Асинхронные потоки

Введение в асинхронные потоки

Корутины позволяют возвращать одиночные значения. Для этого мы можем, к примеру, создавать корутину с помощью построителя async. Но Kotlin также позволяет создавать асинхронные потоки (Asynchronous Flow), которые возвращают набор объектов.

В принципе для получения набора объектов мы могли бы в корутине возвращать коллекцию элементов, например, список List, наподобие следующего:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope<Unit>{
    launch {
        getUsers().forEach { user -> println(user) }
    }
}

suspend fun getUsers(): List<String> {
    delay(1000L)  // имитация продолжительной работы
    return listOf("Tom", "Bob", "Sam")
}
```

Однако проблема таких коллекций в том, что они одномоментно возвращают все объекты. Например, если в списке ожидается 1000 объектов, то соответственно пока функция getUsers() не возвратит список из 1000 объектов (например, получая их из базы данных или из внешнего интернет-ресурса), мы не сможем манипулировать объектами из этого списка.

Эту проблему в Kotlin как раз и позволяют решить асинхронные потоки. Изменим пример выше с применением асинхронных потоков:

```
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.*

suspend fun main(){
    getUsers().collect { user -> println(user) }
}

fun getUsers(): Flow<String> = flow {
    val database = listOf("Tom", "Bob", "Sam")  // условная база данных
    var i = 1;
    for (item in database){
        delay(400L) // имитация продолжительной работы
        println("Emit $i item")
        emit(item) // емитируем значение
        i++
```

```
    }
}
```

Для создания асинхронного потока данных применяется интерфейс Flow. То есть по сути асинхронный поток - это объект Flow. Он типизируется типом тех данных, которые должны передаваться в потоке. В данном случае передаем строки, поэтому Flow типизируется типом String.

```
fun getUsers(): Flow<String>
```

При этом при определении функции-потока (в данном случае функции getUsers) необязательно использовать модификатор suspend.

Для создания объекта Flow применяется специальная функция flow()

```
fun getUsers(): Flow<String> = flow {

    // создание асинхронного потока в функции flow
}
```

В самой функции в данном случае имитируется получение объектов из условной базы данных, коей здесь для простоты служит список List. В цикле пробегаемся по этому списку и отправляем в поток текущий объект с помощью функции emit():

```
emit(item) // передаем значение в поток
```

Это ключевой момент. Благодаря этому внешний код сможет получит переданное через emit() в поток значение и использовать его.

Для индикации номера отправляемого объекта я добавил переменную-счетчик i, которая увеличивается при переходе к другому объекту списка. Вывод номера отправляемого объекта позволяет увидеть, что получение внешним кодом объектов из списка происходит по мере его передачи в поток с помощью функции emit(), а не когда будут отправлены все объекты из списка.

Во внешнем коде в функции main вызываем функцию-поток getUsers(). Для управления объектами из потока для интерфейса Flow определен ряд функций, одной из которых является функция collect(). В качестве параметра она принимает функцию, в которую передает эмитируемый объект из потока. Так, в данном случае это просто функция вывода на консоль:

```
getUsers().collect { user -> println(user) }
```

В итоге мы получим следующий консольный вывод:

```
Emit 1 item
Tom
Emit 2 item
Bob
Emit 3 item
Sam
```

Таким образом, программа не ждет, когда функция getUsers возвратит все строки. А получает строки по мере их отправки в поток через функцию emit().

Другой пример - создадим и используем асинхронный поток чисел:

```
import kotlinx.coroutines.flow.*

suspend fun main(){
    getNumbers().collect { number -> println(number) }
}

fun getNumbers(): Flow<Int> = flow{
    for(item in 1..5){
        emit(item * item)
    }
}
```

Здесь в принципе все то же самое. Функция getNumbers() представляет асинхронный поток объектов Int. В качестве объектов в поток добавляются квадраты чисел от 1 до 5. Консольный вывод программы:

```
1
4
9
16
25
```

Запуск Flow

Стоит отметить, что асинхронный поток не запускается, пока не будет применена терминальная операция над получаемыми даными, например, функция collect():

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val numberFlow = getNumbers()       // поток создан, но не запущен
    println("numberFlow has created")
    println("launch collect function")
    numberFlow.collect { number -> println(number) }   // запуск потока
```

```
}

fun getNumbers() = flow{
    println("numberFlow has started")
    for(item in 1..5){
        emit(item * item)
    }
}
```

Консольный вывод программы:

```
numberFlow has created
launch collect function
numberFlow has started
1
4
9
16
25
```

Создание асинхронного потока

Для создания асинхронного потока можно применять различные способы. Рассмотрим три основных способа.

Функция flow

Функция-построитель потока flow() позволяет задать логику передачи объектов в поток. Она может применяться как к отдельной функции, так и сама по себе. Например, создание потока на базе функции:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val numberFlow = getNumbers()
    numberFlow.collect{n -> println(n)}
}

fun getNumbers() = flow{
    for(item in 1..5){
        emit(item * item)
    }
}
```

Здесь построитель flow создает поток на базе функции getNumbers(), передавая в поток квадраты значений от 1 до 5

Консольный вывод:

```
1
4
9
16
25
```

Определять поток в виде отдельной функции, как в примере выше, необязательно. Например:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = flow {
        val usersList = listOf("Tom", "Bob", "Sam")
        for (item in usersList){
            emit(item)
        }
    }
    userFlow.collect({user -> println(user)})
}
```

Здесь определена переменная userFlow, которая имеет тип Flow и которая представляет поток, создаваемый построителем flow. В данном случае flow передает в поток объекты из списка строк. Консольный вывод программы:

```
Tom
Bob
Sam
```

Функция flowOf

Специальная функция-строитель flowOf() создает поток из набора переданных в функцию значений.

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val numberFlow : Flow<Int> = flowOf(1, 2, 3, 5, 8)
    numberFlow.collect{n -> println(n)}
}
```

В данном случае в функцию построителя асинхронного потока flowOf() передается 5 значений типа Int, поэтому создаваемый поток будет имет тип Flow. Все переданные значения будут автоматически эмитироваться в поток. А получить их можно также как и в общем случае, например, через функцию collect().

Консольный вывод программы:

```
1
2
3
5
8
```

Аналогичный пример со строками:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = flowOf("Tom", "Sam", "Bob")
    userFlow.collect({user -> println(user)})
}
```

Метод asFlow

Стандартные коллекции и последовательности в Kotlin имеют метод расширения asFlow(), который позволяет преобразовать коллекцию или последовательность в поток:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    // преобразование последовательности в поток
    val numberFlow : Flow<Int> = (1..5).asFlow()
    numberFlow.collect{n -> println(n)}

    // преобразование коллекции List<String> в поток
    val userFlow = listOf("Tom", "Sam", "Bob").asFlow()
    userFlow.collect({user -> println(user)})
}
```

Операции с потоками

Для работы с потоками в Kotlin для интерфейса Flow определено ряд функций. В зависимости от того, возвращают они конкретное значение или обработанный поток эти функции делятся на два вида:

терминальные функции и промежуточные функции. Рассмотрим основные из них.

Терминальные функции потоков Терминальные функции потоков (terminal operators) представляют suspend-функции, которые позволяют непосредственно получать объекты из потока или возвращают какое-то конечное значение:

- collect(): получает из потока переданные значения
- toList(): преобразует поток значений в коллекцию List
- toSet(): преобразует поток значений в коллекцию Set
- first() / firstOrNull(): получает первый объект из потока
- last() / lastOrNull(): получает последний объект из потока
- single() / singleOrNull(): ожидает получение одного объекта из потока
- count(): получает количество элементов в потоке
- reduce(): получает результат определенной операции над элементами потока
- fold(): получает результат определенной операции над элементами потока, в отличие от функции reduce() принимает начальное значение

Промежуточные функции

Промежуточные функции (Intermediate operator) принимают поток и возвращают обработанный поток.

- combine(): объединяет два потока в один, после применения к их элементам функции преобразования
- drop(): исключает из начала потока определенное количество значений и возвращает полученный поток
- filter(): фильтрует поток, оставляя те элементы, которые соответствуют условию
- filterNot(): фильтрует поток, оставляя те элементы, которые НЕ соответствуют условию
- filterNotNull(): фильтрует поток, удаляя все элементы, которые равны null
- map(): применяет к элементам потока фукцию преобразования
- onEach(): применяет к элементам потока определенную функцию перед тем, как они будут переданы в возвращаемый поток
- take(): выбирает из потока определенное количество элементов
- transform(): применяет к элементам потока функцию преобразования
- zip(): из двух потоков создает один, применяя к их элементам функцию преобразования

Функции count, take и drop. Количество элементов в потоке

Функция count

Оператор count получает количество объектов в потоке:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = listOf("Tom", "Bob", "Sam").asFlow()
    println("Count: ${userFlow.count()}")       // Count: 3
}
```

Также мы можем передать в функцию count() условие в виде функции, которая возвращает объект Boolean, то есть true или false. Тогда функция count() возвратит количество элементов, которые соответствуют этому условию:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = listOf("Tom", "Bob", "Kate", "Sam", "Alice").asFlow()
    val count = userFlow.count{ username -> username.length > 3 }
    println("Count: $count")       // Count: 2
}
```

В данном случае в качестве условия передается функция username -> username.length > 3 . Ее параметр - это объект потока. Здесь мы говорим учитывать строку, если ее длина больше 3 символов. В итоге в приведеном списке только два объекта соответствуют этому условию, поэтому переменная count будет равна 2.

Функция take

Оператор take ограничивает количество элементов в потоке. В качестве параметра она принимает количество элементов с начала потока, которые надо оставить в потоке:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = listOf("Tom", "Bob", "Kate", "Sam", "Alice").asFlow()
```

```
    userFlow.take(3).collect{user -> println(user)}
}
```

В примере выше оставляем первые три элемента:

```
Tom
Bob
Kate
```

Функция drop

Оператор drop удаляет из потока определенное количество элементов. В качестве параметра она принимает количество элементов с начала потока, которые надо убрать из потока:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = listOf("Tom", "Bob", "Kate", "Sam", "Alice").asFlow()
    userFlow.drop(3).collect{user -> println(user)}
}
```

В примере выше убираем первые три элемента, в итоге в потоке останется два последних элемента:

```
Sam
Alice
```

Функции first, last, single

first/firstOrNull

Метод first() получает первый объект списка:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = listOf("Tom", "Bob", "Kate", "Sam", "Alice").asFlow()
    val firstUser = userFlow.first()
    println("First User: $firstUser")       // First User: Tom
}
```

Также метод first() может в качестве параметра принимать функцию-условие, которая возвращает объект Boolean. Тогда first() возвращает первый элемент потока, который соответствует этому условию:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = listOf("Tom", "Bob", "Kate", "Sam", "Alice").asFlow()
    val firstUser = userFlow.first{ name-> name.length > 3}
    println("First User: $firstUser")       // First User: Kate
}
```

Здесь мы получаем первый элемент, длина которого больше 3 символов. В потоке это строка "Kate".

Однако может сложиться ситуация, когда условию не соответствует ни один из элементов потока:

```
val userFlow = listOf("Tom", "Bob", "Sam").asFlow()
val firstUser = userFlow.first{ name-> name.length > 3}
```

Или поток банально пуст:

```
val userFlow = listOf<String>().asFlow()
val firstUser = userFlow.first()
```

В обоих случаях при работе программы вылетит исключение. В этом случае мы, конечно, может обрабатывать исключение с помощью try..catch. Однако, в качестве альтернативы Kotlin предоставляет метод firstOrNull(), который возвращает null, если поток пуст или ни один из его элементов не соответствуют условию:

```
val userFlow = listOf<String>().asFlow()
val firstUser1 = userFlow.firstOrNull()
val firstUser2 = userFlow.firstOrNull{ name-> name.length > 3}
```

В этом случае мы можем проверить полученное значение на null:

```
if(firstUser1 == null)
    println("User not found")
else
    println("First User: $firstUser1")
```

last / lastOrNull

Функция last() вытягивает из потока последний элемент, только в отличие от функции first() (по крайней мере для версии Kotlin 1.5.1) она не принимает условие: также может принимать условие:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = listOf("Tom", "Bob", "Alice", "Sam").asFlow()
    val lastUser = userFlow.last()
    println("Last User: $lastUser")       // Last User: Sam
}
```

Если есть вероятность, что поток будет пуст, то можно использовать функцию lastOrNull, которая возвращает null в случае отсутствия элементов:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = listOf("Tom", "Bob", "Alice", "Sam").asFlow()
    val lastUser = userFlow.lastOrNull()
    if(lastUser!=null) println("Last User: $lastUser")
}
```

single / singleOrNull

Функция single() возвращает единственный элемент потока, если поток содержит только один элемент. Если поток не содержит элементов генерируется исключение NoSuchElementException, а если в потоке больше одного элемента - исключение IllegalStateException.

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = listOf("Tom").asFlow()
    try {
        val singleUser = userFlow.single()
        println("Single User: $singleUser")
    }
    catch(e:Exception) { println(e.message)  }
}
```

В качестве альтернативы можно использовать метод singleOrNull(), который возвращает null, если поток пуст или если в потоке больше одного элемента.

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = listOf("Tom", "Bob").asFlow()
    val singleUser = userFlow.singleOrNull()
    if(singleUser!=null)
        println("Single User: $singleUser")
    else
        println("Not found")
}
```

Преобразование данных. Функции map и transform

Функция map

Оператор map() преобразует данные потока. В качестве параметра он принимает функцию преобразования. Функция преобразования принимает в качестве единственного параметра объект из потока и возвращает преобразованные данные.

```
inline fun <T, R> Flow<T>.map(
    crossinline transform: suspend (value: T) -> R
): Flow<R> (source)
```

Рассмотрим на примере:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val peopleFlow = listOf(
        Person("Tom", 37),
        Person("Sam", 41),
        Person("Bob", 21)
    ).asFlow()

    peopleFlow.map{ person -> person.name }
        .collect { personName -> println(personName) }
}

data class Person(val name: String, val age: Int)
```

В данном случае определяем поток данных peopleFlow, который содержит объекты типа Person. Далее к этому потоку применяется функция map(), в которую передается следующая функция преобразования:

```
person -> person.name
```

То есть функция преобразования принимает в качестве параметра person объект типа Person и возвращает значение его поля name - то есть строку. Таким образом, функция map() из потока объектов Person создаст поток обектов String.

Далее к полученному потоку объектов String применяем функцию collect(), в которой получаем каждую строку из потока и выводим ее на консоль:

```
collect { personName -> println(personName) }
```

Консольный вывод программы:

```
Tom
Sam
Bob
```

Другой пример:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val peopleFlow = listOf(
        Person("Tom", 37),
        Person("Bill", 5),
        Person("Sam", 14),
        Person("Bob", 21),
    ).asFlow()

    peopleFlow.map{ person -> object{
        val name = person.name
        val isAdult = person.age > 17
    }}.collect { user -> println("name: ${user.name}   adult:  ${user.isAdult} ")}
}

data class Person(val name: String, val age: Int)
```

В данном случае функция map преобразует поток данных типа Person в поток объектов анонимного типа, который определяет два поля: name (имя) и isAdult (больше ли пользователю 17 лет). Соответственно в функции collect мы получаем объекты этого анонимного типа и выводим их данные на консоль. Консольный вывод:

```
```

```
name: Tom   adult:  true
name: Bill   adult:  false
name: Sam   adult:  false
name: Bob   adult:  true
```

Функция transform

Оператор transform также позволяет выполнять преобразование объектов в потоке. В отличие от map она позволяет использовать функцию emit(), чтобы передавать в поток произвольные объекты.

```
inline fun <T, R> Flow<T>.transform(
    crossinline transform: suspend FlowCollector<R>.(value: T) -> Unit
): Flow<R> (source)
```

Оператор transform принимает функцию, которая получает в качестве параметра объект потока. Например:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val peopleFlow = listOf(
        Person("Tom", 37),
        Person("Bill", 5),
        Person("Sam", 14),
        Person("Bob", 21),
    ).asFlow()

    peopleFlow.transform{ person ->
        if(person.age > 17){
            emit(person.name)
        }
    }.collect { personName -> println(personName)}
}

data class Person(val name: String, val age: Int)
```

В данном случае получаем в функции transform объект Person из потока. Если поле age этого объекта больше 17, то передаем его в новый поток через функцию emit(). Консольный вывод программы:

```
Tom
Bob
```

Причем при обработке одного объекта мы можем несколько раз вызывать функцию emit(), передавая таким образом в потоке несколько объектов:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val numbersFlow = listOf(2, 3, 4).asFlow()
    numbersFlow.transform{ n ->
        emit(n)
        emit(n * n)
    }.collect { n -> println(n)}
}
```

Например, здесь преобразуем поток чисел. Причем в выходной поток передается само число и его квадрат. Таким образом, на консоли мы увидим:

```
2
4
3
9
4
16
```

Фильтрация данных

Функция filter

Оператор filter выполняет фильтрацию объектов в потоке. В качестве параметра он принимает функцию-условие, которая получает объект потока и возвращает true (если объект подходит фильтрацию) и false (если не проходит):

```
inline fun <T> Flow<T>.filter(
    crossinline predicate: suspend (T) -> Boolean
): Flow<T> (source)
```

Рассмотрим на примере:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val peopleFlow = listOf(
        Person("Tom", 37),
```

```
        Person("Bill", 5),
        Person("Sam", 14),
        Person("Bob", 21),
    ).asFlow()

    peopleFlow.filter{ person -> person.age > 17}
        .collect { person -> println("name: ${person.name}   age:  ${person.age}
")}
}

data class Person(val name: String, val age: Int)
```

Здесь для фильтрации применяется условие person.age > 17. Если возраст пользователя больше 17, то он оказывается в выходном потоке. Консольный вывод программы:

```
name: Tom   age:  37
name: Bob   age:  21
```

takeWhile

Кроме того, Kotlin предоставляет еще ряд операций фильтрации для различных ситуаций. Например, оператор takeWhile выбирает из потока элементы, пока будет истино некоторое условие:

```
fun <T> Flow<T>.takeWhile(
    predicate: suspend (T) -> Boolean
): Flow<T>
```

Рассмотрим на примере:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val peopleFlow = listOf(
        Person("Tom", 37),
        Person("Alice", 32),
        Person("Bill", 5),
        Person("Sam", 14),
        Person("Bob", 25),
    ).asFlow()

    peopleFlow.takeWhile{ person -> person.age > 17}
        .collect { person -> println("name: ${person.name}   age:  ${person.age}
")}
}

data class Person(val name: String, val age: Int)
```

В данном случае оператор takeWhile будет извлекать в возвращаемый поток все объекты Person, у которых значение age больше 17. Консольный вывод программы:

```
name: Tom   age:  37
name: Alice   age:  32
```

Из консольного вывода мы видим, что несмотря на то, что в потоке еще есть объекты Person, которые соответствуют условию, но столкнувшись с первым объектом, который не соответствует условию, takeWhile перестает выбирать объекты в возвращаемый поток.

dropWhile

Оператор dropWhile противоположен по своему действию оператору takeWhile. dropWhile удаляет из потока элементы, пока они не начнут соответствовать некоторому условию:

```
fun <T> Flow<T>.dropWhile(
    predicate: suspend (T) -> Boolean
): Flow<T>
```

Так, возьмем предыдущий пример, только вместо takeWhile используем dropWhile:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val peopleFlow = listOf(
        Person("Tom", 37),
        Person("Alice", 32),
        Person("Bill", 5),
        Person("Sam", 14),
        Person("Bob", 25),
    ).asFlow()

    peopleFlow.dropWhile{ person -> person.age > 17}
        .collect { person -> println("name: ${person.name}   age:  ${person.age}
")}
}

data class Person(val name: String, val age: Int)
```

В данном случае оператор dropWhile пропустит первые два объекта Person, которые соответствуют условию person.age > 17. Третий элемент потока НЕ соответствует этому условию, поэтому начиная с третьего элемента все остальные элементы попадут в возвращаемый поток (даже если они опять же соответствуют данному условию). Консольный вывод программы:

```
name: Bill   age:  5
name: Sam   age:  14
name: Bob   age:  25
```

Сведение данных. Функции reduce и fold

Функция reduce

Оператор reduce сводит все значения потока к одному значению:

```
suspend fun <S, T : S> Flow<T>.reduce(
    operation: suspend (accumulator: S, value: T) -> S
): S (source)
```

reduce принимает функцию, которая имеет два параметра. Первый параметр при первом запуске представляет первый объект потока, а при последующих запусках - результат функции над предыдущими объектами. А второй параметр функции - следующий объект.

Например, у нас есть поток чисел, найдем сумму всех чисел в потоке:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val numberFlow = listOf(1, 2, 3, 4, 5).asFlow()
    val reducedValue = numberFlow.reduce{ a, b -> a + b }
    println(reducedValue)   // 15
}
```

Здесь при первом запуске в функции в reduce параметр a равен 1, а параметр b равен 2.

При втором запуске параметр a содержит результат предыдущего выполнения функции, то если число 1 + 2 = 3, а параметр b равен 3 - следующее число в потоке.

Или другой пример со строками:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = listOf("Tom", "Bob", "Kate", "Sam", "Alice").asFlow()
    val reducedValue = userFlow.reduce{ a, b -> "$a $b" }
    println(reducedValue)   // Tom Bob Kate Sam Alice
}
```

Здесь reduce соединяет все строки в одну.

Функция fold

Функция fold также сводит все элементы потока в один. Но в отличие от оператора reduce оператор fold в качестве первого параметра принимает начальное значение:

```
inline suspend fun <T, R> Flow<T>.fold(
    initial: R,
    crossinline operation: suspend (acc: R, value: T) -> R
): R (source)
```

Например:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val userFlow = listOf("Tom", "Bob", "Kate", "Sam", "Alice").asFlow()
    val foldedValue = userFlow.fold("Users:", { a, b -> "$a $b" })
    println(foldedValue)   // Users: Tom Bob Kate Sam Alice
}
```

В данном случае начальным значением является строка "Users:", к которой затем добавляются остальные элементы потока данных.

Объединение потоков

Оператор zip позволяет объединить два потока данных:

```
fun <T1, T2, R> Flow<T1>.zip(
    other: Flow<T2>,
    transform: suspend (T1, T2) -> R
): Flow<R> (source)
```

Оператор zip принимает два параметра. Первый параметр - поток данных, с которым надо выполнить объединение. Второй параметр - собственно функция объединения. Она принимает соответствующие элементы обоих потоков в качестве параметров и возвращает результат их объединения.

Например, соединим два потока:

```
import kotlinx.coroutines.flow.*
```

```
suspend fun main(){

    val english = listOf("red", "yellow", "blue").asFlow()
    val russian = listOf("красный", "желтый", "синий").asFlow()
    english.zip(russian) { a, b -> "$a: $b" }
        .collect { word -> println(word) }
}
```

Здесь оператор zip последовательно перебирает элементы из обоих потоков - english и russian. Элемент первого потока, на котором вызывается оператор zip, передается в параметр a, а элемент второго потока - в параметр b. Функция соединения объединяет оба элемента в одну строку. И каждая такая строка из двух элементов передается в качестве элемента в новый создаваемый поток. Консольный вывод программы:

```
red: красный
yellow: желтый
blue: синий
```

Хотя в примере выше оператор zip объединял потоки одного типа - String и возвращал поток того же типа, но в реальности объединяемые потоки могут представлять разные данные, а возвращаемый поток - еще один тип данных:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val names = listOf("Tom", "Bob", "Sam").asFlow()
    val ages = listOf(37, 41, 25).asFlow()
    names.zip(ages) { name, age -> Person(name, age) }
        .collect { person -> println("Name: ${person.name}   Age: ${person.age}")
}
}

data class Person(val name: String, val age: Int)
```

В данном случае функция zip, которая вызывается на потоке типа String, объединяет его элементы с элементами потока типа Int. Причем результатом такого объединения становится поток объектов типа Person. Результат работы программы:

```
Name: Tom   Age: 37
Name: Bob   Age: 41
Name: Sam   Age: 25
```