Что такое «корутины»?

Корутина — это шаблон параллельного (concurrency) программирования для упрощения написания асинхронного кода.

Фактически корутины — это легковесные потоки.

Почему корутины?

- Они позволяют последовательно писать асинхронны код. Это упрощает чтение кода (заменяет код с callback’ами)

- Они похожи на потоки, но более эффективны. Несколько корутин могут работать в одном потоке.

- Хотя количество потоков ограничено, можете использовать неограниченное количество корутин.

Что это такое suspend-функция?

Корутины основаны на обычных функциях, только для их использования нужно добавить ключевое слово suspend.

Это функции, которые в какой-то момент приостанавливают выполнение корутины, а затем снова возобновляют выполнение кода, когда результат готов.

- Suspend функции могут выполняться только внутри другой корутины или внутри другой suspend функции.

- Внутри suspend функции можно запускать асинхронный код, не блокируя основной поток.

- Suspend - функции могут выполняться либо в фоновом режиме, либо в основном потоке. Ключевое слово suspend не указывает, в каком потоке будет выполняться код.

CoroutineContext

Для того чтобы указать, где запускать код внутри корутины нужен Coroutine Context. Он определяет правила и конфигурацию как и где будет выполняться корутина.

CoroutineContext состоит из элементов:

Job - по сути, это задача, которая выполняется в фоновом режиме.

- Coroutine job созданная с помощью launch. Запускает определенный блок кода и завершается после выполнении этого блока.

- CompletableJob созданная с помощью Job(). Завершается после вызова CompletableJob.complete. Можно использовать либо Job, либо SupervisorJob.

CoroutineDispatcher (по дефолту Dispatchers.Default) - отправляет работу соответствующему потоку. Аналог Rx планировщиков.

Dispatchers.Main - главный поток android (для вызова suspend функций, вызова UI функций, обновлени LiveData)

Dispatchers.IO - работа с БД, сетью, файлами

Dispatchers.Default - ЦП интенсивные задачи (сортировка и др. алгоритмы, парсинг json)

Есть Dispatchers.Unconfined это advanced механизм. Официальная документация не рекомендует использовать его в общем коде.

CoroutineName (опционалтно, по дефолту “coroutine”) - имя корутины, полезно для отладки.

CoroutineExceptionHandler (опционалтно) - обрабатывает неперехваченные исключения.

При создании CoroutineContext, элементы можно объединять с помощью “+”:

val myContext: CoroutineContext = Job() + Dispatchers.Main  
val myScope = CoroutineScope(myContext + CoroutineName("name"))

Как запустить корутину?

Для запуска корутины требуется CoroutineScope, который принимает CoroutineContext в качестве параметра в конструктор. Когда у вас есть CoroutineContext, вы можете использовать методы launch () или async () для запуска новой корутины.

launch () - запускает новую корутину «запустил и забыл» - это означает, что она не вернет результат вызывающей стороне.

async () - запускает новую корутину и позволяет вернуть результат с помощью функции приостановки, называемой await ().

runBlocking: в основном используется для тестирования. Он выполняет фоновые задачи синхронно, избегая завершения теста до завершения фоновой задачи. В проде не рекомендуется использовать.

Каждая создаваемая вами корутина:

- возвращает Job, которое можно использовать для управления его жизненным циклом.

- наследует CoroutineContext от своего родителя (другой корутины или CoroutineScope) и переопределяет параметры контекста, если вы их указали.

Корутины гарантируют:

Когда область действия (scope) отменяется, все ее корутины отменяются.

Когда suspend функция возвращает значение (return), вся ее работа выполнена.

Когда в корутине происходит ошибка, уведомляется ее вызывающая функция или область действия.

CoroutinesScope передается по наследству - новые корутины, запущенные вашими корутины, будут иметь такую же область видимости.

В Android часто имеет смысл связать CoroutineScope с экраном пользователя (обычно путем запуска корутины в ViewModel). Когда пользователь покидает экран, связанный CoroutineScope может отменить всю работу.

Существующие scope’ы:

- Во ViewModels можно использовать extension property «viewModelScope» для запуска корутин. Под капотом используется CoroutineContext с SupervisorJob и Dispatchers.Main.

- Для LiveData — liveData (из ktx библиотеки) можно использовать для вызова suspend функций и возвращать результат в виде LiveData-объекта.

Best practices

1. Каждая suspend функция должна быть main-safe. Можно безопасно вызывать из Dispatchers.Main, даже если они извлекаются из сети и записываются в базу данных.

2. Всегда инжектить Dispatchers в классы. Это делает тестирование проще, т.к. можно заменять их в unit и интеграционных тестах.

3. Слой ViewModel/Presenter должен создавать корутины. UI слой должен быть «тупым» и не вызывать бизнес-логику.

4. Слои расположенные ниже ViewModel/Presenter слоя должны предоставлять suspend функции.

5. Используйте coroutineScope или supervisorScope если нужно создать дополнительные корутины. Тогда вызывающий класс (обычно это ViewModel) будет иметь возможность контролировать выполнение, жизненный цикл и отмену.

Job - контролирует жизненный цикл корутины. По сути, это задача, которая выполняется в фоновом режиме, и ее можно интерпретировать как действие, жизненный цикл которого уничтожается после завершения.

Можно использовать либо Job, либо SupervisorJob. Job отменяет всех своих дочерних элементов, если один из них зафейлился. SupervisorJob - нет, поэтому обычно следует использовать SupervisorJob для области верхнего уровня. Job может находиться в следующих состояниях: Новое, Активное, Завершение, Завершено, Отмена и Отменено. Нельзя поменять состояние, но можно получить получить статус с помощью: isActive, isCancelled и isCompleted.