Введение в Coroutines (сопрограммы)

Эта лаборатория познакомит вас с параллелизмом, который является важнейшим навыком для разработчиков Android, который необходимо понять, чтобы обеспечить удобство работы с пользователем. **Параллелизм** предполагает одновременное выполнение нескольких задач в вашем приложении. Например, ваше приложение может получать данные с веб-сервера или сохранять пользовательские данные на устройстве, одновременно реагируя на события ввода пользователя и соответствующим образом обновляя пользовательский интерфейс.

Чтобы одновременно выполнять работу в вашем приложении, вы будете использовать **сопрограммы** Kotlin . Сопрограммы позволяют приостановить выполнение блока кода, а затем возобновить его позже, чтобы тем временем можно было выполнить другую работу. Сопрограммы упрощают написание **асинхронного** кода, а это означает, что одну задачу не нужно полностью завершать перед запуском следующей задачи, что позволяет одновременно выполнять несколько задач.

В этой лаборатории кода вы познакомитесь с некоторыми базовыми примерами на Kotlin Playground, где вы получите практику работы с сопрограммами, чтобы освоиться с асинхронным программированием.

Что вы узнаете

- Как сопрограммы Kotlin могут упростить асинхронное программирование
- Цель структурированного параллелизма и почему это важно

2. Синхронный код

Простая программа

В **синхронном** коде одновременно выполняется только одна концептуальная задача. Вы можете думать об этом как о последовательном линейном пути. Одна задача должна завершиться полностью, прежде чем будет запущена следующая. Ниже приведен пример синхронного кода.

ткройте Kotlin Playground.

амените этот код следующим кодом для программы, которая показывает прогноз солнечной погоды. В функции `main()` сначала распечатываем текст: `Weather forecast`. Затем распечатываем: `Sunny`.

```
fun main() {
   println("Weather forecast")
   println("Sunny")
}
```

апустите код. Результат выполнения приведенного выше кода должен быть:

```
Weather forecast
Sunny
```

`println()` является синхронным вызовом, поскольку задача вывода текста на выходные данные завершается до того, как выполнение может перейти к следующей строке кода. Поскольку каждый вызов функции `main()` синхронен, `main()` синхронна и вся функция. Является ли функция синхронной или асинхронной, определяется частями, из которых она состоит.

Синхронная функция возвращает значение только тогда, когда ее задача полностью завершена. Таким образом, после выполнения последнего оператора печати `main()` вся работа завершена. Функция `main()` возвращается, и программа завершается.

Добавить задержку

Теперь давайте представим, что для получения прогноза солнечной погоды требуется сетевой запрос к удаленному веб-серверу. Имитируйте сетевой запрос, добавив задержку в код перед выводом сообщения о том, что прогноз погоды солнечный.

начала добавьте `import kotlinx.coroutines.*` в начало кода перед функцией `main()`. Это импортирует функции, которые вы будете использовать, из библиотеки сопрограмм

змените свой код, добавив вызов `delay(1000)`, который задерживает выполнение оставшейся части функции `main()` на `1000` миллисекунды или 1 секунду. Вставьте вызов п

```
import kotlinx.coroutines.*

fun main() {
   println("Weather forecast")
   delay(1000)
   println("Sunny")
}
```

`delay()` - это специальная **функция приостановки,** предоставляемая библиотекой сопрограмм Kotlin. Выполнение функции `main()`приостанавливается в этот момент, а затем возобновляется по истечении заданной продолжительности задержки (в данном случае одна секунда).

Если вы попытаетесь запустить программу на этом этапе, произойдет ошибка компиляции: `Suspend function 'delay' should be called only from a coroutine or another suspend function`.

В целях изучения сопрограмм в Kotlin Playground вы можете обернуть существующий код вызовом функции `runBlocking()` из библиотеки сопрограмм.

`runBlocking()` запускает цикл событий, который может обрабатывать несколько задач одновременно, продолжая каждую задачу с того места, где она была остановлена, когда она готова к возобновлению.

л
еместите существующее содержимое функции `main()`в тело вызова `runBlocking {}`. Тело `runBlocking{}` выполняется в новой сопрограмме.

```
import kotlinx.coroutines.*

fun main() {
   runBlocking {
      println("Weather forecast")
      delay(1000)
      println("Sunny")
   }
}
```

runBlocking() синхронен; он не вернется, пока не будет завершена вся работа в его лямбда-блоке. Это означает, что он будет ждать delay() завершения работы в вызове (пока не пройдет одна секунда), а затем продолжит выполнение оператора печати Sunny. Как только вся работа в runBlocking() функции завершена, функция возвращается, что завершает программу.

апустите программу. Вот результат:

```
Weather forecast
Sunny
```

Вывод такой же, как и раньше. Код по-прежнему синхронен — он выполняется по прямой линии и одновременно выполняет только одно действие. Однако разница сейчас в том, что из-за задержки он длится более длительный период времени.

«co-» в сопрограмме означает кооператив. Код взаимодействует, чтобы совместно использовать базовый цикл событий, когда он приостанавливается в ожидании чего-либо, что позволяет тем временем выполнять другую работу. (Часть «-routine» в «coroutine» означает набор инструкций, подобных функции.) В случае этого примера сопрограмма приостанавливается, когда достигает вызова delay(). Другая работа может быть выполнена в ту секунду, когда сопрограмма приостановлена (хотя в этой программе другой работы нет). По истечении времени задержки сопрограмма возобновляет выполнение и может приступить к печати Sunnyна выходе.

Примечание. Как правило, используйте такую функцию только в учебных целях runBlocking(). main()В коде вашего приложения Android это не требуется, runBlocking()поскольку Android предоставляет цикл событий для вашего приложения для обработки возобновленной работы, когда оно становится готовым. runBlocking()Однако может быть полезен в ваших тестах и может позволить вашему тесту ожидать определенных условий в вашем приложении, прежде чем вызывать тестовые утверждения.

Приостановка функций

Если фактическая логика выполнения сетевого запроса на получение данных о погоде становится более сложной, вы можете выделить эту логику в отдельную функцию. Давайте проведем рефакторинг кода, чтобы увидеть его эффект.

звлеките код, который имитирует сетевой запрос данных о погоде, и переместите его в отдельную функцию с именем printForecast(). Звонок printForecast()с runBlocking()кода.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        printForecast()
    }
}

fun printForecast() {
    delay(1000)
    println("Sunny")
}
```

Если вы запустите программу сейчас, вы увидите ту же ошибку компиляции, которую видели ранее. Функцию приостановки можно вызвать только из сопрограммы или другой функции приостановки, поэтому определите ее `printForecast()`как `suspend`функцию.

> обавьте модификатор `suspend` непосредственно перед ключевым словом `fun` в объявлении функции `printForecast()`, чтобы сделать ее приостанавливающей функцией.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        printForecast()
    }
}

suspend fun printForecast() {
    delay(1000)
    println("Sunny")
}
```

Помните, что `delay()` это приостанавливающая функция, и теперь вы тоже создали `printForecast()` приостанавливающую функцию.

Приостановленная **функция** аналогична обычной функции, но ее можно приостановить и возобновить позже. Для этого функции приостановки можно вызывать только из других функций приостановки, которые делают эту возможность доступной.

Приостановляющая функция может содержать ноль или более точек приостановки.

Точка приостановки — это место внутри функции, где выполнение функции может быть приостановлено. Как только выполнение возобновится, оно продолжится с того места, где оно остановилось в последний раз, и продолжит выполнение остальной части функции.

> отренируйтесь, добавив в свой код еще одну приостанавливающую функцию под объявлением функции `printForecast()`. Вызовите эту новую функцию приостановки получение данных о температуре для прогноза погоды.

Внутри функции также задержите выполнение на `1000` миллисекунды, а затем выведите на выход значение температуры, например, в `30` градусах Цельсия. Вы можете использовать escape-последовательность `"\u00b0"`, чтобы распечатать символ градуса `°`.

```
suspend fun printTemperature() {
    delay(1000)
    println("30\u00b0C")
}
```

> ызовите новую `printTemperature()` функцию из вашего `runBlocking()`кода в `main()`функции. Вот полный код:

```
import kotlinx.coroutines.*

fun main() {
```

```
    runBlocking {
        println("Weather forecast")
        printForecast()
        printTemperature()
    }
}

suspend fun printForecast() {
    delay(1000)
    println("Sunny")
}

suspend fun printTemperature() {
    delay(1000)
    println("30\u00b0C")
}
```

апустите программу. Результат должен быть:

```
Weather forecast
Sunny
30°C
```

В этом коде сопрограмма сначала приостанавливается с задержкой в printForecast()функции приостановки, а затем возобновляется после этой задержки в одну секунду. Текст Sunny выводится на выход. Функция printForecast()возвращается обратно вызывающему объекту.

Далее printTemperature()вызывается функция. Эта сопрограмма приостанавливается, когда достигает delay()вызова, а затем возобновляется через секунду и завершает вывод значения температуры на выход. printTemperature()функция завершила всю работу и возвращается.

В runBlocking()теле больше нет задач для выполнения, поэтому runBlocking()функция возвращается, и программа завершается.

Как упоминалось ранее, runBlocking()он синхронен, и каждый вызов в теле будет вызываться последовательно. Обратите внимание, что хорошо спроектированная функция приостановки возвращает значение только после завершения всей работы. В результате эти функции приостановки выполняются одна за другой.

Необязательно) Если вы хотите узнать, сколько времени потребуется для выполнения этой программы с задержками, вы можете обернуть свой код в вызов, measureTimeMillis()который вернет время в миллисекундах, необходимое для запуска переданного блока кода. Добавьте оператор импорта (import kotlin.system.*), чтобы получить доступ к этой функции. Распечатайте время выполнения и разделите его на 1000.0, чтобы преобразовать миллисекунды в секунды.

```
import kotlin.system.*
import kotlinx.coroutines.*

fun main() {
    val time = measureTimeMillis {
        runBlocking {
            println("Weather forecast")
```

```
        printForecast()
        printTemperature()
    }
  }
  println("Execution time: ${time / 1000.0} seconds")
}
suspend fun printForecast() {
  delay(1000)
  println("Sunny")
}

suspend fun printTemperature() {
  delay(1000)
  println("30\u00b0C")
}
```

Выход:

```
Weather forecast
Sunny
30°C
Execution time: 2.128 seconds
```

Вывод показывает, что выполнение заняло ~ 2,1 секунды. (Точное время выполнения может быть для вас немного другим.) Это кажется разумным, поскольку каждая из приостанавливающих функций имеет задержку в одну секунду.

До сих пор вы видели, что код сопрограммы по умолчанию вызывается последовательно. Если вы хотите, чтобы все выполнялось одновременно, вы должны быть ясными, и вы узнаете, как это сделать, в следующем разделе. Вы будете использовать цикл совместных событий для одновременного выполнения нескольких задач, что ускорит время выполнения программы.

3. Асинхронный код

launch()

Используйте `launch()`функцию из библиотеки сопрограмм, чтобы запустить новую сопрограмму. Чтобы выполнять задачи одновременно, добавьте в свой код несколько функций `launch()`, чтобы одновременно выполнялось несколько сопрограмм.

Сопрограммы в Kotlin следуют ключевой концепции, называемой **структурированный параллелизм**, где ваш код по умолчанию является последовательным и взаимодействует с базовым циклом событий, если вы явно не запрашиваете параллельное выполнение (например, с помощью `launch()`). Предполагается, что если вы вызываете функцию, она должна полностью завершить свою работу к моменту возврата, независимо от того, сколько сопрограмм она могла использовать в деталях своей реализации. Даже если произойдет сбой с исключением, после того как исключение будет создано, в функции больше не будет ожидающих задач. Следовательно, вся работа завершается, как только поток управления возвращается из функции, независимо от того, вызвала ли она исключение или успешно завершила свою работу.

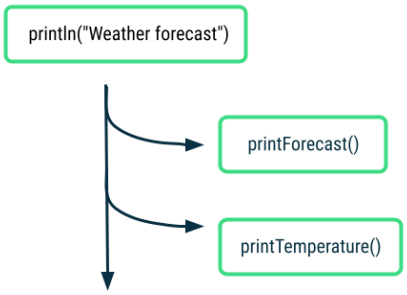
> ачните с кода из предыдущих шагов. Используйте функцию `launch()`для перемещения каждого вызова в собственную сопрограмму `printForecast()`и `printTemperature()`в нее.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        launch {
            printForecast()
        }
        launch {
            printTemperature()
        }
    }
}

suspend fun printForecast() {
    delay(1000)
    println("Sunny")
}

suspend fun printTemperature() {
    delay(1000)
    println("30\u00b0C")
}
```

апустите программу. Вот результат:

```
Weather forecast
Sunny
30°C
```

Результат тот же, но вы, возможно, заметили, что программа запускается быстрее. Раньше вам приходилось ждать полного завершения функции приостановки `printForecast()`, прежде чем переходить к `printTemperature()`. Теперь `printForecast()`и `printTemperature()`могут работать одновременно, поскольку они находятся в разных сопрограммах.



Вызов `launch { printForecast() }`вернется до завершения всей работы `printForecast()`. В этом красота сопрограмм. Вы можете перейти к следующему вызову `launch()`, чтобы запустить следующую сопрограмму. Аналогично, `launch { printTemperature() }`также возвращается еще до того, как вся работа будет завершена.

Необязательно) Если вы хотите увидеть, насколько теперь быстрее работает программа, вы можете добавить `measureTimeMillis()`код для проверки времени выполнения.

```
import kotlin.system.*
import kotlinx.coroutines.*

fun main() {
    val time = measureTimeMillis {
        runBlocking {
            println("Weather forecast")
            launch {
                printForecast()
            }
            launch {
                printTemperature()
            }
        }
    }
    println("Execution time: ${time / 1000.0} seconds")
}

...
```

Выход:

```
Weather forecast
Sunny
30°C
Execution time: 1.122 seconds
```

Вы можете видеть, что время выполнения сократилось с ~ 2,1 секунды до ~ 1,1 секунды, поэтому программа будет выполняться быстрее, если вы добавите параллельные операции! Вы можете удалить этот код измерения времени, прежде чем переходить к следующим шагам.

Как вы думаете, что произойдет, если вы добавите еще один оператор печати после второго вызова `launch()`, до конца кода `runBlocking()`? Где это сообщение появится в выводе?

> змените код `runBlocking()`, добавив дополнительный оператор печати перед концом этого блока.

```
...

fun main() {
    runBlocking {
        println("Weather forecast")
        launch {
            printForecast()
        }
        launch {
            printTemperature()
        }
        println("Have a good day!")
    }
```

```
}

...
```

апустите программу и вот результат:

```
Weather forecast
Have a good day!
Sunny
30°C
```

Из этого вывода вы можете заметить, что после запуска двух новых сопрограмм для `printForecast()`и `printTemperature()`вы можете перейти к следующей инструкции, которая печатает `Have a good day!`. Это демонстрирует принцип «выстрелил и забыл» `launch()`. Вы запускаете новую сопрограмму с помощью `launch()`, и вам не нужно беспокоиться о том, когда ее работа будет завершена.

Позже сопрограммы завершат свою работу и распечатают оставшиеся выходные инструкции. Как только вся работа (включая все сопрограммы) в теле вызова `runBlocking()` завершена, происходит возврат `runBlocking()` и программа завершается.

Теперь вы изменили свой синхронный код на **асинхронный** . Когда асинхронная функция возвращает значение, задача может быть еще не завершена. Это то, что вы видели в случае с `launch()`. Функция вернулась, но ее работа еще не завершена. Используя `launch()`, в вашем коде можно одновременно выполнять несколько задач, что является мощной возможностью для использования в разрабатываемых вами приложениях Android.

асинхронный()

В реальном мире вы не будете знать, сколько времени займут сетевые запросы прогноза и температуры. Если вы хотите отображать единый отчет о погоде после выполнения обеих задач, текущего подхода `launch()`недостаточно. Вот тут-то и появляется `async()`.

Используйте функцию `async()`из библиотеки сопрограмм, если вам важно, когда сопрограмма завершится, и вам нужно возвращаемое значение.

Функция `async()` возвращает объект типа `Deferred`, что похоже на обещание, что результат будет там, когда он будет готов. Вы можете получить доступ к результату объекта, `Deferred` используя `await()`.

1. Сначала измените функции приостановки, чтобы они возвращали `String` вместо печати данных прогноза и температуры. Обновите имена функций с `printForecast()` и `printTemperature()` на `getForecast()`и `getTemperature()`.

```
...

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(1000)
    return "30\u00b0C"
```

```
}
```

змените свой код runBlocking()так, чтобы он использовал async() вместо двух сопрограмм именами forecast и temperature, которые являются объектами Deferred, содержащими результат типа String. (Указание типа не является обязательным из-за вывода типа в Kotlin, но оно включено ниже, чтобы вы могли более четко видеть, что возвращается в результате вызовов async().

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        val forecast: Deferred<String> = async {
            getForecast()
        }
        val temperature: Deferred<String> = async {
            getTemperature()
        }
        ...
    }
}

...
```

озже в сопрограмме, после двух вызовов async(), вы можете получить доступ к результатам этих сопрограмм, вызвав await()объекты Deferred. В этом случае вы можете распечатать значение каждой сопрограммы, используя forecast.await()и temperature.await().

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        val forecast: Deferred<String> = async {
            getForecast()
        }
        val temperature: Deferred<String> = async {
            getTemperature()
        }
        println("${forecast.await()} ${temperature.await()}")
        println("Have a good day!")
    }
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}
```

```
suspend fun getTemperature(): String {
   delay(1000)
   return "30\u00b0C"
}
```

апустите программу, и результат будет такой:

```
Weather forecast
Sunny 30°C
Have a good day!
```

Аккуратный! Вы создали две сопрограммы, которые запускались одновременно для получения прогноза и данных о температуре. Когда каждый из них завершился, они вернули значение. Затем вы объединили два возвращаемых значения в один оператор печати: `Sunny 30°C`.

Примечание. В качестве реального примера `async()`,вы можете посмотреть эту часть приложения Now in Android . В классе SyncWorker вызов `sync()`возвращает логическое значение, если синхронизация с определенной серверной частью прошла успешно. Если какая-либо из операций синхронизации не удалась, приложению необходимо выполнить повторную попытку.

Параллельная декомпозиция

Мы можем продолжить этот пример с погодой и посмотреть, как сопрограммы могут быть полезны при параллельной декомпозиции работы. Параллельная декомпозиция предполагает разбиение проблемы на более мелкие подзадачи, которые можно решать параллельно. Когда результаты подзадач будут готовы, их можно объединить в конечный результат.

В своем коде извлеките логику прогноза погоды из тела `runBlocking()`в одну `getWeatherReport()`функцию, которая возвращает объединенную строку `Sunny 30°C`.

пределите новую функцию приостановки `getWeatherReport()` в своем коде.

становите функцию, равную результату вызова функции `coroutineScope{}` с пустым лямбда-блоком, который в конечном итоге будет содержать логику для получения прогноза погоды.

```
...

suspend fun getWeatherReport() = coroutineScope {

}

...
```

`coroutineScope{}` создает локальную область для этой задачи отчета о погоде. Сопрограммы, запущенные в этой области, группируются внутри этой области, что имеет последствия для отмены и исключений, о которых вы скоро узнаете.

теле `coroutineScope()` создайте две новые сопрограммы `async()`, используемые для получения прогноза и данных о температуре соответственно. Создайте строку отчета о погоде, объединив результаты двух сопрограмм. Сделайте это, вызвав `await()`каждый из `Deferred`объектов, возвращаемых вызовами `async()`. Это гарантирует, что каждая

сопрограмма завершит свою работу и вернет результат, прежде чем мы вернемся из этой функции.

```
...

suspend fun getWeatherReport() = coroutineScope {
   val forecast = async { getForecast() }
   val temperature = async { getTemperature() }
   "${forecast.await()} ${temperature.await()}"
}

...
```

ызовите эту новую функцию `getWeatherReport()`из `runBlocking()`. Вот полный код:

```
import kotlinx.coroutines.*

fun main() {
   runBlocking {
      println("Weather forecast")
      println(getWeatherReport())
      println("Have a good day!")
   }
}

suspend fun getWeatherReport() = coroutineScope {
   val forecast = async { getForecast() }
   val temperature = async { getTemperature() }
   "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
   delay(1000)
   return "Sunny"
}

suspend fun getTemperature(): String {
   delay(1000)
   return "30\u00b0C"
}
```

апустите программу, и вы увидите этот вывод:

```
Weather forecast
Sunny 30°C
Have a good day!
```

Результат тот же, но здесь есть несколько примечательных выводов. Как упоминалось ранее, `coroutineScope()`вернется только после завершения всей своей работы, включая любые запущенные сопрограммы. В этом случае обе сопрограммы `getForecast()`и `getTemperature()`должны завершить работу и вернуть соответствующие результаты. Затем текст `Sunny` и `30°C` объединяются и возвращаются из области видимости. Этот прогноз погоды `Sunny 30°C`

выводится на выход, и вызывающая сторона может перейти к последнему оператору печати `Have a good day!`.

При использовании `coroutineScope()`, даже несмотря на то, что функция внутренне выполняет работу параллельно, вызывающему объекту она кажется синхронной операцией, поскольку `coroutineScope` не вернется, пока вся работа не будет завершена.

Ключевой момент структурированного параллелизма заключается в том, что вы можете объединить несколько одновременных операций в одну синхронную операцию, где параллелизм — это деталь реализации. Единственное требование к вызывающему коду — находиться в функции приостановки или сопрограмме. В остальном структура вызывающего кода не обязана учитывать детали параллелизма.

4. Исключения и отмена

Теперь поговорим о некоторых ситуациях, когда может возникнуть ошибка или какая-то работа может быть отменена.

Введение в исключения

Исключение — это неожиданное событие, которое происходит во время выполнения вашего кода. Вам следует реализовать соответствующие способы обработки этих исключений, чтобы предотвратить сбой вашего приложения и негативное влияние на работу пользователя.

Вот пример программы, которая завершается досрочно с исключением. Программа предназначена для расчета количества пицц, которые сможет съесть каждый человек, путем деления `numberOfPizzas / numberOfPeople`. Допустим, вы случайно забыли установить фактическое значение `numberOfPeople`.

```
fun main() {
   val numberOfPeople = 0
   val numberOfPizzas = 20
   println("Slices per person: ${numberOfPizzas / numberOfPeople}")
}
```

Когда вы запустите программу, она выйдет из строя с арифметическим исключением, поскольку вы не можете разделить число на ноль.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
 at FileKt.main (File.kt:4)
 at FileKt.main (File.kt:-1)
 at jdk.internal.reflect.NativeMethodAccessorImpl.invoke0 (:-2)
```

Эта проблема имеет простое решение: вы можете изменить начальное значение `numberOfPeople` на ненулевое число. Однако по мере того, как ваш код становится более сложным, в некоторых случаях вы не можете предвидеть и предотвратить возникновение всех исключений.

Что происходит, когда одна из ваших сопрограмм завершается с ошибкой с исключением? Чтобы это узнать, измените код из программы погоды.

Исключения **с сопрограммами**

ачните с погодной программы из предыдущего раздела.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        println(getWeatherReport())
        println("Have a good day!")
    }
}

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }
    "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(1000)
    return "30\u00b0C"
}
```

В одной из приостанавливающих функций намеренно создайте исключение, чтобы увидеть, каков будет эффект. Это имитирует непредвиденную ошибку при получении данных с сервера, что вполне вероятно.

> функцию `getTemperature()` добавьте строку кода, вызывающую исключение. Напишите выражение throw, используя ключевое слово `throw` в Kotlin, за которым следует новый экземпляр исключения, простирающегося от Throwable.

Например, вы можете создать `AssertionError` и передать строку сообщения, которая более подробно описывает ошибку: `throw AssertionError("Temperature is invalid")`. Вызов этого исключения останавливает дальнейшее выполнение функции `getTemperature()`.

```
...

suspend fun getTemperature(): String {
    delay(500)
    throw AssertionError("Temperature is invalid")
    return "30\u00b0C"
}
```

Вы также можете изменить задержку `500` для метода `getTemperature()` на миллисекунды , чтобы знать, что исключение произойдет до того, как другая функция `getForecast()`сможет завершить свою работу.

апустите программу, чтобы увидеть результат.

```
Weather forecast
Exception in thread "main" java.lang.AssertionError: Temperature is invalid
 at FileKt.getTemperature (File.kt:24)
 at FileKt$getTemperature$1.invokeSuspend (File.kt:-1)
 at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith (ContinuationImpl.kt:33)
```

Чтобы понять это поведение, вам нужно знать, что между сопрограммами существуют отношения «родитель-потомок». Вы можете запустить сопрограмму (известную как дочерняя) из другой сопрограммы (родительской). Запуская больше сопрограмм из этих сопрограмм, вы можете создать целую иерархию сопрограмм.

Выполнение сопрограммы `getTemperature()`и выполнение сопрограммы `getForecast()`являются дочерними сопрограммами одной и той же родительской сопрограммы. Поведение, которое вы видите с исключениями в сопрограммах, связано со структурированным параллелизмом. Когда одна из дочерних сопрограмм завершается сбоем и вызывает исключение, оно распространяется вверх. Родительская сопрограмма отменяется, что, в свою очередь, отменяет любые другие дочерние сопрограммы (например, `getForecast()`в данном случае работающую сопрограмму). Наконец, ошибка распространяется вверх, и программа аварийно завершает работу с расширением `AssertionError`.

Исключения Try-Catch

Если вы знаете, что определенные части вашего кода могут вызвать исключение, вы можете окружить этот код блоком try-catch . Вы можете перехватить исключение и более изящно обработать его в своем приложении, например, показывая пользователю полезное сообщение об ошибке. Вот фрагмент кода того, как это может выглядеть:

```
try {
   // Some code that may throw an exception
} catch (e: IllegalArgumentException) {
   // Handle exception
}
```

Этот подход также работает для асинхронного кода с сопрограммами. Вы по-прежнему можете использовать выражение try-catch для перехвата и обработки исключений в сопрограммах. Причина в том, что при структурированном параллелизме последовательный код по-прежнему остается синхронным, поэтому блок try-catch будет работать ожидаемым образом.

```
...

fun main() {
   runBlocking {
      ...
      try {
         ...
         throw IllegalArgumentException("No city selected")
         ...
      } catch (e: IllegalArgumentException) {
         println("Caught exception $e")
         // Handle error
```

```
    }
  }
}

...
```

Чтобы удобнее обрабатывать исключения, измените программу погоды, чтобы она перехватывала добавленное ранее исключение и выводила его в выходные данные.

нутри функции `runBlocking()` добавьте блок try-catch вокруг кода, вызывающего сообщение о том, что прогноз погоды недоступен.

```
import kotlinx.coroutines.*

fun main() {
  runBlocking {
    println("Weather forecast")
    try {
      println(getWeatherReport())
    } catch (e: AssertionError) {
      println("Caught exception in runBlocking(): $e")
      println("Report unavailable at this time")
    }
    println("Have a good day!")
  }
}

suspend fun getWeatherReport() = coroutineScope {
  val forecast = async { getForecast() }
  val temperature = async { getTemperature() }
  "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
  delay(1000)
  return "Sunny"
}

suspend fun getTemperature(): String {
  delay(500)
  throw AssertionError("Temperature is invalid")
  return "30\u00b0C"
}
```

апустите программу, и теперь ошибка корректно обрабатывается, и программа может успешно завершить выполнение.

```
Weather forecast
Caught exception in runBlocking(): java.lang.AssertionError: Temperature is invalid
```

```
Report unavailable at this time
Have a good day!
```

Из вывода вы можете видеть, что `getTemperature()` выдает исключение. В теле функции `runBlocking()` вы помещаете вызов `println(getWeatherReport())` в блок try-catch. Вы улавливаете тип исключения, который ожидался (`AssertionError` в случае этого примера). Затем вы печатаете исключение на выходе, за которым `"Caught exception"` следует строка сообщения об ошибке. Чтобы обработать ошибку, вы сообщаете пользователю, что прогноз погоды недоступен, с помощью дополнительного `println()` оператора: `Report unavailable at this time`.

Обратите внимание, что такое поведение означает, что если произойдет сбой при получении температуры, то прогноз погоды вообще не будет (даже если был получен действительный прогноз).

В зависимости от того, как вы хотите, чтобы ваша программа вела себя, существует альтернативный способ обработки исключения в программе погоды.

> еместите обработку ошибок так, чтобы поведение try-catch действительно происходило внутри сопрограммы, запущенной `async()` для получения температуры. Таким образом, в сводке погоды все равно можно будет распечатать прогноз, даже если температура упала. Вот код:

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        println(getWeatherReport())
        println("Have a good day!")
    }
}

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async {
        try {
            getTemperature()
        } catch (e: AssertionError) {
            println("Caught exception $e")
            "{ No temperature found }"
        }
    }

    "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
```

```
    delay(500)
    throw AssertionError("Temperature is invalid")
    return "30\u00b0C"
}
```

апустите программу.

```
Weather forecast
Caught exception java.lang.AssertionError: Temperature is invalid
Sunny { No temperature found }
Have a good day!
```

Из выходных данных вы можете видеть, что вызов `getTemperature()`не удался из-за исключения, но внутренний код `async()`смог перехватить это исключение и корректно обработать его, заставив сопрограмму по-прежнему возвращать сообщение `String`о том, что температура не найдена. Сводку погоды все еще можно распечатать с успешным прогнозом `Sunny`. В прогнозе погоды отсутствует температура, но вместо нее появляется сообщение, поясняющее, что температура не найдена. Это более удобно для пользователя, чем сбой программы из-за ошибки.

Полезный способ подумать об этом подходе к обработке ошибок заключается в том, что он является производителем `async()`, когда с ним запускается сопрограмма. `await()` является потребителем, поскольку он ожидает получения результата от сопрограммы. Производитель делает работу и выдает результат. Потребитель потребляет результат. Если в производителе есть исключение, то потребитель получит это исключение, если оно не будет обработано, и сопрограмма завершится ошибкой. Однако если производитель может перехватить и обработать исключение, то потребитель не увидит это исключение и увидит действительный результат.

Вот код `getWeatherReport()`еще раз для справки:

```
suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async {
        try {
            getTemperature()
        } catch (e: AssertionError) {
            println("Caught exception $e")
            "{ No temperature found }"
        }
    }

    "${forecast.await()} ${temperature.await()}"
}
```

В этом случае производитель (`async()`) смог перехватить и обработать исключение и при этом вернуть `String`результат `"{ No temperature found }"`. Потребитель (`await()`) получает этот `String` результат, и ему даже не нужно знать, что произошло исключение. Это еще один вариант корректной обработки исключения, которое, как вы ожидаете, может произойти в вашем коде.

Примечание. Исключения распространяются по-разному для сопрограмм, начинающихся `launch()`с `async()`. В сопрограмме, запущенной с помощью `launch()`,

исключение генерируется немедленно, поэтому вы можете окружить код блоком try-catch, если ожидается, что он выдаст исключение. См. пример .

Предупреждение. В операторе try-catch в коде сопрограммы избегайте перехвата общих исключений Exception, поскольку они включают в себя очень широкий диапазон исключений. Вы можете случайно обнаружить и подавить ошибку, которая на самом деле является ошибкой, которую следует исправить в вашем коде. Другая важная причина заключается в том, что отмена сопрограмм, которая обсуждается далее в этом разделе, зависит от CancellationException. Поэтому, если вы поймаете какой-либо тип Exception включения CancellationExceptions, не вызывая его повторно, поведение отмены в ваших сопрограммах может вести себя иначе, чем ожидалось. Вместо этого перехватите исключение определенного типа, которое, как вы ожидаете, будет вызвано вашим кодом.

Теперь вы узнали, что исключения распространяются вверх по дереву сопрограмм, если они не обработаны. Также важно быть осторожным, когда исключение распространяется до корня иерархии, что может привести к сбою всего вашего приложения. Дополнительные сведения об обработке исключений см. в блоге «Исключения в сопрограммах» и в статье «Обработка исключений в сопрограммах» .

Отмена

Аналогичная тема с исключениями — отмена сопрограмм. Этот сценарий обычно управляется пользователем, когда событие заставило приложение отменить ранее начатую работу.

Например, предположим, что пользователь выбрал в приложении предпочтение, согласно которому он больше не хочет видеть в приложении значения температуры. Они хотят знать только прогноз погоды (например Sunny), но не точную температуру. Следовательно, отмените сопрограмму, которая в данный момент получает данные о температуре.

начала начните с начального кода ниже (без отмены).

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        println(getWeatherReport())
        println("Have a good day!")
    }
}

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }
    "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
```

```
    delay(1000)
    return "30\u00b0C"
}
```

осле некоторой задержки отмените сопрограмму, которая получала информацию о температуре, чтобы в вашем отчете о погоде отображался только прогноз. Измените возвращаемое значение блока `coroutineScope`, чтобы оно представляло собой только строку прогноза погоды.

```
...

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }

    delay(200)
    temperature.cancel()

    "${forecast.await()}"
}

...
```

апустите программу. Теперь вывод следующий. Сводка погоды состоит только из прогноза погоды `Sunny`, но не из температуры, поскольку эта сопрограмма была отменена.

```
Weather forecast
Sunny
Have a good day!
```

Здесь вы узнали, что сопрограмму можно отменить, но это не повлияет на другие сопрограммы в той же области, а родительская сопрограмма не будет отменена.

Примечание . Вы можете узнать больше об отмене сопрограмм в этом блоге разработчиков Android. Отмена должна быть совместной, поэтому вы должны реализовать свою сопрограмму так, чтобы ее можно было отменить.

В этом разделе вы увидели, как отмены и исключения ведут себя в сопрограммах и как это связано с иерархией сопрограмм. Давайте узнаем больше о формальных концепциях сопрограмм, чтобы вы могли понять, как все важные части собираются вместе.

5. Концепции сопрограмм

При асинхронном или одновременном выполнении работы возникают вопросы, на которые вам необходимо ответить: как будет выполняться работа, как долго должна существовать сопрограмма, что должно произойти, если она будет отменена или завершится с ошибкой и многое другое. Сопрограммы следуют принципу **структурированного параллелизма** , который заставляет вас отвечать на эти вопросы, когда вы используете сопрограммы в своем коде, используя комбинацию механизмов.

Работа

Когда вы запускаете сопрограмму с функцией `launch()`, она возвращает экземпляр `Job`. Задание содержит дескриптор или ссылку на сопрограмму, поэтому вы можете управлять ее жизненным циклом.

```
val job = launch { ... }
```

Примечание. Объект `Deferred`, возвращаемый из сопрограммы, запущенной с помощью функции , также является объектом `async()` и содержит будущий результат сопрограммы `Job`.

Задание можно использовать для управления жизненным циклом или продолжительностью жизни сопрограммы, например, для отмены сопрограммы, если задача вам больше не нужна.

```
job.cancel()
```

С помощью задания вы можете проверить, активно ли оно, отменено или завершено. Задание считается завершенным, если сопрограмма и все запущенные ею сопрограммы завершили всю свою работу. Обратите внимание, что сопрограмма могла завершиться по другой причине, например, из-за отмены или сбоя из-за исключения, но задание все равно считается завершенным на этом этапе.

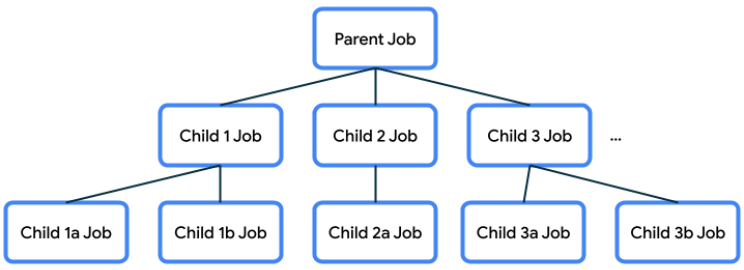
Джобс также отслеживает отношения родитель-потомок между сопрограммами.

Иерархия должностей

Когда сопрограмма запускает другую сопрограмму, задание, которое возвращается из новой сопрограммы, называется дочерним элементом исходного родительского задания.

```
val job = launch {
   ...

   val childJob = launch { ... }

   ...
}
```

Эти родительско-дочерние отношения образуют иерархию должностей, в которой каждое задание может запускать задания и так далее.



Эти отношения между родителями и детьми важны, поскольку они будут диктовать определенное поведение ребенку и родителю, а также другим детям, принадлежащим тому же родителю. Вы видели такое поведение в предыдущих примерах с программой погоды.

- Если родительское задание отменяется, то и его дочерние задания также отменяются.
- Когда дочернее задание отменяется с помощью `job.cancel()`, оно завершается, но не отменяет родительское задание.
- Если задание завершается сбоем из-за исключения, оно отменяет своего родителя с этим исключением. Это называется распространением ошибки вверх (к родителю, родителю родителя и т. д.). .

CoroutineScope

Сопрограммы обычно запускаются в файле `CoroutineScope`. Это гарантирует, что у нас не будет сопрограмм, которые неуправляемы и теряются, что может привести к пустой трате ресурсов.

`launch()`и `async()`являются функциями расширения на `CoroutineScope`. Вызовите `launch()`или `async()`в области, чтобы создать новую сопрограмму в этой области.

А `CoroutineScope`привязан к жизненному циклу, который устанавливает границы того, как долго будут жить сопрограммы в этой области. Если область действия отменяется, то ее задание отменяется, и эта отмена распространяется на его дочерние задания. Если дочернее задание в области завершается сбоем из-за исключения, то другие дочерние задания отменяются, родительское задание отменяется, а исключение повторно генерируется вызывающему объекту.

CoroutineScope в Kotlin Playground

В этой лаборатории кода вы использовали `runBlocking()`файл, который обеспечивает `CoroutineScope`вашу программу. Вы также узнали, как использовать функцию `coroutineScope { }`для создания новой области видимости `getWeatherReport()`.

CoroutineScope в приложениях для Android

Android обеспечивает поддержку области действия сопрограммы в сущностях с четко определенным жизненным циклом, таких как `Activity`(`lifecycleScope`) и `ViewModel`(`viewModelScope`). Сопрограммы, запущенные в этих областях, будут придерживаться жизненного цикла соответствующего объекта, например `Activity` или `ViewModel`.

Например, предположим, что вы запускаете сопрограмму `Activity` с предоставленной областью сопрограммы с именем `lifecycleScope`. Если активность будет уничтожена, `lifecycleScope` будет отменена, и все ее дочерние сопрограммы также будут автоматически отменены. Вам просто нужно решить, является ли сопрограмма, следующая за жизненным циклом, тем поведением `Activity`, которое вам нужно.

Детали реализации CoroutineScope

Если вы проверите исходный код `CoroutineScope.kt` на предмет того, как он реализован в библиотеке сопрограмм Kotlin, вы увидите, что он объявлен как интерфейс `CoroutineScope` и содержит переменную `CoroutineContext`.

Функции `launch()`и `async()`создают новую дочернюю сопрограмму в этой области, и дочерний элемент также наследует контекст из области. Что содержится в контексте? Давайте обсудим это дальше.

СопрограммаКонтекст

`CoroutineContext` предоставляет информацию о контексте, в котором будет работать сопрограмма. `CoroutineContext` по сути, это карта, в которой хранятся элементы, где каждый

элемент имеет уникальный ключ. Это не обязательные поля, но вот несколько примеров того, что может содержаться в контексте:

- name - имя сопрограммы для ее уникальной идентификации.
- job — управляет жизненным циклом сопрограммы
- диспетчер — отправляет работу в соответствующий поток
- обработчик исключений — обрабатывает исключения, создаваемые кодом, выполняемым в сопрограмме

Примечание. Это значения по умолчанию для `CoroutineContext`, которые будут использоваться, если вы не укажете для них значения:

- «сопрограмма» для имени сопрограммы
- нет родительской работы

  для диспетчера сопрограммы
- нет обработчика исключений

Каждый из элементов контекста может быть добавлен вместе с оператором `+`. Например, `CoroutineContext` можно определить следующим образом:

```
Job() + Dispatchers.Main + exceptionHandler
```

Поскольку имя не указано, используется имя сопрограммы по умолчанию.

Внутри сопрограммы, если вы запустите новую сопрограмму, дочерняя сопрограмма унаследует родительскую сопрограмму `CoroutineContext`, но заменит задание специально для только что созданной сопрограммы. Вы также можете переопределить любые элементы, унаследованные от родительского контекста, передав аргументы функциям `launch()` или `async()` для тех частей контекста, которые вы хотите изменить.

```
scope.launch(Dispatchers.Default) {
   ...
}
```

Вы видели упоминание диспетчера несколько раз. Его роль заключается в отправке или назначении работы потоку. Давайте обсудим потоки и диспетчеры более подробно.

Диспетчер

Сопрограммы используют диспетчеры для определения потока, который будет использоваться для их выполнения. Поток может быть запущен, выполнить некоторую работу (выполнить некоторый код), а затем завершиться , когда больше нечего делать.

Когда пользователь запускает ваше приложение, система Android создает новый процесс и единственный поток выполнения вашего приложения, который называется основным **потоком** . Основной поток обрабатывает множество важных операций вашего приложения, включая системные события Android, рисование пользовательского интерфейса на экране, обработку событий пользовательского ввода и многое другое. В результате большая часть кода, который вы пишете для своего приложения, скорее всего, будет выполняться в основном потоке.

Когда речь идет о поточном поведении вашего кода, нужно понимать два термина: **блокирующий** и **неблокирующий** . Обычная функция блокирует вызывающий поток до завершения его работы. Это означает, что он не передает вызывающий поток до тех пор, пока работа не будет завершена, поэтому в это время никакая другая работа не может быть выполнена. И наоборот, неблокирующий код оставляет вызывающий поток до тех пор, пока не будет выполнено определенное условие, поэтому вы тем временем можете выполнять другую работу. Вы можете использовать асинхронную функцию для выполнения неблокирующей работы, поскольку она возвращается до завершения своей работы.

В случае приложений Android вам следует вызывать код блокировки в основном потоке только в том случае, если он будет выполняться достаточно быстро. Цель состоит в том, чтобы основной поток оставался разблокированным, чтобы он мог немедленно выполнить работу при возникновении нового события. Этот основной поток является **потоком пользовательского интерфейса** для ваших действий и отвечает за рисование пользовательского интерфейса и события, связанные с пользовательским интерфейсом. Когда на экране происходят изменения, пользовательский интерфейс необходимо перерисовывать. Для чего-то вроде анимации на экране пользовательский интерфейс необходимо часто перерисовывать, чтобы он выглядел как плавный переход. Если основному потоку необходимо выполнить длительный блок работы, экран не будет обновляться так часто, и пользователь увидит резкий переход (известный как «зависание»), либо приложение может зависнуть или ответить медленно.

Следовательно, нам необходимо переместить все долго выполняющиеся рабочие элементы из основного потока и обрабатывать их в другом потоке. Ваше приложение начинается с одного основного потока, но вы можете создать несколько потоков для выполнения дополнительной работы. Эти дополнительные потоки можно назвать рабочими потоками. Для длительной задачи совершенно нормально заблокировать рабочий поток на долгое время, потому что в это время основной поток разблокируется и может активно отвечать пользователю.

Kotlin предоставляет несколько встроенных диспетчеров:

- используйте этот диспетчер для запуска сопрограммы в основном потоке Android. Этот диспетчер используется в основном для обработки обновлений и взаимодействий пользовательского интерфейса, а также для выполнения быстрой работы.
- этот диспетчер оптимизирован для выполнения дискового или сетевого ввода-вывода вне основного потока. Например, читать или записывать файлы, а также выполнять любые сетевые операции.
- это диспетчер по умолчанию, используемый при вызове `launch()`и `async()`, когда в их контексте не указан диспетчер. Вы можете использовать этот диспетчер для выполнения ресурсоемкой работы вне основного потока. Например, обработка файла растрового изображения.

Примечание. Также
есть `Executor.asCoroutineDispatcher()`расширения `Handler.asCoroutineDispatcher()`, если вам нужно сделать `CoroutineDispatcher`из них `Handler`или `Executor`те, которые у вас уже есть.

Попробуйте следующий пример в Kotlin Playground, чтобы лучше понять диспетчеры сопрограмм.

амените любой код, который у вас есть в Kotlin Playground, на следующий код:

```
import kotlinx.coroutines.*

fun main() {
   runBlocking {
      launch {
         delay(1000)
```

```
        println("10 results found.")
    }
    println("Loading...")
  }
}
```

еперь оберните содержимое запущенной сопрограммы вызовом `withContext()` для изменения того `CoroutineContext`, внутри чего выполняется сопрограмма, и, в частности, переопределите диспетчер. Переключитесь на использование программе `Dispatchers.Main`).

```
...

fun main() {
  runBlocking {
    launch {
      withContext(Dispatchers.Default) {
        delay(1000)
        println("10 results found.")
      }
    }
    println("Loading...")
  }
}
```

Переключение диспетчеров возможно, поскольку `withContext()`само по себе является приостанавливающей функцией. Он выполняет предоставленный блок кода, используя новый файл `CoroutineContext`. Новый контекст берется из контекста родительского задания (внешнего `launch()`блока), за исключением того, что он переопределяет диспетчер, используемый в родительском контексте, указанным здесь: `Dispatchers.Default`. Вот как мы можем перейти от выполнения работы `Dispatchers.Main` к использованию `Dispatchers.Default`.

апустите программу. Результат должен быть:

```
Loading...
10 results found.
```

о

```
import kotlinx.coroutines.*

fun main() {
  runBlocking {
    println("${Thread.currentThread().name} - runBlocking function")
      launch {
      println("${Thread.currentThread().name} - launch function")
      withContext(Dispatchers.Default) {
        println("${Thread.currentThread().name} - withContext function")
        delay(1000)
        println("10 results found.")
      }
```

ы

п
е
ч

```
        println("${Thread.currentThread().name} - end of launch function")
    }
    println("Loading...")
  }
}
```

апустите программу. Результат должен быть:

```
main @coroutine#1 - runBlocking function
Loading...
main @coroutine#2 - launch function
DefaultDispatcher-worker-1 @coroutine#2 - withContext function
10 results found.
main @coroutine#2 - end of launch function
```

Из этого вывода видно, что большая часть кода выполняется в сопрограммах основного потока. Однако для части вашего кода в блоке `withContext(Dispatchers.Default)`, которая выполняется в сопрограмме в рабочем потоке диспетчера по умолчанию (который не является основным потоком). Обратите внимание, что после `withContext()`возврата сопрограмма возвращается к работе в основном потоке (о чем свидетельствует оператор вывода: `main @coroutine#2 - end of launch function`). Этот пример демонстрирует, что вы можете переключить диспетчер, изменив контекст, используемый для сопрограммы.

Если у вас есть сопрограммы, запущенные в основном потоке, и вы хотите переместить определенные операции из основного потока, вы можете использовать `withContext` для переключения диспетчера, используемого для этой работы. Выберите подходящий из доступных диспетчеров: `Main`, `Default`, и `IO`в зависимости от типа операции. Затем эту работу можно поручить потоку (или группе потоков, называемой пулом потоков), предназначенному для этой цели. Сопрограммы могут приостанавливать себя, и диспетчер также влияет на то, как они возобновляются.

Обратите внимание, что при работе с популярными библиотеками, такими как Room и Retrofit, вам, возможно, не придется самостоятельно переключать диспетчер явно, если код библиотеки уже выполняет эту работу, используя альтернативный диспетчер сопрограмм, например. `Dispatchers.IO`. В таких случаях функции `suspend`, которые обнаруживают эти библиотеки, могут уже быть **безопасными для основного потока** и могут быть вызваны из сопрограммы, работающей в основном потоке. Сама библиотека будет обрабатывать переключение диспетчера на тот, который использует рабочие потоки.

Теперь у вас есть общий обзор важных частей сопрограмм и той роли, которую `CoroutineScope`, `CoroutineContext`, `CoroutineDispatcher`и `Jobs` играют в формировании жизненного цикла и поведения сопрограммы.

6. Заключение

Вы узнали, что сопрограммы очень полезны, поскольку их выполнение можно приостановить, освободив базовый поток для выполнения другой работы, а затем возобновить работу сопрограммы позже. Это позволяет вам выполнять параллельные операции в вашем коде.

Код сопрограммы в Котлине следует принципу структурированного параллелизма. По умолчанию он последовательный, поэтому вам нужно явно указать, хотите ли вы параллелизм (например, используя `launch()`или `async()`). Благодаря структурированному параллелизму вы можете объединить несколько одновременных операций в одну синхронную операцию, где параллелизм — это деталь реализации. Единственное требование к вызывающему коду — находиться в функции приостановки или сопрограмме. В остальном структура вызывающего

кода не обязана учитывать детали параллелизма. Это упрощает чтение и анализ вашего асинхронного кода.

Структурированный параллелизм отслеживает каждую запущенную сопрограмму в вашем приложении и гарантирует, что они не потеряются. Сопрограммы могут иметь иерархию: задачи могут запускать подзадачи, которые, в свою очередь, могут запускать подзадачи. Джобс поддерживает отношения «родитель-потомок» между сопрограммами и позволяет вам контролировать жизненный цикл сопрограммы.

Запуск, завершение, отмена и сбой — это четыре общие операции при выполнении сопрограммы. Чтобы упростить поддержку параллельных программ, структурированный параллелизм определяет принципы, которые формируют основу для управления общими операциями в иерархии:

> **апуск:** запуск сопрограммы в области, которая имеет определенную границу продолжительности ее жизни.
>
> **авершение:** задание не будет завершено, пока не будут завершены его дочерние задания.
>
> **тмена:** эта операция должна распространяться вниз. Когда сопрограмма отменяется, дочерние сопрограммы также должны быть отменены.
>
> **еудача:** Эта операция должна распространяться вверх. Когда сопрограмма генерирует исключение, родительская программа отменит все свои дочерние элементы, отменит себя и распространит исключение до своего родителя. Это продолжается до тех пор, пока сбой не будет обнаружен и обработан. Это гарантирует, что любые ошибки в коде будут правильно сообщены и никогда не будут потеряны.

Благодаря практической практике работы с сопрограммами и пониманию концепций, лежащих в основе сопрограмм, вы теперь лучше подготовлены к написанию параллельного кода в своем приложении для Android. Используя сопрограммы для асинхронного программирования, ваш код становится проще читать и анализировать, он становится более устойчивым в ситуациях отмены и исключений, а также обеспечивает более оптимальную и отзывчивую работу для конечных пользователей.

Краткое содержание

- Сопрограммы позволяют вам писать долго выполняющийся код, который работает одновременно, не изучая новый стиль программирования. Выполнение сопрограммы по задумке является последовательным.
- Сопрограммы следуют принципу структурированного параллелизма, который помогает гарантировать, что работа не будет потеряна и не привязана к области с определенной границей продолжительности ее жизни. Ваш код по умолчанию является последовательным и взаимодействует с базовым циклом событий, если только вы явно не запрашиваете параллельное выполнение (например, с помощью `launch()`или `async()`). Предполагается, что если вы вызываете функцию, она должна полностью завершить свою работу (если не произойдет сбой с исключением) к моменту возврата, независимо от того, сколько сопрограмм она могла использовать в деталях своей реализации.
- Модификатор `suspend` используется для обозначения функции, выполнение которой можно приостановить и возобновить позже.
- Функция `suspend` может быть вызвана только из другой приостанавливающей функции или из сопрограммы.
- Вы можете запустить новую сопрограмму, используя функции расширения `launch()` или `async()` в `CoroutineScope`.
- Джобс играет важную роль в обеспечении структурированного параллелизма, управляя жизненным циклом сопрограмм и поддерживая отношения родитель-потомок.

контролирует время жизни сопрограмм через свое задание и рекурсивно обеспечивает отмену и другие правила для своих дочерних элементов и их дочерних элементов.

определяет поведение сопрограммы и может включать ссылки на задание и диспетчер сопрограммы.

- Сопрограммы используют `CoroutineDispatcher` для определения потоков, которые будут использоваться для их выполнения.

Узнать больше

- Сопрограммы Kotlin на Android
- Дополнительные ресурсы для сопрограмм и потоков Kotlin
- Руководство по сопрограммам
- Контекст сопрограммы и диспетчеры
- Отмены и исключения в сопрограммах
- Сопрограммы на Android
- Сопрограммы Котлина 101
- KotlinConf 2019: Сопрограммы! Надо поймать их всех!