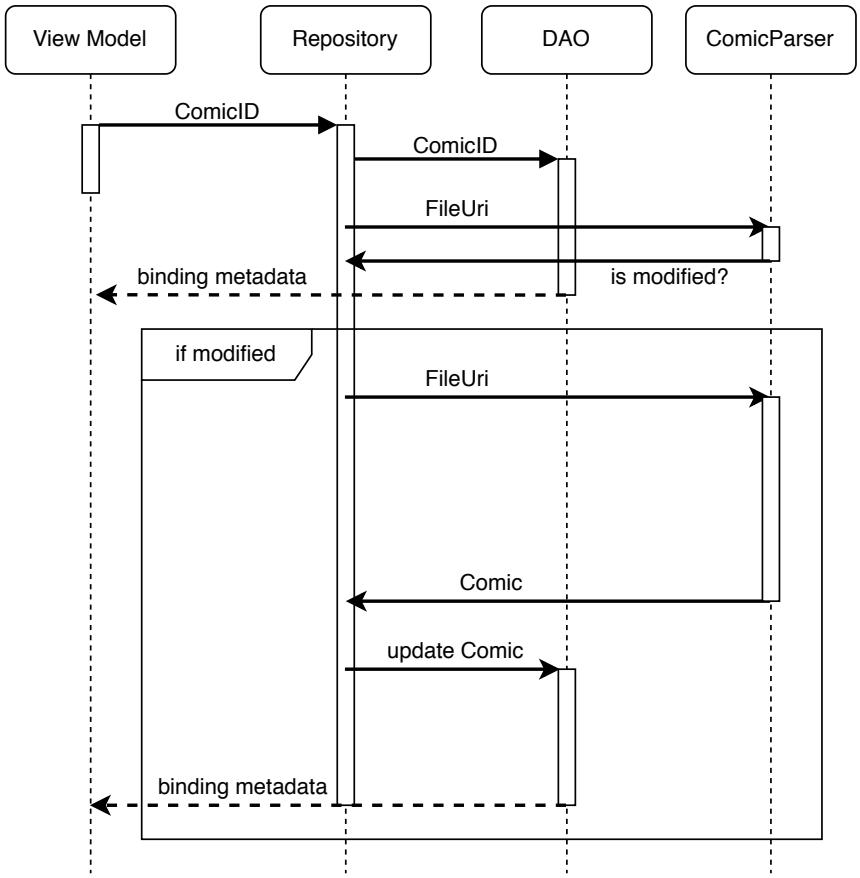
Dựa vào **??**, ta thiết kế được ba nguồn dữ liệu (model) sau:

Bảng 1.1: Ba nguồn dữ liệu tương đương với **??**

Tên	Tương đương với	Mục đích
`ComicParser`	Remote Data Source	Quét tệp để lấy metadata cập nhật nhất
DAO	Model	Lấy metadata từ cơ sở dữ liệu, tránh quét đi quét lại
Repository	Repository	Tổng hợp hai nguồn trên

Cụ thể hơn, `ComicParser` là bộ quét metadata tệp truyện (thuộc module Parser & Scanner, sẽ được mô tả sau). Lớp này nhận vào URI rồi trả về metadata của tệp truyện tương ứng dưới dạng đối tượng `Comic`.

Khi cần đọc dữ liệu metadata từ tệp truyện, ba thành phần này tương tác như sau:



Hình 1.3: Tương tác của ba nguồn dữ liệu, mũi tên gạch đứt thể hiện tính năng data binding

Mấu chốt ở đây là `ComicParser` dù có dữ liệu chính xác (trong trường hợp một ứng dụng khác sửa metadata tệp truyện) nhưng tốc độ rất chậm, còn cơ sở dữ liệu không chính xác nhưng rất nhanh, do đó *cơ sở dữ liệu làm bộ đệm cho parser*.

Repository làm nhiệm vụ gọi cả hai nguồn dữ liệu trên và cập nhật cơ sở dữ liệu (nếu cần) thay cho View/ViewModel. Hiện tại, Repository có thể không làm được nhiều, tuy nhiên nó giúp ích cho *khả năng mở rộng* của ứng dụng. Ví dụ, trong tương lai yacv có thể liên kết với một bên thứ ba cung cấp metadata cho truyện, khi đó để tích hợp API thì chỉ cần sửa phần Repository.

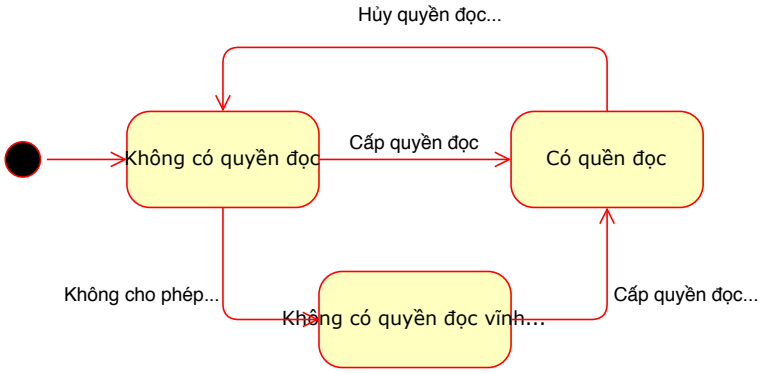
Cũng cần chú ý rằng việc đọc metadata từ tệp tin không phải là yêu cầu của mọi màn hình (cụ thể chỉ Màn hình Metadata cần), do đó trong đa số các ca sử dụng, *DAO đóng vai trò Model*, thay cho Repository. Tương tác trong Hình 1.3 vẫn được duy trì, tuy không có cả Repository lẫn `ComicParser`.

1.2.2 Màn hình Quyền đọc

Do sự phức tạp trong việc xin quyền của Android, một màn hình riêng để xin quyền đọc dữ liệu được tách ra khỏi Màn hình Thư viện, gọi là *Màn hình Quyền đọc*. Màn hình này sẽ là *màn hình đầu tiên* hiển thị khi dùng ứng dụng.

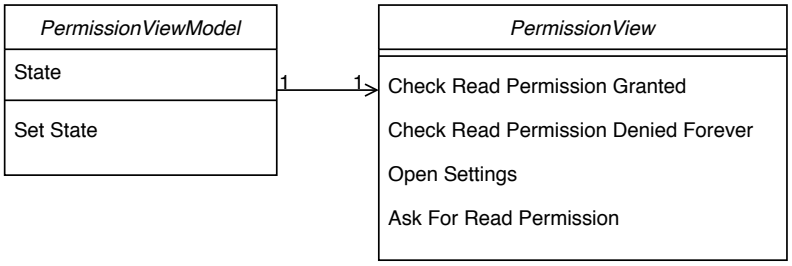
- Nếu có quyền đọc: chuyển ngay sang Màn hình Thư viện
- Nếu không: nêu lí do cần quyền, gợi ý người dùng cấp quyền

Hình sau mô tả trạng thái cấp quyền đọc của yacv (cũng như mọi quyền của một ứng dụng Android cơ bản nói chung).



Hình 1.4: Trạng thái cấp quyền của một ứng dụng Android

Dựa theo Hình 1.4, ta có biểu đồ lớp của ViewModel và View như sau:



Hình 1.5: Biểu đồ lớp của Màn hình Quyền đọc

1.2.3 Màn hình Thư viện

Màn hình Thư viện là một trong hai màn hình của ca sử dụng Duyệt truyện, bên cạnh Màn hình Thư mục.

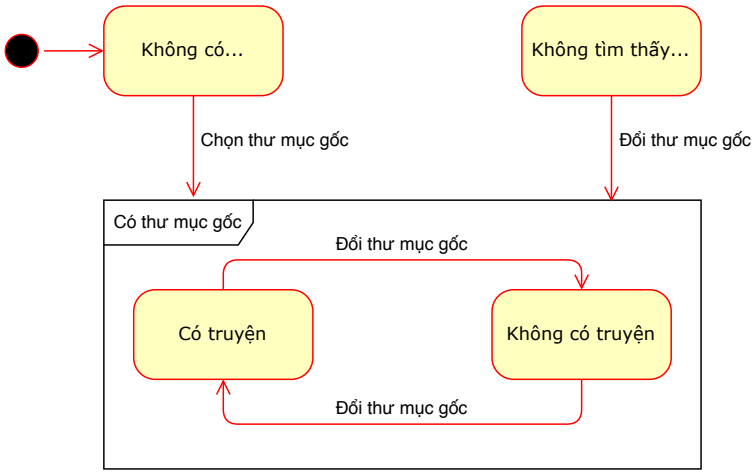
Như đã phân tích ở **??**, Màn hình Thư viện cần hiển thị cả lỗi và gợi ý, bên cạnh việc hiển thị danh sách thư mục và chọn thư mục gốc. Do đó, phần này chia ra làm hai phần con tương ứng.

Chọn thư mục gốc và hiển thị danh sách thư mục

Để chọn thư mục gốc, người dùng ấn nút Đổi thư mục gốc để kích hoạt hộp thoại Chọn thư mục (picker), rồi chọn một thư mục trong đó. Luồng chạy của yacv sẽ được nêu sau, ở mục riêng về Scanner.

Ngoại lệ trong Màn hình Thư viện

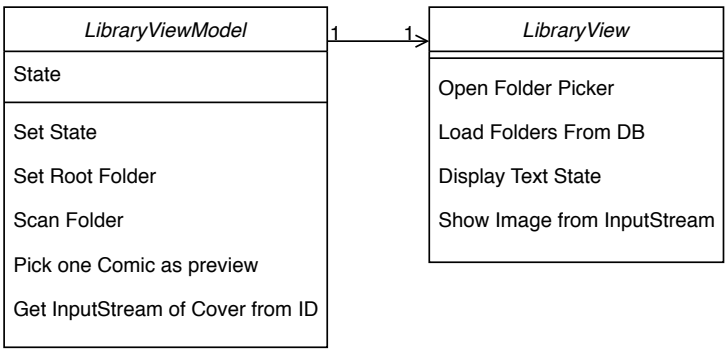
Ngoại lệ ở đây chỉ cả trường hợp không tìm thấy thư mục, lẫn trường hợp không quét được thư mục vì các lí do đã nêu trong **??**. Khi này, ứng dụng hiển thị một hàng chữ để gợi ý về việc nên làm.



Hình 1.6: Trạng thái của Màn hình Thư viện

Hình sau là biểu đồ trạng thái, cũng là mô tả về nội dung gợi ý. Riêng trạng thái "Có truyện" là trạng thái hiển thị danh sách thư mục trong luồng cơ bản đã nêu trên, tức thư mục được hiển thị đầy đủ thay vì chỉ hiện thông báo lỗi.

Tổng hợp lại, ta có biểu đồ lớp của Màn hình Thư viện như sau:



Hình 1.7: Biểu đồ lớp của Màn hình Thư viện

Nhắc lại, cách truy cập để lấy `InputStream` của ảnh từ `ComicID` có thể tham khảo từ Màn hình Đọc truyện.

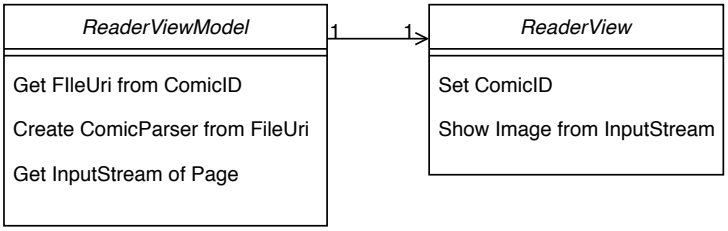
1.2.4 Màn hình Thư mục

Màn hình Thư mục là một trong hai màn hình của ca sử dụng Duyệt truyện, bên cạnh Màn hình Thư viện.

Trong khi phân tích yêu cầu, ta đã phân tích được rằng màn hình hiển thị danh sách truyện - một phần trong ca sử dụng tìm kiếm - phải có giao diện giống Màn hình Thư mục, vì đều hiển thị danh sách truyện. Do đó, hai màn hình này được gộp lại, gọi chung là *Màn hình Danh sách truyện*, và sẽ được mô tả sau.

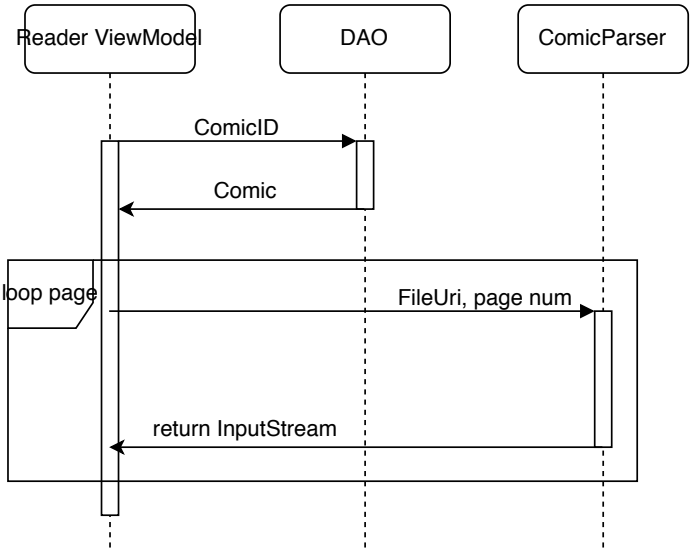
1.2.5 Màn hình Đọc truyện

Để hiển thị các trang truyện, Màn hình Đọc truyện cần nhận `ComicID` (hoặc một đối tượng `Comic` hoàn chỉnh, tuy nhiên cốt yếu vẫn là thông tin `ComicID`) của một tệp truyện, sau đó đưa cho `ComicParser` để lấy luồng đọc cho từng trang truyện. Hình 1.9 là biểu đồ tuần tự của màn hình này.



Hình 1.8: Biểu đồ lớp của Màn hình Đọc truyện

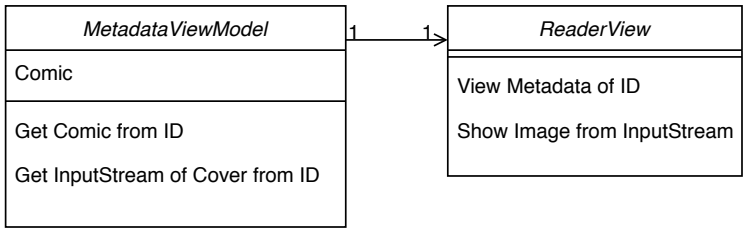
Ta cũng có biểu đồ lớp tương ứng trong Hình 1.8.



Hình 1.9: Biểu đồ tuần tự của Màn hình Đọc truyện

1.2.6 Màn hình Metadata

Màn hình Metadata tuân theo biểu đồ tuần tự đã nêu ở Hình 1.3. Biểu đồ lớp tương ứng của màn hình này như sau:



Hình 1.10: Biểu đồ lớp của Màn hình Metadata

1.2.7 Màn hình Tìm kiếm

Màn hình Tìm kiếm là hai trong ba màn hình của ca sử dụng tìm kiếm truyện, bên cạnh Màn hình Danh sách truyện (là tổng quát hóa của Màn hình Thư mục, đã nhắc ở trên). Màn hình Danh sách truyện sẽ được thiết kế ở ngay mục sau, còn mục này tập trung vào Màn hình Tìm kiếm.

Không khó để thấy thực ra Màn hình Tìm kiếm Tổng quan và Màn hình Tìm kiếm Chi tiết thực ra là một màn hình, về mặt thị giác:

- Điểm giống:
  - Cả hai cùng hiển thị danh sách.
  - Các phần tử cùng loại trong hai màn hình có cách hiển thị giống nhau, chuyển đến các màn hình giống nhau.
- Điểm khác: Danh sách trong Màn hình Tìm kiếm Tổng quan có *thêm*:
  - Hiển thị bìa với một số kết quả
  - Có thanh ngăn cách
  - Có nút "Xem thêm"

Do vậy, nếu thiết kế phù hợp, hoàn toàn có thể gộp hai màn hình này. Thiết kế sau giúp thỏa mãn việc này:

- Màn hình nhận vào một tham số chứa *câu truy vấn*. Tham số này thuộc một trong hai kiểu:
  - `QuerySingleType`: chứa câu truy vấn và *một* bảng để tìm kiếm
  - `QueryMultipleTypes`: chứa câu truy vấn và một *danh sách* bảng để tìm kiếm

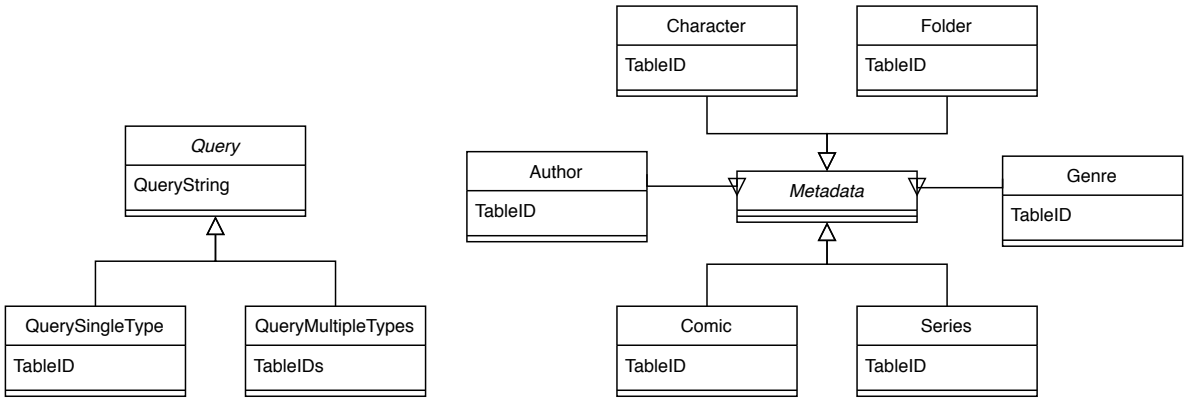
  "Bảng để tìm kiếm" thực ra là một số quy định trước, ví dụ nếu là số `0` thì bảng được tìm là `Comic`,...

- ViewModel tìm kiếm dựa vào tham số truy vấn
  - Nếu tham số là `QuerySingleType`: truy vấn và hiển thị kết quả như thông thường
  - Nếu tham số là `QueryMultipleTypes`: truy vấn các bảng, gộp kết quả lại và thêm kiểu kết quả đặc biệt là *Placeholder* và *SeeMore* vào vị trí phù hợp để hiển thị lần lượt nhóm kết quả và nút "Xem thêm"
- Các kết quả, bao gồm hai dạng kết quả đặc biệt ở trên, cài đặt chung giao diện `Metadata`, để có thể được gộp thành một danh sách

Nói ngắn gọn, hai màn hình cùng hiển thị một danh sách, danh sách này có một số phần tử đánh dấu đặc biệt. Ta dùng lại ví dụ về truy vấn `Watchmen` ở **??** để minh họa:

- Truy vấn `QueryMultipleTypes` được gửi đến Màn hình Tìm kiếm. Câu truy vấn là `Watchmen`, các bảng cần tìm là mọi bảng.
- ViewModel tìm `Watchmen` trong mọi bảng, tìm được:
  - 3 tệp truyện trong bảng `Comic`
  - 2 bộ truyện trong bảng `Series`
- Màn hình hiển thị:
  1. Dòng `Truyện`, rồi 3 tệp truyện (cùng với ảnh bìa)
  2. Dòng `Bộ truyện`, 1 bộ truyện, rồi dòng `Xem thêm`
- Khi ấn vào:
  - Một trong ba tệp truyện: Đưa đến Màn hình Đọc truyện tương ứng.
  - Một bộ truyện: Chuyển đến Màn hình Danh sách truyện, chứa các truyện trong bộ đó.
  - Nút `Xem thêm`: Chuyển đến Màn hình Tìm kiếm, lần này tham số là một `QuerySingleType`, với câu truy vấn là `Watchmen`, còn bảng để tìm là `Series`. Hai bộ truyện kết quả được hiển thị đầy đủ. Chọn một bộ truyện lúc này giống với chọn bộ truyện ở trên (sang Màn hình Danh sách truyện).

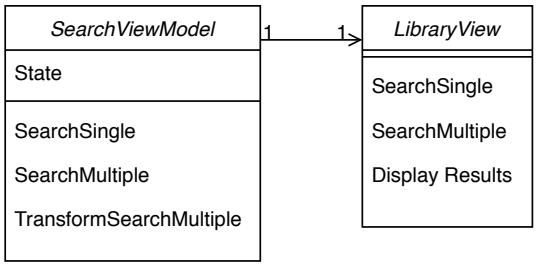
Biểu đồ lớp của các đối tượng liên quan như sau:



(a) Biểu đồ lớp `Query` (b) Biểu đồ lớp `Metadata`

Hình 1.11: Biểu đồ các lớp liên quan đến Màn hình Tìm kiếm

Biểu đồ lớp của bản thân Màn hình Tìm kiếm như sau:



Hình 1.12: Biểu đồ lớp của Màn hình Tìm kiếm

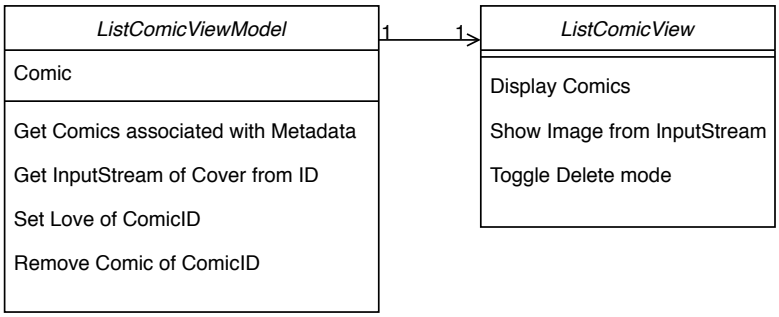
Do mỗi DAO trả kết quả của bảng tương ứng về ở dạng danh sách, nên khi dùng `QueryMultipleTypes`, các danh sách kết quả lẻ này tổng hợp, và thêm hai kiểu kết quả đặc biệt. Hàm `TransformSearchMultiple()` là để "làm phẳng" mảng kết quả hai chiều như trên.

1.2.8 Màn hình Danh sách truyện

Màn hình này là màn hình thứ ba trong chuỗi các màn hình liên quan đến ca sử dụng tìm kiếm, đồng thời đóng vai trò của Màn hình Thư mục (do là phiên bản tổng quát hơn của nó).

Màn hình này nhận vào một tham số kiểu `Metadata` thay vì một `Query`, và trả về danh sách các *tệp truyện* - `Comic` - có liên kết với tham số đầu vào.

Biểu đồ lớp của Màn hình Danh sách truyện như sau:



Hình 1.13: Biểu đồ lớp của Màn hình Danh sách truyện

1.3 Module Parser & Scanner

1.3.1 `ComicParser`

`ComicParser` là một trong các thành phần trung tâm của yacv. Lớp này nhận vào URI trỏ đến một tệp truyện, và đọc nội dung tệp truyện đó ra. Cần gọi đủ tên là `ComicParser`, vì phải phân biệt với hai parser (bộ đọc/giải mã) khác nhưng tích hợp trong nó:

Bảng 1.2: Hai kiểu parser trong `ComicParser`

	Parser cho tệp nén	Parser cho metadata
Đầu vào	Tệp nén	Tệp metadata
Đầu ra	Các tệp con hoặc tương đương (`InputStream`,…)	Đối tượng `Comic`

`ComicParser` được thiết kế theo kiểu "lười", có nghĩa là không có thông tin nào được đọc ra cho đến khi thực sự cần. Lý do vẫn là vấn đề về hiệu năng, vì gần như mọi thao tác đọc trong SAF - hệ thống đọc ghi tệp của Android - đều rất chậm.

Parser cho tệp nén

Parser cho tệp nén hiện gồm một giao diện và hai lớp:

- `ArchiveParser`: giao diện chung cho mọi parser tệp nén
- `CBZParser`: parser riêng cho tệp CBZ
- `ArchiveParserFactory`: giúp khởi tạo các parser

`ArchiveParser` là một giao diện (interface), định nghĩa một số phương thức chung mọi parser cho tệp nén đều phải có. Do hiện tại yacv mới hỗ trợ định dạng CBZ, chỉ có lớp `CBZParser` cài đặt giao diện này.

Trong `ArchiveParser`, có hai phương thức quan trọng:

- `getEntryOffsets()`: Phương thức này trả về một từ điển như sau:
  - Khóa: tên tệp lẻ
  - Giá trị: offset tệp lẻ, tức vị trí tệp lẻ trong tệp nén
- `readEntryAtOffset()`: Phương thức này nhận vào một offset, và trả về `InputStream` tương ứng với tệp lẻ ở offset đó bằng cách "nhảy cóc" đến đúng chỗ và đọc.

`ArchiveParserFactory` là một lớp theo mẫu thiết kế factory, nhận vào URI của tệp truyện và trả về `ArchiveParser` để đọc loại tệp truyện đó (ví dụ, nếu URI có đuôi CBZ thì trả về một đối tượng `CBZParser`). Do `ArchiveParser` cần một số cài đặt khởi tạo riêng, nên mới cần một lớp riêng để tạo parser. Chữ "Factory" thể hiện lớp này sử dụng mẫu thiết kế factory.

Parser cho metadata

yacv hiện hỗ trợ định dạng ComicRack, được giới thiệu chi tiết trong Phụ lục 2. Định dạng này là một tệp tin XML, do đó được đọc đơn giản bằng các thư viện XML sẵn có.

Để mở rộng định dạng tệp đọc, có thể dùng mẫu thiết kế factory như đã dùng với parser cho tệp. Theo cách này, các parser cần có hàm `parse()` trả về một đối tượng `Comic` và nhận hai tham số:

- Nội dung tệp metadata: ở dạng chuỗi thông thường
- Tên tệp metadata: tên tệp giúp phân biệt các định dạng tệp với nhau

`CBZParser` là lớp cài đặt giao diện `ArchiveParser`. Như đã phân tích ở Chương 2, hệ thống đọc ghi tệp SAF của Android chỉ cho phép đọc ghi tuần tự. Việc tạo ra mảng offset không đơn giản, do phần mục lục của tệp ZIP nằm ở cuối, và có nhiều thao tác cần dò ngược từ cuối lên.

Để giải quyết vấn đề danh sách offset, có hai cách đơn giản nhất:

- Chép toàn bộ tệp truyện vào phần bộ nhớ riêng của ứng dụng
  - Ưu: Phần bộ nhớ này vẫn được dùng API File của Java, do đó có thể đọc ghi ngẫu nhiên, cho phép đọc mục lục rất nhanh.

  - Nhược: Tệp truyện rất nặng (vài chục đến vài trăm MB) dẫn đến tốn cả dung lượng đĩa lẫn băng thông đọc/ghi. Ghi xóa liên tục cũng có hại cho bộ nhớ thể rắn của điện thoại.

- Đọc tệp ZIP ở chế độ đọc tuần tự

  - Ưu: Dùng ngay được với cơ chế đọc qua `InputStream` của SAF
  - Nhược: Do không có mục lục, dữ liệu phải được "dò" từ từ để đọc từng tệp lẻ một. Hậu quả là phương pháp này vừa tốn băng thông đọc, vừa tốn CPU để giải nén những tệp không cần thiết.

`CBZParse` giải quyết vấn đề này bằng cách làm giả một luồng nhập ngẫu nhiên, được miêu tả rõ hơn trong Phụ lục 3. Cách làm đó có thể được tóm tắt như sau:

- Hai phần đầu tệp nén được lưu đệm trong RAM, do là hai phần có nhiều truy cập nhất trong khi đọc mục lục
- Các phần còn lại được đọc xuôi khi cần theo luồng nhập `InputStream`, nếu đọc ngược sẽ phải tạo mới luồng nhập

Kết quả là mục lục đọc được mà chỉ cần:

- Trung bình hai lần đọc tuần tự theo `InputStream`
- Không phải ghi ra đĩa
- Không phải giải nén những tệp không cần thiết

Tổng hợp lại `ComicParser`

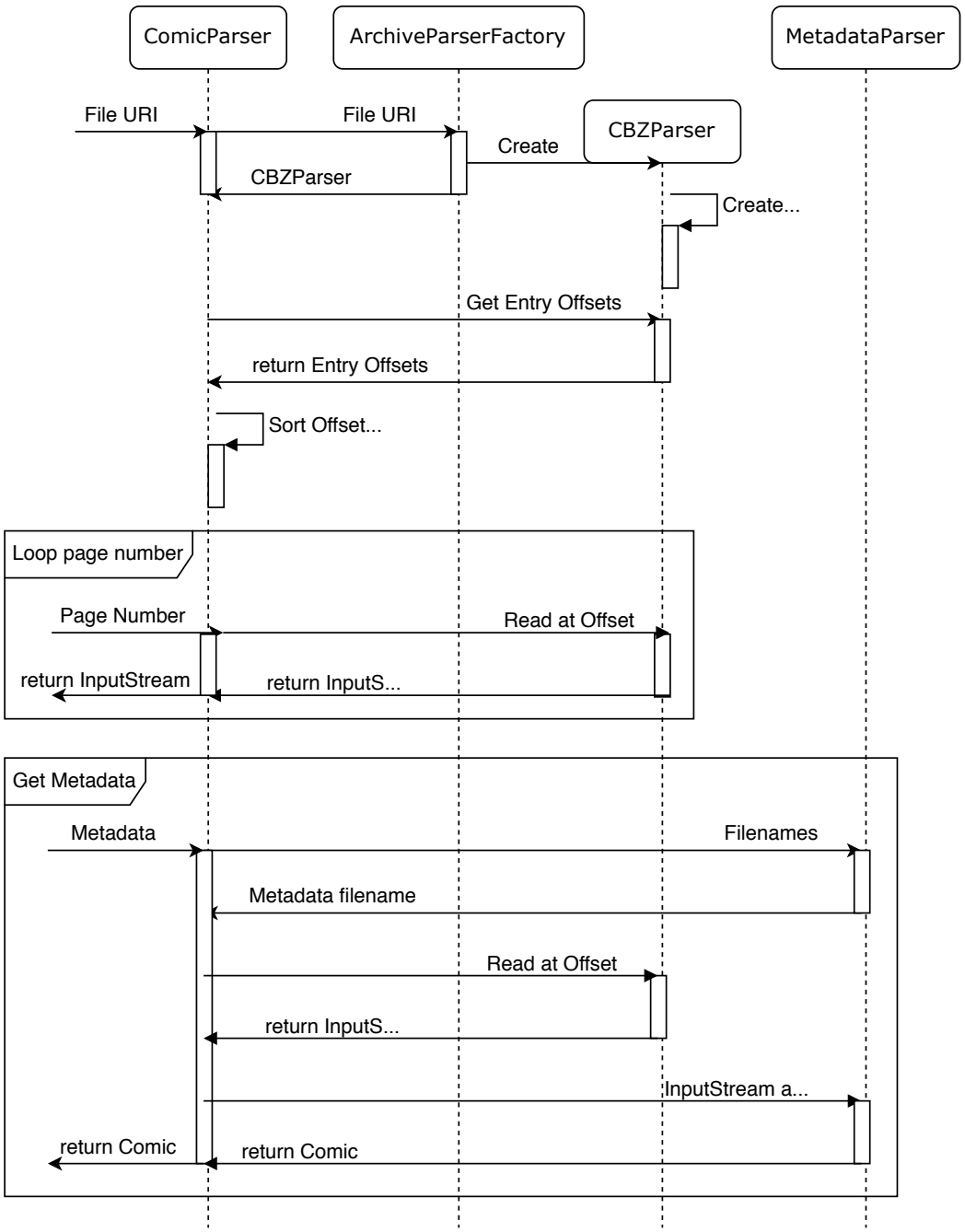
Tương tác trong một ca sử dụng hiển thị của `ComicParser` được mô tả trong Hình 1.14.

Ở đây cần làm rõ chi tiết về việc sắp xếp tệp theo tên. Không có quy chuẩn cho tên trang truyện, tuy nhiên đa số các tệp truyện đặt tên theo định dạng sau:

```
X-Men Vol 40 1.jpg
|     |      |
|     |      Trang truyện số
|     Số Volume, Number,...
Tên tệp truyện
```

Vấn đề với định dạng này xuất hiện khi truyện có nhiều hơn 10 trang. Khi sắp xếp tệp ảnh theo ABC, các trang sẽ có thứ tự như sau:

```
X-Men Vol 40 1.jpg
X-Men Vol 40 10.jpg
X-Men Vol 40 11.jpg
...
X-Men Vol 40 19.jpg
X-Men Vol 40 2.jpg
...
```



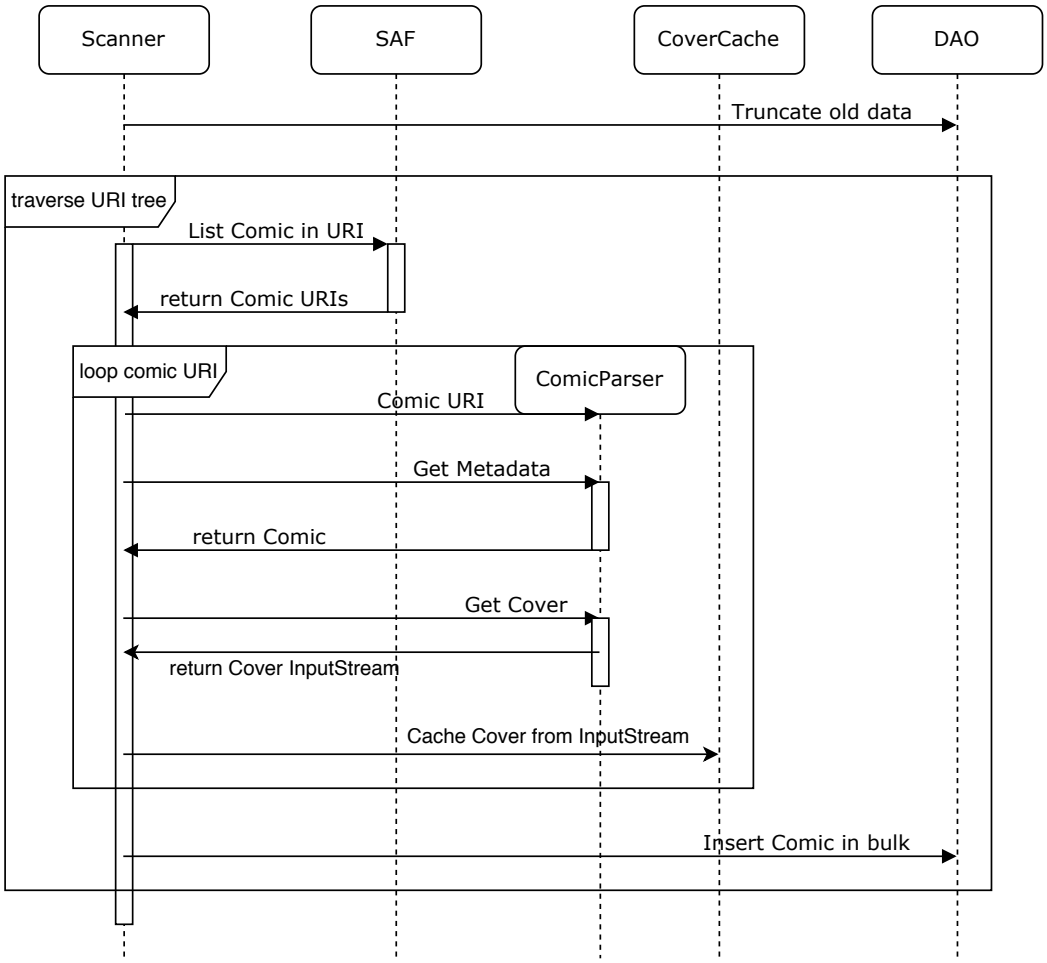
Hình 1.14: `ComicParser` và các thành phần của nó

Ta thấy ngay rằng thứ tự tệp ảnh bị đảo lộn. Để giải quyết vấn đề này, cần viết hàm so sánh riêng cho tên tệp ảnh. Ý tưởng ở đây là gom những kí tự số liên tiếp với nhau thành một "kí tự" rồi mới so sánh. Đoạn mã giả ở trang sau trình bày thuật toán:

```
def compare(str1, str2):
    arrs = []

    for str in [str1, str2]:
        arrtmp = []
        acc = []

        for char in str:
            if is_number(char):
                acc.append(char)
            else:
                if len(acc) != 0:
                    acc = ''.join(acc)
                    acc = to_num(acc)
                    arrtmp.append(acc)
                    acc = 0
                arrtmp.append(to_codepoint(char))

    return compare_left_to_right(arrs[0], arrs[1])
```

Scanner

Đây là lớp phục vụ cho tính năng quét truyện trong yacv. Biểu đồ luồng của ca sử dụng *quét mới* được thể hiện trong Hình 1.15.



Hình 1.15: Biểu đồ tuần tự của ca sử dụng quét mới tệp truyện

Một chi tiết mới trong biểu đồ này là lớp `ImageCache`, sẽ được giới thiệu ở mục về cache ảnh.

Ca sử dụng quét lại cũng có thiết kế gần tương tự, trong đó bỏ bước xóa dữ liệu, và thêm một bước quét cơ sở dữ liệu sau cùng để xóa truyện không còn trong bộ nhớ. Việc cập nhật và thêm truyện mới được thực hiện trong vòng lặp lớn bình thường.

Scanner nhận vào URI của thư mục gốc, rồi lặp qua từng tệp con, cháu,... Nếu đó là tệp truyện, nó gọi 'ComicParser' để lấy metadata, rồi lưu vào cơ sở dữ liệu qua DAO.

Quá trình quét tệp này giống như duyệt cây, do đó có hai cách cơ bản:

- Duyệt theo độ sâu (depth-first search, gọi tắt là DFS)
- Duyệt theo độ rộng (breadth-first search, gọi tắt là BFS)

Trong trường hợp cụ thể này, DFS được chọn. Lý do cho lựa chọn này là DFS có thể phát hiện *thư mục* nhanh hơn nhiều so với BFS. Mỗi khi gặp thư mục, DFS xử lí (đi xuống các thư mục con) ngay, thay vì thêm vào hàng đợi. Do phát hiện được thư mục

nhanh hơn BFS, Màn hình Thư viện, vốn hiển thị danh sách các *thư mục*, cũng hiển thị sớm hơn. Dù thời gian quét tổng thể không thay đổi, người dùng được thấy thư mục sớm hơn giúp tạo cảm giác ứng dụng khá nhanh.

1.4 Module Image Loader

Module Image Loader chịu trách nhiệm trích xuất và lưu đệm (cache) tệp ảnh. Module này gồm hai phần như sau:

1.4.1 Image Extractor

Đây thực chất là một lớp đóng gói quanh `ComicParser`. Bản thân chức năng trích xuất được thực hiện trong `ComicParser` (qua các đối tượng `ArchiveParser`), tuy nhiên Image Extractor thực hiện một số tối ưu giúp việc hiển thị ảnh nhanh chóng hơn.

Ở các mục trước, ta đã tệp lẻ trang truyện không được lưu theo thứ tự đọc. Do đó, cần mục lục tệp nén để có thể nhảy cóc đến trang truyện theo yêu cầu. Tuy nhiên, việc tải ảnh còn có thể tối ưu hơn nữa. Ta xét ví dụ trong Bảng 1.3:

Bảng 1.3: Danh sách các tệp ảnh trong một tệp truyện nén

Thứ tự trong tệp nén	Tên tệp ảnh	Dung lượng
1	`7.jpg`	100KB
2	`6.jpg`	100KB
3	`8.jpg`	100KB
4	`1.jpg`	100KB
5	`3.jpg`	100KB
6	`4.jpg`	100KB
7	`2.jpg`	100KB
8	`5.jpg`	100KB

Hiển nhiên, thứ tự ảnh cần xem là từ tệp `1.jpg` đến tệp `8.jpg`. Ta xem xét và cải tiến các chiến lược tải ảnh qua các tiểu mục tiếp theo.

Tải theo yêu cầu

Ảnh được tải theo đúng yêu cầu ngay lúc đó. Quá trình đọc tệp tin như sau:

- Đọc `1.jpg`: 100KB (bản thân ảnh) + 300KB (do trước khi đọc được `1.jpg` cần đi qua 3 ảnh `7.jpg`, `6.jpg`, `8.jpg`)

- Đọc `2.jpg`: 100KB + 600KB
- …

Vậy để đọc hết truyện, cần đọc 4500KB. Chưa hết, luồng nhập phải được tạo mới mỗi lần đọc (tức bằng số trang truyện), gây ra nhiều overhead. Lý do là bản thân SAF là một Content Provider, do đó nó nằm ở một tiến trình (process) riêng, và cần cơ chế liên lạc xuyên tiến trình (IPC) - vốn đắt đỏ về mặt tính toán trên mọi hệ điều hành - để gọi.

Nguyên nhân của cả hai điểm yếu trên là việc không sử dụng lại luồng nhập (mỗi `InputStream` chỉ đọc một ảnh). Phương án tiếp theo cần xử lí được điểm yếu này.

Tối thiểu hóa số luồng đọc

Để giảm số luồng đọc, ta cần kiểm soát một vài luồng đọc, và phân mỗi trang truyện cho một luồng đọc cụ thể. Để tối thiểu hóa số luồng đọc, ta cần dùng thêm thuật toán *chuỗi con tăng dài nhất* (longest increasing subsequence). Thuật toán cuối cùng thể hiện bằng mã giả như sau:

```
def minStream(pages):
    stream_count = 0
    map_idx_to_stream = []    # Từ điển ánh xạ số trang - số luồng

    while len(pages) != 0:
        # Dây trang tiến lên
        lis = longest_increasing_subsequence(pages)

        for page in lis:
            pages.remove_at(page)    # Bỏ trang trong dây khỏi danh sách
            map_idx_to_stream[page] = stream_count    # Gán số luồng hiện tại

        stream_count += 1

    return map_idx_to_stream
```

Thuật toán nhận vào một mảng `pages` là *thứ tự trong tệp nén* của từng trang truyện. Thuật toán trả về một từ điển như sau:

- Khóa: trang truyện số (bắt đầu từ trang 1)
- Giá trị: số luồng

Ta nhận thấy thuật toán hiển nhiên cho (xấp xỉ) số luồng ít nhất có thể, vì mỗi lần chia trang cho các luồng, ta chọn một bộ trang tăng dần dài nhất lúc đó.

Áp dụng vào ví dụ đang dùng, ta có:

- Luồng 0: đọc trang 1, 3, 4, 5
- Luồng 1: đọc trang 7, 8
- Luồng 2: đọc trang 6
- Luồng 3: đọc trang 2

Vậy để đọc hết truyện, cần đọc 2000KB, và 4 lần tạo mới luồng nhập, khá tốt so với phương pháp đầu.

Tới đây chỉ cần một số chỉnh sửa nhỏ: Giới hạn số luồng nhập. Trường hợp xấu nhất là thứ tự trong tệp ZIP ngược với thứ tự đọc, do đó có nhiều luồng mà mỗi luồng chỉ để đọc một trang truyện. yacv tránh điều này bằng cách giới hạn chỉ có 4 luồng nhập cùng lúc. Những trang không ở trong phạm vi của các luồng này quay về cách đọc nhảy cóc thông thường, không dùng lại luồng nhập chờ sẵn.

1.4.2 Image Cache

Bản thân Image Cache *không* phải là một đoạn mã, đối tượng, mà chỉ là một thư mục. yacv có 2 loại/thư mục cache, cho hai trường hợp hiển thị:

- Cache ảnh bìa
- Cache trang truyện

Lý do cần đến hai bộ cache khác nhau là vì dung lượng lớn của truyện. Các thư viện cache ảnh chọn mốc 100-200MB cho thư mục cache ảnh, đồng thời dùng cố định phương pháp thay thế LRU (nếu cache đầy sẽ xóa ảnh lâu nhất không được dùng). Khi đọc một bộ truyện dung lượng lớn hơn mốc này, toàn bộ ảnh trong cache sẽ sớm bị thay thế bởi ảnh của trang truyện. Sau khi đọc, quay về các màn hình, ảnh bìa dùng để hiển thị trong hai màn hình duyệt truyện bị mất, gây suy giảm trải nghiệm người dùng. Do đó, cần phải tách hai thư mục cache này ra để tránh ảnh hưởng đến cache trang bìa.

Cache trang truyện

Khác với cache trang truyện, cache trang bìa *không* có liên quan tới Image Loader. Thực ra ảnh bìa thì cũng được trích xuất bởi `ComicParser`, tuy nhiên với mỗi tệp truyện chỉ

cần trích ra một ảnh bìa, do đó không cần cơ chế tái sử dụng luồng đọc phức tạp của Image Loader.

Cache trang bìa lại gồm 2 thư mục cache:

1. Cache ảnh hiển thị thực tế

   Cache ảnh được hiển thị bởi `ImageView`. Ảnh này là ảnh bìa đã được cắt (xem phần thiết kế màn hình duyệt truyện) và được thu phóng về chính xác kích cỡ khung nhìn. Dung lượng cache là 100MB.

   Cùng một bìa có hai kiểu hiển thị

   Thư mục cache này được quản lí bởi thư viện hiển thị ảnh. Thư viện đó nhận luồng đọc ảnh (từ `ComicParser`), ghi vào thư mục cache này, và xóa ảnh để giải phóng dung lượng khi cần.

2. Cache ảnh bìa thu nhỏ

   Cache ảnh bìa thu nhỏ, khoảng 50KB mỗi ảnh, chưa bị cắt. Dung lượng cache là 10MB.

Lí do riêng phần bìa cần hai thư mục cache là vì cache ảnh hiển thị thực tế, giống với cache trang truyện, có thể bị xóa bất cứ lúc nào. Do đó cần có một cache rất nhỏ gọn, nằm ở thư mục riêng mà Android không xóa được, chứa ảnh bìa chất lượng thấp, để khi ảnh bìa bị xóa vẫn có một bản bìa nhỏ để hiển thị trong khi chờ ảnh bìa chất lượng cao, có cắt cúp phù hợp được sinh lại.

Trong Hình 1.15, có một lớp `ImageCache` nhận `InputStream` của ảnh bìa và cache ảnh, đó chính là lớp quản lí và sinh ảnh bìa thu nhỏ.