# Kotlin Coroutine

添加Kotlin Coroutines的依赖：

```
implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.2.0'
// 协程核心库
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.4.3"
// 协程Android支持库
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.4.3"
// 协程Java8支持库
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-jdk8:1.4.3"
```

# 协程的基本概念

## 什么是协程

协程基于线程，是一种轻量级线程。

现阶段对于协程并没有一种官方的明确定义，但是和线程一样，协程可以处理并发事件。

Kotlin中的协程可以将异步代码通过同步化的方式运行，杜绝回调地狱。

协程最核心的点就是，函数或者一段程序能够被挂起，稍后再在挂起的位置恢复。

## 协程在Android开发中用来解决什么问题

- 处理耗时任务，这种任务常常会阻塞主线程。
- 保证主线程安全，即确保安全的从主线程调用任何suspend函数。
- 异步逻辑同步化。

# 协程中的挂起，调度和恢复

- 挂起：挂起是一种操作，经过挂起之后的函数，会被阻塞，即暂停运行，直到主动或者被动的恢复运行。
- 调度：调度是协程的调度器根据情况，选择执行哪些可以运行的协程代码块。得到调度之后的协程代码才可能运行，未经过调度的代码不会得到执行。
- 恢复：被挂起的函数经过特定的操作之后（如网络请求返回，数据库请求返回）会被重新在挂起的地方执行。

# 协程的简单使用

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        btn.setOnClickListener {
            GlobalScope.launch(Dispatchers.Main) {
                val user = withContext(Dispatchers.IO) {
                    userServiceApi.getUser()
                }
                nameTextView.text = user.name
            }
        }
    }
}
```

其中，GlobalScope的代码段就是一个协程的简单使用，首先通过协程放在主线程的调度器中运行，但是网络请求需要在IO环境中执行，因此需要切换到IO环境再去请求数据，数据请求返回之后会把数据通过textview显示出来。

需要注意的是，网络请求可能会花很久，因此，网络请求的代码处会被挂起，直到请求返回，返回之后协程会被恢复。

# 协程的创建

上面的代码简单的进行了协程的创建，以下是几个重要的概念

- 协程作用域：协程是一种特殊的轻量级线程，因此协程需要在特殊的环境中运

行，即协程的作用域。在上面的代码中，GlobalScope就是一个协程作用域，里面所有的代码都会在该协程作用域中运行。GlobalScope是一个全局的协程作用域，属于顶层的协程作用域。

- 协程调度器：协程的调度器是用来管理协程的运行状态的装置。不同的协程调度器负责不同的协程调度，有主线程协程调度器，IO协程调度器，CPU密集型任务调度器。

## 协程的挂起和恢复

- suspend：也被成为挂起或者暂停，用于暂停执行当前的协程任务，并保存所有的局部变量。
  被suspend修饰的函数被称为挂起函数（可被挂起函数）。
- resume：用于让已经暂停的协程从暂停的位置继续执行。

采用suspend函数的方法重写上面的代码，如下：

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        btn.setOnClickListener {
            GlobalScope.launch(Dispatchers.Main) {
                // 在主线程中去获取user信息和展示name
                getUser()
            }
        }
    }

    /**

     * 这是一个可以挂起的函数，通过上下文可以知道，这个函数在主线程的环境中运行
     * 首先通过get方法去获取user信息，再进行展示。
     */
    private suspend fun getUser() {
        val user = get()
        showUser(user)
    }
```

```
22
23      /**
24       * 这也是一个挂起函数，因为网络请求是一个耗时操作，因此需要切换到IO环境去执行。
25       * 当网络请求还没有返回的时候，这个函数会被挂起，当网络请求返回之后，这个函数会被恢
26      复。
27       */
28      private suspend fun get(): User {
29          return withContext(Dispatchers.IO) {
30              userServiceApi.getUser()
31          }
32      }
33
34      /**
35       * 这个函数继承Main的调度器的上下文，因此可以操作UI。
36       */
37      private fun showUser(user: User) {
38          nameTextView.text = user.name
39      }
40  }
```

# 挂起函数

- 使用suspend关键字修饰的函数被称为挂起函数。
- 挂起函数只能在协程体内或者其他的其他的挂起函数内调用。

最简单的协程挂起函数就是delay，当运行到delay时，函数会被挂起，delay的时候到了之后，函数会被恢复。

# 协程的两部分

- Kotlin的协程实现分为两个层次
    - 基础设施层：标准库的协程API，主要对协程提供了概念上和与以上最基本的支持
    - 业务框架层：协程的上层框架支持

以下的代码利用Kotlin协程的基础设施层创建一个协程：

```kotlin
import kotlin.coroutines.*

fun main() {
    val continuation = suspend {
        5
    }.createCoroutine(object : Continuation<Int> {
        /**
         * Continuation需要一个协程的上下文，这里可以简单给一个空的协程上下文
         */
        override val context: CoroutineContext
            get() = EmptyCoroutineContext

        /**
         * 协程恢复的时候的回调
         */
        override fun resumeWith(result: Result<Int>) {
            println("result: ${result.getOrNull()}")
        }
    })
    continuation.resume(Unit)
}
```

# 协程的调度器

协程的调度器是一个管理和运行协程的装置，所有的协程都必须在调度器中运行，及时他们在主线
程上运行也是如此。

- Dispachers.Main：Android上的主线程，用来处理UI交互和一些轻量级的任务
    - 调用suspend函数
    - 调用UI函数，更新UI
    - 更新LiveData
- Dispatchers.IO：IO调度器，该调度器中的协程不在主线程中运行，IO调度区专门为磁盘和网络IO进行了优化

    - 读写数据库
    - 文件读写
    - 网络处理
- Dispatchers.Default：默认调度器，专门为CPU密集型任务进行了优化
    - 数组排序
    - JSON数据解析
    - 处理差异判断

# 协程的任务泄露

当某一个协程任务丢失，无法追踪，会导致内存，CPU，磁盘等资源浪费，甚至发送一个无用的网络请求，这种情况称之为任务泄露。

为了能够避免协程任务泄露，Kotlin协程引入了结构化并发机制。

比如，当我们进行一个页面时，会发送相关的网络请求，如果此时使用的是顶级协程，如果在网络请求返回之前，该页面已经被关闭，但是协程依然会持续运行，但是协程已经无法对该页面进行修改和更新，此时就会发生协程的任务泄露。

# 结构化并发

使用结构化并发可以做到：

- 取消任务，当某项任务不再需要时取消它。
- 追踪任务，当任务正在执行时，追踪它。
- 发出错误信号，当协程失败时，发出错误信号表明有错误发生。

简单来说，结构化并发可以对协程的进行有组织的进行管理。其中，协程作用域是结构化并发的有效手段。

# CoroutineScope

定义协程必须指定其CoroutineScope,它会跟踪所有协程，同样它还可以取消由它所启动的所有协
程。

常用的相关API有：

- GlobalScope：生命周期是process级别的，即使Activity或Fragment已经被销毁，协程仍然在
  执行。
- MainScope：在Activity中使用，可以在onDestroy()中取消协程。
- viewModelScope：只能在ViewModel中使用，绑定ViewModel的生命周期。
- lifecycleScope：只能在Activity、Fragment中使用，会绑定Activity和Fragment的生命周期。

# 协程启动和取消

launch与async构建器都用来启动新协程

- launch，返回一个Job并且不附带任何结果值。
- async，返回一个Deferred, Deferred也是一个Job，可以使用await()在一个延期的值上得到它
  的最终结果。

```
import kotlinx.coroutines.async
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun testCoroutineBuilder() {
    return runBlocking {
        val job1 = launch {
            delay(200)
            println("job1 finished")
        }
        val job2 = async {
            delay(200)
            println("job2 finished")
            "job2 result"
        }
        println(job2.await())
    }
}
```

```
fun main() {
    testCoroutineBuilder()
}
```

得到的结果如下：

```
job1 finished
job2 finished
job2 result
```

可以看到，job1和job2都会得到执行，但是job2会带有协程的执行结果，这里的结果只是一个简单
的字符串，然后我们可以通过job2.await()的方式获取相对应的结果。

本质上，async启动的是一个Deffered对象，Deffered是Job的一个子类。而launch启动的就是一个普通的Job。

有时候会出现一个任务需要等待另外一个任务执行完成之后再启动，此时可以通过join或者await方
法进行等待一个作业。

- join和await
- 组合并发

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun testCoroutineBuilder() {
    return runBlocking {
        val job1 = launch {
            println("Start1:" + System.currentTimeMillis())
            delay(200)
            println("One")
            println("Job1 Finished: " +
System.currentTimeMillis())
        }
```

```
        job1.join()
        val job2 = launch {
            println("Start2:" + System.currentTimeMillis())
            delay(200)
            println("Two")
            println("Job2 Finished: " +
System.currentTimeMillis())
        }
        val job3 = launch {
            println("Start3:" + System.currentTimeMillis())
            delay(200)
            println("Three")
            println("Job3 Finished: " +
System.currentTimeMillis())
        }
    }
}

fun main() {
    testCoroutineBuilder()
}
```

运行代码，可以得到如下的结果：

```
Start1:1636257516186
One
Job1 Finished: 1636257516405
Start2:1636257516415
Start3:1636257516415
Two
Job2 Finished: 1636257516621
Three
Job3 Finished: 1636257516621
```

可以看到的是，job1最先开始执行，但是在创建job2之前，调用了job1.join()函数。

关于join函数，kotlin官方的定义如下：

```
public interface Job : CoroutineContext.Element {
    ...

    /**
     * Suspends coroutine until this job is complete. This invocation resumes normally (without exception)
     * when the job is complete for any reason and the [Job] of the invoking coroutine is still [active][isActive].
     * This function also [starts][Job.start] the corresponding coroutine if the [Job] was still in _new_ state.
     *
     * Note, that the job becomes complete only when all its children are complete.
     *
     * This suspending function is cancellable and **always** checks for the cancellation of invoking coroutine's Job.
     * If the [Job] of the invoking coroutine is cancelled or completed when this
     * suspending function is invoked or while it is suspended, this function
     * throws [CancellationException].
     *
     * In particular, it means that a parent coroutine invoking `join` on a child coroutine that was started using
     * `launch(coroutineContext) { ... }` builder throws [CancellationException] if the child
     * had crashed, unless a non-standard [CoroutineExceptionHandler] if installed in the context.
     *
     * This function can be used in [select] invocation with [onJoin] clause.
     * Use [isCompleted] to check for completion of this job without waiting.
     *
     * There is [cancelAndJoin] function that combines an invocation of [cancel] and `join`.
     */
    public suspend fun join()

```

```
    ...
}
```

简单来说，join函数也是一个挂起函数，它会将对应的协程挂起，直到当前的job完成。

在上面的代码中，代码执行到join之后，runblocking的整个协程体会被挂起，直到job1执行完毕，再继续从join之后继续执行代码。

而后继续创建job2和job3，这两个子协程会在之后的一段时间内得到调度执行。从结果可以看出，
job2和job3几乎是同一时刻得到调度执行。

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun testCoroutineBuilder() {
    return runBlocking {
        val job1 = launch {
            println("Start1:" + System.currentTimeMillis())
            delay(200)
            println("One")
            println("Job1 Finished: " +
System.currentTimeMillis())
        }
        val job2 = launch {
            println("Start2:" + System.currentTimeMillis())
            delay(200)
            println("Two")
            println("Job2 Finished: " +
System.currentTimeMillis())
        }
        job1.join()
        job2.join()
        val job3 = launch {
            println("Start3:" + System.currentTimeMillis())
            delay(200)
            println("Three")
```

```
            println("Job3 Finished: " +
System.currentTimeMillis())
        }
    }
}

fun main() {
    testCoroutineBuilder()
}
```

```
Start1:1636257719509
Start2:1636257719517
One
Job1 Finished: 1636257719721
Two
Job2 Finished: 1636257719724
Start3:1636257719728
Three
Job3 Finished: 1636257719933
```

对于上面的一段代码，可以看到首先会创建两个不同的子协程job1和job2，得到调度之后就会被执
行，之后遇到了job1的join函数，此时代码会被挂起，一直等到job1的协程完成执行，而后又遇到了
job2的join函数，一直等到job2的协程执行完成。需要注意的是，在上面的代码中，因为job1和job2几
乎是同时执行，因此也几乎是结束执行，当运行到job2.join()时，协程很快会挂起然后结束。对于更加
复杂的任务，job2可能会挂起比较久，可能会有更加明显的感受。

对于async的await也是类似的原理。

```
import kotlinx.coroutines.async
import kotlinx.coroutines.delay
import kotlinx.coroutines.runBlocking
import kotlin.system.measureTimeMillis

fun testCoroutineLearning() = runBlocking {
```

```
    val time = measureTimeMillis {
        val job1 = async {
            delay(2000)
            19
        }
        val job2 = async {
            delay(2000)
            20
        }
        println("result: " + (job1.await() + job2.await()))
    }
    println(time)
}

fun main() {
    testCoroutineLearning()
}
```

运行上面的代码，可以得到下面的结果：

```
result: 39
2087
```

可以看出，得到输出结果的事件约为2000毫秒，说明两个job是几乎同时执行的。

上面的代码首先会创建两个不同的子协程，得到调度之后会得到运行（上面的两个子协程很简单，
因此这两个协程几乎会同时得到调度执行，但是运行到job1.await()获取job1的结果时，整个协程会被挂起，直到job1的结果返回，然后再运行到job2.await()时，整个协程又会被挂起，直到job2的结果返回。但是由于job1和job2几乎是并行执行，因此这里可以很快恢复（resume），几乎不会花费很多时间。所以最后的结果是两个job并行的运行总时间，大约是2000毫秒。

但是对于下面的一段代码：

```
import kotlinx.coroutines.async
import kotlinx.coroutines.delay
import kotlinx.coroutines.runBlocking
import kotlin.system.measureTimeMillis
```

```
5
6  fun testCoroutineLearning() = runBlocking {
7      val time = measureTimeMillis {
8          val job1 = async {
9              delay(2000)
10             19
11         }.await()
12         val job2 = async {
13             delay(2000)
14             20
15         }.await()
16         println("result: " + (job1 + job2))
17     }
18     println(time)
19 }
20
21 fun main() {
22     testCoroutineLearning()
23 }
```

可以得到的结果如下：

```
1  result: 39
2  4082
```

可以发现，运行时间大约是4000毫秒，和上面的分析思路类似，job1创建之后会被await()，此时会直接挂起协程，直到job1的结果得到返回，期间会被挂起2000毫秒。和job1类似，job2创建之后会直接await()，直到对应的结果返回，期间也会被挂起2000毫秒。然后再输出最后的结果和时间，时间是两次被挂起2000毫秒的总和，约为4000毫秒。

虽然结果是正确的，但是job1和job2是一个串行执行的状态，没有完全利用可以并发执行的优势。

# 协程的启动模式

观察launch或者async的函数定义：

```
/**
 * Launches new coroutine without blocking current thread and returns a reference to the coroutine as a [Job].
 * The coroutine is cancelled when the resulting job is [cancelled][Job.cancel].
 *
 * Coroutine context is inherited from a [CoroutineScope], additional context elements can be specified with [context] argument.
 * If the context does not have any dispatcher nor any other [ContinuationInterceptor], then [Dispatchers.Default] is used.
 * The parent job is inherited from a [CoroutineScope] as well, but it can also be overridden
 * with corresponding [coroutineContext] element.
 *
 * By default, the coroutine is immediately scheduled for execution.
 * Other start options can be specified via `start` parameter. See [CoroutineStart] for details.
 * An optional [start] parameter can be set to [CoroutineStart.LAZY] to start coroutine _lazily_. In this case,
 * the coroutine [Job] is created in _new_ state. It can be explicitly started with [start][Job.start] function
 * and will be started implicitly on the first invocation of [join][Job.join].
 *
 * Uncaught exceptions in this coroutine cancel parent job in the context by default
 * (unless [CoroutineExceptionHandler] is explicitly specified), which means that when `launch` is used with
 * the context of another coroutine, then any uncaught exception leads to the cancellation of parent coroutine.
 *
 * See [newCoroutineContext] for a description of debugging facilities that are available for newly created coroutine.
```

```
 *
 * @param context additional to [CoroutineScope.coroutineContext] context of the coroutine.
 * @param start coroutine start option. The default value is [CoroutineStart.DEFAULT].
 * @param block the coroutine code which will be invoked in the context of the provided scope.
 **/
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job {
    val newContext = newCoroutineContext(context)
    val coroutine = if (start.isLazy)
        LazyStandaloneCoroutine(newContext, block) else
        StandaloneCoroutine(newContext, active = true)
    coroutine.start(start, coroutine, block)
    return coroutine
}
```

可以看到该函数有三个参数组成：

- context：这个参数表示的协程运行的上下文环境，默认是空的上下文环（EmptyCoroutineContext）。
- start：这个参数表示的是协程的启动模式，有一个默认的启动模式。该参数可以有特定的几种启动模式。
    - DEFAULT：协程创建后，立即开始调度，在调度前如果协程被取消，其将直接进入取消响应的状态。
    - ATOMIC：协程创建后，立即开始调度，协程执行到第一个挂起点之前不响应取消。
    - LAZY：只有协程被需要时，包括主动调用协程的start、join或者await等函数时才会开始调度，如果调度前就被取消，那么该协程将直接进入异常结束状态。
    - UNDISPATCHED：协程创建后立即在当前函数调用栈中执行，直到遇到第一个真正挂起的点。

## DEFAULT

```
package com.example.coroutinelearning

import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

/**
 * 启动模式，缺省时
 *
 * 协程立即等待被调度执行(等待被调度，不是立即执行)
 *
 * 打印：
 * 1
 * 3
 * 2
 */
fun coroutineStart() = runBlocking {
    println("1")
    launch {
        println("2")
    }
    println("3")
    delay(3000)

}

fun main() {
    coroutineStart()
}
```

协程采用缺省的启动器, 当父协程执行完1、3后就会调度子协程执行2。

## ATOMIC

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
```

```
4
5  /**
6   * ATOMIC启动模式，立即等待被调度执行，并且开始执行前无法被取消，直到执行完毕或者遇到第一个挂起点。
7   *
8   * 该示例可以看出在同样经过cancel操作后，atomic协程依旧会被启动，而其它则不会启动了
9   * 打印出：
10  * 1
11  * 3
12  * atomic run
13  */
14 fun coroutineStart() = runBlocking {
15     println("1")
16     val job = launch(start = CoroutineStart.ATOMIC) {
17         println("atomic run")
18     }
19     job.cancel()
20     println("3")
21 }
22
23 fun main() {
24     coroutineStart()
25 }
```

```
1  import kotlinx.coroutines.*
2
3  /**
4   *
5   * 该示例演示atomic被cancel后遇到第一个挂起点取消运行的效果
6   *
7   * 打印出：
8   *
9   * 1
10  * 2
11  * atomic run
12  * 3
13  */
14 fun coroutineStart() = runBlocking {
```

```
    println("1")
    val job = launch(start = CoroutineStart.ATOMIC) {
        println("atomic run")
        //遇到了挂起点，但是cancel发生在delay之前，因此可以正常打印出后面的"atomic end"。
        delay(3000)
        println("atomic end")
    }
    job.cancel()
    println("2")
    delay(5000)
    println("3")
}

fun main() {
    coroutineStart()
```

```
import kotlinx.coroutines.*

/**
 * 该例子可以看出，在未运行前，default,lazy可以被cancel取消，
 * unDidpatcher因为会立即在当前线程执行，所以该例子中的cancel本身没啥意义了
 *
 * 输出:
 * 1
 * unDispatcherJob run
 * 2
 * atomic run
 */
fun coroutineStart() = runBlocking {
    println("1")
    val job = launch(start = CoroutineStart.ATOMIC) {
        println("atomic run")
    }
    job.cancel()
    val defaultJob = launch(start = CoroutineStart.DEFAULT) {
        println("default run")
    }
    defaultJob.cancel()
```

```
    val lazyJob = launch(start = CoroutineStart.LAZY) {
        println("lazyJob run")
    }
    lazyJob.start()
    lazyJob.cancel()
    val unDispatcherJob = launch(start =
CoroutineStart.UNDISPATCHED) {
        println("unDispatcherJob run")
    }
    unDispatcherJob.cancel()
    println("2")
}

fun main() {
    coroutineStart()
}
```

## LAZY

```
package com.example.coroutinelearning

import kotlinx.coroutines.CoroutineStart
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

class CoroutineLearning{
    fun testCoroutineLazyStart() = runBlocking {
        println("1")
        val job  = launch(start = CoroutineStart.LAZY) {
            println("2")
        }
        println("3")
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineLazyStart()
    }
```

```
}
```

运行上面的代码可以发现，代码输出1和3之后就会一直不退出，这是因为继承`runBlocking`协程上下文的协程一直没有执行完毕，所以程序一直不会退出。

如果换成`GlobalScope.launch`并使用`yield`让渡执行权，输出的结果就是：

```
package com.example.coroutinelearning

import android.provider.Settings
import kotlinx.coroutines.*

class CoroutineLearning{
    fun testCoroutineLazyStart() = runBlocking {
        println("1")
        val job  = GlobalScope.launch(start = CoroutineStart.LAZY)
{
            println("2")
        }
        yield()
        println("3")
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineLazyStart()
    }
}
```

内部创建的协程依旧没有执行，所以需要显式地调用`start`或者`join`来执行对应的协程体。

```
package com.example.coroutinelearning

import kotlinx.coroutines.CoroutineStart
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

```

```
class CoroutineLearning{
    fun testCoroutineLazyStart() = runBlocking {
        println("1")
        val job  = launch(start = CoroutineStart.LAZY) {
            println("2")
        }
        job.start()
        println("3")
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineLazyStart()
    }
}
```

执行结果如下：

```
1
3
2
```

或者使用`join`来插入到父协程的执行过程中：

```
package com.example.coroutinelearning

import kotlinx.coroutines.CoroutineStart
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

class CoroutineLearning{
    fun testCoroutineLazyStart() = runBlocking {
        println("1")
        val job  = launch(start = CoroutineStart.LAZY) {
            println("2")
        }
        job.join()
        println("3")
```

```
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineLazyStart()
    }
}
```

```
1
2
3
```

## UNDISPATCHED

协程创建后立即在当前函数调用栈中执行，直到遇到第一个真正挂起的点。UNDISPATCHED启动模式，立即运行该协程体内容（相比其它启动方式少了等待过程）。

```
package com.example.coroutinelearning

import kotlinx.coroutines.CoroutineStart
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

class CoroutineLearning{
    fun testCoroutineUndispatchedStart() = runBlocking {
        println("1")
        val job  = launch(start = CoroutineStart.UNDISPATCHED) {
            println("2")
        }
        println("3")
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineUndispatchedStart()
```

```
    }
}
```

输出是：

```
1
2
3
```

2之所以会在3之前打印，是因为我们使用了UNDISPATCHED启动模式，一旦创建协程完毕，就会立刻在当前的调用栈中被执行。

如果此时打印出内部的协程的线程环境，可以发现：

```
package com.example.coroutinelearning

import kotlinx.coroutines.CoroutineStart
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

class CoroutineLearning{
    fun testCoroutineUndispatchedStart() = runBlocking {
        println("1")
        val job  = launch(context = Dispatchers.IO, start =
CoroutineStart.UNDISPATCHED) {
            println("2")
            println(Thread.currentThread())
        }
        println("3")
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineUndispatchedStart()
    }
}
```

```
1
2
Thread[main,5,main]
3
```

可以发现，内部的协程也是在主线程中运行的，哪怕我们在启动的时候明确说明了调度器是IO调度器，这说明，UNDISPATCHED启动模式会直接使用当前的父协程的线程环境，所以协程体会直接在当前的函数栈（上下文）中执行。上面的`rubBlocking`是在主线程环境中运行的，那么创建的协程也是在主线程环境下运行。

如果换成其他的启动模式，则相应的线程环境也会被更改，如：

```
package com.example.coroutinelearning

import kotlinx.coroutines.CoroutineStart
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

class CoroutineLearning{
    fun testCoroutineUndispatchedStart() = runBlocking {
        println("1")
        val job  = launch(context = Dispatchers.IO, start =
CoroutineStart.DEFAULT) {
            println("2")
            println(Thread.currentThread())
        }
        println("3")
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineUndispatchedStart()
    }
}
```

这里会输出： Thread[DefaultDispatcher-worker-1,5,main]，因为启动模式是默认的启动模式，因此会在新的线程上运行。

# 协程的作用域构建器

## coroutineScope与runBlocking

- runBlocking是常规函数，而coroutineScope是挂起函数。
- 它们都会等待其协程体以及所有子协程结束，主要区别在于runBlocking方法会阻塞当前线程来等待，而coroutineScope只是挂起，会释放底层线程用于其他用途。

## coroutineScope和survivorScope

- coroutineScope: 一个协程失败了，所有其他兄弟协程也会被取消。
- supervisorScope： 一个协程失败了，不会影响其他兄弟协程。

如果有两个兄弟协程，那么彼此之间相互独立，正常情况下都可以完成任务：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning{
    fun testSiblingCoroutines() = runBlocking {
        coroutineScope {
            val job1 = launch {
                delay(400)
                println("Job1 finished")
            }

            val job2 = launch {
                delay(200)
                println("Job2 finished")
            }
        }
    }
}

```

```
fun main() {
    CoroutineLearning().apply {
        testSiblingCoroutines()
    }
}
```

```
Job2 finished
Job1 finished
```

之所以job2先结束，是因为job2的等待时间更短，只等待了200毫秒。

当给job2加上一个认为抛出的异常时：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning{
    fun testSiblingCoroutines() = runBlocking {
        coroutineScope {
            val job1 = launch {
                delay(400)
                println("Job1 finished")
            }

            val job2 = launch {
                delay(200)
                println("Job2 finished")
                throw IllegalArgumentException()
            }
        }
    }
}

fun main() {
    CoroutineLearning().apply {
        testSiblingCoroutines()
    }
}
```

```
Job2 finished
Exception in thread "main" java.lang.IllegalArgumentException
    at com.example.coroutinelearning.CoroutineLearning$testSiblingCoroutines$1$1$job2$1.invokeSuspend(CoroutineLeaning.kt:16)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTaskKt.resume(DispatchedTask.kt:234)
    at kotlinx.coroutines.DispatchedTaskKt.dispatch(DispatchedTask.kt:166)
    at kotlinx.coroutines.CancellableContinuationImpl.dispatchResume(CancellableContinuationImpl.kt:369)
    at kotlinx.coroutines.CancellableContinuationImpl.resumeImpl(CancellableContinuationImpl.kt:403)
    at kotlinx.coroutines.CancellableContinuationImpl.resumeImpl$default(CancellableContinuationImpl.kt:395)
    at kotlinx.coroutines.CancellableContinuationImpl.resumeUndispatched(CancellableContinuationImpl.kt:491)
    at kotlinx.coroutines.EventLoopImplBase$DelayedResumeTask.run(EventLoop.common.kt:489)
    at kotlinx.coroutines.EventLoopImplBase.processNextEvent(EventLoop.common.kt:274)
    at kotlinx.coroutines.BlockingCoroutine.joinBlocking(Builders.kt:84)
    at kotlinx.coroutines.BuildersKt__BuildersKt.runBlocking(Builders.kt:59)
    at kotlinx.coroutines.BuildersKt.runBlocking(Unknown Source)
```

```
    at kotlinx.coroutines.BuildersKt__BuildersKt.runBlocking$default(Builders.kt:38)
    at kotlinx.coroutines.BuildersKt.runBlocking$default(Unknown Source)
    at com.example.coroutinelearning.CoroutineLearning.testSiblingCoroutines(CoroutineLeaning.kt:6)
    at com.example.coroutinelearning.CoroutineLeaningKt.main(CoroutineLeaning.kt:24)
    at com.example.coroutinelearning.CoroutineLeaningKt.main(CoroutineLeaning.kt)
```

可以看到，job1的根本就没有进行输出，这是因为coroutineScope内，如果有一个协程出现了异常失败了，那么剩下的协程也会被取消。

如果将coroutineScope替换成survivorScope：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning{
    fun testSiblingCoroutines() = runBlocking {
        supervisorScope {
            val job1 = launch {
                delay(400)
                println("Job1 finished")
            }

            val job2 = launch {
                delay(200)
                println("Job2 finished")
                throw IllegalArgumentException()
            }
        }
    }
```

```
}

fun main() {
    CoroutineLearning().apply {
        testSiblingCoroutines()
    }
}
```

```
Job2 finished
Exception in thread "main" java.lang.IllegalArgumentException
    at com.example.coroutinelearning.CoroutineLearning$testSiblingCoroutines$1$1$job2$1.invokeSuspend(CoroutineLeaning.kt:16)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTaskKt.resume(DispatchedTask.kt:234)
    at kotlinx.coroutines.DispatchedTaskKt.dispatch(DispatchedTask.kt:166)
    at kotlinx.coroutines.CancellableContinuationImpl.dispatchResume(CancellableContinuationImpl.kt:369)
    at kotlinx.coroutines.CancellableContinuationImpl.resumeImpl(CancellableContinuationImpl.kt:403)
    at kotlinx.coroutines.CancellableContinuationImpl.resumeImpl$default(CancellableContinuationImpl.kt:395)
    at kotlinx.coroutines.CancellableContinuationImpl.resumeUndispatched(CancellableContinuationImpl.kt:491)
    at kotlinx.coroutines.EventLoopImplBase$DelayedResumeTask.run(EventLoop.common.kt:489)
    at kotlinx.coroutines.EventLoopImplBase.processNextEvent(EventLoop.common.kt:274)
```

```
13      at
    kotlinx.coroutines.BlockingCoroutine.joinBlocking(Builders.kt:84)
14      at
    kotlinx.coroutines.BuildersKt__BuildersKt.runBlocking(Builders.kt:
    59)
15      at kotlinx.coroutines.BuildersKt.runBlocking(Unknown Source)
16      at
    kotlinx.coroutines.BuildersKt__BuildersKt.runBlocking$default(Buil
    ders.kt:38)
17      at kotlinx.coroutines.BuildersKt.runBlocking$default(Unknown
    Source)
18      at
    com.example.coroutinelearning.CoroutineLearning.testSiblingCorouti
    nes(CoroutineLeaning.kt:6)
19      at
    com.example.coroutinelearning.CoroutineLeaningKt.main(CoroutineLea
    ning.kt:24)
20      at
    com.example.coroutinelearning.CoroutineLeaningKt.main(CoroutineLea
    ning.kt)
21  Job1 finished
```

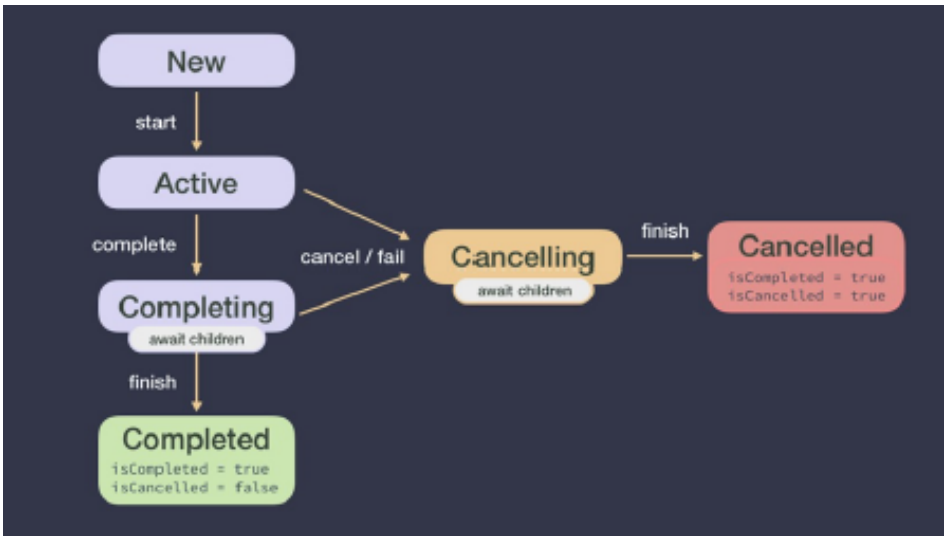
虽然也有异常抛出，但是job1也顺利的完成了执行，可以正确输出。

以上就是coroutineScope和survivorScope的区别。

# Job对象和Job的生命周期

对于每一个创建的协程（通过launch或者async)，会返回一个Job实例，该实例是协程的唯一标示，并且负责管理协程的生命周期。

一个任务可以包含一系列状态：新创建(New)、活跃(Active)、完成中(Completing）、已完成(Completed)、取消中(Cancelling)和已取消(Cancelled)。虽然我们无法直接访问这些状态，但是我们可以访问Job的属性：isCancelled和isCompleted。




## 创建和取消协程作用域

### 创建

事实上，我们可以创建一个自己的协程作用域。如下：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlin.coroutines.EmptyCoroutineContext

class CoroutineLearning{
    fun testOwnCoroutineScope() = runBlocking {
        val scope = CoroutineScope(EmptyCoroutineContext)

        scope.launch {
            delay(100)
            println("Job1 finished")
        }

        scope.launch {
            delay(100)
            println("Job2 finished")
        }
    }
}

fun main() {
```

```
    CoroutineLearning().apply {
        testOwnCoroutineScope()
    }
}
```

运行之后，发现没有任何输出，这说明在代码中创建的两个协程没有得到调度和执行。这是因为在上面的代码中，手动创建的协程作用域运行的协程上下文是一个空的协程上下文，没有继承runBlocking的协程上下文，因此，runBlocking不会等待这两个协程运行结束就直接结束运行了。

为了解决这个问题，有两种方式挂起

- runBlocking协程体：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlin.coroutines.EmptyCoroutineContext

class CoroutineLearning{
    fun testOwnCoroutineScope() = runBlocking {
        val scope = CoroutineScope(EmptyCoroutineContext)

        scope.launch {
            delay(100)
            println("Job1 finished")
        }

        scope.launch {
            delay(100)
            println("Job2 finished")
        }

        delay(1000)
    }
}

fun main() {
    CoroutineLearning().apply {
```

```
        testOwnCoroutineScope()
    }
}
```

- 让手动创建的协程作用域继承runBlocking的上下文，这样，runBlocking协程体就会等待里面的两个协程执行完成之后再退出。

```
package com.example.coroutinelearning

import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

class CoroutineLearning{
    fun testOwnCoroutineScope() = runBlocking {
        val scope = CoroutineScope(this.coroutineContext)

        scope.launch {
            delay(100)
            println("Job1 finished")
        }

        scope.launch {
            delay(100)
            println("Job2 finished")
        }
    }
}

fun main() {
    CoroutineLearning().apply {
        testOwnCoroutineScope()
    }
}
```

上面的两种代码都可以让两个协程体正确输出，但是很明显，第二种方式是更加合适的选择。

**取消**

- 取消作用域会取消它的子协程。
- 被取消的子协程并不会影响其余兄弟协程。兄弟协程之间是相互独立的。
- 协程通过抛出一个特殊的异常CancellationException来处理取消操作。
- 所有kotlinx.coroutines中的挂起函数(withContext、delay等）都是可取消的。

```
package com.example.coroutinelearning

import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

class CoroutineLearning{
    fun testCoroutineCancel() = runBlocking {
        val scope = CoroutineScope(this.coroutineContext)

        val job = scope.launch {
            delay(1000)
            println("Job finished")
        }

        delay(100)
        job.cancel()
        job.join()
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineCancel()
    }
}
```

上面的代码是不会有任何的输出的，因为我们在子协程有打印输出之前就将对应的Job取消掉了，所有根本不会有输出。

但是取消之后按照上面的说明应当会有一个异常抛出，实际上也没有。这是因为即使这个异常抛出了，协程也会被认为是一个正常的状态，该异常被静默处理掉了，如果想要手动的去处理该异常，可以对协程体加上try-catch的异常捕捉块。

```
package com.example.coroutinelearning

import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

class CoroutineLearning{
    fun testCoroutineCancel() = runBlocking {
        val scope = CoroutineScope(this.coroutineContext)

        val job = scope.launch {
            try {
                delay(1000)
                println("Job finished")
            } catch (e:Exception) {
                println("Caught Exception: ${e.message}")
            }
        }

        delay(100)
        job.cancel()
        job.join()
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineCancel()
    }
}
```

```
Caught Exception: StandaloneCoroutine was cancelled
```

观察cancel函数，可以发现可以带有一个自定义的CancellationException的参数，这个参数默认是空的。如果这个参数被指定了，那么这个异常被捕获的时候就会被使用。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning{
    fun testCoroutineCancel() = runBlocking {
        val scope = CoroutineScope(this.coroutineContext)

        val job = scope.launch {
            try {
                delay(1000)
                println("Job finished")
            } catch (e:Exception) {
                println("Caught Exception: ${e.message}")
            }
        }

        delay(100)
        job.cancel(CancellationException("这个协程被取消了"))
        job.join()
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineCancel()
    }
}
```

```
Caught Exception: 这个协程被取消了
```

## CPU密集型任务的取消

isActive是一个可以被使用在CoroutineScope中的扩展属性，检查Job是否处于活跃状态。

ensureActive()，如果job处于非活跃状态，这个方法会立即抛出异常。

yield函数会检查所在协程的状态，如果已经取消，则抛出CancellationException予以响应。此外，它还会尝试出让线程的执行权，给其他协程提供执行机会。

### isActive

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning {
    fun testCoroutineIsActive() = runBlocking {

        val startTime = System.currentTimeMillis()

        // CPU密集型任务在Default调度器中运行，在主线程中通过isActive取消不了
        val job = launch(Dispatchers.Default) {
            var nextPrintTime = startTime
            var i = 0
            while (i < 5) {
                // 每秒打印消息两次
                if (System.currentTimeMillis() >= nextPrintTime) {
                    println("Job: I'm sleeping ${i}...")
                    i += 1
                    nextPrintTime += 500L
                }
            }
        }

        delay(1300)
        println("main: I'm tired of waiting...")
        job.cancelAndJoin() // 取消一个作业并且等待它结束
        println("main: Now I can quit...")
```

```
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineIsActive()
    }
}
```

对于上面的这种循环执行的代码，一直会依赖于CPU进行时间的比较，循环结束的判读等，上面的代码执行之后，得到的结果如下：

```
Job: I'm sleeping 0...
Job: I'm sleeping 1...
Job: I'm sleeping 2...
main: I'm tired of waiting...
Job: I'm sleeping 3...
Job: I'm sleeping 4...
main: Now I can quit...
```

协程并没有被取消，依旧走完了全部的代码流程。

这实际上是一种Kotlin协程对于CPU密集型任务的一种保护机制，原因在于CPU密集型任务往往会保存大量的临时变量，比如数组，各种对象等等，如果直接取消协程操作，这些对象和变量会被丢弃，从而造成资源的浪费以及可能会引起的不安全的操作。

实际上，一个Job被cancel之后，会将Job内的状态从isActive为true的状态转变为false的状态，因此，可以利用这个状态位来判断CPU密集型任务是否被取消。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning {
    fun testCoroutineIsActive() = runBlocking {
        //CPU密集型任务取消
        //isActive是一个可以被使用在CoroutineScope中的扩展属性，检查Job是否处于活跃状态。
        val startTime = System.currentTimeMillis()
```

```
        //CPU密集型任务在Default调度器中运行，在主线程中通过isActive取消不了
        val job = launch(Dispatchers.Default) {
            var nextPrintTime = startTime
            var i = 0
            while (i < 5 && isActive) {
                // 每秒打印消息两次
                if (System.currentTimeMillis() >= nextPrintTime) {
                    println("Job: I'm sleeping ${i}...")
                    i += 1
                    nextPrintTime += 500L
                }
            }
        }

        delay(1300)
        println("main: I'm tired of waiting...")
        job.cancelAndJoin() // 取消一个作业并且等待它结束
        println("main: Now I can quit...")
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineIsActive()
    }
}
```

运行结果如下：

```
Job: I'm sleeping 0...
Job: I'm sleeping 1...
Job: I'm sleeping 2...
main: I'm tired of waiting...
main: Now I can quit...
```

可以看到，该协程任务被正确取消了。

### ensureActive

其实也可以利用ensureActive函数来进行任务的取消，如：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning {
    fun testCoroutineEnsureActive() = runBlocking {
        //CPU密集型任务取消
        val startTime = System.currentTimeMillis()
        //CPU密集型任务在Default调度器中运行
        val job = launch(Dispatchers.Default) {
            var nextPrintTime = startTime
            var i = 0
            while (i < 5) {
                ensureActive()
                // 每秒打印消息两次
                if (System.currentTimeMillis() >= nextPrintTime) {
                    println("Job: I'm sleeping ${i}...")
                    i += 1
                    nextPrintTime += 500L
                }
            }
        }

        delay(1300)
        println("main: I'm tired of waiting...")
        job.cancelAndJoin() // 取消一个作业并且等待它结束
        println("main: Now I can quit...")
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineEnsureActive()
    }
}
```

代码输出如下：

```
Job: I'm sleeping 0...
Job: I'm sleeping 1...
Job: I'm sleeping 2...
main: I'm tired of waiting...
main: Now I can quit...
```

可以看到上面的任务也已经被正常取消掉了。

观察ensureActive这个函数，可以发现本质上调用的是协程上下文的ensureActive函数：

```
/**
 * Ensures that current scope is [active] [CoroutineScope.isActive].
 *
 * If the job is no longer active, throws [CancellationException].
 * If the job was cancelled, thrown exception contains the original cancellation cause.
 * This function does not do anything if there is no [Job] in the scope's [coroutineContext][CoroutineScope.coroutineContext].
 *
 * This method is a drop-in replacement for the following code, but with more precise exception:
 * ```
 * if (!isActive) {
 *     throw CancellationException()
 * }
 * ```
 *
 * @see CoroutineContext.ensureActive
 */
public fun CoroutineScope.ensureActive(): Unit = coroutineContext.ensureActive()
```

然后再获取对应的Job对象，对Job对象应用ensureActive函数。

```
/**
```

```
 * Ensures that job in the current context is [active] [Job.isActive].
 *
 * If the job is no longer active, throws [CancellationException].
 * If the job was cancelled, thrown exception contains the original cancellation cause.
 * This function does not do anything if there is no [Job] in the context, since such a coroutine cannot be cancelled.
 *
 * This method is a drop-in replacement for the following code, but with more precise exception:
 * ```
 * if (!isActive) {
 *     throw CancellationException()
 * }
 * ```
 */
public fun CoroutineContext.ensureActive() {
    get(Job)?.ensureActive()
}
```

最后可以看到本质上ensureActive也是利用了isActive的这个标志位来进行判断协程的状态，但是如果isActive返回的是false，那么该函数会抛出一个CancellationException的异常。这个异常会被协程静默处理掉，因此可以做到安全退出协程。想要捕获该异常，需要使用try-catch代码块。

### yield

yield函数会检查所在协程的状态，如果已经取消，则抛出CancellationException予以响应。此外，它还会尝试出让线程的执行权，给其他协程提供执行机会。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning {
    fun testCoroutineYield() = runBlocking {
        // CPU密集型任务取消
```

```
        // yield函数会检查所在协程的状态，如果已经取消，则抛出CancellationException予以响应。
        // 此外，它还会尝试出让线程的执行权，给其他协程提供执行机会。
        // 如果要处理的任务属于：
        // 1) CPU 密集型，2) 可能会耗尽线程池资源，3) 需要在不向线程池中添加更多线程的前提下允许线程处理其他任务，那么请使用 yield()。
        val startTime = System.currentTimeMillis()
        // CPU密集型任务在Default调度器中运行
        val job = launch(Dispatchers.Default) {
            var nextPrintTime = startTime
            var i = 0
            while (i < 5) {
                yield()
                // 每秒打印消息两次
                if (System.currentTimeMillis() >= nextPrintTime) {
                    println("Job: I'm sleeping ${i}...")
                    i += 1
                    nextPrintTime += 500L
                }
            }
        }

        delay(1300)
        println("main: I'm tired of waiting...")
        job.cancelAndJoin() // 取消一个作业并且等待它结束
        println("main: Now I can quit...")
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineYield()
    }
}
```

代码运行的结果和上面的一致。

## 协程取消的副作用

如果在协程中读取了系统资源时，如果协程被取消，那么有可能会出现资源没有办法释放的情况，这种就会导致系统资源的浪费。

在finally中释放资源。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning {
    fun testCoroutineFinally() = runBlocking {

        val job = launch {
            try {
                repeat(100) { i ->
                    println("Job: I'm sleeping $i...")
                    delay(500)
                }
            } finally {
                println("Job: I'm running finally...")
            }
        }

        delay(1300) // 延迟一段时间
        println("main: I'm tired of waiting...")
        job.cancelAndJoin() // 取消一个作业并且等待它结束
        println("main: Now I can quit...")
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineFinally()
    }
}
```

运行结果如下：

```
Job: I'm sleeping 0...
Job: I'm sleeping 1...
Job: I'm sleeping 2...
main: I'm tired of waiting...
Job: I'm running finally...
main: Now I can quit...
```

可以看到，协程取消之后正确执行了finally里面的代码。

**use标准库函数**

该函数只能被实现了Closeable的对象使用，程序结束的时候会自动调用close 方法，适合文件对象。

假设有一个名为text.txt的文件，该文件位于C盘的根目录下，里面有一段文本内容。

那么标准的读取文件的方法是：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import java.io.BufferedReader
import java.io.FileReader

class CoroutineLearning {
    fun testCoroutineUseFunction() = runBlocking {
        //读取文件方式一
        val br = BufferedReader(FileReader("C:\\text.txt")) //打开文件读取
        with(br) { //对br中的属性和方法直接进行操作
            var line: String?
            while (true) {
                line = readLine() ?: break //读取一行数据，若为空则退出循环
                println(line) // 打印读取的数据
            }
            close()
        }
```

```
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineUseFunction()
    }
}
```

上面的代码执行的结果就是文件中的每一行的内容。

那么BufferReader继承了Closeable的接口，因此可以使用use标准库函数。这个库函数会在代码执行的最后自动调用close函数，避免了打开资源而忘记释放的问题。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import java.io.BufferedReader
import java.io.FileReader

class CoroutineLearning {
    fun testCoroutineUseFunction() = runBlocking {
        // 读取文件方式二，会自动调用close函数进行文件关闭
        BufferedReader(FileReader("C:\\text.txt")).use {
            var line: String?
            while (true) {
                line = readLine() ?: break //读取一行数据，若为空则退出循环
                println(line) // 打印读取的数据
            }
        }
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineUseFunction()
    }
}
```

## 不能被取消的任务

处于取消中状态的协程不能够挂起（运行不能取消的代码），当协程被取消后需要调用挂起函数，我们需要将清理任务的代码放置于NonCancellable CoroutineContext中。这样会挂起运行中的代码，并保持协程的取消中状态直到任务处理完成。

例如，有以下的代码：

```
package com.example.coroutinelearning

import kotlinx.coroutines.cancelAndJoin
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

class CoroutineLearning {
    fun testCoroutineCancellation() = runBlocking {
        val job = launch {
            try {
                repeat(100) { i ->
                    println("Job: I'm sleeping $i ...")
                    delay(500)
                }
            } finally {
                println("Job: I'm running finally")
                delay(1000L)
                println("Job: And I've just delayed for 1 sec
because I'm non-cancellable")
            }
        }

        delay(1300L) // 延迟一段时间
        println("main: I'm tired of waiting!")
        job.cancelAndJoin() // 取消该作业并等待它结束
        println("main: Now I can quit.")
    }
}

fun main() {
```

```
    CoroutineLearning().apply {
        testCoroutineCancellation()
    }
}
```

代码运行结果如下：

```
Job: I'm sleeping 0 ...
Job: I'm sleeping 1 ...
Job: I'm sleeping 2 ...
main: I'm tired of waiting!
Job: I'm running finally
main: Now I can quit.
```

经过1300毫秒的延迟之后，协程被取消，协程被取消之后会抛出CancellationException的异常，经过捕捉之后会进入finally的代码块。但是在finally中我们使用了delay这个挂起函数，此时协程已经是一个被取消的状态了，因此后续的代码输出就无法进行，从而在上面的输出中无法找到相对应的文本。

此时，可以使用NonCancellable这个CoroutineContext（协程上下文环境）。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning {
    fun testCoroutineCancellation() = runBlocking {
        val job = launch {
            try {
                repeat(100) { i ->
                    println("Job: I'm sleeping $i ...")
                    delay(500)
                }
            } finally {
                withContext(NonCancellable) {
                    println("Job: I'm running finally")
                    delay(1000L)
                    println("Job: And I've just delayed for 1 sec
because I'm non-cancellable")
```

```
                }
            }
        }

        delay(1300L) // 延迟一段时间
        println("main: I'm tired of waiting!")
        job.cancelAndJoin() // 取消该作业并等待它结束
        println("main: Now I can quit.")
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineCancellation()
    }
}
```

输出如下：

```
Job: I'm sleeping 0 ...
Job: I'm sleeping 1 ...
Job: I'm sleeping 2 ...
main: I'm tired of waiting!
Job: I'm running finally
Job: And I've just delayed for 1 sec because I'm non-cancellable
main: Now I can quit.
```

在这个上下文环境中，可以调用挂起函数并保持协程的取消中状态直到任务处理完成。因此， finally中的代码可以完整的执行完毕并输出。

## 超时任务

很多情况下取消一个协程的理由是它有可能超时。利用withTimeout可以调用一个可能会超时的任务。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

```

```
class CoroutineLearning {
    fun testCoroutineTimeout() = runBlocking {
        withTimeout(1300) {
            repeat(100) { i ->
                println("I'm sleeping $i ...")
                delay(500)
            }
        }
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineTimeout()
    }
}
```

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
Exception in thread "main"
kotlinx.coroutines.TimeoutCancellationException: Timed out waiting
for 1300 ms
    at
kotlinx.coroutines.TimeoutKt.TimeoutCancellationException(Timeout.
kt:186)
    at kotlinx.coroutines.TimeoutCoroutine.run(Timeout.kt:156)
    at
kotlinx.coroutines.EventLoopImplBase$DelayedRunnableTask.run(Event
Loop.common.kt:497)
    at
kotlinx.coroutines.EventLoopImplBase.processNextEvent(EventLoop.co
mmon.kt:274)
    at
kotlinx.coroutines.DefaultExecutor.run(DefaultExecutor.kt:69)
    at java.lang.Thread.run(Thread.java:748)
```

可以看到，代码中重复执行了1000此挂起500毫秒的操作，很明显，这个操作无法在1300毫秒之内完成。执行了1300毫秒之后，代码会被认为是超时，最后会抛出一个超时的异常TimeoutCancellationException，该异常并不会被协程静默处理，而是直接抛了出来。
但是很多情况下并不希望直接抛出异常，而是返回一个空值，比如网络请求的时候，返回一个空值会比直接抛出异常更友好一些。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning {
    fun testCoroutineTimeout() = runBlocking {
        val result = withTimeoutOrNull(1300) {
            repeat(100) { i ->
                println("I'm sleeping $i ...")
                delay(500)
            }
        }
        println("Result is $result")
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineTimeout()
    }
}
```

输出如下：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
Result is null
```

可以看到，当执行的任务超时的时候，协程会直接返回空值null，从而避免了异常的抛出。

# 协程的上下文和异常处理

## 协程上下文的定义

CoroutineContext是一组用于定义协程行为的元素。它由如下几项构成：

- Job：控制协程的生命周期
- CoroutineDispatcher：向合适的线程分发任务
- CoroutineName：协程的名称，调试的时候很有用
- CoroutineExceptionHandler：处理未被捕捉的异常

## 协程上下文的元素

有时我们需要在协程上下文中定义多个元素。我们可以使用+操作符来实现。比如说，我们可以显式指定一个调度器来启动协程并且同时显式指定一个命名：

```
1  package com.example.coroutinelearning
2
3  import kotlinx.coroutines.*
4
5  class CoroutineLearning {
6      fun testCoroutineContext() = runBlocking {
7          launch(Dispatchers.Default +
   CoroutineName("my_coroutine")) {
8              println("I'm working in thread
   ${Thread.currentThread().name}, ${this.coroutineContext}")
9          }
10     }
11 }
12
13 fun main() {
14     CoroutineLearning().apply {
15         testCoroutineContext()
16     }
17 }
```

输出如下：

```
I'm working in thread DefaultDispatcher-worker-1,
[CoroutineName(my_coroutine), StandaloneCoroutine{Active}@c195565,
Dispatchers.Default]
```

可以看到，输出有正确的自定的协程上下文的名称。

因为CoroutineContext实现了plus的运算符重载，因此可以直接将两个CoroutineContext进行相加。需要注意的是，Dispatchers.Main等调度器也是一个CoroutineContext，因为这些调度器也实现了CoroutineContext这个接口。包括一个Job也实现了CoroutineContext这个接口。

## 协程上下文的继承

对于新创建的协程，它的CoroutineContext会包含一个全新的Job实例，它会帮助我们控制协程的生命周期。而**剩下的元素会从CoroutineContext的父类继承**，该父类可能是另外一个协程或者创建该协程的CoroutineScope。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning {
    fun testCoroutineContext() = runBlocking {
        val scope = CoroutineScope(Job() + Dispatchers.IO +
CoroutineName("test"))

        val job = scope.launch {
            println("1. ${coroutineContext[Job]}
${Thread.currentThread().name}, ${this.coroutineContext}")
            val result = async {
                println("2. ${coroutineContext[Job]}
${Thread.currentThread().name}, ${this.coroutineContext}")
                "OK"
            }.await()
        }
        job.join()
    }
}

```

```
fun main() {
    CoroutineLearning().apply {
        testCoroutineContext()
    }
}
```

代码输出如下：

```
1. StandaloneCoroutine{Active}@270732dc DefaultDispatcher-worker-1, [CoroutineName(test), StandaloneCoroutine{Active}@270732dc, Dispatchers.IO]
2. DeferredCoroutine{Active}@75c42cb3 DefaultDispatcher-worker-3, [CoroutineName(test), DeferredCoroutine{Active}@75c42cb3, Dispatchers.IO]
```

从上面的代码中可以看出，首先创建了一个新的协程作用域，名字是scope，然后从这个scope中开启了一个新的协程job，很明显，job这个协程会继承scope中的上下文环境，包括调度器和协程名称等信息。所以在job协程体中输出的协程上下文是scope的上下文环境，协程的名字也是test。

result这个协程使用的默认的job的上下文环境，所以自然也会继承job的协程上下文环境，因此输出的协程名称也是test。这里隐式地调用了this.launch这个方法。

## 协程上下文的继承公式

**协程的上下文 = 默认值 + 继承的CoroutineContext + 参数**

- 一些元素包含默认值：Dispatchers.Default是默认的CoroutineDispatcher，以
  及"coroutine"作为默认的CoroutineName。
- 继承的CoroutineContext是CoroutineScope或者其父协程的CoroutineContext
- 传入协程构建起的参数的优先级高于继承的上下文参数，因此会覆盖对应的参数值。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import okhttp3.Dispatcher
```

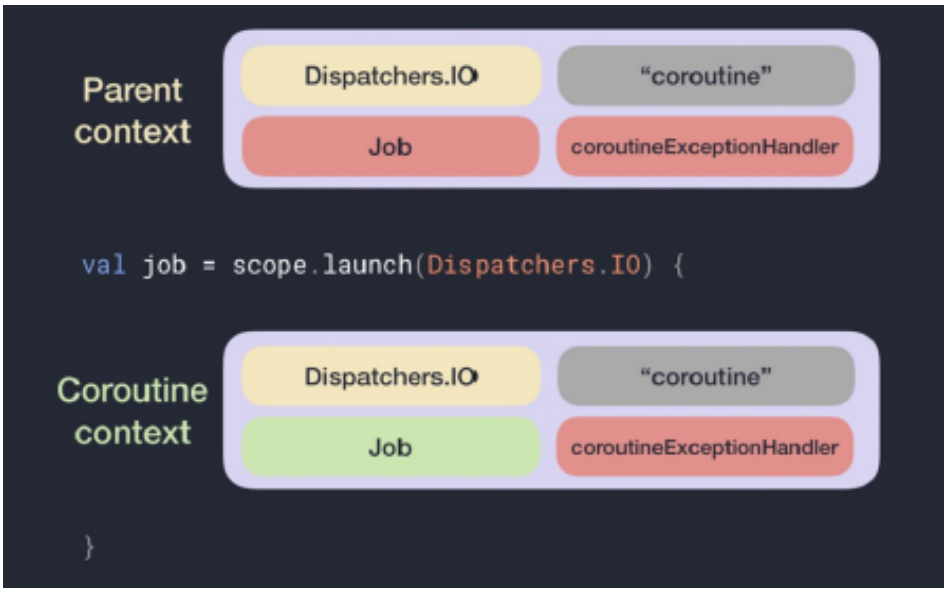
```
5
6  class CoroutineLearning {
7      fun testCoroutineContext() = runBlocking {
8
9          // 协程的异常处理器
10         val coroutineExceptionHandler = CoroutineExceptionHandler
   { _, exception ->
11             println("Caught $exception")
12         }
13         val scope = CoroutineScope(Job() + Dispatchers.Default +
   coroutineExceptionHandler)
14         val job1 = scope.launch(Dispatchers.IO) {
15             //新协程，由于制定了新的调度器，这个新的调度器会覆盖原来的从父协
   程中继承的调度器
16             println("1. " + this.coroutineContext)
17         }
18
19         val job2 = scope.launch {
20             //新协程，这里没有指定新的调度器，所以新的协程会直接继承父协程中
   的调度器
21             println("2. " + this.coroutineContext)
22         }
23         job1.join()
24         job2.join()
25     }
26 }
27
28 fun main() {
29     CoroutineLearning().apply {
30         testCoroutineContext()
31     }
32 }
```

上面的代码输出如下：

```
1. [com.example.coroutinelearning.CoroutineLearning$testCoroutineContext$1$invokeSuspend$$inlined$CoroutineExceptionHandler$1@2c5aecaf, StandaloneCoroutine{Active}@45980939, Dispatchers.IO]
2. [com.example.coroutinelearning.CoroutineLearning$testCoroutineContext$1$invokeSuspend$$inlined$CoroutineExceptionHandler$1@2c5aecaf, StandaloneCoroutine{Active}@115d70f4, Dispatchers.Default]
```

可以发现，第一行的输出已经不是使用的默认的调度器了，而是`Dispatchers.IO`。

最终的父级CoroutineContext会内含Dispatchers.IO而不是scope对象中的Dispatchers.Deault，因为它被协程构建器中的参数覆盖了。此外，注意一下父级CoroutineContext里的Job是scope对象的Job（红色），而新的Job实例（绿色）会赋值给新的协程的CoroutineContext。




# 协程的异常处理的必要性

当应用出现一些意外情况时，给用户提供合适的体验非常重要，一方面，目睹应用崩溃是个很糟糕的体验，另一方面，在用户操作失败时，也必须要能给出正确的提示信息。

协程构建器有两种形式：**自动传播异常（launch与actor），向用户暴露异常（async与produce）**当这些构建器用于创建一个根协程时（该协程不是另一个协程的子协程），前者这类构建器，异常会在它发生的第一时间被抛出，而后者则依赖用户来最终消费异常，例如通过await或receive。

首先看下面的两个异常的抛出：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import okhttp3.Dispatcher
import java.lang.ArithmeticException
import java.lang.IndexOutOfBoundsException

class CoroutineLearning {
    fun testCoroutineException() = runBlocking {
        val job = GlobalScope.launch {
            throw IndexOutOfBoundsException()
        }
        job.join()

        val deferred = GlobalScope.async {
            println("async")
            throw ArithmeticException()
            "OK"
        }
        deferred.await()
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineException()
    }
}
```

代码输出如下：

```
Exception in thread "DefaultDispatcher-worker-1"
java.lang.IndexOutOfBoundsException
    at
com.example.coroutinelearning.CoroutineLearning$testCoroutineExcep
tion$1$job$1.invokeSuspend(CoroutineLeaning.kt:11)
    at
kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(Con
tinuationImpl.kt:33)
    at
kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(Corouti
neScheduler.kt:571)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTas
k(CoroutineScheduler.kt:750)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(
CoroutineScheduler.kt:678)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(Corout
ineScheduler.kt:665)
async
Exception in thread "main" java.lang.ArithmeticException
    at
com.example.coroutinelearning.CoroutineLearning$testCoroutineExcep
tion$1$deferred$1.invokeSuspend(CoroutineLeaning.kt:17)
    at
kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(Con
tinuationImpl.kt:33)
    at
kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(Corouti
neScheduler.kt:571)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTas
k(CoroutineScheduler.kt:750)
```

```
16      at
    kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(
    CoroutineScheduler.kt:678)
17      at
    kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(Corout
    ineScheduler.kt:665)
```

可以看到两个异常都被抛出，且都没有被捕获。

对于launch函数启动的协程来说，需要在launch的协程体内进行异常的捕获，在join的时候进行异常捕获是无效的，如下：

```
1  package com.example.coroutinelearning
2
3  import kotlinx.coroutines.*
4  import okhttp3.Dispatcher
5  import java.lang.ArithmeticException
6  import java.lang.Exception
7  import java.lang.IndexOutOfBoundsException
8
9  class CoroutineLearning {
10     fun testCoroutineException() = runBlocking {
11         val job = GlobalScope.launch {
12             try {
13                 throw IndexOutOfBoundsException()
14             } catch (e: Exception) {
15                 println("1. Caught IndexOutOfBoundsException")
16             }
17         }
18         job.join()
19
20         val job2 = GlobalScope.launch {
21             throw IndexOutOfBoundsException()
22         }
23         try {
24             job2.join()
25         } catch (e: Exception) {
26             println("2. Caught IndexOutOfBoundsException")
27         }
```

```
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineException()
    }
}
```

输出结果如下：

```
1. Caught IndexOutOfBoundsException
Exception in thread "DefaultDispatcher-worker-1" java.lang.IndexOutOfBoundsException
    at com.example.coroutinelearning.CoroutineLearning$testCoroutineException$1$job2$1.invokeSuspend(CoroutineLeaning.kt:21)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(CoroutineScheduler.kt:571)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTask(CoroutineScheduler.kt:750)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(CoroutineScheduler.kt:678)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(CoroutineScheduler.kt:665)
```

可以看到，1处的异常被正确的捕获，2处的异常还是被抛了出来，并没有被捕获。

对于async函数，则和launch启动函数相反，需要在await的调用出进行异常捕获，如下：

```kotlin
package com.example.coroutinelearning

import kotlinx.coroutines.*
import okhttp3.Dispatcher
import java.lang.ArithmeticException
import java.lang.Exception
import java.lang.IndexOutOfBoundsException

class CoroutineLearning {
    fun testCoroutineException() = runBlocking {
        val job = GlobalScope.async {
            try {
                throw IndexOutOfBoundsException()
            } catch (e: Exception) {
                println("1. Caught IndexOutOfBoundsException")
            }
            "OK"
        }
        job.await()

        val job2 = GlobalScope.async {
            throw IndexOutOfBoundsException()
            "OK"
        }
        try {
            job2.await()
        } catch (e: Exception) {
            println("2. Caught IndexOutOfBoundsException")
        }
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineException()
    }
}
```

第一段代码就是普通的通过try-catch进行捕获，这个代码和协程没有关系。所以可以有正确的输出。

第二段代码直接在协程体内抛出异常，然后在await方法处进行异常捕获，因此也可以有正确的输出。

如果不调用await，异常是不会被抛出和捕获的。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import okhttp3.Dispatcher
import java.lang.ArithmeticException
import java.lang.Exception
import java.lang.IndexOutOfBoundsException

class CoroutineLearning {
    fun testCoroutineException() = runBlocking {
        // 3. 在协程体内直接抛出异常，不调用await方法，而是挂起外部协程1000毫秒。
        // 此时，这个异常不会向用户进行暴露，所以这段代码不会有输出
        val job2 = GlobalScope.async {
            throw IndexOutOfBoundsException()
            "OK"
        }
        delay(100)
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineException()
    }
}
```

上面的代码不会有任何输出，因为异常没有向用户暴露。

## 非根协程的异常

其他协程所创建的协程中，产生的异常总是会被传播。

如下的代码：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import okhttp3.Dispatcher
import java.lang.ArithmeticException
import java.lang.Exception
import java.lang.IndexOutOfBoundsException

class CoroutineLearning {
    fun testCoroutineException() = runBlocking {
        val scope = CoroutineScope(Job())
        val job = scope.launch {
            async {
                // 如果async抛出异常，launch就会立即抛出异常，而不会调用await方法。
                throw IllegalArgumentException()
            }
        }
        job.join()
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineException()
    }
}
```

输出结果是：

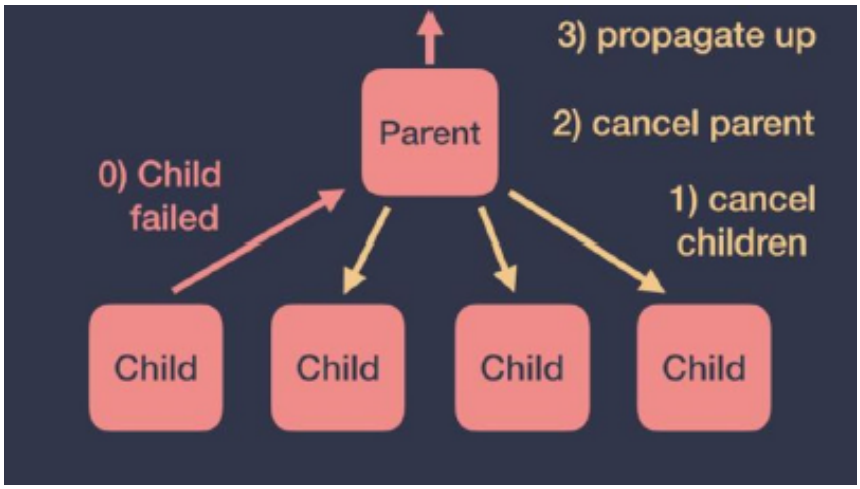
```
Exception in thread "DefaultDispatcher-worker-2"
java.lang.IllegalArgumentException
    at
com.example.coroutinelearning.CoroutineLearning$testCoroutineExcept
ion$1$job$1$1.invokeSuspend(CoroutineLeaning.kt:15)
    at
kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(Cont
inuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(Coroutin
eScheduler.kt:571)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTask
(CoroutineScheduler.kt:750)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(C
oroutineScheduler.kt:678)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(Corouti
neScheduler.kt:665)
```

可以看到，异常直接被抛出，并且没有调用对应的await方法。这是因为创建的async协程是launch内部的协程，不是一个根协程，因此抛出的异常会直接传递给外部的协程。

## 异常的传播特性

当一个协程由于一个异常而运行失败时，它会传播这个异常并传递给它的父级。接下来，父级会进行下面几步操作：

- 取消它自己的子级
- 取消它自己
- 将异常传播并传递给它的父级




但是有时候抛出异常之后，不希望影响其兄弟协程，因为他们可能在完成很重要的工作，所以这个时候只希望抛出异常的协程得到取消，其他的协程正常运行。

## SupervisorJob

使用SupervisorJob时，一个子协程的运行失败不会影响到其他子协程。SupervisorJob不会传播异常给它的父级，它会让子协程自己处理异常。

这种需求常见于在作用域内定义作业的UI组件，如果任何一个UI的子作业执行失败了，它并不总是有必要取消整个UI组件，但是如果UI组件被销毁了，由于它的结果不再被需要了，它就有必要使所有的子作业执行失败。

```
1  package com.example.coroutinelearning
2
3  import kotlinx.coroutines.*
4  import okhttp3.Dispatcher
5  import java.lang.ArithmeticException
6  import java.lang.Exception
7  import java.lang.IllegalArgumentException
8  import java.lang.IndexOutOfBoundsException
9
10 class CoroutineLearning {
11     fun testCoroutineSupervisorScope() = runBlocking {
12         val supervisorScope = CoroutineScope(SupervisorJob())
13
14         val job1 = supervisorScope.launch {
15             delay(100)
16             println("child 1")
17             throw IllegalArgumentException()
```

```
        }

        val job2 = supervisorScope.launch {
            try {
                delay(4000)
            } catch (e: Exception) {
                e.printStackTrace()
            } finally {
                println("child 2 finished...")
            }
        }
        joinAll(job1, job2)
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineSupervisorScope()
    }
}
```

代码输出如下：

```
child 1
Exception in thread "DefaultDispatcher-worker-1"
java.lang.IllegalArgumentException
    at
com.example.coroutinelearning.CoroutineLearning$testCoroutineSuper
visorScope$1$job1$1.invokeSuspend(CoroutineLeaning.kt:17)
    at
kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(Con
tinuationImpl.kt:33)
    at
kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(Corouti
neScheduler.kt:571)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTas
k(CoroutineScheduler.kt:750)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(
CoroutineScheduler.kt:678)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(Corout
ineScheduler.kt:665)
child 2 finished...
```

可以发现，job1抛出了一个异常，但是job2并没有受到影响，依然正确的执行完了所有的任务。

如果将SupervisorJob换成普通的Job，则有：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import okhttp3.Dispatcher
import java.lang.ArithmeticException
import java.lang.Exception
import java.lang.IllegalArgumentException
import java.lang.IndexOutOfBoundsException

```

```
class CoroutineLearning {
    fun testCoroutineSupervisorScope() = runBlocking {
        val scope = CoroutineScope(Job())

        val job1 = scope.launch {
            delay(100)
            println("child 1")
            throw IllegalArgumentException()
        }

        val job2 = scope.launch {
            try {
                delay(4000)
            } catch (e: Exception) {
                println(e)
            } finally {
                println("child 2 finished...")
            }
        }
        joinAll(job1, job2)
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineSupervisorScope()
    }
}
```

```
child 1
kotlinx.coroutines.JobCancellationException: Parent job is Cancelling; job=JobImpl{Cancelling}@a8f63b0
child 2 finished...
Exception in thread "DefaultDispatcher-worker-1" java.lang.IllegalArgumentException
    at com.example.coroutinelearning.CoroutineLearning$testCoroutineSupervisorScope$1$job1$1.invokeSuspend(CoroutineLeaning.kt:17)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(CoroutineScheduler.kt:571)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTask(CoroutineScheduler.kt:750)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(CoroutineScheduler.kt:678)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(CoroutineScheduler.kt:665)
```

可以发现，在job2中捕获了JobCancellationException的异常，说明job1的异常影响了job2，导致job2也被取消了。

## supervisorScope

前面简单使用过supervisorScope来进行创建彼此独立的子协程。

在supervisorScope中，其中一个子协程因为异常取消了是不会影响其他的兄弟协程的，如：

```
package com.example.coroutinelearning

```

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking
import kotlinx.coroutines.supervisorScope

class CoroutineLearning {
    fun testCoroutineSupervisorScope() = runBlocking {
        supervisorScope {
            launch {
                delay(100)
                println("child 1")
                throw IllegalArgumentException()
            }

            try {
                delay(2000)
            } catch (e:Exception) {
                println(e)
            } finally {
                println("child 2 finished...")
            }
        }
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineSupervisorScope()
    }
}
```

输出如下：

```
child 1
Exception in thread "main" java.lang.IllegalArgumentException
    at
com.example.coroutinelearning.CoroutineLearning$testCoroutineSuper
visorScope$1$1$1.invokeSuspend(CoroutineLeaning.kt:16)
```

```
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTaskKt.resume(DispatchedTask.kt:234)
    at kotlinx.coroutines.DispatchedTaskKt.dispatch(DispatchedTask.kt:166)
    at kotlinx.coroutines.CancellableContinuationImpl.dispatchResume(CancellableContinuationImpl.kt:369)
    at kotlinx.coroutines.CancellableContinuationImpl.resumeImpl(CancellableContinuationImpl.kt:403)
    at kotlinx.coroutines.CancellableContinuationImpl.resumeImpl$default(CancellableContinuationImpl.kt:395)
    at kotlinx.coroutines.CancellableContinuationImpl.resumeUndispatched(CancellableContinuationImpl.kt:491)
    at kotlinx.coroutines.EventLoopImplBase$DelayedResumeTask.run(EventLoop.common.kt:489)
    at kotlinx.coroutines.EventLoopImplBase.processNextEvent(EventLoop.common.kt:274)
    at kotlinx.coroutines.BlockingCoroutine.joinBlocking(Builders.kt:84)
    at kotlinx.coroutines.BuildersKt__BuildersKt.runBlocking(Builders.kt:59)
    at kotlinx.coroutines.BuildersKt.runBlocking(Unknown Source)
    at kotlinx.coroutines.BuildersKt__BuildersKt.runBlocking$default(Builders.kt:38)
    at kotlinx.coroutines.BuildersKt.runBlocking$default(Unknown Source)
```

```
    at com.example.coroutinelearning.CoroutineLearning.testCoroutineSupervisorScope(CoroutineLeaning.kt:11)
    at com.example.coroutinelearning.CoroutineLeaningKt.main(CoroutineLeaning.kt:32)
    at com.example.coroutinelearning.CoroutineLeaningKt.main(CoroutineLeaning.kt)
child 2 finished...
```

但是如果是在supervisorScope这个作用域中抛出了异常，那么该作用域下的所有的子协程都会被取消。

```kotlin
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning {
    fun testCoroutineSupervisorScope() = runBlocking<Unit> {
        supervisorScope {
            val child = launch {
                try {
                    println("The child is sleeping...")
                    delay(2000)
                } catch (e: Exception) {
                    println(e)
                } finally {
                    println("The child is cancelled...")
                }
            }
            yield() // 出度CPU等资源的使用权
            println("Throwing an exception from scope")
            throw AssertionError() // 在supervisorScope中抛出一个异常
        }
    }
}

fun main() {
```

```
    CoroutineLearning().apply {
        testCoroutineSupervisorScope()
    }
}
```

输出结果如下：

```
The child is sleeping...
Throwing an exception from scope
kotlinx.coroutines.JobCancellationException: Parent job is Cancelling; job=SupervisorCoroutine{Cancelling}@17d10166
The child is cancelled...
Exception in thread "main" java.lang.AssertionError
    at com.example.coroutinelearning.CoroutineLearning$testCoroutineSupervisorScope$1$1.invokeSuspend(CoroutineLeaning.kt:21)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at kotlinx.coroutines.EventLoopImplBase.processNextEvent(EventLoop.common.kt:274)
    at kotlinx.coroutines.BlockingCoroutine.joinBlocking(Builders.kt:84)
    at kotlinx.coroutines.BuildersKt__BuildersKt.runBlocking(Builders.kt:59)
    at kotlinx.coroutines.BuildersKt.runBlocking(Unknown Source)
    at kotlinx.coroutines.BuildersKt__BuildersKt.runBlocking$default(Builders.kt:38)
    at kotlinx.coroutines.BuildersKt.runBlocking$default(Unknown Source)
    at com.example.coroutinelearning.CoroutineLearning.testCoroutineSupervisorScope(CoroutineLeaning.kt:6)
```

```
16      at
    com.example.coroutinelearning.CoroutineLeaningKt.main(CoroutineLea
    ning.kt:28)
17      at
    com.example.coroutinelearning.CoroutineLeaningKt.main(CoroutineLea
    ning.kt)
```

从输出结果来看，当supervisorScope的作用域内（不是子协程内部）抛出一个异常时，该作用域下的子协程都会被取消。

# 异常的捕获

- 使用CoroutineExceptionHandler对协程的异常进行捕获。
- 以下的条件被满足时，异常就会被捕获：
    - 时机：异常是被自动抛出异常的协程所抛出的（使用launch，而不是async时）
    - 位置：在CoroutineScope的CoroutineContext中或在一个根协程(CoroutineScope或者supervisorScope的直接子协程）中。
- 这里的根协程指的是这个协程不会任何协程的子协程，也就是这个协程的父协程不存在。根协程和协程的创建方式无关，可以通过lifecycleScope，ViewModelScope，GlobalScope等创建，只要创建的是根协程就可以通过异常捕获。

比如有以下的代码：

```
1  package com.example.coroutinelearning
2
3  import kotlinx.coroutines.*
4  import java.lang.ArithmeticException
5
6  class CoroutineLearning {
7      fun testCoroutineException() = runBlocking<Unit> {
8          val handler = CoroutineExceptionHandler {
   coroutineContext, throwable ->
9              println("Caught $throwable")
10         }
11
```

```
        val job = GlobalScope.launch(handler) {
            throw AssertionError()
        }

        val deferred = GlobalScope.async(handler) {
            throw ArithmeticException()
            "OK"
        }

        job.join()
        deferred.await()
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineException()
    }
}
```

上面的代码输出的结果如下：

```
Caught java.lang.AssertionError
Exception in thread "main" java.lang.ArithmeticException
    at com.example.coroutinelearning.CoroutineLearning$testCoroutineException$1$deferred$1.invokeSuspend(CoroutineLeaning.kt:17)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(CoroutineScheduler.kt:571)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTask(CoroutineScheduler.kt:750)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(CoroutineScheduler.kt:678)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(CoroutineScheduler.kt:665)
```

可以发现，只有job的异常被捕获了，deffered的异常没有被捕获。这是因为job采用的启动方式是launch，而且是在GlobalScope中启动的一个根协程，因此，他的异常是可以被捕获的。

如果采用的是下面的代码：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning {
    fun testCoroutineException() = runBlocking<Unit> {
        val handler = CoroutineExceptionHandler { coroutineContext, throwable ->
            println("Caught $throwable")
        }

```

```
        val scope = CoroutineScope(Job())
        val job = scope.launch(handler) {
            launch {
                throw IllegalArgumentException()
            }
        }
        job.join()
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineException()
    }
}
```

输出结果是：

```
Caught java.lang.IllegalArgumentException
```

可以发现，此时异常被handler正确的捕获到了。

如果将上面的代码改写成：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning {
    fun testCoroutineException() = runBlocking<Unit> {
        val handler = CoroutineExceptionHandler {
coroutineContext, throwable ->
            println("Caught $throwable")
        }

        val scope = CoroutineScope(Job())
        val job = scope.launch {
            launch(handler) {
                throw IllegalArgumentException()
```

```
            }
        }
        job.join()
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineException()
    }
}
```

```
Exception in thread "DefaultDispatcher-worker-2"
java.lang.IllegalArgumentException
    at
com.example.coroutinelearning.CoroutineLearning$testCoroutineExcept
ion$1$job$1$1.invokeSuspend(CoroutineLeaning.kt:14)
    at
kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(Cont
inuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(Coroutin
eScheduler.kt:571)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTask
(CoroutineScheduler.kt:750)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(C
oroutineScheduler.kt:678)
    at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(Corouti
neScheduler.kt:665)

```

此时的异常是不能被捕获的。原因在于内部的子协程抛出异常时，是直接向外部的父协程抛出异常，而此时父协程并没有对应的异常处理机制，因此会直接报错。

# Android中的全局异常处理

全局异常处理器可以获取到所有协程未处理的未捕获异常，不过它并不能对异常进行捕获，虽然不能阻止程序崩溃，全局异常处理器在程序调试和异常上报等场景中仍然有非常大的用处。

我们需要在classpath下面创建META-INF/services目录，并在其中创建一个kotlinx.coroutines.CoroutineExceptionHandler的文件，文件内容就是我们的全局异常处理器的全类名。

# 取消与异常

- 取消与异常紧密相关，协程内部使用CancellationException来进行取消，这个异常会被忽略。
- 当子协程被取消时，不会取消它的父协程。
- 如果一个协程遇到了CancellationException以外的异常，它将使用该异常取消它的父协程。当父协程的所有子协程都结束后，异常才会被父协程处理。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*

class CoroutineLearning {
    fun testCoroutineCancellationAndException() =
runBlocking<Unit> {
        val job = launch {
            val child = launch {
                try {
                    delay(Long.MAX_VALUE)
                } catch (e: Exception) {
                    println(e)
                } finally {
                    println("Child has been cancelled...")
                }
            }
            yield()
            println("Cancelling child")
            child.cancelAndJoin()
```

```
            yield()
            println("Parent is not cancelled...")
        }
        job.join()
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineCancellationAndException()
    }
}
```

输出结果是：

```
Cancelling child
kotlinx.coroutines.JobCancellationException: StandaloneCoroutine was cancelled; job=StandaloneCoroutine{Cancelling}@3581c5f3
Child has been cancelled...
Parent is not cancelled...
```

可以看出，即使是抛出了JobCancellationException这样的异常，但是这个异常是一种较为安全的异常，协程一般会对该异常进行静默处理。

查看下面的代码：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import java.lang.Exception

class CoroutineLearning {
    fun testCoroutineCancellationAndException() = runBlocking<Unit> {
        val handler = CoroutineExceptionHandler { coroutineContext, throwable ->
            println("Caught $throwable")
        }

```

```
        val job = GlobalScope.launch(handler) {
            launch {
                try {
                    delay(Long.MAX_VALUE)
                } catch (e: Exception) {
                    println(e)
                } finally {
                    withContext(NonCancellable) {
                        println("Children are cancelled, but exception is not handled until all children terminate")
                        delay(100)
                        println("The first child finished its non cancellable block")
                    }
                }
            }

            launch {
                delay(10)
                println("Second child throws an exception")
                throw ArithmeticException()
            }
        }
        job.join()
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineCancellationAndException()
    }
}
```

输出结果是：

```
Second child throws an exception
kotlinx.coroutines.JobCancellationException: Parent job is Cancelling; job=StandaloneCoroutine{Cancelling}@749ac36d
Children are cancelled, but exception is not handled until all children terminate
The first child finished its non cancellable block
Caught java.lang.ArithmeticException
```

首先是启动两个协程，在第二个协程运行很短的时间之后，抛出一个异常。此时父协程会受到异常信息，然后对其下的所有子协程进行取消操作，这样会让协程1取消，抛出了JobCancellationException异常（但是这个异常会被静默处理），接着进入协程1的finally代码块，进行输出和打印（这里采用的是不可取消的任务）。只有协程1结束之后，父协程才回去处理异常，利用handler进行输出和打印。

## 异常聚合

当协程的多个子协程因为异常而失败时，一般情况下取第一个异常进行处理。在第一个异常之后发生的所有其他异常，都将被绑定到第一个异常之上。

可以利用handler中的 `exception.suppressed` 数组去获取所有的异常。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import java.io.IOException
import java.lang.Exception

class CoroutineLearning {
    fun testCoroutineExceptionCluster() = runBlocking<Unit> {
        val handler = CoroutineExceptionHandler { coroutineContext, throwable ->
            println("Caught $throwable ${throwable.suppressed.contentToString()}")
        }

        val job = GlobalScope.launch(handler) {
            val job1 = launch {
                try {
```

```
                    delay(Long.MAX_VALUE)
                } finally {
                    throw ArithmeticException() // 2
                }
            }

            val job2 = launch {
                try {
                    delay(Long.MAX_VALUE)
                } finally {
                    throw IndexOutOfBoundsException() // 3
                }
            }

            val job3 = launch {
                delay(100)
                throw IOException() // 1
            }
        }
        job.join()
    }
}

fun main() {
    CoroutineLearning().apply {
        testCoroutineExceptionCluster()
    }
}
```

输出结果如下：

```
Caught java.io.IOException [java.lang.ArithmeticException,
java.lang.IndexOutOfBoundsException]
```

可以发现，首先是job3在运行过程中抛出了异常，这样会导致其他的协程job1和job2得到取消，然后在各自的finally代码块中抛出各自的异常。很显然，后两个异常因为job3的异常而产生，因此后面两个异常会附加在job3的异常后面，通过`exception.suppressed`获取异常数组信息。

# Flow

## Flow的基本概念

很多时候，我们想要协程可以返回多个值，比如下载文件的时候，需要经常去根据文件的下载状态去更新界面上的进度。很明显，下载文件可以放在一个协程中处理，但是协程如何更加方便的去返回下载的进度呢？Flow提供了一种可以参考的方法。

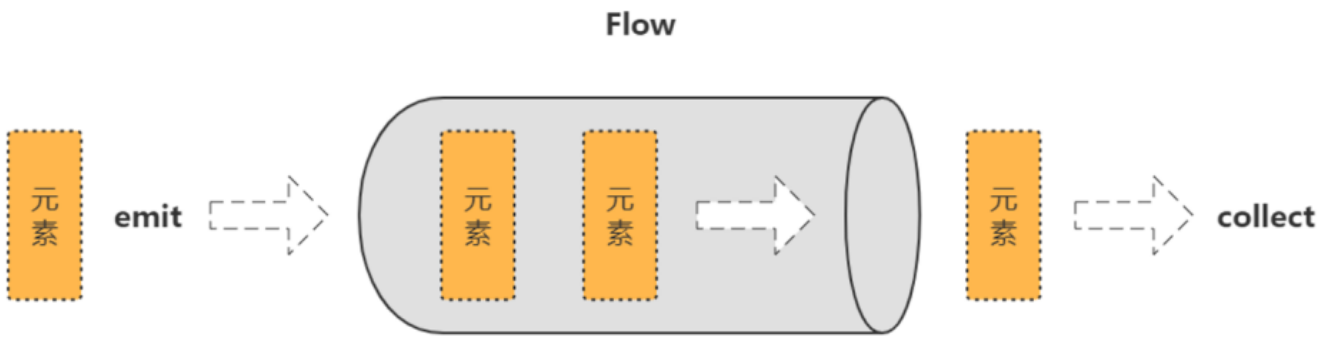
挂起函数可以异步的返回单个值， 但是该如何异步返回多个计算好的值呢?

异步返回多个值的方案可以有以下几个

- 集合
- 序列
- 挂起函数
- Flow

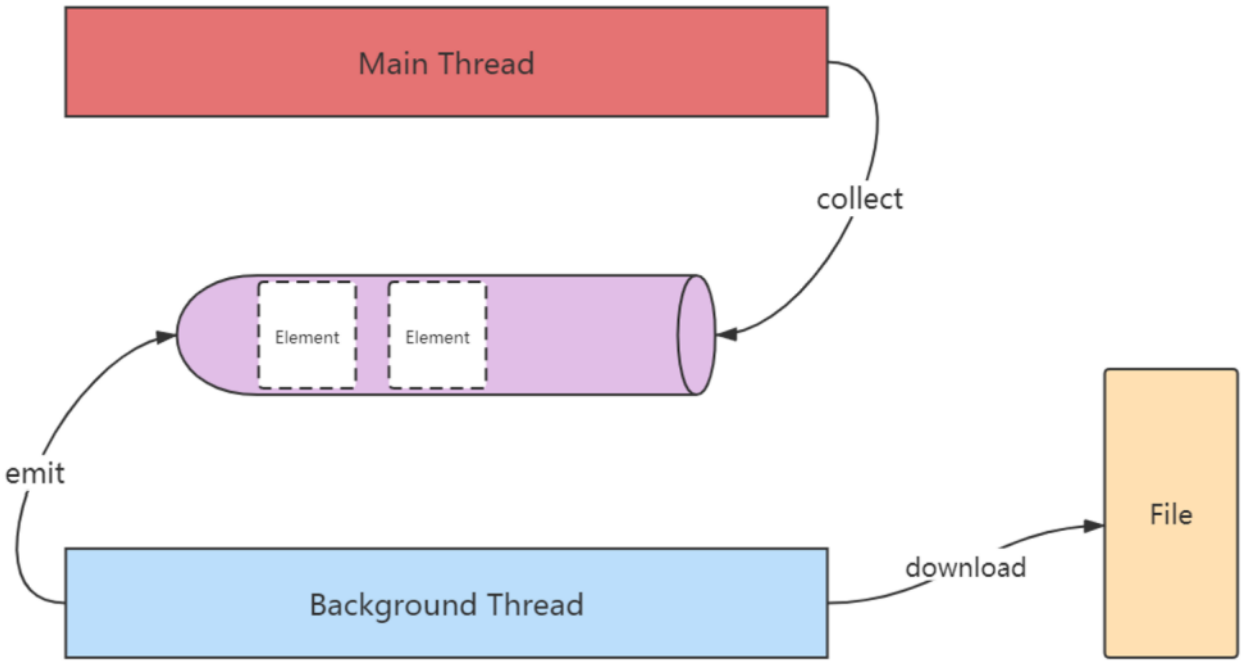
### Flow和其他方式的区别

- 名为flow的Flow类型构建器函数。
- flow{...} 构建块中的代码可以挂起。
- 函数simpleFlow不再标有suspend修饰符。
- Flow使用emit函数发射值。
- Flow使用collect函数收集值。



## Flow的应用

在Android当中， 文件下载是Flow的一个非常典型的应用。




## Flow的简单使用

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.flow

class FlowLearning {
    /**
     * 返回一个简单的包含Int数据的flow
     * 在这个Flow中，每隔1000毫秒发送一个Int数据。（向Flow中发送数据采用emit方法）
     */
    fun simpleFlow() = flow<Int>  {
        for (i in 1..3) {
            delay(1000)
            emit(i)
        }
    }
```

```
18
19      /**
20       * 使用Flow。
21       * 获取flow中的数据，需要使用Flow中提供的collect方法。
22       */
23      fun testSimpleFlow() = runBlocking {
24          simpleFlow().collect {
25              println(it)
26          }
27      }
28  }
29
30  fun main() {
31      FlowLearning().testSimpleFlow()
32  }
```

运行上面的代码，每隔一秒钟会输出一个数字，会依次输出1，2，3。

由于上面的代码都是在协程中运行的，因此即使协程中会耗费很多时间，对于主线程也是几乎没有影响的。且上面的代码每一个都是异步地返回一个数字，正好符合下载文件更新进度的需求。

观察`flow`函数：

```
1   /**
2    * Creates flow from the given suspendable [block].
3    *
4    * Example of usage:
5    * ```
6    * fun fibonacci(): Flow<Long> = flow {
7    *     emit(1L)
8    *     var f1 = 1L
9    *     var f2 = 1L
10   *     repeat(100) {
11   *         var tmp = f1
12   *         f1 = f2
13   *         f2 += tmp
14   *         emit(f1)
```

```
 *     }
 * }
 * ```
 *
 * `emit` should happen strictly in the dispatchers of the [block] in order to preserve flow context.
 * For example, the following code will produce [IllegalStateException]:
 * ```
 * flow {
 *     emit(1) // Ok
 *     withContext(Dispatcher.IO) {
 *         emit(2) // Will fail with ISE
 *     }
 * }
 * ```
 * If you want to switch the context where this flow is executed use [flowOn] operator.
 */
@FlowPreview
public fun <T> flow(@BuilderInference block: suspend FlowCollector<T>.() -> Unit): Flow<T> {
    return object : Flow<T> {
        override suspend fun collect(collector: FlowCollector<T>) {
            SafeCollector(collector, coroutineContext).block()
        }
    }
}
```

可以看出，该函数返回的是一个Flow对象。传递给该函数的是一个可挂起的对于`FlowCollector`的一个扩展函数。

观察`emit`方法：

```
/**
 * [FlowCollector] is used as an intermediate or a terminal collector of the flow and represents
 * an entity that accepts values emitted by the [Flow].
```

```
 *
 * This interface usually should not be implemented directly, but rather used as a receiver in [flow] builder when implementing a custom operator.
 * Implementations of this interface are not thread-safe.
 */
@FlowPreview
public interface FlowCollector<in T> {

    /**
     * Collects the value emitted by the upstream.
     */
    public suspend fun emit(value: T)
}
```

可以看出，该方法是`FlowCollector`这个接口中唯一的方法，注释说明该函数的主要作用就是收集上游数据流中发送的数据。

观察`collect`方法：

```
/**
 * Terminal flow operator that collects the given flow with a provided [action].
 * If any exception occurs during collect or in the provided flow, this exception is rethrown from this method.
 *
 * Example of use:
 * ```
 * val flow = getMyEvents()
 * try {
 *     flow.collect { value ->
 *         println("Received $value")
 *     }
 *     println("My events are consumed successfully")
 * } catch (e: Throwable) {
 *     println("Exception from the flow: $e")
 * }
 * ```
 */
```

```
@FlowPreview
public suspend fun <T> Flow<T>.collect(action: suspend (value: T)
-> Unit): Unit =
    collect(object : FlowCollector<T> {
        override suspend fun emit(value: T) = action(value)
    })
```

该方法是一个Flow的扩展方法，是一个Flow中的最终的一个方法调用。该方法就是从Flow中获取数据。因此collect方法是一个Flow的末端操作符。

# Flow特性

## Flow只一种冷流

Flow是一种类似千序列的冷流， flow构建器中的代码直到流被收集的时候才运行。

下面的代码可以提供一个较好的解读：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.flow

class FlowLearning {
    fun simpleFlow() = flow<Int> {
        println("Flow started")
        for (i in 1..3) {
            delay(1000)
            emit(i)
        }
    }

    fun testColdFlow() = runBlocking {
        val flow = simpleFlow()
        println("Calling collect...")
        flow.collect { value -> println(value) }
        println("Calling collect again...")
        flow.collect { value -> println(value) }
```

```
    }
}

fun main() {
    FlowLearning().testColdFlow()
}
```

代码输出如下：

```
Calling collect...
Flow started
1
2
3
Calling collect again...
Flow started
1
2
3
```

可以看出，我们首先通过`simpleFlow`函数构建一个Flow对象，但是这个Flow对象中的代码并没有立刻被执行，直到我们对这个Flow对象调用collect方法进行数据收集之后，才会运行Flow对象中的代码。然后再次对该Flow对象调用collect方法，Flow对象中的代码又会得到运行。

这个就是Flow是一个冷流的例子。

## Flow的连续性

- 流的每次单独收集都是按顺序执行的， **除非使用特殊操作符**。
- 从上游到下游每个过渡操作符都会处理每个发射出的值， 然后再交给末端操作符。

在这一点上，Flow是一个Queue的实现，都是先进先出FIFO的数据结构。

```
package com.example.coroutinelearning

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

```
5
6  class FlowLearning {
7      fun testFlowContinuation() = runBlocking {
8          (1..5).asFlow().filter {
9              it % 2 == 0
10         }.map {
11             "string $it"
12         }.collect {
13             println("Collect: $it")
14         }
15     }
16 }
17
18 fun main() {
19     FlowLearning().testFlowContinuation()
20 }
```

输出如下：

```
1  Collect: string 2
2  Collect: string 4
```

在上面的代码中，首先利用`(1..5).asFlow()`的方式构建了一个简单的数据流，这个数据流中，1到5按照顺序进行发送（emit）。

接着调用`filter`操作符筛选出符合要求的数据，这里只需要使用所有的偶数的数据。

最后调用Flow的末端操作符`collect`对数据进行收集和输出。

从最后的输出可以看出，所有的结果都是按照发送的顺序进行排列的，这也说明Flow是一个体现了FIFO思想的数据模型。

# Flow的构建

除了上面的使用显示的`flow`函数构建一个Flow对象之外，还有一些很方便的方法来构建Flow对象。

- `flowOf`：该构建器定义了一个发射固定数据集合的Flow对象。

- `.asFlow()`：`.asFlow()`是一个扩展函数，可以将各种集合和序列转化成Flow对象。

```
package com.example.coroutinelearning

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

class FlowLearning {
    fun testFlowConstruction1() = runBlocking {
        (1..5).asFlow().collect {
            println("Collect: $it")
        }
    }

    fun testFlowConstruction2() = runBlocking {
        flowOf("one", "two", "three")
                .onEach { delay(1000) }
                .collect{
                    println("Collect: $it")
                }

    }
}

fun main() {
    val a = FlowLearning()

    a.testFlowConstruction1()

    a.testFlowConstruction2()
}
```

```
Collect: 1
Collect: 2
Collect: 3
Collect: 4
Collect: 5
Collect: one
Collect: two
Collect: three
```

从上面的代码可以看出，使用上面的两个构建器以及`flow`方法可以很方便地构建出各种数据流。

## Flow的上下文

- 流的收集总是在调用协程的上下文中发生， 流的该属性称为**上下文保存**。
- flow{...} 构建器中的代码必须遵循上下文保存属性， 并且不允许从其他上下文中发射（emit）。
- 使用`flowOn`**操作符**可以用于更改流发射的上下文。

回到最开始的我们创建一个Flow的代码：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.flow

class FlowLearning {
    /**
     * 返回一个简单的包含Int数据的flow
     * 在这个Flow中，每隔1000毫秒发送一个Int数据。（向Flow中发送数据采用emit方法）
     */
    fun simpleFlow() = flow<Int>  {
        for (i in 1..3) {
            delay(1000)
            emit(i)
        }
    }
```

```
18
19      /**
20       * 使用Flow。
21       * 获取flow中的数据，需要使用Flow中提供的collect方法。
22       */
23      fun testSimpleFlow() = runBlocking {
24          simpleFlow().collect {
25              println(it)
26          }
27      }
28  }
29
30  fun main() {
31      FlowLearning().testSimpleFlow()
32  }
```

观察上面的`simpleFlow`函数，可以发现并没有使用`suspend`关键字，但是创建流的过程中需要使用一个协程的上下文，因此也是可以在创建流的过程中使用任意的挂起函数的。但是上面的代码中并没有对这个上下文环境进行指定，那么这个协程上下文是在什么时候设置的呢？其实在`collect`收集数据的过程中，会将协程的上下文环境传递给对应的流的构建过程，也就是说在一般情况下（即不对协程的上下文进行特别处理时），流的创建过程和流的收集过程会在同一个协程上下文的环境中。

```
1   package com.example.coroutinelearning
2
3   import kotlinx.coroutines.*
4   import kotlinx.coroutines.flow.*
5
6   class FlowLearning {
7       fun simpleFlow() = flow<Int> {
8           println("Flow started: " + Thread.currentThread().name)
9           for (i in 1..3) {
10              delay(100)
11              emit(i)
12          }
13      }
14
15      fun testFlowContext1() = runBlocking {
```

```
        println("Flow in IO Thread...")
        withContext(Dispatchers.IO) {
            simpleFlow().collect {
                println("Collected $it ${Thread.currentThread().name}")
            }
        }
    }

    fun testFlowContext2() = runBlocking {
        println("Flow in Main Thread...")
        simpleFlow().collect {
            println("Collected $it ${Thread.currentThread().name}")
        }
    }
}

fun main() {
    val a = FlowLearning()

    a.testFlowContext1()

    a.testFlowContext2()
}
```

```
Flow in IO Thread...
Flow started: DefaultDispatcher-worker-2
Collected 1 DefaultDispatcher-worker-2
Collected 2 DefaultDispatcher-worker-2
Collected 3 DefaultDispatcher-worker-1
Flow in Main Thread...
Flow started: main
Collected 1 main
Collected 2 main
Collected 3 main
```

可以发现，在上述的代码中，Flow的收集是在主线程中，Flow构建的过程也是在主线程中，Flow的收集是在IO线程中，Flow构建的过程也是在IO线程中。因此，在不对Flow的协程上下文进行特殊设置的情况下，流的创建过程和流的收集过程会在同一个协程上下文的环境中。

但是往往很多时候我们需要Flow的构建和收集在不同的线程中，比如下载文件的时候，下载过程需要在IO线程中，然后发送进度数据，更新UI上的进度信息需要在主线程中，因此需要对Flow的发送和收集过程进行线程切换。

## `flowOn`操作符

很自然的，可以根据之前的协程中线程切换的方法尝试在Flow的创建过程中进行线程切换。

```
package com.example.coroutinelearning

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

class FlowLearning {
    fun simpleFlow() = flow<Int> {
        withContext(Dispatchers.IO) {
            println("Flow started: " +
Thread.currentThread().name)
            for (i in 1..3) {
                delay(100)
                emit(i)
            }
        }
    }

    fun testFlowContext() = runBlocking {
        println("Flow in Main Thread...")
        simpleFlow().collect {
            println("Collected $it
${Thread.currentThread().name}")
        }
    }
}
```

```
24
25 fun main() {
26     val a = FlowLearning()
27     a.testFlowContext()
28 }
```

在Android Studio中，会直接在`withContext`处报错，如下：

```
1 Using 'withContext(CoroutineContext, suspend () -> R): Unit' is an
  error. withContext in flow body is deprecated, use flowOn instead
```

在Flow中，不能使用`withContext`进行线程切换，Flow构建过程中的这个方法已经被废弃了。还提示使用`flowOn`操作符。

使用`flowOn`操作符进行线程环境切换也十分简单。如下：

```
1  package com.example.coroutinelearning
2
3  import kotlinx.coroutines.*
4  import kotlinx.coroutines.flow.*
5
6  class FlowLearning {
7      fun simpleFlow() = flow<Int> {
8          println("Flow started: " + Thread.currentThread().name)
9          for (i in 1..3) {
10             delay(100)
11             emit(i)
12         }
13     }.flowOn(Dispatchers.IO)
14
15     fun testFlowContext() = runBlocking {
16         println("Flow in Main Thread...")
17         simpleFlow().collect {
18             println("Collected $it
   ${Thread.currentThread().name}")
19         }
20     }
21 }
22
```

```
fun main() {
    val a = FlowLearning()
    a.testFlowContext()
}
```

```
Flow in Main Thread...
Flow started: DefaultDispatcher-worker-1
Collected 1 main
Collected 2 main
Collected 3 main
```

可以看到，使用`flowOn`操作符之后，构建流的过程处在IO线程中，而流的收集仍然是处于主线程的环境中。

`flowOn`操作符的函数签名如下：

```
/**
 * The operator that changes the context where this flow is executed to the given [flowContext].
 * This operator is composable and affects only preceding operators that do not have its own context.
 * This operator is context preserving: [flowContext] **does not** leak into the downstream flow.
 *
 * For example:
 * ```
 * withContext(Dispatchers.Main) {
 *     val singleValue = intFlow // will be executed on IO if context wasn't specified before
 *         .map { ... } // Will be executed in IO
 *         .flowOn(Dispatchers.IO)
 *         .filter { ... } // Will be executed in Default
 *         .flowOn(Dispatchers.Default)
 *         .single() // Will be executed in the Main
 * }
 * ```
 * For more explanation of context preservation please refer to [Flow] documentation.
 *
```

```
 * This operator uses a channel of the specific [bufferSize] in order to switch between contexts,
 * but it is not guaranteed that the channel will be created, implementation is free to optimize it away in case of fusing.
 *
 * @throws [IllegalArgumentException] if provided context contains [Job] instance.
 */
@FlowPreview
public fun <T> Flow<T>.flowOn(flowContext: CoroutineContext, bufferSize: Int = 16): Flow<T>
```

## `launchIn`操作符

有时候也可以在指定的协程中收集流。这个时候我们就可以使用`launchIn`操作符进行线程切换。使用launch In替换collect我们可以在单独的协程中启动流的收集。

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

class FlowLearning {
    private fun simpleFlow() = flow<Int> {
        println("Flow started: " + Thread.currentThread().name)
        for (i in 1..3) {
            delay(100)
            emit(i)
        }
    }.flowOn(Dispatchers.IO)

    fun testFlowContext() = runBlocking {
        simpleFlow().onEach {
            println("LaunchIn: $it " + Thread.currentThread().name)
        }.launchIn(this).join()
    }
}

fun main() {
    val a = FlowLearning()
```

```
    a.testFlowContext()
}
```

```
Flow started: DefaultDispatcher-worker-1
LaunchIn: 1 main
LaunchIn: 2 main
LaunchIn: 3 main
```

可以看到，在`testFlowContext`方法中，我们调用`onEach`操作符对每个数据进行打印并输出对应的线程，接着调用`launchIn`操作符进行线程切换。

**需要注意的是，对于Flow，影响都是从下游开始，逐级影响上游的操作。**

所以，对于上面的代码，下游的`launchIn`会将线程的上下文切换成主线程，这个上下文的变化会影响他上游的操作符，即`onEach`，因此输出会是在主线程中。

在`onEach`的上游，调用了`flowOn`操作符，会将线程环境切换成IO线程，因此，这个操作会影响这个操作符上游的操作符的线程环境。所以最上游的`flow`会在IO线程中进行发送数据。

这个特点正好说明**Flow是一个冷流**。下面的代码是一个非常直观的说明Flow的上下文是从下流影响上流的说明

```
package com.example.coroutinelearning

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

class FlowLearning {

    fun testFlowContext() = runBlocking {
        flow<Int> {
            println("flow generated in :
${Thread.currentThread().name}")
            for (i in 0..3) {
                emit(i)
            }
        }.flowOn(Dispatchers.IO)
                .map {
```

```
                println("square in : ${Thread.currentThread().name}")
                it * it
            }.flowOn(Dispatchers.Default).filter {
                println("filter in : ${Thread.currentThread().name}")
                it > 3
            }.flowOn(Dispatchers.IO).map {
                "$it"
            }.onEach { println("onEach in : ${Thread.currentThread().name}") }.launchIn(this).join()
    }
}

fun main() {
    val a = FlowLearning()
    a.testFlowContext()
}
```

```
flow generated in : DefaultDispatcher-worker-3
square in : DefaultDispatcher-worker-5
square in : DefaultDispatcher-worker-5
square in : DefaultDispatcher-worker-5
square in : DefaultDispatcher-worker-5
filter in : DefaultDispatcher-worker-3
filter in : DefaultDispatcher-worker-3
filter in : DefaultDispatcher-worker-3
filter in : DefaultDispatcher-worker-3
onEach in : main
onEach in : main
```

可以看出线程环境的切换都是下游影响上游。因此，如果没有`collect`操作进行收集，是不会触发上游的代码运行的。

关于`launchIn`操作符，官方的函数签名如下：

```
/**
 * Terminal flow operator that [launches][launch] the [collection][collect] of the given flow in the [scope].
```

```
 * It is a shorthand for `scope.launch { flow.collect() }`.
 *
 * This operator is usually used with [onEach], [onCompletion] and [catch] operators to process all emitted values
 * handle an exception that might occur in the upstream flow or during processing, for example:
 *
 * ```
 * flow
 *     .onEach { value -> updateUi(value) }
 *     .onCompletion { cause -> updateUi(if (cause == null) "Done" else "Failed") }
 *     .catch { cause -> LOG.error("Exception: $cause") }
 *     .launchIn(uiScope)
 * ```
 *
 * Note that resulting value of [launchIn] is not used the provided scope takes care of cancellation.
 */
public fun <T> Flow<T>.launchIn(scope: CoroutineScope): Job = scope.launch {
    collect() // tail-call
}
```

可以看出，该操作符需要一个协程的上下文作为参数，返回的是一个Job对象。既然返回的是一个Job对象，那么对于Job的一切操作都可以进行，包括`join`，`cancel`等。所以上面的代码中我们调用了`join`操作符，让主线程等待Flow的结束。

## 流的取消

流采用与协程同样的协作取消。 像往常一样，流的收集可以是当流在一个可取消的挂起函数（如delay）中挂起的时候取消。

```
package com.example.coroutinelearning

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

```

```
class FlowLearning {

    fun simpleFlow() = flow<Int> {
        for (i in 1..3) {
            delay(1000)
            println("Emitting $i")
            emit(i)
        }
    }

    fun testFlowCancellation() = runBlocking {
        withTimeoutOrNull(2500) {
            simpleFlow().collect {
                println("Collecting $it")
            }
        }
    }
}

fun main() {
    FlowLearning().testFlowCancellation()
}
```

```
Emitting 1
Collecting 1
Emitting 2
Collecting 2
```

例如上面的代码，设置了每隔一秒发送一个数据的Flow，同时设置了一个超时的协程任务，2500毫秒之后取消协程，那么对于处在写成内部的Flow的收集也会被取消。所以可以看出只发射和收集到了前面两个元素，第三个元素并没有被收集到。

## Flow的取消检测

为方便起见， 流构建器对每个发射值执行附加的 ensureActive 检测以进行取消，这意味着从flow{...} 发出的繁忙循环是可以取消的。

出于性能原因， 大多数其他流操作不会自行执行其他取消检测， 在协程处于繁忙循环的情况下， 必须明确检测是否取消。

通过cancellable操作符来执行此操作。

```
package com.example.coroutinelearning

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

class FlowLearning {

    fun simpleFlow() = flow<Int> {
        for (i in 1..5) {
            println("Emitting $i")
            emit(i)
        }
    }

    fun testFlowCancellation() = runBlocking {
        simpleFlow().collect{
            println("Collecting $it")
            if (it == 3) {
                cancel()
            }
        }
    }
}

fun main() {
    FlowLearning().testFlowCancellation()
}
```

```
Emitting 1
Collecting 1
Emitting 2
Collecting 2
Emitting 3
Collecting 3
Emitting 4
Exception in thread "main"
kotlinx.coroutines.JobCancellationException: BlockingCoroutine was
cancelled; job=BlockingCoroutine{Cancelled}@4e515669
```

上面的代码中，当收集到3这个元素的时候取消当前的Flow。因此可以看到对应的输出只是发送出了4这个元素，但是并没有被收集，因为此时Flow已经被取消了。

但是上面的代码使用了`flow{ ... }`构建器来构建一个Flow，因此这个Flow是可以被取消的。

如果是下面的繁忙任务Flow，就不能够被取消。

```
package com.example.coroutinelearning

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

class FlowLearning {

    fun simpleFlow() = (1 .. 5).asFlow()

    fun testFlowCancellation() = runBlocking {
        simpleFlow().collect{
            println("Collecting $it")
            if (it == 3) {
                cancel()
            }
        }
    }
}

fun main() {
```

```
    FlowLearning().testFlowCancellation()
}
```

```
Collecting 1
Collecting 2
Collecting 3
Collecting 4
Collecting 5
Exception in thread "main"
kotlinx.coroutines.JobCancellationException: BlockingCoroutine was
cancelled; job=BlockingCoroutine{Cancelled}@22927a81
```

这个例子很好的说明了上面的第二点，即大多数其他流操作不会自行执行其他取消检测， 在协程处于繁忙循环的情况下， 必须明确检测是否取消。如果想要当前的繁忙流可以被取消，那么可以使用`cancallable`操作符来明确说明当前的流是可以被取消的。

```
package com.example.coroutinelearning

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

class FlowLearning {

    fun simpleFlow() = (1 .. 5).asFlow()

    fun testFlowCancellation() = runBlocking {
        simpleFlow().cancellable().collect{
            println("Collecting $it")
            if (it == 3) {
                cancel()
            }
        }
    }
}

fun main() {
    FlowLearning().testFlowCancellation()
}
```

```
Collecting 1
Collecting 2
Collecting 3
Exception in thread "main"
kotlinx.coroutines.JobCancellationException: BlockingCoroutine was
cancelled; job=BlockingCoroutine{Cancelled}@f2a0b8e
```

可以看到，上面的流被正确的取消了。

# 背压

有时候会出现这种情况，当生产者生成的数据太快，以至于消费者完全来不及消费掉生产出的数据，这种情况下就会产生背压。

解决背压往往有两种方式，一个是让生产者生产数据慢一点，或者让消费者消费数据的效率更高一点。总的方式就是让两者的速度尽可能保持在一个平衡的方式。

下面的代码显示的是通常情况下发送数据和处理数据的情况

```kotlin
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.measureTimeMillis

class FlowLearning {

    fun simpleFlow() = flow<Int> {
        for (i in 1..3) {
            delay(100) // 模拟每隔100毫秒发送一次数据
            println("Emitting $i")
            emit(i)
        }
    }

    fun testFlowBackPressure() = runBlocking {
        val time = measureTimeMillis {
            simpleFlow().collect {
                delay(500) // 模拟消费数据需要花费500毫秒的时间
```

```
                println("Collecting $it
${Thread.currentThread().name}")
            }
        }

        println("Collecting data cost time : $time")
    }
}

fun main() {
    FlowLearning().testFlowBackPressure()
}
```

```
Emitting 1
Collecting 1 main
Emitting 2
Collecting 2 main
Emitting 3
Collecting 3 main
Collecting data cost time : 1932
```

可以发现，由于Flow的冷流特性，只有下游有动作，上游才会有相应的动作，因此，每次需要数据的时候，都需要等待上游生成数据，因此一来一回，时间就会逐渐累加起来。上面的例子中，消费需要花费500毫秒，生成需要花费100毫秒，一个需要3个数据，因此一共大约需要$(100+500)*3=1800$毫秒的时间。

背压的存在往往会降低运行的效率。因此可以通过一些手段提高代码的效率。

## `flowOn`

在上面的例子中，Flow都是在主线程中运行，因此可以将数据发送的代码块放到其他线程中运行，比如Default调度器中。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.measureTimeMillis

```

```
class FlowLearning {

    fun simpleFlow() = flow<Int> {
        for (i in 1..3) {
            delay(100) // 模拟每隔100毫秒发送一次数据
            println("Emitting $i")
            emit(i)
        }
    }.flowOn(Dispatchers.Default)

    fun testFlowBackPressure() = runBlocking {
        val time = measureTimeMillis {
            simpleFlow().collect {
                delay(500) // 模拟消费数据需要花费500毫秒的时间
                println("Collecting $it
${Thread.currentThread().name}")
            }
        }

        println("Collecting data cost time : $time")
    }
}

fun main() {
    FlowLearning().testFlowBackPressure()
}
```

```
Emitting 1
Emitting 2
Emitting 3
Collecting 1 main
Collecting 2 main
Collecting 3 main
Collecting data cost time : 1864
```

可以发现，切换线程之后，数据会优先被发送出来，然后在被接受，由于数据发送和接受是两个独立的线程，因此可以在一定程度上减少代码的执行时间。

但是这并不是处理背压最好的方法。

下面的几个方法能够从机制上减少背压出现时的影响，提高代码的执行效率。

## `buffer`方法

`buffer`方法可以并发运行流中发射元素的代码。

该方法使用起来很十分简单：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.measureTimeMillis

class FlowLearning {

    fun simpleFlow() = flow<Int> {
        for (i in 1..3) {
            delay(100) // 模拟每隔100毫秒发送一次数据
            println("Emitting $i")
            emit(i)
        }
    }

    fun testFlowBackPressure() = runBlocking {
        val time = measureTimeMillis {
            simpleFlow().buffer(50).collect {
                delay(500) // 模拟消费数据需要花费500毫秒的时间
                println("Collecting $it
${Thread.currentThread().name}")
            }
        }

        println("Collecting data cost time : $time")
    }
}

fun main() {
    FlowLearning().testFlowBackPressure()
```

```
}
```

```
Emitting 1
Emitting 2
Emitting 3
Collecting 1 main
Collecting 2 main
Collecting 3 main
Collecting data cost time : 1846
```

可以看出，`buffer`方法可以并发的执行数据的发射方法，即尽可能先将数据发送出来并缓存，然后再依次进行收集，这样可以减少发射数据所消耗的时间。

从最后的结果也可以看出来时间会比不使用`buffer`方法的情况小，这是因为发送数据是并发的，相当于只消耗了约100毫秒。

`buffer`可以理解为将整个Flow的管道进行延长，提前将数据放入管道。

## `conflate`方法

`coinflate`方法可以合并发射项，不对每个值进行处理。换句话说，当生产方发射数据的速度大于消费数据的速度的时候，接收端永远只能拿到生产方最新发射的数据。

该方法使用起来也很方便，如下：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.measureTimeMillis

class FlowLearning {

    fun simpleFlow() = flow<Int> {
        for (i in 1..3) {
            delay(100) // 模拟每隔100毫秒发送一次数据
            println("Emitting $i")
            emit(i)
        }
```

```
    }

    fun testFlowBackPressure() = runBlocking {
        val time = measureTimeMillis {
            simpleFlow().conflate().collect {
                delay(500) // 模拟消费数据需要花费500毫秒的时间
                println("Collecting $it ${Thread.currentThread().name}")
            }
        }

        println("Collecting data cost time : $time")
    }
}

fun main() {
    FlowLearning().testFlowBackPressure()
}
```

```
Emitting 1
Emitting 2
Emitting 3
Collecting 1 main
Collecting 3 main
Collecting data cost time : 1309
```

从最后的结果可以看出，最后的结果并没有收集到全部的数据，这是因为采用了`conflate`方法之后，总是获取最新的值。

首先，第100毫秒时，发送方会发射1，接收到接收到了1，然后花了500毫秒的时间去处理。但是在第200毫秒的时候，发送方发射了数据2，此时接收端还没有把上一条数据处理完，只能先暂时忽略掉数据2。然后第300毫秒的时候，发送方发射出了数据3，但是接收端还是没有处理完数据1，因此数据3也只能被暂时忽略。

时间来到第600毫秒的时候，此时接收端处理完了数据1，然后发现发送方最新的数据是数据3，然后直接拿数据3进行消费并打印。因此虽然数据1，2，3都被发射了出来，但是只有数据1和数据3得到了处理。

两者对比，明显能发现使用conflate的例子替我们忽略了很多无法即时处理的数据。通过这种方法，在一定程度上可以提高执行的代码效率。

## `conflateLast`方法

该方法的含义是，如果当前有数据已经发送到了接收端（消费者）手上，但是上一条数据还没有被接收端（消费者）处理完，那么接收端直接停止上一条数据的处理工作，而直接开始进行新数据的处理。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.measureTimeMillis

class FlowLearning {

    fun simpleFlow() = flow<Int> {
        for (i in 1..3) {
            delay(100) // 模拟每隔100毫秒发送一次数据
            println("Emitting $i")
            emit(i)
        }
    }

    fun testFlowBackPressure() = runBlocking {
        val time = measureTimeMillis {
            simpleFlow().collectLatest{
                delay(500) // 模拟消费数据需要花费500毫秒的时间
                println("Collecting $it
${Thread.currentThread().name}")
            }
        }

        println("Collecting data cost time : $time")
    }
}

fun main() {
```

```
    FlowLearning().testFlowBackPressure()
}
```

```
Emitting 1
Emitting 2
Emitting 3
Collecting 3 main
Collecting data cost time : 1361
```

可以看出，和`conflate`方法不同，`collectLast`方法永远优先处理最新的数据，哪怕手上的数据只处理了一半，也会立刻停下手上的事情，去处理最新的数据。这是因此`collectLast`方法会默认为最新的数据优先级最高。

同样，和`conflate`方法类似，由于忽略了一些数据，所以时间上会有所减少。

# Flow操作符

## `map`操作符

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.measureTimeMillis

class FlowLearning {

    fun simpleFlow() = flow<Int> {
        for (i in 1..3) {
            delay(100) // 模拟每隔100毫秒发送一次数据
            emit(i)
        }
    }

    fun testFlowBackPressure() = runBlocking {
        simpleFlow().map {
            it * it
        }.map {
```

```
            "data is $it"
        }.collect {
            println(it)
        }
    }
}

fun main() {
    FlowLearning().testFlowBackPressure()
}
```

```
data is 1
data is 4
data is 9
```

map操作符是最常用的操作符之一，它会对每一个接收到的数据进行变换，然后继续把数据传递给下游。比如上面的代码经过了两个map操作符，第一个会将所有的数据进行平方操作，第二个会将所有的数据转换成特定格式的字符串。最后经过collect的末端操作符打印输出。

## `transform`操作符

当然很多时候对一个数据不会只进行依次单一的变换，这个时候就会用到`transform`操作符了。它可以向下游发送任意数量的数据。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.measureTimeMillis

class FlowLearning {

    fun simpleFlow() = flow<Int> {
        for (i in 1..3) {
            delay(100) // 模拟每隔100毫秒发送一次数据
            emit(i)
        }
```

```
    }

    fun testFlowBackPressure() = runBlocking {
        simpleFlow().transform {
            if (it == 3) {
                emit(it)
            } else {
                emit(it * it)
                emit(it * it * it)
            }
        }.collect {
            println(it)
        }
    }
}

fun main() {
    FlowLearning().testFlowBackPressure()
}
```

```
1
1
4
8
3
```

从上面的代码可以看出，`transform`可以向下游发送任意多个数据，比如当遇到3的时候就直接向下游发送3，反之则发送数据的平方和立方。在`transform`方法中，向下游发送数据依旧采用的是`emit`方法。

## `take`限长操作符

有时候，发送端会发送出更多的数据，但是下游的处理不需要那么多，可能只需要前面的若干个数据就够了，这个时候就可以使用`take`限长操作符。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
```

```
import kotlinx.coroutines.flow.*
import java.lang.Exception
import kotlin.system.measureTimeMillis

class FlowLearning {

    fun simpleFlow() = flow<Int> {
        try {
            emit(1)
            emit(2)
            println("This line will not be printed...")
            emit(3)
        } catch (e: Exception) {
            println(e)
        } finally {
            println("Finally executed...")
        }
    }

    fun testFlowBackPressure() = runBlocking {
        simpleFlow().take(2).collect {
            println(it)
        }
    }
}

fun main() {
    FlowLearning().testFlowBackPressure()
}
```

```
1
2
kotlinx.coroutines.flow.internal.AbortFlowException: Flow was aborted, no more elements needed
Finally executed...
```

从最后的结果可以看出来，经过限长操作符`take`之后，下游的流只会接收到固定长度的数据，其他的数据都会被忽略掉，因为这个时候流的上游操作已经被终止了，抛出了`AbortFlowException`的异常。这个异常也会被Flow内部静默处理掉，因此是安全的，可以忽略掉。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import java.lang.Exception
import kotlin.system.measureTimeMillis

class FlowLearning {

    fun simpleFlow() = flow<Int> {
            emit(1)
            emit(2)
            println("This line will not be printed...")
            emit(3)
    }

    fun testFlowBackPressure() = runBlocking {
        simpleFlow().take(2).collect {
            println(it)
        }
    }
}

fun main() {
    FlowLearning().testFlowBackPressure()
}
```

```
1
2
```

由于`take`操作符的存在，整个上游都已经被终止了，那么自然也就不会有对应的输出了。

## 末端操作符

末端操作符是在留上用于启动收集的挂起函数。`collect`是最基础的末端操作符，但是还有另外一些更方便使用的末端操作符

- 转换成各种集合的操作符，如`toList`和`toSet`等。
- 获取第一个（`first`）值与确保流发射单个（`single`）值得操作符
- 使用`reduce`和`flod`操作符将流规约到单个值。

下面是几个末端操作符的使用示例：

```
package com.example.coroutinelearning

import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking

class FlowLearning {

    fun testToListOperator() = runBlocking {
        val list = (1..5).asFlow().onEach { delay(100) }.map { it * it }.toList()
        println("Collecting list: $list")
    }

    fun testToSetOperator() = runBlocking {
        val set = (1..5).asFlow().onEach { delay(100) }.map { it * it }.toSet()
        println("Collecting set: $set")
    }

    fun testFirstOperator1() = runBlocking {
        val firstValue = (1..5).asFlow().onEach { delay(100) }.first()
        println("Collecting first value: $firstValue")
    }

```

```
    fun testFirstOperator2() = runBlocking {
        val firstValue = (1..5).asFlow().map { it * it }.first {
            it > 10
        }
        println("Collecting first value with predicate:
$firstValue")
    }

    fun testReduceOperator() = runBlocking {
        val value = (1..5).asFlow().map { it * it }.reduce {
accumulator, value ->
            accumulator + value
        }
        println("Collecting reduced value: $value")
    }

    fun testFoldOperator() = runBlocking {
        val value = (1..5).asFlow().onEach { delay(100)
}.fold(100) { val1, val2 ->
            val1 + val2
        }
        println("Collecting folded value: $value")
    }
}

fun main() {
    FlowLearning().apply {
        testToListOperator()
        testToSetOperator()
        testFirstOperator1()
        testFirstOperator2()
        testReduceOperator()
        testFoldOperator()
    }
}
```

```
Collecting list: [1, 4, 9, 16, 25]
Collecting set: [1, 4, 9, 16, 25]
Collecting first value: 1
Collecting first value with predicate: 16
Collecting reduced value: 55
Collecting folded value: 115
```

关于`reduce`操作符，可以查看下面的函数签名和实现方式：

```
/**
 * Accumulates value starting with the first element and applying [operation] to current accumulator value and each element.
 * Throws [NoSuchElementException] if flow was empty.
 */
public suspend fun <S, T : S> Flow<T>.reduce(operation: suspend (accumulator: S, value: T) -> S): S {
    var accumulator: Any? = NULL

    collect { value ->
        accumulator = if (accumulator !== NULL) {
            @Suppress("UNCHECKED_CAST")
            operation(accumulator as S, value)
        } else {
            value
        }
    }

    if (accumulator === NULL) throw NoSuchElementException("Empty flow can't be reduced")
    @Suppress("UNCHECKED_CAST")
    return accumulator as S
}
```

简单来说，`reduce`操作符会取第一个返回的数据作为一个基准，然后对之后收集的数据依次进行指定的操作`operator`，最后将结果返回。

关于`fold`操作符，查看下面的函数签名和实现：

```kotlin
/**
 * Accumulates value starting with [initial] value and applying [operation] current accumulator value and each element
 */
public suspend inline fun <T, R> Flow<T>.fold(
    initial: R,
    crossinline operation: suspend (acc: R, value: T) -> R
): R {
    var accumulator = initial
    collect { value ->
        accumulator = operation(accumulator, value)
    }
    return accumulator
}
```

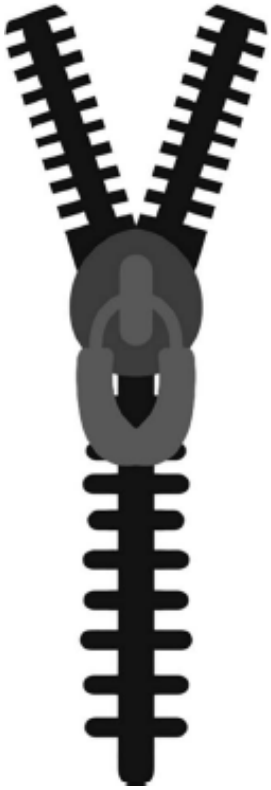
可以发现，`fold`操作符包含两个部分，一个是初始化的值，另外一个是对流中每一个数据需要进行的操作。当初始化数据是0的时候，`fold`和`reduce`基本上是等价的。

## 组合操作符

很多情况下，我们希望对两个流进行整合，组合成一个流进行操作，这个时候可以使用组合操作符。

### `zip`操作符

就像 Katlin标准库中的Sequence.zip扩展函数一样，流拥有一个`zip`操作符用于组合两个流中的相关值。



下面的代码简单展示了`zip`操作符的使用方法：

```
package com.example.coroutinelearning

import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking

class FlowLearning {
    fun testZipOperator() = runBlocking {
        val nums = (1..3).asFlow().onEach { delay(300) }
        val strs = flowOf("One", "Two", "Three").onEach {
delay(400) }

        val startTime = System.currentTimeMillis()
        nums.zip(strs){a, b ->
            "$a -> $b"
        }.collect{
            println("$it at ${System.currentTimeMillis() -
startTime} ms after start")
        }
    }
}
```

```
20
21 fun main() {
22     FlowLearning().apply {
23         testZipOperator()
24     }
25 }
```

```
1 -> One at 520 ms after start
2 -> Two at 937 ms after start
3 -> Three at 1348 ms after start
```

可以发现，zip操作符会等待进行组合，当数据不足以进行组合的时候，程序会被挂起，直到收到了两个数据进行组合并传递给下游。

如果两个流的长度不一致，zip操作符会选择端的流进行组合，较长的那一个流后序的数据会被直接舍弃。如：

```
1  package com.example.coroutinelearning
2
3  import kotlinx.coroutines.delay
4  import kotlinx.coroutines.flow.*
5  import kotlinx.coroutines.runBlocking
6
7  class FlowLearning {
8      fun testZipOperator() = runBlocking {
9          val nums = (1..3).asFlow().onEach { delay(300) }
10         val strs = flowOf("One", "Two", "Three", "Four").onEach {
11             delay(400)
12             println("Emitting $it")
13         }
14
15         val startTime = System.currentTimeMillis()
16         nums.zip(strs) { a, b ->
17             "$a -> $b"
18         }.collect {
19             println("$it at ${System.currentTimeMillis() -
   startTime} ms after start")
20         }
21     }
```

```
}

fun main() {
    FlowLearning().apply {
        testZipOperator()
    }
}
```

```
Emitting One
1 -> One at 528 ms after start
Emitting Two
2 -> Two at 935 ms after start
Emitting Three
3 -> Three at 1336 ms after start
```

### `combine`操作符

和上面的zip操作符类似，combine也是将两个流组合起来的操作符，但是combine不会忽略掉较长的流的尾部数据，而是通过计算两个流发送数据的最新状态来进行组合。

比如，当FlowA发送了一个最新的数据，此时，combine操作符会从FlowB中获取最新的发送的数据，然后将两者进行组合，继续发送给下游，反之如果FlowB发送了一个最新的数据，combine操作符也会将FlowA中目前最新发送的数据取出进行组合，然后再继续发送给下游。

```
package com.example.coroutinelearning

import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking

class FlowLearning {
    fun testCombineOperator() = runBlocking {
        val flow = flowOf(1, 2, 3).onEach { delay(1000) }
        val flow2 = flowOf("a", "b", "c", "d").onEach {
delay(1500) }
        val startTime = System.currentTimeMillis()
        flow.combine(flow2) { i, s ->
```

```
            println("Combine data at ${System.currentTimeMillis() - startTime}")
            "$i -> $s"
        }.collect {
            println(it)
        }
    }
}

fun main() {
    FlowLearning().apply {
        testCombineOperator()
    }
}
```

```
Combine data at 1680
1 -> a
Combine data at 2179
2 -> a
Combine data at 3193
3 -> b
Combine data at 4703
3 -> c
Combine data at 6214
3 -> d
```

## 展平流操作符

流表示异步接收的值序列， 所以很容易遇到这样的情况：每个值都会触发对另一个值序列的请求， 然而， 由千流具有异步的性质， 因此需要不同的展平模式， 为此， 存在一系列的流展平操作符：

**`flattenConcat`**

函数签名如下：

```
/**
 * Flattens the given flow of flows into a single flow in a sequentially manner, without interleaving nested flows.
 * This method is conceptually identical to `flattenMerge(concurrency = 1)` but has faster implementation.
 *
 * Inner flows are collected by this operator *sequentially*.
 */
@FlowPreview
public fun <T> Flow<Flow<T>>.flattenConcat(): Flow<T>
```

可以发现，该函数可以将两个或者多个流整合成一个流，首先会处理第一个流中的所有元素，等第一个流中的数据全部处理完成之后再挨个发送第二个数据流中的数据，以此类推。

```
package com.example.coroutinelearning

import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking

class FlowLearning {
    fun testFlattenConcatOperator() = runBlocking {
        val flowA = (1..5).asFlow().onEach { delay(100) }
        val flowB = flowOf("one", "two", "three","four","five").onEach { delay(150) }

        flowOf(flowA,flowB)
                .flattenConcat()
                .collect{ println(it) }
    }
}

fun main() {
    FlowLearning().apply {
        testFlattenConcatOperator()
    }
}
```

```
1
2
3
4
5
one
two
three
four
five
```

**`flattenMerge`**

将给定的流平展为单个流，对并发收集的流数量有 [并发] 限制。和上面的`flattenConcat`不同的是，`flattenMerge`会建立一个并发条件用于并发执行对应的流。

该函数可以指定一个并发执行的数量concurrency，如果 [concurrency] 大于 1，则该操作符并发收集内部流。`concurrency == 1` 这个操作符和 [flattenConcat] 是一样的。

```
package com.example.coroutinelearning

import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking

class FlowLearning {
    fun testFlattenMergeOperator() = runBlocking {
        val flowA = (1..5).asFlow().onEach { delay(100) }
        val flowB = flowOf("one", "two", "three", "four", "five").onEach { delay(150) }
        val flowC = flowOf("一", "二", "三", "四", "五").onEach { delay(50) }
        flowOf(flowA, flowB, flowC)
                .flattenMerge(2)
                .collect { println(it) }
    }
}
```

```

fun main() {
    FlowLearning().apply {
        testFlattenMergeOperator()
    }
}
```

```
1
one
2
two
3
4
three
5
一
four
二
三
四
five
五
```

可以发现，上面的代码中，flowA和flowB首先会并行执行发送数据，然后，等到flowA发送完毕之后，flowC开始和flowB并行发送数据，直到最后全部的数据都发送完毕。

### `flatMapConcat`操作符

首先来看下`flatMapConcate`操作符的源码：

```
/**
 * Transforms elements emitted by the original flow by applying [transform], that returns another flow,
 * and then concatenating and flattening these flows.
 *
 * This method is a shortcut for `map(transform).flattenConcat()`. See [flattenConcat].
 *
 * Note that even though this operator looks very familiar, we discourage its usage in a regular application-specific flows.
 * Most likely, suspending operation in [map] operator will be sufficient and linear transformations are much easier to reason about.
 */
@FlowPreview
public fun <T, R> Flow<T>.flatMapConcat(transform: suspend (value: T) -> Flow<R>): Flow<R> =
    map(transform).flattenConcat()
```

可以看到，该操作符首先会对当前流的买一个元素都应用一个`transform`变换，需要注意的是，这个`transform`返回的是一个流。接着对所有的map之后的流的流（`Flow<Flow<T>>`）应用`flattenConcat`操作符，依次链接所有的流中的元素发送给下游。

```
package com.example.coroutinelearning

import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking

class FlowLearning {
    private fun requestFlow(i : Int) = flow<String> {
        emit("Emitting $i : First")
        delay(500)
        emit("Emitting $i : Second")
    }

    fun testFlatMapConcatOperator() = runBlocking {
```

```
        val flowA = (1..5).asFlow().onEach { delay(100) }
        val startTime = System.currentTimeMillis()
        flowA.flatMapConcat { requestFlow(it) }.collect{
            println("$it at ${System.currentTimeMillis() - startTime} ms after start")
        }
    }
}

fun main() {
    FlowLearning().apply {
        testFlatMapConcatOperator()
    }
}
```

```
Emitting 1 : First at 178 ms after start
Emitting 1 : Second at 681 ms after start
Emitting 2 : First at 791 ms after start
Emitting 2 : Second at 1302 ms after start
Emitting 3 : First at 1411 ms after start
Emitting 3 : Second at 1918 ms after start
Emitting 4 : First at 2028 ms after start
Emitting 4 : Second at 2529 ms after start
Emitting 5 : First at 2634 ms after start
Emitting 5 : Second at 3147 ms after start
```

可以发现，因为使用了`flattenConcat`操作符，所有的数据都是按照顺序接收的。

### `flatMapMerge`

首先看下`flatMapMerge`操作符的源码：

```
/**
 * Transforms elements emitted by the original flow by applying [transform], that returns another flow,
 * and then merging and flattening these flows.
 *
 * This operator calls [transform] *sequentially* and then merges the resulting flows with a [concurrency]
```

```
 * limit on the number of concurrently collected flows.
 * It is a shortcut for `map(transform).flattenMerge(concurrency)`.
 * See [flattenMerge] for details.
 *
 * Note that even though this operator looks very familiar, we discourage its usage in a regular application-specific flows.
 * Most likely, suspending operation in [map] operator will be sufficient and linear transformations are much easier to reason about.
 *
 * ### Operator fusion
 *
 * Applications of [flowOn], [buffer], [produceIn], and [broadcastIn] _after_ this operator are fused with
 * its concurrent merging so that only one properly configured channel is used for execution of merging logic.
 *
 * @param concurrency controls the number of in-flight flows, at most [concurrency] flows are collected
 * at the same time. By default it is equal to [DEFAULT_CONCURRENCY].
 */
@FlowPreview
public fun <T, R> Flow<T>.flatMapMerge(
    concurrency: Int = DEFAULT_CONCURRENCY,
    transform: suspend (value: T) -> Flow<R>
): Flow<R> =
    map(transform).flattenMerge(concurrency)
```

和`flatMapConcat`类似，也是最终会调用`map`操作符将流中的每一个元素都转换成一个流，最后再调用`flattenMerge`并发的去发送流中的数据给下游。所以`flatMapMerge`自然也会带有一个关于并发数量的参数`concurrency`。

```
package com.example.coroutinelearning

import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking
```

```
6
7  class FlowLearning {
8      private fun requestFlow(i : Int) = flow<String> {
9          emit("Emitting $i : First")
10         delay(500)
11         emit("Emitting $i : Second")
12     }
13
14     fun testFlatMapMergeOperator() = runBlocking {
15         val flowA = (1..5).asFlow().onEach { delay(100) }
16         val startTime = System.currentTimeMillis()
17         flowA.flatMapMerge(2){requestFlow(it)}.collect{
18             println("$it at ${System.currentTimeMillis() -
   startTime} ms after start")
19         }
20     }
21 }
22
23 fun main() {
24     FlowLearning().apply {
25         testFlatMapMergeOperator()
26     }
27 }
```

```
1  Emitting 1 : First at 330 ms after start
2  Emitting 2 : First at 388 ms after start
3  Emitting 1 : Second at 834 ms after start
4  Emitting 3 : First at 834 ms after start
5  Emitting 2 : Second at 900 ms after start
6  Emitting 4 : First at 944 ms after start
7  Emitting 3 : Second at 1344 ms after start
8  Emitting 5 : First at 1344 ms after start
9  Emitting 4 : Second at 1446 ms after start
10 Emitting 5 : Second at 1856 ms after start
```

可以发现，上面的代码中并发数量是2，因此总是会出现两个流的数据交替输出的情况。

### `flatMapLastest`

再来看下`flatMapLatest`的函数定于：

```
/**
 * Returns a flow that switches to a new flow produced by [transform] function every time the original flow emits a value.
 * When the original flow emits a new value, the previous flow produced by `transform` block is cancelled.
 *
 * For example, the following flow:
 * ```
 * flow {
 *     emit("a")
 *     delay(100)
 *     emit("b")
 * }.flatMapLatest { value ->
 *     flow {
 *         emit(value)
 *         delay(200)
 *         emit(value + "_last")
 *     }
 * }
 * ```
 * produces `a b b_last`
 *
 * This operator is [buffered][buffer] by default and size of its output buffer can be changed by applying subsequent [buffer] operator.
 */
@ExperimentalCoroutinesApi
public inline fun <T, R> Flow<T>.flatMapLatest(@BuilderInference crossinline transform: suspend (value: T) -> Flow<R>): Flow<R> =
    transformLatest { emitAll(transform(it)) }
```

从含义上来看，首先依然会对原始流中的数据进行变换，转换成一个新的流。当接收到一个新的流的数据时，如果此时上一个元素数据还没有被处理完毕，则这个流就直接会被取消。

```kotlin
package com.example.coroutinelearning

import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking

class FlowLearning {
    private fun requestFlow(i : Int) = flow<String> {
        emit("Emitting $i : First")
        delay(500)
        emit("Emitting $i : Second")
    }

    fun testFlatMapLatestOperator() = runBlocking {
        val flowA = (1..5).asFlow().onEach { delay(100) }
        val startTime = System.currentTimeMillis()
        flowA.flatMapLatest{requestFlow(it)}.collect{
            println("$it at ${System.currentTimeMillis() - startTime} ms after start")
        }
    }
}

fun main() {
    FlowLearning().apply {
        testFlatMapLatestOperator()
    }
}
```

```
Emitting 1 : First at 397 ms after start
Emitting 2 : First at 513 ms after start
Emitting 3 : First at 625 ms after start
Emitting 4 : First at 732 ms after start
Emitting 5 : First at 838 ms after start
Emitting 5 : Second at 1346 ms after start
```

从最后的结果来看。当原始的流发送数据1的时候，下游的`flatMapLatest`将这个数据1转换成两个字符串，只不过中间会挂起500毫秒。因此首先这个转换过之后的流会输出`emitting 1: First`的数据，然后挂起500毫秒。当接收到数据2的时候，前一个数据1转换之后的流还处在挂起状态，此时直接被取消，马上开始处理数据2的工作流，因此也会马上打印`emiting 2: First`的数据，然后挂起500毫秒。过了100毫秒之后接收到了数据3，此时数据2的工作流依然是挂起状态，也会被直接取消，马上开始数据3的工作流。以此类推，直到最后的数据5可以完整的输出数据。

# 流的异常处理

当运算符中的发射器或代码抛出异常时， 有几种处理异常的方法：

- try/catch块
- catch函数

比如可以在收集元素时进行异常捕获。

```
package com.example.coroutinelearning

import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking
import java.lang.Exception

class FlowLearning {
    fun simpleFlow() = flow<Int> {
        for (i in 1..3){
            println("Emitting $i")
            emit(i)
        }
    }

    fun testFlowException() = runBlocking {
        try{
            simpleFlow().collect{
                println("Collecting $it...")
                check(it <= 1) {
                    "Collect $it"
```

```
                }
            }
        } catch (e: Exception) {
            println("Caught $e")
        }
    }
}

fun main() {
    FlowLearning().apply {
        testFlowException()
    }
}
```

```
Emitting 1
Collecting 1...
Emitting 2
Collecting 2...
Caught java.lang.IllegalStateException: Collect 2
```

也有可能在构建流的时候产生了异常。

```
package com.example.coroutinelearning

import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking
import java.lang.ArithmeticException
import java.lang.Exception

class FlowLearning {
    fun simpleFlow() = flow<Int> {
        for (i in 1..3) {
            println("Emitting $i")
            if (i == 2) {
                throw ArithmeticException("Div 0")
            } else {
                emit(i)
            }
```

```
18
19          }
20      }
21
22      fun testFlowException() = runBlocking {
23          simpleFlow().catch { e: Throwable ->
24              println("Caught $e")
25              emit(100)
26          }.collect {
27              println("Collecting $it...")
28          }
29      }
30  }
31
32  fun main() {
33      FlowLearning().apply {
34          testFlowException()
35      }
36  }
```

```
1  Emitting 1
2  Collecting 1...
3  Emitting 2
4  Caught java.lang.ArithmeticException: Div 0
5  Collecting 100...
```

这个时候我们可以在流的下游进行异常捕获，使用`catch`操作符。同时，catch到异常之后，我们也可以向下游发送一个临时的补救性质的数据，依然采用的是`emit`方法。需要注意，一旦发生了异常，流的流动就会被打断，因此上面的代码是不会发送数据3的。

## 流的完成

当流收集完成时（普通情况或异常情况）， 它可能需要执行一个动作。

- 命令式finally块
- onCompletion声明式处理

下面的代码显示的是使用`finally`代码块用来处理流的完成事件：

```kotlin
package com.example.coroutinelearning

import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.runBlocking

class FlowLearning {
    fun simpleFlow() = flow<Int> {
        for (i in 1..3) {
            emit(i)
        }
    }

    fun testFlowComplete() = runBlocking {
        try {
            simpleFlow().collect {
                println("Collecting $it...")
            }
        } finally {
            println("Done!")
        }
    }
}

fun main() {
    FlowLearning().apply {
        testFlowComplete()
    }
}
```

```
Collecting 1...
Collecting 2...
Collecting 3...
Done!
```

下面的代码显示的是使用`onCompletion`操作符来关注流的完成情况：

```kotlin
package com.example.coroutinelearning
```

```
2
3  import kotlinx.coroutines.flow.collect
4  import kotlinx.coroutines.flow.flow
5  import kotlinx.coroutines.flow.onCompletion
6  import kotlinx.coroutines.runBlocking
7
8  class FlowLearning {
9      fun simpleFlow() = flow<Int> {
10         for (i in 1..3) {
11             emit(i)
12         }
13     }
14
15     fun testFlowComplete() = runBlocking {
16         simpleFlow().onCompletion {
17             println("Done!")
18         }.collect {
19             println("Collecting $it...")
20         }
21     }
22 }
23
24 fun main() {
25     FlowLearning().apply {
26         testFlowComplete()
27     }
28 }
```

```
1  Collecting 1...
2  Collecting 2...
3  Collecting 3...
4  Done!
```

当然有时候流会在数据构建变换和发送的过程中出现异常，`onCompletion`也是可以检测到异常的出现的。注意仅仅是异常的出现，但是不会捕获异常。

例如

```
1  package com.example.coroutinelearning
```

```

import kotlinx.coroutines.flow.catch
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.flow.onCompletion
import kotlinx.coroutines.runBlocking
import java.lang.ArithmeticException

class FlowLearning {
    fun simpleFlow() = flow<Int> {
        for (i in 1..3) {
            if (i != 2)
                emit(i)
            else {
                throw ArithmeticException("Div 0")
            }
        }
    }

    fun testFlowComplete() = runBlocking {
        simpleFlow().onCompletion { cause: Throwable? ->
            if (cause != null) {
                println("Flow Completed Exceptionally...")
            } else {
                println("Flow Completed Done!")
            }
        }.collect {
            println("Collecting $it...")
        }
    }
}

fun main() {
    FlowLearning().apply {
        testFlowComplete()
    }
}
```

```
Collecting 1...
```

```
Flow Completed Exceptionally...
Exception in thread "main" java.lang.ArithmeticException: Div 0
    at com.example.coroutinelearning.FlowLearning$simpleFlow$1.invokeSuspend(CoroutineMain.kt:16)
    at com.example.coroutinelearning.FlowLearning$simpleFlow$1.invoke(CoroutineMain.kt)
    at kotlinx.coroutines.flow.SafeFlow.collectSafely(Builders.kt:61)
    at kotlinx.coroutines.flow.AbstractFlow.collect(Flow.kt:212)
    at kotlinx.coroutines.flow.FlowKt__EmittersKt$onCompletion$$inlined$unsafeFlow$1.collect(SafeCollector.common.kt:114)
    at com.example.coroutinelearning.FlowLearning$testFlowComplete$1.invokeSuspend(CoroutineMain.kt:39)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at kotlinx.coroutines.EventLoopImplBase.processNextEvent(EventLoop.common.kt:274)
    at kotlinx.coroutines.BlockingCoroutine.joinBlocking(Builders.kt:84)
    at kotlinx.coroutines.BuildersKt__BuildersKt.runBlocking(Builders.kt:59)
    at kotlinx.coroutines.BuildersKt.runBlocking(Unknown Source)
    at kotlinx.coroutines.BuildersKt__BuildersKt.runBlocking$default(Builders.kt:38)
    at kotlinx.coroutines.BuildersKt.runBlocking$default(Unknown Source)
    at com.example.coroutinelearning.FlowLearning.testFlowComplete(CoroutineMain.kt:21)
```

```
    at com.example.coroutinelearning.CoroutineMainKt.main(CoroutineMain.kt:36)
    at com.example.coroutinelearning.CoroutineMainKt.main(CoroutineMain.kt)
```

可以发现，异常还是被抛了出来，所以`omCompletion`是无法捕获异常的，只能观察到异常的产生。

如果需要捕获异常，还是需要使用`catch`操作符。

```
package com.example.coroutinelearning

import kotlinx.coroutines.flow.catch
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.flow.onCompletion
import kotlinx.coroutines.runBlocking
import java.lang.ArithmeticException

class FlowLearning {
    fun simpleFlow() = flow<Int> {
        for (i in 1..3) {
            if (i != 2)
                emit(i)
            else {
                throw ArithmeticException("Div 0")
            }
        }
    }

    fun testFlowComplete() = runBlocking {
        simpleFlow().onCompletion { cause: Throwable? ->
            if (cause != null) {
                println("Flow Completed Exceptionally...")
            } else {
                println("Flow Completed Done!")
            }
```

```
        }.catch {e: Throwable ->
            println("Caught $e")
        }.collect {
            println("Collecting $it...")
        }
    }
}

fun main() {
    FlowLearning().apply {
        testFlowComplete()
    }
}
```

```
Collecting 1...
Flow Completed Exceptionally...
Caught java.lang.ArithmeticException: Div 0
```

如果在下游的数据处理中产生了异常，`onCompletion`也是可以感知到的。

```
package com.example.coroutinelearning

import kotlinx.coroutines.flow.catch
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.flow.onCompletion
import kotlinx.coroutines.runBlocking
import java.lang.ArithmeticException

class FlowLearning {
    fun simpleFlow() = flow<Int> {
        for (i in 1..3) {
            emit(i)
        }
    }

    fun testFlowComplete() = runBlocking {
```

```
        simpleFlow().onCompletion { cause: Throwable? ->
            if (cause != null) {
                println("Flow Completed Exceptionally...")
            } else {
                println("Flow Completed Done!")
            }
        }.collect {
            println("Collecting $it...")
            if (it == 2)
                throw ArithmeticException("Div 0")
        }
    }
}

fun main() {
    FlowLearning().apply {
        testFlowComplete()
    }
}
```

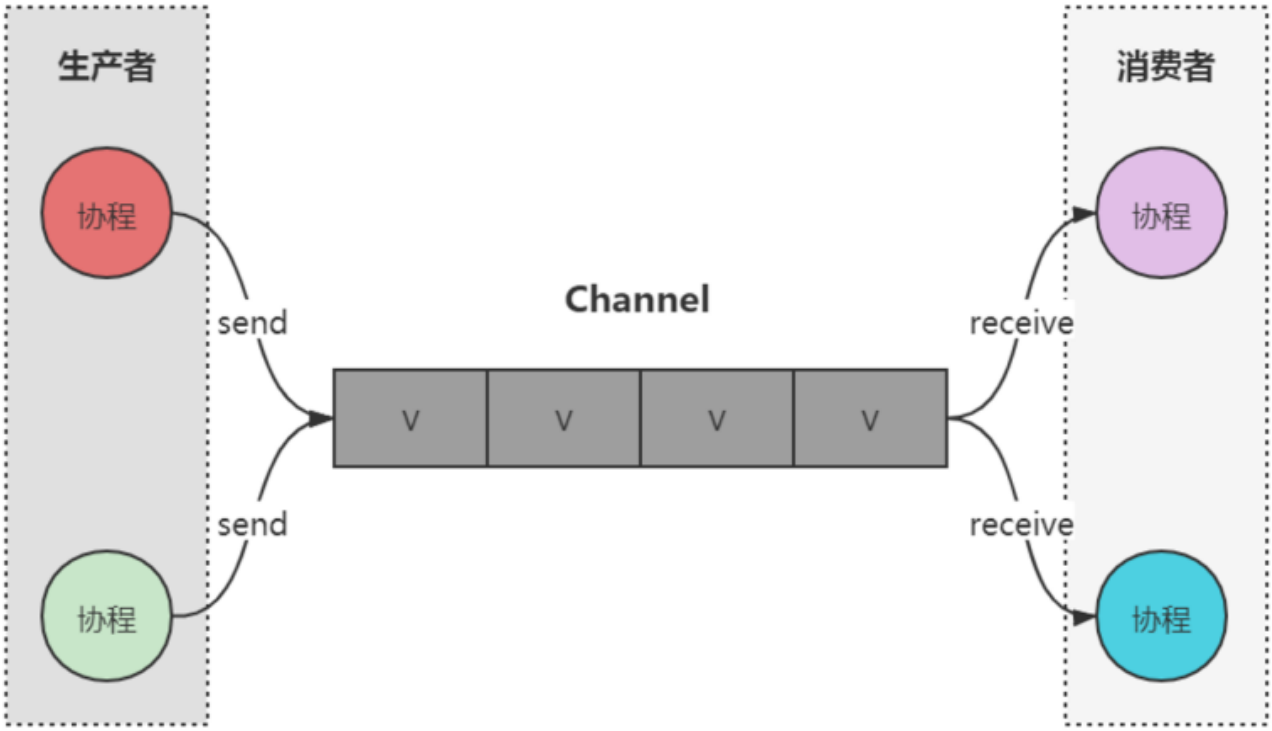
```
Collecting 1...
Collecting 2...
Flow Completed Exceptionally...
Exception in thread "main" java.lang.ArithmeticException: Div 0
```

可以发现，最后在收集阶段抛出的异常，`onCompletion`同样是可以感知到的，但是不会捕获和处理异常。

# Channel（通道）

## 认识Channel

Channel实际上是一个**并发安全的队列**，它可以用来连接协程，实现不同协程的通信。




下面的代码展示了一个简单的Channel的使用：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.Channel

class ChannelLearning {
    fun testChannelBuild() = runBlocking {

        val channel: Channel<Int> = Channel()

        val producer = GlobalScope.launch {
            var i = 0
            while (true) {
                delay(1000)
                channel.send(i)
                println("Sending $i")
                i += 1

            }
        }
```

```
21
22          val consumer = GlobalScope.launch {
23              while (true) {
24                  val element = channel.receive()
25                  println("Receiving $element...")
26              }
27          }
28
29          joinAll(producer, consumer)
30      }
31  }
32
33  fun main() {
34      ChannelLearning().apply {
35          testChannelBuild()
36      }
37  }
```

```
1   Sending 0
2   Receiving 0...
3   Sending 1
4   Receiving 1...
5   Sending 2
6   Receiving 2...
7   Sending 3
8   Receiving 3...
9   Sending 4
10  Receiving 4...
11  Sending 5
12  Receiving 5...
13  ...
```

在上面的代码中，首先创建了一个Channel的实例，指定数据类型是Int类型。接着我们构建了两个协程，一个协程是生产者，另外一个协程是消费者。对于生产者来说，每隔一秒钟会向channel内部发送一个数据（使用`channel.send`方法），对于生产者来说会不断地向channel索要数据（使用`channel.receive`方法）。所以就会看到最后的输出，sending和receiving一直交替出现。

Channel的send和receive函数都是挂起函数。

## Channel的容量

Channel实际上就是一个队列， 队列中一定存在缓冲区， 那么一旦这个缓冲区满了，并且也一直没有人调用 receive并取走函数， send就需要挂起。 故意让接收端的节奏放慢， 发现send总是会挂起， 直到 receive之后才会继续往下执行。

同样，如果缓冲区是空的，那么receive函数会被挂起，直到有数据被放入缓冲区。

在Channel的构造函数中可以指定缓冲区的大小。如下面的代码：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.Channel

class ChannelLearning {
    fun testChannelBuild() = runBlocking {

        val channel: Channel<Int> = Channel(3)

        val producer = GlobalScope.launch {
            var i = 0
            while (true) {
                delay(100)
                channel.send(i)
                println("Sending $i")
                i += 1

            }
        }

        val consumer = GlobalScope.launch {
            while (true) {
                delay(200)
                val element = channel.receive()
                println("Receiving $element...")
            }
```

```
        }

        joinAll(producer, consumer)
    }
}

fun main() {
    ChannelLearning().apply {
        testChannelBuild()
    }
}
```

在上面的代码中，首先建构了一个大小是3的Channel对象，然后在构建一个每隔100毫秒发送一个数据的生产者，再构建一个每隔200毫秒接收数据的消费者。很明显，消费者的速度是比不上生产者的，那么最后缓冲区一定会满。

```
Sending 0
Receiving 0...
Sending 1
Sending 2
Receiving 1...
Sending 3
Sending 4
Receiving 2...
Sending 5
Receiving 3...
Sending 6
Receiving 4...
Sending 7
Receiving 5...
Sending 8
Receiving 6...
Sending 9
Receiving 7...
...
```

可以看到，当缓冲区的大小是3的时候，生产者会尽可能先填满缓冲区，所以一开始生产者的sending消息出现的比较多。但是由于消费者的效率低于生产者，所以当缓冲区满了之后，生产者就开始等待消费者取出数据，然后再向Channel内发送数据，速度也会受到消费者速度的影响而和消费者的步调保持一致，所以后面就是sending和receiving的字样交替出现。

## 迭代Channel

Channel本身确实像序列， 所以我们在读取的时候可以直接获取一个Channel的iterator。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.Channel

class ChannelLearning {
    fun testChannelIterator() = runBlocking {

        val channel: Channel<Int> = Channel(Channel.UNLIMITED)

        val producer = GlobalScope.launch {
            for (i in 1..5) {
                val data = i * i
                println("sending $data")
                channel.send(data)
            }
            println("Sending done!")
        }

        val consumer = GlobalScope.launch {
            val iterator = channel.iterator()
            while (iterator.hasNext()) {
                val element = iterator.next()
                println("Receiving $element")
                delay(100)
```

```
                if (channel.isEmpty) { // 注意：如果缺少这个if判断，consumer这个协程将会一直循环
                    break
                }
            }
            println("done")
        }

        joinAll(producer, consumer)
    }
}

fun main() {
    ChannelLearning().apply {
        testChannelIterator()
    }
}
```

```
sending 1
sending 4
sending 9
sending 16
sending 25
Sending done!
Receiving 1
Receiving 4
Receiving 9
Receiving 16
Receiving 25
done
```

再上面的代码中，首先会构建一个长度无限的Channel队列，然后生产者会快速地向channel内部发送5个数据。在消费者这边，可以通过iteration迭代器来依次获取channel内部的数据，然后输出。

当然也可以使用`in`关键字来对channel进行遍历，如下：

```
package com.example.coroutinelearning

```

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.Channel

class ChannelLearning {
    fun testChannelIterator() = runBlocking {

        val channel: Channel<Int> = Channel(Channel.UNLIMITED)

        val producer = GlobalScope.launch {
            for (i in 1..5) {
                val data = i * i
                println("sending $data")
                channel.send(data)
            }
            println("Sending done!")
        }

        val consumer = GlobalScope.launch {
            for (element in channel) {
                println("Received: $element")
                if (channel.isEmpty) { // 注意：如果缺少这个if判断,
consumer这个协程将会一直循环
                    break
                }
            }
        }

        joinAll(producer, consumer)
    }
}

fun main() {
    ChannelLearning().apply {
        testChannelIterator()
    }
}
```

```
sending 1
sending 4
sending 9
sending 16
sending 25
Sending done!
Received: 1
Received: 4
Received: 9
Received: 16
Received: 25
```

## produce与actor

构造生产者与消费者的便捷方法。

我们可以通过produce方法启动一个生产者协程， 并返回一个ReceiveChannel, 其他协程就可以用这个Channel来接收数据了。 反过来， 我们可以用actor启动一个消费者协程。

利用produce启动一个生产者协程的示例如下：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.ReceiveChannel
import kotlinx.coroutines.channels.produce

class ChannelLearning {
    fun testChannelProduce() = runBlocking {
        val receiverChannel : ReceiveChannel<Int> =
GlobalScope.produce {
            repeat(10) {
                delay(100)
                send(it)
            }
        }

        val consumer = GlobalScope.launch {
```

```
            for (i in receiverChannel) {
                println("received : $i")
            }
        }
        consumer.join()
    }
}

fun main() {
    ChannelLearning().apply {
        testChannelProduce()
    }
}
```

```
received : 0
received : 1
received : 2
received : 3
received : 4
received : 5
received : 6
received : 7
received : 8
received : 9
```

在上面的代码中，通过`GlobalScope.produce`的方法去获取一个`ReceiveChannel`实例，在这个实例中，我们每隔100毫秒依次发送0至9这十个数据。

消费者协程则和之前的消费者协程类似，通过`in`关键字逐个去获取`ReceiveChannel`中的数据，然后打印输出。

既然有方法可以很方便的创建一个生产者协程，那么一定会有对应的创建消费者的方便的方法，这个就是`actor`方法。

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.ReceiveChannel
import kotlinx.coroutines.channels.SendChannel
```

```
import kotlinx.coroutines.channels.actor
import kotlinx.coroutines.channels.produce

class ChannelLearning {
    fun testChannelActor() = runBlocking {
        val sendChannel: SendChannel<Int> = GlobalScope.actor {
            while (true) {
                val element = receive()
                println("Receiving : $element")
            }
        }

        val producer = GlobalScope.launch {
            for (i in 0 .. 3) {
                sendChannel.send(i)
            }
        }

        producer.join()
    }
}

fun main() {
    ChannelLearning().apply {
        testChannelActor()
    }
}
```

```
Receiving : 0
Receiving : 1
Receiving : 2
Receiving : 3
```

## Channel关闭

produce和actor返回的Channel都会随着对应的协程执行完毕而关闭， 也正是这样， Channel才被称为**热数据流**。

对于一个Channel, 如果我们调用了它的close方法， 它会立即停止接收新元素， 也就是说这时它的`isClosedForSend`会立即返回true。而由千Channel缓冲区的存在， 这时候可能还有一些元素没有被处理完，因此要等所有的元素都被读取之后`isClosedForReceive`才会返回true。

Channel的生命周期最好由主导方来维护， 建议**由主导的一方实现关闭**。

下面的代码显示了Channel关闭时的各个状态位标记（`isClosedForSend`和`isClosedForReceive`）的情况：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

class ChannelLearning {
    fun testChannelClose() = runBlocking {
        val channel = Channel<Int>(3)
        //生产者
        val producer = GlobalScope.launch {
            // 发送三个数据
            List(3) {
                channel.send(it)
                println("send $it")
            }

            // 发送完数据之后马上关闭通道
            channel.close()
            // 打印此时的Channel的状态
            println("""close channel.
                |  - ClosedForSend: ${channel.isClosedForSend}
                |  - ClosedForReceive: ${channel.isClosedForReceive}""".trimMargin())
        }

        //消费者
        val consumer = GlobalScope.launch {
            // 取出所有的数据
            for (element in channel){
```

```
                println("receive $element")
                delay(1000)
            }
            // 打印此时的Channel的状态
            println("""After Consuming.
                |   - ClosedForSend: ${channel.isClosedForSend}
                |   - ClosedForReceive:
${channel.isClosedForReceive}""".trimMargin())
        }

        joinAll(producer, consumer)
    }
}

fun main() {
    ChannelLearning().apply {
        testChannelClose()
    }
}
```

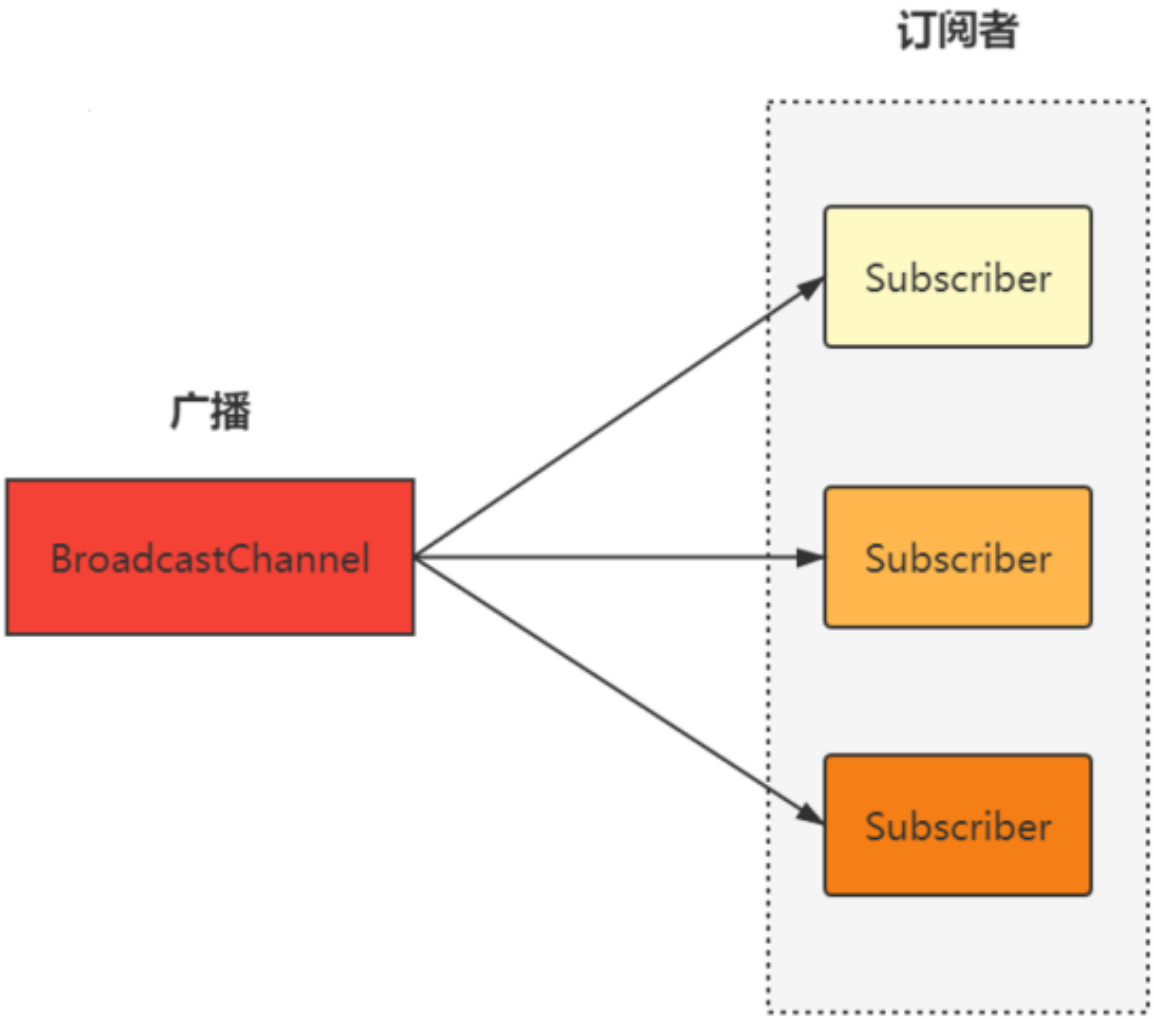
```
send 0
receive 0
send 1
send 2
close channel.
  - ClosedForSend: true
  - ClosedForReceive: false
receive 1
receive 2
After Consuming.
   - ClosedForSend: true
   - ClosedForReceive: true
```

可以看出，当在发送端发送完全部的数据之后，关闭通道会将通道的`isClosedForSend`设置为true，表示通道的的发送接口已经关闭了。但是由于消费者还没有将全部的数据取出来，因此通道的`isClosedForReceive`仍然是false，表明此时通道的接收端的接口还是处于开放的状态，接收端的协程还是可以正确地从通道中取出数据。

在消费者协程中，等到全部的数据都取出来之后，此时通道内部没有了数据，这个时候因为通道发送端已经关闭了，不会再有新的数据进来了，因此一旦通道内部没有了数据，接收端的接口也就可以关闭了。因此最后的结果就是`isClosedForReceive`的标志位被设置成了true。

# BroadcastChannel

前面提到， 发送端和接收端在Channel 中存在一对多的情形， 从数据处理本身来讲， 虽然有多个接收端， 但是同一个元素只会被一个接收端读到。广播则不然，**多个接收端不存在互斥行为**。




```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

```

```
class ChannelLearning {
    fun testBroadcastChannel() = runBlocking {
        // 创建一个BroadcastChannel对象
        val broadcastChannel = BroadcastChannel<Int>
(Channel.BUFFERED)
        val producer = GlobalScope.launch {
            List(3){
                delay(100)
                broadcastChannel.send(it)
            }
            broadcastChannel.close()
        }

        // 生成三个独立的消费者协程，每一个协程都可以向BroadcastChannel索
要数据。
        val consumerList = List(3){ index ->
            GlobalScope.launch {
                // 从BroadcastChannel中获取一个独立的接收数据的通道。
                val receiveChannel =
broadcastChannel.openSubscription()
                for (i in receiveChannel){
                    println("[#$index] received: $i")
                }
            }
        }
        producer.join()
        consumerList.joinAll()
    }
}

fun main() {
    ChannelLearning().apply {
        testBroadcastChannel()
    }
}
```

```
[#0] received: 0
[#1] received: 0
[#2] received: 0
[#0] received: 1
[#1] received: 1
[#2] received: 1
[#0] received: 2
[#1] received: 2
[#2] received: 2
```

上面的代码中，首先构造出了一个生产者协程和三个相互独立的消费者协程。它们之间的关系是一对多的关系，即生产者发送的数据需要被每一个消费者协程接收到。在生产者协程内部，我们每隔100毫秒发送一次数据，然后关闭BroadcastChannel的输入端。在消费者协程内部需要先获取一个BroadcastChannel的接收通道（ReceiveChannel），每一个接收通道都是独立的，然后再从这个通道中获取发送的数据。

观察一下`openSubscription`的签名，如下：

```
public interface BroadcastChannel<E> : SendChannel<E> {
    /**
     * Subscribes to this [BroadcastChannel] and returns a channel to receive elements from it.
     * The resulting channel shall be [cancelled] [ReceiveChannel.cancel] to unsubscribe from this
     * broadcast channel.
     */
    public fun openSubscription(): ReceiveChannel<E>

    ...
}
```

发现所谓的订阅操作只是获取一个新的ReceiveChannel对象而已。

从最后的数据输出可以看出来，每一个消费者协程都可以拿到所有的数据，而不是每一个数据都只能被消费一次。

构建BroadcastChannel的方法不止上面一种，也可以从普通的Channel进行转化。

```kotlin
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

class ChannelLearning {
    fun testBroadcastChannel() = runBlocking {
        // 构建一个普通的Channel
        val channel = Channel<Int>()
        // 将这个Channel转换成一个BroadcastChannel，缓冲区的大小是3
        val broadcastChannel = channel.broadcast(3)
        val producer = GlobalScope.launch {
            List(3){
                delay(100)
                broadcastChannel.send(it)
            }
            broadcastChannel.close()
        }

        // 生成三个独立的消费者协程，每一个协程都可以向BroadcastChannel索要数据。
        val consumerList = List(3){ index ->
            GlobalScope.launch {
                // 从BroadcastChannel中获取一个独立的接收数据的通道。
                val receiveChannel = broadcastChannel.openSubscription()
                for (i in receiveChannel){
                    delay(500)
                    println("[#$index] received: $i")
                }
            }
        }
        producer.join()
        consumerList.joinAll()
    }
}

fun main() {
    ChannelLearning().apply {
```

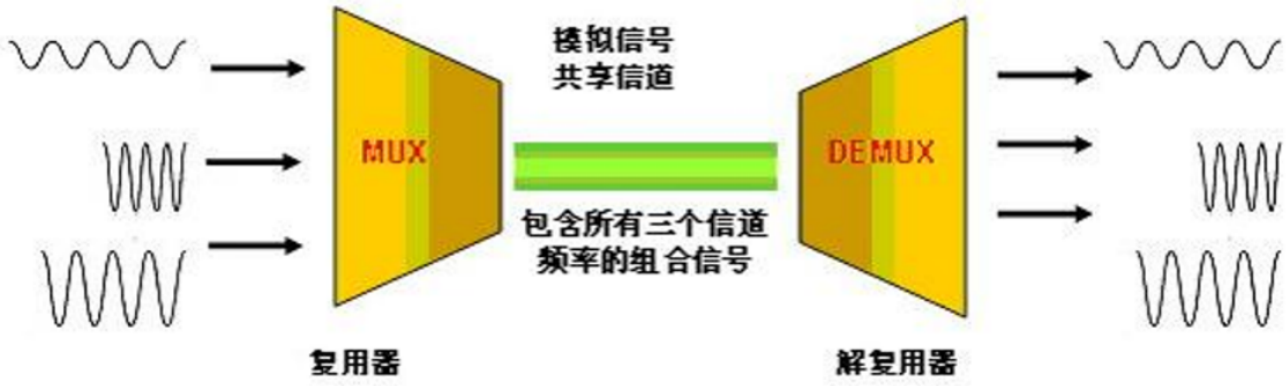
```
38          testBroadcastChannel()
39      }
40  }
```

结果和上面的是一致的：

```
[#0] received: 0
[#1] received: 0
[#2] received: 0
[#0] received: 1
[#1] received: 1
[#2] received: 1
[#1] received: 2
[#0] received: 2
[#2] received: 2
```

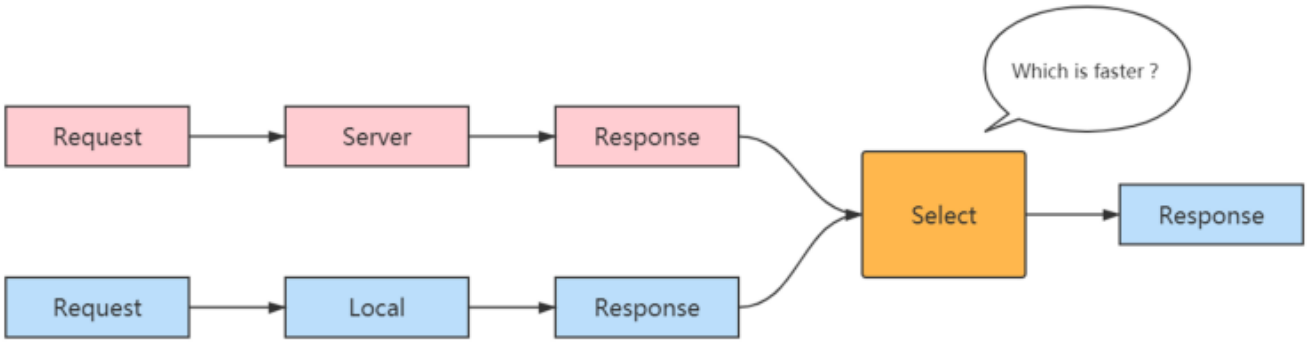
# select-多路复用

数据通信系统或计算机网络系统中， 传输媒体的带宽或容量往往会大千传输单一信号的需求， 为了有效地利用通信线路， 希望一个信道同时传输多路信号，这就是所谓的多路复用技术(Multiplexing)。




## 复用多个await

两个API分别从网络和本地缓存获取数据， 期望哪个先返回就先用哪个做展示。




比如有下面的代码，本地缓存有一份Json文件，网络端也有一份Json文件，可能在某一个时刻两个文件同时开始读取，那么为了用户体验，谁最优先读取完成就先展示哪一份文件。

注意，因为读取本地文件的速度肯定比网络下载快，因为需要人为地进行挂起操作，来模拟读取耗时。

网络文件使用的是`http://guolin.tech/api/china`，是一份中国省市的数据。

本地使用的文件可以是任意的文本文件。

```
import android.content.pm.PackageManager
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.core.app.ActivityCompat
import androidx.core.content.ContextCompat
import androidx.lifecycle.lifecycleScope
import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.coroutines.*
import kotlinx.coroutines.selects.select
import okhttp3.OkHttpClient
import okhttp3.Request
import okhttp3.Response
import java.io.File
import java.io.IOException
import java.util.jar.Manifest

class MainActivity : AppCompatActivity() {

    private val URL = "http://guolin.tech/api/china"
```

```
20
21      private val LOCAL = "/storage/emulated/0/Android/local.json"
22
23      override fun onCreate(savedInstanceState: Bundle?) {
24          super.onCreate(savedInstanceState)
25          setContentView(R.layout.activity_main)
26
27          btn.setOnClickListener {
28              lifecycleScope.launch {
29                  selectAwait()
30              }
31          }
32      }
33
34      private fun getUserFromLocal() =
    lifecycleScope.async(Dispatchers.IO) {
35          delay(1000) // 人为地模拟读取本地文件延迟
36          File(LOCAL).readText()
37      }
38
39      private fun getUserFromInternet(url: String) =
    lifecycleScope.async(Dispatchers.IO) {
40
41          var data = ""
42          try {
43              val client = OkHttpClient()
44              val request = Request.Builder().url(url).build()
45
46              val response: Response? = try {
47                  client.newCall(request).execute()
48              } catch (e: IOException) {
49                  e.printStackTrace()
50                  null
51              }
52
53              data = response?.body()?.string() ?: ""
54          } catch (e: Exception) {
55              e.printStackTrace()
56          }
```

```
        data
    }

    private suspend fun selectAwait() {
        val localRequest = getUserFromLocal()
        val remoteRequest = getUserFromInternet(URL)

        val response: ResponseData<String> = select {
            localRequest.onAwait { ResponseData(it, true) }
            remoteRequest.onAwait { ResponseData(it, false) }
        }
        withContext(Dispatchers.Main) {
            tv_content.text = response.value
        }
    }
}

data class ResponseData<T>(val value: T, val isLocal: Boolean)
```

在上面的代码中，设计了一个按钮，按钮按下之后会进行通道的多路复用`selectAwait`方法。这一部分是在协程中执行的。

在`selectAwait()`函数内部，分别启动了两个IO的协程，其中一个是`getInfoFromLocal`用以从手机上的文件种读取相关的信息，另外一个是`getInfoFromInternet`，用以从网络上下载相关的信息。因为从手机上读取文件会比从网络上下载快很多，因此在从手机上读取的过程中，人为的挂起1秒钟。

然后我们使用`select`方法进行多路复用。对两个协程的结果进行监听，哪一个协程的结果优先返回就使用哪一个作为最后的结果。因为两个协程返回的都是String类型，我们对最后的结果进行包装，包装成一个`ResponseData`类，并返回结果。然后在主线程中展示最后的数据结果。

可以发现，因为我们有人为的阻塞，所以几乎总是会展示网络文本。如果将人为的阻塞去掉，那么几乎总是会展示本地文本。

## 复用多个Channel

跟await类似， 多路复用会接收到最快的那个channel消息。

```
package com.example.coroutinelearning

import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.channels.Channel
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking
import kotlinx.coroutines.selects.select

class CoroutineLearning {
    fun testSelectChannel() = runBlocking {
        val channels = listOf(Channel<Int>(), Channel<Int>())

        val job1 = GlobalScope.launch {
            delay(100)
            channels[0].send(200)
        }

        val job2 = GlobalScope.launch {
            delay(50)
            channels[1].send(100)
        }

        val result = select<Int?> {
            channels.forEach { channel ->
                channel.onReceive { it }
            }
        }

        println(result)
    }
}

fun main() {
    CoroutineLearning().apply {
```

```
        testSelectChannel()
    }
}
```

```
100
```

很明显，因为job2的等待时间更少，因此会更快的将数据发送出去，在`select`中也是最快的收集到job2发送过来的数据的，这里使用`onReceive`方法来进行数据接收。

## SelectClause

我们怎么知道哪些事件可以被select呢？其实所有能够被select的事件都是SelectClauseN类型，包括：

- SelectClause0: 对应事件没有返回值， 例如join没有返回值， 那么onJoin就是SelectClauseN类型。 使用时，`onJoin`的参数是一个无参函数。
- SelectClause1 :对应事件有返回值， 前面的onAwait和onReceive都是此类情况。
- SelectClause2: 对应事件有返回值， 此外还需要一个额外的参数，例如Channel.onSend有两个参数，第一个是Channel数据类型的值， 表示即将发送的值；第二个是发送成功时的回调参数。

如果我们想要确认挂起函数是否支持select，只需要查看其**是否存在对应的SelectClauseN类型可回调即可**。

### selectClause0：`onJoin`

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.selects.select

class CoroutineLearning {
    fun testSelectClause0() = runBlocking {

        val job1 = GlobalScope.launch {
            delay(100)
            println("Job1")
```

```
        }
        val job2 = GlobalScope.launch {
            delay(10)
            println("Job2")
        }

        select<Unit> {
            job1.onJoin {
                println("Job1 onJoin")
            }
            job2.onJoin {
                println("Job2 onJoin")
            }
        }
        yield()
    }
}

fun main() {
    CoroutineLearning().apply {
        testSelectClause0()
    }
}
```

```
Job2
Job2 onJoin
```

可以发现，最后是Job2先执行完毕，`onJoin`传入的是一个无参数的lambda表达式。

### selectClause1：`onAwait`和`onReceive`

和前面的代码一样，上面的两个方法需要传入带有一个参数的lambda表达式。

### selectClause2：`onSend`

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
```

```
import kotlinx.coroutines.channels.Channel
import kotlinx.coroutines.selects.select

class CoroutineLearning {
    fun testSelectClause2() = runBlocking {
        val channels = listOf(Channel<Int>(), Channel<Int>())
        println(channels)

        launch(Dispatchers.IO) {
            select<Unit?> {
                val job1 = launch {
                    delay(10)
                    channels[0].onSend(200) { sendChannel ->
                        println("sent on $sendChannel")
                    }
                }

                val job2 = launch {
                    delay(100)
                    channels[1].onSend(100) { sendChannel ->
                        println("sent on $sendChannel")
                    }
                }
            }
        }

        GlobalScope.launch {
            println(channels[0].receive())
        }

        GlobalScope.launch {
            println(channels[1].receive())
        }
        yield()
    }
}

fun main() {
    CoroutineLearning().apply {
```

```
        testSelectClause2()
    }
}
```

代码输出如下：

```
[RendezvousChannel@573fd745{EmptyQueue},
RendezvousChannel@5d6f64b1{EmptyQueue}]
200
sent on RendezvousChannel@573fd745{EmptyQueue}
```

在上面的代码中，首先会创建两个通道Channel对象，然后启动三个协程，其中一个用来发送数据，另外两个分别监听上面创建的两个通道对象。

在发送数据的协程中，又会在`select`方法中开启了两个协程，这一次，`select`方法会优先选择最快执行完成的协程。由于job1等待的时间更短，因此job1会优先执行，然后在发送数据时，使用`onSend`方法。`onSend`方法需要两个参数，第一个参数是发送的数据，第二个参数是发送成功时候的回调函数，这里只是简单的进行了打印输出。

从最后的结果来看，的确是job1的数据被发送了出来，job2的数据并没有被发送出来或者接收到。

## Flow实现多路复用

多数情况下， 我们可以通过构造合适的Flow来实现多路复用的效果。

如下的代码：

```
import android.content.pm.PackageManager
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.core.app.ActivityCompat
import androidx.core.content.ContextCompat
import androidx.lifecycle.lifecycleScope
import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.selects.select
import okhttp3.OkHttpClient
```

```
import okhttp3.Request
import okhttp3.Response
import java.io.File
import java.io.IOException
import java.util.jar.Manifest

class MainActivity : AppCompatActivity() {

    private val URL = "http://guolin.tech/api/china"

    private val LOCAL = "/storage/emulated/0/Android/local.json"

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        btn.setOnClickListener {
            lifecycleScope.launch {
                listOf<() -> Deferred<String>>(::getUserFromLocal,
::getUserFromInternet).map { function ->
                    function.call()
                }.map { deferred ->
                    flow { emit(deferred.await()) }
                }.merge().collect {
                    println(it)
                }
            }
        }
    }

    private fun getUserFromLocal() =
lifecycleScope.async(Dispatchers.IO) {
        delay(1000) // 人为地模拟读取本地文件延迟
        File(LOCAL).readText()
    }

    private fun getUserFromInternet() =
lifecycleScope.async(Dispatchers.IO) {

```

```
        var data = ""
        try {
            val client = OkHttpClient()
            val request = Request.Builder().url(URL).build()

            val response: Response? = try {
                client.newCall(request).execute()
            } catch (e: IOException) {
                e.printStackTrace()
                null
            }

            data = response?.body()?.string() ?: ""
        } catch (e: Exception) {
            e.printStackTrace()
        }
        data
    }
}

data class ResponseData<T>(val value: T, val isLocal: Boolean)
```

可以看到，上面的代码首先会构建包含需要调用的函数的list，然后分别调用map操作符调用函数，然后再将得到的deferred对象构建出流并发射出去，然后再将流进行整合并收集。

在收集的过程中，可以对数据进行处理，从而实现多路复用的效果。

# 并发安全

我们使用线程在解决并发问题的时候总是会遇到线程安全的问题， 而Java平台上的Kotlin协程实现免不了存在并发调度的情况， 因此线程安全同样值得留意。

比如有如下的**不安全的协程自增操作**：

```
package com.example.coroutinelearning

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.Channel
```

```
import kotlinx.coroutines.selects.select

class CoroutineLearning {
    fun testUnsafeConcurrent() = runBlocking {
        var count = 0
        List(1000) {
            GlobalScope.launch { count ++ }
        }.joinAll()
        println(count)
    }
}

fun main() {
    CoroutineLearning().apply {
        testUnsafeConcurrent()
    }
}
```

可以发现最后的结果总是小于1000，这是因为在高并发的条件下，如果不做控制，有可能会在上一个协程还没有将结果写入到内存中的时候，当前协程就已经读取了数据并开始自增。这里谈论上一个和下一个协程是不严谨的，所有的协程都是平等的竞争者，不存在先后高低，只有调度的先后。

对于上面的情况，可以使用原子性的整型类（`AtomicInteger`）进行自增操作。

```
package com.example.coroutinelearning

import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.joinAll
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking
import java.util.concurrent.atomic.AtomicInteger

class CoroutineLearning {
    fun testSafeConcurrent() = runBlocking {
        val count = AtomicInteger(0)
        List(1000) {
            GlobalScope.launch { count.incrementAndGet() }
        }.joinAll()
```

```
        println(count.get())
    }
}

fun main() {
    CoroutineLearning().apply {
        testSafeConcurrent()
    }
}
```

上面的代码就可以安全地得到正确的计算结果。

## 协程的并发工具

除了我们在线程中常用的解决并发问题的手段之外， 协程框架也提供了一些并发安全的工具，包括：

- Channel：并发安全的消息通道， 我们已经非常熟悉。
- Mutex: 轻量级锁， 它的lock和unlock从语义上与线程锁比较类似， 之所以轻量是因为它在获取不到锁时不会阻塞线程， 而是挂起等待锁的释放。
- Semaphore: 轻量级信号量， 信号量可以有多个， 协程在获取到信号量后即可执行并发操作。当 Semaphore的参数为1 时， 效果等价于Mutex。

锁和信号量的使用可以具体参考操作系统相关的知识。

### Mutex

Mutex就是互斥锁，一旦获取不到，就需要挂起等待。使用完毕需要及时释放锁。

```
package com.example.coroutinelearning

import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.joinAll
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking
import kotlinx.coroutines.sync.Mutex
import kotlinx.coroutines.sync.withLock

class CoroutineLearning {
    fun testSafeConcurrentTools() = runBlocking {
```

```
        var count = 0
        val mutex = Mutex()
        List(1000) {
            GlobalScope.launch {
                mutex.withLock {
                    count++
                }
            }
        }.joinAll()
        println(count)
    }
}

fun main() {
    CoroutineLearning().apply {
        testSafeConcurrentTools()
    }
}
```

## Semaphore

```
package com.example.coroutinelearning

import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.joinAll
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking
import kotlinx.coroutines.sync.Mutex
import kotlinx.coroutines.sync.Semaphore
import kotlinx.coroutines.sync.withLock
import kotlinx.coroutines.sync.withPermit

class CoroutineLearning {
    fun testSafeConcurrentTools() = runBlocking {
        var count = 0
        val semaphore = Semaphore(1)
        List(1000) {
            GlobalScope.launch {
                semaphore.withPermit {
```

```
                    count ++
                }
            }
        }.joinAll()
        println(count)
    }
}

fun main() {
    CoroutineLearning().apply {
        testSafeConcurrentTools()
    }
}
```

## 避免访问外部可变状态

编写函数时要求它不得访问外部状态， 只能基千参数做运算， 通过返回值提供运算结果。

```
package com.example.coroutinelearning

import kotlinx.coroutines.async
import kotlinx.coroutines.runBlocking

class CoroutineLearning {
    fun testAvoidingAccessingOuterVariable() = runBlocking {
        var count = 0
        count += List(1000) {
            async { 1 }
        }.map { it.await() }.sum()
        println(count)
    }
}

fun main() {
    CoroutineLearning().apply {
        testAvoidingAccessingOuterVariable()
    }
}
```

上面的代码将count变量放到了所有协程的外部，也不在协程内部访问，自然也就不会有并发访问的问题。

# 协程Flow的综合使用

## Flow和文件下载

在文件下载的过程中，最核心的就是文件下载的下载器（DownloadManager），所以首先来定义一个DownloadManager：

```
import com.example.coroutinelearning.utils.copyTo
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.catch
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.flow.flowOn
import okhttp3.OkHttpClient
import okhttp3.Request
import java.io.File
import java.io.IOException

object DownloadManager {
    fun download(url:String, file:File) : Flow<DownloadStatus>{
        return flow<DownloadStatus> {
            val request = Request.Builder().url(url).get().build()
            val response =
OkHttpClient.Builder().build().newCall(request).execute()
            if (response.isSuccessful) {
                val body = response.body()!!
                val total = body.contentLength()
                file.outputStream().use { fileOutputStream ->
                    val input = body.byteStream()
                    var emittedProgress = 0L
                    input.copyTo(fileOutputStream) { bytesCopied -
>
                        val progress = bytesCopied * 100 / total
                        if (progress - emittedProgress > 5) {
```

```
                emit(DownloadStatus.Progress(progress.toInt()))
                                emittedProgress = progress
                            }
                        }
                    }
                    emit(DownloadStatus.Done(file))
                } else {
                    throw IOException(response.toString())
                }
            }.catch {
                file.delete()
                emit(DownloadStatus.Error(it))
            }.flowOn(Dispatchers.IO)
        }
    }
```

在代码中定义一个下载状态的类（DownloadStatus），这个类定义了所有的下载状态，包括下载中，下载错误，下载完成等。

```
import java.io.File

sealed class DownloadStatus {
    data class Progress(val value: Int) : DownloadStatus() // 下载中，下载进度
    data class Error(val throwable: Throwable) : DownloadStatus() // 下载错误
    data class Done(val file: File) : DownloadStatus() // 下载完成
    object None : DownloadStatus() // 其他下载状态
}
```

下面的扩展函数用来从数据流中复制字节数据。

```
import java.io.InputStream
import java.io.OutputStream

inline fun InputStream.copyTo(out: OutputStream, bufferSize: Int = DEFAULT_BUFFER_SIZE, progress: (Long)-> Unit): Long {
    var bytesCopied: Long = 0
```

```
    val buffer = ByteArray(bufferSize)
    var bytes = read(buffer)
    while (bytes >= 0) {
        out.write(buffer, 0, bytes)
        bytesCopied += bytes
        bytes = read(buffer)

        progress(bytesCopied)
    }
    return bytesCopied
}
```

绘制简单的加载界面，上面有一个按钮，点击之后开始下载文件，界面中间有一个进度条，进度条下方还有一个文本，都是用来显示下载的进度的。

```
<!-- activity_main.xml -->
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/start_download"
        android:text="Start Downloading"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />

    <ProgressBar
        android:id="@+id/progress_bar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        style="?android:attr/progressBarStyleHorizontal"
```

```
        tools:progress="50"
        app:layout_constraintTop_toBottomOf="@id/start_download"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"/>
    <TextView
        android:id="@+id/progress_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toBottomOf="@id/progress_bar"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        android:textColor="@color/black"
        android:textSize="40sp"
        android:text="0%"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

最后是Activity的代码：这里我们下载位于`http://guolin.tech/book.png`的一张图片。

```
import android.os.Bundle
import android.util.Log
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.lifecycle.lifecycleScope
import com.example.coroutinelearning.download.DownloadManager
import com.example.coroutinelearning.download.DownloadStatus
import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.coroutines.flow.collect
import java.io.File


class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        start_download.setOnClickListener {
```

```
19              val path = getExternalFilesDir(null)?.path
20              if (path != null) {
21                  val file = File(path, "pic.jpg")
22                  lifecycleScope.launchWhenCreated {
23
 DownloadManager.download("http://guolin.tech/book.png", file)
24                          .collect { status: DownloadStatus ->
25                              when (status) {
26                                  is DownloadStatus.Progress -> {
27                                      progress_bar.progress =
status.value
28                                      progress_text.text =
"${status.value}%"
29                                  }
30                                  is DownloadStatus.Error -> {
31
 Toast.makeText(this@MainActivity, "下载错误", Toast.LENGTH_SHORT)
32                                          .show()
33                                      Log.e("ning",
status.throwable.toString())
34                                  }
35                                  is DownloadStatus.Done -> {
36                                      progress_bar.progress = 100
37                                      progress_text.text = "100%"
38
 Toast.makeText(this@MainActivity, "下载完成",
Toast.LENGTH_SHORT).show()
39                                  }
40                                  else -> {
41                                      Log.d("ning", "下载失败.")
42                                  }
43                              }
44                          }
45                  }
46              }
47          }
48      }
49  }
```

需要注意的是，上面的代码需要添加网络权限：

```
<uses-permission android:name="android.permission.INTERNET" />
```

# Flow和Room的综合利用

# Flow和Retrofit的综合利用

# 冷流和热流

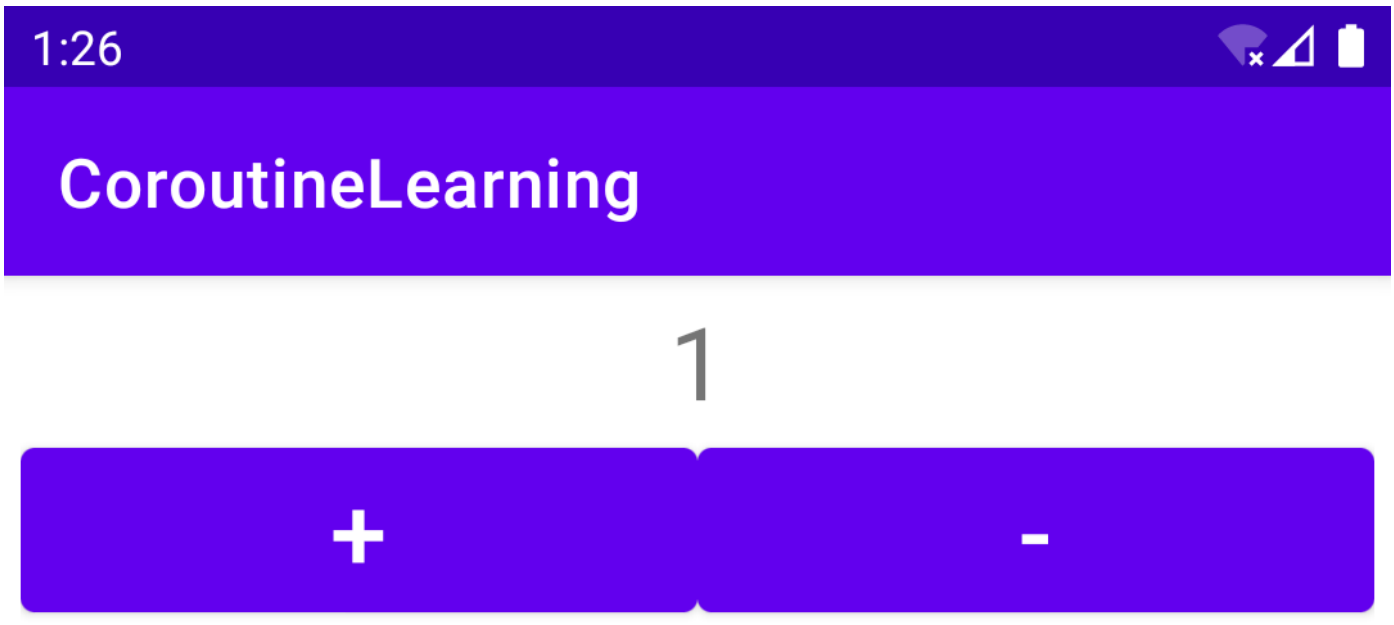
Flow 是冷流， 什么是冷流？简单来说， 如果 Flow 有了订阅者 Collector 以后， 发射出来的值才会实实在在的存在于内存之中， 这跟懒加载的概念很像。

与之相对的是热流， StateFlow 和 Shared Flow 是热流， 在垃圾回收之前， 都是存在内存之中， 并且处于活跃状态的。

## StateFlow

State Flow是一个状态容器式可观察数据流，可以向其收集器发出当前状态更新和新状态更新。 还可通过其value属性读取当前状态值。

例如有以下的示例：



点击加号可以让上面的数字加1，点击减号可以让上面的数字减1。这样的功能可以通过LiveData的方式实现，也可以通过StateFlow 的方式实现。

首先来构建基本的UI：

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="5dip">

        <TextView
            android:id="@+id/tv_number"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="center"
            android:text="0"
            android:textSize="30sp" />

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="horizontal">

            <Button
                android:id="@+id/btn_plus"
                android:layout_width="0dip"
                android:layout_height="wrap_content"
                android:layout_weight="1"
                android:text="+"
                android:textSize="30sp" />

            <Button
                android:id="@+id/btn_minus"
                android:layout_width="0dip"
                android:layout_height="wrap_content"
```

```
                android:layout_weight="1"
                android:text="-"
                android:textSize="30sp" />

        </LinearLayout>

    </LinearLayout>

</LinearLayout>
```

然后构建相对应的ViewModel：

```
class NumberViewModel : ViewModel() {
    // 创建一个MutableStateFlow的实例，初始值是0
    val number: MutableStateFlow<Int> = MutableStateFlow(0)

    // 对number这个MutableStateFlow里面的数据进行加1的操作。当数据发生变化的时候，会通过Flow的形式发送出去。
    fun increase() {
        number.value += 1
    }

    // 对number这个MutableStateFlow里面的数据进行减1的操作。当数据发生变化的时候，会通过Flow的形式发送出去。
    fun decrease() {
        number.value -= 1
    }
}
```

可以看到，ViewModel中包含一个名为number的MutableStateFlow的实例，并提供了两个方法对这个实例内的数据进行加减操作。

再然后就是最后的Activity的实现，如下：

```
class MainActivity : AppCompatActivity() {

    private val viewModel by viewModels<NumberViewModel>()


```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        btn_plus.setOnClickListener {
            viewModel.increase()
        }

        btn_minus.setOnClickListener {
            viewModel.decrease()
        }

        // 创建一个协程环境，对number这个MutableStateFlow进行数据的收集，并更新UI。
        lifecycleScope.launchWhenCreated {
            viewModel.number.collect { value ->
                tv_number.text = "$value"
            }
        }
    }
}
```

MainActivity中，对两个按钮的点击事件进行了处理，一个调用increase方法，另一个调用decrease方法。

然后创建一个协程，在协程内部对number这个StateFlow进行流数据的收集，然后更新UI。

运行上面的代码，就可以借用StateFlow的方式实现数据的加加减减的操作。

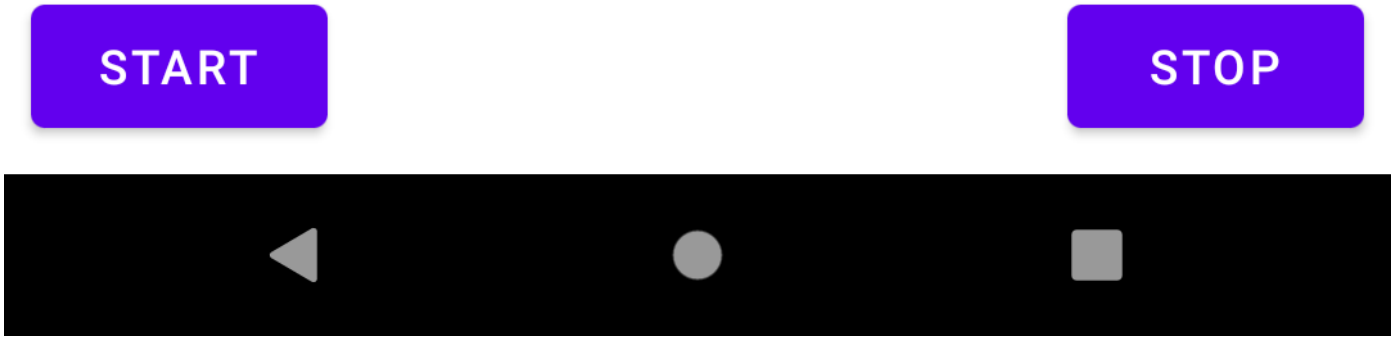
## SharedFlow

Shared Flow 会向从其中收集值的所有使用方发出数据。

SharedFlow的作用类似于广播。

例如下面的示例，点击开始之后，向SharedFlow中发送数据，数据会被三个独立的Fragment分别接受，三个Fragment接收的数据都是相同的。


CoroutineLearning
1638600747143
1638600747143
1638600747143




首先来编写需要的SharedFlow的代码，如下：

```
/**
 * 通过单例类的形式来创建一个SharedFlow的实例，这样可以被全局访问。
 */
object LocalEventBus {
    /**
     * 创建一个MutableSharedFlow实例，泛型是Event类型。
     */
    val events = MutableSharedFlow<Event>()

    /**
     * 通过emit方法发送数据到SharedFlow数据流中。
     */
    suspend fun postEvent(event: Event) {
        events.emit(event)
    }
}

/**
 * 定义所发送的事件类型，内部的数据是一个Long类型的时间戳。
 */
data class Event(val timestamp: Long)
```

再实现内部小的Fragment，UI如下，只是一个简单的TextView：

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    tools:context=".TextFragment">

    <TextView
        android:id="@+id/tv_time"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:text="--"
        android:textSize="36sp" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

再来实现TextFragment，如下：

```
class TextFragment : Fragment(){
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_text, container, true)
    }

    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        /**
         * 创建一个协程，在协程内部对LocalEventBus内的SharedFlow进行数据收集，然后进行UI
         */
        lifecycleScope.launchWhenCreated {
            LocalEventBus.events.collect {
                tv_time.text = it.timestamp.toString()
            }
        }
    }
}
```

接着实现MainActivity和对应的ViewModel。

首先来实现MainActivity的ViewModel，如下：

```
class NumberViewModel : ViewModel() {
    private lateinit var job: Job

    /**
     * 开始刷新数据，不断地将数据发送到LocalEventBus中的SharedFlow中。
     * 需要注意的是，需要在一个协程中进行发送。
     * 将创建的协程保存在Job中。
     */
    fun startRefresh() {
        job = viewModelScope.launch(Dispatchers.IO) {
            while (true) {

 LocalEventBus.postEvent(Event((System.currentTimeMillis())))
            }
        }
    }

    /**
     * 将协程取消，停止刷新，也就是停止向SharedFlow中发送数据。
     */
    fun stopRefresh() {
        job.cancel()
    }
}
```

MainActivity的UI定义如下：

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">
```

```
8
9       <LinearLayout
10          android:layout_width="match_parent"
11          android:layout_height="match_parent"
12          android:orientation="vertical">
13
14          <fragment
15              android:id="@+id/one"
16
    android:name="com.example.coroutinelearning.TextFragment"
17              android:layout_width="match_parent"
18              android:layout_height="0dp"
19              android:layout_weight="1" />
20
21          <View
22              android:layout_width="match_parent"
23              android:layout_height="5dp"
24              android:background="@color/black" />
25
26          <fragment
27              android:id="@+id/two"
28
    android:name="com.example.coroutinelearning.TextFragment"
29              android:layout_width="match_parent"
30              android:layout_height="0dp"
31              android:layout_weight="1" />
32
33          <View
34              android:layout_width="match_parent"
35              android:layout_height="5dp"
36              android:background="@color/black" />
37
38          <fragment
39              android:id="@+id/three"
40
    android:name="com.example.coroutinelearning.TextFragment"
41              android:layout_width="match_parent"
42              android:layout_height="0dp"
43              android:layout_weight="1" />
```

```
    </LinearLayout>

    <Button
        android:id="@+id/btn_start"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|start"
        android:layout_marginLeft="8dp"
        android:layout_marginBottom="8dp"
        android:text="start" />

    <Button
        android:id="@+id/btn_stop"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_marginRight="8dp"
        android:layout_marginBottom="8dp"
        android:text="stop" />

</FrameLayout>
```

MainActivity的具体实现如下：

```
class MainActivity : AppCompatActivity() {

    private val viewModel by viewModels<NumberViewModel>()


    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        btn_start.setOnClickListener {
            viewModel.startRefresh()
        }

        btn_stop.setOnClickListener {
```

```
15                viewModel.stopRefresh()
16            }
17        }
18    }
```

将上面的代码运行起来，就可以实现SharedFlow向所有的接收值发送数据的简单实现。

# Flow和Jetpack Paging3

## Paging3

- 加载数据的流程