

МНОГОПОТОЧНОСТЬ



Kotlin Coroutines

Корутины - паттерн проектирования, предназначенный для написания асинхронных программ, способных выполнять несколько задач одновременно.

Корутина (coroutine) - сопрограмма, которая выполняется в контексте реального потока. При создании, она не накладывает больших дополнительных расходов на систему, так как не является отдельным потоком. Во время выполнения может быть приостановлена в определенной точке приостановки (suspension point), с сохранением своего полного состояния, для передачи управления другой корутине.



Kotlin Coroutines

Преимущества по сравнению с потоками:

- **Более легковесные** - на создание потока выделяется много ресурсов системы. Количество потоков ограничено.
- **Эффективное использование ресурсов** - пока одна корутина находится в состоянии ожидания, другая может выполняться на потоке - поток не простаивает.
- **Упрощение написания асинхронного кода** - написание асинхронного кода осуществляется в последовательном стиле, что упрощает чтение и поддержку кода.
- **Поддержка отмены и обработки ошибок** - удобные инструменты для обработки ошибок и отмены выполнения корутин, как отдельных экземпляров, так и целых блоков.



Kotlin Coroutines

suspend - ключевое слово в языке Kotlin, говорит о том, что работа функции может быть приостановлена, без блокировки потока выполнения, и возобновлена в будущем.

```
suspend fun doSomeWork(): SomeResult { ... }
```



Kotlin Coroutines

Suspend функцию можно запустить только из другой suspend функции, либо из специального корутин-билдера (Coroutine Builder).

Coroutine bulder - функции для создания и запуска корутин. Предоставляют разные возможности для запуска корутин, позволяют задавать им определенные свойства. Существует 2 типа корутин-билдеров: **launch** и **async**

Kotlin Coroutines. Launch

launch - запускает корутину, которая не возвращает результат вычисления. Принимает на вход блок кода, который будет выполняться асинхронно.

launch

Kotlin

```
coroutineScope.launch {  
    delay(2_000L)  
    println("Completed")  
}
```

Kotlin Coroutines. Async

async - запускает корутину, которая возвращает результат вычисления. Принимает на вход блок кода, который будет выполняться асинхронно. В отличие от `launch`, возвращает объект `Deferred`, представляющий собой отложенный результат выполнения корутины.

async

Kotlin

```
coroutineScope {  
    val deferredResult: Deferred<Int> = async {  
        delay(1_000L)  
        return@async 100  
    }  
    // Получение/ожидание результата корутины  
    val result = deferredResult.await()  
    println("Result: $result")  
}
```



Kotlin Coroutines.

Области видимости

Coroutine Scope - основной компонент для управления корутинами. Предоставляет возможность запускать и отменять корутины. Управляет их жизненным циклом. А так же несет дополнительную информацию о том, на каком потоке происходит их выполнение.


Kotlin Coroutines. GlobalScope

GlobalScope - глобальный скоуп, не привязан к ЖЦ компонентов андроида, поэтому не рекомендуется к использованию. Может быть использован, если нужно выполнять какую-то работу на протяжении всего ЖЦ приложения.

GlobalScope

Kotlin

```
GlobalScope.launch {  
    while (true) {  
        delay(1_000L)  
        logSomething()  
    }  
}
```



Kotlin Coroutines. ViewModelScope

`viewModelScope` - привязан к ЖЦ вьюмодели. Когда вьюмодель уничтожается, то отменяются все корутины, запущенные в данном скоупе.

Kotlin Coroutines. LifecycleScope

lifecycleScope - привязан к ЖЦ объектов LifecycleOwner (Activity, Fragment). Скоуп с корутинами автоматически отменяются при уничтожении активити или фрагмента.

GlobalScope

Kotlin

```
class FragmentImpl : Fragment() {  
    override fun onCreateView(...) {  
        super.onCreateView(...)  
        lifecycleScope.launch {  
            // doSomeWork()  
        }  
    }  
}
```

Kotlin Coroutines. CoroutineScope

Можно создать свой собственный CoroutineScope. В этом случае придется самому контролировать ЖЦ скоупа.

CoroutineScope

Kotlin

```
class FragmentImpl : Fragment() {  
    val coroutineScope = CoroutineScope(Dispatchers.IO)  
    override fun onCreate(savedInstanceState: Bundle?) {  
        ...  
        coroutineScope.launch {  
            // doSomeWork()  
        }  
    }  
    override fun onDestroy() {  
        ...  
        coroutineScope.cancel()  
    }  
}
```

Kotlin Coroutines. LifecycleScope

runBlocking - используется для запуска скоупа, код которого будет выполняться синхронно. Блокирует текущий поток до тех пор, пока все корутины не завершат свое выполнение. Используется в функции main или при написании тестов.

runBlocking

Kotlin

```
fun main(args: Array<String>) = runBlocking {  
    doSomeWork()  
}
```

Kotlin Coroutines. CoroutineContext

Корутины выполняются в определенном контексте.

CoroutineContext представляет собой индексированный набор элементов и доступен внутри корутины через свойство `coroutineContext`.

Элементы (**CoroutineContext.Element**):

- **coroutineContext[Job]** - достать текущую Job корутины
- **coroutineContext[CoroutineName]** - достать имя корутины
- **coroutineContext[CoroutineExceptionHandler]** - достать обработчик ошибок
- **coroutineContext[ContinuationInterceptor]** - достать диспетчер корутины

Можно создать свой элемент **CoroutineContext.Element**

Kotlin

Coroutines. Job

Когда мы создаем корутину, через `launch` и `async`, нам возвращается объект задачи `Job`. Через него мы можем управлять корутиной.

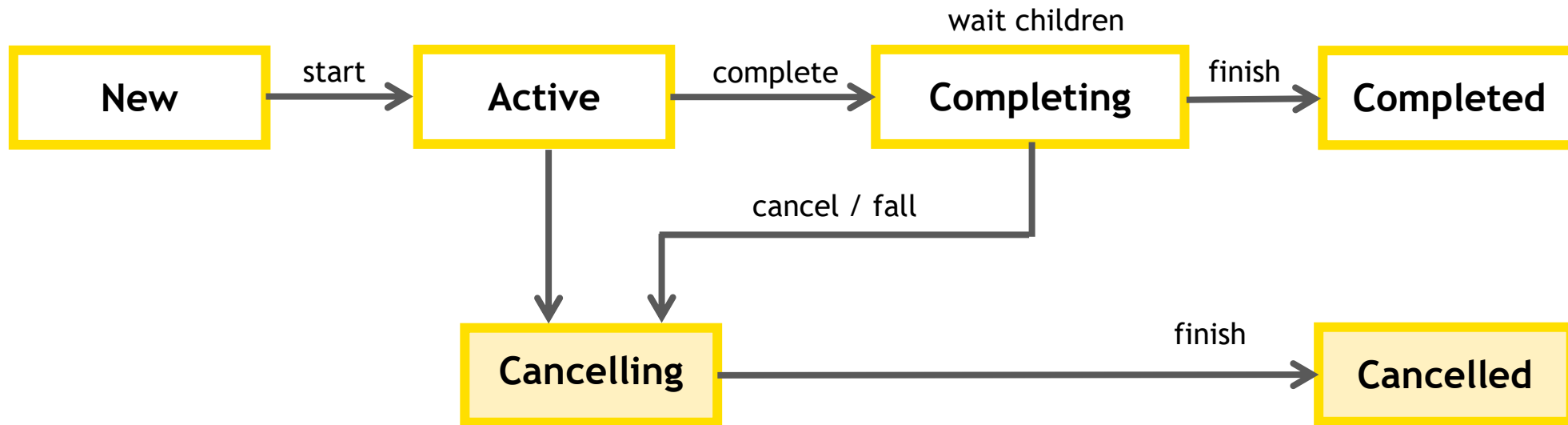
Задачи могут быть связаны друг с другом, образуя иерархию задач, что позволяет контролировать группу корутин.

job

Kotlin

```
val job: Job = CoroutineScope(Dispatchers.IO).launch {  
    doSomeWork()  
}
```

Kotlin Coroutines. Job





Kotlin Coroutines. Job

isActive - активна ли корутина (Не отменена и не завершена)

isCancelled - отменена ли корутина

isCompleted - завершена ли корутина (Не важно каким образом)

join - дождаться выполнения корутины, блокируя текущий поток.

invokeOnCompletion - регистрирует callback, который вызывается при завершении корутины

Kotlin Coroutines. Job

children - получить список
дочерних корутин.

job

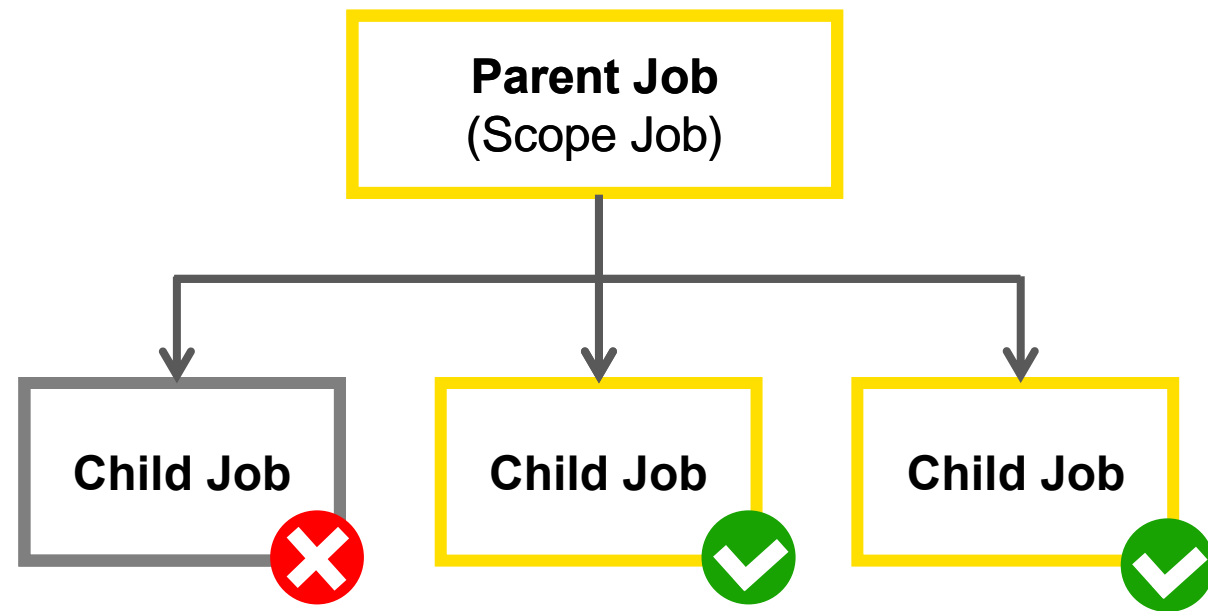
Kotlin

```
val job: Job = CoroutineScope(Dispatchers.IO).launch {  
    launch {  
        delay(2000L)  
    }  
    launch {  
        delay(2000L)  
    }  
}  
println(job.children.count()) // Вывод: 2
```

Kotlin Coroutines. Cancellation

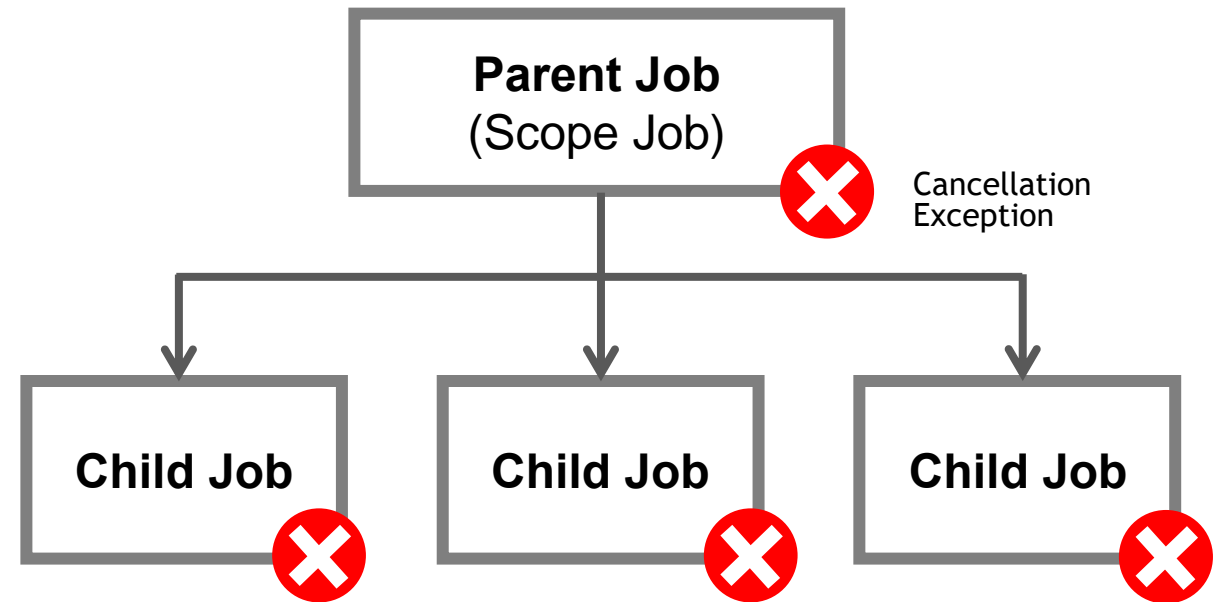
`job.cancel()` - отменить выполнение корутины и ее дочерних корутин (Не гарантирует моментальную отмену). Выбрасывает `CancellationException`.

`job.cancelAndJoin()` - отменить выполнение и дождаться завершения отмены.



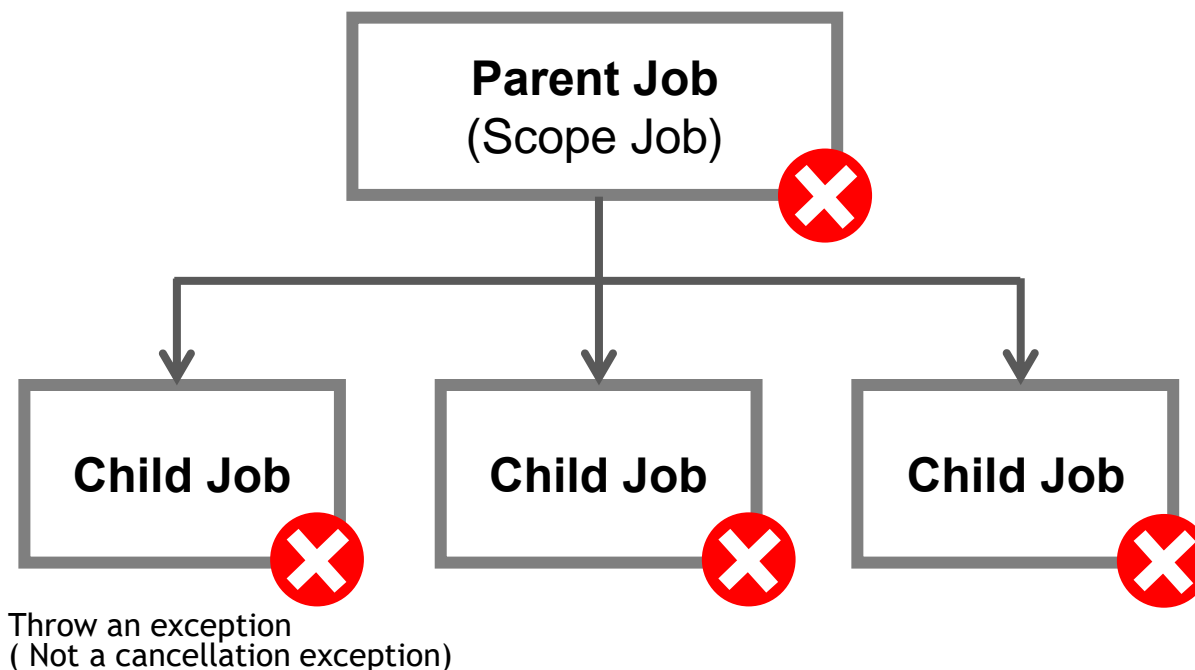
Kotlin Coroutines. Cancellation

`coroutineScope.cancel()`



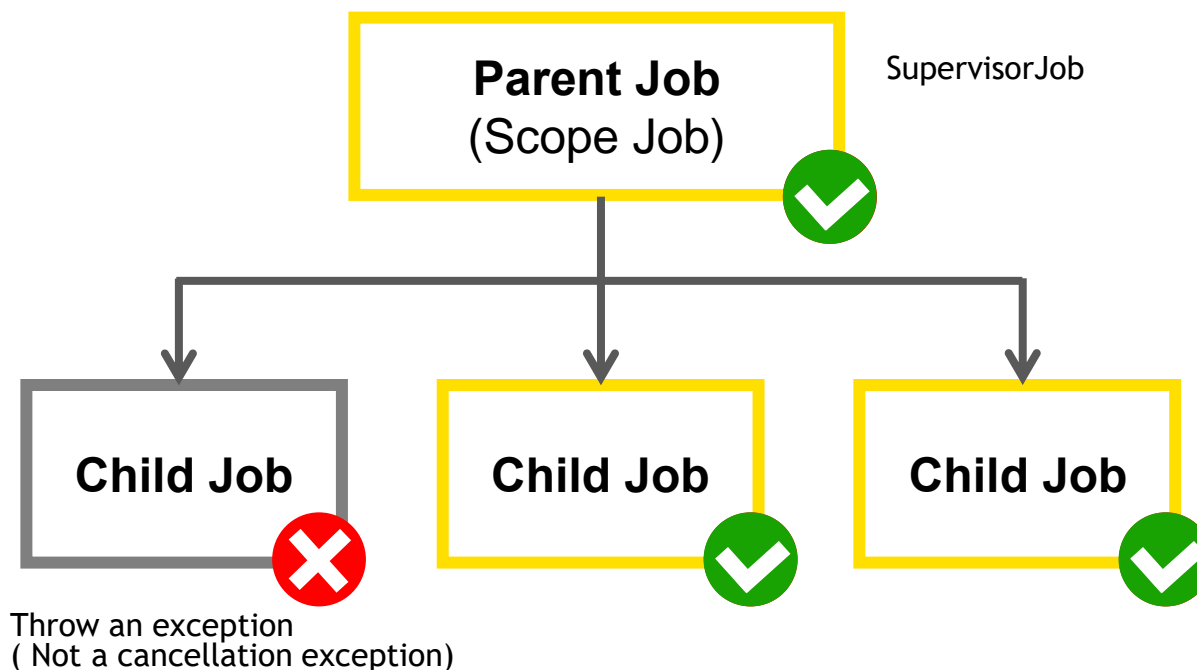
Kotlin Coroutines. Cancellation

В корутине выбросилось
исключение отличное от
CancellationException.



Kotlin Coroutines. Cancellation

SupervisorJob - позволяет дочерним корутинам падать с исключением, не затрагивая другие корутины.



Kotlin Coroutines. SupervisorJob

// Вывод в консоль:

```
Exception in thread "DefaultDispatcher-  
worker-3"  
java.lang.IllegalStateException: Some  
Exception  
...  
Job 1 is done!
```

coroutines

Kotlin

```
val scope = CoroutineScope(Dispatchers.IO + SupervisorJob())  
val job1 = scope.launch {  
    delay(2000)  
    println("Job 1 is done!")  
}  
val job2 = scope.launch {  
    delay(1000)  
    doSomeWorkWithException()  
}
```

Kotlin Coroutines. SupervisorJob

SupervisorJob - не работает в данном случае:

coroutines

Kotlin

```
val scope = CoroutineScope(Dispatchers.IO + SupervisorJob())
scope.launch {
    launch {
        delay(1000)
        doSomeWorkWithException()
    }
    delay(2000)
    println("Job 1 is done!")
}
```


Kotlin Coroutines. SupervisorJob

`supervisorScope` - наследует родительский контекст, но заменяет `Job` на `SupervisorJob` для дочерних корутин.

// Вывод в консоль:

```
Exception in thread "main"  
java.lang.IllegalStateException: Some  
Exception  
    Job 1 is done!
```

coroutines

Kotlin

```
supervisorScope() {  
    launch {  
        delay(2000)  
        println("Job 1 is done!")  
    }  
    launch {  
        delay(1000)  
        doSomeWorkWithException()  
    }  
}
```

Kotlin Coroutines. Обработка исключений

Сработает:

```
Ok                                                                    Kotlin
CoroutineScope(Dispatchers.IO).launch {
    try {
        doSomeWorkWithException()
    } catch (t: Throwable) {
        println(t)
    }
}
```

Не сработает:

```
Not Ok                                                                Kotlin
try {
    CoroutineScope(Dispatchers.IO).launch {
        doSomeWorkWithException()
    }
} catch (t: Throwable) {
    println(t)
}
```

Kotlin Coroutines. Обработка ИСКЛЮЧЕНИЙ

CoroutineExceptionHandler -
позволяет обрабатывать
необработанные исключения в
корутинах.

Exceptions

Kotlin

```
val exceptionHandler =  
    CoroutineExceptionHandler { coroutineContext, throwable ->  
        logError(throwable)  
    }  
val scope = CoroutineScope(Dispatchers.IO)  
scope.launch(exceptionHandler) { doSomeWorkWithException() }  
  
// либо:  
  
val scope = CoroutineScope(Dispatchers.IO + exceptionHandler)  
scope.launch() { doSomeWorkWithException() }
```

Kotlin Coroutines. Обработка исключений

Результат: краш приложения
Корутина при ошибке проверяет, является ли ее родитель другой корутиной или CoroutineScope, до тех пор по цепочке, пока не достигнет CoroutineScope. По итогу, родительская корутина ответственна за обработку ошибок в дочерних корутинах.

Не работает:

Not Ok

Kotlin

```
val scope = CoroutineScope(Dispatchers.IO)
scope.launch() {
    launch(exceptionHandler) { doSomeWorkWithException() }
}
```

Kotlin Coroutines. Обработка исключений

Срабатывает при использовании **SupervisorJob**:

Ok

Kotlin

```
val scope = CoroutineScope(Dispatchers.IO)
scope.launch() {
    launch(SupervisorJob() + exceptionHandler) {
        doSomeWorkWithException()
    }
}
```

Kotlin Coroutines. Dispatchers

Dispatchers.Main — выполнение корутины будет в главном потоке. Должен применяться для операций, которые затрагивают пользовательский интерфейс.

Dispatchers.IO — используется для выполнения операций ввода-вывода (I/O).

Dispatchers.Default — это диспетчер, который используется по умолчанию. Он предназначен для выполнения вычислительных задач и использует общий фоновый пул потоков.

Dispatchers.Unconfined — запускает корутину в вызывающем потоке, но только до первой приостановки. После приостановки корутина возобновляет работу в потоке, который полностью определяется вызванной suspend-функцией. Не рекомендуется использовать в общем коде. Полезен в ситуациях, когда операция в корутине должна быть выполнена немедленно.

Kotlin Coroutines. Dispatchers

withContext - сменить поток
выполнения корутины.

withContext

Kotlin

```
CoroutineScope(Dispatchers.Default).launch {  
    // Default dispatcher  
    doSomeWork()  
    withContext(Dispatchers.IO) {  
        // IO dispatcher  
        doSomeWork()  
    }  
}
```

Flow

flow - представляет собой поток значений, вычисляемых асинхронно.

flow {...} - является холодным потоком, чтобы его запустить необходимо вызвать терминальный оператор **collect**. Каждый подписчик на холодный поток будет работать независимо от других.

flow

Kotlin

```
flow<Int> {  
    for (i in 0 until 10) {  
        emit(i)  
    }  
}.collect { emittedValue ->  
    println(emittedValue)  
}
```


Flow

emit - отправить элемент в поток, блокирует выполнение, пока консьюмер не обработает значение.

collect - терминальный оператор, запускает выполнение flow, собирает и обрабатывает значения, отправленные функцией **emit**.

collectLatest - работает аналогично **collect**, но обрабатывает только последние значения отправленные функцией **emit**.

onEach - выполнить какое-то стороннее действие с новым элементом.

map - преобразовать элемент в другой.

flowOn - выполнить вышестоящие операции на определенном диспетчере.

launchIn - терминальный оператор, запускает флоу в определенном скоупе.

catch - ловит исключения из вышестоящих операторов.

onCompletion - вызывается когда основной flow успешно закончил завершил свою работу. Может дополнительно эмитить значения.

Flow

Вывод без ошибок:

OnEach: 10

OnEach: 20

OnEach: 30

OnEach mapped: 300

OnEach mapped: -1

Вывод при ошибке в map:

OnEach: 10

OnEach: 20

OnEach: 30

Exception: Some exception

OnEach mapped: 0

OnEach mapped: -1

flow

Kotlin

```
val scope = CoroutineScope(Dispatchers.Default)
flowOf(10, 20, 30)
    .onEach { println("OnEach: $it") }
    .mapLatest { value -> // Dispatchers.IO
        delay(100)
        value * 10
    }
    .flowOn(Dispatchers.IO)
    .catch { exception -> // Dispatchers.Default
        println("Exception: $exception")
        emit(0)
    }
    .onCompletion { emit(-1) }
    .onEach { println("OnEach mapped: $it") }
    .launchIn(scope)
```

Flow

Combine - собирает и преобразовывает последние заэмиченные элементы из флоу.

Вывод:

10 a

20 b

30 b

flow

Kotlin

```
val f1 = flowOf(10, 20, 30)
val f2 = flowOf("a", "b")
f1.combine(f2) { f1Value, f2Value ->
    "$f1Value $f2Value"
}.collect(::println)
```

Flow

Zip - в отличии от `combine` дожидается, пока оба `flow` сделают `emit` нового значения. Перестает делать операции преобразования, когда один из флоу закончил работать.

Вывод:

10 a

20 b

flow

Kotlin

```
val f1 = flowOf(10, 20, 30)
val f2 = flowOf("a", "b")
f1.combine(f2) { f1Value, f2Value ->
    "$f1Value $f2Value"
}.collect(::println)
```

Flow

SharedFlow - горячий поток, активен всегда без явного вызова оператора collect. Рассылают актуальные значения всем подписчикам (broadcast).

StateFlow - горячий поток, частный случай sharedFlow, для работы с одним актуальным элементом.

MutableSharedFlow, **MutableStateFlow** - позволяют обновлять/добавлять значения в поток.

flow

Kotlin

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

Flow

Вывод:

Current value = 0
Collected value = 0
Collected value = 10

flow

Kotlin

```
val stateFlow = MutableStateFlow<Int>(0)
stateFlow.onEach {
    println("Collected value = $it")
}.launchIn(coroutineScope)
println("Current value = ${stateFlow.value}")
delay(100)
stateFlow.value = 10
```

Channel

Channel - более низкоуровневое api для коммуникации между корутинами. В отличие от `sharedFlow` элемент получает только один из консьюмеров.

- **send** - отправить данные в канал
- **receive** - получить элемент из канала
- **consumeEach** - подписаться на входящие элементы из канала
- **receiveAsFlow** - получать элементы в виде flow
- **close** - закрыть канал

channel

Kotlin

```
public fun <E> Channel(  
    capacity: Int = RENDEZVOUS,  
    onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND,  
    onUndeliveredElement: ((E) -> Unit)? = null  
): Channel<E>
```

Channel

Вывод:

consume: 1

consume: 2

asFlow: 0

asFlow: 4

consume: 3

asFlow: 5

channel

Kotlin

```
val channel = Channel<Int>()
launch {
    delay(100L)
    for (i in 0 until 6) {
        channel.send(i)
    }
    channel.close()
}
launch {
    channel.consumeEach { println("consume: $it") }
}
channel.receiveAsFlow()
    .onEach { println("asFlow: $it") }
    .launchIn(coroutineScope)
```




True Engineering

630128, г. Новосибирск,
ул. Кутателадзе, 4г

(383) 363-33-51, 363-33-50
info@trueengineering.ru
trueengineering.ru

Новосибирский
Государственный
Университет