Advanced Kotlin

Матвей Попов

План

- Многопоточность в глобальном контексте
- Многопоточность в контексте Java
- Многопоточность в контексте Kotlin
- Kotlin Coroutines, live coding
- Сборщики проектов
- Gradle: зачем и почему
- Домашнее задание

Краткий экскурс в многопоточность

Основные определения

Многозадачность

Свойство операционной системы или среды выполнения обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких задач.

Многопоточность

Свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что **процесс**, порождённый в операционной системе, может состоять из нескольких **потоков**, выполняющихся «параллельно», то есть без предписанного порядка во времени

Процессы и потоки

С точки зрения пользователя

Процесс - экземпляр программы во время выполнения

Потоки - ветви кода, выполняющиеся «параллельно»

С точки зрения операционной системы

Процесс - это абстракция, реализованная на уровне операционной системы

Процесс - просто контейнер, в котором находятся ресурсы программы

Процесс содержит:

- Адресное пространство
- Потоки
- Открытые файлы
- Дочерние процессы
- И т.д.

Поток - это абстракция, реализованная на уровне операционной системы

Поток - просто контейнер, в котором хранится информация о состоянии выполнения программы

Поток содержит:

- Счетчик команд
- Регистры
- Стек

И в чем же отличия?

Процесс - заявка на все виды ресурсов

Поток - заявка на процессорное время

Процесс - способ сгруппировать
данные и ресурсы

Поток - это единица
выполнения

А что там есть еще?

- Планирование потоков
- Состояние потоков
- Приоритет потоков
- Системные вызовы
- Режимы доступа

Почему это все важно?

- Понимание работы потоков залог предсказуемой эксплуатации приложения
- Понимание работы потоков залог написания правильного многопоточного кода

НЕ ВСЕГДА

os.Thread == lang.Thread

Многопоточность в Java

Процесс в контексте Java её среда выполнения: JRE

```java
public class Main {

    1 usage
    public static class MyThread extends Thread {

        @Override
        public void run() {
            System.out.println("Hello from MyThread");
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread myThread = new MyThread();

        System.out.println("Hello from Main thread");

        myThread.start();
        myThread.join();

        System.out.println("Buy from Main thread");
    }
}

```

Java multithreading

- java.lang.Thread == os.Thread
- Есть стандартные примитивы синхронизации
- Богатая библиотека java.util.concurrent

Все же круто, зачем нам что-то еще?

А как мы вообще используем эту вашу многопоточность?

98% IO

Operations

И что же это значит?

- Потоки глобально можно разделить на CPU-based и IO-based
- Почти всегда мы выполняем IO операции
- При выполнении IO операции мы большую часть времени ждем

И что в этом плохого?

- Создание потоков - дорого
- Переключение контекста потоков - дорого
- Обслуживание потоков - дополнительная память
- Управление потоков - дорого
- Написание правильного многопоточного кода - очень сложно

И что же с этим делать?

ThreadPool-ы и ExecutorService-ы

- Создаем какой-то контейнер потоков
- Закидываем задачи куда-то “внутрь”
- Не думаем о thread-management
- Можем гибко конфигурировать и использовать разные стратегии

А что делать с блокировкой потоков на IO?

Давайте использовать callback-и!

Callback - функция обратного вызова

Пишите вы код, а потом…

```java
public void getCredentialsWithCallback(String login, Handler<AsyncResult<Credentials>> resultHandler) {
    client.prepare( query: """
        SELECT user_id, password FROM app.credentials
        WHERE provider = :provider AND login = :login
        """, statementHandler -> {
        if (statementHandler.succeeded()) {
            client.execute(statementHandler.result().bind().setString( name: "login", login), selectHandler -> {
                if (selectHandler.succeeded()) {
                    selectHandler.result().one(fetchHandler -> {
                        if (fetchHandler.succeeded()) {
                            Row row = fetchHandler.result();
                            if (row == null) {
                                resultHandler.handle(Future.failedFuture(new NoSuchElementException("Wrong login or password")));
                            } else {
                                resultHandler.handle(Future.succeededFuture(new Credentials(login, row.getString( name: "password"), row.getUUID( name: "user_id"))));
                            }
                        } else {
                            resultHandler.handle(Future.failedFuture(fetchHandler.cause()));
                        }
                    });
                } else {
                    resultHandler.handle(Future.failedFuture(selectHandler.cause()));
                }
            });
        } else {
            resultHandler.handle(Future.failedFuture(statementHandler.cause()));
        }
    });
}
```

А может тогда попробуем реактивное программирование?

Реактивное программирование

Парадигма программирования, ориентированная на потоки данных и распространение изменений.

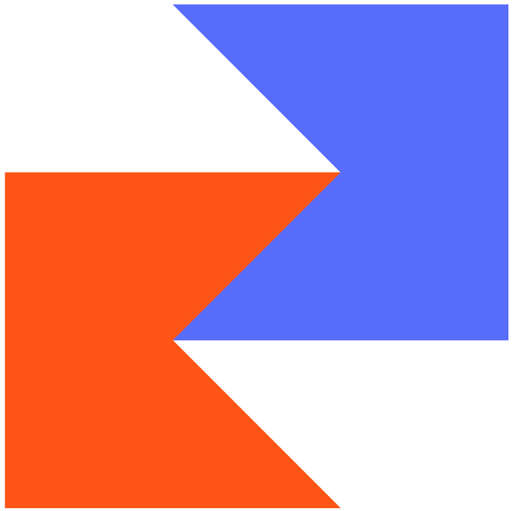
```
public Observable<String> send(String command) {
    return Observable.just(command)
            .doOnNext(cmd -> checkConnection())
            .map(cmd -> cmd.getBytes())
            .map(bytes -> addHeader(bytes))
            .map(bytes -> sendBytes(bytes))
            .timeout(MAX_SEND_TIMEOUT_MS, TimeUnit.MILLISECONDS)
            .map(result -> readAnswer())
            .doOnError(throwable -> disconnect())
            .retry(MAX_RETRY_COUNT)
            .subscribeOn(Schedulers.io());
    }
```

Ну вроде круто, правда же?

- Нужно хорошо знать API библиотеки
- Каждое действие это некоторый оператор
- Достаточно специфичный подход к написанию кода
- Как интегрироваться с другими системами?

- Решение не из “коробки”
- В разы нагружаем GC

И что же нам тогда делать?



Kotlin Coroutines

Многопоточность в Kotlin

- Примитивы синхронизации такие же как и в Java
- Есть java.util.concurrent
- Есть kotlin.concurrent - просто набор Extensions

Из коробки имеем все механизмы Java

А что такое Coroutines?

Сопрограмма(coroutine)

Программный модуль, особым образом организованный для обеспечения взаимодействия с другими модулями по принципу кооперативной многозадачности: модуль приостанавливается в определённой точке, сохраняя полное состояние (включая стек вызовов и счётчик команд), и передаёт управление другому, тот, в свою очередь, выполняет задачу и передаёт управление обратно, сохраняя свои стек и счётчик.

А что такое Kotlin Coroutines?

A **coroutine** is an instance of a suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one.

API Kotlin Coroutines

Что же нам нужно чтобы запустить coroutine?

CoroutineScope

Что это такое и зачем нужен?

- Основной компонент для управления coroutine
- Предоставляет API для запуска и отмены coroutine
- Определяет на каком потоке/потоках должны выполнятся coroutine
- Управляет жизненным циклом coroutine

CoroutineDispatcher

- Определяют контекст выполнения coroutine

- Указывает на каком потоке или потоках будет выполнятся coroutine

Какие бывают?

1. Dispatchers.Main - использует некоторый главный поток

2. Dispatchers.IO - использует ThreadPool оптимизированный для IO операций

3. Dispatchers.Default - дефолтный, оптимизированный для вычислительных задач

4. Dispatchers.Unconfined - не ограничивает выполнение каким-либо потоком

CoroutineBuilders

- Функции, которые используются для создания и запуска coroutine
- Удобный способ для определения асинхронных операций
- Позволяют управлять их поведением и свойствами

Какие бывают?

Launch

```kotlin
fun main(args: Array<String>) = runBlocking(Dispatchers.Default) {
    println("Kotlin coroutines example")

    launch {
        println("First coroutine launch")
        delay( timeMillis: 3000)
        println("First coroutine finish")
    }
    launch {
        println("Second coroutine launch")
        delay( timeMillis: 2000)
        println("Second coroutine finish")
    }
    launch {
        println("Third coroutine launch")
        delay( timeMillis: 3000)
        println("Third coroutine finish")
    }

    println("All coroutines launched")
}

```

Async

```kotlin

fun main(args: Array<String>) = runBlocking(Dispatchers.Default) {
    println("Kotlin coroutines example")

    val firstCoroutine: Deferred<Int> = async {
        println("First coroutine launch")
        delay( timeMillis: 3000)
        println("First coroutine finish")
        return@async 12
    }
    val secondCoroutine: Deferred<Int> = async {
        println("Second coroutine launch")
        delay( timeMillis: 1000)
        println("Second coroutine finish")
        return@async 50
    }
    val result = firstCoroutine.await() + secondCoroutine.await()

    println("All coroutines launched: $result")
}

```

Job

- Некоторый объект задачи
- Позволяет управлять и отменять выполнение задачи
- Могут быть связаны друг с другом

```kotlin
fun main(args: Array<String>) = runBlocking(Dispatchers.Default) {
    println("Kotlin coroutines example")

    val firstJob: Job = launch {
        println("First coroutine launch")
        delay( timeMillis: 3000)
    }
    val secondJob: Job = launch {
        println("Second coroutine launch")
        delay( timeMillis: 2000)
    }

    firstJob.invokeOnCompletion {
        println("First coroutine finish")
    }
    secondJob.invokeOnCompletion {
        println("Second coroutine finish")
    }

    println("All coroutines launched")
}

```

Suspend или функции приостановки

- suspend - ключевое слово в языке Kotlin
- Используется для обозначения функций, которые приостанавливают свое выполнения

```kotlin
suspend fun main(args: Array<String>) {
    val coroutineScope = CoroutineScope(Dispatchers.IO)
    val jobs = mutableListOf<Job>()

    repeat( times: 100) {
        val job = coroutineScope.launch {
            val loadedData = loadInformationFromServer( path: "Path($it)")
            println(loadedData)
        }
        jobs.add(job)
    }

    jobs.forEach { it.join() }
}

suspend fun loadInformationFromServer(path: String): String {
    delay( timeMillis: 3500)
    return "Loaded data from $path"
}

```

Немного live coding

Задача: Сервис по выполнению заказов

Выводы

Kotlin Coroutines - это лаконично и просто!

- Интегрированы с языком
- Эффективное использование ресурсов
- Упрощение асинхронного кода
- Поддержка отмены и обработки ошибок

Материалы

Их очень много!
Полный список будет в отдельном файле

Спасибо за уделенное время!