Chương 1

Kiến thức nền tảng

Chương này giới thiệu sơ qua về các nền tảng trong quá trình xây dựng ứng dụng.

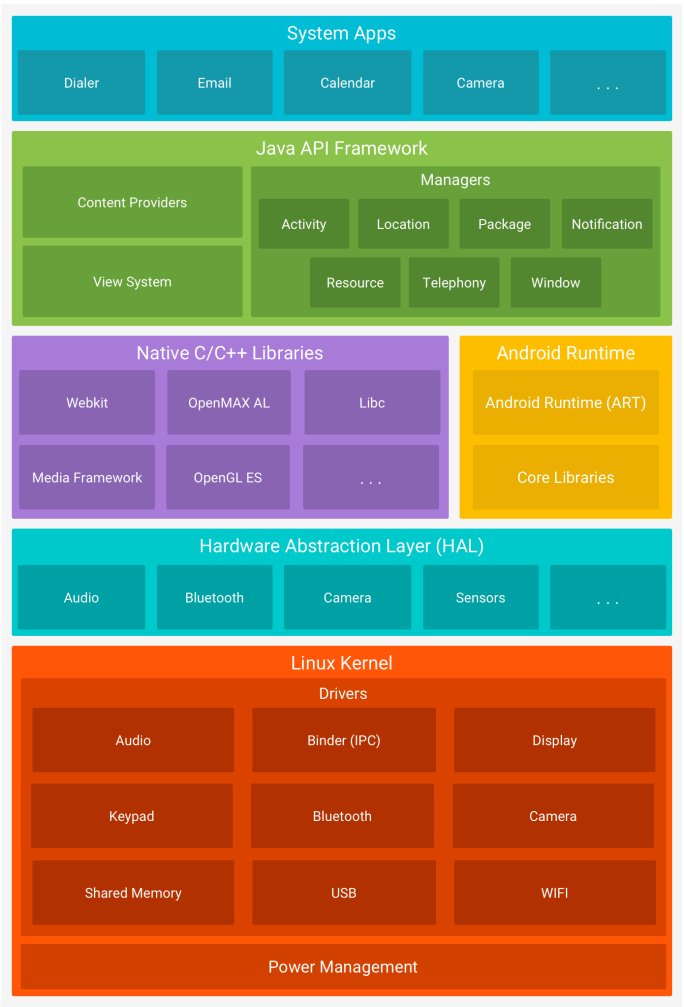
- Hai nền tảng đầu tiên liên quan đến nhau, là tiền đề cho toàn bộ ứng dụng sẽ được giới thiệu trước, gồm hệ điều hành Android và ngôn ngữ lập trình Kotlin.
- Tiếp theo, lựa chọn về kiến trúc tổng quan, liên quan đến giao diện của ứng dụng được trình bày.
- Sau đó, cơ sở dữ liệu một số phần mở rộng của nó dùng trong ứng dụng sẽ được nhắc qua.
- Cuối cùng là thông tin về CBZ - định dạng tệp tin mà ứng dụng đọc.

1.1 Hệ điều hành Android

Android là một hệ điều hành di động do Google phát triển. Android dùng nhân Linux và được thiết kế cho màn hình cảm ứng. Cùng với iOS của Apple, Android trở thành một phần không thể thiếu của cuộc cách mạng di động bắt đầu vào cuối những năm 2000.

Google mua lại phiên bản Android đầu của công ty khởi nghiệp cùng tên vào năm 2005, và phát triển nó từ đó. Ngoài Google, Android còn nhận đóng góp lớn từ cộng đồng, do có mã nguồn mở (phần lớn dùng giấy phép Apache); tên chính thức của dự án là Android Open Source Project. Dù vậy, mọi thiết bị Android thương mại đều có ứng dụng độc quyền. Ví dụ đáng kể là bộ Google Mobile Services, chứa những ứng dụng thiết yếu như trình duyệt Chrome hay chợ Play Store. Về mặt này, Android khá giống Chrome: thành phần cốt lõi kĩ thuật được phát triển theo mô hình mã nguồn mở (AOSP và Chromium), còn thành phần liên quan đến trải nghiệm người dùng được phát triển riêng.

Android được phân lớp như sau:



Hình 1.1: Các phân lớp của Android [5]

- Nhân Linux (Linux Kernel):

  Android dùng nhánh hỗ trợ dài hạn (LTS) của Linux. Khác kiểu phát triển distro trên máy tính (chủ yếu thay đổi ngoài nhân), Google sửa nhân khá nhiều trước khi tích hợp.

- Lớp phần cứng trừu tượng (Hardware Abstraction Layer):

  Tầng này đưa ra giao diện chung cho mỗi kiểu phần cứng (máy ảnh, loa,...), giúp các tầng trên có thể dùng phần cứng mà không quan tâm đến chi tiết riêng.

- Android Runtime (ART):

  Ứng dụng Java cần thêm một ứng dụng để chuyển bytecode thành mã máy. Trên máy bàn, đó là các máy ảo Java (JVM). Trên Android, Android Runtime nhận nhiệm vụ này. Hai máy ảo này khác nhau ở chỗ ART *biên dịch* bytecode thành mã máy (trước khi chạy - AOT), còn JVM *thông dịch* bytecode thành mã máy (trong khi chạy).

- Thư viện C/C++:

  Tầng này phục vụ một số ứng dụng dùng NDK (từ Java gọi C) như trò chơi điện tử.

- Khung phát triển ứng dụng (Java API Framework):

  Mọi ứng dụng Java được viết nhờ sử dụng các thành phần của tầng này thông qua API Java. Tầng này cung cấp toàn bộ tính năng của Android cho lập trình viên, bao gồm các yếu tố cơ bản như giao diện (View System), truy xuất,...

- Ứng dụng hệ thống (System Apps)

  Android đi kèm với một số ứng dụng hệ thống như ứng dụng SMS, trình duyệt,... Google cho phép thay thế đa số các ứng dụng này với ứng dụng bên thứ ba, tuy nhiên có những ngoại lệ như ứng dụng Cài đặt (Settings).

Gần như mọi ứng dụng Android cơ bản đều dùng View System trong tầng Khung phát triển để viết giao diện, và yacv không là ngoại lệ. yacv còn sử dụng thành phần Content Provider, cụ thể là bộ Storage Access Framework, và sẽ được đề cập ở các phần sau.

1.1.1 Android Jetpack

Jetpack là bộ thư viện giúp viết ứng dụng Android nhanh gọn, ít lỗi hơn so với việc tự viết những đoạn mã tương tự. Jetpack gồm hai thành phần chính:

- AndroidX (trước gọi là Thư viện Hỗ trợ - Support Library): đưa *API* của hệ điều hành mới lên máy cũ
- Architecture Component: đưa ra *thư viện* hoàn toàn mới

Việc cập nhật Android rất khó khăn do phải chờ nhà sản xuất tối ưu. Do đó, Jetpack, nhất là AndroidX, rất cần thiết. Chú ý rằng Jetpack chỉ có ích cho lập trình viên (API mới tiện hơn thực ra là wrapper của API sẵn có), chứ không cập nhật tính năng hệ thống.

yacv sử dụng nhiều thành phần của Jetpack, trong đó đáng kể đến ba thư viện sau:

- LiveData: giúp giao diện luôn được cập nhật theo dữ liệu mới nhất
- ViewModel: giúp tách dữ liệu và giao diện
- Room: đơn giản hóa việc lưu dữ liệu trong SQLite (sẽ được giới thiệu ở mục sau)

1.2 Ngôn ngữ lập trình Kotlin

Java là ngôn ngữ lập trình đầu tiên được hỗ trợ trên Android, nhưng không phải duy nhất. Từ 2019, Google khuyên lập trình viên viết ứng dụng trên Kotlin, một ngôn ngữ mới do JetBrains phát triển. Giới thiệu lần đầu vào năm 2011, Kotlin được định hướng trở thành lựa chọn thay thế cho Java. Điều đó thể hiện ở việc Kotlin tương thích hoàn toàn với Java (từ Java gọi được Kotlin và ngược lại), do cùng được biên dịch thành JVM bytecode.

Kotlin hơn Java ở tính ngắn gọn. Do được phát triển mới, Kotlin không cần tương thích ngược, cho phép dùng các cú pháp hiện đại, gọn ghẽ. Do được một công ty phát triển, Kotlin không cần chờ các cuộc họp phức tạp để đạt đồng thuận về tính năng mới. Đồng thời, công ty cũng mở mã nguồn của Kotlin và chương trình dịch, giúp đẩy nhanh quá trình phát triển và tạo thiện cảm cộng đồng cho một ngôn ngữ non trẻ.

Sau đây là tóm tắt một số đặc điểm kĩ thuật của Kotlin:

- Về mô hình, Kotlin hỗ trợ hướng đối tượng như Java, nhưng còn có hướng hàm, thể hiện ở tính năng hàm ẩn danh (lambda), và hàm được coi là first-class.

- Về hệ thống kiểu, Kotlin giống hệt Java:
  - Là kiểu tĩnh (statically typed), tức kiểu được kiểm tra khi biên dịch (thay vì khi chạy, như Python, JavaScript,…)
  - Là kiểu mạnh (strongly typed), tức không cho phép chuyển kiểu ngầm
- Về cú pháp, Kotlin gọn và hiện đại: bỏ dấu `;` cuối dòng, template literal,…
- Về chống lỗi, Kotlin "né" `NullPointerException` do buộc người viết đánh dấu rõ rằng một đối tượng có thể bị `null` bằng hậu tố `?` ở khai báo kiểu. Từ đó, Kotlin biết chính xác đối tượng có thể là `null` hay không, và buộc xử lí nếu có.

Do Google khuyên dùng Kotlin, tôi cho rằng khóa luận này là một cơ hội phù hợp để thử Kotlin thay vì dùng Java quen thuộc, và quyết định chọn viết yacv bằng Kotlin.

1.2.1 Coroutine

Giới thiệu

Một thư viện quan trọng của kotlin là *coroutine*. Coroutine giúp viết ứng dụng có tính tương tranh (concurrency) và bất đồng bộ (asynchronous) một cách đơn giản hơn.

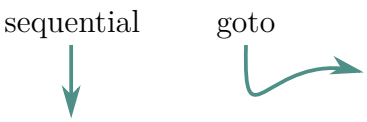
Về cơ bản, coroutine giống với luồng (thread), nhưng nhẹ hơn. Coroutine dùng mô hình *đa nhiệm hợp tác* (cooperative multitasking), khác với luồng hay dùng đa nhiệm ưu tiên (preemptive multitasking). Mấu chốt khác biệt của chúng là đa nhiệm hợp tác có các "điểm dừng" do người viết tạo; khi chạy đến đó, coroutine có thể dừng, nhả CPU cho việc khác, rồi tiếp tục việc đang dở sau. Ngược lại, đa nhiệm ưu tiên có thể buộc luồng đang chạy ngừng lại bất kì lúc nào để ưu tiên chạy luồng khác. Đây cũng là điểm làm coroutine nhẹ hơn: việc chuyển ngữ cảnh (context switching) được kiểm soát và cắt giảm, do chuyển sang một coroutine khác chưa chắc đã chuyển sang một luồng hệ điều hành khác.

Với những điều trên, coroutine chưa làm được nhiều. Roman Elizarov, trưởng dự án Kotlin, hướng coroutine trong Kotlin theo một ý tưởng mới: *tương tranh có cấu trúc* (structured concurrency, từ đây gọi tắt là SC). Ý tưởng này tiếp tục đơn giản hóa việc viết những đoạn mã tương tranh bằng cách áp đặt một số giới hạn, cấu trúc cơ bản. Kết quả là coroutine trong Kotlin hỗ trợ việc xử lí lỗi và ngừng tác vụ bất đồng bộ tốt hơn việc dùng luồng, hay các thư viện tương tranh như RxJava.

Coroutine được dùng để tăng tốc những đoạn mã chạy chậm trong yacv (sẽ được mô tả sau). Ngoài cải thiện hiệu năng, coroutine và SC còn cho phép viết mã ngắn, rõ ràng hơn. Do có tác động lớn, cả hai sẽ được giới thiệu kĩ hơn ở phần này.

Bài học từ quá khứ: lập trình cấu trúc

Để hiểu SC, ta so sánh nó với *lập trình cấu trúc* (structured programming). Để hiểu về lập trình cấu trúc, ta phải tìm về *lập trình phi cấu trúc* (non-structured programming), với đặc điểm là lệnh nhảy `GOTO`. Trong buổi đầu của máy tính, lệnh này được dùng nhiều vì hợp với cách máy tính chạy.

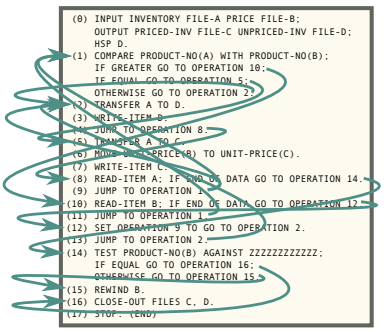


Hình 1.2: Lập trình phi cấu trúc với `GOTO` [6]

Vấn đề của lập trình phi cấu trúc, hay của `GOTO`, gồm:

1. Khó nắm bắt luồng chương trình

   Khi đã chạy `GOTO`, các lệnh phía sau nó không biết khi nào mới chạy, vì chương trình chuyển sang lệnh khác mà không trở lại. Luồng chạy trở thành một đống "mì trộn" như Hình 1.3, thay vì tuần tự từ trên xuống. Tệ hơn, tính trừu tượng bị phá vỡ: khi gọi hàm, thay vì có thể bỏ qua chi tiết bên trong, ta phải biết rõ để xem có lệnh nhảy bất ngờ nào không.



Hình 1.3: Sự lộn xộn của lập trình phi cấu trúc [6]

2. Không cài đặt được các chức năng mới (ngoại lệ, quản lí tài nguyên tự động,...)

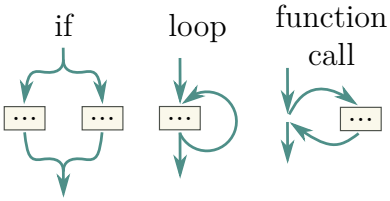
   Xét ví dụ Java sau về quản lí tài nguyên tự động:

```java
try (Scanner scanner = new Scanner(new File("f.txt"))) {
    goto(SOMEWHERE);     // Giả sử Java có GOTO
}
```

   Do không trả lại luồng điều khiển, việc đóng luồng nhập từ tệp cũng không chắc chắn xảy ra, dẫn đến rò rỉ tài nguyên, làm khối lệnh vô dụng.

   Điều gần tương tự cũng khiến việc xử lí ngoại lệ và nhiều tính năng khác trở nên rất khó đạt được, một khi ngôn ngữ cho phép `GOTO`.

Lập trình cấu trúc đơn giản hóa luồng chạy bằng cách giới hạn các lệnh nhảy còn `if`, `for` và gọi hàm. Khác biệt mấu chốt so với `GOTO` là chúng *trả luồng điều khiển* về điểm gọi, thể hiện rõ ở Hình 1.4. Theo định nghĩa, ba lệnh trên giải quyết được hậu quả 1. Đồng thời, hậu quả 2 cũng được giải quyết, do ngôn ngữ đã có cấu trúc (cụ thể là có call stack).



Hình 1.4: Ba cấu trúc cơ bản: rẽ nhánh `if`, lặp `for` và gọi hàm [6]

Ngày nay, ba cấu trúc trên là phần không thể thiếu trong mọi ngôn ngữ lập trình, còn `GOTO` chỉ dùng trong hợp ngữ. Quá khứ cho thấy nếu áp dụng

một số cấu trúc, giới hạn, ta có thể giải quyết vấn đề một cách tinh tế và gọn gàng. Trong trường hợp này, SC có thể loại bỏ một số điểm yếu của các API tương tranh/bất đồng bộ truyền thống, giống cách lập trình cấu trúc đã làm.

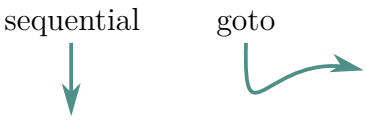
Áp dụng vào hiện tại: tương tranh cấu trúc

Trước hết, ta xem xét hai kiểu API tương tranh hay dùng hiện nay [6]:

Bảng 1.1: Hai kiểu API tương tranh hay dùng [6]

Tên	Giải thích	Ví dụ
Tương tranh	Chạy một hàm theo cách tương tranh với luồng chạy hiện tại	`Thread(func).start() # Python`
Bất đồng bộ	Chạy một hàm khi có sự kiện xảy ra (callback)	`element.onclick = callback; // JS`

Qua Hình 1.5, không khó để thấy rằng mọi vấn đề của lập trình phi cấu trúc đều lặp lại với hai API trên:

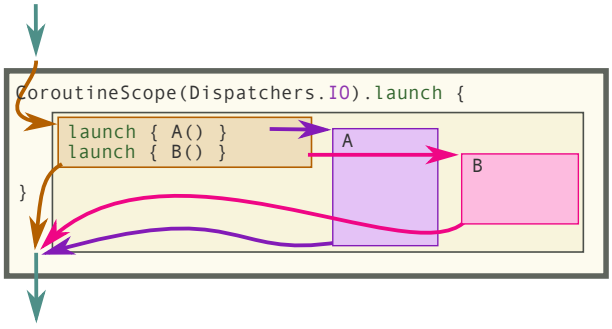


Hình 1.5: Tương tranh phi cấu trúc với `goroutine` - API kiểu tương tranh [6]

- Ta xem lại ví dụ quản lí tài nguyên tự động. Nếu có luồng thực thi khác tương tranh với luồng chính, thì khi luồng chính đóng `Scanner`, có thể luồng kia đang đọc dở. Do đó, tính năng này không thể hoạt động.
- Với tính năng bắt ngoại lệ, nếu có ngoại lệ ở luồng tương tranh, ta cũng không có cách nào để biết, và buộc phải kệ nó.

Trên thực tế, có thể cài đặt chức năng trên với API hiện tại, tuy vậy đó đều là cách xử lí riêng, bất tiện. Ví dụ, ES6 có `catch()` để bắt ngoại lệ trong `Promise` mà không (thể) dùng cấu trúc `try-catch` sẵn có. Với SC, các vấn đề này đều được giải quyết.

Trong Hình 1.6, ta xét một đoạn mã tương tranh dùng coroutine trong Kotlin, tức dùng SC (không phải coroutine trong mọi ngôn ngữ đều dùng mô hình này).



Hình 1.6: Tương tranh cấu trúc dùng coroutine trong Kotlin [6]

Nguyên tắc của SC là: *coroutine cha chờ mọi coroutine con chạy xong*, kể cả khi nó xong trước. Nguyên tắc này đảm bảo rằng khi một hàm kết thúc, không còn tác vụ tương tranh nữa, và điểm gọi nhận lại luồng điều khiển hợp nhất (các mũi tên chụm lại ở góc trái dưới Hình 1.6). Đột nhiên, hai tính năng có vấn đề ở trên lại hoạt động:

- Quản lí tài nguyên tự động: do đảm bảo trả luồng điều khiển, tài nguyên đảm bảo được đóng; do không còn tác vụ con, tài nguyên không bị đọc sau đóng.

- Bắt ngoại lệ: do cấu trúc cha-con (khác với các API hiện tại cho rằng hai tác vụ tương tranh là ngang hàng), coroutine con có thể ném ngoại lệ để coroutine cha bắt.

Chú ý là các API hiện tại không phải không làm được nguyên tắc trên, vấn đề là thực hiện một cách *tự động* và *đảm bảo*. Ví dụ, trong JS, để tuân theo SC, ta phải nhớ `await` với mọi hàm `async`. Do không có ràng buộc chặt chẽ này, các tính năng ngôn ngữ mới cũng khó cài đặt như đã phân tích.

Do trong các ngôn ngữ khác, nguyên tắc trên chỉ là một ca sử dụng, việc ép buộc viết theo ca sử dụng này đòi hỏi lập trình viên thay đổi suy nghĩ về tương tranh. Đổi lại, chương trình trở nên sáng rõ, giống những đoạn mã viết theo kiểu tuần tự truyền thống.

Một khi vấn đề tương tranh được giải quyết hoặc đơn giản hóa, việc song song hóa (paralellization) để tăng tốc ứng dụng chỉ còn là một chi tiết cài đặt.

Tóm tắt

Coroutine với SC là một trong những tính năng quan trọng nhất của Kotlin, giúp tăng tốc những đoạn mã chạy chậm trong yacv. Mấu chốt của SC được tóm gọn trong Hình 1.6. Dù khá mới (Martin Sústrik, tác giả của ZeroMQ, nêu ý tưởng này năm 2016), mô hình này được cải thiện liên tục, có thư viện ở nhiều ngôn ngữ như Java (Loom), Python (Trio),… Điều này cho thấy ý tưởng có ý nghĩa lớn, giúp đơn giản hóa tư duy về tương tranh.

1.3 Mẫu thiết kế MVVM và Kiến trúc khuyên dùng

1.3.1 Mẫu thiết kế MVVM

Cũng như các tác vụ lập trình khác, lập trình giao diện sử dụng nguyên lý Separation of Concern, hiểu đơn giản là chia tách chức năng. Nhiều năm kinh nghiệm cho thấy giao diện nên được chia làm hai phần chính tách biệt nhau:

- Model: dữ liệu để hiển thị (trả lời câu hỏi “cái gì”); liên quan đến đối tượng, mảng,…

- View: cách để hiển thị dữ liệu đó (trả lời câu hỏi “như thế nào”); liên quan đến các yếu tố giao diện như nút, danh sách,…

Sự tách biệt thể hiện ở chỗ Model không được biết View. Khi này, giao diện và nghiệp vụ có thể phát triển khá độc lập với nhau, giúp giảm thời gian phát triển. Ngược lại, View

có biết Model không là tùy vào cách triển khai cụ thể. Có sự bất đối xứng này là do View luôn liên quan chặt chẽ đến framework, khác với Model thường đơn giản.

Khó khăn ở đây là làm sao để kết nối hai thành phần riêng biệt kia. Nhiều mô hình cố giải quyết vấn đề này, tiêu biểu là MVC, MVP và MVVM. Ta lần lượt xem xét chúng để thấy rằng MVVM phù hợp nhất với Android, do đó được chọn làm nền tảng cho Kiến trúc Google khuyên dùng.

MVC: Model - View - Controller

Phương hướng đầu tiên được thử là MVC, vốn phổ biến vào thời điểm Android ra đời. Do được phát minh từ lâu và mỗi framework lại có cách giải thích khác nhau, nên không có một mô hình cụ thể về cách ba thành phần trên tương tác. Tuy vậy, vẫn có vài điểm chung không đổi:

1. Controller nhận thao tác người dùng
2. Sau đó, controller cập nhật Model và View

Ngay ở đây, ta đã thấy điểm yếu của MVC khi áp dụng vào Android. Trong Android, ứng dụng sử dụng Activity và Fragment để viết giao diện. Hai đối tượng này cũng kiêm luôn việc nhận thao tác người dùng, tức chúng là *cả View và Controller*. Mục đích tách ra ba đối tượng do đó không thể làm được.

Hiện nay, MVC trên Android được coi là lỗi thời, không phù hợp.

MVP: Model - View - Presenter

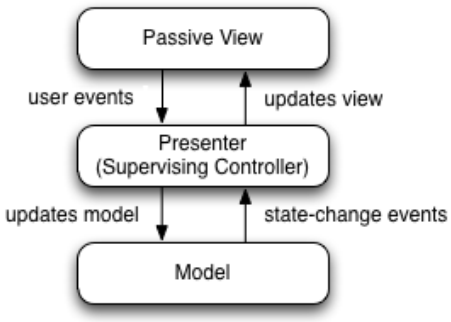
Năm 2012, Robert Martin "Uncle Bob" xuất bản một bài viết nổi tiếng về kiến trúc phần mềm: Clean Architecture. MVP, vốn được phát triển từ lâu, được đông đảo lập trình viên chọn để triển khai Clean Architecture trên Android. Trước khi Google chọn MVVM, đây là hướng đi mới, có kì vọng cao sau nhiều thất bại trong việc đưa MVC vào Android.

Nhiệm vụ của ba thành phần như sau:

- Model: vẫn như trong MVC
- View: hiển thị dữ liệu; nhận tương tác người dùng để chuyển sang Presenter
- Presenter: trung gian giữa Model và View: nhận tương tác từ View, gọi/thay đổi Model, cập nhật View

Ta thấy điểm yếu View-Controller nhập nhằng được khắc phục, khi View kiêm luôn việc nhận tương tác. Đồng thời, Model và View hoàn toàn không biết nhau, đúng theo nguyên lý tách lớp của Clean Architecture.

Ta xét một ứng dụng ToDo đơn giản, trong đó các công việc có thể được đánh dấu đã hoàn thành. Trong ứng dụng, màn hình hiển thị số việc đã và chưa hoàn thành. Trong màn hình đó, tương tác của MVP như sau:



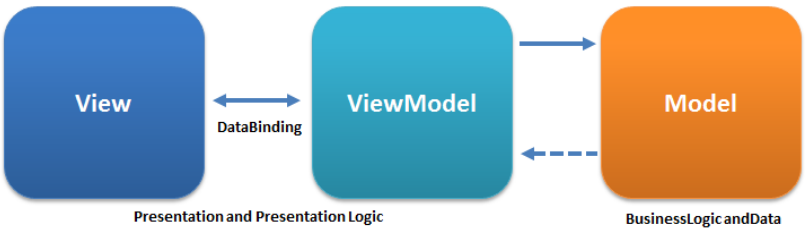
Hình 1.7: Kiến trúc MVP [3]

1. Presenter lấy tất cả công việc trong Model
2. Presenter đếm số việc hoàn thành, chưa hoàn thành
3. Presenter gọi hàm của View, truyền hai số đếm được ở trên vào

Đến đây, thiết kế đã khá hoàn chỉnh và phù hợp với Android.

MVVM: Model - View - View Model

Ta quay về chủ đề chính: MVVM. MVVM giống MVP ở chỗ View Model (từ đây gọi tắt là VM) kết nối View và Model như Presenter. Điểm khác biệt là cách truyền dữ liệu [4]:



Hình 1.8: Kiến trúc MVVM [4]

- Trong MVP, Presenter gọi hàm của View để truyền dữ liệu cho View
- Trong MVVM, VM dùng *data binding* để truyền dữ liệu cho View

Data binding là cơ chế để *tự động* đưa dữ liệu vào thành phần hiển thị. Quay lại ví dụ ToDo ở trên, nếu dùng data binding để "gắn" (bind) danh sách công việc vào View, thì khi danh sách thay đổi, View cũng tự động thay đổi theo.

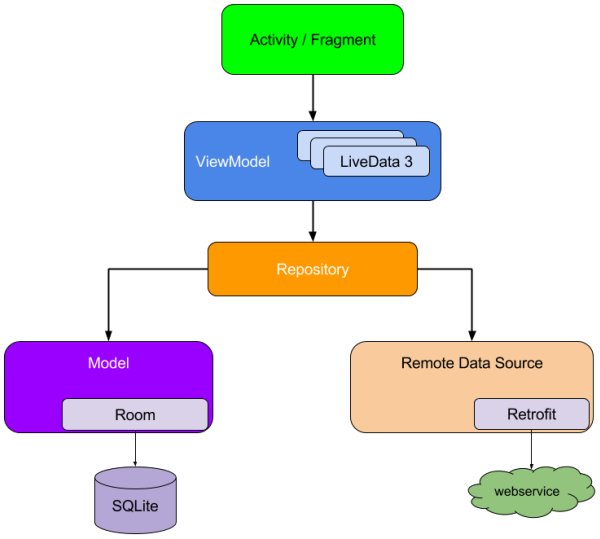
Do dùng data binding thay vì gọi hàm thủ công, VM không cần có tham chiếu tới View, khác với Presenter. Điều này giúp liên kết View - VM thêm *lỏng lẻo* (loose coupling), giúp kiểm thử dễ dàng hơn. Phần còn lại của hai mô hình giống nhau: View cần biết VM để chuyển tương tác; VM cần biết Model để lấy dữ liệu.

Do là mô hình phù hợp nhất trong cả ba với riêng Android, MVVM được chọn làm nền tảng cho Kiến trúc Google khuyên dùng.

1.3.2 Kiến trúc Google khuyên dùng

Kiến trúc Google khuyên dùng có gốc là mô hình MVVM, có dạng như Hình 1.9:

- Repository là Model trong MVVM, giúp VM lấy dữ liệu mà không cần quan tâm dữ liệu lấy từ đâu: cơ sở dữ liệu, gọi API qua mạng,...
- LiveData là cơ chế data binding dùng luồng dữ liệu (stream)
- Activity/Fragment là View



Hình 1.9: Kiến trúc Google khuyên dùng [2]

Repository là một điểm được chi tiết hóa so với MVVM. Google cho rằng một ứng dụng không nên hoàn toàn vô dụng nếu không có mạng. Do đó, cần có hai nguồn dữ liệu: dữ liệu từ máy chủ, và dữ liệu đệm, ngoại tuyến. Khi có nhiều nguồn dữ liệu, mẫu thiết kế Repository là lựa chọn hiển nhiên để trừu tượng hóa chúng.

yacv sử dụng kiến trúc này, dù không có tính năng liên quan đến mạng. Lý do là yacv có tính năng quét truyện hoạt động chậm giống như giao tiếp mạng, nên cần dữ liệu đệm và dữ liệu quét thực tế.

1.4 Cơ sở dữ liệu SQLite

SQLite là một hệ quản trị cơ sở dữ liệu quan hệ (RDBMS). Từ "Lite" trong tên có nghĩa là "nhỏ", thể hiện mục tiêu thiết kế chính của nó là nhỏ gọn. SQLite có thể được nhúng vào phần mềm khác ở dạng thư viện, thay vì là một phần mềm riêng với cấu trúc chủ-khách như MySQL,... Ngay từ những phiên bản đầu, Android đã tích hợp SQLite, giúp lập trình viên không phải nhúng SQLite vào từng ứng dụng.

Để đạt mục tiêu, SQLite chỉ giữ các tính năng SQL cốt lõi (tạo/đọc/sửa/xóa), giao dịch (có ACID), và chỉ tối ưu cho việc truy cập từ một ứng dụng cùng lúc. Các tính năng thường có trong RDBMS cho máy chủ, như nhân bản (replication), chia dữ liệu tự động (sharding), khóa dòng, đọc ghi nhiều luồng cùng lúc,... được loại bỏ. Do đó, với nhu cầu lưu trữ đơn giản, SQLite vừa nhanh vừa gọn.

yacv dùng SQLite để lưu đệm thông tin truyện, tránh quét nhiều lần.

1.4.1 Thư viện ORM Room

Room là một thư viện thuộc Jetpack. Đây có thể xem là một thư viện ORM đơn giản cho SQLite. Room tự động làm nhiều công việc liên quan đến SQL:

- Tạo bảng: Người viết chỉ cần khai báo các đối tượng dữ liệu như một lớp (class) thông thường, rồi đánh dấu với Annotation và interface của Room. Sau đó, Room sinh các bảng tương ứng.
- Truy vấn: Người viết chỉ cần viết lệnh SQL. Sau đó, Room sinh hàm truy vấn tương ứng, chuyển dữ liệu dạng đối tượng sang dạng để lưu trong bảng và ngược lại.
- Kiểm tra truy vấn khi biên dịch: Dò lỗi lệnh SQL mà không cần chờ đến khi chạy.
- Kiểm soát lược đồ (schema): Khi thêm/sửa/xóa bảng/cột, Room luôn phát hiện và ép viết cơ chế cập nhật. Do đó, ứng dụng dùng lược đồ cũ khi được cập nhật sẽ biết cách sửa cơ sở dữ liệu đến phiên bản lược đồ mới.
- Tương thích với LiveData: Giúp View cập nhật theo cơ sở dữ liệu.

1.4.2 Tìm kiếm văn bản

Tìm kiếm văn bản (full-text search, hay gọi tắt là FTS) là một trong số ít các tính năng nâng cao được giữ lại trong SQLite. Cũng như các thư viện tìm kiếm khác, SQLite cài đặt chức năng này bằng chỉ mục đảo (inverted index) - một cấu trúc giống từ điển:

1. Khi dữ liệu văn bản được ghi, nó được tách thành các từ.
2. Các từ được đưa vào chỉ mục đảo: khóa là từ đó, giá trị là khóa đại diện `rowid`.

FTS khác với đánh chỉ mục thường (cũng là chỉ mục đảo nhưng cho kiểu dữ liệu thông thường) ở bước 1: *từng từ* được tách ra, còn chỉ mục thường dùng *cả* văn bản. Do đó, khi tìm từ lẻ, FTS có thể tìm hàng có từ đó rất nhanh. Điểm yếu là ghi chậm, kích cỡ lớn hơn chỉ mục thường. Nếu bản thân dữ liệu văn bản trong cột là một khối, ví dụ như email, chỉ mục thường đã đủ tốt, không cần FTS. Ngoài ra, nếu muốn, ở bước này có thể dùng kĩ thuật rút gọn từ (stemming; chỉ đúng với tiếng Anh), giúp tìm kiếm linh động hơn.

yacv có tính năng tìm kiếm tiêu đề, tên nhân vật,… đều là những câu văn, đoạn văn. Do đó, FTS có vai trò không thể thiếu để tăng tốc tìm kiếm trong ứng dụng.

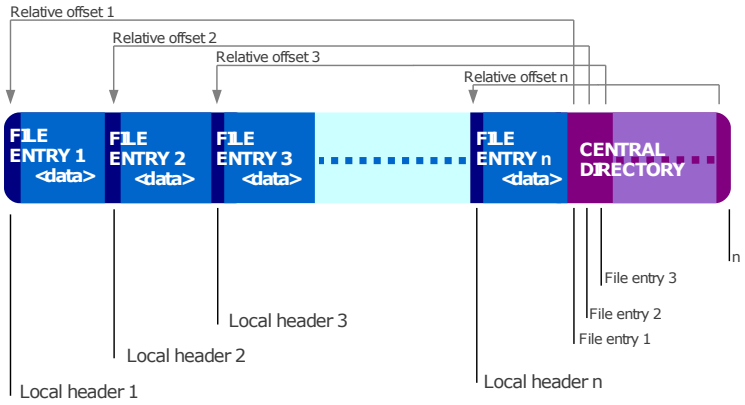
1.5 Định dạng tệp nén ZIP và CBZ

Các tệp truyện mà yacv đọc có định dạng CBZ, bản chất chính là tệp nén ZIP. Do yêu cầu của các phần sau, định dạng tệp ZIP cũng cần được trình bày ở mức cơ bản.

1.5.1 Định dạng tệp nén ZIP

ZIP là một định dạng tệp nén không mất mát (lossless). Được phát minh vào năm 1989 bởi Phil Katz, ZIP đã trở thành định dạng nén tiêu chuẩn, được hỗ trợ trên gần như mọi nền tảng, bao gồm Android.

ZIP thực chất là một định dạng chứa (container), chuyên chứa dữ liệu nén, chứ không phải thuật toán nén; thuật toán nén hay dùng nhất trong ZIP là DEFLATE. Một trong các mục tiêu của ZIP là giúp việc sửa tệp nén (thêm, sửa, xóa tệp con trong tệp ZIP) nhanh nhất có thể. Mục tiêu đó dẫn đến thiết kế sau [1], được thể hiện trong Hình 1.10:



Hình 1.10: Cấu trúc tệp nén ZIP [7]

- Thuật toán nén mỗi tệp gốc thành một tệp nhị phân, ở đây gọi là *tệp nén lẻ* (data trong Hình 1.10). Sau đó, các tệp nén lẻ này được nối thành tệp ZIP cuối cùng.
- Ở đầu mỗi tệp nén lẻ là một header gọi là *File Entry* để lưu thông tin liên quan, trong đó có *tên tệp gốc* và *vị trí bắt đầu* (offset), tức số byte tính từ đầu tệp ZIP đến tệp nén lẻ tương ứng.
- Ở *cuối* tệp ZIP, sau khi đã nối các tệp nén lẻ cùng header lại, các header được gom lại, lưu một lần nữa vào một cấu trúc gọi là *Central Directory*. Có thể so sánh File Entry như các *đề mục*, còn Central Directory là *mục lục*.

Ta phân tích kĩ hơn:

- Do nén riêng từng tệp, có thể dùng thuật toán khác nhau tối ưu với từng tệp.
- Cũng do nén riêng và có mục lục, việc thêm/sửa/xóa (gọi chung là sửa) và đọc có thể được thực hiện với từng tệp lẻ, thay vì phải giải nén, sửa, rồi nén lại toàn bộ.
- Mục lục đặt ở cuối là tối ưu:
  - Giả sử mục lục đặt ở đầu. Khi sửa, toàn bộ các tệp nén lẻ phải di chuyển để tạo chỗ cho mục lục mới (giống như thêm một phần tử vào mảng ở vị trí đầu: toàn bộ các phần tử sau bị đẩy lên để tạo chỗ trống).

– Do mục lục nằm ở cuối tệp ZIP, nên khi sửa chỉ cần đẩy các tệp nén lẻ từ chỗ sửa. Trước đây tối ưu này rất quan trọng. Do đĩa mềm - phương tiện chia sẻ chủ yếu thời đó - có dung lượng nhỏ, tệp ZIP có thể phải cắt ra cho vừa [7]. Thiết kế này cho phép chỉ sửa lại dữ liệu ở một số đĩa, thay vì toàn bộ.

– Hơn nữa, nếu mục lục ở đầu thì ngay trong khi nén, các tệp nén lẻ bị di chuyển như trên do mục lục liên tục được cập nhật.

Tóm lại, cấu trúc tệp nén ZIP cho phép sửa và giải nén từng tệp gốc rất dễ dàng. Nhiều định dạng tệp nén khác (TAR, 7z,…) không có tính năng này, do gộp toàn bộ các tệp vào rồi nén một thể. Trong trường hợp đó, tệp nén được gọi là *đặc* (solid), và rất khó để đọc/sửa tệp lẻ.

1.5.2 Định dạng tệp truyện CBZ

Tệp truyện CBZ chỉ là một tệp nén ZIP thông thường, trong đó có:

- Các tệp ảnh trang truyện: Các tệp này là tệp ảnh JPEG, PNG,… thông thường. Tên tệp được đánh số tăng dần để biểu thị thứ tự trang. Tuy nhiên, không có một cấu trúc/định dạng tên tệp nào được thống nhất.
- (Tùy chọn) Một tệp metadata: Có nhiều định dạng metadata. Hiện nay, yacv chấp nhận định dạng ComicInfo, được trình bày ở **??**. Định dạng này lưu trong một tệp có tên `ComicInfo.xml`.