

demo-kt为android端协程基础使用

kt-coroutine为本课件示例代码

Coroutine

在讲协程之前我们先不讲Kotlin的协程，因为Kotlin的协程是属于操作系统协程的一种实现，要知根知底就得从操作系统的进程 线程 再到协程开始聊起

进程

基本概念

进程（英语：*process*），是指计算机中已执行的程序。进程曾经是分时系统的基本运作单位。在面向进程设计的系统（如早期的UNIX，Linux 2.4及更早的版本）中，进程是程序的基本执行实体；在面向线程设计的系统（如当代多数操作系统、Linux 2.6及更新的版本）中，进程本身不是基本执行单位，而是线程的容器。

——摘自维基百科

发展

最开始时候的操作系统是没有进程这个概念的，当有多个程序需要执行的时候。是采用“排队”的方式进行的。比如有两个程序A，B。由于那时候没有进程的概念，一次只能执行一个程序，每个程序独占整个计算机的资源。执行完成A以后就人工地或者通过一个调度程序自动去切换到B程序。

这种执行方式在现在看来是非常落后地，因为一个计算机一次只能执行一个程序。

但是这种执行方式在那个时期其实也没有造成过大地困扰，因为计算机的算力还不是很，除了切换程序的执行稍微麻烦点，要耗费一些额外的时间以外，一个程序独占整个计算机资源也不算太离谱。

后来就有了进程的概念(这里省略了好几个操作系统的发展阶段)，一个进程不严谨的说可以看出是一个应用程序(其实确实不太严谨)，一个应用可以有一个进程，也可以有两个进程或者多个。

进程是线程的容器，他表示的是操作系统为程序分配资源的基本单位，操作系统为每个需要执行的程序分配资源的时候就是直接分配的进程。

线程

基本概念

线程（英语：*thread*）是操作系统能够进行运算调度的最小单位。大部分情况下，它被包含在进程之中，是进程中的实际运作单位。

——摘自维基百科

小结

简单来说进程就是操作系统分配资源的单位（或者说是线程的容器），然后线程就属于是进行执行调度的最小单位。

当我们打开一个软件后，操作系统会为我们的软件分配进程，然后再为进程内分配一些资源，比如线程或者内存啥的。

协程

什么是协程

协程（英语：*coroutine*）是计算机程序的一类组件，推广了协作式多任务的子例程，允许执行被挂起与被恢复。相对子例程而言，协程更为一般和灵活，但在实践中使用没有子例程那样广泛。协程更适合于用来实现彼此熟悉的程序组件，如协作式多任务、异常处理、事件循环、迭代器、无限列表和管道。

根据高德纳的说法，马尔文·康威于1958年发明了术语“*coroutine*”并用于构建汇编程序[1]，关于协程最初的出版解说在1963年发表[2]。

——摘自维基百科

- 协作式多任务

协作式多任务（*Cooperative Multitasking*），是一种多任务方式，多任务是使电脑能同时处理多个程序的技术，相对于抢占式多任务（*Preemptive multitasking*），协作式多任务要求每一个运行中的程序，定时放弃自己的执行权利，告知操作系统可让下一个程序执行[1][2]。

一颗处理器同一时间只能处理一个程序，要同时处理多个程序，必须将处理器于相对于用户来说相当短的时间，划分给不同的程序运行，以使各个不同程序都能执行部分工作，使用户错觉以为各个程序都同时被执行，例如浏览器能处理“下卷”的动作，同时MP3播放器将声音解码。如果某程序因设计不良或出现故障而不释放执行权，整个操作系统便告停顿。

——摘自维基百科

- 子例程

在计算机科学中，子程序（德语：unterprogramm，英语：subroutine, subprogram, callable unit），是一个大型程序中的某部分代码，由一个或多个语句块组成。它负责完成某项特定任务，而且相较于其他代码，具备相对的独立性。

一般会有输入参数并有返回值，提供对过程的封装和细节的隐藏。这些代码通常被集成成为软件库。

函数在面向过程的语言中已经出现。是结构（struct）和类（class）的前身。本身就是对具有相关性语句的归类和对某过程的抽象。

——摘自维基百科

协程&线程

协程非常类似于线程。但是协程是协作式多任务的，而线程典型是抢占式多任务的。这意味着协程提供并发性而非并行性。协程超过线程的好处是它们可以用于硬性实时的语境（在协程之间的切换不需要涉及任何系统调用或任何阻塞调用），这里不需要用来守卫关键区块的同步性原语（primitive）比如互斥锁、信号量等，并且不需要来自操作系统的支持。有可能以一种对调用代码透明的方式，使用抢占式调度的线程实现协程，但是会失去某些利益（特别是对硬性实时操作的适合性和相对廉价的相互之间切换）。

线程是协作式多任务的轻量级线程，本质上描述了同协程一样的概念。其区别，如果一定要说有的话，是协程是语言层级的构造，可看作一种形式的控制流，而线程是系统层级的构造，可看作恰巧没有并行运行的线程。这两个概念谁有优先权是争议性的：线程可看作为协程的一种实现[6]，也可看作实现协程的基底[7]。

——摘自维基百科

- 协程类似于线程
- 协程的调度是协作式的，线程是抢占式的
- 协程切换不需要涉及任何系统调用和阻塞调用
- 线程可看作为协程的一种实现[6]，也可看作实现协程的基底[7]

为什么有协程

协程的出现是因为线程在高并发存在很多痛点。

1.线程是需要向操作系统交互的

整个意味着不自由，重量级。

2，线程的管理调度是困难的

多线程高并发是一个难点，虽然java底层就支持多线程，在顶层父类Object内部就提供了多线程并发的相关调度方法什么notify，notifyAll，wait。但是不好用啊。对于客户端来说多线程高并发并不多，但是对于服务端来说，用户量一上来就是非常恐怖的时期。轻则服务体验不佳，重则机子带不动crash。

■

然后协程定义上与线程类似，但不需要向操作系统申请资源，又有协作式多任务调度的特性。

所以总的来说协程在概念上来说是解决并发问题的。解决多线程调度的不灵活的问题。

关于协程的误区

协程是什么？

协程非常类似于线程。但是协程是协作式多任务的，而线程典型是抢占式多任务的。

线程可看作为协程的一种实现[6]，也可看作实现协程的基底[7]。

这几句话不是我说的，是维基百科的。

所以定义上来说协程的定义是非常的模糊的。

我们所知道的也只有它的部分特性

- 用户态——不与OS打交道，创建不需要向OS申请资源
- 协作式——不同于线程的抢占式执行调度，协程的执行调度协作式的。
- 挂起恢复——协程是协作式的，协作式是由挂起恢复提供的支持。线程之所有不具备协作式的特点是因为对于线程的状态转变需要操作系统调度，不灵活。

有了协程就不要线程了嘛？

未必

线程主要是用于解决并行问题的，协程是用于解决并发问题的。

毫不客气地说他们都不在一个频道。

而且维基百科给的一套定义也没有解释地非常清楚。

我连协程是什么都不知道影响我学习Kotlin协程嘛？

不影响

进程和线程是操作系统地概念，协程是编程语言层级地概念。

协程是一种抽象的概念，抽象的意思是表示不具体，不可描述的，模糊的东西。

不同的编程语言都有一些属于自己的实现。所以当刚接触协程的时候，模糊和不解才是正常的，如果说看了就啥都懂了，那很不正常.....（这种人建议给他抓起来

协程起源于一种[汇编语言](#)方法，但有一些[高级编程语言](#)支持它。早期的例子包括[Simula](#)^[12]、[Smalltalk](#)和[Modula-2](#)。更新近的例子是[Ruby](#)、[Lua](#)、[Julia](#)和[Go](#)。

- | | | | |
|---|---|--|---|
| <ul style="list-style-type: none">• Aikido• AngelScript• Ballerina• BCPL• Pascal (Borland Turbo Pascal 7.0带有uThreads模块)• BETA• BLISS• C++ (自从C++20)• C# (自从2.0)• Chuck• CLU• D• Dynamic C | <ul style="list-style-type: none">• Erlang• F#• Factor• GameMonkey Script• GDScript (Godot的脚本语言)• Go• Haskell^[13]^[14]• 高级汇编语言^[15]• Icon• Io• JavaScript (自从1.7, 标准化于ECMAScript 6^[16], ECMAScript 2017还包括async/await支持)• Julia^[17] | <ul style="list-style-type: none">• Kotlin (自从1.1) ^[18]• Limbo• Lua^[19]• Lucid• μC++• Modula-2• Nemerle• Perl 5 (使用Coro模块^[20])• PHP (带有hiphop-php^[21], 原生支持自从PHP 5.5)• Picolisp• Prolog• Python (自从2.5^[22], 带有改进支持自从3.3, 带有显式语法自从3.5^[23]) | <ul style="list-style-type: none">• Raku^[24]• Ruby• Sather• Scheme• Self• Simula 67• Smalltalk• Squirrel• Stackless Python• SuperCollider^[25]• Tcl (自从8.6)• urbiscript |
|---|---|--|---|

所以总结一句话就是：协程的是一种对于编程语言的抽象的概念，而特定编程语言的协程是对于这个概念的具体实现。

采用面向对象的说法就是：协程类似于一个抽象类，*Kotlin*协程就是这个抽象类的具体实现。

所以不是很了解协程，去学*Kotlin*协程有影响嘛？没有！！！

为什么要学Kotlin Coroutine

*github*上有很多的开源库，其中有一个非常出名，非常实用的框架叫做*Rxjava*，它的出现使得一些多线程并发问题变得简单。但是使用它其实门槛是比较高的，所以对于新人来说，使用它难度是比较大的，而*kotlin*的协程使用门槛低，而且非常方便简洁。所以渐渐的就有了取代*Rxjava*地位的趋势。更何况这是*Kotlin*语言的特性，所以就非常有学习的必要

发展

*Kotlin*协程是用于解决并发问题的，在*Kotlin*出现之前，我们一般是采用的什么进行一个并发操作呢？

- 回调
- RxJava
- Coroutine

最开始进行异步的操作是采用的回调，但是由于回调...嗯“太好用了”，一层层的回调嵌套，狗看了都直摇头。不美观，不简洁，而且还难以读懂。

然后就有了*rxjava*，*rxjava*很大程度上消除了回调的嵌套，但是呢，操作符多（复杂），门槛高的缺点还是引得一些人吐槽。

紧接着有了*kt-coroutine*，它弥补了*rxjava*复杂的缺点，容易上手，灵活度和自由度高，现在成了替换*rxjava*的方案。

给出一个需求定义3个耗时任务A，B，C，执行顺序A->B->C。

下面采用3种方式进行对比.

回调

定义回调接口

```
fun interface TaskACallback {  
    fun executeA()  
}  
  
fun interface TaskBCallback {  
    fun executeB()  
}  
  
fun interface TaskCCallback {  
    fun executeC()  
}
```

模拟耗时任务

```
fun doTaskA(taskA: TaskACallback) {  
    thread {  
        println("taskA:假装我在做耗时任务")  
        Thread.sleep(1000)  
        taskA.executeA()  
    }  
}  
  
fun doTaskB(taskB: TaskBCallback) {  
    thread {  
        println("taskB:假装我在做耗时任务")  
        Thread.sleep(1000)  
        taskB.executeB()  
    }  
}  
  
fun doTaskC(taskC: TaskCCallback) {  
    thread {  
        println("taskC:假装我在做耗时任务")  
        Thread.sleep(1000)  
        taskC.executeC()  
    }  
}
```

测试

```
fun main() {  
    doTaskA {  
        println("taskA总算是执行完了")  
        //写入一些逻辑  
    }  
    doTaskB {
```

```

        println("taskB总算是执行完了")
        //写入一些逻辑
        doTaskC{
            println("taskC总算是执行完了")
            //写入一些逻辑
        }
    }
}
}
}

```

可以明显的看出，随着业务逻辑的复杂化，嵌套层数会越来越多，如果来10个耗时任务，就得要10个接口，进行10次回调，那...

Rxjava

模拟耗时任务

```

fun taskA() = Observable.create<Unit> { it.onNext(Unit) }
    .map {
        println("taskA:假装我在做耗时任务")
        Thread.sleep(1000)
    }

fun taskB() = Observable.create<Unit> { it.onNext(Unit) }
    .map {
        println("taskB:假装我在做耗时任务")
        Thread.sleep(1000)
    }

fun taskC() = Observable.create<Unit> { it.onNext(Unit) }
    .map {
        println("taskC:假装我在做耗时任务")
        Thread.sleep(1000)
    }

```

测试

```

fun main() {
    taskA()
        .flatMap {
            println("taskA完成")
            taskB()
        }
        .flatMap {
            println("taskB完成")
            taskC()
        }
        .subscribeOn(Schedulers.io())
        .subscribe()
}

```

```
Thread.sleep(10000)
}
```

可以很明显地发现rxjava去除了回调，提高了可读性，但是上手难度较大，有不少地新东西。各种各样地操作符（这里用的比较少，真实使用中会使用到很多），对于我这种菜鸡来说使用起来也是颇为吃力。复杂性没有降低，可读性有了质的提升。

kt-coroutine

编写耗时任务

```
suspend fun taskA() {
    println("taskA:假装我在做耗时任务")
    delay(1000)
}

suspend fun taskB() {
    println("taskB:假装我在做耗时任务")
    delay(1000)
}

suspend fun taskC() {
    println("taskC:假装我在做耗时任务")
    delay(1000)
}
```

测试

```
fun main() {
    runBlocking {
        taskA()
        println("TaskA finished")
        taskB()
        println("TaskB finished")
        taskC()
    }
}
```

提升很明显，逻辑清晰，代码简洁，可读性高。

对比

回调 < RxJava < Coroutine

Kotlin Coroutine是什么？

就目前来看，协程被分成了两个层次。

1.基础层

是抽象概念上协程的实现。在`kotlin`中这一层次的内容为`kotlin`语言提供了协程的支持，主要也就是实现了挂起和恢复。

2.框架层

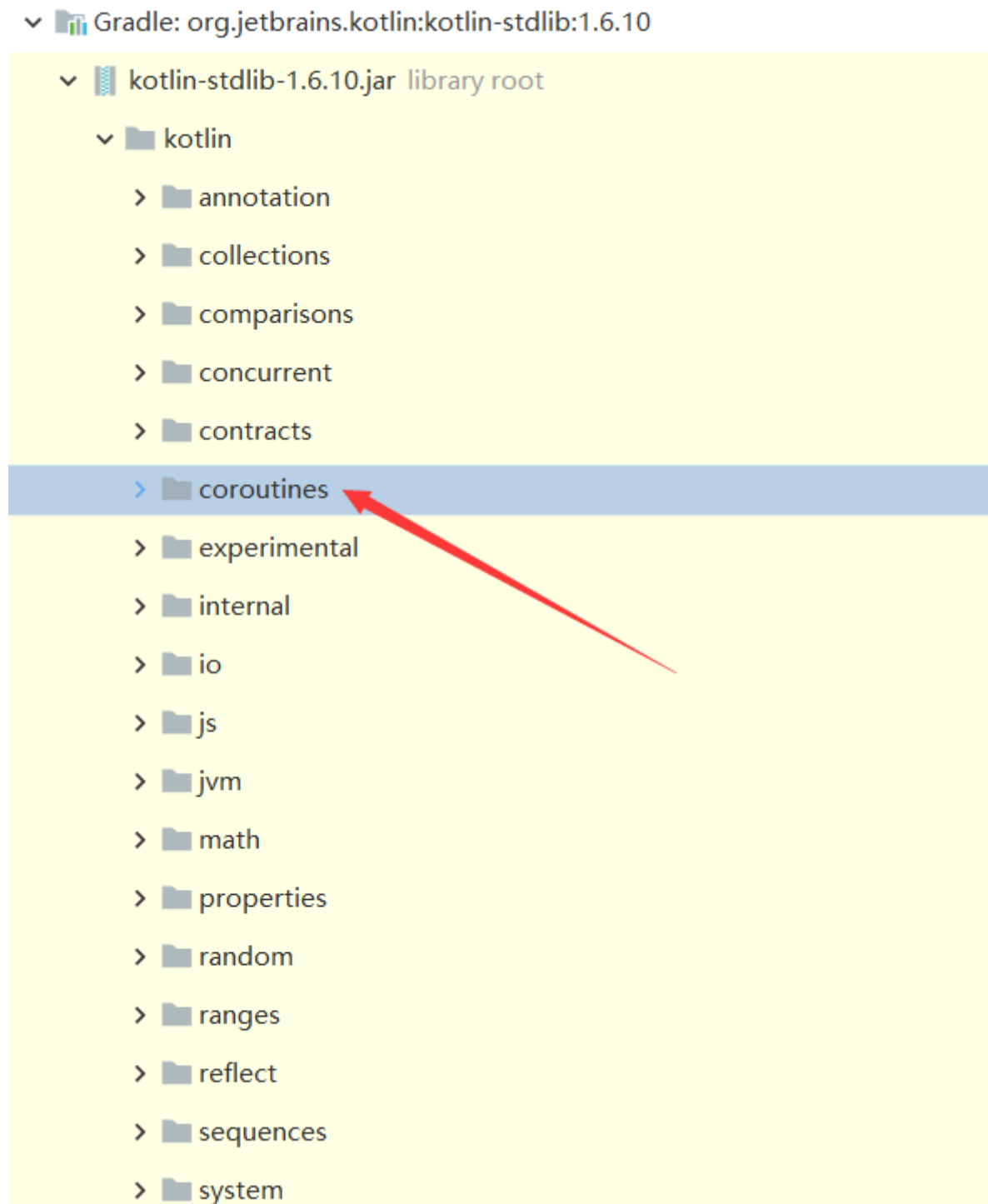
为基于协程这一语言特性，开发了一些实用的框架。比如`flow`，`channel`，`stateFlow`等一系列。

基础层

基础层中基础二字很有灵性，它表示这是`kotlin coroutine`中比较底层，重要的东西（可以说是最重要的东西。）

基础层的内容为`kotlin`语言提供了协程的最基本支持，也就是挂起和恢复。

它的源码集中在`kotlin`语言的标准库里面。



如果想要它的源码可以去maven里面找找这两个jar包的source file

kotlin-stdlib-common

[Maven Repository: org.jetbrains.kotlin » kotlin-stdlib-common » 1.6.10](https://mvnrepository.com/artifact/org.jetbrains.kotlin/kotlin-stdlib-common/1.6.10)

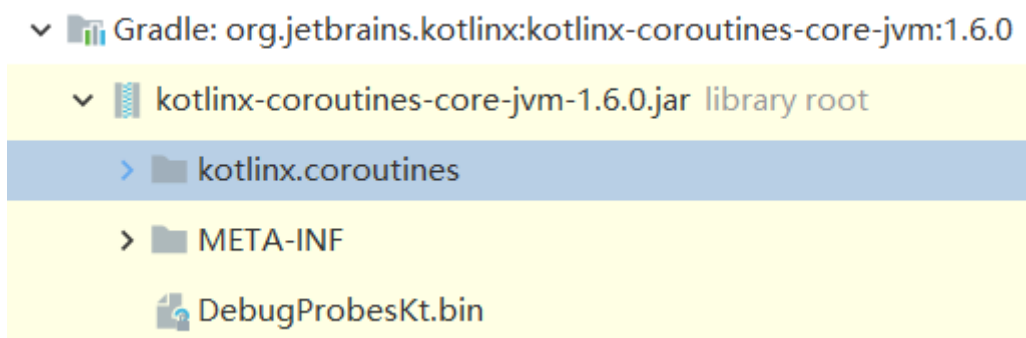
kotlin-stdlib-jvm

[https://mvnrepository.com/artifact/org.jetbrains.kotlin/kotlin-stdlib-jvm](https://mvnrepository.com/artifact/org.jetbrains.kotlin/kotlin-stdlib-jvm/1.6.10)

框架层

框架层是基于基础层提供的语言支持，提供了另一套实用的框架。主要有flow, channel, stateFlow, runBlocking, GlobalScope, Dispatchers, delay 等等.....

这部分内容被划分到了kotlinx-coroutines-core里面了。



小结

- kotlin coroutine分为两个层次。
- 基础层以及框架层，基础层提供了语言支持，粗浅可任务是对suspend关键字的支持
- 框架层提供了实用框架的支持，类似于我们一些常用的框架，比如rxjava, retrofit等等，只不过这个框架是基于协程基础层的，是jetbrains官方写的。

基础层

suspend function & suspend lambda 初探

*suspend*是一个关键字，*suspend {}*只是一个内联函数

suspend function

当一个函数被suspend修饰的时候，那么它就是一个挂起函数。挂起函数和普通函数相比是类似的，只是多了两个特异的功能：挂起，恢复。

挂起函数的声明如下。

```
suspend fun suspendFuc() {  
    println("这是一个挂起函数")  
    delay(1000)  
    println("挂起函数被挂起了")  
}
```

在增添功能的同时也多了约束。因为要兼容jvm平台的其他函数。

- 挂起函数只能在其他的挂起函数或者协程作用域中调用

协程作用域中调用

```
fun main() {
    GlobalScope.launch {
        suspendFuc()
    }
}
```

挂起函数中调用

```
suspend fun main() {
    suspendFuc()
}
```

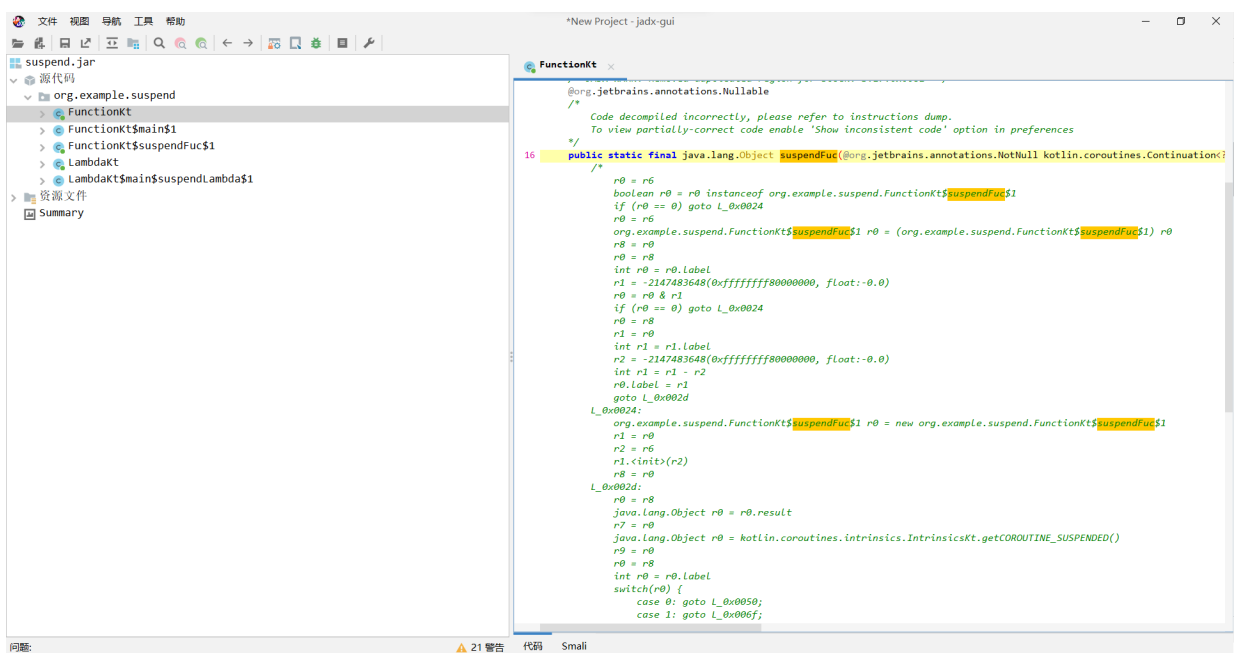
原理分析

kotlin是支持jvm平台的，jvm平台只认字节码，所以无论java也好，kotlin也好，你产物都只能是字节码才能在jvm平台运作，kotlin引入了新东西不假，但是看不太懂实现，我们就可以将其产物反编译成java(因为java中规中矩，（绕脑子的新特性比较少）

怎么把kotlin代码转化为java代码这是个技术活。

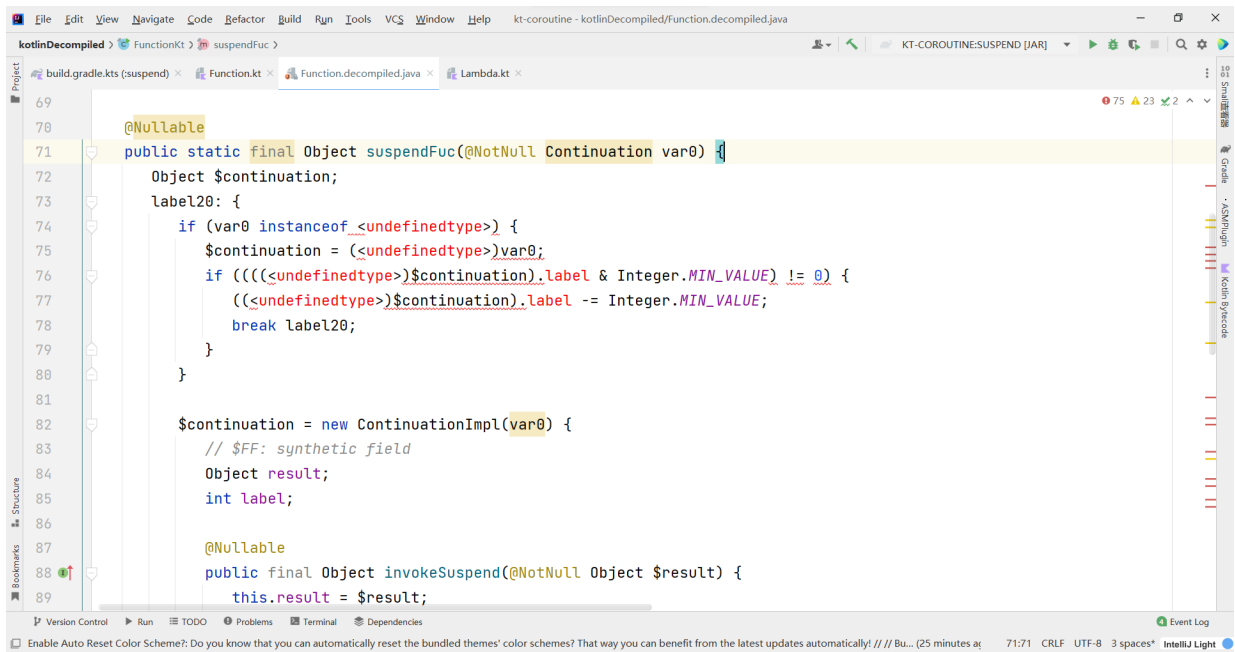
有两种可选方案，一是用idea/as自带的show kotlin bytecode，二是将它打包为jar然后通过反编译工具查看。

- 反编译jar包



很遗憾反编译失败了，函数体部分的内容被做了手脚，只有字节码。

- show kotlin bytecode



好在show kotlin bytecode生效了

可以发现suspendFunc从形参到返回值再到函数体都被修改了。（好坏哦

```
@Nullable
public static final Object suspendFuc(@NotNull Continuation var0) {
    Object $continuation;
    label20: {
        if (var0 instanceof <undefinedtype>) {
            $continuation = (<undefinedtype>)var0;
            if (((<undefinedtype>)$continuation).label & Integer.MIN_VALUE) != 0) {
                ((<undefinedtype>)$continuation).label -= Integer.MIN_VALUE;
                break label20;
            }
        }

        $continuation = new ContinuationImpl(var0) {
            // $FF: synthetic field
            Object result;
            int label;

            @Nullable
            public final Object invokeSuspend(@NotNull Object $result) {
                this.result = $result;
                this.label |= Integer.MIN_VALUE;
                return FunctionKt.suspendFuc(this);
            }
        };

        Object $result = ((<undefinedtype>)$continuation).result;
```

```

Object var4 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
String var1;
switch(((<undefinedtype>)$continuation).label) {
case 0:
    ResultKt.throwOnFailure($result);
    var1 = "这是一个挂起函数";
    System.out.println(var1);
    ((<undefinedtype>)$continuation).label = 1;
    if (DelayKt.delay(1000L, (Continuation)$continuation) == var4)
{
        return var4;
    }
    break;
case 1:
    ResultKt.throwOnFailure($result);
    break;
default:
    throw new IllegalStateException("call to 'resume' before
'invoke' with coroutine");
}

var1 = "挂起函数被挂起了";
System.out.println(var1);
return Unit.INSTANCE;
}

```

经过多次的测试我们会发现。

所有的挂起函数函数形参都会被插入一个Continuation，返回值都会被修改为Object。函数体都会魔改。

suspend a():Unit -> Object a(Continuation continuation)

suspend b(args1:Int):Unit -> Object b(int args1,Continuation continuation)

.....

依次类推。

函数体

分为3个部分

- label20那段代码进行continuation的初始化
- switch部分代码进行执行调度
- 最后收尾

```

@Nullable
public static final Object suspendFuc(int var0, @NotNull Continuation var1) {
    Object $continuation;
    label20: {
        Object $result = ((<undefinedtype>)$continuation).result;
        Object var5 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
        String var2;
        switch(((<undefinedtype>)$continuation).label) {
            ...

        var2 = "挂起函数被挂起了";
        System.out.println(var2);
        return Unit.INSTANCE;
    }
}

```

Continuation初始化

代码属于是伪代码，看起来有点吃力，但是也还好。

这里涉及一个Integer小知识点，Integer是4个字节也就是说，他是32位2进制。

但是Integer的最大，最小值如下。

最大值的二进制表示为

0111 1111 1111 1111 1111 1111 1111 1111(2) -> 7 f f f f f f f(16)

同理最小值的二进制表示为

1000 0000 0000 0000 0000 0000 0000 0000(2) -> 8 f f f f f f f(16)

关于它的与或操作想必大家都懂，就不啰嗦了。

A constant holding the minimum value an int can have, -2^{31} .

```
@Native public static final int MIN_VALUE = 0x80000000;
```

A constant holding the maximum value an int can have, $2^{31}-1$.

```
@Native public static final int MAX_VALUE = 0x7fffffff;
```

```

label20: {
    if (var0 instanceof <undefinedtype>) {
        $continuation = (<undefinedtype>)var0;
        if (((<undefinedtype>)$continuation).label &
Integer.MIN_VALUE) != 0) {
            ((<undefinedtype>)$continuation).label -=
Integer.MIN_VALUE;
            break label20;
        }
    }
}

```

```

    }
}

$continuation = new ContinuationImpl(var0) {
    // $FF: synthetic field
    Object result;
    int label;

    @Nullable
    public final Object invokeSuspend(@NotNull Object $result) {
        this.result = $result;
        this.label |= Integer.MIN_VALUE;
        return FunctionKt.suspendFuc(this);
    }
};
}

```

初始化的逻辑不难，也就是先判断一下label的值与Integer.MIN_VALUE相与是否为0
不为0就跳过初始化，为0就进行初始化。

这样的意义是为了保证挂起函数在被*invokeSuspend*的时候不会被重复创建*Continuation*。也就是确保一个挂起函数内只有一个*continuation*实例。

switch调度

在了解这个变换之前需要一点点的前置知识。也就是*Kotlin*协程的挂起和恢复。

挂起就是中断，也就是停下来了，恢复呢也就是在挂起处恢复执行。

这个有点像我们播放视频一样，挂起对应*pause*，恢复对应*continue*。

这个特性很强大，但是有一定限制，*kotlin*协程并不能在任何位置被挂起，他只有可能在调用挂起函数的时候被挂起，调用普通函数是不可以被挂起的。

*switch*调度实则是将我们原来定义的挂起函数体进行重新的组合。

为什么需要重新组合？且听我细细道来

首先挂起函数需要挂起和恢复。

也就是说需要在一个地方停下来，同时还要在这个地方恢复回来。

这个在普通函数内是看似无法实现的。

因为在一个地方停下来，又恢复。需要一个东西来保存我们的执行位置。

然而挂起函数却通过一系列特殊的操作实现了这个停止和恢复的逻辑。

它的挂起和恢复依靠两个东西

- Continuation
- switch代码块

Continuation译文叫做继续，连续。所以呢它充当的角色就是挂起函数执行状态的记录者。

而switch代码块有很多的分支，而每个分支就是一个可以被挂起的位置（简而言之每个挂起函数的调用都会被放到一个分支里（不一定是switch的分支））。（因为挂起函数不是每个地方都可以被挂起，挂起函数只有可能在调用挂起函数的时候被挂起）。

说再多也无意义，看代码才是真的。

```
switch(((<undefinedtype>)$continuation).label) {
    case 0:
        ResultKt.throwOnFailure($result);
        var1 = "这是一个挂起函数";
        System.out.println(var1);
        ((<undefinedtype>)$continuation).label = 1;
        if (DelayKt.delay(1000L, (Continuation)$continuation) == var4)
        {
            return var4;
        }
        break;
    case 1:
        ResultKt.throwOnFailure($result);
        break;
    default:
        throw new IllegalStateException("call to 'resume' before
        'invoke' with coroutine");
}
```

可以看到挂起的switch是从0开始依次递增。

我们的delay被放到了一个单独的分支。也就是case0.

执行过程就是是先会执行case0，然后打印字符串。然后更新label，然后再调用delay。delay的调用会判断是否被挂起，如果被挂起了就直接return，否者就是直接break。

由于示例代码只调用了一次挂起函数，不能很好的反映规律，我们改一下示例代码，多调用几次观察一下变化。

```

suspend fun suspendFuc() {
    println("这是一个挂起函数")
    delay(1000)
    println("挂起函数被挂起了")
    delay(1000)
    println("挂起函数被挂起了")
    delay(1000)
    println("挂起函数被挂起了")
    delay(1000)
    println("挂起函数被挂起了")
}

```

switch代码块变得复杂了.

但是依然有规律可寻

可以发现代码块呈现出一种类似于洋葱的结构，一层包一层（有没有回调的那味道？）

每一层代码块里面都会有一个挂起函数的调用，而执行哪个代码块是由switch调度的。

```

label40: {
    Object var4;
    label39: {
        label38: {
            Object $result = ((<undefinedtype>)$continuation).result;
            var4 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
            switch(((<undefinedtype>)$continuation).label) {
            case 0:
                ResultKt.throwOnFailure($result);
                var1 = "这是一个挂起函数";
                System.out.println(var1);
                ((<undefinedtype>)$continuation).label = 1;
                if (DelayKt.delay(1000L, (Continuation)$continuation)
== var4) {
                    return var4;
                }
                break;
            case 1:
                ResultKt.throwOnFailure($result);
                break;
            case 2:
                ResultKt.throwOnFailure($result);
                break label38;
            case 3:
                ResultKt.throwOnFailure($result);
                break label39;
            case 4:
                ResultKt.throwOnFailure($result);
                break label40;
            }
        }
    }
}

```

```

        default:
            throw new IllegalStateException("call to 'resume'
before 'invoke' with coroutine");
        }

        var1 = "挂起函数被挂起了";
        System.out.println(var1);
        (((<undefinedtype>) $continuation).label = 2;
        if (DelayKt.delay(1000L, (Continuation) $continuation) ==

var4) {
            return var4;
        }
    }

    var1 = "挂起函数被挂起了";
    System.out.println(var1);
    (((<undefinedtype>) $continuation).label = 3;
    if (DelayKt.delay(1000L, (Continuation) $continuation) ==

var4) {
        return var4;
    }
}

    var1 = "挂起函数被挂起了";
    System.out.println(var1);
    (((<undefinedtype>) $continuation).label = 4;
    if (DelayKt.delay(1000L, (Continuation) $continuation) == var4)
    {
        return var4;
    }
}

```

比如：

case0执行的是label38代码块中的switch部分。

case1执行的是label38代码块中switch下方的代码块。

case2执行的是label38到label39的代码块部分。

case3执行的是label39到label40的代码块部分。

case4执行的是label40下方的代码块部分。

而每一个分支的执行路径中只有一个挂起函数的调用，而且会更新continuation中的label值。

也就是说执行路径是

执行case0->挂起,恢复->执行case1->挂起, 恢复->执行case2....

其中挂起是依据一行代码：**return**

恢复也是通过一行代码：**invokeSuspend**

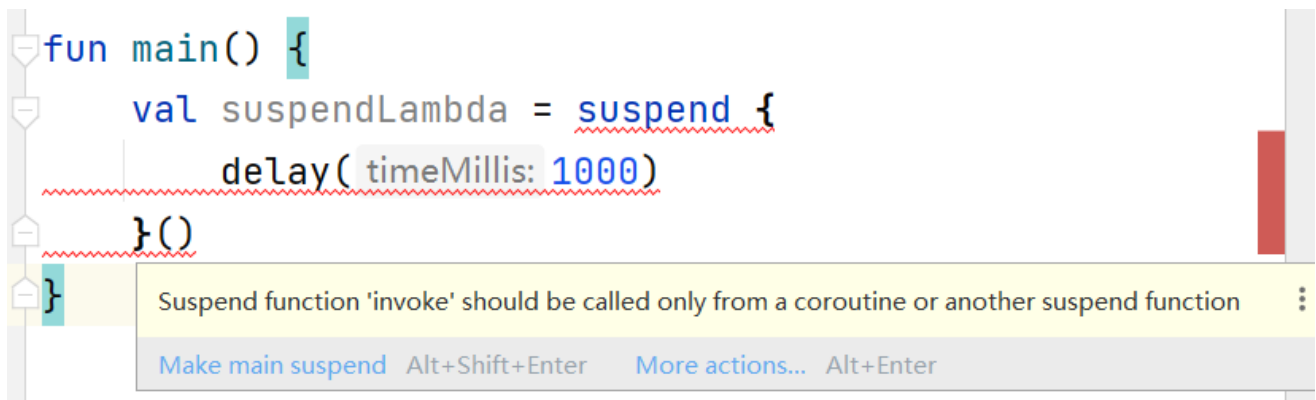
经过上述的分析我们对挂起函数有了最基本的认识。

suspend lambda

```
fun main() {  
    val suspendLambda = suspend {  
        delay(1000)  
    }  
}
```

这样我们就定义了一个suspend lambda。注意只是定义了，就好像把挂起函数的引用赋值给了一个变了一样，我们还没有调用它，调用suspend lambda需要通过invoke去调用。

不过呢suspend lambda也需要遵循挂起函数的限制。



原理分析

或许你在疑惑为什么要把suspend function和suspend lambda分开来分析。

因为他们的实现有点不太一样

- show kotlin bytecode

```
Function1 var1 = (Function1) (new Function1((Continuation) null) {  
    int label;  
  
    @Nullable  
    public final Object invokeSuspend(@NotNull Object $result)  
{  
        Object var2 = IntrinsicsKt.getCOROUTINE_SUSPENDED();  
        switch(this.label) {  
            case 0:  
                ResultKt.throwOnFailure($result);  
                this.label = 1;  
                if (DelayKt.delay(1000L, this) == var2) {  
                    return var2;  
                }  
            }  
        }  
    }  
});
```

```

        }
        break;
    case 1:
        ResultKt.throwOnFailure($result);
        break;
    default:
        throw new IllegalStateException("call to 'resume'
before 'invoke' with coroutine");
    }

    return Unit.INSTANCE;
}

@NotNull
public final Continuation create(@NotNull Continuation
completion) {
    Intrinsics.checkNotNullParameter(completion,
"completion");
    Function1 var2 = new <anonymous constructor>
(completion);
    return var2;
}

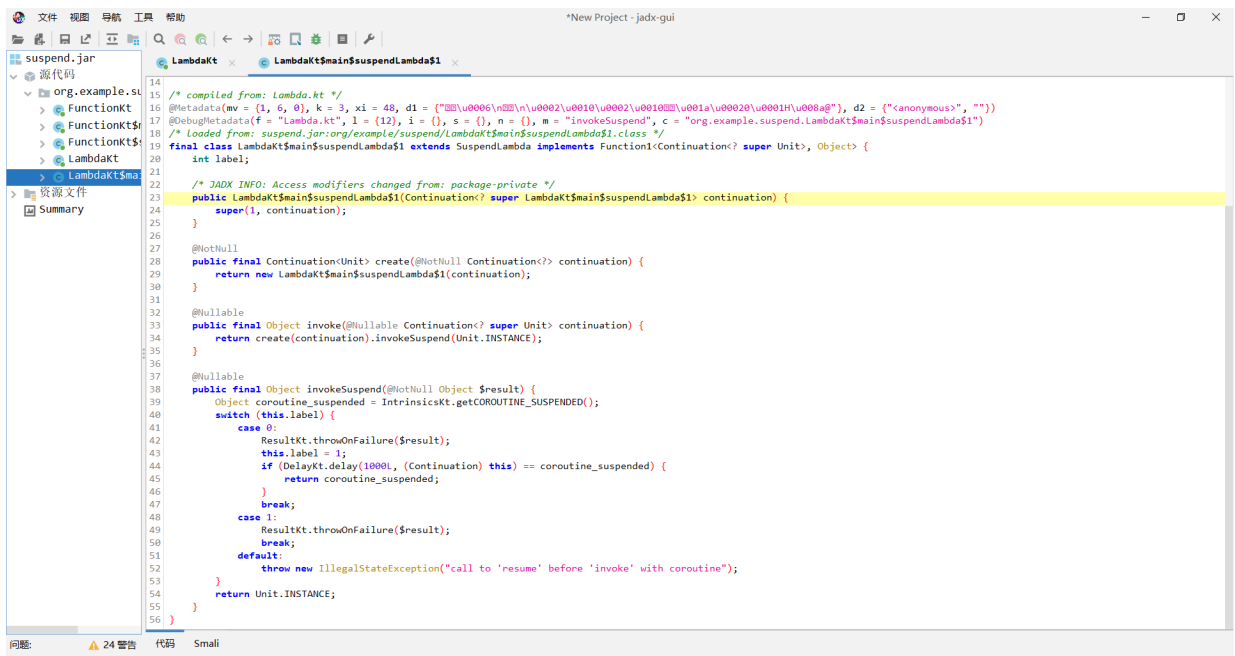
public final Object invoke(Object var1) {
    return
((<undefinedtype>)this.create((Continuation)var1)).invokeSuspend(Uni
t.INSTANCE);
}
});

```

有些奇怪，创建的时候需要传入Continuation，不过这个时候传入的是null，在调用invoke的时候才new了一个新的Function返回。有些绕弯子的赶紧。

为了保险起见，把jar包也反编译一下

- decompile



和show kotlin bytecode有点差别了。

这个**lambda**还实现了一个叫**SuspendLambda**的抽象类。

所以现在出现分歧了。到底是相信show kotlin bytecode还是相信jadx的反编译结果。

个人是选择相信jadx，因为jar包已经是写死了，几乎是不会发生什么变化了。而show kotlin bytecode是怎么工作的我们暂时不得而知，而且show kotlin bytecode并不是说和jadx的反编译结果完全不一样，而是说少了一点东西。所以这里选择相信jadx的结果，因为比show kotlin bytecode更为详细。（如果有狠人能读懂字节码，那当我没说，字节码更是直击本质

小结

我们直接定义的suspend function和suspend lambda有点区别，suspend function是通过改变函数的形参，返回值和函数体的。而suspend lambda在suspend function的基础上还修改了它的父类让他继承自SuspendLambda这个抽象类。这看似不起眼，但是会为我们后续深入去学习挂起函数提供非常有力的帮助。

suspend深入

suspend function

该部分内容会一步步分析挂起函数从挂起到恢复的内容。抓好扶手，快车即刻出发了。

测试代码

```
suspend fun main() {  
    test()  
}  
  
suspend fun test() {  
    println("1")  
    delay(1000)  
}
```

```

        println("2")
        delay(1000)
        println("3")
        delay(1000)
        println("4")
        delay(1000)
        println("5")
        delay(1000)
    }

```

suspend main

只要是suspend函数都需要一个continuation

而这个suspend main明面上好像是没有continuation，那肯定是暗地里给了continuation的。

反编译一下

```

public static void main(String[] var0) {
    RunSuspendKt.runSuspend(new FunctionUpperKt$$$main(var0));
}

```

调用了runSuspend，而RunSuspend就是最初的Continuation

```

@SinceKotlin("1.3")
internal fun runSuspend(block: suspend () -> Unit) {
    val run = RunSuspend()
    block.startCoroutine(run)
    run.await()
}

```

然后就调用到了挂起函数test这里。

```

@Nullable
public static final Object test(@NotNull Continuation var0) {
    Object $continuation;
    label57: {...}

    String var1;
    Object var4;
    label48: {...}

    var1 = "5";
    System.out.println(var1);
    (((<undefinedtype>)$continuation).label = 5;
    if (DelayKt.delay(1000L, (Continuation)$continuation) == var4) {...} else {...}
}

```

第一步为test这个挂起函数初始化continuation。

这里之前已经分析过了就不分析了。

```
Object $continuation;
```

```
label57: {
```

```
    if (var0 instanceof <undefinedtype>) {...}
```

```
    $continuation = new ContinuationImpl(var0) {...};
```

```
}
```

第二步主要是进行的挂起函数的流程调度

代码如下

```
String var1;
Object var4;
label48: {
    label47: {
        label46: {
            Object $result = ((<undefinedtype>) $continuation).result;
            var4 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
            switch (((<undefinedtype>) $continuation).label) {
                case 0:
                    ResultKt.throwOnFailure($result);
                    var1 = "1";
                    System.out.println(var1);
                    ((<undefinedtype>) $continuation).label = 1;
                    if (DelayKt.delay(1000L, (Continuation) $continuation)
== var4) {
                        return var4;
                    }
                    break;
                case 1:
                    ResultKt.throwOnFailure($result);
                    break;
                case 2:
                    ResultKt.throwOnFailure($result);
                    break label46;
                case 3:
                    ResultKt.throwOnFailure($result);
                    break label47;
                case 4:
                    ResultKt.throwOnFailure($result);
                    break label48;
                case 5:
                    ResultKt.throwOnFailure($result);
                    return Unit.INSTANCE;
            }
        }
    }
}
```



```

        default:
            throw new IllegalStateException("call to 'resume'
before 'invoke' with coroutine");
        }

        var1 = "2";
        System.out.println(var1);
        (((<undefinedtype>)$continuation).label = 2;
        if (DelayKt.delay(1000L, (Continuation)$continuation) ==
var4) {
            return var4;
        }
    }

    var1 = "3";
    System.out.println(var1);
    (((<undefinedtype>)$continuation).label = 3;
    if (DelayKt.delay(1000L, (Continuation)$continuation) ==
var4) {
        return var4;
    }
}

var1 = "4";
System.out.println(var1);
(((<undefinedtype>)$continuation).label = 4;
if (DelayKt.delay(1000L, (Continuation)$continuation) == var4)
{
    return var4;
}
}

```

1.获取执行结果获取挂起标记

```

Object $result = (((<undefinedtype>)$continuation).result;
var4 = IntrinsicsKt.getCOROUTINE_SUSPENDED();

```

2.进入switch分支

```

switch(((<undefinedtype>)$continuation).label) {
case 0:
    ResultKt.throwOnFailure($result);
    var1 = "1";
    System.out.println(var1);
    ((<undefinedtype>)$continuation).label = 1;
    if (DelayKt.delay(1000L, (Continuation)$continuation) == var4) {...}
    break;
case 1:
    ResultKt.throwOnFailure($result);
    break;
case 2:
    ResultKt.throwOnFailure($result);
}

```

除了case 0和default以外语序所有分支都是有规律的。不过总之每个分支都引导到了一个挂起函数调用处。（这里只调用了挂起函数delay）

调用delay会返回返回一个枚举类也就是COROUTINE_SUSPENDED,最后挂起结束后会调用BaseContinuationImpl中的resumeWith。

```

public final override fun resumeWith(result: Result<Any?>) {
    // This loop unrolls recursion in current.resumeWith(param) to
    // make saner and shorter stack traces on resume
    var current = this
    var param = result
    while (true) {
        // Invoke "resume" debug probe on every resumed
        // continuation, so that a debugging library infrastructure
        // can precisely track what part of suspended callstack was
        // already resumed
        probeCoroutineResumed(current)
        with(current) {
            val completion = completion!! // fail fast when trying
            // to resume continuation without completion
            val outcome: Result<Any?> =
                try {
                    val outcome = invokeSuspend(param)
                    if (outcome === COROUTINE_SUSPENDED) return
                    Result.success(outcome)
                } catch (exception: Throwable) {
                    Result.failure(exception)
                }
            releaseIntercepted() // this state machine instance is
            // terminating
            if (completion is BaseContinuationImpl) {
                // unrolling recursion via loop
                current = completion
            }
        }
    }
}

```

```
        param = outcome
    } else {
        // top-level completion reached -- invoke and return
        completion.resumeWith(outcome)
        return
    }
}
}
```

`resumeWith`的实现很简单，只要挂起函数代码没执行完毕and没有被挂起就会死循环其调用 `invokeSuspend`一直到挂起函数执行完毕。最后当当前挂起函数执行完毕会继续调用 挂起函数的调用函数。（也就是调用这个挂起函数的挂起函数，直到所有的层级执行完毕才 `return`）。

小结

挂起函数的实现较为简单

- 挂起的实现就是 `return` 当前函数
- 恢复逻辑实现是通过使用 `Continuation` 保存挂起函数调用信息，再通过 `resumeWith` 实现挂起函数的恢复逻辑。

后续学习

上述讲述的内容较为抽象和简略，基础层内容远不止这些，后续学习可参考自我的github的 `coroutine` 部分。[链接](#)

高级框架层

基础层提供了最底层的挂起，恢复的支持。在次基础上开发一套实用的框架。

常见的 `delay`、`flow`、`channel`、`dispatchers` 等等。

一切包名为 `kotlinx.coroutines` 开头的都是高级框架层里面的东西。

具体 `api` 可参考[官方文档](#)

delay

我们经常实用这个东西。它的出现是用于取代 `sleep` 的。

当我们在一个挂起函数中实用 `sleep` 的时候，会报黄

```
suspend fun main(){
```

```
    Thread.sleep( millis: 1000)
```

```
}
```

Possibly blocking call in non-blocking context could lead to thread starvation

Wrap call in 'withContext' Alt+Shift+Enter More actions... Alt+Enter

在非阻塞式的上下文中调用这行代码有可能会造成线程匮乏。

也就是说suspend本来是非阻塞式的，可能会导致线程资源没有得到很好的利用。

delay & sleep

优劣已经有了明显的区分，在资源利用的方面。*sleep*明显略于*delay*

这就得好好讲讲Thread.sleep了，Thread.sleep是一个native的调用，因为线程属于是操作系统的资源，所以一切与线程控制相关的都需要和操作系统打交道。

sleep是一个非阻塞式的调用，它没有让线程卡着，但是它使得线程进入了休眠状态，虽然没有阻塞但是这个线程也不能用了。这会导致资源的利用低，因为在线程休眠的那段时间完全可以腾出来执行其他的任务。

而delay就采用的另外的方法使得资源的利用率得以提升，当你在一个函数内调用delay的时候不会导致线程休眠，而只会让协程被挂起，协程被挂起的以后之后当前线程就自由了，该干嘛干嘛，当挂起恢复的时候，又会继续执行。这样就减少了休眠导致线程资源利用低的问题。

实现

- 测试代码

```
suspend fun main() {  
    delay(1000)  
}
```

- 流程分析

delay具体实现

```
public suspend fun delay(timeMillis: Long) {  
    if (timeMillis <= 0) return // don't delay  
    return suspendCancellableCoroutine { cont:  
        CancellableContinuation<Unit> ->  
            // if timeMillis == Long.MAX_VALUE then just wait forever like  
            awaitCancellation, don't schedule.  
            if (timeMillis < Long.MAX_VALUE) {  
                cont.context.delay.scheduleResumeAfterDelay(timeMillis,  
cont)  
            }  
        }  
    }  
}
```

先判断timMillis的合法性

然后调用suspendCancellableCoroutine

而关于suspendCancellableCoroutine在作甚，源码里的注释写的很清楚

Suspends the coroutine like suspendCoroutine, but providing a CancellableContinuation to the block.

类似于suspendCoroutine,但是在block中提供了CancellableContinuation

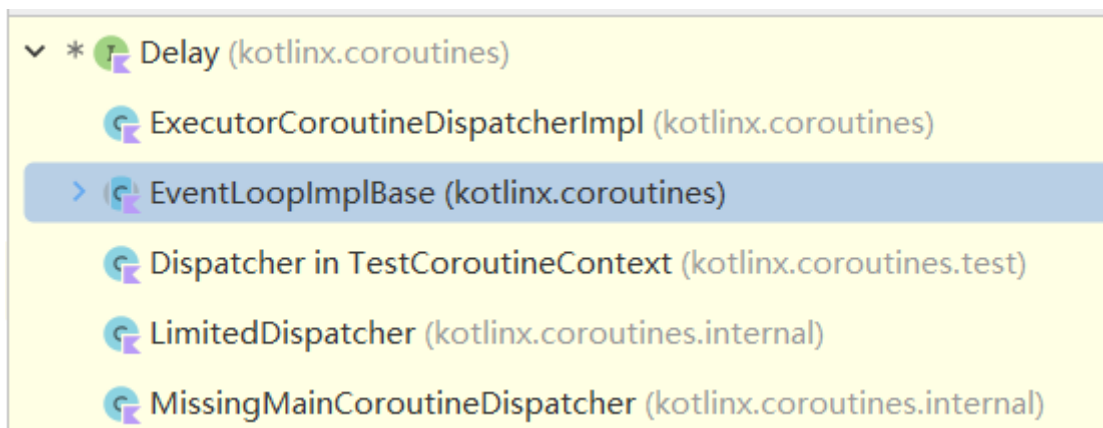
然后开始调度delay了

```
cont.context.delay.scheduleResumeAfterDelay(timeMillis, cont)
```

然后拿了context内的delay,而delay是是CoroutineContext的一个扩展属性。

```
internal val CoroutineContext.delay: Delay get() =  
    get(ContinuationInterceptor) as? Delay ?: DefaultDelay
```

将CoroutineContext的拦截器拿出来，然后强转为Delay，失败就使用默认的Delay实现实现Delay的就如下几个类。（这个和delay实现有些关联）



紧接着看看默认实现

DefaultExecutor

```
internal actual object DefaultExecutor : EventLoopImplBase(), Runnable
```

由于使用的suspend main所以delay使用的default。

```

229 while (processNextEvent() <= 0) { /* spin */ }
230 // reschedule the rest of delayed tasks
231 rescheduleAllDelayed()
232 }
233
234 public override fun scheduleResumeAfterDelay(timeMillis: Long, continuation: Cance
235 val timeNanos = delayToNanos(timeMillis) timeMillis: 1000
236 if (timeNanos < MAX_DELAY_NS) {
237     val now = nanoTime()
238     DelayedResumeTask(nanoTime: now + timeNanos, continuation).also { task ->
239         continuation.disposeOnCancellation(task)
240         schedule(now, task)
241     }
242 }
243 }
244
245 protected fun scheduleInvokeOnTimeout(timeMillis: Long, block: Runnable): Disposab
246 val timeNanos = delayToNanos(timeMillis)
247 return if (timeNanos < MAX_DELAY_NS) {
248     val now = nanoTime()
249     DelayedRunnableTask(nanoTime: now + timeNanos, block).also { task ->

```

new了一个Task，然后加入了取消的回调，继续调用schedule调度

schedule就蛮有意思的，当满足调度的时候就调用unpack

```

public fun schedule(now: Long, delayedTask: DelayedTask) {
    when (scheduleImpl(now, delayedTask)) {
        SCHEDULE_OK -> if (shouldUnpark(delayedTask)) unpark()
        SCHEDULE_COMPLETED -> reschedule(now, delayedTask)
        SCHEDULE_DISPOSED -> {} // do nothing -- task was already
disposed
        else -> error("unexpected result")
    }
}

```

```

protected actual fun unpark() {
    val thread = thread // atomic read
    if (Thread.currentThread() != thread)
        unpark(thread)
}

```

也就开启一个线程。

不同于sleep直接让当前线程休眠，delay没有挂起当前的线程，而是直接把休眠这一任务放到了一个新的线程里面。

随着源码分析的逐步深入，你会发现kotlin协程的“线程味”越来越重。

因为kotlin协程本就不是为了淘汰线程，只是为了让线程变得更好用，简而言之也就是封装线程。

flow

*flow*类似于*rxjava*，但比*rxjava*更为轻量级，实现上更为简洁。

*flow*的操作符相比于*rxjava*要少上不多，但是由于*flow*大量运用挂起函数这使得*flow*的灵活度很高，*flow*在很大程度上能替换*rxjava*，但是在复杂逻辑下*rxjava*更具优势，这并不影响*flow*接入我们的日常使用。

flow基本使用

```
suspend fun main() {
    testFlow()
        .map {
            (it + 1).toString()
        }
        .collect {
            println(it)
        }
}

fun testFlow() = flow<Int> {
    repeat(100) {
        delay(1000)
        emit(it)
    }
}
```

可以发现

- 创建*flow*不会挂起，但是订阅*flow*是会被挂起的。
- 而且在调用*collect*订阅之前是不会触发*flow*向下游*emit*数据的。（因为这个特性所以说*flow*叫做冷流，也就是说只有当有消费者订阅的时候才会触发数据流的流动。于此相反的还有一个热流，无论是否有订阅，都会引发数据流的流动）。
- *flow*与*rxjava*较为类似

flow创建

*flow*创建操作符主要有两类一类是自由型，一类是方便的扩展。

flow

这种方式的自由度比较高。

```
fun flowBuilder1() = flow<Int> {
    repeat(1000) {
        emit(it)
    }
}
```

flowOf

```
fun flowBuilder3() = flowOf(1,2,3,4,5)
```

asFlow

```
fun flowBuilder2() = (0..1).asFlow()
```

转换操作符

map

*map*操作符是将上游的数据进行一次转化然后传播到下游。对于挂起函数提供了一个特性的支持，也就是无论这个转化有多么耗时，或者说异步逻辑多么复杂都可以同步的逻辑代码写出来(这也是挂起函数的优势所在)。

测试代码

```
suspend fun main() {
    mapTransform().map {
        it + 1
    }.collect {
        println(it)
    }
}

fun mapTransform() = flow<Int> {
    repeat(1000) {
        emit(it)
    }
}
```

map传入的lambda是一个挂起的lambda，而且会把返回值发送给下游。

transform

*transform*操作符相比于前面的*map*来说，灵活度要高上不少，它可以实现大多数的操作符，因为他把*collector*暴露了出来。这样整个执行逻辑就完全暴露给我们自己定义了


```
suspend fun main() {
    flowTransform().transform<Int,String> {
        emit(" emit1: $it ")
        emit(" emit2: $it ")
    }
}

fun flowTransform() = flow<Int> {
    repeat(1000) {
        emit(it)
    }
}
```

size-limiting operation (限长操作符)

*flow*是数据流，也就是说数据从一个数据源开始，不断流向下游发送数据。

限长操作符也是操作符的一种，它提供的功能也就是限制数据流的长度，只接受特定长度的数据，在接受特定长度的数据以后就直接取消*flow*。

take

```
suspend fun main() {
    takeOperation()
        .take(10)
        .collect {
            println(it)
        }
}

fun takeOperation() = flow<Int> {
    repeat(1000) {
        emit(it)
    }
}
```

terminal flow operator(终端操作符or末端操作符)

终端操作符用于开启*flow*收集，类似于*collect*。

toList & toSet

```
suspend fun main() {
    terminalOperation()
        .collect()

    terminalOperation()
        .toList()
}
```

```

        terminalOperation()
            .toSet()
    }

    fun terminalOperation() = flow<Int> {
        repeat(1000) {
            emit(it)
        }
    }
}

```

first & single

first 和 *single* 很类似，但是差别也很明显，

first 之后取第一个数据，而 *single* 也是类似的只会取第一个数据。

但是 *single* 会去判断是否有多的数据，如果有多的就抛异常，而 *first* 则不会去管那些，只要能拿到第一个数据即可。

简而言之就是，*first* 只管拿第一个数据，能拿到那就完事大吉。

而 *single* 不仅要拿到第一个，它还要 *flow* 中只有一个数据，如果超过一个就会抛异常。

```

suspend fun main() {
    terminalOperation()
        .first()

    terminalOperation()
        .single()
}

fun terminalOperation() = flow<Int> {
    repeat(1000) {
        emit(it)
    }
}

```

reduce & fold

reduce 和 *fold* 其实是类似的，*fold* 可看成是 *reduce plus*，*reduce* 只能用于那种累计前和累计后的值是相同的操作，而 *fold* 就没有那种限制。

```

suspend fun main() {
    terminalOperation().reduce { accumulator, value ->
        accumulator + value
    }

    terminalOperation().fold("") { acc: String, value: Int ->

```

```

        acc + value.toString()
    }
}

fun terminalOperation() = flow<Int> {
    repeat(1000) {
        emit(it)
    }
}

```

flow context(flow协程上下文)

*flow*上下文牵扯到了，*flow*的协程切换。

withContext

由于*flow*默认是在当前的协程上下文开启订阅，所以想要切换协程的上下文，最容易想到的方法就是通过*withContext*切换协程的上下文。

```

suspend fun main() {
    withContext(Dispatchers.Default) {
        flowContext().collect {
            println(Thread.currentThread().name)
        }
    }
}

fun flowContext() = flow<Int> {
    repeat(100) {
        emit(it)
    }
}

```

不过呢使用*withContext*是有一定限制的，不允许在*flow*代码块内使用*withContext*，因为*flow*不允许*collect*协程与构建器的*emit*所在协程不一致。否则就会报错

flowOn

这个操作是用于修改*flow*的执行的协程上下文

```

suspend fun main() {
    flowContext()
        .flowOn(Dispatchers.IO)
        .collect {
            println(Thread.currentThread().name)
        }
}

```

```
fun flowContext() = flow<Int> {  
    repeat(100) {  
        println("emit ${Thread.currentThread().name}")  
        emit(it)  
    }  
}
```

flow基础流程解析

看似很恐怖，但是你要记住`flow`是借鉴的`rxjava`，`rxjava`核心就是它的操作符，它的操作符可以是非常庞大的，但是原理都是类似的。一通百通。

`flow`有三类操作符

1. 创建操作符
2. 转换操作符
3. 终端操作符

后续的分析也会按照这个进行分类

创建操作符

创建操作符有三类

1. `asFlow`
2. `flowOf`
3. `flow`

```
fun createOperation1() = (0..1).asFlow()  
  
fun createOperation2() = flowOf(1, 2, 3)  
  
fun createOperation3() = flow {  
    repeat(100) {  
        emit(it)  
    }  
}
```

`asFlow`和`flowOf`大多底层是直接调用`flow`

所以分析创建操作符只需分析`flow`即可

然而`flow`只是创建了一个`Flow`的实现类仅此而已

```
public fun <T> flow(@BuilderInference block: suspend FlowCollector<T>.{()
-> Unit}): Flow<T> = SafeFlow(block)
```

转化操作符

```
fun main() {
    createOperation1()
        .map { it * it }
        .map { it.toString() }
}
```

转换操作符中大多是直接调用的transform

如map, flatMapXXX, filter, 等待一系列。

但是你在点开transform一看会发现，它底层是创建一个flow的匿名对象，然后把FlowCollector的扩展高阶函数暴露出去。

所以呢搞半天flow从创建到转化只是在不断的new Flow实现，根本没有开始调度执行。

终端操作符

终端操作符只是让flow开始流动起来，它像是一个开关，让flow流动起来。

```
suspend fun main() {
    createOperation1()
        .map { it * it }
        .map { it.toString() }
        .collect()
}
```

串联分析

```
suspend fun main() {
    flow {
        repeat(100) {
            emit(it)
        }
    }
        .map { it * it }
        .map { it.toString() }
        .collect()
}
```

- flow做了两件事情就是：创建一个SafeFlow，并提供向下游发送的方法
- map做了好几件事情
 - 返回一个新的流

- 当新的流被订阅的时候订阅上游的流
- 向下游发送数据
- `collect`引发整个数据流的订阅关系，使流开始流动。

`flow`中有两个比较重要的概念那就是`Flow`以及`FlowCollector`。

`flow`框架遵循响应式编程，“流式”的概念比较重。在`flow`中分了很多的“流”，下游的流订阅上游的流，上游的“流”可以向下游的流发送内容。

其中`Flow`就是“流”本身，`FlowCollector`用于上下流发送数据。

`flow`建立订阅分为两个步骤。

- 在执行到`collect`前会形成下游`collect`上游的关系
- 在执行`collect`后会形成上游到下游的数据通道

由于上面的订阅才使得上下游的数据通信成为可能。

- `flow`创建分析

```
flow {  
    repeat(100) {  
        emit(it)  
    }  
}
```

创建了一个`SafeFlow`，然后返回。

- `map`

```
public inline fun <T, R> Flow<T>.map(crossinline transform: suspend  
(value: T) -> R): Flow<R> = transform { value ->  
    return@transform emit(transform(value))  
}
```

```
internal inline fun <T, R> Flow<T>.unsafeTransform(  
    @BuilderInference crossinline transform: suspend FlowCollector<R>.  
(value: T) -> Unit  
) : Flow<R> = unsafeFlow { // Note: unsafe flow is used here, because  
unsafeTransform is only for internal use  
    collect { value ->  
        // kludge, without it Unit will be returned and TCE won't kick  
in, KT-28938  
        return@collect transform(value)  
    }  
}
```

```
internal inline fun <T> unsafeFlow(@BuilderInference crossinline block:
suspend FlowCollector<T>.(()) -> Unit): Flow<T> {
    return object : Flow<T> {
        override suspend fun collect(collector: FlowCollector<T>) {
            collector.block()
        }
    }
}
```

类似的返回了一个Flow的匿名实现类

- 订阅

订阅先是触发了末端流（也就是最后一个map）的collect

```
public suspend fun Flow<*>.collect(): Unit = collect(NopCollector)
```

然后map所对应的流是这样的

```
internal inline fun <T> unsafeFlow(@BuilderInference crossinline block:
suspend FlowCollector<T>.(()) -> Unit): Flow<T> {
    return object : Flow<T> {
        override suspend fun collect(collector: FlowCollector<T>) {
            collector.block()
        }
    }
}
```

它会调用this.collect

```
internal inline fun <T, R> Flow<T>.unsafeTransform(
    @BuilderInference crossinline transform: suspend FlowCollector<R>.
(value: T) -> Unit
): Flow<R> = unsafeFlow { // Note: unsafe flow is used here, because
unsafeTransform is only for internal use
    collect { value ->
        // kludge, without it Unit will be returned and TCE won't kick
in, KT-28938
        return@collect transform(value)
    }
}
```

this属于是上游的流，所以就一层层传递到了最顶层，也就是flow构建器那层。

这里还需要注意一点的是collect的新参，this.collect需要一个FlowCollector,而FlowCollector是一个SAM接口，也就是说在订阅上游的时候传入了当前流的FlowCollector。

顶层的collect，将下游的collector包了一层，然后调用了collectSafely

```
public final override suspend fun collect(collector: FlowCollector<T>) {
    val safeCollector = SafeCollector(collector, coroutineContext)
    try {
        collectSafely(safeCollector)
    } finally {
        safeCollector.releaseIntercepted()
    }
}
```

```
private class SafeFlow<T>(private val block: suspend FlowCollector<T>.{()
-> Unit}) : AbstractFlow<T>() {
    override suspend fun collectSafely(collector: FlowCollector<T>) {
        collector.block()
    }
}
```

调用了block, 而block也就是flow传入的高阶函数,
也就是这个

```
{
    repeat(100) {
        emit(it)
    }
}
```

在调用emit的时候就开启往下游发送数据了.

顶层的Collector有些不太一样。是个SafeCollector

```
override suspend fun emit(value: T) {
    return suspendCoroutineUninterceptedOrReturn { uCont ->
        try {
            emit(uCont, value)
        } catch (e: Throwable) {
            // Save the fact that exception from emit (or even check
            context) has been thrown
            lastEmissionContext = DownstreamExceptionElement(e)
            throw e
        }
    }
}
```

在检查了一些参数以后会调用emitFun (他是emit的函数类型引用), 这里是调用SafeCollector包裹的collector的emit, 这样就调用到了flow构建器下游的collector的emit函数。


```
private fun emit(uCont: Continuation<Unit>, value: T): Any? {
    val currentContext = uCont.context
    currentContext.ensureActive()
    // This check is triggered once per flow on happy path.
    val previousContext = lastEmissionContext
    if (previousContext !== currentContext) {
        checkContext(currentContext, previousContext, value)
    }
    completion = uCont
    return emitFun(collector as FlowCollector<Any?>, value, this as
Continuation<Unit>)
}
```

下游的emit调用了外层的扩展函数

```
internal inline fun <T, R> Flow<T>.unsafeTransform(
    @BuilderInference crossinline transform: suspend FlowCollector<R>.(value: T) -> Unit
): Flow<R> = unsafeFlow { // Note: unsafe flow is used here, because unsafeTransform is only for internal use
    collect { value ->
        // kludge, without it Unit will be returned and TCE won't kick in, KT-28938
        return@collect transform(value)
    }
}
```

外层扩展函数呢，调用了map传入的高阶函数，然后发送给下游

```
public inline fun <T, R> Flow<T>.map(crossinline transform: suspend
(value: T) -> R): Flow<R> = transform { value ->
    return@transform emit(transform(value))
}
```

下游又会触发，进一步进行转化。最后到尾端。

```
internal inline fun <T, R> Flow<T>.unsafeTransform(
    @BuilderInference crossinline transform: suspend FlowCollector<R>.(value: T) -> Unit
): Flow<R> = unsafeFlow { // Note: unsafe flow is used here, because unsafeTransform is only for internal use
    collect { value ->
        // kludge, without it Unit will be returned and TCE won't kick in, KT-28938
        return@collect transform(value)
    }
}
```

最后又会返回到flow构建器的block，继续执行直到执行完毕。

小结

可以发现flow其实实现上没有那么难，也就是上下游的订阅以及数据交互，只要把这个理清了，debug就能很顺利看懂代码。

每一个操作符底层几乎都new了一个Flow的实例，每个Flow都支持被订阅，但是在订阅的时候得交出一个FlowCollect。

正是因为如此就有了Flow下游订阅上游，上游传递数据给下游。

collect使得**flow**开始流动，同时也建立了**flow**上下游关系。

emit使得上游的数据能发送到下游。(本质上也就是调用了下游操作符的**lambda**)

flow协程切换源码分析

*flow*的协程切换依托于操作符。所以分析具体的实现只需抓住操作符进行解析即可。

具体的操作符有以下两种

- *flowOn*
- *launchIn*

flowOn

Changes the context where this flow is executed to the given context. This operator is composable and affects only preceding operators that do not have its own context. This operator is context preserving: context does not leak into the downstream flow.

在给定的*context*中执行*flow*，此操作符只会影响到它以前的操作符(没有*context*的操作符)，这个操作符会保存上下文，但是上下文不会泄漏到下游。

测试代码

除了*flowOn*其余的操作符知根知底。

```
suspend fun main() {
    contextSwitchFlow()
        .map {
            println("map1: ${Thread.currentThread().name}")
        }
        .map {
            println("map2: ${Thread.currentThread().name}")
        }
        .flowOn(Dispatchers.IO)
        .map {
            println("map3: ${Thread.currentThread().name}")
        }
        .map {
            println("map4: ${Thread.currentThread().name}")
        }
        .flowOn(Dispatchers.Default)
        .collect()
}

fun contextSwitchFlow() = flow {
    repeat(10) {
        emit(Unit)
    }
}
```

```
}  
}
```

flowOn在作甚

依据传入的额context和this的不同返回不同的Flow实例

其中如果context是Empty那就啥也不干。

如果this是FusibleFlow那就融合一下

其他的情况就会直接返回一个名为ChannelFlowOperatorImpl的Flow的实现类。

```
public fun <T> Flow<T>.flowOn(context: CoroutineContext): Flow<T> {  
    checkFlowContext(context)  
    return when {  
        context == EmptyCoroutineContext -> this  
        this is FusibleFlow -> fuse(context = context)  
        else -> ChannelFlowOperatorImpl(this, context = context)  
    }  
}
```

flowOn操作符具体实现

前面已经分析了flowOn操作符的表层。也就是返回了一个特殊的Flow实现类。（后续所有的操作符都可以采用这种方式去分析）。

现在我们可以去看看flowOn操作符的具体实现内容。

突破口呢？当然是collect了，这是ChannelFlowOperatorImpl的collect。

（其实它的实现是来源于它的父类ChannelFlowOperator）

```
override suspend fun collect(collector: FlowCollector<T>) {  
    // Fast-path: When channel creation is optional (flowOn/flowWith  
    operators without buffer)  
    if (capacity == Channel.OPTIONAL_CHANNEL) {  
        val collectContext = coroutineContext  
        val newContext = collectContext + context // compute  
        resulting collect context  
        // #1: If the resulting context happens to be the same as it  
        was -- fallback to plain collect  
        if (newContext == collectContext)  
            return flowCollect(collector)  
        // #2: If we don't need to change the dispatcher we can go  
        without channels
```

```

        if (newContext[ContinuationInterceptor] ==
collectContext[ContinuationInterceptor])
            return collectWithContextUndispatched(collector,
newContext)
    }
    // Slow-path: create the actual channel
    super.collect(collector)
}

```

上述代码只做了一件事情就是过滤。

过滤不需要进行协程切换的部分。

真正进行协程切换的操作在`super.collect`

```

override suspend fun collect(collector: FlowCollector<T>): Unit =
    coroutineScope {
        collector.emitAll(produceImpl(this))
    }

```

将当前协程转为一个`coroutineScope`然后创建一个`produceChannel`，把`produceChannel`的内容发送给下游。

不难看出`produceImpl`实则就是利用扩展底层去调用生成了`channel`。值得注意的是`collect`实在相应的`context`内调用的，这个`context`实则是`launchIn`传入的`context`

```

public open fun produceImpl(scope: CoroutineScope): ReceiveChannel<T> =
    scope.produce(context, produceCapacity, onBufferOverflow, start
= CoroutineStart.ATOMIC, block = collectToFun)

// shared code to create a suspend lambda from collectTo function in
one place
internal val collectToFun: suspend (ProducerScope<T>) -> Unit
    get() = { collectTo(it) }

protected override suspend fun collectTo(scope: ProducerScope<T>) =
    flowCollect(SendingCollector(scope))

override suspend fun flowCollect(collector: FlowCollector<T>) =
    flow.collect(collector)

```

小结

`flow`很重要的就是它的`collect`以及`emit`，`collect`完成了下游向上游订阅这一操作，而`emit`实现了向下游发送数据的操作。

事实上也如此，**flow**由一个个操作符链式调用完成一系列地操作，而操作符底层又是依靠**emit**与**collect**形成上下游订阅关系。

flowOn作为**flow**操作符的一种也难以脱离，它协程切换底层是采用的**channel**。

原理也就是**new**一个**channel**进行上下游的通信。

将**collect**操作放到了**flowOn**所对应的协程内，然后**emit**在**flowOn**所允许的协程内死循环进行。

这么说**flowOn**只是上游的协程上下文，并不会导致下游协程上下文的改变。

launchIn

基础使用

*launchIn*用于决定*collect*所在的协程上下文

```
flow<Int> {
    repeat(100) {
        delay(5000)
        println("emit ${Thread.currentThread().name}")
        emit(it)
    }
}

    .launchIn(CoroutineScope(newSingleThreadContext("a")))
```

很是清楚明了了，利用传入的**CoroutineScope** launch一个协程collect

```
public fun <T> Flow<T>.launchIn(scope: CoroutineScope): Job =
    scope.launch {
        collect() // tail-call
    }
```

小结

- **flow**是一个类似于**rxjava**的框架
- **flow**相比于**rxjava**，**flow**使用了**kt**的高级语法特性，大量使用高阶函数和协程。
- **flow**的执行依靠的是各类操作符联合实现的。（或者说是操作符链接共同完成）
- **flow**操作符的链接（或者说上下游通信）是依靠**emit**以及**collect**实现的
- **flow**的上下文切换依靠**channel**实现，改变上游的协程上下文，下游保持不变。在**flowOn**传入的**context**中开启一个**channel**的**producer**，然后在当前协程死循环去接受上游**channel**传来的数据。

channel

*channel*在概念上就是一个阻塞队列

简单使用

属于上手摸一摸*channel*了

很明显，在创建Channel对象以后可以通过send和receive来发射，获取管道内的元素。

```
runBlocking {
    val channel = Channel<Int>()
    launch {
        repeat(5) {
            channel.send(it)
        }
    }
    repeat(5) { println(channel.receive()) }
    println("Done!").takeIf { true }
}
```

进阶用法

- 管道的获取与关闭

如果把channel看成是一个水龙头，那么我们不仅要会怎么打开它，肯定还是得学怎么关闭的。（浪费水资源可耻）。

同理，只开channel不关闭。就会导致消费者不知道流什么时候关闭，就会一直等待生产者发送消息。

```
suspend fun sendAndClose() = coroutineScope {
    val channel = Channel<Int>()
    launch {
        repeat(10) {
            channel.send(it)
        }
        channel.close()
    }

    for (element in channel){
        println(element)
    }
}
```

- 花式消费

```
channel.consumeEach {

}

channel.consume {

}

channel.consumeAsFlow()
    .onEach {  }
    .map {  }
```

- channel的扩展函数

我们在使用`channel`的时候通常需要手动去`new`一个`channel`，然后在`CoroutineScope`中去`new`一个协程，然后新的协程里面去收集或者发送流。振奋人心的是`kotlinx`包里面有对应的扩展

produce

```
fun CoroutineScope.produceSome() = produce<Int> {
    repeat(3) {
        delay(1000)
        send(it)
    }
}

suspend fun main(): Unit = coroutineScope {
```

```
        produceSome()
            .consumeAsFlow()
            .collect {
                println(Thread.currentThread().name)
            }
    }
}
```

`producer`为我们使用`channel`发送数据提供便利。

我们不需要直接`new Channel`，因为`produce`内部已经帮我们`new`好了，`produce`的返回值是对应的消费者`Channel`。

这里需要注意的是`receiveAsFlow`与`consumeAsFlow`是不一样的。

`receive`只表明了我接受一个`channel`的数据流，不代表我要消费他，所以当有多个消费者对`channel`进行订阅的时候，每个消费者都能拿到对应的内容。

然而`consumeAsFlow`就完全不一样，他需要消费`channel`的数据流，这样当有数据流发送过来的时候，谁获取这个数据流是需要竞争的，最早获取数据流的消费者会将数据流消费，这样后续的其他消费者都拿不到了。

以下两种情形下的输出结果，可以说明`receive`和`consume`的差距。

```
suspend fun main(): Unit = coroutineScope {

    repeat(10) {
        launch {
            produceSome()
                .receiveAsFlow()
                .collect {
                    println(it)
                }
        }
    }
}
```



```
suspend fun main(): Unit = coroutineScope {

    repeat(10) {
        launch {
            produceSome()
                .consumeAsFlow()
                .collect {
                    println(it)
                }
        }
    }
}
```

actor

*actor*是为消费者服务的，他底层自动帮我们*new*了一个*Channel*，于此同时还开了一个协程进行消费，最后返回了一个*SendChannel*供给生产者。

```
suspend fun main(): Unit = coroutineScope {

    with(consumeSome()) {
        repeat(10) {
            send(it)
        }
    }
}

fun CoroutineScope.consumeSome() = actor<Int> {
    repeat(10) {
        delay(1000)
        val re = receive()
        println(re)
    }
}
```

- buffer(缓存)

*Channel*是有缓存的，在默认的情况下，缓存时关闭的，因为缓存是0

但是我们是可以进行自定义大小的。

这个缓存有啥用呢？

*channel*在进行*send*的时候不是直接把内容发送给下游，他是先把数据发送到缓存里面，然后下游就从缓存里面去取，这就面临一个问题，生产者如果生产地过快了，怎么办？缓存被塞满了，这个涉及到背压。毫无疑问*channel*提高了灵活性。

源码分析

*channel*好像内容不是很多。因为就基础来看就两个东西，*send*和*receive*。除此之外就是一些零散的扩展。相比于前面的*flow*难度要小不少。

channel 是什么？

*channel*在概念上是和阻塞队列是类似的。

如果从类的结构上来说，*Channel*就是一个具备收，发能力的通道。

```
public interface Channel<E> : SendChannel<E>, ReceiveChannel<E>
```

数据可以从这个通道的一段发送往另一端。很符合生产者消费者的模型。

channel & flow 区别？

最大的差别就是：

*flow*是冷流，*channel*是热流。

测试代码

```
val channel = Channel<Int>()

suspend fun main() = coroutineScope{
    launch {
        repeat(10) {
            channel.send(it)
        }
    }

    repeat(10){
        println(channel.receive())
    }
}
```

具体执行流程

- channel构建

channel的构建依靠一个顶层函数

```
public fun <E> Channel(
    capacity: Int = RENDEZVOUS,
    onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND,
    onUndeliveredElement: ((E) -> Unit)? = null
): Channel<E> =
    when (capacity) {
        RENDEZVOUS -> {
            if (onBufferOverflow == BufferOverflow.SUSPEND)
                RendezvousChannel(onUndeliveredElement) // an
efficient implementation of rendezvous channel
            else
                ArrayChannel(1, onBufferOverflow,
onUndeliveredElement) // support buffer overflow with buffered
channel
        }
        CONFLATED -> {
            require(onBufferOverflow == BufferOverflow.SUSPEND) {
                "CONFLATED capacity cannot be used with non-default
onBufferOverflow"
            }
            ConflatedChannel(onUndeliveredElement)
        }
        UNLIMITED -> LinkedListChannel(onUndeliveredElement) //
ignores onBufferOverflow: it has buffer, but it never overflows
        BUFFERED -> ArrayChannel( // uses default capacity with
SUSPEND
```

```

        if (onBufferOverflow == BufferOverflow.SUSPEND)
CHANNEL_DEFAULT_CAPACITY else 1,
        onBufferOverflow, onUndeliveredElement
    )
    else -> {
        if (capacity == 1 && onBufferOverflow ==
BufferOverflow.DROP_OLDEST)
            ConflatedChannel(onUndeliveredElement) // conflated
implementation is more efficient but appears to work in the same way
        else
            ArrayChannel(capacity, onBufferOverflow,
onUndeliveredElement)
    }
}

```

很清晰。

该函数有三个参数。

capacity-用于决定Channel的容量，这里的参数可以是一个正数或者是Channel.Factory里面定义的。

onBufferOverflow-用于决定Channel缓存满了的时候的解决策略。

onUndeliveredElement-用于处理发送但是没有被消费者接受的元素。

when表达式对**capacity**（缓存容量）进行了分类，共5大类

- RENDEZVOUS即无缓存

在此基础上还细分了两类

一类是缓存溢出策略为*suspend*

另外一类就是缓存溢出策略为合并。

- CONFLATED即只有一个缓存容量

*conflated*有两层限制.

1.缓存大小固定为1

2.不支持缓存溢出策略*BufferOverflow.SUSPEND*

- UNLIMITED即无限缓存容量

没什么说的，就无限容量。值得注意的是因为容量无限，所以不会出现缓存溢出，所以在构建上直接忽略了传入的缓存溢出策略

- BUFFERED即默认缓存容量

缓存溢出策略为*SUSPEND*的时候采取默认缓存容量即64，其他情况下采用1个大小的缓存容量。（因为多了也用不上，不采取*SUSPEND*策略以后，消息是合并式的，一个容量就够了）

- 自定义缓存容量

- send

`send`用于向`channel`中发射数据。它充当的角色是生产者。

接着分析一下默认的实现。

这里好像没有`send`。实现应该在它的父类里面

```
internal open class RendezvousChannel<E>(onUndeliveredElement:
    OnUndeliveredElement<E>?) : AbstractChannel<E>
    (onUndeliveredElement) {
    protected final override val isBufferAlwaysEmpty: Boolean get()
    = true
    protected final override val isBufferEmpty: Boolean get() = true
    protected final override val isBufferAlwaysFull: Boolean get() =
    true
    protected final override val isBufferFull: Boolean get() = true
}
```

最后在`AbstractSendChannel`中找到了具体的实现。

一看好像还挺简单的。

```
public final override suspend fun send(element: E) {
    // fast path -- try offer non-blocking
    if (offerInternal(element) == OFFER_SUCCESS) return
    // slow-path does suspend or throws exception
    return sendSuspend(element)
}
```

`offerInternal`

Tries to add element to buffer or to queued receiver. Return type is
`OFFER_SUCCESS` | `OFFER_FAILED` | `Closed`.

尝试向缓存或者队列里面的接收器里面添加元素。返回值有三个，`success`，`failed`，`closed`，每一个对应该操作的状态。

```
protected open fun offerInternal(element: E): Any {
    while (true) {
        val receive = takeFirstReceiveOrPeekClosed() ?: return
        OFFER_FAILED
        val token = receive.tryResumeReceive(element, null)
        if (token != null) {
            assert { token === RESUME_TOKEN }
            receive.completeResumeReceive(element)
            return receive.offerResult
        }
    }
}
```

从队列里面获取一个receive请求，拿到了就将对应的element传入并激活receive对应的挂起函数。

相应的，如果没有拿到对应的receive请求，就调用return，执行流就到了sendSuspend。

这里通过调用suspendCancellableCoroutineReusable挂起当前挂起函数，并获取当前协程的上下文，这里呢直接死循环调用enqueue，直到入队列成功后return。

```
private suspend fun sendSuspend(element: E): Unit =
    suspendCancellableCoroutineReusable sc@ { cont ->
        loop@ while (true) {
            if (isFullImpl) {
                val send = if (onUndeliveredElement == null)
                    SendElement(element, cont) else
                    SendElementWithUndeliveredHandler(element, cont,
                        onUndeliveredElement)
                val enqueueResult = enqueueSend(send)
                when {
                    enqueueResult == null -> { // enqueued successfully
                        cont.removeOnCancellation(send)
                        return@sc
                    }
                    enqueueResult is Closed<*> -> {

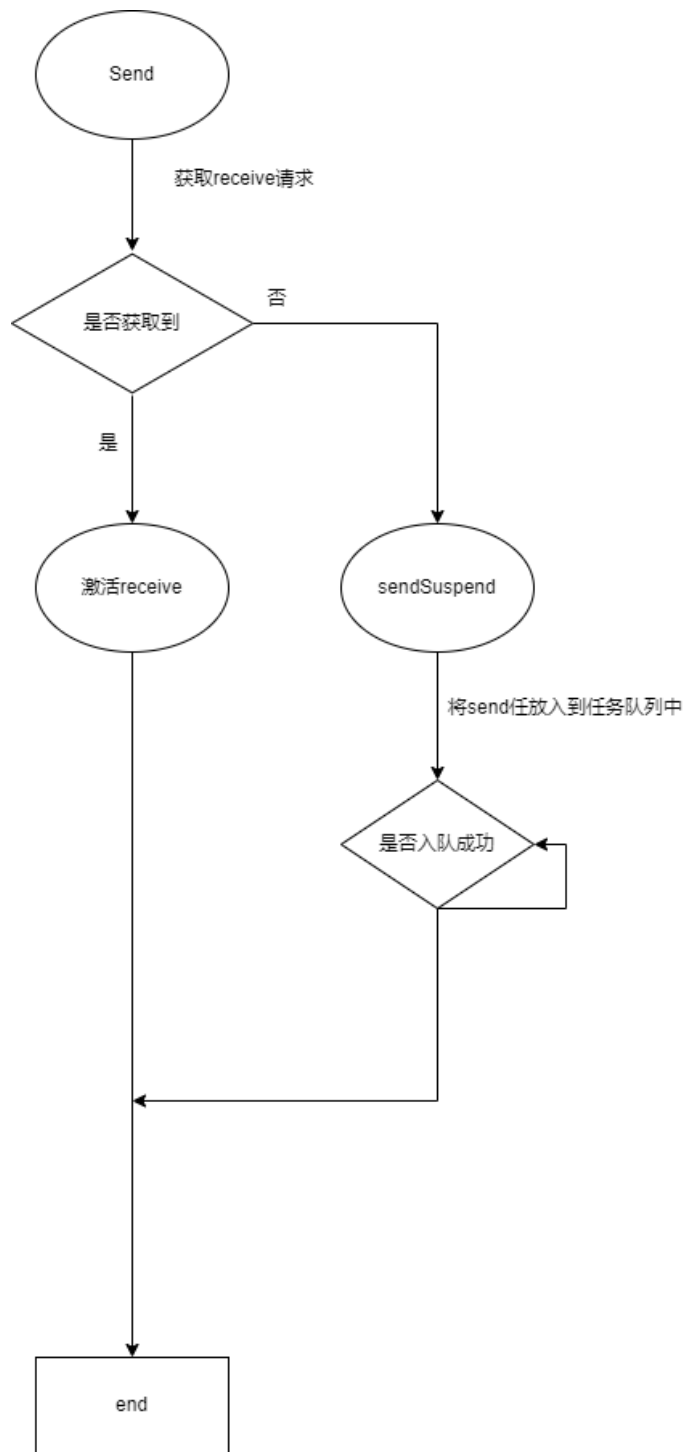
                }
                cont.helpCloseAndResumeWithSendException(element, enqueueResult)
                return@sc
            }
            enqueueResult === ENQUEUE_FAILED -> {} // try to
            offer instead
            enqueueResult is Receive<*> -> {} // try to offer
            instead
            else -> error("enqueueSend returned $enqueueResult")
        }
    }
```

```

    }
}
// hm... receiver is waiting or buffer is not full. try to
offer
val offerResult = offerInternal(element)
when {
    offerResult === OFFER_SUCCESS -> {
        cont.resume(Unit)
        return@sc
    }
    offerResult === OFFER_FAILED -> continue@loop
    offerResult is Closed<*> -> {
        cont.helpCloseAndResumeWithSendException(element,
offerResult)
        return@sc
    }
    else -> error("offerInternal returned $offerResult")
}
}
}

```

执行流程如下：



- receive

*receive*用于将*channel*中的数据拿出来。

*receive*具体实现如下.

可以发现*receive()*的实现内容和*send()*是很类似的。


```

public final override suspend fun receive(): E {
    // fast path -- try poll non-blocking
    val result = pollInternal()
    /*
        * If result is Closed -- go to tail-call slow-path that will
        allow us to
        * properly recover stacktrace without paying a performance
        cost on fast path.
        * We prefer to recover stacktrace using suspending path to
        have a more precise stacktrace.
    */
    @Suppress("UNCHECKED_CAST")
    if (result != POLL_FAILED && result !is Closed<*>) return
    result as E
    // slow-path does suspend
    return receiveSuspend(RECEIVE_THROWS_ON_CLOSE)
}

```

pollInternal()

```

protected open fun pollInternal(): Any? {
    while (true) {
        val send = takeFirstSendOrPeekClosed() ?: return POLL_FAILED
        val token = send.tryResumeSend(null)
        if (token != null) {
            assert { token === RESUME_TOKEN }
            send.completeResumeSend()
            return send.pollResult
        }
        // too late, already cancelled, but we removed it from the
        queue and need to notify on undelivered element
        send.undeliveredElement()
    }
}

```

类似于offerInternal

先从队列里面获取send请求，如果拿到了就激活send，否则就return
调用receiveSuspend

```

private suspend fun <R> receiveSuspend(receiveMode: Int): R =
    suspendCancellableCoroutineReusable { cont ->
        val receive = if (onUndeliveredElement == null)

```

```

        ReceiveElement(cont as CancellableContinuation<Any?>,
receiveMode) else
        ReceiveElementWithUndeliveredHandler(cont as
CancellableContinuation<Any?>, receiveMode, onUndeliveredElement)
while (true) {
    if (enqueueReceive(receive)) {
        removeReceiveOnCancel(cont, receive)
        return@sc
    }
    // hm... something is not right. try to poll
    val result = pollInternal()
    if (result is Closed<*>) {
        receive.resumeReceiveClosed(result)
        return@sc
    }
    if (result != POLL_FAILED) {
        cont.resume(receive.resumeValue(result as E),
receive.resumeOnCancellationFun(result as E))
        return@sc
    }
}
}
}

```

`receiveSuspend`执行逻辑如下.

类似于前面的`sendSuspend`

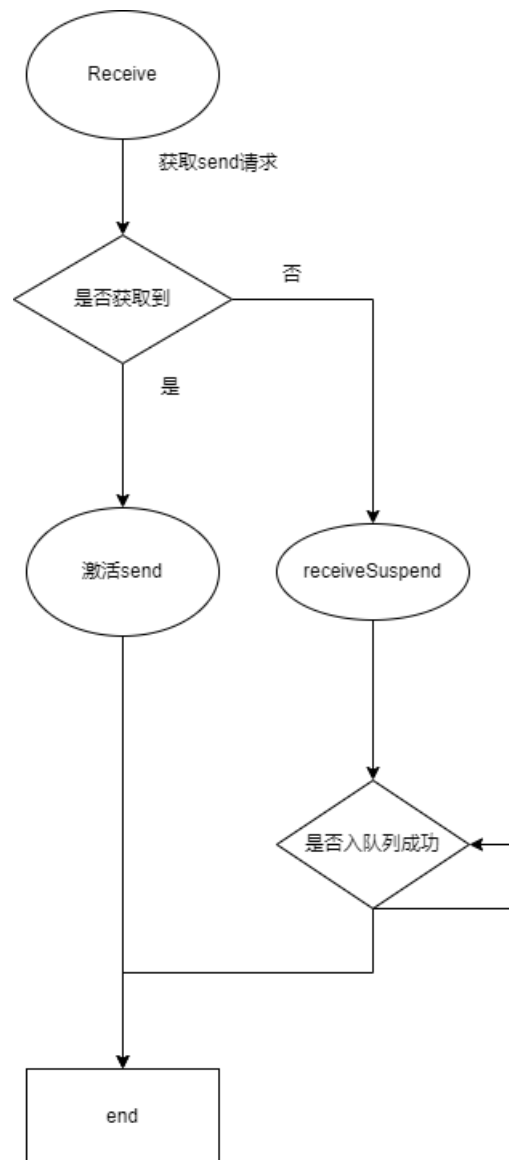
```

private suspend fun <R> receiveSuspend(receiveMode: Int): R =
suspendCancellableCoroutineReusable sc@ { cont ->
    val receive = if (onUndeliveredElement == null)
        ReceiveElement(cont as CancellableContinuation<Any?>,
receiveMode) else
        ReceiveElementWithUndeliveredHandler(cont as
CancellableContinuation<Any?>, receiveMode, onUndeliveredElement)
    while (true) {
        if (enqueueReceive(receive)) {
            removeReceiveOnCancel(cont, receive)
            return@sc
        }
        // hm... something is not right. try to poll
        val result = pollInternal()
        if (result is Closed<*>) {
            receive.resumeReceiveClosed(result)
            return@sc
        }
        if (result != POLL_FAILED) {
            cont.resume(receive.resumeValue(result as E),
receive.resumeOnCancellationFun(result as E))
            return@sc
        }
    }
}

```

```
}  
}  
}
```

流程图如下



对比`send`和`receive`的操作，可以发现。

他们的操作逻辑不是神似，是一模一样好吧。

简述：

`send`会在任务队列中寻找`receive`，找到了就`resume receive`，没拿到就把`send`放入任务队列，方便`receive`时候从任务队列获取，紧接着就是挂起。

`receive`会在任务队列中寻找`send`，找到了就`resume send`，没拿到就把`receive`放入任务队列，方便`send`时候从任务队列中获取，最后挂起。

整体流程就是 `send`和`resume`操作互相`invoke`，`send`完成以后等待`receive`获取数据，接着`receive`又会去`invoke send`，这样就达到了`channel`的通信。（一来一回）

这样默认的Channel实现(即无缓存，缓存溢出策略为挂起的Channel实现)已经分析完毕了。

剩余Channel实现类似，可自行进行分析。

小结

channel类似于一个阻塞队列，它是生产-消费模型的很好的实现。没有多少的操作符，就基础来看就只有两个方法，send & receive。

ChannelFlow & CallbackFlow & ...

- *ChannelFlow*
- *CallbackFlow*
- *StateFlow*
- *SharedFlow*

这些都是属于Flow的一部分。

ChannelFlow

Creates an instance of a cold Flow with elements that are sent to a SendChannel provided to the builder's block of code via ProducerScope. It allows elements to be produced by code that is running in a different context or concurrently. The resulting flow is cold, which means that block is called every time a terminal operator is applied to the resulting flow.

- 创建一个Flow实例
- 但是元素不是直接发送到下游，而是发送到一个channel里面

channelFlow好像就只有一个构建方式。采用channelFlow操作符。

```
public fun <T> channelFlow(@BuilderInference block: suspend
    ProducerScope<T>().() -> Unit): Flow<T> =
    ChannelFlowBuilder(block)
```

当你觉得它陌生的时候，点开它看看。

```
private open class ChannelFlowBuilder<T>(
    private val block: suspend ProducerScope<T>().() -> Unit,
    context: CoroutineContext = EmptyCoroutineContext,
    capacity: Int = BUFFERED,
    onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND
) : ChannelFlow<T>(context, capacity, onBufferOverflow) {
    override fun create(context: CoroutineContext, capacity: Int,
onBufferOverflow: BufferOverflow): ChannelFlow<T> =
        ChannelFlowBuilder(block, context, capacity, onBufferOverflow)

    override suspend fun collectTo(scope: ProducerScope<T>) =
        block(scope)

    override fun toString(): String =
        "block[$block] -> ${super.toString()}"
}
```

继续点开。

```
public abstract class ChannelFlow<T>(
    // upstream context
    @JvmField public val context: CoroutineContext,
    // buffer capacity between upstream and downstream context
    @JvmField public val capacity: Int,
    // buffer overflow strategy
    @JvmField public val onBufferOverflow: BufferOverflow
) : FusibleFlow<T>
```

FusibleFlow这好像有些熟悉在分析Flow的flowOn操作符的时候看过几眼。

所以呢，它的实现其实和flowOn有点点类似。

底层使用channel。

简单使用

```
suspend fun main() {
    val ch = channelFlow<Int> {
        repeat(10) {
            delay(100)
            send(it)
        }
    }.map {
        println(it)
    }
```

```
}

ch.collect()
ch.collect()

}
```

查看输出结果可以发现，ChannelFlow其实是一个冷流，我们每次collect操作都会触发channelFlow的lambda向下游发送数据。唯一不同的它不是向下游直接发送数据，而是发送数据给channel仅此而已。

用法

对于一项技术我们可以不是很熟悉它的原理，因为原理是用来加深我们对于这项技术的理解的，你可以没有它，但是有会更好。

但是对于一项技术会用它的是必须的，如果学了它没用武之地，学它干嘛。（反正我懒得学）。

关于它的用法源码是有注释的。

Examples of usage:

```
fun <T> Flow<T>.merge(other: Flow<T>): Flow<T> = channelFlow {
    // collect from one coroutine and send it
    launch {
        collect { send(it) }
    }
    // collect and send from this coroutine, too, concurrently
    other.collect { send(it) }
}

fun <T> contextualFlow(): Flow<T> = channelFlow {
    // send from one coroutine
    launch(Dispatchers.IO) {
        send(computeIoValue())
    }
    // send from another coroutine, concurrently
    launch(Dispatchers.Default) {
        send(computeCpuValue())
    }
}
```

channelFlow是一种flow，（一种内部封装了channel的flow）

它相比于寻常flow特殊的也就是内部封装了一个channel

这又会使得channelFlow的使用场景上有什么变化嘛?

先看两个实例

```
fun <T> Flow<T>.merge(other: Flow<T>): Flow<T> = channelFlow {  
    // collect from one coroutine and send it  
    launch {  
        collect { send(it) }  
    }  
    // collect and send from this coroutine, too, concurrently  
    other.collect { send(it) }  
}  
  
fun <T> contextualFlow(): Flow<T> = channelFlow {  
    // send from one coroutine  
    launch(Dispatchers.IO) {  
        send(computeIoValue())  
    }  
    // send from another coroutine, concurrently  
    launch(Dispatchers.Default) {  
        send(computeCpuValue())  
    }  
}
```

由于channel的‘热’特性，所以它的主要使用场景就是进行数据的合并。

第一个实例将两个Flow就行合并。

第二个实例开了两个协程将两个不同的数据源中的数据进行合并。

由此可见**ChannelFlow**的主要用途还是将不同来源的数据进行合并。

这点flow也可以做到，但是使用ChannelFlow会更简单。

CallbackFlow

就简单使用来看非常简单

```
suspend fun main() {
    val ch = channelFlow<Int> {
        repeat(10) {
            delay(100)
            println(Thread.currentThread().name)
            send(it)
        }
    }

    ch.collect()
    ch.collect()
}
```

如果对比一下你会发现其实这玩意和ChannelFlow好像是一毛一样的。

确实是这样的

当点开源码的时候也是这样的

```
public fun <T> callbackFlow(@BuilderInference block: suspend
    ProducerScope<T>.( ) -> Unit): Flow<T> = CallbackFlowBuilder(block)
```

```
private class CallbackFlowBuilder<T>(
    private val block: suspend ProducerScope<T>.( ) -> Unit,
    context: CoroutineContext = EmptyCoroutineContext,
    capacity: Int = BUFFERED,
    onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND
) : ChannelFlowBuilder<T>(block, context, capacity, onBufferOverflow) {

    override suspend fun collectTo(scope: ProducerScope<T>) {
        super.collectTo(scope)
        /*
         * We expect user either call `awaitClose` from within a block
         (then the channel is closed at this moment)
         * or being closed/cancelled externally/manually. Otherwise
         "user forgot to call
         * awaitClose and receives unhelpful ClosedSendChannelException
         exceptions" situation is detected.
         */
        if (!scope.isClosedForSend) {
            throw IllegalStateException(
                """
                'awaitClose { yourCallbackOrListener.cancel() }'
                should be used in the end of callbackFlow block.
            """
            )
        }
    }
}
```



```

        Otherwise, a callback/listener may leak in case of
external cancellation.

        See callbackFlow API documentation for the details.
        """.trimIndent()
    )
}

override fun create(context: CoroutineContext, capacity: Int,
onBufferOverflow: BufferOverflow): ChannelFlow<T> =
    CallbackFlowBuilder(block, context, capacity, onBufferOverflow)
}

```

大抵也就是一个channelFlow只是部分参数不一样了而已。

在使用上呢。这东西适用于将回调转为flow。

CallbackFlow -> Callback to Flow

用法

```

fun flowFrom(api: CallbackBasedApi): Flow<T> = callbackFlow {
    val callback = object : Callback { // Implementation of some
callback interface
        override fun onNextValue(value: T) {
            // To avoid blocking you can configure channel capacity
using
            // either buffer(Channel.CONFLATED) or
buffer(Channel.UNLIMITED) to avoid overflow
            trySendBlocking(value)
                .onFailure { throwable ->
                    // Downstream has been cancelled or failed, can log
here
                }
        }
        override fun onApiError(cause: Throwable) {
            cancel(CancellationException("API Error", cause))
        }
        override fun onCompleted() = channel.close()
    }
    api.register(callback)
    /*
    * Suspends until either 'onCompleted'/'onApiError' from the
callback is invoked
    */
}

```

```
    * or flow collector is cancelled (e.g. by 'take(1)' or because a
    collector's coroutine was cancelled).
    * In both cases, callback will be properly unregistered.
    */
    awaitClose { api.unregister(callback) }
}
```

也就是说将Callback包了一层Flow，而这个Flow又依靠Channel发送数据。

这样就做到了回调转Flow的操作。

如果你问为什么会有回调转Flow的需求？

这个嘛。如果你用过很多回调的话你应该不会问这个问题，因为回调很难用（~~谁~~都不用 还是得用。）

相比来说回调用着没有那么舒服。Flow就api来说比起回调要优雅不少。（如果不是对代码有洁癖的人，其实这样的需求也不是很常见，所以算是做个了解。通常情况下可以直接使用回调，但是如果回调的函数太多了的话，可以考虑使用Flow来替换。）

StateFlow

StateFlow的创建较为简单。

其实和Jetpack architecture 组件LiveData使用较为类似。

```
val stateFlow: StateFlow<Int> = MutableStateFlow(1)
```

基础使用

可以发现它的api介于flow和livedata之间

- 获取的stateFlow必须直接或者间接调用collect，否则消费者是拿不到数据的（即使StateFlow是热流）
- 发送数据依靠setValue和compareAndSet

```
suspend fun main() {
    val stateFlow: MutableStateFlow<Int> = MutableStateFlow(1)

    stateFlow
        .onEach {
            println(it)
        }
}
```

```

    }
    .launchIn(CoroutineScope(Dispatchers.IO))

repeat(100) {
    delay(1000)
    stateFlow.value = it
}

}

```

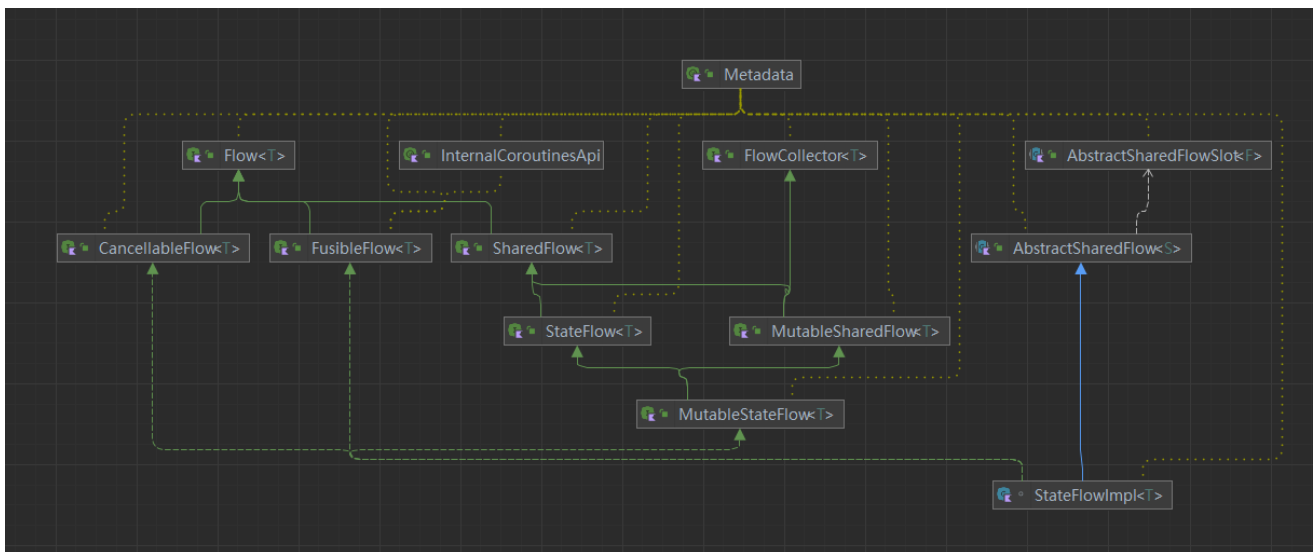
分析底层实现。

```

public fun <T> MutableStateFlow(value: T): MutableStateFlow<T> =
    StateFlowImpl(value ?: NULL)

```

类结构关系。



别的没看出多少，有一点是可以很容易发现的，那就是，**StateFlow**的父类是**SharedFlow**（所以**SharedFlow**和**StateFlow**在某种程度上是有共性的。）这也是属于后话了。

不同总体来说他们都是**Flow**，有这点就够了。

所以在认知上可认为他们是实用场景不同的**flow**即可。

- collect分析

```

override suspend fun collect(collector: FlowCollector<T>): Nothing {
    val slot = allocateSlot()
    try {

```

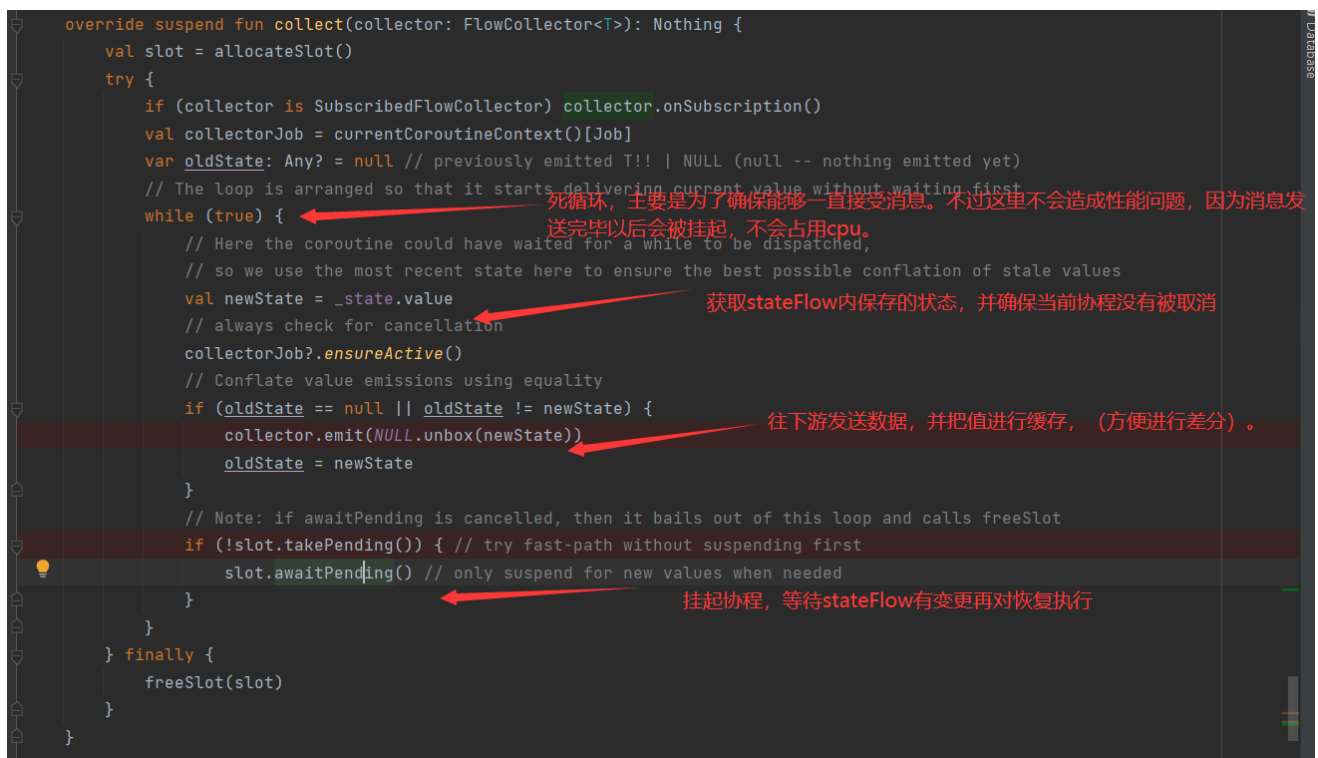
```

        if (collector is SubscribedFlowCollector)
collector.onSubscription()
        val collectorJob = currentCoroutineContext()[Job]
        var oldState: Any? = null // previously emitted T!! | NULL (null
-- nothing emitted yet)
        // The loop is arranged so that it starts delivering current
value without waiting first
        while (true) {
            // Here the coroutine could have waited for a while to be
dispatched,
            // so we use the most recent state here to ensure the best
possible conflation of stale values
            val newState = _state.value
            // always check for cancellation
            collectorJob?.ensureActive()
            // Conflate value emissions using equality
            if (oldState == null || oldState != newState) {
                collector.emit(NULL.unbox(newState))
                oldState = newState
            }
            // Note: if awaitPending is cancelled, then it bails out of
this loop and calls freeSlot
            if (!slot.takePending()) { // try fast-path without
suspending first
                slot.awaitPending() // only suspend for new values when
needed
            }
        }
    } finally {
        freeSlot(slot)
    }
}

```

核心逻辑就是查看stateFlow内部的value是否变化，如果变化了就往下游发送。

发送完毕以后就挂起等待变更后又重复执行操作。



- setValue分析

看似很短，实际上确实很短

```
public override var value: T
    get() = NULL.unbox(_state.value)
    set(value) { updateState(null, value ?: NULL) }
```

updateState的代码挺长的。这主要是为了确保线程安全。所以进行了不少加锁等操作。

不过核心逻辑也就是，

```

private fun updateState(expectedState: Any?, newState: Any): Boolean {
    var curSequence = 0
    var curSlots: Array<StateFlowSlot?> = this.slots // benign race, we will not use it
    synchronized( lock: this) {
        val oldState = _state.value
        if (expectedState != null && oldState != expectedState) return false // CAS support
        if (oldState == newState) return true // Don't do anything if value is not changing, but CAS -> true
        _state.value = newState
        curSequence = sequence
        if (curSequence and 1 == 0) { // even sequence means quiescent state flow (no ongoing update)
            curSequence++ // make it odd
            sequence = curSequence
        } else {
            // update is already in process, notify it, and return
            sequence = curSequence + 2 // change sequence to notify, keep it odd
            return true // updated
        }
        curSlots = slots // read current reference to collectors under lock
    }
    /*
    Fire value updates outside of the lock to avoid deadlocks with unconfined coroutines.
    Loop until we're done firing all the changes. This is a sort of simple flat combining that
    ensures sequential firing of concurrent updates and avoids the storm of collector resumes
    when updates happen concurrently from many threads.
    */
    while (true) {
        // Benign race on element read from array

```

修改_state的值

```

        while (true) {
            // Benign race on element read from array
            curSlots?.forEach { it: StateFlowSlot?
                it?.makePending()
            }
            // check if the value was updated again while we were updating the old one
            synchronized( lock: this) {
                if (sequence == curSequence) { // nothing changed, we are done
                    sequence = curSequence + 1 // make sequence even again
                    return true // done, updated
                }
                // reread everything for the next loop under the lock
                curSequence = sequence
                curSlots = slots
            }
        }
    }
}

```

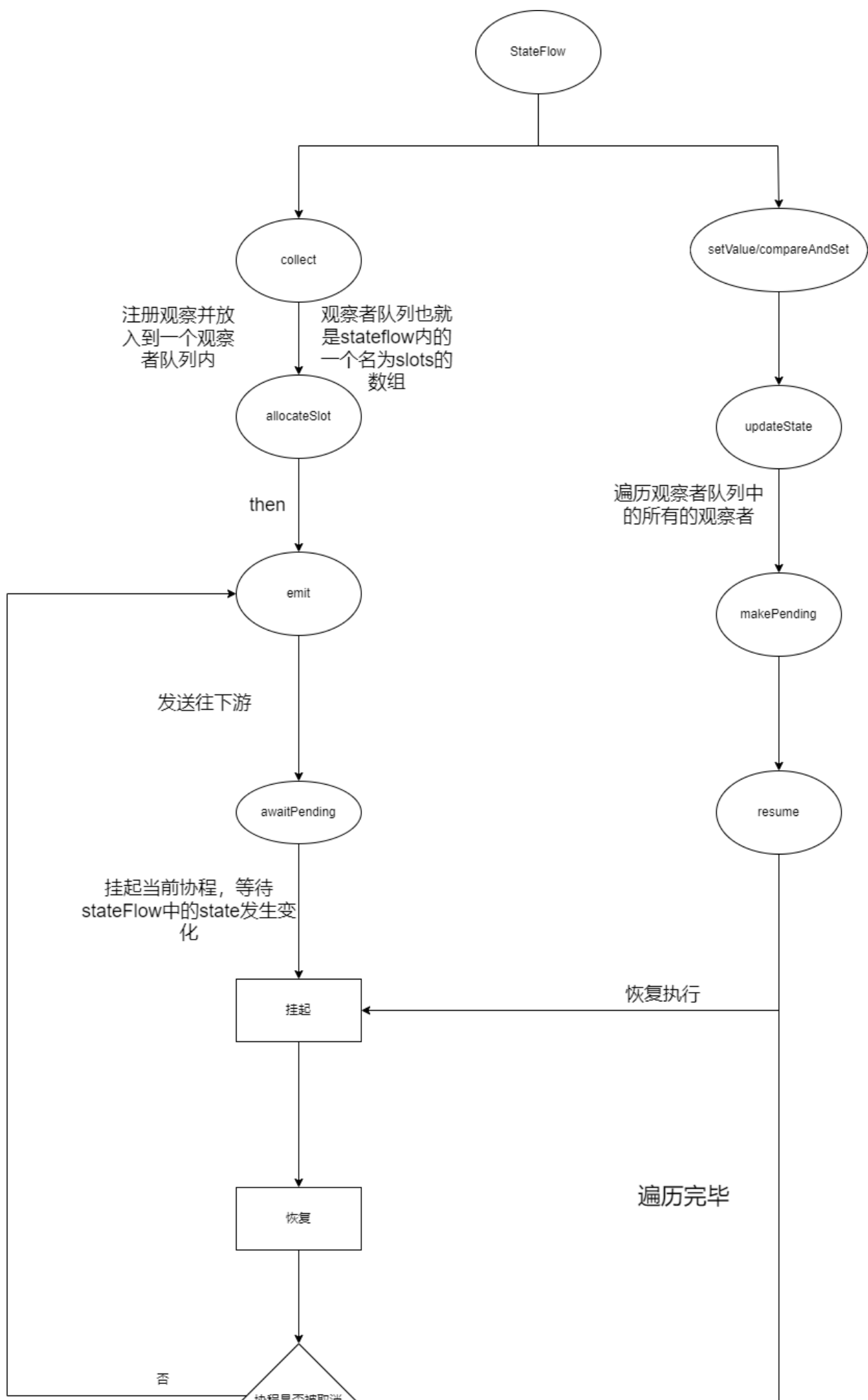
恢复每一个订阅者

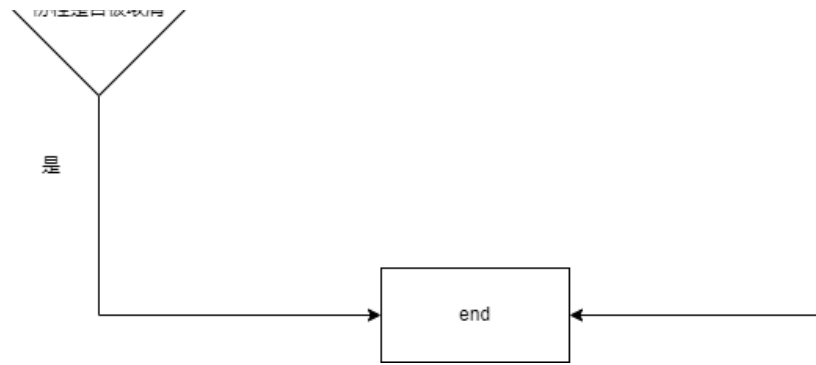
小结

就使用和定位上来看，其实StateFlow和LiveData非常类似。StateFlow可认为是LiveData plus

拥有比起LiveData更为强大的功能，同时也是一个基于kotlin协程的东西。

流程图





SharedFlow

如果有注意到前面讲`StateFlow`的话会发现，其实，他是`StateFlow`的父类。

基础使用

和`stateFlow`其实是差不多的

```
suspend fun main() = runBlocking{
    val sharedFlow = MutableSharedFlow<Int>(1)

    sharedFlow
        .onEach {
            println("A:$it")
        }
        .launchIn(CoroutineScope(Dispatchers.IO))

    sharedFlow
        .onEach {
            println("B:$it")
        }
        .launchIn(CoroutineScope(Dispatchers.Default))

    repeat(1000) {
        sharedFlow.emit(it)
    }
}
```

`launchIn`会在传入的`scope`内开启一个协程并`collect`。

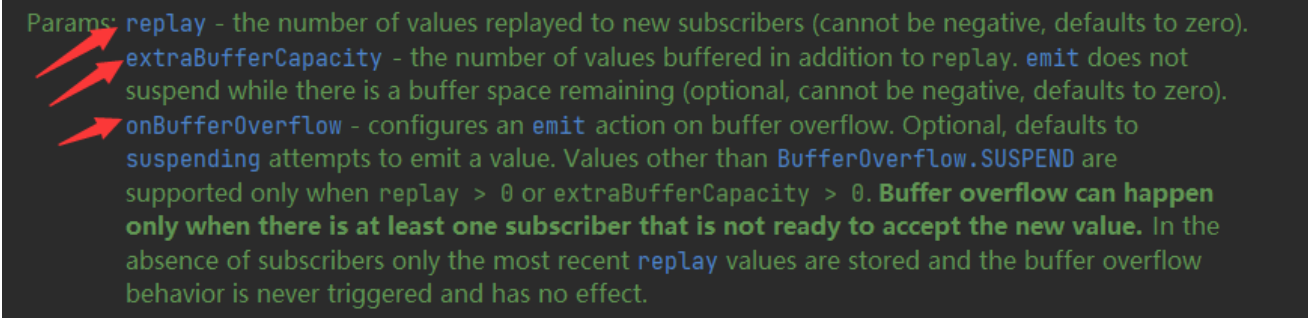
之所以使用`launchIn`原因于`StateFlow`是类似的，因为它的`collect`实质上就是一个死循环。在没有收到消息的时候会直接挂起当前协程，而如果这个时候没有生产者发送数据的话，当前协程会在`collect`处被永久性观其，也就是说`collect`后的代码永远得不到执行。

进阶使用

如果你点开`SharedFlow`的构建器，你会发现它其实可以进行部分的自定义。

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

查看一下注释



Params: `replay` - the number of values replayed to new subscribers (cannot be negative, defaults to zero).
`extraBufferCapacity` - the number of values buffered in addition to `replay`. `emit` does not suspend while there is a buffer space remaining (optional, cannot be negative, defaults to zero).
`onBufferOverflow` - configures an `emit` action on buffer overflow. Optional, defaults to `suspending` attempts to emit a value. Values other than `BufferOverflow.SUSPEND` are supported only when `replay > 0` or `extraBufferCapacity > 0`. **Buffer overflow can happen only when there is at least one subscriber that is not ready to accept the new value.** In the absence of subscribers only the most recent `replay` values are stored and the buffer overflow behavior is never triggered and has no effect.

`replay` - 发送给新订阅者的值的数量

`extraBufferCapacity` - 额外的缓存数(总缓存数 = `replay` + `extraBufferCapacity`)

`onBufferOverflow` - 缓存溢出策略。

需要注意缓存策略：

- 1.至少有一个订阅者才会触发缓存溢出策略。（如果没有订阅者，最近的值才会保存到`replay`缓存里面）
- 2.除`BufferOverflow.SUSPEND`外的所有缓存策略只有当`replay + extraBufferCapacity > 0`的时候才支持。

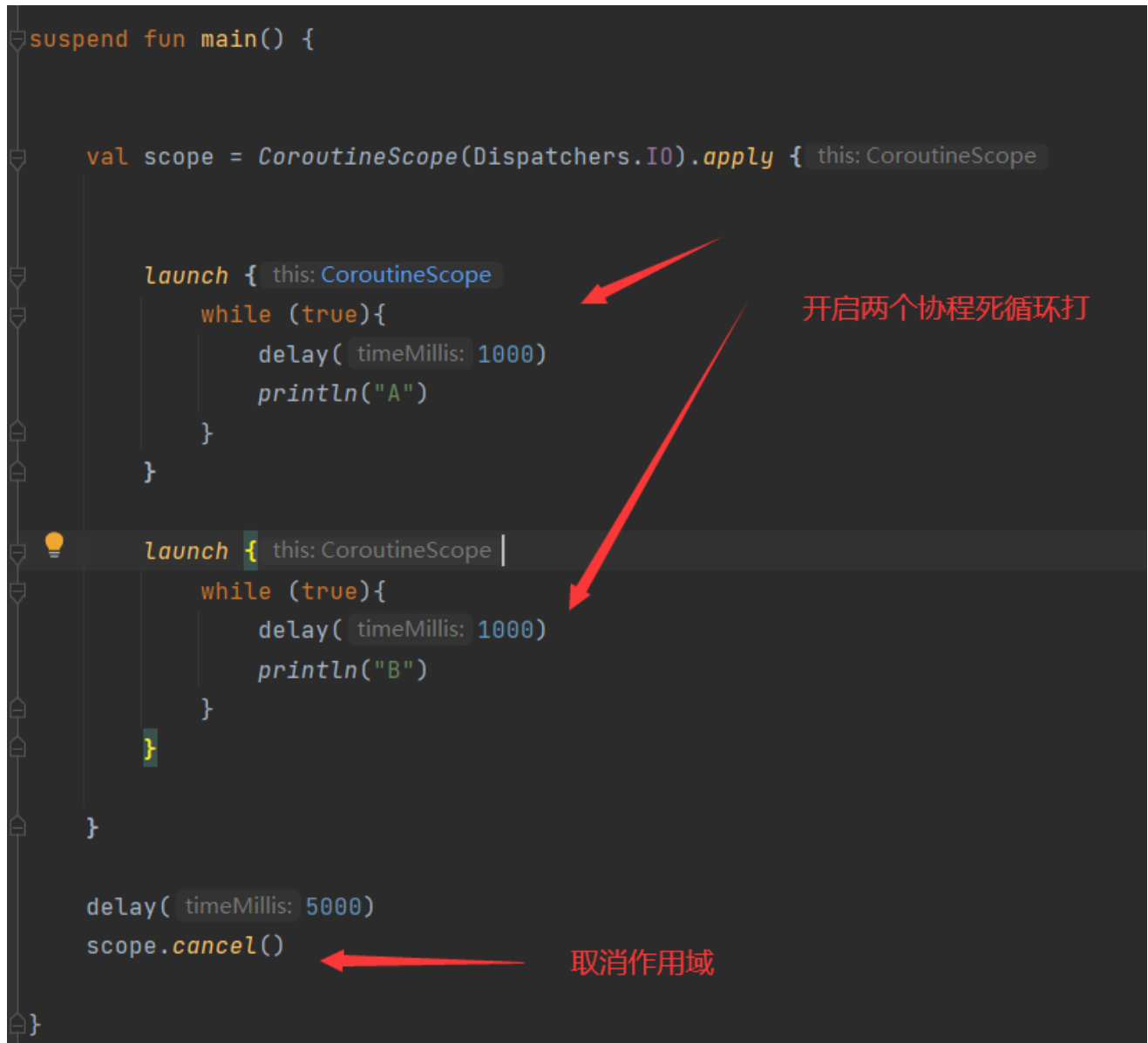
源码分析就略过了，如果前面`StateFlow`的源码有分析过的会发现`SharedFlow`和`StateFlow`是相似的。

cancellation

kotlin 协程是结构化并发的，这里的结构化最大的体现就是父子协程间的关系。以及 CoroutineScope 这个概念。

协程取消有几个明显的规律。

- 取消作用域会取消它的所有子协程



cancel会取消所有的子协程执行。

- 被取消的子协程不会影响兄弟协程

```

suspend fun main():Unit = coroutineScope { this: CoroutineScope

    val job1 = launch { this: CoroutineScope
        while (true){
            delay( timeMillis: 1000)
            println("A")
        }
    }

    val job2 = launch { this: CoroutineScope
        while (true){
            delay( timeMillis: 1000)
            println("B")
        }
    }

    delay( timeMillis: 1000)
    job1.cancel()
}

```

开启两个协程

delay后取消兄弟协程

- 协程取消是通过抛出一个CancellationException完成的。

```

suspend fun main() :Unit = runBlocking { this:
    val job = launch { this: CoroutineScope
        try {
            delay( timeMillis: 100000000)
        }catch (e:Exception){
            e.printStackTrace()
        }
    }

    delay( timeMillis: 1000)
    job.cancel()
}

```

注意事项

协程的取消分为两类

- cpu密集型
- io密集型

我们很多的协程之所以能够被取消是因为官方对我们的取消做了兼容处理的。所以调用cancel以后很顺其自然的我们就取消成功了。但总是有特殊的场景我们是无法取消的。比如。

```
fun main():Unit = runBlocking {  
  
    val job = launch {  
        while (true) {  
            println("A")  
            Thread.sleep(1000)  
        }  
    }  
  
    delay(1000)  
    job.cancelAndJoin()  
  
}
```

坏起来了，取消失效了。

主要源于协程的取消是需要一些特殊条件的，我们前面的协程之所以能被取消是官方做了适配。遇上官方适配的方式，也只能挠头。

不过要想取消它其实也不难。

有以下几种方法可以取消

1.每次循环拿job的执行状态，如果是活跃才继续执行



```
fun main():Unit = runBlocking { this: CoroutineScope  
    val job = launch(Dispatchers.IO){ this: CoroutineS  
        while (true && isActive) {  
            println("A")  
            Thread.sleep( millis: 1000)  
        }  
    }  
  
    delay( timeMillis: 1000)  
    job.cancelAndJoin()  
}
```

2.通过调用ensureAction

```

fun main():Unit = runBlocking { this: CoroutineScope
    val job = launch(Dispatchers.IO){ this: CoroutineScope
        while (true) {
            println("A")
            Thread.sleep( millis: 1000)
            ensureActive()
        }
    }

    delay( timeMillis: 1000)
    job.cancelAndJoin()
}

```

3.通过yield进行让位

(本质是还是调用的ensureActive)

yield会让当前协程直接挂起，这很符合让位这一说法。

和线程的yield有些类似，但不完全是。

```

fun main():Unit = runBlocking { this: CoroutineScope
    val job = launch(Dispatchers.IO){ this: CoroutineScope
        while (true) {
            println("A")
            Thread.sleep( millis: 1000)
            yield()
        }
    }

    delay( timeMillis: 1000)
    job.cancelAndJoin()
}

```

异常处理

异常传递规则

- 当一个协程由于一个异常而运行失败的时候，它会传递这个异常并传递给它的父级。接着它的父级会进行如下的几步操作。
 - 取消它的子协程
 - 取消它自己
 - 将异常传播给它的父级
- 对于顶级协程，`async`的异常会被自动`catch`，但是`launch`则不会。

```
suspend fun main() {  
  
    val scope = CoroutineScope(Dispatchers.IO)  
  
    val job1 = scope.launch { this: CoroutineScope  
        throw Exception("A")  
    }  
  
    job1.join()  
}
```

结果是carsh。

```
suspend fun main() {  
  
    val scope = CoroutineScope(Dispatchers.IO)  
  
    val job1 = scope.async { this: CoroutineScope  
        throw Exception("A")  
    }  
  
    job1.join()  
}
```

结果是空白，什么都没有。

- 非根协程的异常会直接抛出给父协程。

```

suspend fun main():Unit= coroutineScope { this: Co

    launch { this: CoroutineScope

        val defferJob = async { this: CoroutineScope

            delay( timeMillis: 1000)

            throw IllegalStateException("emm")

            1 ^async

        }

    }

}

```

job & supervisorJob

kotlin协程异常传播都会由子协程往父协程去传递，这点没什么问题。很符合我们的感官。

but向父协程传播异常后就出大问题了。他把父协程给取消了。也就是说任意一个异常的触发都可能会导致整个作用域的取消。（这很不好

不过好在上述情况只是默认的，也就是说我们可以改。

接下来就得隆重介绍job和supervisorJob

- job是默认的传播策略，出现异常以后，向父协程传播并取消当前协程，然后父协程也会被取消并向上传播。
- supervisorJob则将它取消策略进行了修改，也就是说异常出现了，继续往上传播，但是只取消当前协程，父协程，父父协程....能拿到异常，但是他们不会被取消了。

```

public fun SupervisorJob(parent: Job? = null) : CompletableJob =
    SupervisorJobImpl(parent)

```

```

public fun Job(parent: Job? = null): CompletableJob = JobImpl(parent)

```

```

private class SupervisorJobImpl(parent: Job?) : JobImpl(parent) {
    override fun childCancelled(cause: Throwable): Boolean = false
}

```

CoroutineExceptionHandler

CoroutineExceptionHandler用于捕获异常，通常情况下是放在根协程下的，他会捕获上报的所有异常。

```
suspend fun main(): Unit {  
  
    val scope = CoroutineScope(Dispatchers.IO)  
    scope.launch(CoroutineExceptionHandler { coroutineContext, throwable -> println("catch: $throwable") }) { this: CoroutineScope  
        launch { this: CoroutineScope  
            throw Exception("AA")  
        }  
    }  
  
    delay( timeMillis: 3000)  
}
```

总结

我们到目前为止学习了

- 1.基础概念
- 2.kt基础层以及实现
- 3.kt框架层以及实现

