# Корутины и асинхронность

При первом запуске приложения Android среда выполнения создает один поток, в котором по умолчанию будут выполняться все компоненты приложения. Этот поток обычно называют основным потоком. Основная роль основного потока — обработка пользовательского интерфейса, его событий и взаимодействие с элементами интерфейса. Любые дополнительные компоненты, запускаемые в приложении, по умолчанию также будут выполняться в основном потоке.

Любой код внутри приложения, который выполняет какую-нибудь трудоемкую задачу с использованием основного потока, приведет к тому, что все приложение заблокируется до тех пор, пока задача не будет завершена. То есть грубо говоря приложение зависает. Это далеко нелучшая ситуация, которая отрицательно влияет на впечатление пользователя от взаимодействия с приложением. И чтобы избавиться от подобных ситуаций мы можем использовать в приложении корутины. Более подробно про корутины можно посмотреть на этом же сайте metanit.com в основном руководстве по языку Kotlin в главе Корутины. В данной же статье мы вкратце затронем все основные моменты, связанные с корутинами.

Корутины представляют блоки кода, которые выполняются асинхронно и не блокируют поток, из которого они запускаются. Корутины не эквивалентны потокам, хотя и используют их. Проблема с потоками заключается в том, что они представляют собой ограниченный ресурс и являются дорогостоящими с точки зрения возможностей процессора и накладных расходов системы. В фоновом режиме выполняется большой объем работы по созданию, планированию и уничтожению потока. Хотя современные процессоры могут выполнять большое количество потоков, фактическое количество потоков, которые могут выполняться параллельно в любой момент времени, ограничено количеством ядер процессора (хотя новые процессоры имеют 8 или более ядер, большинство устройств Android содержат процессоры с 4 ядрами). А когда требуется больше потоков, чем имеется ядер в процессоре, система должна выполнить планирование потоков, чтобы решить, как выполнение этих потоков будет распределяться между доступными ядрами.

Чтобы избежать подобных накладных расходов, вместо того, чтобы запускать новый поток для каждой корутины и затем уничтожать его при выходе из корутины, Kotlin поддерживает пул активных потоков и управляет тем, как корутины назначаются этим потокам. Когда активная корутина приостанавливается, она сохраняется средой выполнения Kotlin, и вместо нее начинает работу другая корутина. Когда корутина возобновляет свою работу, она просто восстанавливается в существующем незанятом потоке из пула и продолжает выполнение до тех пор, пока не завершится или не будет приостановлена. При таком подходе ограниченное количество потоков эффективно используется для выполнения асинхронных задач с возможностью выполнения большого количества одновременных задач без ухудшения производительности, которое могло бы произойти при использовании стандартной многопоточности.

Все корутины должны выполняться в определенной области (scope), что позволяет управлять ими как группами, а не как отдельными корутинами. Назначив для некоторой области группу корутин, эту группу можно массово отменить, когда ее корутины больше не нужны.

Kotlin и Android предоставляют некоторые встроенные области корутин (scope), а также возможность создавать собственные области корутин с помощью класса CoroutineScope. Основные встроенные

области корутин:

- GlobalScope: используется для запуска корутин верхнего уровня, которые привязаны ко всему жизненному циклу приложения. Поскольку корутины в этой области могут продолжать работать, когда в этом нет необходимости (например, когда объект Activity завершает свою работу), использование этой области не рекомендуется для использования в приложениях Android.
- ViewModelScope: применяется при использовании компонента ViewModel архитектуры Jetpack. Корутины, запущенные в этой области из объекта ViewModel, автоматически завершаются системой при уничтожении соответствующего объекта ViewModel.
- LifecycleScope: создается для каждого компонента с жизненным циклом и удаляется, когда уничтожается соответствующий компонент.

Самый простой способ определить область действия корутин внутри компонента состоит в вызове функции rememberCoroutineScope()

```
val coroutineScope = rememberCoroutineScope()
```

Объект coroutineScope определяет область действия корутин. Он объявляет диспетчер, который будет использоваться для запуска корутин, а также применяется для включения корутин в область видимости. Все запущенные в этой области корутины можно завершить с помощью вызова метода cancel():

```
coroutineScope.cancel()
```

Для определения корутин применяется особый тип функций Kotlin - suspend-функции. Подобные функции объявляются с помощью ключевого слова suspend, которое указывает Kotlin, что функцию можно приостановить и возобновить позже, что позволяет выполнять длительные вычисления без блокировки основного потока. Например:

```
suspend fun doWork() {

     // Здесь выполняем длительную задачу

}
```

## Диспетчеры корутин

Диспетчер отвечает за назначение корутин определенным потокам, а также за приостановку и возобновление работы корутины в течение ее жизненного цикла. Kotlin поддерживает потоки для различных типов асинхронной активности, и при запуске корутины можно указать конкретного диспетчера из следующих вариантов:

- Dispatchers.Main: запускает корутину в основном потоке и подходит для корутин, которым надо взаимодействовать с пользовательским интерфейсом, а также в качестве универсального

варианта для выполнения легких задач.

- Dispatchers.IO: рекомендуется для корутин, которые выполняют операции ввода-вывода - операции с сетью, диском или базой данных.
- Dispatchers.Default: предназначен для задач, которые требуют интенсивного использования процессора, таких как сортировка данных или выполнение сложных вычислений.

Например, следующий код запускает корутину с помощью диспетчера ввода-вывода:

```
coroutineScope.launch(Dispatchers.IO) {

    // выполнение операций с сетью, базой данных или файлами

}
```

## Строители корутин

Строители корутин (coroutine builders) создают и запускают корутины. Kotlin предоставляет следующие типы строителей корутин:

- launch: запускает корутину, не блокируя текущий поток, и не возвращает результат вызывающей стороне. Вызывается из suspend-функции и применяется когда не требуются результаты корутины
- async: запускает корутину и позволяет вызывающей стороне дождаться результата с помощью функции await(), не блокируя текущий поток. Применяется для параллельного запуска нескольких корутин. Вызывается только из другой suspend-функции.
- withContext: позволяет запускать корутину в контексте, отличном от того, который используется родительской корутиной
- coroutineScope: подходит для ситуаций, когда suspend-функция запускает несколько корутин, которые будут выполняться параллельно, и когда некоторые действия должны выполняться только тогда, когда все корутины завершаются. Если эти корутины запускаются с помощью построителя coroutineScope, вызывающая функция не завершится до тех пор, пока не завершатся все дочерние корутины. При использовании coroutineScope сбой в любой из корутин приведет к отмене всех остальных корутин.
- supervisorScope: аналогичен coroutineScope за исключением того, что сбой в одном дочернем элементе не приводит к отмене других корутин.
- runBlocking: запускает корутину и блокирует текущий поток до тех пор, пока корутина не завершится. Обычно это противоположно тому, что требуется от корутин, но полезно для тестирования кода и интеграции устаревшего кода и библиотек.

# Job

Каждый вызов построителя корутин возвращает объект Job, который можно использовать для отслеживания и управления жизненным циклом соответствующей корутины. Последующие вызовы построителя корутины из уже запущенной корутины создают новые объекты Job, которые станут дочерними элементами родительского объекта Job, образуя дерево отношений родитель-потомок. А отмена родительского объекта Job рекурсивно отменит все его дочерние элементы Job. Однако отмена дочернего элемента не отменяет родителя.

Тип Job обладает рдядом полезных свойств и методов. Свойства isActive, isCompleted и isCancelled связанного объекта Job позволяют определить статус корутины. Метод cancel() завершает объект Job и все дочерние корутины, а вызов метода cancelChildren() завершит только все дочерние корутины. Метод join() можно вызвать для приостановки корутины до тех пор, пока не будут выполнены все дочерние объекты Job. Чтобы выполнить объект Job и завершить его после всех дочерних заданий, вызывается метод cancelAndJoin().

## Вызов корутин к Compose

Рассмотрим пример применения корутин:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.ui.unit.sp
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.compose.runtime.rememberCoroutineScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent{
            val coroutineScope = rememberCoroutineScope()
            Column(Modifier.padding(5.dp).fillMaxSize()) {
                Button(onClick = {
                    coroutineScope.launch {
                        doWork()
                    }
                }) {
                    Text("Click", fontSize = 28.sp)
                }
            }
        }
    }
```

```
}
suspend fun doWork() {
    println("doWork starts")
    delay(5000) // симулируем долгую работу с помощью задержки в 5 секунд
    println("doWork ends")
}
```

Здесь определена suspend-функция doWork, которая просто выводит на консоль пару сообщений. Для симуляции долговременной работы в ней вызывается функция delay(), которая устанавливает задержку в 5 секунд. Но в реальности это могло бы быть обращение к локальным файлам, к базе данных, сетевые запросы или какие-нибудь долговременные вычисления.

Для вызова этой функции создается область корутины:

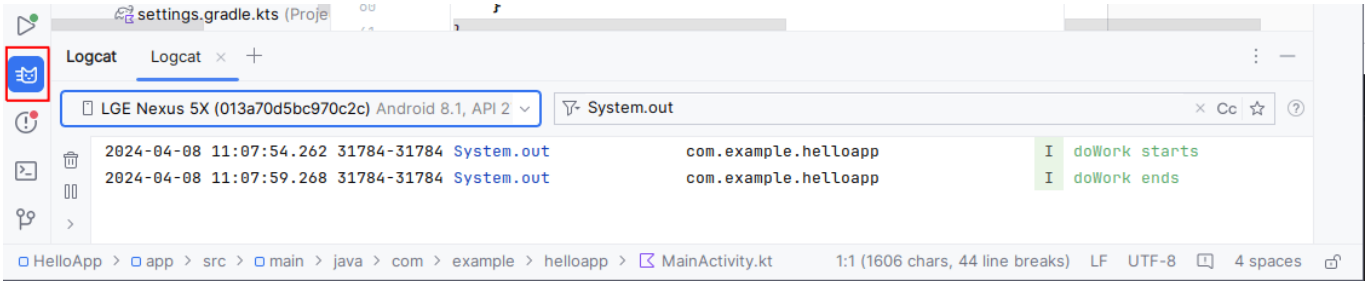
```
val coroutineScope = rememberCoroutineScope()
```

В компоненте Button определяем код обработчика нажатия - в нем запускается корутина:

```
Button(onClick = {
    coroutineScope.launch {
        doWork()
    }
})
```

Для создания корутины здесь применяется построитель coroutineScope.launch, и в области корутины вызывается функция doWork.

Запустим приложение и нажмем на кнопку, и в окне Logcat внизу Android Studio мы сможем увидеть сообщения из функции doWork:

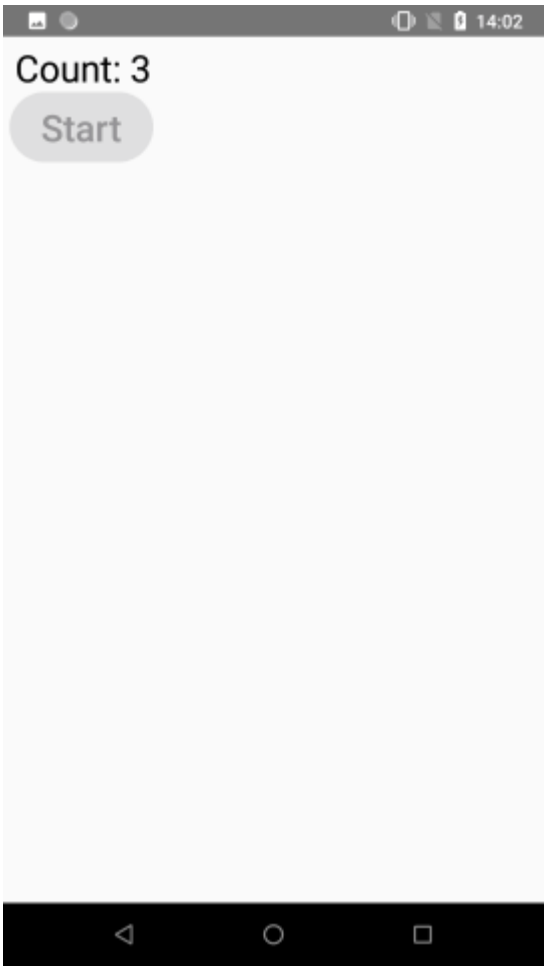



Аналогичным образом можно обращаться из корутин к состоянию компонентов и таким образом воздействовать на компоненты:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
```

```
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.ui.unit.sp
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent{
            val coroutineScope = rememberCoroutineScope()
            val enabled = remember{mutableStateOf(true)}
            val count = remember{mutableStateOf(0)}
            Column(Modifier.padding(5.dp).fillMaxSize()) {
                Text("Count: ${count.value}", Modifier.padding(start = 4.dp),
fontSize = 28.sp)
                Button(onClick = {
                    coroutineScope.launch {
                        enabled.value = false
                        for(n in 1..5){
                            count.value = n
                            delay(1000)
                        }
                        enabled.value = true
                    }
                }, enabled = enabled.value) {
                    Text("Start", fontSize = 28.sp)
                }
            }
        }
    }
}
```

Здесь при нажати на кнопку корутина изменяет значения переменных состояния enabled и count. В данном случае при нажатии на кнопку делаем кнопку недоступной для нажатия, покак не завершит работу корутина. Для этого переключаем значение переменной enabled, которая привязана к одноименному параметру кнопки Button. В корутине в цикле увеличиваем значение count до 5 и затем завершаем корутину, делая кнопку внось активной. А для вывода текущего значения count определен компонент Text.




# LaunchedEffect

Выше корутины запускались по нажатию на кнопку из обработчика события onClick. Но что, если мы хотим запускать корутину в каком-то другом месте, например, непосредственно в функции компонента:

```
suspend fun doWork() {
    println("doWork starts")
    delay(5000)
    println("doWork ends")
}

@Composable
fun HelloWork(n) {
    val coroutineScope = rememberCoroutineScope()
    coroutineScope.launch() {
        doWork()
    }
}
```

При попытке скомпилировать этот код мы получим ошибку

```
Calls to launch should happen inside a LaunchedEffect and not composition
```

Суть ошибки заключается в том, что мы не можем где угодно запускать корутины таким образом внутри компонента, поскольку это может вызвать нежелательные побочные эффекты. В контексте Jetpack Compose побочный эффект возникает, когда асинхронный код вносит изменения в состояние компонента из другой области корутины без учета жизненного цикла этого компонента. И есть вероятность, что корутина может продолжить работу после завершения функции компонента.

Чтобы избежать этой проблемы, необходимо запускать корутины из компонентов LaunchedEffect или SideEffect. Эти два компонента считаются безопасными для запуска корутин, поскольку они знают о жизненном цикле родительского компонента. Когда вызывается функция компонента LaunchedEffect, которая содержит код запуска корутины, корутина немедленно запускается и начинает выполнять асинхронный код. Как только родительский компонент завершается, экземпляр LaunchedEffect и корутина также завершаются.

Общий синтаксис запуска корутин из LaunchedEffect выглядит следующим образом:

```
LaunchedEffect(key1, key2, ...) {

    coroutineScope.launch() {

        // здесь выполняется асинхронный код
    }
}
```

Значения параметров key1, key2 и так далее управляют поведением корутины в течение рекомпозиции. Причем необходимо указать как минимум один такой параметр. Пока значения любого из этих параметров остаются неизменными, LaunchedEffect будет продолжать выполнение одной и той же корутины в течение несколько рекомпозиций родительского компонента. Однако если значение параметра изменится, LaunchedEffect завершит текущую корутину и запустит новую.

Применим компонент LaunchedEffect:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material3.Text
import androidx.compose.ui.unit.sp
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.rememberCoroutineScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent{
```

```
            HelloWork()
        }
    }
}
suspend fun doWork() {
    println("doWork starts")
    delay(5000)
    println("doWork ends")
}

@Composable
fun HelloWork() {
    val coroutineScope = rememberCoroutineScope()
    LaunchedEffect(key1 = Unit) {
        coroutineScope.launch() {
            doWork()
        }
    }
    Text("Hello Work", fontSize = 28.sp)
}
```

Здесь в компоненте HelloWork запускается корутина, которая выполняет функцию doWork. Эта функция для демонстрации просто логгирует на консоль две строки с задержкой в 5 секунд. Для запуска корутины определен компонент LaunchedEffect. Обратите внимание, что в функцию этого компонента в качестве ключа передается значение Unit. Это значение указывает, что корутину не нужно воссоздавать в течение рекомпозиций. В остальном работа с корутиной протекает также. Также для простой демонстрации внутри HelloWork вызывается компонент Text, который выводит надпись "Hello Work".

Запустим приложение, и при запуске компонента, и в окне Logcat внизу Android Studio мы сможем увидеть сообщения из функции doWork:

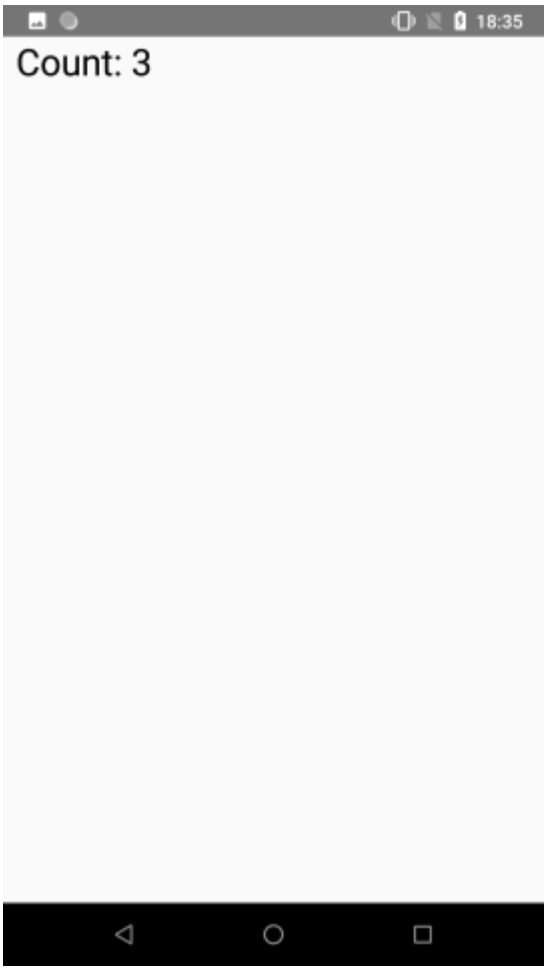

Еще один пример - изменение состояния компонента:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Text
import androidx.compose.ui.unit.sp
import androidx.compose.runtime.Composable
```

```
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent{
            Counter()
        }
    }
}

@Composable
fun Counter() {
    val count = remember{mutableStateOf(0)}
    val coroutineScope = rememberCoroutineScope()
    LaunchedEffect(key1 = Unit) {
        coroutineScope.launch() {
            for(n in 1..5){
                count.value = n
                delay(1000)
            }
        }
    }
    Text("Count: ${count.value}", Modifier.padding(start = 10.dp), fontSize =
28.sp)
}
```

Здесь определен компонент Counter, в котором вложенный компонент Text отображает значение состояния - переменной count. В корутине, которая запускается в LaunchedEffect, в цикле изменяем значение count с 1 до 5 с задержкой в 1 секунду. После этого выходим из цикла, и корутина завершается.

Запустим приложение, и при старте будет запущена корутина, которая будет изменять состояние переменной count:




Кроме LaunchedEffect Jetpack Compose также предоставляет компонент SideEffect. В отличие от LaunchedEffect, корутина в SideEffect выполняется после завершения композиции родительского компонента. Кроме того SideEffect не принимает параметрjd и перезапускается при каждой перекомпозиции родительского компонента.

# Потоки Flow

Потоки являются частью языка программирования Kotlin и предназначены для последовательного возврата нескольких значений из асинхронных задач на основе корутин. Подробнее про потоки можно прочитать на этом в соответствующей главы из руководства по Kotlin: Асинхронные потоки. В данном же случае мы рассмотрим применение потоков в контексте мобильного приложения Android.

## Создание потока Flow

Самую базовую форму потока в Kotlin представляет тип Flow Каждый поток может отправлять данные только одного типа, который должен быть указан при объявлении потока. Например, для потоковой передачи строк можно использовать тип `Flow<String>`.

При объявлении потока нам нужно определить код для генерации данных. Для этого можно использовать специальную функцию-строитель flow(), которая принимает в качестве параметра блок корутин с кодом генерации данным. Например:

```
val myFlow: Flow<String> = flow {

    // генерация значений типа String
}
```

В данном случае определяется поток myFlow, предназначенный для генерации значений String.

В качестве альтернативы можно использовать функцию-строитель потока flowOf() для преобразования фиксированного набора значений в поток:

```
val myFlow2 = flowOf("Tom", "Bob", "Sam")
```

Кроме того, многие типы коллекций Kotlin имеют функцию расширения asFlow(), которую можно вызывать для преобразования данных коллекции в поток. Например, следующий код преобразует массив строковых значений в поток:

```
val myFlow3 = arrayOf<String>("Tom", "Bob", "Sam").asFlow()
```

## Генерация данных

После создания потока следует предоставить механизм для передачи данных во вне. Из трех построителей потоков, которые мы рассмотрели в предыдущем разделе, только построители flowOf() и asFlow() создают потоки, которые автоматически отправляют данные, как только потребитель-внешний код начинает извлекать из потока данные. Однако в случае с функцией flow() нам нужно самим написать код, который будет вручную выдавать каждое значение по мере его появления. Для этого применяется функция emmit(), в которую в качестве аргумента передается текущее значение для передачи из потока. Например:

```
val myFlow: Flow<Int> = flow {

    for (i in 0..9) {
        emit(i)
        delay(2000)
    }
}
```

Здесь myFlow в цикле перебирает числа от 0 до 9 и через вызов emit() возвращает текущее перебираемое число. Для имитации долгой работы на каждой итерации цикла выполняется 2-секундная задержка, что позволяет продемонстрировать асинхронный характер потока.

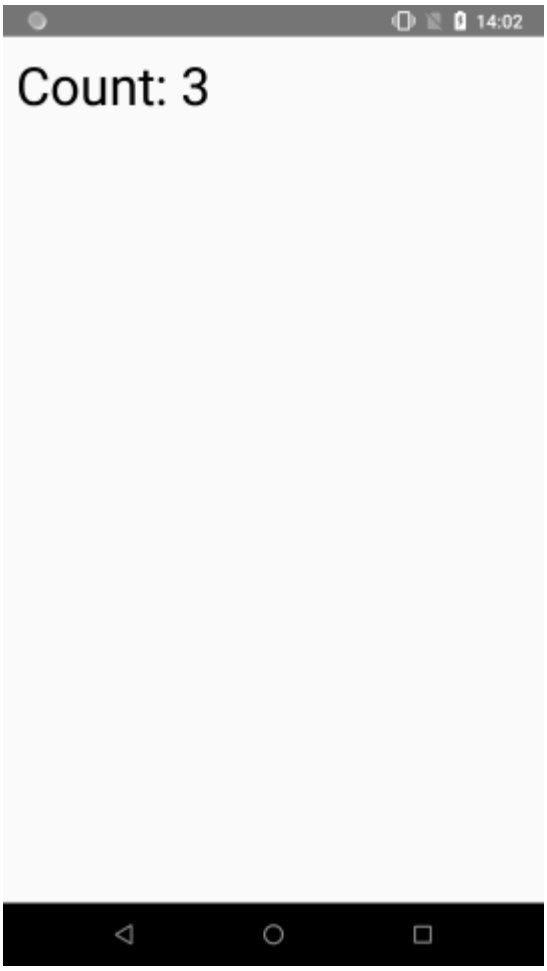
## Поток как состояние и collectAsState()

Jetpack Compose предоставляет функцию collectAsState(), которая позволяет получить текущее значение из потока в виде состояния. ПРи получении нового значения произойдет рекомпозиция интерфейса. Например:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.padding
import androidx.compose.runtime.Composable
import androidx.compose.material3.Text
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.flow

class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val myFlow = flow {
                for (i in 0..9) {
                    emit(i)
                    delay(2000)
                }
            }
            Main(flow = myFlow)
        }
    }
}
@Composable
fun Main(flow: Flow<Int>) {
    val count by flow.collectAsState(initial = 0)
    Text("Count: $count", Modifier.padding(10.dp), fontSize = 40.sp)
}
```

Здесь в лямбда-выражении в методе setContent() создается простейший поток `Flow<Int>`, который генерирует числа от 0 до 9 с интервалом в 2 секунды. Этот поток передается в кастомный компонент Main, где текущее значение этого потока получаем в переменную состояния count. А компонент Text выводит ее значение.

Таким образом при запуске приложения каждые 2 секунды будет генерироваться новое число и помещаться в состояние count, а компонент Text будет его отображать:




## Преобразование потока

И как в общем случае, к потокам Flow в приложении на Jetpack Compose можно применять различные преобразования. Основные из них были рассмотрены на этом сайте в руководстве по Kotlin в статье Операции с потоками. Например, используем ряд операций:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.padding

import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.material3.Text
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.filter
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.flow.map
```

```
class MainActivity : ComponentActivity() {

    val myFlow = flow {
        for (i in 0..9) {
            emit(i)
            delay(2000)
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val newFlow = myFlow.filter {it % 2 == 0}.map{ "Current value = $it"}
            Main(flow = newFlow)
        }
    }
}
@Composable
fun Main(flow: Flow<String>) {
    val count by flow.collectAsState(initial = 0)
    Text("$count", Modifier.padding(10.dp), fontSize = 40.sp)
}
```

На уровне класса MainActivity определяется поток myFlow, который содержит числоа от 0 до 9.

В лямбда-выражении из метода setContent этот поток изменяется - из него выбираются только четные числа, которые трансформируются в строки

```
val newFlow = myFlow.filter {it % 2 == 0}.map{ "Current value = $it"}
```

И этот измененный поток передается в компонент Main.

Или другой пример - используем функцию reduce(), которая сводит все значения потока к одному значению:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.padding

import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.material3.Text
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
```

```
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.flow.reduce

class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val myFlow = flow {
                for (i in 0..9) {
                    emit(i)
                    delay(2000)
                }
            }

            Main(flow = myFlow)
        }
    }
}
@Composable
fun Main(flow: Flow<Int>) {
    var count by remember { mutableStateOf<Int>(0) }
    LaunchedEffect(Unit) {
        flow.reduce { accumulator, value ->
            count = accumulator
            accumulator + value
        }
    }
    Text("$count", Modifier.padding(10.dp), fontSize = 40.sp)
}
Функция reduce() принимает два п
```

Функция reduce() принимает два параметра в виде аккумулятора и текущего значения из потока. Первое значение из потока помещается в аккумулятор, и между аккумулятором и текущим значением выполняется указанная операция (с сохранением результата в аккумуляторе). В данном случае мы просто складываем все числа из потока. Поскольку функция reduce() является suspend-функцией, она выполняется в контексте корутины. И для ее выполнения определяем компонент LaunchedEffect.

# StateFlow

Поток состояния, представленный интерфейсом StateFlow, применяется как способ наблюдения за изменением состояния в приложении. Каждый объект StateFlow используется для хранения одного

значения, которое может меняться со временем, а также для уведомления системы об изменении этого значения. StateFlow ведет себя так же, как LiveData, за исключением того, что LiveData учитывает жизненный цикл и не требует начального значения.

Для создания объекта StateFlow можно использовать встроенную функцию MutableStateFlow(), в которую передается обязательное начальное значение и которая возвращает объект одноименного типа:

```
private val _stateFlow = MutableStateFlow(0)
```

Далее эта переменная будет использоваться для изменения текущего значения состояния из кода приложения. Затем для преобразование в StateFlow у объекта MutableStateFlow вызывается метод asStateFlow():

```
val stateFlow = _stateFlow.asStateFlow()
```

Для изменения состояния измненеия вносятся через свойство value объекта MutableStateFlow:

```
_stateFlow.value += 1
```

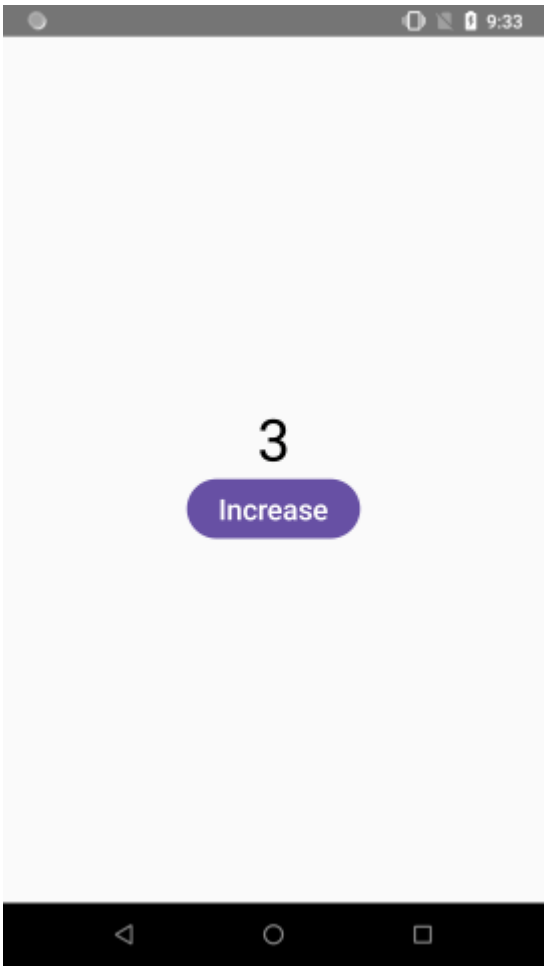
Когда поток активен, состояние можно использовать с помощью метода collectAsState() или напрямую с помощью функции collection() или collectLatest(). Рассмотрим небольшой пример:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material3.Button

import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.material3.Text
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.sp
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow


class MainActivity : ComponentActivity() {
```

```
    private val _stateFlow = MutableStateFlow(0)
    val stateFlow = _stateFlow.asStateFlow()
    fun increase() { _stateFlow.value += 1 }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Main(flow=stateFlow, onIncrease = ::increase)
        }
    }
}

@Composable
fun Main(flow: StateFlow<Int>, onIncrease: ()->Unit) {
    val count by flow.collectAsState()
    Column(Modifier.fillMaxSize()) {
        Text("$count", fontSize = 44.sp)
        Button(onClick = { onIncrease() }) {
            Text("Increase", fontSize = 22.sp)
        }
    }
}
```

Здесь определяем состояние потока StateFlow в виде переменных _stateFlow и stateFlow, а также определяет функцию increase() для изменения состояния.

Компонент Main получает текущее значение потока в переменную count с помощью метода stateFlow.collectAsState(). Для вывода текущего значения определен компонент Text. А компонент Button позволяет по нажатию запустить функциюincrease(), тем самым изменяя значение в потоке.




И в принципе мы могли бы все то же самое сделать и без StateFlow, ограничившись просто переменной состояния. Но StateFlow позволяет нам потреблять новые значения именно как поток. Например, изменим приложение следующим образом:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.Button

import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.material3.Text
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateListOf
import androidx.compose.runtime.remember
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
```

```
import androidx.compose.ui.unit.sp
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow


class MainActivity : ComponentActivity() {

    private val _stateFlow = MutableStateFlow(0)
    val stateFlow = _stateFlow.asStateFlow()
    fun increase() { _stateFlow.value += 1 }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Main(flow=stateFlow, onIncrease = ::increase)
        }
    }
}

@Composable
fun Main(flow: StateFlow<Int>, onIncrease: ()->Unit) {
    val count by flow.collectAsState()
    val messages = remember { mutableStateListOf<Int>()}
    LaunchedEffect(Unit) {
        flow.collect {
            messages.add(it)
        }
    }
    Column(Modifier.fillMaxSize()) {
        Text("$count", fontSize = 44.sp)
        Button(onClick = { onIncrease() }) {
            Text("Increase", fontSize = 22.sp)
        }
        LazyColumn {
            items(messages) {
                Text("Collected Value = $it",Modifier.padding(5.dp))
            }
        }
    }
}
```

Теперь в компоненте Main также определена еще одна переменная состояния - messages, которая хранит некоторый набор сообщений, связанных с состоянием stateFlow:

```
val messages = remember { mutableStateListOf<Int>()}
```

С помощью дополнительного компонента LaunchedEffect запускаем метод collect(), который собирает все новые зиначения из потока и добавляет их в список messages

```
LaunchedEffect(Unit) {
    flow.collect {
        messages.add(it)
    }
}
```

В итоге при изменения значения по нажатию на кнопку также будет пополняться список messages, который выводится в LazyColumn:


# SharedState

SharedFlow предоставляет более универсальный вариант потоковой передачи, чем StateFlow. Ключевые различия между StateFlow и SharedFlow заключаются в следующем:

- При создании объекта SharedFlow начальное значение не указывается
- SharedFlow позволяет получить значения, которые были отправлены до начала сбора данных
- SharedFlow генерирует значения с помощью emit() вместо использования свойства value

Для создания объекта SharedFlow сначала вызывается функция MutableSharedFlow(), которая создает значение типа MutableSharedFlow:

```
public fun <T> MutableSharedFlow(
    replay: Int = 0,
    extraBufferCapacity: Int = 0,
    onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND
): MutableSharedFlow<T>
```

Эта функция принимает три параметра:

- replay: количество значений, воспроизводимых новым подписчикам
- extraBufferCapacity: количество значений, буферизуемых в дополнение к значению параметра replay. emit не приостанавливается, пока остается свободное пространство в буфере.
- onBufferOverflow: настраивает действие отправки при переполнении буфера. Значения, отличные от BufferOverflow.SUSPEND, поддерживаются только в том случае, если replay больше 0 или extraBufferCapacity больше 0. Переполнение буфера может произойти только в том случае, если есть хотя бы один подписчик, который не готов принять новое значение. При отсутствии подписчиков сохраняются только самые последние значения воспроизведения, а поведение переполнения буфера никогда не срабатывает и не имеет никакого эффекта.

Может принимать следующие значения:

-- DROP_LATEST: последнее значение удаляется, когда буфер заполнен, оставляя буфер неизменным при обработке новых значений.

-- DROP_OLDEST: рассматривает буфер как стек, где самое старое значение отбрасывается, чтобы освободить место для нового значения, когда буфер заполнен.

-- SUSPEND: поток приостанавливается, когда буфер заполнен.

После создания у значения MutableSharedFlow вызывается метод asSharedFlow() для получения ссылки SharedFlow.

```
private val _sharedFlow = MutableSharedFlow<Int>()
val sharedFlow = _sharedFlow.asSharedFlow()
```

Значения передаются в поток SharedFlow путем вызова метода emit() объекта MutableSharedFlow:

```
someCoroutineScope.launch {
    for (i in 1..5) {
        _sharedFlow.emit(i)
    }
}
```

Например, возьмем пример с StateFlow из прошлой статьи и изменим его, заменив StateFlow на SharedFlow:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items

import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.material3.Text
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateListOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.MutableSharedFlow
import kotlinx.coroutines.flow.SharedFlow
import kotlinx.coroutines.flow.asSharedFlow
import kotlinx.coroutines.launch


class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val _sharedFlow = MutableSharedFlow<Int>()
            val sharedFlow = _sharedFlow.asSharedFlow()

            val coroutineScope = rememberCoroutineScope()

            LaunchedEffect(Unit) {
                coroutineScope.launch() {
                    for (i in 1..5) {
                        _sharedFlow.emit(i)
                        delay(2000)
                    }
                }
            }

            Main(flow=sharedFlow)
        }
    }
}
```

```
@Composable
fun Main(flow: SharedFlow<Int>) {
    val count by flow.collectAsState(0)
    val messages = remember { mutableStateListOf<Int>()}
    LaunchedEffect(Unit) {
        flow.collect {
            messages.add(it)
        }
    }
    Column(Modifier.fillMaxSize()) {
        Text("$count", fontSize = 44.sp)

        LazyColumn {
            items(messages) {
                Text("Collected Value = $it",Modifier.padding(5.dp))
            }
        }
    }
}
```

Здесь вначале собственно определяем состояние SharedFlow:

```
val _sharedFlow = MutableSharedFlow<Int>()
val sharedFlow = _sharedFlow.asSharedFlow()
```

Для эммитирования значений в поток определяем контекст корутины:

```
val coroutineScope = rememberCoroutineScope()
```

И чтобы при запуске приложения началась генерация значений в поток, определяем компонент LaunchedEffect:

```
LaunchedEffect(Unit) {
    coroutineScope.launch() {
        for (i in 1..5) {
            _sharedFlow.emit(i)
            delay(2000)
        }
    }
}
```

В данном случае в поток поступают числа от 1 до 5 с задержкой в 2 секунды.

Для определения интерфейса определен компонент Main, который принимает поток SharedFlow. В этом компоненте сначала определяем состояние:

```
val count by flow.collectAsState(0)
val messages = remember { mutableStateListOf<Int>()}
```

Переменная count получает текущее значение потока. А переменная messages хранит список полученных из потока значений. Для сбора значений в messages определен компонент LaunchedEffect:

```
LaunchedEffect(Unit) {
    flow.collect {
        messages.add(it)
    }
}
```

Для отображения текущего значения из потока определен компонент Text, а для вывода всех полученных значений - компонент LazyList: