Coroutines

Một coroutines tương tự như một luồng (theo nghĩa đa luồng): nó là một dòng thực thi, với ngăn xếp riêng, các biến cục bộ riêng và con trỏ hướng dẫn của riêng nó nhưng nó chia sẻ các biến toàn cục và hầu hết là bất kỳ thứ gì khác với các coroutines khác. Sự khác biệt chính giữa các luồng và các coroutines là về mặt khái niệm (hoặc theo nghĩa đen trong một máy đa xử lý), một chương trình có các luồng chạy một số luồng song song.

Mặt khác, Coroutines có tính cộng tác: tại bất kỳ thời điểm nào, một chương trình có coroutines chỉ chạy một trong các coroutines của nó và quy trình đang chạy này chỉ tạm ngừng thực thi khi nó yêu cầu tạm dừng một cách rõ ràng.

Coroutines là một cách mới để viết mã không đồng bộ, không chặn (và nhiều hơn nữa)

Coroutines là những sợi có trọng lượng nhẹ. Một chuỗi nhẹ có nghĩa là nó không ánh xạ trên chuỗi gốc, vì vậy nó không yêu cầu chuyển đổi ngữ cảnh trên bộ xử lý, vì vậy chúng nhanh hơn.

Coroutines và các luồng đều là đa nhiệm, nhưng sự khác biệt là các luồng được quản lý bởi OS và coroutines bởi người dùng.

Các Coroutines cung cấp việc thực thi bán đồng thời các đơn vị chương trình (các coroutines) việc thực thi của chúng được xen kẽ, nhưng không bị chồng chéo:

Diagram

Description automatically generated

Diagram

Description automatically generated

Coroutines cung cấp một cách dễ dàng để quản lý luồng và hoàn toàn phù hợp để thực thi các quy trình trên luồng nền khi cần thiết. Mô hình này cũng cung cấp một cách dễ dàng triển khai đồng thời và đồng thời có cấu trúc. Nó có thể dễ dàng mở rộng và có thể bảo trì, giúp chu kỳ sản xuất và hiệu quả thời gian viết mã của lập trình viên, giải quyết lỗi hoặc tổ chức và làm sạch mã tốt hơn. Coroutines có thể cải thiện đáng kể hiệu suất mã và khả năng sử dụng nó như một luồng.

Rxjava

RxJava, viết tắt của “ReactiveX for Java” là một thư viện JVM để thực hiện các chương trình dựa trên sự kiện và không đồng bộ bằng cách sử dụng các chuỗi có thể quan sát được. Các khối xây dựng chính của nó là ba chữ O, Operator, Observer và Observables. Và sử dụng chúng, chúng tôi thực hiện các tác vụ không đồng bộ trong dự án của mình. Nó làm cho đa luồng rất dễ dàng trong dự án của chúng tôi. Nó giúp chúng tôi quyết định luồng nào chúng tôi muốn chạy tác vụ.

Nó cho phép dễ dàng xử lý bộ nhớ cache mà không cần tạo các lớp bộ nhớ đệm. Một trong những điều quan trọng nhất và là một lợi thế lớn của RxJava là giảm 90% bộ nhớ bị rò rỉ, có thể bằng cách thay thế các cơ chế Android tiêu chuẩn, trở nên thực sự hữu ích khi bạn đang phát triển các ứng dụng Android mà không cần phải cân nhắc nhiều nữa. Nó có thể tối ưu hóa mã để cải thiện khả năng phản hồi của ứng dụng và rất dễ mở rộng quy mô nhưng thực sự khó duy trì do việc sử dụng quá nhiều.

Sự phức tạp của nó là một thách thức và xu hướng sử dụng nó ở khắp mọi nơi làm cho mã trở nên rất phức tạp, khó gỡ lỗi hơn và do đó, làm cho lập trình viên lãng phí nhiều thời gian hơn vào nó để giải quyết lỗi, phát triển mã mới xung quanh đó, hoặc thậm chí để duy trì hoặc viết lại mã , vì nó không trả về lỗi và cách duy nhất để gỡ lỗi là khá nguyên thủy. Điều này xảy ra vì trình gỡ lỗi studio không thể gắn vào chuỗi Rx và bạn cần nhật ký để xem điều gì đang xảy ra. Điểm khác về thư viện này: không thể phát ra null, mặc dù không phải là một điều tuyệt vời, vì có thể nó phát ra null có thể được xử lý dễ dàng với các lỗi.

Sự khác biệt là làm thế nào chúng ta có thể xác định các phương thức không đồng bộ? Với RxJava, chúng ta cần sử dụng Singer class làm kiểu cho giá trị trả về:

|  |
| --- |
| interface StackOverflowService {  @GET("/user")  fun getTopUsers(): Single<List<User>>    @GET("/users/{userId}/badge")  fun getBadges (  @Path("userId") userId: Int  ): Single <List <Badge>>    @GET("/users/{userId}/top-tags")  fun getTags (  @Path("userId") userId: Int  ): Single <List <Tag>>  } |

Điều này cho phép sử dụng phương thức subscribe để chờ phản hồi.

Với coroutines tương tự như trên nhưng sử dụng lớp Deferred:

|  |
| --- |
| interface StackOverflowService {  @GET("/users")  fun getTopUsers(): Deferred<List<User>>    @GET("/users/{userId}/badges")  fun getBadges(  @Path("userId") userId: Int  ): Deferred<List<Badge>>    @GET("/users/{userId}/top-tags")  fun getTags(  @Path("userId") userId: Int  ): Deferred<List<Tag>>  } |

Tuy nhiên, bản nâng cấp gần đây của Kotlin cho phép chúng tôi gọi súpending functions, trên thực tế, điều này làm cho mã trông giống như đồng bộ:

|  |
| --- |
| interface StackOverflowService {  @GET("/users")  suspend fun getTopUsers(): List<User>    @GET("/users/{userId}/badges")  suspend fun getBadges(  @Path("userId") userId: Int  ): List<Badge>    @GET("/users/{userId}/top-tags")  suspend fun getTags(  @Path("userId") userId: Int  ): List<Tag>  } |

Nếu chúng ta muốn sử dụng dịch vụ nói trên, RxJava và coroutines yêu cầu các cách tiếp cận khác nhau. Như đã đề cập ở trên, RxJava yêu cầu sử dụng phương thức subscribe:

|  |
| --- |
| class MyViewModel(  private val service: StackOverflowService  ) : ViewModel() {  private val disposable = CompositeDisposable()  fun load() {  disposable +=  service.getTopUsers()  .subscribeOn(io())  .observeOn(mainThread())  .subscribe(  { users -> updateUi(users) },  { e -> updateUi(e) }  )  }    private fun updateUi(s: Any) {  //...  }    override fun onCleared() {  disposable.clear()  }  } |

Phương thức subscribe chấp nhận hai tham số: tham số đầu tiên là tham số chúng ta muốn thực thi, trong khi tham số thứ hai (e -> updateUi (e)) được sử dụng trong trường hợp có lỗi.

Mặc dù RxJava là đồng bộ theo mặc định (và do đó chúng tôi sử dụng đăng ký), việc sử dụng các hàm coroutines và hàm tạm ngưng cho phép chúng tôi viết mã tự động được hiểu là không đồng bộ, ngay cả khi nó xuất hiện là đồng bộ. Để hiểu sự khác biệt này, hãy so sánh đoạn mã sau với đoạn trước: chúng tương đương về chức năng.

|  |
| --- |
| class MyViewModel(  private val service: StackOverflowService  ) : ViewModel() {  fun load() {  viewModelScope.launch {  try {  val users = service.getTopUsers()  updateUi(users)  } catch (e: Exception) {  updateUi(e)  }  }  }    private fun updateUi(s: Any) {  //...  }  } |

Về hiệu suất, Coroutines hiệu quả hơn RxJava vì nó sử dụng ít tài nguyên hơn để thực hiện cùng một tác vụ trong khi cũng nhanh hơn khi làm như vậy. RxJava sử dụng lượng bộ nhớ lớn hơn và yêu cầu nhiều thời gian CPU hơn, điều này dẫn đến mức tiêu thụ pin cao hơn và người dùng có thể bị gián đoạn giao diện người dùng.

Nếu ứng dụng chạy các cuộc gọi mạng một hoặc hai lần cho mỗi lần tạo hoạt động, hãy sử dụng Coroutines. Nếu ứng dụng cần đơn giản và dễ đọc bởi những người viết mã ít kinh nghiệm, hãy sử dụng Coroutines.

Nếu ứng dụng chạy bất kỳ loại tính năng thời gian thực nào, hãy sử dụng RxJava.

Nếu ứng dụng cần mức độ thao tác dữ liệu cao giữa tìm nạp và phát ra, hãy sử dụng RxJava.

Nếu ứng dụng có kiến trúc dựa trên luồng phản ứng, hãy sử dụng RxJava.

Nếu đó là một dự án đa nền tảng dựa trên Kotlin Native, hãy sử dụng Coroutines.

Nếu kiến trúc dựa trên luồng phản ứng, thì hãy sử dụng RxJava

Có thể sử dụng cả hai cùng nhau, ít nhất là nếu ngôn ngữ lập trình là Kotlin.