关于

本篇文章是记录的在协程源码分析的内容合集

Suspend function 1 ——Source

> *Time*： *2022-1-14*——利用*Kotlin ByteCode*插件对挂起函数进行分析。（这个插件不是很好用。）

概述

> 挂起函数初探

挂起函数是什么?

挂起函数能干什么?

那些是挂起函数?

挂起函数怎么实现的?

挂起函数是什么

挂起函数本质还是函数，只不过这个函数在经过Kotlin编译器以后会进行一些特殊的处理。以达到用同步的代码写出异步的操作逻辑。

挂起函数能干什么

就简单而言，挂起函数具有以下两种功能。

——挂起，恢复。

由于这两种功能使得挂起函数具备了——使用同步的写法，达到了异步的逻辑。

哪些是挂起函数

我觉得标题改为 怎么定义，使用挂起函数。会更加贴合实际。hh

关于什么是挂起函数很简单。

这是挂起函数的定义，和普通函数的定义其实是一致的。

```
suspend fun suspendFunc1(){

}
```

除此之外还有一种奇怪的定义方式。（其实也不是很奇怪啦，上面是挂起函数的‘常量’定义方式，下面的就是对应的挂起函数的‘变量’定义模式）

```
suspend {

}
```

这种其实是利用的内联函数内联一个suspend{}

```
public inline fun <R> suspend(noinline block: suspend () -> R): suspend
() -> R = block
```

关于挂起函数的使用嘛，很简单。

只能在一个挂起函数或者协程作用域调用。

- 在挂起函数中调用另一个挂起函数。

```
suspend fun main() {

    suspend {
        delay(1000)
        println("Task finished")
    }()

}
```

- 在协程作用域调用另一个挂起函数。

```
fun main() = runBlocking {
    suspend{
       delay(1000)
       println("Task finished")
    }()
}
```

挂起函数的实现

pre-leading

我们先捋一捋，挂起函数是消除了麻烦的回调对吧。

比如我有一个任务是这样的A->B->C其中ABC都是耗时操作。

如果使用非阻塞的方式就是这样的。（也就是使用回调的方式）。

```
fun main() {
    taskA{
        taskB{
            taskC{
                println("好了这样完成了A->B->C的异步任务")
            }
        }
    }
}
fun interface TaskABack {
    fun taskAFinished()
}

fun interface TaskBBack {
    fun taskBFinished()
}
```

```
fun interface TaskCBack {
    fun taskCFinished()
}

fun taskA(taskABack: TaskABack) {
    println("taskA开始")
    Thread.sleep(1000)
    println("taskA结束了")
    taskABack.taskAFinished()
}

fun taskB(taskBBack: TaskBBack) {
    println("taskB开始")
    Thread.sleep(1000)
    println("taskB结束了")
    taskBBack.taskBFinished()
}

fun taskC(taskCBack: TaskCBack) {
    println("taskC开始")
    Thread.sleep(1000)
    println("taskC结束了")
    taskCBack.taskCFinished()
}
```

由于异步逻辑的嵌套所以有了RxJava

```
package whycoroutine

import io.reactivex.rxjava3.core.Observable

/**
 *@author ZhiQiang Tu
 *@time 2022/1/13  23:43
 *@signature 我将追寻并获取我想要的答案
 */
fun main() {
    Observable.create<Unit> {
        it.onNext(Unit)
    }.flatMap {
        Observable.create<Unit> { it.onNext(taskA {}) }
    }.flatMap {
        Observable.create<Unit> { it.onNext(taskB {}) }
    }.flatMap {
        Observable.create<Unit> {
            it.onNext(taskC {})
        }
    }.subscribe {
        println("好了这样完成了A->B->C的异步任务")
```

```
    }
}


fun interface TaskABack {
    fun taskAFinished()
}

fun interface TaskBBack {
    fun taskBFinished()
}

fun interface TaskCBack {
    fun taskCFinished()
}

fun taskA(taskABack: TaskABack) {
    println("taskA开始")
    Thread.sleep(1000)
    println("taskA结束了")
    taskABack.taskAFinished()
}

fun taskB(taskBBack: TaskBBack) {
    println("taskB开始")
    Thread.sleep(1000)
    println("taskB结束了")
    taskBBack.taskBFinished()
}

fun taskC(taskCBack: TaskCBack) {
    println("taskC开始")
    Thread.sleep(1000)
    println("taskC结束了")
    taskCBack.taskCFinished()
}
```

可以发现RxJava消除了这种回调，

but，他的实现其实是不太简洁的。

为什么说，我们可以发现这种实现方式除了没嵌套了，其实他代码还是挺多的。当然这也不是批判这种简化回调的方式不行，只是不够完美而已。

但这也是没办法的事情，一个框架能做的这样已经很完美了。

总结一句话是复杂性没有消除，甚至相比于回调的复杂性过之，但是消除了回调更加清晰了。

为什么RxJava他没能更简洁？因为你会发现他只是把回调给‘铺平了’（之前的回调是嵌套的，他将这种嵌套的层级关系利用建造者模式给展开了）。

我们看看上面的回调是不是模板化的代码？好像是吧？没有什么与逻辑相关的代码对吧？

有没有一种可能？我们可以利用编译器帮我们生成回调？从而达到简化的效果？

content

前面从为什么会出现异步逻辑不简洁的的的原因进行了分析，因为有万恶的回调。RxJava虽然看似消除了回调，然而它只是把回调换了一个位置，所以只是更清晰了，其实就写的代码而言其实还变多了。

然鹅协程的核心——挂起函数

就不是这么玩的，这丫直接不按套路出牌它利用编译器生成了一系列回调。这样实现了既简洁又清晰的效果。

不信嘛？

按道理suspend main会被多次挂起和恢复对吧。

一次挂起和恢复就像一次回调。

```
package whycoroutine

import kotlinx.coroutines.delay

/**
 *@author ZhiQiang Tu
 *@time 2022/1/14  10:58
 *@signature 我将追寻并获取我想要的答案
 */
suspend fun main() {
    testFun1()
    testFun1()
    testFun1()
    testFun1()
    testFun1()
    testFun1()
    testFun1()
    testFun1()
    testFun1()
}
```

```
suspend fun testFun1() {
    delay(1000)
}
```

decompile一下看看

瞧瞧我发现了什么东西。

有兴趣的可以在suspend main里面多调用几次testFun1，调用个七八十次就更明显了，这回调嵌套想想都恐怖。

Continuation的初始化分析

分析一下main函数的执行。（只分析核心的挂起和恢复过程。）

刚进入的时候

```
Object $continuation;
label97: {
   if (var0 instanceof <undefinedtype>) {
      $continuation = (<undefinedtype>)var0;
      if ((((<undefinedtype>)$continuation).label & Integer.MIN_VALUE) ≠ 0) {
         ((<undefinedtype>)$continuation).label -= Integer.MIN_VALUE;
         break label97;
      }
   }

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

初始化continuation

这是Continuation

```
$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 SuspendCallbackKt.main((Continuation)this);
   }
};
```

很有意思好吧

- label这个intValue存放的是suspend main这个挂起函数执行到了那个地方（因为他也是个挂起函数嘛，要挂起，要恢复，不记录一些比较的执行状态可不行）。
- result就是返回值，suspend main是挂起函数，他会调用其他的挂起函数，会等到其他挂起函数恢复以后，它要拿到这个挂起函数的结果。亦或者是它本身也需要把结果给调用者。主要就是为了方便返回值的获取。
- invokeSuspend就更有意思了。你会发现invokeSuspend需要把result传入，啥意思，比如我suspend main 调用了一个挂起函数add进行1+1的数学运算（CPU密集型任务嘛，/狗头），add会先挂起suspend main，然后等计算完成以后，就会去调用invokeSuspend，使得suspend main恢复执行，这个$result就是add的执行结果2。

然后分析一下函数体，先进行了结果的存储，然后欸label与Integer.MIN_VALUE（最高位为1，31个0）或了一下。也就是说label变为了Integer.MIN_VALUE+label，最后一句就有意思了，把this传入，然后调用本方法。发现了嘛，恢复的本质其实是重新调用被挂起的方法。恢复的时候就会执行前面if的内容了。

```
label97: {
    if (var0 instanceof <undefinedtype>) {
        $continuation = (<undefinedtype>)var0;
        if ((((<undefinedtype>)$continuation).label & Integer.MIN_VALUE) != 0) {
            ((<undefinedtype>)$continuation).label -= Integer.MIN_VALUE;
            break label97;
        }
    }
```

这样就避免了重复new ContinuationImpl。

suspend function 函数体分析

函数体被编译器编译成了一个类似于洋葱的结构，一层套一层的标签

```
Object var3;
label108: {
    label109: {
        label110: {
            label111: {
                label80: {
                    label79: {
                        label78: {
                            Object $result = ((<undefinedtype>)$continuation).result;
                            var3 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
                            switch(((<undefinedtype>)$continuation).label) {...}

                            ((<undefinedtype>)$continuation).label = 2;
                            if (testFun1((Continuation)$continuation) == var3) {
```

好开始分析，

一开头先获取了result返回值，然后获取了CoroutineSingletons.COROUTINE_SUSPENDED这个枚举类的值（方便后续的挂起操作）。

然后进入了一个switch case的嵌套。

他的分支数为挂起点数+1，为啥?

因为suspend main可能会被恢复n+1次。（为甚是可能不是一定，因为挂起点不一定会挂起。），而每一次恢复所需要执行的代码就是一个case分支。由于continuation的label是基本数据类型，在new ContinuationImpl的时候默认会给他赋值为0.所以先会走case 0.

```
case 0:
   ResultKt.throwOnFailure($result);
   ((<undefinedtype>)$continuation).label = 1;
   if (testFun1((Continuation)$continuation) == var3) {
      return var3;
   }
   break;
```

```
internal fun Result<*>.throwOnFailure() {
    if (value is Result.Failure) throw value.exception
}
```

- 第一行通过调用throwOnFailure来判断返回值的合理性。如果是throwable就给他抛出来。
- 第二行更新了label，label向前 更新了一个长度。然后调用了第一个挂起点的内容，判断他的返回值是不是CoroutineSingletons.COROUTINE_SUSPENDED，如果是就返回一个CoroutineSingletons.COROUTINE_SUSPENDED。发现问题了没有
    - 在执行的时候判断是否有没有被挂起其实就是判断返回值是不是对应的枚举类。
    - 而且一个挂起函数在调用另一个挂起函数的时候都会包裹上一层***if***判断返回值是不是***suspend***，如果是就直接***return suspend***，也就是说挂起的真正含义就是直接***return***，不执行了。
    - 还有就是挂起会从最内层开始传递，一直到最外层的挂起函数。

  除此之外还有当没有被挂起的时候会直接执行下一个挂起点的内容，先更新label，调用挂起点。如果还没挂起就调用下下个挂起点，依次类推，如果被挂起了，那就直接返回，等到下次恢复的时候再执行那三套操作（检验值，更新label，调用挂起函数）。

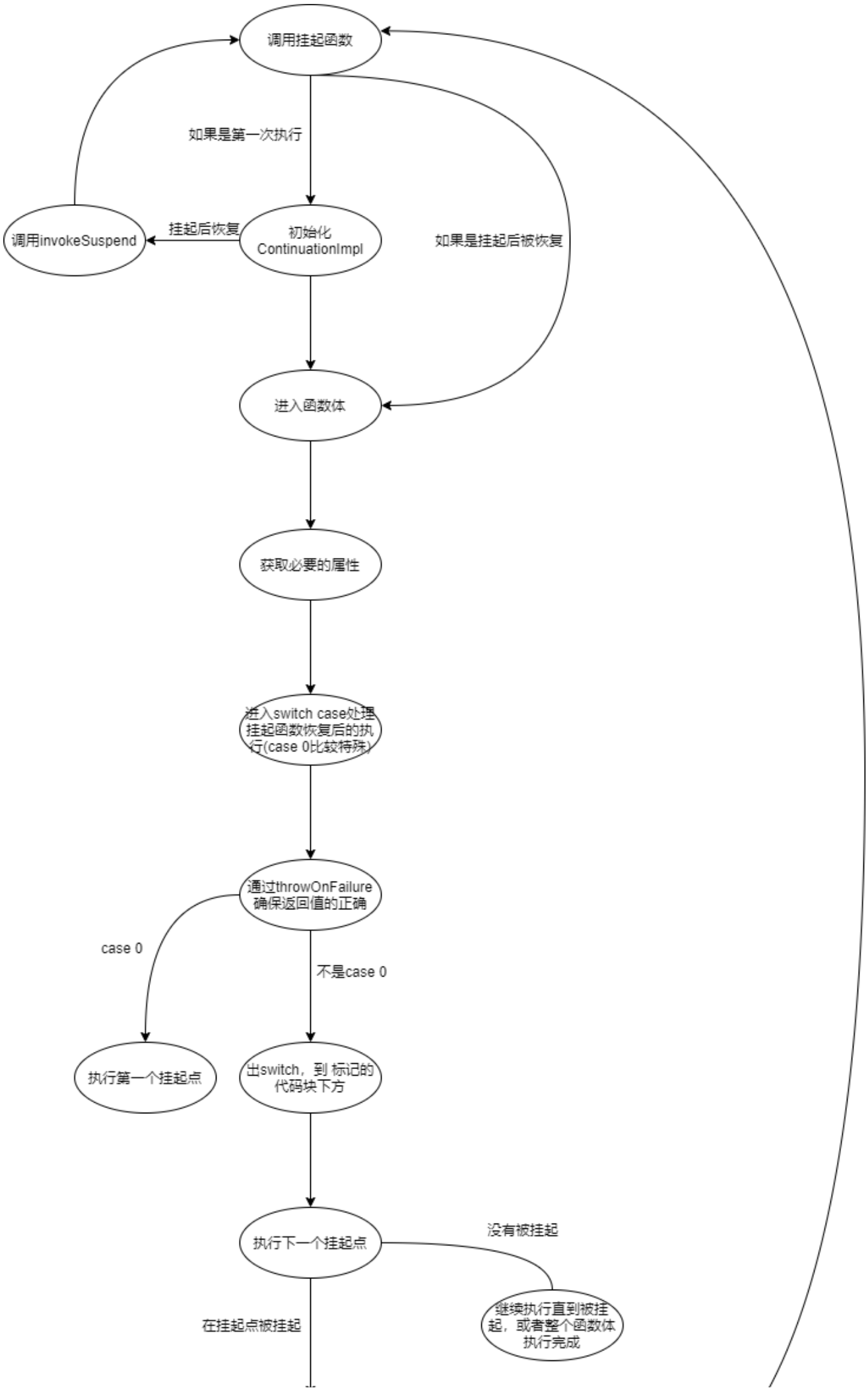
  函数体生成的伪代码如下

```
import kotlin.Unit;
import kotlin.coroutines.intrinsics.CoroutineSingletons;
import kotlin.coroutines.intrinsics.IntrinsicsKt;

public class SuspendBody {
    public static void main(String[] args) {
        label1:
        {
            label2:
            {
                label3:
                {
                    label4:
                    {
                        switch (continuation.label) {
```

```
                            Object result = continuation.result; //获取
continuation里面的返回值
                            Object suspendLabel =
IntrinsicsKt.getCOROUTINE_SUSPENDED();//获取挂起标签
                            case 0:
                                ResultKt.throwOnFailure(result);
                                continuation.label = 1;//更新label
                                if (suspendFun01(continuation) ==
suspendLabel) return;//如果在第一个挂起点被挂起了就直接返回
                                break;
                            case 1:
                                ResultKt.throwOnFailure(result);
                                break;
                            case 2:
                                ResultKt.throwOnFailure(result);
                                break label4;
                            case 3:
                                ResultKt.throwOnFailure(result);
                                break label3;
                            case 4:
                                ResultKt.throwOnFailure(result);
                                break label2;
                            case 5:
                                ResultKt.throwOnFailure(result);
                                return Unit.INSTANCE;
                            default:
                                throw new IllegalStateException("call to
'resume' before 'invoke' with coroutine");
                                break ;
                        }
                        //如果没有被挂起继续执行
                        continuation.label = 2;
                        if (suspendFun02(continuation) == suspendLabel)
return;//如果在第一个挂起点被挂起了就直接返回
                    }
                    continuation.label = 3;
                    if (suspendFun03(continuation) == suspendLabel)
return;
                }
                continuation.label = 4;
                if (suspendFun04(continuation) == suspendLabel) return;
            }
            continuation.label = 5;
            if (suspendFun05(continuation) == suspendLabel) return;
        }
        continuation.label = 6;
        if (suspendFun06(continuation) == suspendLabel) return;

    }
```

```
    }
```

流程图如下


调用挂起函数
如果是第一次执行
调用invokeSuspend
挂起后恢复
初始化ContinuationImpl
如果是挂起后被恢复
进入函数体
获取必要的属性
进入switch case处理挂起函数恢复后的执行(case 0比较特殊)
通过throwOnFailure确保返回值的正确
case 0
不是case 0
执行第一个挂起点
出switch，到 标记的代码块下方
执行下一个挂起点
没有被挂起
在挂起点被挂起
继续执行直到被挂起，或者整个函数体执行完成

Suspend function 2 ——Source

> *Time*: *2022 -1-15*

前面讲了比较常见的挂起函数，除此之外还有一种比较另类的挂起函数。

那就是lambda的形式，

```
fun main() {

    var continuation = object : Continuation<Unit>{
        override val context: CoroutineContext
            get() = EmptyCoroutineContext

        override fun resumeWith(result: Result<Unit>) {
            println("resumed")
        }
    }

    suspend {
        println("before delay")
        delay(1000)
        println("after delay")
        Unit
    }.startCoroutine(continuation)
    Thread.sleep(10000)

}
```

分析流程第一步是一个关键字suspend。

but，点进去会发现其实就是一个内联函数。

```
public inline fun <R> suspend(noinline block: suspend () -> R): suspend
() -> R = block
```

然后调用了suspend()的扩展函数

```
public fun <T> (suspend () -> T).startCoroutine(
    completion: Continuation<T>
) {
    createCoroutineUnintercepted(completion).intercepted().resume(Unit)
}
```

这个扩展函数需要传入一个continuation，这里传进去了一个continuation。然后立即拦截，最后直接resume。resume以后就进入了BaseContinuation的resumeWith方法，然后经由一个死循环最后invokeSuspend，恢复。

```
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
            }
        }
    }
}
```

这段代码的死循环很有意思。不过之后会分析它的原理现在只是摆一下，不打算细说。（透露一下这段代码是挂起函数恢复的核心逻辑。

挂起函数的核心原理

> 前面讲述了两种挂起函数会发现其实是一样的，底层都是创建一个挂起函数然后再调用。但是挂起函数有什么用？简化回调，异步操作同步写法。很爽，但是他是怎么实现的呢？

关于挂起函数的回调

我们知道在创建挂起函数会自动生成一个ContinuationImp,ContinuationImp是BaseContinuation的一个实现类，ContinuationImp在创建的时候需要一个Continuation，而这个传入的Continuation就是为了当挂起函数完毕后调用，比如我suspend a()调用了suspend b，a就会把自己的continuation传给b，然后b在创建ContinuationImp的时候就会把a传入的continuation传入到构造函数。这样但b执行完毕（注意是执行完毕，不是恢复也不是挂起）就可以通过调用invokeSuspend来恢复a的执行。发现了嘛，挂起函数其实是通过编译器生成Continuation来简化回调的，但是它并没有完全去除回调，continuation的一层层持有关系其实就是回调，只是自动生成了回调。

关于挂起函数的挂起

前面稍微提了一下的，挂起的实现很是简单，简单到有些突兀。

每次在挂起函数内部调用另外一个挂起函数的时候你会发现他都会生成一段代码.比如我在suspend a()调用了suspend b()那么调用处是这样的(伪代码，但是效果是等价的)

```
if(b(continuation) == CoroutineSingletons.COROUTINE_SUSPENDED)return;
```

continuation就不说了，调用挂起函数b判断返回值是不是CoroutineSingletons.COROUTINE_SUSPENDED，这是在干嘛?

这是在判断是否被挂起，而那个枚举类就是挂起的标准，如果一个函数想挂起那好很简单你返回值返回COROUTINE_SUSPENDED就行了（严格来说这样是不行的，因为CoroutineSingletons是internal所以得利用官方支持的方法来达到让函数返回这个枚举类）。

所以什么是挂起？很清晰就上面的一行代码，如果函数返回枚举，那就表明它被挂起了，然后return。不执行了。这就是协程的挂起的本质。

关于挂起函数的恢复

或许你在想都return了怎么恢复？还能怎么恢复。当然是重新调用了，这在前面的continuation的invokeSuspend已经分析过了。

continuation不仅是建立了回调关系，除此之外还存储了程序执行的上下文，也就是执行位置。就只是一个变量，label，一个int值，它通过一个switch case依据不同的label值将所有的挂起点都分配到一个独立的分支，每执行一个挂起点前都更新一下label，这样达到了恢复执行的目的。妙不妙，简直妙极了。

suspend lambda

> *Time: 2022-1-24*

测试代码

```
suspend fun main() {
    a {
        println("before delay")
        delay(1000)
        println("after delay")
    }
}

suspend fun a(block: suspend () -> Unit) {
    block()
}
```

反编译

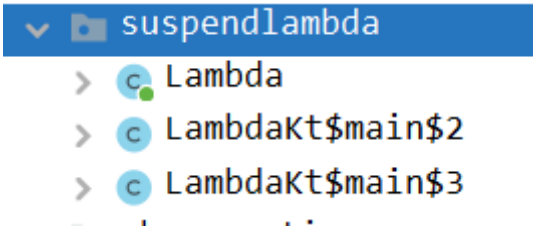
你如果直接反编译suspend的Lambda表达是你会发现是这样的

```
Object var10000 = a((Function1)(new Function1((Continuation)null) {
   int label;

   @Nullable
   public final Object invokeSuspend(@NotNull Object $result) {
```

```
        Object var3 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
        String var2;
        switch(this.label) {
        case 0:
           ResultKt.throwOnFailure($result);
           var2 = "before delay";
           System.out.println(var2);
           this.label = 1;
           if (DelayKt.delay(1000L, this) == var3) {
              return var3;
           }
           break;
        case 1:
           ResultKt.throwOnFailure($result);
           break;
        default:
           throw new IllegalStateException("call to 'resume' before
'invoke' with coroutine");
        }

        var2 = "after delay";
        System.out.println(var2);
        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(Unit.IN
STANCE);
     }
}), $completion);
```

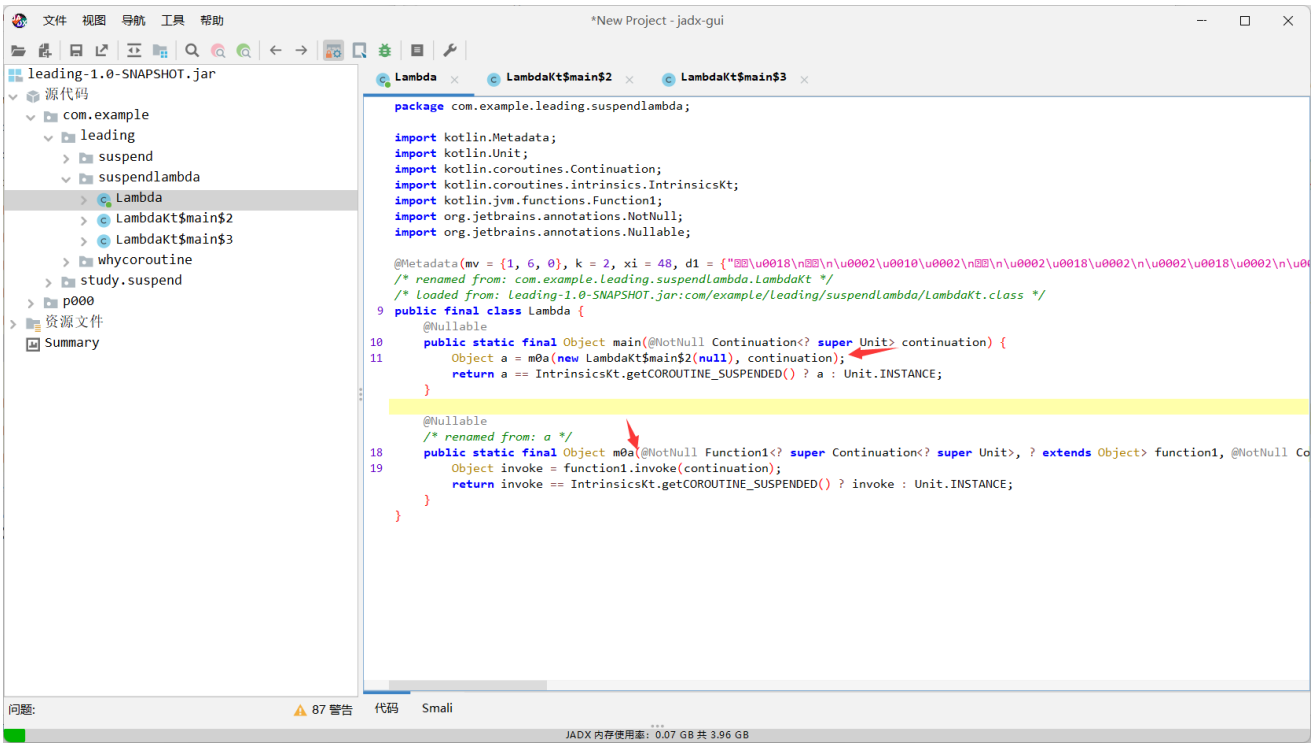
然而真实情况是这样嘛?

生成了3个类

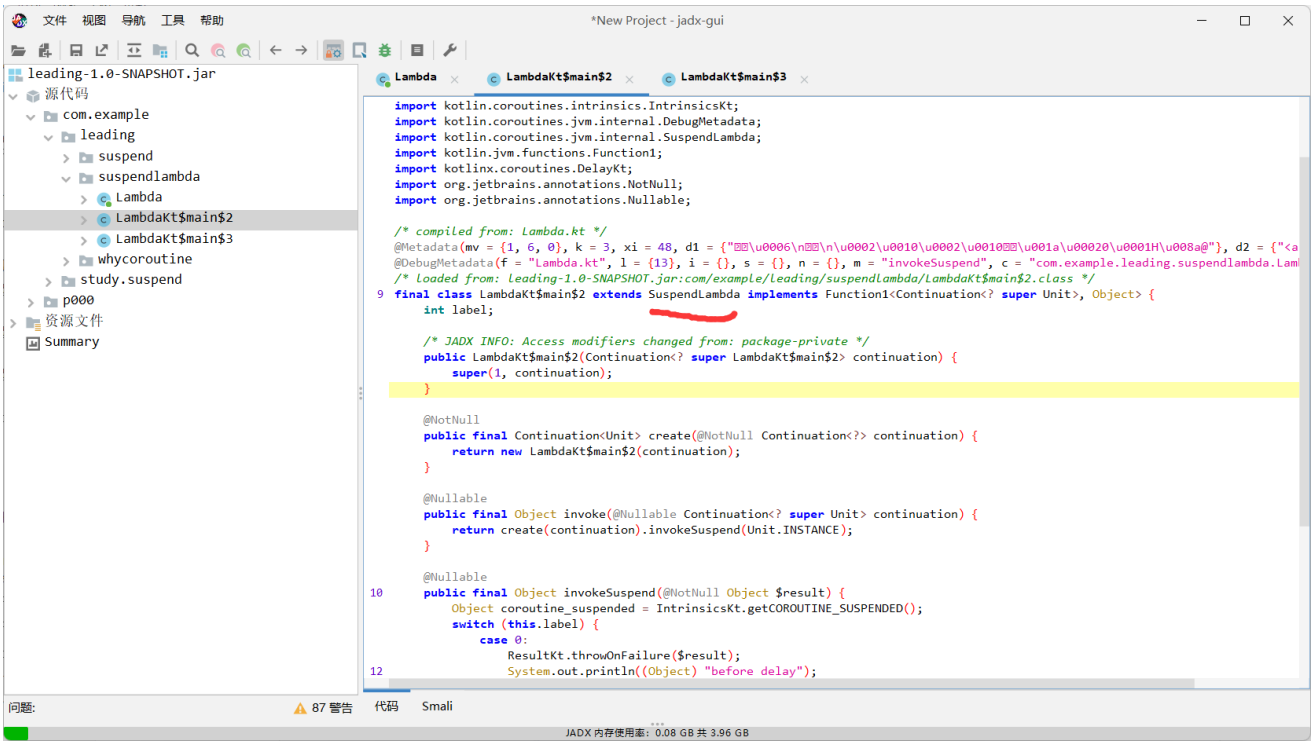


解析

suspend main中调用了挂起函数a，然后挂起函数根据对应的continuation，invoke了这个suspend lambda。



SuspendLambda，很奇怪的是挂起函数体被单独抽离出来了，变成了一个类，在invoke的时候调用create新的实例。



```
@NotNull
    public final Continuation<Unit> create(@NotNull Continuation<?>
continuation) {
        return new LambdaKt$main$2(continuation);
    }

    @Nullable
    public final Object invoke(@Nullable Continuation<? super Unit>
continuation) {
        return create(continuation).invokeSuspend(Unit.INSTANCE);
    }
```

变化案例

suspend ()->Unit会被编译成SuspendLambda，那么如果多加几个参数呢？又或者加receiver呢？

其实还是一样的，只是实现的Fuction接口不再是Function1了而已。

这里不展示了，有兴趣自己build 一个jar包jadx反编译一下即可。

总结

suspend lambda都会在编译后实现SuspendLambda接口。

suspend

> *Time 2022-1-15*

前面只是讲了挂起是怎么实现的，但是我们作为开发者怎么挂起一个函数呢？

这就得用到suspendCoroutine{}了

这个函数是用来获取当前协程的上下文的。

```
public suspend inline fun <T> suspendCoroutine(crossinline block:
(Continuation<T>) -> Unit): T {
    contract { callsInPlace(block, InvocationKind.EXACTLY_ONCE) }
    return suspendCoroutineUninterceptedOrReturn { c: Continuation<T> ->
        val safe = SafeContinuation(c.intercepted())
        block(safe)
        safe.getOrThrow()
    }
}
```

过程不长，首先拦截一下continuation，然后把continuation用SafeContinuation包装一下，然后调用传入的lambda，然后调用了SafeContinuation的getOrThrow

```
internal actual fun getOrThrow(): Any? {
    var result = this.result // atomic read
    if (result === UNDECIDED) {
        if (RESULT.compareAndSet(this, UNDECIDED, COROUTINE_SUSPENDED))
return COROUTINE_SUSPENDED
        result = this.result // reread volatile var
    }
    return when {
        result === RESUMED -> COROUTINE_SUSPENDED // already called
continuation, indicate COROUTINE_SUSPENDED upstream
        result is Result.Failure -> throw result.exception
        else -> result // either COROUTINE_SUSPENDED or data
    }
}
```

代码也不长，逻辑也简单，如果当前协程的状态为UNDECIDED（未定状态），就给内部的result设置为suspend，并且返回suspend。机会来了不是?

but你也发现了他是先执行的block。ok好的。我们来进行第一个尝试。

```
suspend fun main() {
    useSuspend1()
}

suspend fun useSuspend1() = suspendCoroutine<Unit> {

}
```

发生了什么，被挂起了，而且一直挂着的。程序一直在等待着恢复

那怎么恢复？猜都能猜到利用continuation

```
suspend fun useSuspend1() = suspendCoroutine<Unit> {
    it.resume(Unit)
}
```

你会发现程序直接结束了。如果我们点开分析一下resume，你又会发现

```
public inline fun <T> Continuation<T>.resume(value: T): Unit =
    resumeWith(Result.success(value))
```

这丫只是调用了一下resumeWith，那找找SafeContinuation的resumeWith干了啥

```
public actual override fun resumeWith(result: Result<T>) {
    while (true) { // lock-free loop
        val cur = this.result // atomic read
        when {
            cur === UNDECIDED -> if (RESULT.compareAndSet(this,
UNDECIDED, result.value)) return
            cur === COROUTINE_SUSPENDED -> if
(RESULT.compareAndSet(this, COROUTINE_SUSPENDED, RESUMED)) {
                delegate.resumeWith(result)
                return
            }
            else -> throw IllegalStateException("Already resumed")
        }
    }
}
```

如果是UNDECIDED就把value给set进RESULT中，如果是挂起就给设置为resume状态。并调用所包裹的continuation的resumeWith。否则就是死循环抛异常。

出现问题了，在block里面修改了RESULT的值，使得在getOrThorw拿不到UNDECIDED的值，无法返回挂起，换句话说挂起不了了。

换个思路想想，只要让getOrThrow早于resume执行就行了吧?

那这样?

```
suspend fun main() {
    useSuspend1()
}

suspend fun useSuspend1() = suspendCoroutine<Unit> {
    thread {
        (0..10).forEach{it1->
            Thread.sleep(1000)
            println("休眠了${it1}秒")
        }
        it.resume(Unit)
    }
}
```

成功了因为我开启了一个线程在10秒后返回，这样resume就在getOrThrow之后执行了。

resumeWith——Source

> *Time 2022-1-15*

前面其实已经贴了一段，注意这里分析的是BaseContinuationImp的实现(这个实现是协程恢复的核心实现)

```
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
            }
        }
    }
}
```

首先先this作为current，然后拿取result作为param。

进入一个死循环，然后进入with，然后立即调用this.invokeSuspend来获取Result<>如果挂起了直接返回，如果没有就返回执行结果，出错就返回对应的Exception，然后再判断completion时候不BaseContinuationImp(completion就是每次new ContinuationImp传入的Continuation)如果是就将completion作为current，将输出结果作为param，继续上面的循环，否则就调用completion的resumeWith。

你可能会疑惑为啥BaseContinuation需要死循环去执行，为啥，不为啥，因为BaseContinuation地构造里面需要一个Continuation，相应地只要是它的子类就会组合一个Continuation，也就是说BaseContinuation就行是一条卷心菜一样（我虽然卷但是还是菜）一层包一层，所以需要死循环去执行（invokeSuspend）。

所以挂起是一层层向外传入一个枚举类，恢复则是由内向外调用resumeWith(其实说invokeSuspend更合理)，而invokeSuspend又调用了挂起函数本身，挂起函数内部做了状态的保持，这样也就能正常恢复执行。喵不可言。

由此挂起函数地本质已经完毕了，神奇的挂起和恢复特性也被扒了皮。

suspend fun main——Source

> *Time 2022-1-16*

来分析一下代码，就几行

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

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

初步分析

反编译一下。发现什么?

编译了挺多东西的

```java
public final class MainKt {
   @Nullable
   public static final Object main(@NotNull Continuation $completion) {...}

   // $FF: synthetic method
   public static void main(String[] var0) { RunSuspendKt.runSuspend(new MainKt$$$main(var0)); }

   @Nullable
   public static final Object a(@NotNull Continuation $completion) {...}
}
```

- main(Continuation completion)应该是挂起的main，也就是我们代码里面的suspend main
- main(String[] var0)应该是生成的一个真正的main函数。不过嘛这个main被截胡了。
- a就是对应的挂起函数来了

原理分析

RunSuspendKt.runSuspend还点不开是吧。

搜索一下

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

好啊，就三行。嗯。

先创建RunSuspend

然后把传入的suspend lambda开启

然后main线程等待执行完毕。

```
private class RunSuspend : Continuation<Unit> {
    override val context: CoroutineContext
        get() = EmptyCoroutineContext

    var result: Result<Unit>? = null

    override fun resumeWith(result: Result<Unit>) = synchronized(this) {
        this.result = result
        @Suppress("PLATFORM_CLASS_MAPPED_TO_KOTLIN") (this as
Object).notifyAll()
    }
```

```
    fun await() = synchronized(this) {
        while (true) {
            when (val result = this.result) {
                null -> @Suppress("PLATFORM_CLASS_MAPPED_TO_KOTLIN")
(this as Object).wait()
                else -> {
                    result.getOrThrow() // throw up failure
                    return
                }
            }
        }
    }
}
```

startCoroutine

```
public fun <T> (suspend () -> T).startCoroutine(
    completion: Continuation<T>
) {
    createCoroutineUnintercepted(completion).intercepted().resume(Unit)
}
```

内部分为三步，再创建一个协程，然后拦截，然后resume。

create

```
public actual fun <T> (suspend () -> T).createCoroutineUnintercepted(
    completion: Continuation<T>
): Continuation<Unit> {
    val probeCompletion = probeCoroutineCreated(completion)
    return if (this is BaseContinuationImpl)
        create(probeCompletion)
    else
        createCoroutineFromSuspendFunction(probeCompletion) {
            (this as Function1<Continuation<T>, Any?>).invoke(it)
        }
}
```

首先判断你这协程的Continuation是不是BaseContinuation，如果是就调用create，（这个create是编译器生成的）

如果想分析干了什么就自行去看

```
suspend{

}.startCoroutine(...)
```

其中suspend lambda生成的代码。

如果不是BaseContinuationImp，那就调用下面的方法在外面包裹一层。

```
private inline fun <T> createCoroutineFromSuspendFunction(
    completion: Continuation<T>,
    crossinline block: (Continuation<T>) -> Any?
): Continuation<Unit> {
    val context = completion.context
    // label == 0 when coroutine is not started yet (initially) or label
== 1 when it was
    return if (context === EmptyCoroutineContext)
        object : RestrictedContinuationImpl(completion as
Continuation<Any?>) {
            private var label = 0

            override fun invokeSuspend(result: Result<Any?>): Any? =
                when (label) {
                    0 -> {
                        label = 1
                        result.getOrThrow() // Rethrow exception if
trying to start with exception (will be caught by
BaseContinuationImpl.resumeWith
                        block(this) // run the block, may return or
suspend
                    }
                    1 -> {
                        label = 2
                        result.getOrThrow() // this is the result if the
block had suspended
                    }
                    else -> error("This coroutine had already
completed")
                }
        }
    else
        object : ContinuationImpl(completion as Continuation<Any?>,
context) {
            private var label = 0

            override fun invokeSuspend(result: Result<Any?>): Any? =
                when (label) {
                    0 -> {
                        label = 1
                        result.getOrThrow() // Rethrow exception if
trying to start with exception (will be caught by
BaseContinuationImpl.resumeWith
                        block(this) // run the block, may return or
suspend
```

```
                    }
                    1 -> {
                        label = 2
                        result.getOrThrow() // this is the result if the
block had suspended
                    }
                    else -> error("This coroutine had already
completed")
                }
            }
        }
```

这里的BaseContinuation还得分情况，如果上下文为空就是RestrictedContinuationImpl，不为空就是ContinuationImpl。

这样我们的RunSuspend这个Continuation就被包裹上了一层RestrictedContinuationImpl。

intercepted

这个就不讲了，就是把当前的协程Continuation也就是RestrictedContinuationImpl给拦截了，拿到对应的Continuation的实例

resume

resume调用了resumeWith，然后RestrictedContinuationImpl没有覆写父类的resumeWith，也就是说调用了BaseContinuationImpl的实现，这样很合理就调用了RestrictedContinuationImpl的invokeSuspend。

```
override fun invokeSuspend(result: Result<Any?>): Any? =
    when (label) {
        0 -> {
            label = 1
            result.getOrThrow() // Rethrow exception if trying to start
with exception (will be caught by BaseContinuationImpl.resumeWith
            block(this) // run the block, may return or suspend
        }
        1 -> {
            label = 2
            result.getOrThrow() // this is the result if the block had
suspended
        }
        else -> error("This coroutine had already completed")
    }
```

然后调用了lambda

```
createCoroutineFromSuspendFunction(probeCompletion) {
    (this as Function1<Continuation<T>, Any?>).invoke(it)
}
```

然后调用了RunSuspend传入的Lambda

```
final class MainKt$$$main extends Lambda implements Function1 {
   // $FF: synthetic field
   private final String[] args;

   // $FF: synthetic method
   MainKt$$$main(String[] var1) {
      super(1);
      this.args = var1;
   }

   // $FF: synthetic method
   public final Object invoke(Object var1) {
      return MainKt.main((Continuation)var1);
   }
}
```

这样就调用了suspend main，然后suspend main调用了一个delay，返回了一个挂起，然后挂起一层层传递到RestrictedContinuationImpl，等到恢复的时候，又调用RestrictedContinuationImpl的resumeWith，有调用invokeSuspend，最后调用了completion的resumeWith方法。RunSuspend的resumeWith很简单

```
override fun resumeWith(result: Result<Unit>) = synchronized(this) {
    this.result = result
    @Suppress("PLATFORM_CLASS_MAPPED_TO_KOTLIN") (this as
Object).notifyAll()
}
```

就是让把result赋值进去，然后notifyAll。

等等notify谁?

await

notify谁当然是main线程了。

前面的由于suspend main调用了delay被挂起了，所以挂起层层传递，一层层的return，最后跑到了RunSuspend.runSuspend的await。然后呢?

```
fun await() = synchronized(this) {
    while (true) {
        when (val result = this.result) {
            null -> @Suppress("PLATFORM_CLASS_MAPPED_TO_KOTLIN") (this
as Object).wait()
            else -> {
                result.getOrThrow() // throw up failure
                return
            }
        }
    }
}
```

然后就wait了。等待谁？等待RunSuspend叫醒它

前面notify了，所以它醒了，循环回来拿了result然后就return了。

这便是suspend main的整个执行流程。

总结

由于kotlin最早的时候是不支持协程的（不是像go语言层面支持协程，咱这协程是以库的形式加入的）所以suspend main并没有引入什么新的东西，只是偷天换日，给main函数传入了一个Continuation，使得它可以被挂起恢复（还是编译器生成的骚操作）。

delay——Source

> *Time 2022-1-17*

> *Delay*虽然看似简单，*but*它的内部实现其实还是稍微有点复杂的。因为*delay*的实现涉及到*ThreadExecutor*，也就是线程池。（只不过是单线程的线程池）。

```
suspend fun main() {
    println("before delay-->"+Thread.currentThread().name)
    delay(1000)
    println("after delay -->"+Thread.currentThread().name)
    println("before delay-->"+Thread.currentThread().name)
    delay(1000)
    println("after delay -->"+Thread.currentThread().name)
}
```

反编译一下，看看编译器没有干什么。

```
if (DelayKt.delay(1000L, (Continuation)$continuation) == var4) {
   return var4;
}
```

啥也没干

就简单的调用，没闹什么幺蛾子

流程分析

```
public suspend fun delay(timeMillis: Long) {
    if (timeMillis <= 0) return // don't delay
    return suspendCancellableCoroutine sc@ { 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)
        }
    }
}
```

先判断了一个timeMillis的合法性。

然后给当前挂起函数套一层Cancellable

```
public suspend inline fun <T> suspendCancellableCoroutine(
    crossinline block: (CancellableContinuation<T>) -> Unit
): T =
    suspendCoroutineUninterceptedOrReturn { uCont ->
        val cancellable =
CancellableContinuationImpl(uCont.intercepted(), resumeMode =
MODE_CANCELLABLE)
        /*
         * For non-atomic cancellation we setup parent-child
relationship immediately
         * in case when `block` blocks the current thread (e.g. Rx2 with
trampoline scheduler), but
         * properly supports cancellation.
         */
        cancellable.initCancellability()
        block(cancellable)
        cancellable.getResult()
    }
```

实现也不难，首先拦截一下当前的协程，然后new了一个CancellableContinuationImpl，设置了resumeMode传入了拦截到的当前的挂起函数的实例。

初始化取消能力

然后执行了传入的block，最后去调用获取执行的结果。返回。

block的代码也挺简单的。

```
if (timeMillis < Long.MAX_VALUE) {
            cont.context.delay.scheduleResumeAfterDelay(timeMillis,
cont)
        }
```

如果timeMillis合理那就调用Continuation的上下文的扩展delay

context是清一色的EmptyCoroutineContext

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

可见delay是获取不到了，因为EmptyCoroutineContext里面什么都没有，更别提Interceptor了，所以会返回一个DefaultDelay。

```
internal actual val DefaultDelay: Delay = DefaultExecutor
```

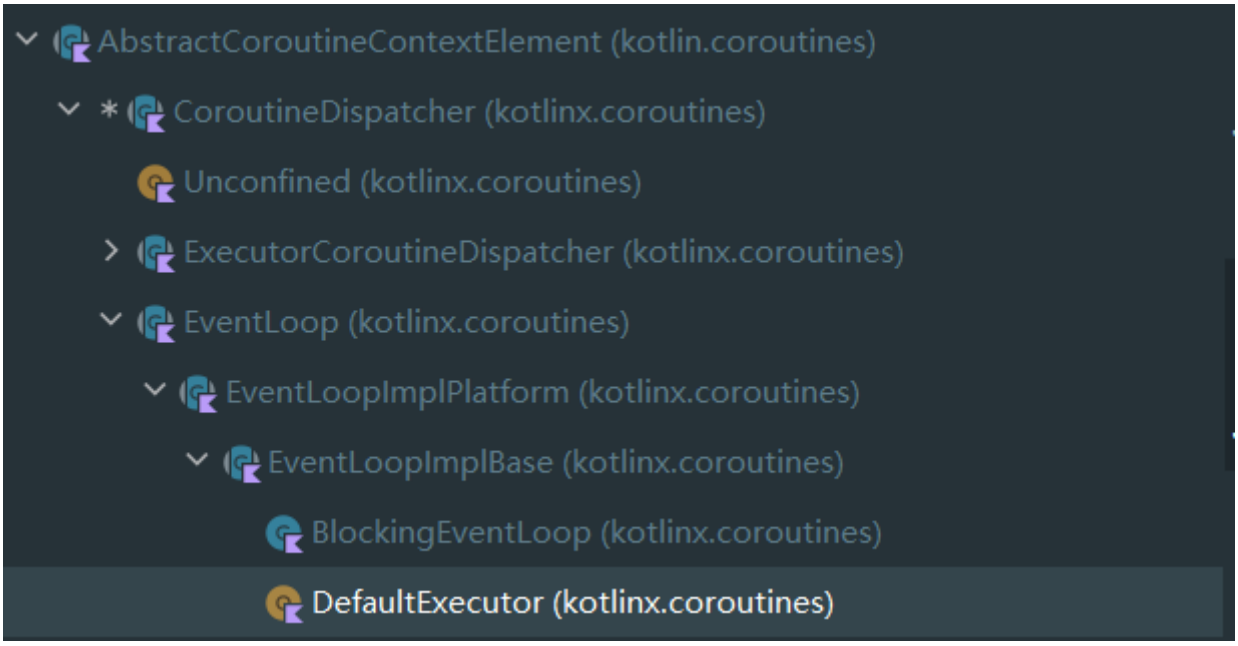
DefaultDelay是返回的一个DefaultExecutor

这个DefaultExecutor很有意思

乍一看是一个线程池，定睛一看还真是

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

这是DefaultExecutor的继承结构。他们都是CoroutineDispatcher



你说这怎们看出来他是一个线程池的？就这好像也看不出来的？

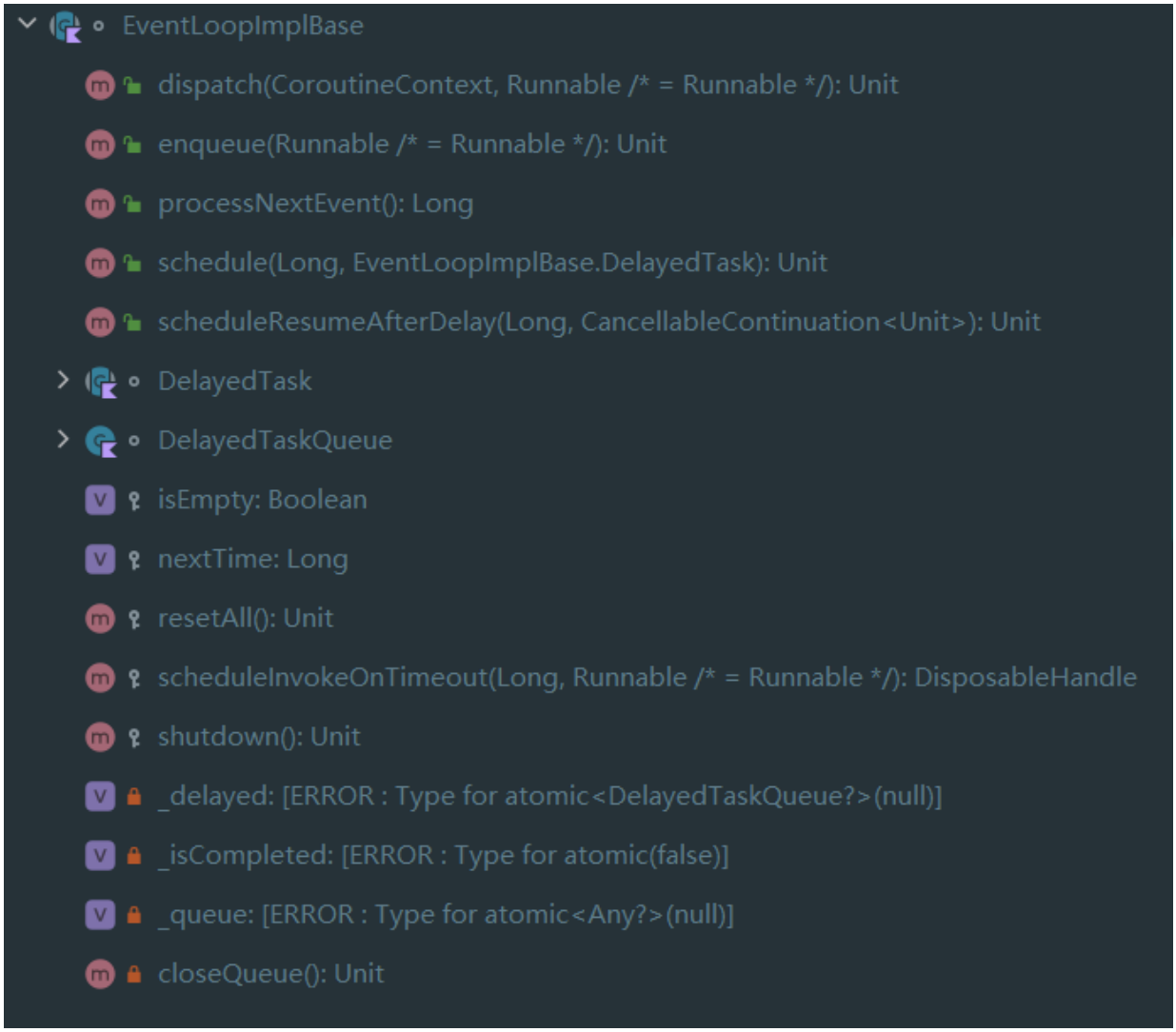
我猜的。没错我猜的。

- 第一个是DefaultExecutor的属性

```
override val thread: Thread
    get() = _thread ?: createThreadSync()
```

- 第二个是它的父类

```
EventLoopImplBase
```



前几个方法和线程池的方法很类似。除此之外它内部还提供了几个内部类DelayedTask,DelayedResumeTask,DelayedRunnableTask,DelayedTaskQueue我想这已经够明显了。这个和线程池的Task非常类似好把。

继续分析

挂起

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

调用了scheduleAfterDelay

```
public override fun scheduleResumeAfterDelay(timeMillis: Long,
continuation: CancellableContinuation<Unit>) {
    val timeNanos = delayToNanos(timeMillis)
    if (timeNanos < MAX_DELAY_NS) {
        val now = nanoTime()
        DelayedResumeTask(now + timeNanos, continuation).also { task ->
            continuation.disposeOnCancellation(task)
            schedule(now, task)
        }
    }
}
```

第一行将Delay的时间长转化为nano也就是毫微，1000的delay时长就是10的9次方

然后就是确保timeNanos没越界

```
private const val MAX_DELAY_NS = Long.MAX_VALUE / 2
```

具体为啥是Long.MAX_VALUE的1/2我也不知道（我只是个菜鸡）。可能是为了防止创建now+timeNanos越界。

now的话就是调用的System.nanoTime(),也就是比System.currentTimeMillis()更精确6个数量级

然后后它new了一个DelayedResumeTask，传入了开始执行的nanoTime，也就是现在的nanoTime+Delay的nanoTime，还传入了一个continuation，（现在知道为啥要包裹一层callable了吧，为了让这个即将放入线程池的任务具备取消的能力。）

```
DelayedResumeTask(now + timeNanos, continuation).also { task ->
    continuation.disposeOnCancellation(task)
    schedule(now, task)
}
```

创教了任务以后加了一个任务取消的回调，

然后开始规划这个任务，传入执行的nanoTime以及创建的任务。（这是要放入线程池的任务队列了。hh）

```
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")
    }
}

private fun scheduleImpl(now: Long, delayedTask: DelayedTask): Int {
```

```
        if (isCompleted) return SCHEDULE_COMPLETED
        val delayedQueue = _delayed.value ?: run {
            _delayed.compareAndSet(null, DelayedTaskQueue(now))
            _delayed.value!!
        }
        return delayedTask.scheduleTask(now, delayedQueue, this)
}
```

执行到了scheduleImpl，判断任务是否完成（肯定没完成啊），然后继续执行。

_delayvalue没有？那初始化，创建一个DelayedTaskQueue（任务算法放入队列里面了），然后去调用scheduleTask让这个创建的任务被执行。

```
@Synchronized
fun scheduleTask(now: Long, delayed: DelayedTaskQueue, eventLoop:
EventLoopImplBase): Int {
    if (_heap === DISPOSED_TASK) return SCHEDULE_DISPOSED
    delayed.addLastIf(this) { firstTask ->
        if (eventLoop.isCompleted) return SCHEDULE_COMPLETED
        if (firstTask == null) {
            delayed.timeNow = now
        } else {
            val firstTime = firstTask.nanoTime
            val minTime = if (firstTime - now >= 0) now else firstTime
            if (minTime - delayed.timeNow > 0) delayed.timeNow = minTime
        }
        if (nanoTime - delayed.timeNow < 0) nanoTime = delayed.timeNow
        true
    }
    return SCHEDULE_OK
}
```

代码不长，注释挺长的（注释我删了）。

如果任务已经被取消那就直接返回。

否则就添加任务（在DelayedTaskQueue里面添加）

最后返回添加任务完毕。然后一层层外传到when里面

```
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")
}
```

既然任务添加成功，那就开始执行了。在次之前进行了一层判断。

然后开始执行了

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

获取thread。判断如果不是当前thread那就开始执行，其实在获取thread的途中创建了一个线程，

而且这个线程还是守护线程。（知道为啥主程序执行完毕，整个执行程序就完成了嘛？知道为啥suspend main要加一个await让main线程等RunSuspend执行结束了嘛？）

```
override val thread: Thread
    get() = _thread ?: createThreadSync()

@Synchronized
private fun createThreadSync(): Thread {
        return _thread ?: Thread(this, THREAD_NAME).apply {
            _thread = this
            isDaemon = true
            start()
        }
}
```

创建后直接就开工了。这时候suspend main也就基本上挂起了，为啥，因为马上就要连续弹栈了，还记得最开始调用delay创建的cancellableContinuation嘛，传入的lambda就4行

```
public suspend inline fun <T> suspendCancellableCoroutine(
    crossinline block: (CancellableContinuation<T>) -> Unit
): T =
    suspendCoroutineUninterceptedOrReturn { uCont ->
        val cancellable =
CancellableContinuationImpl(uCont.intercepted(), resumeMode =
MODE_CANCELLABLE)
        cancellable.initCancellability()
        block(cancellable)
        cancellable.getResult()
    }
```

我们在调用block以后就一路执行到了任务入栈，接下来就得执行 getResult，（这段代码就不分析了）由于没有resume值所以这里会直接return一个SUSPEND的枚举类。

任务调度

我们会发现Runnable它传入的是this。

我们看看。

```
override fun run() {
    ThreadLocalEventLoop.setEventLoop(this)
    registerTimeLoopThread()
    try {
        var shutdownNanos = Long.MAX_VALUE
        if (!notifyStartup()) return
        while (true) {
            Thread.interrupted() // just reset interruption flag
            var parkNanos = processNextEvent()
            if (parkNanos == Long.MAX_VALUE) {
                // nothing to do, initialize shutdown timeout
                val now = nanoTime()
                if (shutdownNanos == Long.MAX_VALUE) shutdownNanos = now
+ KEEP_ALIVE_NANOS
                val tillShutdown = shutdownNanos - now
                if (tillShutdown <= 0) return // shut thread down
                parkNanos = parkNanos.coerceAtMost(tillShutdown)
            } else
                shutdownNanos = Long.MAX_VALUE
            if (parkNanos > 0) {
                // check if shutdown was requested and bail out in this
case
                if (isShutdownRequested) return
                parkNanos(this, parkNanos)
            }
        }
    } finally {
        _thread = null // this thread is dead
        acknowledgeShutdownIfNeeded()
        unregisterTimeLoopThread()
        // recheck if queues are empty after _thread reference was set
to null (!!!)
        if (!isEmpty) thread // recreate thread if it is needed
    }
}
```

- 设置eventLooper
- 注册eventLooperThread
- 设置关闭时间为Long.MAX_VALUE（也就是说这个线程基本上会一直执行）
- 判断一下是否由关闭线程池的请求
- 进入死循环

- 清理interrupted状态
- 去任务队列拿事件

```
override fun processNextEvent(): Long {
    // unconfined events take priority
    if (processUnconfinedEvent()) return 0
    // queue all delayed tasks that are due to be executed
    val delayed = _delayed.value
    if (delayed != null && !delayed.isEmpty) {
        val now = nanoTime()
        while (true) {
            // make sure that moving from delayed to queue removes
from delayed only after it is added to queue
            // to make sure that 'isEmpty' and `nextTime` that check
both of them
            // do not transiently report that both delayed and queue
are empty during move
            delayed.removeFirstIf {
                if (it.timeToExecute(now)) {
                    enqueueImpl(it)
                } else
                    false
            } ?: break // quit loop when nothing more to remove or
enqueueImpl returns false on "isComplete"
        }
    }
    // then process one event from queue
    val task = dequeue()
    if (task != null) {
        task.run()
        return 0
    }
    return nextTime
}
```

核心逻辑就是如果事件到了需要执行的事件就拿出来然后执行，如果没有就返回下个任务执行的时间。

关于执行如下

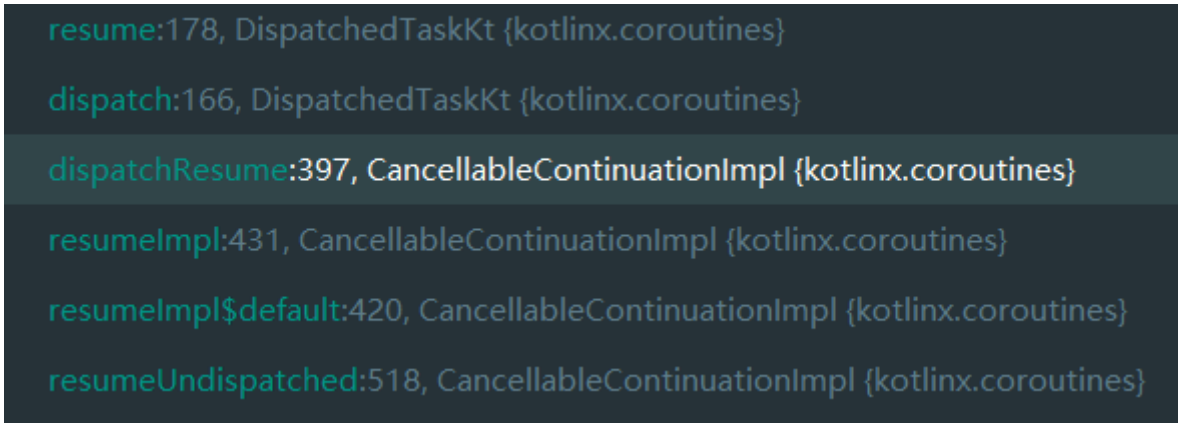
恢复执行

```
override fun run() { with(cont) { resumeUndispatched(Unit) } }
```

```
override fun CoroutineDispatcher.resumeUndispatched(value: T) {
    val dc = delegate as? DispatchedContinuation
    resumeImpl(value, if (dc?.dispatcher === this) MODE_UNDISPATCHED
else resumeMode)
}
```

```
private fun resumeImpl(
    proposedUpdate: Any?,
    resumeMode: Int,
    onCancellation: ((cause: Throwable) -> Unit)? = null
) {
    _state.loop { state ->
        when (state) {
            is NotCompleted -> {
                val update = resumedState(state, proposedUpdate,
resumeMode, onCancellation, idempotent = null)
                if (!_state.compareAndSet(state, update))
return@loop // retry on cas failure
                detachChildIfNonResuable()
                dispatchResume(resumeMode) // dispatch resume, but
it might get cancelled in process
                return // done
            }
            is CancelledContinuation -> {
                /*
                 * If continuation was cancelled, then resume
attempt must be ignored,
                 * because cancellation is asynchronous and may race
with resume.
                 * Racy exceptions will be lost, too.
                 */
                if (state.makeResumed()) { // check if trying to
resume one (otherwise error)
                    // call onCancellation
                    onCancellation?.let { callOnCancellation(it,
state.cause) }
                    return // done
                }
            }
        }
        alreadyResumedError(proposedUpdate) // otherwise, an error
(second resume attempt)
    }
}
```

算了懒得贴代码了，大致执行流程如下，把任务往下分发

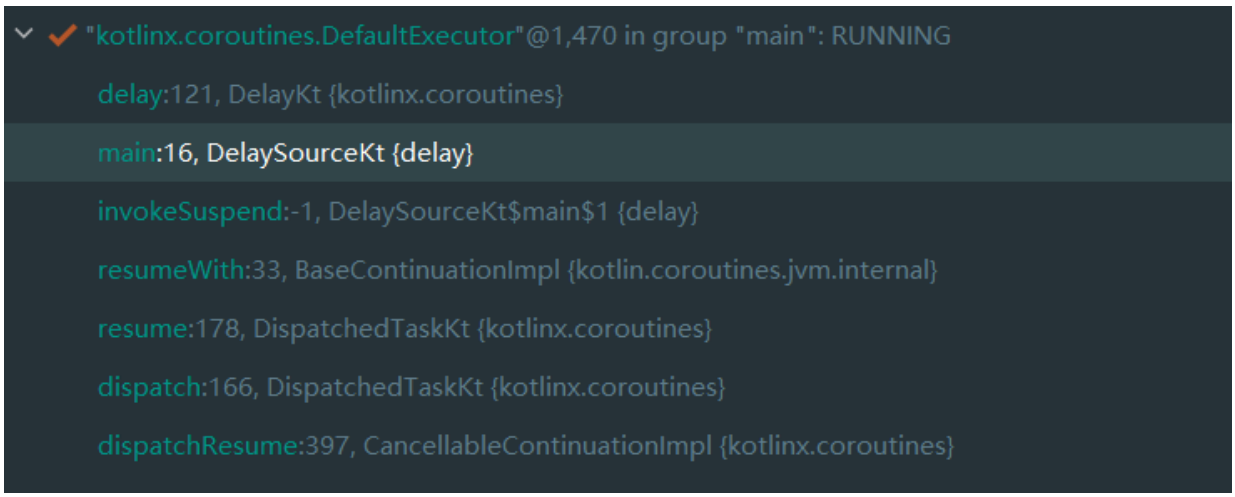


最后在DispatchedTask.resume调用resumeWith

```
when {
    undispatched -> (delegate as
DispatchedContinuation).resumeUndispatchedWith(result)
    else -> delegate.resumeWith(result)
}
```

然后就是一层层地恢复执行了

可以发现现在执行suspend main的线程不再是main线程了，而是DefaultExecutor new地一个守护线程。



然后执行完suspend main之后（两种情况，要么main被挂起 了，要么完全执行完了），返回到了DefaultExecutor的processNextEvent()

```
override fun run() {
    ThreadLocalEventLoop.setEventLoop(this)
    registerTimeLoopThread()
    try {
        var shutdownNanos = Long.MAX_VALUE
        if (!notifyStartup()) return
        while (true) {
            Thread.interrupted() // just reset interruption flag
            var parkNanos = processNextEvent()
            if (parkNanos == Long.MAX_VALUE) {
                // nothing to do, initialize shutdown timeout
```

```
                val now = nanoTime()
                if (shutdownNanos == Long.MAX_VALUE) shutdownNanos = now
+ KEEP_ALIVE_NANOS
                val tillShutdown = shutdownNanos - now
                if (tillShutdown <= 0) return // shut thread down
                parkNanos = parkNanos.coerceAtMost(tillShutdown)
            } else
                shutdownNanos = Long.MAX_VALUE
            if (parkNanos > 0) {
                // check if shutdown was requested and bail out in this
case
                if (isShutdownRequested) return
                parkNanos(this, parkNanos)
            }
        }
    } finally {
        _thread = null // this thread is dead
        acknowledgeShutdownIfNeeded()
        unregisterTimeLoopThread()
        // recheck if queues are empty after _thread reference was set
to null (!!!)
        if (!isEmpty) thread // recreate thread if it is needed
    }
}
```

它先会判断，返回的nextTime是不是maxValue，如果是就等KEEP_ALIVE_NANOS（1秒），否则就等parkNanos这么长时间，然后再去任务队列里面取东西。

除此之外执行过程中遭遇Exception就会关闭线程，然后清空，如果任务队列不为空重新创建线程。然后继续运行。

至此标准挂起函数delay也就分析完成了。

runBlocking——Source

> *Time： 2022-1-18*

> 其实这玩意不难，真的，如果前面的*Delay*看懂了，这个也就是几句话点破的东西，因为*runblocking*和*delay*的*DefaultExecutor*其实有很强的相似性（内部的核心都是*EventLoop*）。

分析方向

给出一个Demo

```
fun main() {
    runBlocking {
        println(Thread.currentThread().name)
        delay(100)
        println(Thread.currentThread().name)
    }
}
```

- 老规矩先Check一下编译器干了啥魔法事件没有

  显然是没有的
- 那么分析方向确定了也就是直接分析标准库内的代码

```
@Throws(InterruptedException::class)
public fun <T> runBlocking(context: CoroutineContext = EmptyCoroutineContext, block: suspend CoroutineScope.() -> T): T {
    contract {...}
    val currentThread = Thread.currentThread()
    val contextInterceptor = context[ContinuationInterceptor]
    val eventLoop: EventLoop?
    val newContext: CoroutineContext
    if (contextInterceptor == null) {
        // create or use private event loop if no dispatcher is specified
        eventLoop = ThreadLocalEventLoop.eventLoop
        newContext = GlobalScope.newCoroutineContext( context: context + eventLoop)
    } else {
        // See if context's interceptor is an event loop that we shall use (to support TestContext)
        // or take an existing thread-local event loop if present to avoid blocking it (but don't create one)
        eventLoop = (contextInterceptor as? EventLoop)?.takeIf { it.shouldBeProcessedFromContext() }
            ?: ThreadLocalEventLoop.currentOrNull()
        newContext = GlobalScope.newCoroutineContext(context)
    }
    val coroutine = BlockingCoroutine<T>(newContext, currentThread, eventLoop)
    coroutine.start(CoroutineStart.DEFAULT, coroutine, block)
    return coroutine.joinBlocking()
}
```

开始分析

进入runBlocking以后直接拿了currentThread和传入的CoroutineContext的ContinuationInterceptor，不过除了currentThread拿到的都是空的，然后创建了一个EventLoop，

```
internal actual fun createEventLoop(): EventLoop =
BlockingEventLoop(Thread.currentThread())
```

传入当前线程。

然后创建了一个新的coroutineContext

然后传入newCoroutineContext和currentThread以及eventLoop创建了一个BlockingCoroutine，

然后调用start

开启一个协程

```
val coroutine = BlockingCoroutine<T>(newContext, currentThread,
eventLoop)
coroutine.start(CoroutineStart.DEFAULT, coroutine, block)
return coroutine.joinBlocking()
```

```
public operator fun <R, T> invoke(block: suspend R.() -> T, receiver: R,
completion: Continuation<T>): Unit =
    when (this) {
        DEFAULT -> block.startCoroutineCancellable(receiver, completion)
        ATOMIC -> block.startCoroutine(receiver, completion)
        UNDISPATCHED -> block.startCoroutineUndispatched(receiver,
completion)
        LAZY -> Unit // will start lazily
    }
```

```
internal fun <R, T> (suspend (R) -> T).startCoroutineCancellable(
    receiver: R, completion: Continuation<T>,
    onCancellation: ((cause: Throwable) -> Unit)? = null
) =
    runSafely(completion) {
        createCoroutineUnintercepted(receiver,
completion).intercepted().resumeCancellableWith(Result.success(Unit),
onCancellation)
    }
```

然后就是标准的拦截协程然后resume

拦截之后就给block包了一层task，然后进入任务队列执行，

```
public final override fun <T> interceptContinuation(continuation:
Continuation<T>): Continuation<T> =
    DispatchedContinuation(this, continuation)
```

然后start就完毕了，之后就是joinBlocking。

```
fun joinBlocking(): T {
    registerTimeLoopThread()
    try {
        eventLoop?.incrementUseCount()
        try {
            while (true) {
                @Suppress("DEPRECATION")
```

```
                if (Thread.interrupted()) throw
InterruptedException().also { cancelCoroutine(it) }
                val parkNanos = eventLoop?.processNextEvent() ?:
Long.MAX_VALUE
                // note: process next even may loose unpark flag, so
check if completed before parking
                if (isCompleted) break
                parkNanos(this, parkNanos)
            }
        } finally { // paranoia
            eventLoop?.decrementUseCount()
        }
    } finally { // paranoia
        unregisterTimeLoopThread()
    }
    // now return result
    val state = this.state.unboxState()
    (state as? CompletedExceptionally)?.let { throw it.cause }
    return state as T
}
```

然后死循环去拿任务，拿到了就执行，没拿到就等到下一个执行完成（直到任务全部完成退出循环）。

总结

简单来说runBlocking其实没干啥事情，就是把当前的线程转化为了EventLoop的thread，这样当函数挂起之前就会往eventLoop里面放task，然后等到函数挂起后main线程就跑回去取任务队列里的任务（怎么看都像是handler），然后执行，就这样在单个线程里面就完成了挂起恢复，异步逻辑同步化。

suspend()->Unit 扩展——Source

> *Time: 2022-1-18*

createCoroutine

代码如下

```
public fun <T> (suspend () -> T).createCoroutine(
    completion: Continuation<T>
): Continuation<Unit> =

 SafeContinuation(createCoroutineUnintercepted(completion).intercepted()
, COROUTINE_SUSPENDED)
```

除此之外还有带receiver的（实现原理类似，不做解析）

就是把continuation 拦截然后在外包裹了一个SafeContinuation

之所以要createCoroutineUnintercepted是为了确保挂起函数具备最基本的挂起和恢复的特性，用Safe包装是为了防止开发者createCoroutine以后多次resume，造成一些安全隐患。

看看具体实现

> 这段代码似乎有些熟悉，我好像见过——蔷

创建一个没有被拦截的协程

```
public actual fun <T> (suspend () -> T).createCoroutineUnintercepted(
    completion: Continuation<T>
): Continuation<Unit> {
    val probeCompletion = probeCoroutineCreated(completion)
    return if (this is BaseContinuationImpl)
        create(probeCompletion)
    else
        createCoroutineFromSuspendFunction(probeCompletion) {
            (this as Function1<Continuation<T>, Any?>).invoke(it)
        }
}
```

算了还是分析一下，当时或许分析的比较草率。

首先是probeCoroutineCreated就是把completion返回了(具体在干哈我也不知道，至少现在是这样)。

然后就根据this直接返回了结果。

这里肯定走的是if分支，因为this 是suspend () -> T，

suspend()->T是个什么类？反编译一下啊

```
public final class SuspendExtKt {
   public static final void main() {
      Function1 var0 = (Function1)(new Function1((Continuation)null) {...});
      ContinuationKt.createCoroutine(var0, (Continuation)(new Continuation() {...}));
   }

   // $FF: synthetic method
   public static void main(String[] var0) { main(); }
}
```

编译器施加了魔法，把suspend ()->T 编译成了一个Function1(SuspendLambda)也就是一个输入值，一个输出值的Function。（输入值是Continuation，输出值是T）。为啥他是BaseContinuationImpl其实我也不知道的（因为SuspendLambda）。

然后调用了create方法传入了 probeCompletion（也就是completion 等价于create时候传入的匿名Continuation ）

然后调用了this.create，点开发现啥也没有。

```
public open fun create(value: Any?, completion: Continuation<*>):
Continuation<Unit> {
    throw UnsupportedOperationException("create(Any?;Continuation) has
not been overridden")
}
```

说是已近被重写了。

那只有可能是在生成的Function1里面了。

还真在这里面呢

```
public final class SuspendExtKt {
   public static final void main() {
      Function1 var0 = (Function1)(new Function1((Continuation)null) {
         int label;

         @Nullable
         public final Object invokeSuspend(@NotNull Object var1) {...}

         @NotNull
         public final Continuation create(@NotNull Continuation completion) {...}

         public final Object invoke(Object var1) {...}
      });
      ContinuationKt.createCoroutine(var0, (Continuation)(new Continuation() {...}));
   }
```

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

create其实也没干什么事情，只是通过传入的Continuation new了一个Function1然后返回。

也就是说返回了一个当前类的实例，为啥要这样做呢？this其实是没有continuation的，至少编译成java的continuation是将null强转为Continuation，这样做可能是为了补全没有传入Continuation的空缺。然后就是拦截了

```
createCoroutineUnintercepted(completion).intercepted()
```

```
public actual fun <T> Continuation<T>.intercepted(): Continuation<T> =
    (this as? ContinuationImpl)?.intercepted() ?: this
```

```
public fun intercepted(): Continuation<Any?> =
    intercepted
        ?:
(context[ContinuationInterceptor]?.interceptContinuation(this) ?: this)
            .also { intercepted = it }
```

拦截的话先是强转然后调用它的intercepted方法如果context里面有ContinuationInterceptor就调用它的interceptContinuation方法。如果没有ContinuationInterceptor那就直接返回this，并把intercepted赋值为this。

拦截的流程就完毕了。

包裹SafeContinuation

然后就创建了一个SafeContinuation然后返回。（值得注意的是，这个SafeContinuation的初始化状态就是COROUTINE_SUSPENDED，也就是说创建即挂起）。

总结

- 利用createCoroutine创建返回的Continuation经过了一层SafeContinuation的包装，并且初始化的值就是挂起状态。
- 开启上述api创建的Coroutine很简单就直接resume即可。

startCoroutine

这个就不讲了

```
createCoroutineUnintercepted(completion).intercepted().resume(Unit)
```

立即resume。

startCoroutineCancellable

```
public fun <T> (suspend () -> T).startCoroutineCancellable(completion:
Continuation<T>): Unit = runSafely(completion) {

 createCoroutineUnintercepted(completion).intercepted().resumeCancellabl
eWith(Result.success(Unit))
}
```

首先进入了runSafely

```
private inline fun runSafely(completion: Continuation<*>, block: () ->
Unit) {
    try {
        block()
    } catch (e: Throwable) {
        dispatcherFailure(completion, e)
    }
}
```

lambda内部的内容和前面几个类似，resume不太一样这resume

```
public fun <T> Continuation<T>.resumeCancellableWith(
    result: Result<T>,
    onCancellation: ((cause: Throwable) -> Unit)? = null
): Unit = when (this) {
    is DispatchedContinuation -> resumeCancellableWith(result,
onCancellation)
    else -> resumeWith(result)
}
```

进入之后先判断是不是DispatchedContinuation，使得话就调用另一个。

如果不是就直接resumeWith相比前面的好像就是多了一个try catch。是这样嘛确实是的。

```
try {
    block()
} catch (e: Throwable) {
    dispatcherFailure(completion, e)
}
```

```
private fun dispatcherFailure(completion: Continuation<*>, e: Throwable)
{
    completion.resumeWith(Result.failure(e))
    throw e
}
```

其实这并不特殊，为啥？因为没有这个也能达到相同的效果，BaseContinuationImpl其实有try catch的逻辑，catch到了依然会resume。

Use this function to start coroutine in a cancellable way, so that it can be cancelled while waiting to be dispatched.

这句话是源码的注释

```
is DispatchedContinuation -> resumeCancellableWith(result,
onCancellation)
```

所以特殊点在于这个resumeCancellableWith

```
inline fun resumeCancellableWith(
    result: Result<T>,
    noinline onCancellation: ((cause: Throwable) -> Unit)?
) {
    val state = result.toState(onCancellation)
    if (dispatcher.isDispatchNeeded(context)) {
        _state = state
        resumeMode = MODE_CANCELLABLE
        dispatcher.dispatch(context, this)
    } else {
```

```
            executeUnconfined(state, MODE_CANCELLABLE) {
                if (!resumeCancelled(state)) {
                    resumeUndispatchedWith(result)
                }
            }
        }
    }
```

先将result转为状态（具体的fail or success）

然后问dispatcher是否往下分发。如果往下分布就分发，否则就是往下执行，但是执行结束了判断一下是否任务被取消了。

asFlow

```
@FlowPreview
public fun <T> (suspend () -> T).asFlow(): Flow<T> = flow {
    emit(invoke())
}
```

代码很简单就不讲解了。

sequence——Source

> *Time 2022 1-19*

概述

sequence其实用的不算是很多，我没见用过几次（还不是因为菜）。

它的很大一个特点就是是在一个线程上挂起和恢复，有点像runBlocking是吧?

但是sequence是非阻塞的挂起和恢复，有意思吧?

具体的使用方法如下

```
fun main() {
    val sequence = sequence<Int> {
        yield(1)
        yield(2)
        yield(3)
    }
    for (i in sequence){
        println(i)
    }
}
```

通过一个顶层函数直接声明。

sequence传入的高阶函数是带有SequenceScope receiver的。

而且这个Scope被打上了一个标签@RestrictsSuspension

这个意思是说，你只能调用this里面的suspend function，也就是只能调用SequenceScope里面的suspend function，其余的任何挂起函数都是不被允许的，如果你在sequence里面调用了delay，它会报错。

> *Restricted suspending functions can only invoke member or extension suspending functions on their restricted coroutine scope*

之所以这样是因为这个受限的挂起函数不会将挂起的一层层向调用处传递，（从只在单线程非阻塞式的挂起和恢复估计也能推断出）。

具体的使用如下

通过yield会实现当前函数的挂起，然后将对应的value返回。也就是说我通过for循环就会使得1 2 3依次打印。

测试代码

```
fun main() {
    val  iterator = sequence<Int> {
        var pre = 1
        var current = 1
        yield(pre)
        yield(current)
        for (i in 0..20){
            yield(pre+current)
            val tmp = pre
            pre = current
            current += tmp
        }
    }
```

```
    for (i in iterator){
        println(i)
    }
}
```

这是一个斐波那契数列的打印。初始 a1 = 1 ， a2 = 1

先记录pre current为1 ，然后把这两个依次yield，然后循环21次，先yield pre+current，然后把pre和current更新。这样消费者会将yield的值依次打印。

好像是有点神奇。

原理分析

方向浅析

编译器只对挂起函数做了点小动作（对了 for in是个语法糖，其实他是拿的Iterator，然后hasNext，next的调用，应该不会有人不知道吧）。

分析方向确定了就是标准库源码。

流程分析

构建Sequence

最外层调用构建sequnce

```
public fun <T> sequence(@BuilderInference block: suspend
SequenceScope<T>.() -> Unit): Sequence<T> = Sequence { iterator(block) }
```

Sequence 是何方神圣?

```
public interface Sequence<out T> {
    public operator fun iterator(): Iterator<T>
}
```

一个可以获取iterator的接口（看来后续的突破口在iterator了嘿）。

继续分析

```
public inline fun <T> Sequence(crossinline iterator: () -> Iterator<T>):
Sequence<T> = object : Sequence<T> {
    override fun iterator(): Iterator<T> = iterator()
}
```

哦Sequence函数就是直接返回一个Sequence接口的实现类。

传入的高阶函数是() -> Iterator

iterator(block)是吧。

```
public fun <T> iterator(@BuilderInference block: suspend
SequenceScope<T>.() -> Unit): Iterator<T> {
    val iterator = SequenceBuilderIterator<T>()
    iterator.nextStep = block.createCoroutineUnintercepted(receiver =
iterator, completion = iterator)
    return iterator
}
```

直接new了一个SequenceBuilderIterator

```
private class SequenceBuilderIterator<T> : SequenceScope<T>(), Iterator<T>, Continuation<Unit> {
    private var state = State_NotReady
    private var nextValue: T? = null
    private var nextIterator: Iterator<T>? = null
    var nextStep: Continuation<Unit>? = null

    override fun hasNext(): Boolean {...}

    override fun next(): T {...}

    private fun nextNotReady(): T {...}

    private fun exceptionalState(): Throwable = when (state) {...}


    override suspend fun yield(value: T) {...}

    override suspend fun yieldAll(iterator: Iterator<T>) {...}

    // Completion continuation implementation
    override fun resumeWith(result: Result<Unit>) {...}

    override val context: CoroutineContext
        get() = EmptyCoroutineContext
}
```

然后在block外套了一层Continuation（也就是SequenceBuilderIterator），然后赋值给iterator.nextStep直接返回。答案呼之欲出了。也就是说所有与sequence挂起恢复的操作都是在这个SequenceBuilderIterator里面实现的。他是sequence实现的核心类

执行调度

那里开始?

for循环欸

```
for (i in iterator){
    println(i)
}
```

这句话真正的意思是啥?

先拿iterator然后while判断iterator里面有没有元素，如果有就拿出来，没有就结束。

我们去SequenceBuilderIterator hasNext，next 打几个断点看看。

```
override fun hasNext(): Boolean {
    while (true) {
        when (state) {
            State_NotReady -> {}
            State_ManyNotReady ->
                if (nextIterator!!.hasNext()) {
                    state = State_ManyReady
                    return true
                } else {
                    nextIterator = null
                }
            State_Done -> return false
            State_Ready, State_ManyReady -> return true
            else -> throw exceptionalState()
        }

        state = State_Failed
        val step = nextStep!!
        nextStep = null
        step.resume(Unit)
    }
}
```

注意state的初始值是State_NotReady。

所以一开始直接跳出when

然后把state设置为 State_Failed

然后拿了nextStep。再置空。

之后将它resume。这样一轮循环就执行完毕了（没事别怕，是死循环）

然后程序就跑到了我们最熟悉的BaseContinuationImpl了。

```
while (true) {
    probeCoroutineResumed(current)
    with(current) {
        val completion = 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()
        if (completion is BaseContinuationImpl) {
            current = completion
            param = outcome
        } else {
            completion.resumeWith(outcome)
            return
        }
    }
}
```

然后进入了死循环，先invokeSuspend了，然后调用yield

```
override suspend fun yield(value: T) {
    nextValue = value
    state = State_Ready
    return suspendCoroutineUninterceptedOrReturn { c ->
        nextStep = c
        COROUTINE_SUSPENDED
    }
}
```

把value保存起来，然后设置状态，然后拦截协程Continuation，把它赋值给nextStep，最后返回一个COROUTINE_SUSPENDED。

然后resume的死循环就break了。

继续执行hashNext。

return了true。

接着就是next了。

```
override fun next(): T {
    when (state) {
        State_NotReady, State_ManyNotReady -> return nextNotReady()
```

```
            State_ManyReady -> {
                state = State_ManyNotReady
                return nextIterator!!.next()
            }
            State_Ready -> {
                state = State_NotReady
                @Suppress("UNCHECKED_CAST")
                val result = nextValue as T
                nextValue = null
                return result
            }
            else -> throw exceptionalState()
        }
    }
```

next好像没干啥，就是把状态切换了，然后把值给拿了，然后置空，然后返回。这样由hasNext->next的过程就完成了。

你或许会不禁感叹，原理好像不是很难。挂起函数有点意思。

yield——Source

> *Time: 2022 1-19*

简述

yield的主要作用是让位，它使用最多的场景就是单线程的恢复和挂起操作。

好吧，我们来尝试一下

测试代码

```
fun main(): Unit = runBlocking {
    launch {
        while (true) {
            //假如做了cpu密集任务
            println("A")
            delay(1000)
            yield()
        }
    }
```

```
    launch {
        while (true) {
            println("B")
            delay(1000)
            yield()
        }
    }
}
```

猜测一下它会怎么执行»

他会AB交错打印。

因为yield表示让位，也就是说先立即挂起，然后把任务放入线程池，这样就给了其他的任务执行的机会。（值得注意的是让位了不代表一定能让出去，任务调度的时候还是在线程池里面取东西，如果其他任务挂起时间太长了还是会选择取让位的协程。比如）

```
fun main(): Unit = runBlocking {
    launch {
        while (true) {
            //假如做了cpu密集任务
            println("A")
            delay(1000)
            yield()
        }
    }

    launch {
        while (true) {
            println("B")
            delay(1000000)
            yield()
        }
    }
}
```

开了两个协程，一个挂起1秒，另外一个1000秒，这样会发生什么呢？也就是1000次有可能让位一次。（简单来说就是我给你执行的机会，你如果不要，不领情那我继续执行了）。

原理分析

```
public suspend fun yield(): Unit = suspendCoroutineUninterceptedOrReturn
sc@ { uCont ->
    val context = uCont.context
    context.ensureActive()
```

```
    val cont = uCont.intercepted() as? DispatchedContinuation<Unit> ?:
return@sc Unit
    if (cont.dispatcher.isDispatchNeeded(context)) {

        cont.dispatchYield(context, Unit)
    } else {

        val yieldContext = YieldContext()
        cont.dispatchYield(context + yieldContext, Unit)

        if (yieldContext.dispatcherWasUnconfined) {
            return@sc if (cont.yieldUndispatched()) COROUTINE_SUSPENDED
else Unit
        }
    }
    COROUTINE_SUSPENDED
}
```

首先确保context是active

然后拦截并强转为DispatchedContinuation

然后询问是否dispatch

```
public open fun isDispatchNeeded(context: CoroutineContext): Boolean =
true
```

```
cont.dispatchYield(context, Unit)
```

继续分发

```
internal fun dispatchYield(context: CoroutineContext, value: T) {
    _state = value
    resumeMode = MODE_CANCELLABLE
    dispatcher.dispatchYield(context, this)
}
```

```
public open fun dispatchYield(context: CoroutineContext, block:
Runnable): Unit = dispatch(context, block)
```

```
public final override fun dispatch(context: CoroutineContext, block:
Runnable) = enqueue(block)
```

```
public fun enqueue(task: Runnable) {
    if (enqueueImpl(task)) {
        unpark()
    } else {
        DefaultExecutor.enqueue(task)
    }
}
```

```
private fun enqueueImpl(task: Runnable): Boolean {
    _queue.loop { queue ->
        if (isCompleted) return false // fail fast if already completed,
may still add, but queues will close
        when (queue) {
            null -> if (_queue.compareAndSet(null, task)) return true
            is Queue<*> -> {
                when ((queue as Queue<Runnable>).addLast(task)) {
                    Queue.ADD_SUCCESS -> return true
                    Queue.ADD_CLOSED -> return false
                    Queue.ADD_FROZEN -> _queue.compareAndSet(queue,
queue.next())
                }
            }
            else -> when {
                queue === CLOSED_EMPTY -> return false
                else -> {
                    // update to full-blown queue to add one more
                    val newQueue = Queue<Runnable>
(Queue.INITIAL_CAPACITY, singleConsumer = true)
                    newQueue.addLast(queue as Runnable)
                    newQueue.addLast(task)
                    if (_queue.compareAndSet(queue, newQueue)) return
true
                }
            }
        }
    }
}
```

调用了半天也就是把DispatchedTask放入到线程池里面去，然后在返回一个COROUTINE_SUSPEND

总结

yield的实现不难，就是把当前的协程上下文的Interceptor拿到手，然后强转为DispatchedTask，最后放入到线程池的任务队列里面，然后挂起协程。

launch——Source

> *Time 2022 -1-20*

简述

launch是协程构建器的一种，在工作或者日常开发中，用的极其频繁。

测试代码

```
fun main(): Unit = runBlocking {
    launch {
        while (true){
            delay(1000)
            println("one loop finished")
        }
    }
}
```

不出意外的话它的执行结果就是每隔1秒打印一句

println("one loop finished")

原理分析

```
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
}
```

关于的一些介绍，源码里注释给的比较详细，不做解释说明。

首先launch是一个CoroutineScope的扩展函数，需要接受一个带有CoroutineScope receiver的挂起函数。（这个函数没有返回值所以后面有了async）。

newCoroutineContext

从名称上看这应该是new了一个CoroutineContext（废话）。

```
public actual fun CoroutineScope.newCoroutineContext(context:
CoroutineContext): CoroutineContext {
    val combined = coroutineContext + context
    val debug = if (DEBUG) combined +
CoroutineId(COROUTINE_ID.incrementAndGet()) else combined
    return if (combined !== Dispatchers.Default &&
combined[ContinuationInterceptor] == null)
        debug + Dispatchers.Default else debug
}
```

调用了coroutineContext的+号重载运算符。（这里不分析）

```
public operator fun plus(context: CoroutineContext): CoroutineContext =
    if (context === EmptyCoroutineContext) this else // fast path --
avoid lambda creation
        context.fold(this) { acc, element ->
            val removed = acc.minusKey(element.key)
            if (removed === EmptyCoroutineContext) element else {
                // make sure interceptor is always last in the context
(and thus is fast to get when present)
                val interceptor = removed[ContinuationInterceptor]
                if (interceptor == null) CombinedContext(removed,
element) else {
                    val left = removed.minusKey(ContinuationInterceptor)
                    if (left === EmptyCoroutineContext)
CombinedContext(element, interceptor) else
                        CombinedContext(CombinedContext(left, element),
interceptor)
                }
            }
        }
```

大致就是把把传入的Context外包一层CombinedContext然后返回。

之后这个就是debug，debug这个是为了方便调试的给协程加了一个id，不用管。

最后就是return了，他这里判断了一下，如果combined不是Dispatchers.Default，而且没有其他的拦截器。那就再CoroutineContext里面硬塞一个Dispatchers.Default，否则就直接把之前的返回。

newCoroutine

依据启动模式new一个协程

```
val coroutine = if (start.isLazy)
    LazyStandaloneCoroutine(newContext, block) else
    StandaloneCoroutine(newContext, active = true)
```

lazy是必须调用start才会启动，StandaloneCoroutine是在launchd的时候就启动。

开始调度（但不保证立即执行）。默认是Default，所以会进入StandaloneCoroutine

start

```
public fun <R> start(start: CoroutineStart, receiver: R, block: suspend
R.() -> T) {
    start(block, receiver, this)
}
```

这丫调用了CoroutineStart这个枚举类的重载方法。（真会玩）

```
public operator fun <R, T> invoke(block: suspend R.() -> T, receiver: R,
completion: Continuation<T>): Unit =
    when (this) {
        DEFAULT -> block.startCoroutineCancellable(receiver, completion)
        ATOMIC -> block.startCoroutine(receiver, completion)
        UNDISPATCHED -> block.startCoroutineUndispatched(receiver,
completion)
        LAZY -> Unit // will start lazily
    }
```

然后根据枚举类的不同，调用不同的方法来启动协程。

先看看Default会怎么启动

```
internal fun <R, T> (suspend (R) -> T).startCoroutineCancellable(
    receiver: R, completion: Continuation<T>,
    onCancellation: ((cause: Throwable) -> Unit)? = null
) =
    runSafely(completion) {
        createCoroutineUnintercepted(receiver,
completion).intercepted().resumeCancellableWith(Result.success(Unit),
onCancellation)
    }
```

这个在分析runBlocking的时候以及分析过了，拦截Continuation，然后包装了一个DispatchedContinuation然后调用dispatch，在线程池的任务队列里面放一个任务。

然后看看ATOMIC

直接start也就是不做取消处理

除此之外就是UNDISPATCHED

```
UNDISPATCHED -> block.startCoroutineUndispatched(receiver, completion)
```

```
internal fun <R, T> (suspend (R) ->
T).startCoroutineUndispatched(receiver: R, completion: Continuation<T>)
{
    startDirect(completion) { actualCompletion ->
        withCoroutineContext(completion.context, null) {
            startCoroutineUninterceptedOrReturn(receiver,
actualCompletion)
        }
    }
}
```

```
private inline fun <T> startDirect(completion: Continuation<T>, block:
(Continuation<T>) -> Any?) {
    val actualCompletion = probeCoroutineCreated(completion)
    val value = try {
        block(actualCompletion)
    } catch (e: Throwable) {
        actualCompletion.resumeWithException(e)
        return
    }
    if (value !== COROUTINE_SUSPENDED) {
        @Suppress("UNCHECKED_CAST")
        actualCompletion.resume(value as T)
    }
}
```

```
internal actual inline fun <T> withCoroutineContext(context:
CoroutineContext, countOrElement: Any?, block: () -> T): T {
    val oldValue = updateThreadContext(context, countOrElement)
    try {
        return block()
    } finally {
        restoreThreadContext(context, oldValue)
    }
}
```

就这样相当于直接拦截了当前的上下文，然后直接在当前线程run。

最后就是这个lazy了，这个lazy啥也没做，直接返回了一个Unit。

他要等到调用job.start才会开启

```
public final override fun start(): Boolean {
    loopOnState { state ->
        when (startInternal(state)) {
            FALSE -> return false
            TRUE -> return true
        }
    }
}
```

```
private fun startInternal(state: Any?): Int {
    when (state) {
        is Empty -> { // EMPTY_X state -- no completion handlers
            if (state.isActive) return FALSE // already active
            if (!_state.compareAndSet(state, EMPTY_ACTIVE)) return RETRY
            onStart()
            return TRUE
        }
        is InactiveNodeList -> { // LIST state -- inactive with a list
of completion handlers
            if (!_state.compareAndSet(state, state.list)) return RETRY
            onStart()
            return TRUE
        }
        else -> return FALSE // not a new state
    }
}
```

```
override fun onStart() {
    continuation.startCoroutineCancellable(this)
}
```

也就是调用了startCoroutineCancellable

如此launch算是解析完成了。

总结

launch实际上就是拦截了一下当前的上下文，然后在线程池的任务队列里面放任务而已。（启动模式为UNDISPATCHED的除外，这玩意是直接在当前Thread立即执行，直到挂起）

async——Source

> *Time: 2022 1-20*

launch虽好，但有一大缺点。就是不能异步返回值。所以async就是为了补全它这一缺点的。

测试代码

```
fun main() = runBlocking {
    val defferJob = async {
        delay(1000)
        "假装有返回值"
    }
    val result = defferJob.await()
    println(result)
}
```

正常情况下，在delay 1秒后会打印处返回值。

分析

```
public fun <T> CoroutineScope.async(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
): Deferred<T> {
    val newContext = newCoroutineContext(context)
    val coroutine = if (start.isLazy)
        LazyDeferredCoroutine(newContext, block) else
        DeferredCoroutine<T>(newContext, active = true)
    coroutine.start(start, coroutine, block)
    return coroutine
}
```

这没来由的熟悉感

newCoroutineContext

```
val newContext = newCoroutineContext(context)
```

这个就不分析了，前面launch已近分析过了

newCoroutine

```
val coroutine = if (start.isLazy)
    LazyDeferredCoroutine(newContext, block) else
    DeferredCoroutine<T>(newContext, active = true)
```

一样的套路根据是否是lazy启动方式来new不同的协程。

start

```
coroutine.start(start, coroutine, block)
```

莫名的熟悉感

依然调用的CoroutineStart的invoke重载

```
public operator fun <T> invoke(block: suspend () -> T, completion:
Continuation<T>): Unit =
    when (this) {
        DEFAULT -> block.startCoroutineCancellable(completion)
        ATOMIC -> block.startCoroutine(completion)
        UNDISPATCHED -> block.startCoroutineUndispatched(completion)
        LAZY -> Unit // will start lazily
    }
```

一样的启动方式。

还是放入线程池的任务队列。

唯一的差别就是返回值不一样了

await

```
override suspend fun await(): T = awaitInternal() as T
```

```
internal suspend fun awaitInternal(): Any? {
    // fast-path -- check state (avoid extra object creation)
    while (true) { // lock-free loop on state
        val state = this.state
        if (state !is Incomplete) {
            // already complete -- just return result
            if (state is CompletedExceptionally) { // Slow path to
recover stacktrace
                recoverAndThrow(state.cause)
            }
            return state.unboxState()
```

```
        }
        if (startInternal(state) >= 0) break // break unless needs to
retry
    }
    return awaitSuspend() // slow-path
}
```

```
private fun startInternal(state: Any?): Int {
    when (state) {
        is Empty -> { // EMPTY_X state -- no completion handlers
            if (state.isActive) return FALSE // already active
            if (!_state.compareAndSet(state, EMPTY_ACTIVE)) return RETRY
            onStart()
            return TRUE
        }
        is InactiveNodeList -> { // LIST state -- inactive with a list
of completion handlers
            if (!_state.compareAndSet(state, state.list)) return RETRY
            onStart()
            return TRUE
        }
        else -> return FALSE // not a new state
    }
}
```

由于没有完成所以出了循环，然后进入了awaitSuspend。

```
private suspend fun awaitSuspend(): Any? =
suspendCoroutineUninterceptedOrReturn { uCont ->
    /*
     * Custom cold here, so that parent coroutine that is using await
     * on its child deferred (async) coroutine would throw the exception
that this child had
     * thrown and not a JobCancellationException.
     */
    val cont = AwaitContinuation(uCont.intercepted(), this)
    // we are mimicking suspendCancellableCoroutine here and call
initCancellability, too.
    cont.initCancellability()

 cont.disposeOnCancellation(invokeOnCompletion(ResumeAwaitOnCompletion(c
ont).asHandler))
    cont.getResult()
}
```

然后拿了当前的Continuation实例，然后包裹了一层AwaitContinuation

然后在调用cont.disposeOnCancellation()加入了一个完成的回调。

```
private fun installParentHandle(): DisposableHandle? {
    val parent = context[Job] ?: return null // don't do anything
without a parent
    // Install the handle
    val handle = parent.invokeOnCompletion(
        onCancelling = true,
        handler = ChildContinuation(this).asHandler
    )
    parentHandle = handle
    return handle
}
```

拿了上下文的job，然后再job里面加了一个完成的回调。

这样我们的分析重点就到了ResumeAwaitOnCompletion

然后等上面的任务完成以后就会直接调用continuation.resume(state.unboxState() as T)这样就拿到了async的返回值。

```
override fun invoke(cause: Throwable?) {
    val state = job.state
    assert { state !is Incomplete }
    if (state is CompletedExceptionally) {
        // Resume with with the corresponding exception to preserve it
        continuation.resumeWithException(state.cause)
    } else {
        // Resuming with value in a cancellable way (AwaitContinuation
is configured for this mode).
        @Suppress("UNCHECKED_CAST")
        continuation.resume(state.unboxState() as T)
    }
}
```

总结

async和launch其实基本上是一致的，async是在launch的基础上加了一个任务完成的回调，等aysnc的任务完成就调用对应回调然后resume这样就在await这个挂起点恢复了，然后继续执行。

CoroutineContext——Source

> *Time：2022-1-21 CoroutineContext plus，minusKey，fold，自定义CoroutineContext*
>
> *Time：2022-1-23 CoroutineContext Key与AbstractCoroutineContextKey*

```
public interface CoroutineContext {

    Returns the element with the given key from this context or null.

    public operator fun <E : Element> get(key: Key<E>): E?


    Accumulates entries of this context starting with initial value and applying operation from left to
    right to current accumulator value and each element of this context.

    public fun <R> fold(initial: R, operation: (R, Element) → R): R


    Returns a context containing elements from this context and elements from other context. The
    elements from this context with the same key as in the other one are dropped.

    public operator fun plus(context: CoroutineContext): CoroutineContext ={...}


    Returns a context containing elements from this context, but without an element with the specified
    key.

    public fun minusKey(key: Key<*>): CoroutineContext


    Key for the elements of CoroutineContext. E is a type of element with this key.

    public interface Key<E : Element>


    An element of the CoroutineContext. An element of the coroutine context is a singleton context by
    itself.

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

CoroutineContext是一个接口（听君一席话，如听一席话）

CoroutineContext的自我介绍

这又是个什么东西？源码注释是这样写的

> *Persistent context for the coroutine. It is an indexed set of Element instances. An indexed set is a mix between a set and a map. Every element in this set has a unique Key.*

为协程保存环境，他是一个支持索引的**Element**实例的集合，他是**Set**和**Map**的混合，每一个Element元素都有独一无二的Key。

清晰明了，CoroutineContext是保存协程的环境的，它本身是一种自定义的数据结构，介于Set和Map之间。

CoroutineContext的实现

实现？接口里面给了实现的啊。

get

> *Returns the element with the given key from this context or null.*

```
public operator fun <E : Element> get(key: Key<E>): E?
```

这是get的重载运算符。依据传入的key来获取对应的元素。不过没给出实现，这个得子类自己实现。but如果你查看它的实现类的时候会发现，kotlin.coroutines.CoroutineContext.Element也就是这个接口的内部接口继承了CoroutineContext给出了元素的默认实现。

```
public interface Element : CoroutineContext {
        /**
         * A key of this coroutine context element.
         */
    public val key: Key<*>

    public override operator fun <E : Element> get(key: Key<E>): E? =
    @Suppress("UNCHECKED_CAST")
    if (this.key == key) this as E else null
    ...
}
```

就是判断一下内部保存的Key和传入的key是否一致，一致就返回元素this，否则就null。

fold

> *Accumulates entries of this context starting with initial value and applying operation from left to right to current accumulator value and each element of this context.*

```
public fun <R> fold(initial: R, operation: (R, Element) -> R): R
```

从初始值开始累加该上下文的条目，并从左到右对当前累加器值和该上下文的每个元素进行操作。

好像没看太懂。

看看Element的默认实现

```
public override fun <R> fold(initial: R, operation: (R, Element) -> R):
R =
    operation(initial, this)
```

就是把传入的高阶函数invoke?

minus

> *Returns a context containing elements from this context, but without an element with the specified key.*

```
public fun minusKey(key: Key<*>): CoroutineContext
```

返回一个上下文，这个上下文中的元素不包含匹配给出的Key的元素。有点绕啊。

实现依然是在Element里面

```
public override fun minusKey(key: Key<*>): CoroutineContext =
    if (this.key == key) EmptyCoroutineContext else this
```

如果包含key返回空的CoroutineContext，否者就返回this。和get是相反的。

plus

> *Returns a context containing elements from this context and elements from other context. The elements from this context with the same key as in the other one are dropped.*

这个的实现是在CoroutineContext里面了。

返回一个上下文包含this的所有元素和传入上下文的所有元素。（会把具有相同Key的元素删除掉。）

好像不是很通顺的样子。

看看源码

```
public operator fun plus(context: CoroutineContext): CoroutineContext =
    if (context === EmptyCoroutineContext) this else // fast path --
avoid lambda creation
        context.fold(this) { acc, element ->
            val removed = acc.minusKey(element.key)
            if (removed === EmptyCoroutineContext) element else {
                // make sure interceptor is always last in the context
(and thus is fast to get when present)
                val interceptor = removed[ContinuationInterceptor]
                if (interceptor == null) CombinedContext(removed,
element) else {
                    val left = removed.minusKey(ContinuationInterceptor)
                    if (left === EmptyCoroutineContext)
CombinedContext(element, interceptor) else
                        CombinedContext(CombinedContext(left, element),
interceptor)
                }
            }
        }
```

如果传入的Context是Empty的，那么直接返回this即可。

否则就调用fold之后呢又是new Combinded又是调用minusKey又是调用fold。好像看上去没啥逻辑，but我们知道它的最终目的是啥。这就够了。

测试代码

不得不说前面那段代码好像不太好看，所以还是上测试代码来校验一下是怎么实现的。

```
fun main() {
    val context = EmptyCoroutineContext
    println(context.javaClass)

    val context1 = context + CoroutineName("Test")
    println(context1.javaClass)

    val context2 =
        context1 + CoroutineExceptionHandler { coroutineContext,
throwable -> println(coroutineContext.toString() + throwable.toString())
}
    println(context2.javaClass)

    val context3 = context2 + Dispatchers.Main
    println(context3.javaClass)

    val context4 = context3 + NonCancellable
    println(context4.javaClass)
```

```
}
```

测试代码对上下文进行了4次加法运算。

最后的terminal结果为

> *class kotlin.coroutines.EmptyCoroutineContext*
> *class kotlinx.coroutines.CoroutineName*
> *class kotlin.coroutines.CombinedContext*
> *class kotlin.coroutines.CombinedContext*
> *class kotlin.coroutines.CombinedContext*

发现了嘛?

在第一次new了一个EmptyCoroutineContext后对加法运算的结果是CoroutineName其余的全是CombinedContext。

所以我们的分析变成了四个。

EmptyCoroutineContext，CoroutineName，CombinedContext，Element（为什么有这玩意，因为它提供了默认的实现。）

流程分析

EmptyCoroutineContext

```
public object EmptyCoroutineContext : CoroutineContext, Serializable {
    private const val serialVersionUID: Long = 0
    private fun readResolve(): Any = EmptyCoroutineContext

    public override fun <E : Element> get(key: Key<E>): E? = null
    public override fun <R> fold(initial: R, operation: (R, Element) ->
R): R = initial
    public override fun plus(context: CoroutineContext):
CoroutineContext = context
    public override fun minusKey(key: Key<*>): CoroutineContext = this
    public override fun hashCode(): Int = 0
    public override fun toString(): String = "EmptyCoroutineContext"
}
```

很简单

get返回空。

fold直接发挥initial。

plus返回传入的元素。

minusKey直接返回this。

总的来说EmptyCoroutineContext就是一个空壳除了是CoroutineContext，其余啥也不是。

一次合并

```
val context1 = EmptyCoroutineContext + CoroutineName("Test")
```

显然直接返回了CoroutineName

CoroutineName

```
public data class CoroutineName(
    val name: String
) : AbstractCoroutineContextElement(CoroutineName) {
    public companion object Key : CoroutineContext.Key<CoroutineName>
    override fun toString(): String = "CoroutineName($name)"
}
```

```
public abstract class AbstractCoroutineContextElement(public override
val key: Key<*>) : Element
```

CoroutineName是个什么东西？（一个比EmptyCoroutineContext还懒的家伙。）直接复用AbstractCoroutineContextElement,然后AbstractCoroutineContextElement又是实现的Element接口，所以CoroutineName就是用的Element的默认实现。

```
public interface Element : CoroutineContext {
    public val key: Key<*>

    public override operator fun <E : Element> get(key: Key<E>): E? =
        @Suppress("UNCHECKED_CAST")
        if (this.key == key) this as E else null

    public override fun <R> fold(initial: R, operation: (R, Element) ->
R): R =
        operation(initial, this)

    public override fun minusKey(key: Key<*>): CoroutineContext =
        if (this.key == key) EmptyCoroutineContext else this
}
```

这个之前已经分析过了，而且本身难度不大。

另外如果你点开***Coroutine***标准库的其他例如***Dispatcher CoroutineId***等一系列***Element***的实现的时候，大多都是用的***Element***的默认实现。

二次合并

```
val context1 = EmptyCoroutineContext + CoroutineName("Test") +
CoroutineExceptionHandler { coroutineContext, throwable ->
println(coroutineContext.toString() + throwable.toString()) }
```

在第一次的基础上加入了一个CoroutineExceptionHandler

```
public operator fun plus(context: CoroutineContext): CoroutineContext =
    if (context === EmptyCoroutineContext) this else // fast path --
avoid lambda creation
        context.fold(this) { acc, element ->
            val removed = acc.minusKey(element.key)
            if (removed === EmptyCoroutineContext) element else {
                // make sure interceptor is always last in the context
(and thus is fast to get when present)
                val interceptor = removed[ContinuationInterceptor]
                if (interceptor == null) CombinedContext(removed,
element) else {
                    val left = removed.minusKey(ContinuationInterceptor)
                    if (left === EmptyCoroutineContext)
CombinedContext(element, interceptor) else
                        CombinedContext(CombinedContext(left, element),
interceptor)
                }
            }
        }
```

调用plus了，先判断context是不是Empty，显然不是。

调用了fold，this是CoroutineName ,coroutine 是 CoroutineExceptionHandler。

removed显然是CoroutineName，

所以进入了else，之后的ContinuationInterceptor拿不到new了一个CombinedContext，left是CoroutineName，right是CoroutineExceptionHandler。

如果Interceptor拿到了还分类判断，如果removed没东西了，就把interceptor往右边堆，Empty就不放入到CombinedContext里去了，如果不空那就嵌套组合两层，把interceptor往右边堆。分析结束。

总结(二次分析)

比如有两个CoroutineContext A和B（如果B是Element的话）。

step 1

先在A里面找找有没有B，如果有就删除，如果A里面删除之后啥也没有了，那就直接返回B。这段逻辑主要是为了防止重复添加相同的元素。比如这样。

```
fun main() {
    var context:CoroutineContext = CoroutineName("A")
    context += CoroutineName("B")
    println(context.javaClass.toString()+"->" + context)
    context += CoroutineName("C")
    println(context.javaClass.toString()+"->" + context)
}
```

> *class kotlinx.coroutines.CoroutineName->CoroutineName(B)*
> *class kotlinx.coroutines.CoroutineName->CoroutineName(C)*

在协程中添加相同类型的上下文，后面的会覆盖之前的。

step 2

如果A减去了B（Element）剩下的不是空的话，那么又会尝试去拿ContinuationInterceptor，没拿到那就直接把第一步的结果和B合成一个CombinedContext。

如果拿到了，那不好意思还得把ContinuationInterceptor给减了，然后根据减了以后的coroutineContext是不是空的，如果是空的，那就把element放左边，然后ContinuationInterceptor放右边。如果不是空的，那么得嵌套两次。构建的CombinedContext层级关系如下。

（（A-B-ContinuationInterceptor，B），ContinuationInterceptor）。

（其实这段如果简单的来说就是把A中的ContinuationInterceptor提出来放在最右边。）

```
fun main() {
    var context=  Dispatchers.IO + CoroutineName("A")
    println(context)
    val context2  = Dispatchers.IO + CoroutineName("B") + NonCancellable
+ CoroutineExceptionHandler { coroutineContext, throwable ->
println(coroutineContext.toString() + throwable.toString()) }
    println(context2)
}
```

> *[CoroutineName(A), Dispatchers.IO]*
> *[CoroutineName(B), NonCancellable,*
> *TestKt$main$$inlined$CoroutineExceptionHandler$1@593634ad, Dispatchers.IO]*

CombinedContext

这个代码就有点点多

get

```
override fun <E : Element> get(key: Key<E>): E? {
    var cur = this
    while (true) {
        cur.element[key]?.let { return it }
        val next = cur.left
        if (next is CombinedContext) {
            cur = next
        } else {
            return next[key]
        }
    }
}
```

我们知道在构建CombinedContext的时候会传入两个CoroutineContext，一个left一个right。CombinedContext就像是一个卷，两个包一层比如这样（（（A，B），C），D）,F),所以要遍历他就是从最右边开始（没发现嘛右边的都是单个元素），然后依次向左边深入。

get也确实是这样的，来了一个死循环，先将右边的元素调用一下get看能不能拿到值，拿到了就返回。拿不到就向左边迭代，然后判断，左边是不是CombinedContext，是的话循环迭代，不是的话就get一下无论拿不拿得到值都return了（最左边都拿不到说明就是没有这个Element）。

fold

```
public override fun <R> fold(initial: R, operation: (R, Element) -> R):
R =
    operation(left.fold(initial, operation), element)
```

fold就简单一些，递归调用left的fold（有点像深度优先遍历了hh）

minusKey

```
public override fun minusKey(key: Key<*>): CoroutineContext {
    element[key]?.let { return left }
    val newLeft = left.minusKey(key)
    return when {
        newLeft === left -> this
        newLeft === EmptyCoroutineContext -> element
        else -> CombinedContext(newLeft, element)
    }
}
```

先判断右边的element里面有没有，有就返回left，然后递归调用left的minusKey，最后判断一下newLeft和left是否相等，相等说明没有找到，newLeft为空说明这个是在最左边，其他情况就是在中间，所以就会一层层地从断开处重新连接。这样最后返回的CoroutineContext就是一个去除掉对应key的CoroutineContext

注意plus还是用的默认实现。

除此之外还有一些其他的实现，比如什么size，contains，containsAll

三次合并

```
val context1 = EmptyCoroutineContext + CoroutineName("Test") +
CoroutineExceptionHandler { ...} + Dispatchers.Main
```

```
public operator fun plus(context: CoroutineContext): CoroutineContext =
    if (context === EmptyCoroutineContext) this else // fast path --
avoid lambda creation
        context.fold(this) { acc, element ->
            val removed = acc.minusKey(element.key)
            if (removed === EmptyCoroutineContext) element else {
                val interceptor = removed[ContinuationInterceptor]
                if (interceptor == null) CombinedContext(removed,
element) else {
                    val left = removed.minusKey(ContinuationInterceptor)
                    if (left === EmptyCoroutineContext)
CombinedContext(element, interceptor) else
                        CombinedContext(CombinedContext(left, element),
interceptor)
                }
            }
        }
```

移除‘被加数’协程上下文中的作为‘加数’的协程上下文，然后拿ContinuatiInonInterceptor把ContinuatiInonInterceptor放到最右边。然后返回对应的协程上下文。

四次合并

类似不再分析。

总结

如此CoroutineContext就分析完成了，惊不惊喜？CoroutineContext竟然是一种自定义的介于Set和Map的数据结构。

在这个数据结构种添加元素只需要用+号即可，他会自动转为CombinedContext。

Key & AbstractCoroutineKey

Key

```
public interface Key<E : Element>
```

Key的代码如上，很短。就像一个标记接口，but这里这里需要传入一个Element的泛型这样使得Key与泛型进行绑定。（最好保持Key为单例，因为索引的时候是依据Key进行索引的，也就是说同一类型的Element如果不是单例那么就会导致一个CoroutineContext混乱，不好取，同时里面的Element也是不合理，比如一个CoroutineContext里面有两个相同类型的Element？这显然是荒谬的）

```
fun main() {
    val coroutineContext = A("A") + A("AA")
    println(coroutineContext)
}

data class A(val str: String) : AbstractCoroutineContextElement(AKey())
{

}

class AKey : CoroutineContext.Key<A>
```

运行一下你就会发现coroutineContext内部有两个A，这显然是有些不太符合常理的，而且不好索引这两个元素，除非你把两个Key都保存了。否则就是很麻烦的。

AbstractCoroutineKey

```
public abstract class AbstractCoroutineContextKey<B : Element, E : B>(
    baseKey: Key<B>,
    private val safeCast: (element: Element) -> E?
) : Key<E> {
    private val topmostKey: Key<*> = if (baseKey is
AbstractCoroutineContextKey<*, *>) baseKey.topmostKey else baseKey

    internal fun tryCast(element: Element): E? = safeCast(element)
    internal fun isSubKey(key: Key<*>): Boolean = key === this ||
topmostKey === key
}
```

这是一个有想法的Key。

通常的一个元素只能由一个Key来索引的。(如果你在AbstractCoroutineElement中传入的是Key的直接实现子类)。

一切总是有特例的。比如?

```
@ExperimentalStdlibApi
fun main() {
    val x = Dispatchers.IO + CoroutineName("我是一个CoroutineName一鸭一鸭
哟")
    println(x[ContinuationInterceptor])
    println(x[CoroutineDispatcher])
}
```

输出结果猜猜是什么。

> *Dispatchers.IO*
> *Dispatchers.IO*

很奇怪吧。点开Dispachers.IO看看他是啥。

```
public val IO: CoroutineDispatcher = DefaultScheduler.IO
```

他是CoroutineDIspatcher。CoroutineDispatcher是ContinuationInterceptor的子类。

欧我懂了，就多态是吧？？？一定是这样的OOP我学得可好了。

很遗憾不是。建议回去看看Element以及CombinedContext中get的实现是怎么样子的。

```
override fun <E : Element> get(key: Key<E>): E? {
    var cur = this
    while (true) {
        cur.element[key]?.let { return it }
        val next = cur.left
        if (next is CombinedContext) {
            cur = next
        } else {
            return next[key]
        }
    }
}
```

Combinded直接丢锅给了Element。

```
public override operator fun <E : Element> get(key: Key<E>): E? =
    @Suppress("UNCHECKED_CAST")
    if (this.key == key) this as E else null
```

然而Element的实现就有些耐人寻味了，它直接调用的equals方法。

所以多态嘛是不现实的。

再仔细看看你会发现他其实是在创建CoroutineDispacher的时候传入了一个很奇怪的Key

```
public abstract class CoroutineDispatcher :
    AbstractCoroutineContextElement(ContinuationInterceptor),
ContinuationInterceptor {

    /** @suppress */
    @ExperimentalStdlibApi
    public companion object Key :
AbstractCoroutineContextKey<ContinuationInterceptor,
CoroutineDispatcher>(
        ContinuationInterceptor,
        { it as? CoroutineDispatcher })
    }
```

开始步入正题了，这个AbstractCoroutineContextKey是干什么用的，它是一个Key，一个绑定了父Key的Key，也就是说这个Key是有层级关系的，这种层级关系emm，怎么像是继承，一个Key内包含一个父Key。那么说他是可以用来描述任何Element的继承关系了？确实是这样的。

如果你点开他这源码认证看看你就知道为啥了。

```
@SinceKotlin("1.3")
@ExperimentalStdlibApi
public abstract class AbstractCoroutineContextKey<B : Element, E : B>(
    baseKey: Key<B>,
    private val safeCast: (element: Element) -> E?
) : Key<E> {
    private val topmostKey: Key<*> = if (baseKey is
AbstractCoroutineContextKey<*, *>) baseKey.topmostKey else baseKey

    internal fun tryCast(element: Element): E? = safeCast(element)
    internal fun isSubKey(key: Key<*>): Boolean = key === this ||
topmostKey === key
}
```

有一个safeCast讲Element转为E? ，还有一个baseKey，一个topmostKey（这是获取的最顶层的Key，类似于继承体系中的超类）。

- topmost

  topmost的实现不难，也就是baseKey也就是说父Key是抽象Key那么就往上继续拿topmostKey，这样最后拿到的就是老祖宗Key。

- tryCast

  调用init时候传入的safeCast转化为E?(不是转化为父Key的类型)

- isSubKey

  判断传入的key是不是Key本身，是不是最顶层的Key。（这行代码只是判断的该类型是否是超类的子类。所以真正拿subKey的逻辑在强转，如果类型转化成功，那么就说明是子Key，没成功说明不是，直接返回）。

自定义CoroutineContext

这里可以模仿已知的几种CoroutineContext来实现自定义

比如

- CoroutineExceptionHandler
- CoroutineDispatcher
- CoroutineName
- NonCancellable
- CoroutineId
- Job

以后会发现有两个共同点

- 父类都是AbstractCoroutineContextElement
- 都有一个名为Key的伴生对象，直接或者间接的实现了CoroutineContext.Key

这里拿两个做对比一个是CoroutineName

```
public data class CoroutineName(
    val name: String
) : AbstractCoroutineContextElement(CoroutineName) {
    public companion object Key : CoroutineContext.Key<CoroutineName>
    override fun toString(): String = "CoroutineName($name)"
}
```

CoroutineName是最标准的实现类

还有一个比较特殊的，那就是CoroutineDispatcher。

```
public abstract class CoroutineDispatcher :
    AbstractCoroutineContextElement(ContinuationInterceptor),
ContinuationInterceptor {
    @ExperimentalStdlibApi
    public companion object Key :
AbstractCoroutineContextKey<ContinuationInterceptor,
CoroutineDispatcher>(
        ContinuationInterceptor,
        { it as? CoroutineDispatcher })
}
```

它又伴生对象，But，伴生对象不是实现的CoroutineContext.Key而是AbstractCoroutineContextKey，这主要源于

CoroutineDispatcher有很多的元素，比如Dispachers.Main,EventLoop,ExecutorCoroutineDispatcher，....毫无疑问我们可以使用CoroutineDispatcher.Key作为他们的key，but，有的时候我们想更为精准地从CoroutineContext这个集合里面拿取我们想要地元素，比如有这么一个类。

```
public abstract class ExecutorCoroutineDispatcher: CoroutineDispatcher()
```

如果我们要在能使用CoroutineDispatcher.Key拿这个实例的基础上，还使用ExecutorCoroutineDispatcher.Key拿实例怎么办？（也就是说我想通过两个不同的key来拿这个实例）

这时候直接使用CoroutineContext.Key是不行的。

然而AbstractCoroutineContextKey是支持的。

因为你会发现它内部传入了两个泛型类型

```
public abstract class AbstractCoroutineContextKey<B : Element, E : B>
```

B以及E，B是BaseKey，E是依附于BaseKey的子key。也就是说你可以用BaseKey去拿这个实例，也可以用子Key去拿。

```
fun main() {
    val x = Executors.newFixedThreadPool(3).asCoroutineDispatcher() +
CoroutineName("")
    println(x[CoroutineDispatcher.Key])
    println(x[ExecutorCoroutineDispatcher])

}
```

> *java.util.concurrent.ThreadPoolExecutor@5387f9e0[Running, pool size = 0, active threads = 0, queued tasks = 0, completed tasks = 0]*
> *java.util.concurrent.ThreadPoolExecutor@5387f9e0[Running, pool size = 0, active threads = 0, queued tasks = 0, completed tasks = 0]*

两个自定义CoroutineContext的实例

```
data class CustomCoroutineContext1(
   val  content:String
) : AbstractCoroutineContextElement(Key) {
    companion object Key : CoroutineContext.Key<CustomCoroutineContext1>
}
```

```
data class CustomCoroutineContext2(
    val content: String,
):AbstractCoroutineContextElement(Key),MyCustomCoroutine{
    @OptIn(ExperimentalStdlibApi::class)
    override operator fun <E : CoroutineContext.Element> get(key:
CoroutineContext.Key<E>): E? {
        return getPolymorphicElement(key)
    }
}

interface MyCustomCoroutine : CoroutineContext.Element{
    companion object Key:CoroutineContext.Key<MyCustomCoroutine>
}
```

总结

自定义可以通过使用这两种，but并不代表只有这两种方式。

他们的区别很明显，一个只能使用一个Key索引。

一个是既可以使用BaseKey也可以使用子Key进行索引。

CoroutineContextElement——Source

> *Time*：*2022-1-24*

ContinuationInterceptor

之所以讲这个是因为它作用很强大。就好比网络请求的拦截器一样，有妙用。

简单使用

```
class LogInterceptor : ContinuationInterceptor {

    override val key: CoroutineContext.Key<*>
        get() = ContinuationInterceptor

    override fun <T> interceptContinuation(continuation:
Continuation<T>): Continuation<T> {
        return LogContinuation(continuation)
    }
}

class LogContinuation<T> (private val continuation:Continuation<T>):
Continuation<T> by continuation {
```

```
    override fun resumeWith(result: Result<T>) {
        println("resume前面")
        continuation.resumeWith(result)
        println("resume后")
    }
}
```

```
fun main() {
    suspend {

        delay(1000)
        delay(2000)
        delay(3000)

    }.startCoroutine(object : Continuation<Unit>{
        override val context: CoroutineContext
            get() = LogInterceptor()

        override fun resumeWith(result: Result<Unit>) {
            println("finished")
        }

    })

    Thread.sleep(100000)

}
```

这样就完成了resume的拦截，每次resume都会先调用resumeWith方法。

拦截流程分析

这个Continuation是什么时候被拦截了？在create之后start之前。

stdlib的几个suspend函数的扩展都给出了对应的原理，先create然后拦截，然后resume。

所以对于拦截的操作是基础框架在开启协程之前就做了的。

```
public fun <T> (suspend () -> T).startCoroutine(
    completion: Continuation<T>
) {
    createCoroutineUnintercepted(completion).intercepted().resume(Unit)
}
```

```
public fun <R, T> (suspend R.() -> T).createCoroutine(
    receiver: R,
    completion: Continuation<T>
): Continuation<Unit> =
    SafeContinuation(createCoroutineUnintercepted(receiver,
completion).intercepted(), COROUTINE_SUSPENDED)
```

```
public fun <T> (suspend () -> T).startCoroutineCancellable(completion:
Continuation<T>): Unit = runSafely(completion) {

 createCoroutineUnintercepted(completion).intercepted().resumeCancellabl
eWith(Result.success(Unit))
}
```

ContinuationInterceptor分析

```
public interface ContinuationInterceptor : CoroutineContext.Element {
    The key that defines the context interceptor.
    companion object Key : CoroutineContext.Key<ContinuationInterceptor>

    /** Returns continuation that wraps the original [continuation], thus intercepting all resumptions. ...*/
    public fun <T> interceptContinuation(continuation: Continuation<T>): Continuation<T>

    /** Invoked for the continuation instance returned by [interceptContinuation] when the original ...*/
    public fun releaseInterceptedContinuation(continuation: Continuation<*>) {...}

    public override operator fun <E : CoroutineContext.Element> get(key: CoroutineContext.Key<E>): E? {...}


    public override fun minusKey(key: CoroutineContext.Key<*>): CoroutineContext {...}
}
```

内容不多，就定义一个Key，重写了get和minusKey方法，加入了两个拦截方法。

get & minusKey

为什么要重写?

为了适配子类多Key索引的情况，也就是适配AbstractCoroutineKey

```
public override operator fun <E : CoroutineContext.Element> get(key:
CoroutineContext.Key<E>): E? {
    @OptIn(ExperimentalStdlibApi::class)
    if (key is AbstractCoroutineContextKey<*, *>) {
        @Suppress("UNCHECKED_CAST")
        return if (key.isSubKey(this.key)) key.tryCast(this) as? E else
null
    }
    @Suppress("UNCHECKED_CAST")
```

```
    return if (ContinuationInterceptor === key) this as E else null
}


public override fun minusKey(key: CoroutineContext.Key<*>):
CoroutineContext {
    @OptIn(ExperimentalStdlibApi::class)
    if (key is AbstractCoroutineContextKey<*, *>) {
        return if (key.isSubKey(this.key) && key.tryCast(this) != null)
EmptyCoroutineContext else this
    }
    return if (ContinuationInterceptor === key) EmptyCoroutineContext
else this
}
```

逻辑不难，加一个判断，如果传入的Key是AbstractKey，就在试试this.key是不是子key。然后依据情况返回。

如果只是普通的Key那么就直接抄Element的默认实现。

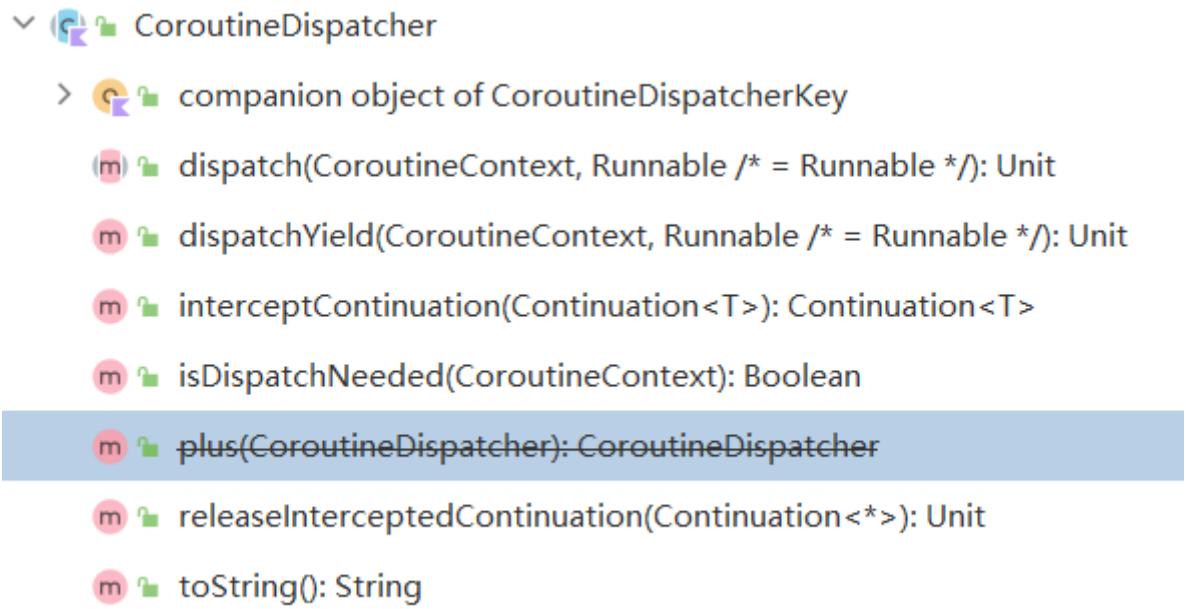
interceptContinuation & release...

进行拦截和进行释放的操作，实现是空的，需要实现类实现。

CoroutineDispatcher

如果你点开CoroutineDispatcher去查看它的父类的时候，你会发现，父类是ContinuationInterceptor，所以它其实也就是一个拦截器。

CoroutineDispatcher简要介绍



Base class to be extended by all coroutine dispatcher implementations.

The following standard implementations are provided by kotlinx.coroutines as properties on the Dispatchers object:

- Dispatchers.Default
- Dispatchers.IO
- Dispatchers.Unconfined
- Private thread pools can be created with newSingleThreadContext and newFixedThreadPoolContext.
- An arbitrary Executor can be converted to a dispatcher with the asCoroutineDispatcher extension function.

Dispatchers（调度器）的Base类，标准的实现有5种.

CoroutineDispatcher源码分析

Key

既然是自定义的Element那么总得有Key的

```
public companion object Key :
AbstractCoroutineContextKey<ContinuationInterceptor,
CoroutineDispatcher>(
    ContinuationInterceptor,
    { it as? CoroutineDispatcher })
```

这里选取的是抽象的。

isDispatchNeeded

> *Returns `true` if the execution of the coroutine should be performed with [dispatch] method. The default behavior for most dispatchers is to return `true`.*

return true表明需要调度，也就是说是在线程池里面跑的。（简单来说就是问问你要不要放入任务队列）

> *If this method returns `false`, the coroutine is resumed immediately in the current thread*

return false表示不需要，会直接在当前线程resume。

dispatch & dispatchYield

这就是runnable入任务队列。

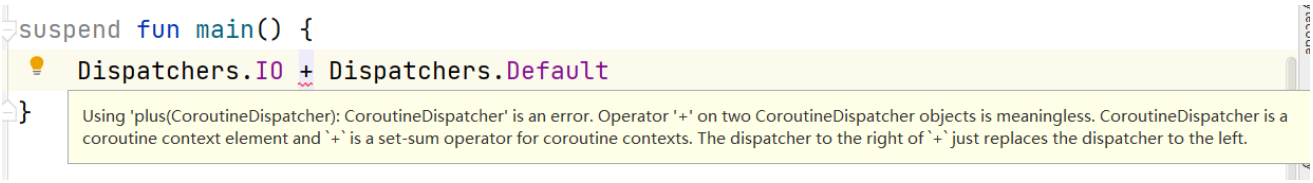
interceptContinuation & release...

```
public final override fun <T> interceptContinuation(continuation:
Continuation<T>): Continuation<T> =
    DispatchedContinuation(this, continuation)

public final override fun releaseInterceptedContinuation(continuation:
Continuation<*>) {
    /*
     * Unconditional cast is safe here: we only return
DispatchedContinuation from `interceptContinuation`,
     * any ClassCastException can only indicate compiler bug
     */
    val dispatched = continuation as DispatchedContinuation<*>
    dispatched.release()
}
```

拦截即包装一层DispatchedContinuation，释放即通知DispatchedContinuation释放。

plus

plus被标记为了废弃，因为右边的会覆盖左边的，没有任何意义。会报错。



Dispatchers

```
public expect object Dispatchers {

    public val Default: CoroutineDispatcher

    public val Main: MainCoroutineDispatcher

    public val Unconfined: CoroutineDispatcher
}
```

```
public actual object Dispatchers {
    @JvmStatic
    public actual val Default: CoroutineDispatcher =
createDefaultDispatcher()

    @JvmStatic
    public actual val Main: MainCoroutineDispatcher get() =
MainDispatcherLoader.dispatcher

    @JvmStatic
    public actual val Unconfined: CoroutineDispatcher =
kotlinx.coroutines.Unconfined

    @JvmStatic
    public val IO: CoroutineDispatcher = DefaultScheduler.IO
}
```

JVM的实现类加入了一个IO，不过看这几个已近大致知道分析哪几个类了。

Default

```
public actual val Default: CoroutineDispatcher = DefaultScheduler
```

DefaultScheduler

```
internal object DefaultScheduler : SchedulerCoroutineDispatcher(
    CORE_POOL_SIZE, MAX_POOL_SIZE,
    IDLE_WORKER_KEEP_ALIVE_NS, DEFAULT_SCHEDULER_NAME
) {
    // Shuts down the dispatcher, used only by Dispatchers.shutdown()
    internal fun shutdown() {
        super.close()
    }

    // Overridden in case anyone writes (Dispatchers.Default as
ExecutorCoroutineDispatcher).close()
    override fun close() {
        throw UnsupportedOperationException("Dispatchers.Default cannot
be closed")
    }

    override fun toString(): String = "Dispatchers.Default"
}
```

DefaultScheduler是一个SchedulerCoroutineDispatcher

然后它是ExecutorCoroutineDispatcher的一个子类

```
internal open class SchedulerCoroutineDispatcher(
    private val corePoolSize: Int = CORE_POOL_SIZE,
    private val maxPoolSize: Int = MAX_POOL_SIZE,
    private val idleWorkerKeepAliveNs: Long = IDLE_WORKER_KEEP_ALIVE_NS,
    private val schedulerName: String = "CoroutineScheduler",
) : ExecutorCoroutineDispatcher()
```

而ExecutorCoroutineDispatcher是一个抽象的Element（这架势一看就是线程池。）

```
public abstract class ExecutorCoroutineDispatcher:
CoroutineDispatcher(), Closeable {
    /** @suppress */
    @ExperimentalStdlibApi
    public companion object Key :
AbstractCoroutineContextKey<CoroutineDispatcher,
ExecutorCoroutineDispatcher>(
        CoroutineDispatcher,
        { it as? ExecutorCoroutineDispatcher })


    public abstract val executor: Executor


    public abstract override fun close()
}
```

线程池参数

```
internal val CORE_POOL_SIZE = systemProp(
    "kotlinx.coroutines.scheduler.core.pool.size",
    AVAILABLE_PROCESSORS.coerceAtLeast(2),
    minValue = CoroutineScheduler.MIN_SUPPORTED_POOL_SIZE
)
```

```
internal val MAX_POOL_SIZE = systemProp(
    "kotlinx.coroutines.scheduler.max.pool.size",
    CoroutineScheduler.MAX_SUPPORTED_POOL_SIZE,
    maxValue = CoroutineScheduler.MAX_SUPPORTED_POOL_SIZE
)
```

```
internal val IDLE_WORKER_KEEP_ALIVE_NS = TimeUnit.SECONDS.toNanos(
    systemProp("kotlinx.coroutines.scheduler.keep.alive.sec", 60L)
)
```

```
internal const val DEFAULT_SCHEDULER_NAME = "DefaultDispatcher"
```

除了名字不能变其余都可以。

核心实现

核心实现都交给了SchedulerCoroutineDispatcher

然而它又丢锅给了coroutineScheduler，

```
private fun createScheduler() =
    CoroutineScheduler(corePoolSize, maxPoolSize, idleWorkerKeepAliveNs,
schedulerName)
```

```
internal class CoroutineScheduler(
    @JvmField val corePoolSize: Int,
    @JvmField val maxPoolSize: Int,
    @JvmField val idleWorkerKeepAliveNs: Long =
IDLE_WORKER_KEEP_ALIVE_NS,
    @JvmField val schedulerName: String = DEFAULT_SCHEDULER_NAME
) : Executor, Closeable
```

而它就是个封装好的线程池。

Unconfined

```
public actual val Unconfined: CoroutineDispatcher =
kotlinx.coroutines.Unconfined
```

```
internal object Unconfined : CoroutineDispatcher() {

    @ExperimentalCoroutinesApi
    override fun limitedParallelism(parallelism: Int):
CoroutineDispatcher {
        throw UnsupportedOperationException("limitedParallelism is not
supported for Dispatchers.Unconfined")
    }

    override fun isDispatchNeeded(context: CoroutineContext): Boolean =
false

    override fun dispatch(context: CoroutineContext, block: Runnable) {
        /** It can only be called by the [yield] function. See also cold
of [yield] function. */
        val yieldContext = context[YieldContext]
        if (yieldContext != null) {
            // report to "yield" that it is an unconfined dispatcher and
don't call "block.run()"
            yieldContext.dispatcherWasUnconfined = true
```

```
            return
        }
        throw
UnsupportedOperationException("Dispatchers.Unconfined.dispatch function
can only be used by the yield function. " +
            "If you wrap Unconfined dispatcher in your cold, make sure
you properly delegate " +
            "isDispatchNeeded and dispatch calls.")
    }

    override fun toString(): String = "Dispatchers.Unconfined"
}
```

关于它的描述只有一句话

> *A coroutine dispatcher that is not confined to any specific thread*

这玩意不是线程池。

Dispatchers.Unconfined.dispatch function can only be used by the yield function.If you wrap Unconfined dispatcher in your code, make sure you properly delegate isDispatchNeeded and dispatch calls.

他是为Yield而生的。

IO

```
public val IO: CoroutineDispatcher = DefaultIoScheduler
```

```
internal object DefaultIoScheduler : ExecutorCoroutineDispatcher(),
Executor {

    private val default = UnlimitedIoScheduler.limitedParallelism(
        systemProp(
            IO_PARALLELISM_PROPERTY_NAME,
            64.coerceAtLeast(AVAILABLE_PROCESSORS)
        )
    )

    override val executor: Executor
        get() = this

    override fun execute(command: java.lang.Runnable) =
dispatch(EmptyCoroutineContext, command)
```

```
    @ExperimentalCoroutinesApi
    override fun limitedParallelism(parallelism: Int):
CoroutineDispatcher {
        // See documentation to Dispatchers.IO for the rationale
        return UnlimitedIoScheduler.limitedParallelism(parallelism)
    }

    override fun dispatch(context: CoroutineContext, block: Runnable) {
        default.dispatch(context, block)
    }

    @InternalCoroutinesApi
    override fun dispatchYield(context: CoroutineContext, block:
Runnable) {
        default.dispatchYield(context, block)
    }

    override fun close() {
        error("Cannot be invoked on Dispatchers.IO")
    }

    override fun toString(): String = "Dispatchers.IO"
}
```

它委托给了一个线程池，然后这个线程池是LimitedDispatcher

```
public open fun limitedParallelism(parallelism: Int):
CoroutineDispatcher {
    parallelism.checkParallelism()
    return LimitedDispatcher(this, parallelism)
}
```

LimitedDispatcher又委托给了dispatcher

```
internal class LimitedDispatcher(
    private val dispatcher: CoroutineDispatcher,
    private val parallelism: Int
) : CoroutineDispatcher(), Runnable, Delay by (dispatcher as? Delay ?:
DefaultDelay)
```

所以锅给了UnlimitedIoScheduler

然后它丢锅给了

```
@InternalCoroutinesApi
override fun dispatchYield(context: CoroutineContext, block: Runnable) {
    DefaultScheduler.dispatchWithContext(block, BlockingContext, true)
}

override fun dispatch(context: CoroutineContext, block: Runnable) {
    DefaultScheduler.dispatchWithContext(block, BlockingContext, false)
}
```

所以实际上还是DefaultScheduler

WithContext——Source

> *Time: 2022-1-26*

withContext可以做到切换线程然后切换回来

测试代码

```
fun main() {
    GlobalScope.launch {
        println(Thread.currentThread().name)
        withContext(Dispatchers.IO){
            delay(1000)
        }
        println(Thread.currentThread().name)
    }
    Thread.sleep(100000)
}
```

执行结果

> *DefaultDispatcher-worker-1*
> *DefaultDispatcher-worker-1*

执行流程分析

GlobalScope.launch

```
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
}
```

加了一个newCoroutineContext 会确保CoroutineContext内部包含一个Dispatcher，如果没有Dispatcher，会默认添加一个Dispatchers.Default.

然后new了一个StandaloneCoroutine

然后通过start开启协程，最后调用了block.startCoroutineCancellable(completion)

```
public fun <T> (suspend () -> T).startCoroutineCancellable(completion:
Continuation<T>): Unit = runSafely(completion) {

 createCoroutineUnintercepted(completion).intercepted().resumeCancellabl
eWith(Result.success(Unit))
}
```

createCoroutineUnintercepted确保了挂起函数具备挂起和恢复的能力，如果不具备会在外面包裹一层continuationImpl。

然后拦截。

```
public actual fun <T> Continuation<T>.intercepted(): Continuation<T> =
    (this as? ContinuationImpl)?.intercepted() ?: this

public fun intercepted(): Continuation<Any?> =
        intercepted
            ?:
(context[ContinuationInterceptor]?.interceptContinuation(this) ?: this)
                .also { intercepted = it }
```

这段代码之前点开很多次。but以前不明白到底干了啥，现在知道了。Dispatcher.Default是DefaultScheduler是CoroutineDispatcher

```
public final override fun <T> interceptContinuation(continuation:
Continuation<T>): Continuation<T> =
    DispatchedContinuation(this, continuation)
```

这下明了了，原来是包裹了一层DispatchedContinuation

然后resumeCancellableWith(Result.success(Unit))

```
public fun <T> Continuation<T>.resumeCancellableWith(
    result: Result<T>,
    onCancellation: ((cause: Throwable) -> Unit)? = null
): Unit = when (this) {
    is DispatchedContinuation -> resumeCancellableWith(result,
onCancellation)
    else -> resumeWith(result)
}
```

```
inline fun resumeCancellableWith(
    result: Result<T>,
    noinline onCancellation: ((cause: Throwable) -> Unit)?
) {
    val state = result.toState(onCancellation)
    if (dispatcher.isDispatchNeeded(context)) {
        _state = state
        resumeMode = MODE_CANCELLABLE
        dispatcher.dispatch(context, this)
    } else {
        executeUnconfined(state, MODE_CANCELLABLE) {
            if (!resumeCancelled(state)) {
                resumeUndispatchedWith(result)
            }
        }
    }
}
```

肯定会进入if 因为CoroutineDispatcher默认是直接返回true，然后调用了dispatch，这里dispatcher是Dispatchers.Default。（具体的实现在CoroutineScheduler）

1.6.0和1.5.X的实现或许有些不同

```
fun dispatch(block: Runnable, taskContext: TaskContext =
NonBlockingContext, tailDispatch: Boolean = false) {
    trackTask() // this is needed for virtual time support
    val task = createTask(block, taskContext)
```

```
    // try to submit the task to the local queue and act depending on
the result
    val currentWorker = currentWorker()
    val notAdded = currentWorker.submitToLocalQueue(task, tailDispatch)
    if (notAdded != null) {
        if (!addToGlobalQueue(notAdded)) {
            // Global queue is closed in the last step of close/shutdown
-- no more tasks should be accepted
            throw RejectedExecutionException("$schedulerName was
terminated")
        }
    }
    val skipUnpark = tailDispatch && currentWorker != null
    // Checking 'task' instead of 'notAdded' is completely okay
    if (task.mode == TASK_NON_BLOCKING) {
        if (skipUnpark) return
        signalCpuWork()
    } else {
        // Increment blocking tasks anyway
        signalBlockingWork(skipUnpark = skipUnpark)
    }
}
```

这里它直接放入任务队列了。

withContext

```
public suspend fun <T> withContext(
    context: CoroutineContext,
    block: suspend CoroutineScope.() -> T
): T {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return suspendCoroutineUninterceptedOrReturn sc@ { uCont ->
        // compute new context
        val oldContext = uCont.context
        val newContext = oldContext + context
        // always check for cancellation of new context
        newContext.ensureActive()
        // FAST PATH #1 -- new context is the same as the old one
        if (newContext === oldContext) {
            val coroutine = ScopeCoroutine(newContext, uCont)
            return@sc coroutine.startUndispatchedOrReturn(coroutine,
block)
        }
```

```
        // FAST PATH #2 -- the new dispatcher is the same as the old one
(something else changed)
        // `equals` is used by design (see equals implementation is
wrapper context like ExecutorCoroutineDispatcher)
        if (newContext[ContinuationInterceptor] ==
oldContext[ContinuationInterceptor]) {
            val coroutine = UndispatchedCoroutine(newContext, uCont)
            // There are changes in the context, so this thread needs to
be updated
            withCoroutineContext(newContext, null) {
                return@sc coroutine.startUndispatchedOrReturn(coroutine,
block)
            }
        }
        // SLOW PATH -- use new dispatcher
        val coroutine = DispatchedCoroutine(newContext, uCont)
        block.startCoroutineCancellable(coroutine, coroutine)
        coroutine.getResult()
    }
}
```

立即拦截当前的Continuation，然后new一个context，进行了判断。

两条fast path，

- 一newContext和oldContext一致。
- 二他们context不一致but他们的调度器一致。

有共同点就是他们的调度器一致，也就是是他们运行的线程池，又或说他们运行的线程是一致的。所以没必要进行线程的切换。

所以它直接要门直接运行（环境一致），要么在当前线程运行，but会在给定的上下文运行（调度器一致但是Context中有其他不一致的东西）。

```
val coroutine = DispatchedCoroutine(newContext, uCont)
block.startCoroutineCancellable(coroutine, coroutine)
coroutine.getResult()
```

满路径就没办法了，因为必须切线程所以new了一个DispatchedCoroutine

然后调用了它的核心的一点就是它这里套了一层DispatchedCoroutine也就是说他把自己的执行环境保存了下来。这样经过调用startCoroutineCancellable包装一层DispatchedContinuation这样就在DispatchedCoroutine中跑起来了。

然后等任务结束以后，又到了GlobalScope的DispatchedContinuation里面执行。这样就完成了一次线程切换。

Flow

使用

Flow是非阻塞的

```
fun simple() = flow<Int>{
    repeat(3){
        delay(300)
        emit(it)
    }
}

fun main(): Unit = runBlocking {
    launch {
        repeat(10){
            println("I am not blocked")
            delay(100)
        }
    }
    simple().collect {
        println("get:${it}")
    }
}
```

> *I am not blocked*
> *I am not blocked*
> *I am not blocked*
> *get:0*
> *I am not blocked*
> *I am not blocked*
> *I am not blocked*
> *get:1*
> *I am not blocked*
> *I am not blocked*
> *I am not blocked*
> *get:2*
> *I am not blocked*
>
> *Process finished with exit code 0*

Flow可以用flow来构建

flow是冷流

```
fun main() {
    simple()
}

fun simple() = flow<Int> {
    repeat(10){
        println("emit $it")
        emit(it)
        delay(1000)
    }
}
```

什么都没打印

流的取消

```
fun main(): Unit = runBlocking {
    withTimeoutOrNull(1000){
        flow<Int> {
            repeat(20){
                delay(100)
                emit(it)
            }
        }.collect {
            println(it)
        }
    }
}
```

withTimeOutOrNull执行给定时间后取消

流的构建器

- asFlow
- flow

转化器

```
fun main(): Unit = runBlocking {
    flow<Int> {
        emit(1)
        emit(2)
        emit(3)
    }
        .transform {
            emit(it.toString())
        }.collect {
            println(it)
        }
}
```

```
flow<Int> {
    emit(1)
}.map {
    it.toString()
}.collect {
    println(it)
}
```

其他

- take
- reduce/collect(末端操作符)

流的上下文

```
fun main():Unit = runBlocking {
    withContext(coroutineContext){
        simple().collect {
            println(it)
        }
    }
}

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

流的上下文默认是在collect的协程

flow的内部是不允许切换的

```
fun simple(): Flow<Int> = flow {
    // 在流构建器中更改消耗 CPU 代码的上下文的错误方式
    withContext(Dispatchers.Default) {
        for (i in 1..3) {
            Thread.sleep(100) // 假装我们以消耗 CPU 的方式进行计算
            emit(i) // 发射下一个值
        }
    }
}

fun main() = runBlocking<Unit> {
    simple().collect { value -> println(value) }
}
```

这样会直接抛出一个异常

flowOn

flowOn可以改变flow的发送的协程即emit的协程

```
fun main(): Unit = runBlocking {
    simple().flowOn(Dispatchers.Default).collect {
        println(Thread.currentThread().name)
    }
}

fun simple() = flow<Int> {
    repeat(100) {
        delay(1000)
        println(Thread.currentThread().name)
        emit(it)
    }
}
```

缓冲

```
fun main(): Unit = runBlocking {
    simple().flowOn(Dispatchers.Default).collect {
        println(Thread.currentThread().name)
    }
}

fun simple() = flow<Int> {
    repeat(100) {
        delay(1000)
        println(Thread.currentThread().name)
        emit(it)
    }
}
```

默认的情况下缓冲的数量是64，缓冲的策略是挂起。

合并

conflat

```
fun simple() = flow<Int> {
    repeat(3) {
        delay(100)
        emit(it)
    }
}.conflate()
```

```
fun main() = runBlocking {
    simple()
        .collect {
            delay(300)
            println("get ${it}")
        }
}
```

哪取了最新的值。

collectLast

```
fun simple() = flow<Int> {
    repeat(10){
        delay(100)
        emit(it)
    }
}


fun main():Unit = runBlocking {
    simple()
        .collectLatest {
            delay(300)
            println(it)
        }
}
```

zip

zip可用于组合两个流中的值

和rxjava的zip类似

combine

combine类似于zip是一种组合，但是不一样的是他是直接合并当前还存在的值而不用一只等待。

flatMapConcat

展开流，不过是顺序展开

```
fun main(): Unit = runBlocking {
    flow<Int> {
        repeat(10) {
            delay(1000)
            emit(it)
        }
    }
        .flatMapConcat {
            produceFlow(it)
        }
        .collect {
            println(it)
        }
}


fun produceFlow(value: Int) = flow<String> {
    repeat(10) {
        delay(1000)
        emit(value.toString() + it.toString())
    }
}
```

flatMapMerge

并行发送，保证内容发送出去了，但是不一定有序

flatMapLastest

```
fun main(): Unit = runBlocking {
    flow<Int> {
        repeat(10) {
            emit(it)
            delay(100)
        }
    }
        .flatMapLatest{
            produceFlow(it)
        }
        .collect {
            println(it)
        }
}
```

```
fun produceFlow(value: Int) = flow<String> {
    repeat(10) {
        delay(1000)
        emit(value.toString() + it.toString())
    }
}
```

同样是把值进行合并，但是有些不同的是，它如果接受到了下一个值，就会把上一个给取消。

catch

```
flow<Int> {
    emit(1)
    emit(0)
    repeat(10) {
        emit(it + 1)
    }
}
    .map {
        1 / it
    }
    .catch { e ->
        println("error: ${e}")
        emit(1)
    }
    .collect {
        println("suscess:$it")
    }
```

catch捕捉上游异常

onComplete

launchIn

流的取消

对于繁忙的任务(CPU密集型)的任务，它或者只是单纯的阻塞住了，所以挂起这一特性无法对他进行取消。

这时候就可以利用

.cancellable()

Source

> *Time*：*2022-1-28*

flow和RxJava在用法上是类似的。

那么他们在设计上是否是类似的呢?

我的答案是神似。

测试代码

```
flow<Int> {
    println("producer:" + Thread.currentThread().name)
    emit(1)
    println("producer:" + Thread.currentThread().name)
    emit(2)
}
    .map {
        it
    }
    .map {
        it
    }
    .map {
        it
    }
    .collect {
        println("collect")
    }
```

flow

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

将传入的高阶函数进行包裹。new了一个新的Flow对象

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()
        }
    }
}
```

所以最后就是new了一个Flow返回。

collect

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

调用了Flow的collect方法。

流程分析

首先完成构建，然后进行collect。

but，好像思路断了。感觉仅仅这样的话好像还真的完成不了流式的api调用。实际上是可以的，我们可以好好审视一下collect

```
collect(object : FlowCollector<T> {
        override suspend fun emit(value: T) = action(value)
    })
```

collect有调用了collect，也就是this.collect,

而this是Flow，它是在map的时候new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()
        }
    }
}
```

找到了它直接调用了外出的block，然后我们继续跟踪

```
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)
    }
}
```

然后它有调用了collect，递归是吧。

ok我懂。

然后呢到了最顶层的flow了。

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

SafeFlow好像是没有collect呢，不急它的collect被覆写了。

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

在collector上包裹了一层SafeCollector，然后调用了collectSafely...

然后flow这个lambda被执行了。

```
flow<Int> {
    println("producer:" + Thread.currentThread().name)
    emit(1)
    println("producer:" + Thread.currentThread().name)
    emit(2)
}
```

然后执行到了emit，emit嗯。

这个emit调用的是collector的。

这个collector是SafeCollector，

```
override suspend fun emit(value: T) {
    return suspendCoroutineUninterceptedOrReturn sc@{ 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
        }
    }
}
```

```
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>)
}
```

调用到了emit(uCont: Continuation, value: T)然后调用了emitFun

emitFun?

```
private val emitFun =
    FlowCollector<Any?>::emit as Function3<FlowCollector<Any?>, Any?,
Continuation<Unit>, Any?>
```

乍一看好像什么头绪都没，但是你看它的参数你就知道了，也就是一函数类型的。

再看它的调用处。

```
emitFun(collector as FlowCollector<Any?>, value, this as
Continuation<Unit>)
```

感觉就是直接调用了collector.emit，说实话我不是很懂为啥它不直接collector.emit.

然后跑到了下游的emit中去了，而下游的collector实在collect的时候new出来了的。

```
collect(object : FlowCollector<T> {
    override suspend fun emit(value: T) = action(value)
})
```

这样说下游的emit就是这个了，调用了一个高阶函数

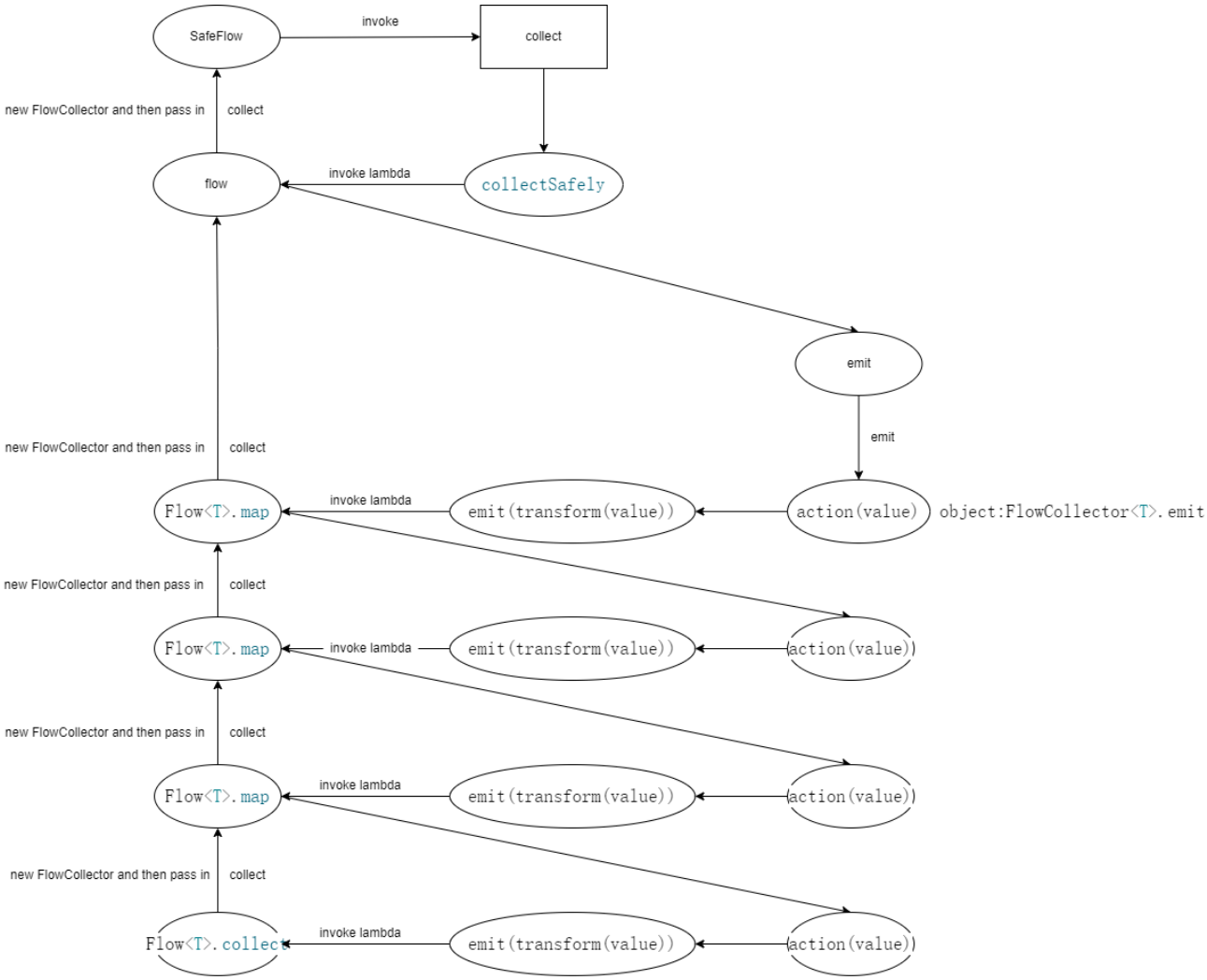
层层外传，最后就调用了这里

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

执行了map的lambda然后又调用了下层的emit。

总结

Flow的流式api的实现比较简单，它利用了扩展函数在collect得时候建立订阅关系，然后从最上游，开始一级级地先下游发送事件。这样就达到了rxjava流式api的效果。



Channel

> *Time*：*2022-1-30*

channel是通道，说通道或许过于陌生，但是如果说是阻塞队列或许就熟悉些。

延期的值提供了一种便捷的方法使单个值在多个协程之间进行相互传输。 通道提供了一种在流中传输值的方法。

使用

send/receive

```
fun main(): Unit = runBlocking {
    val channel = Channel<Int>()
    launch {
        for (x in 1..10){
            channel.send(x)
        }
    }

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

迭代/关闭 Channel

```
fun main(): Unit = runBlocking {
    val channel = Channel<Int>()
    launch {
        for (x in 1..10) {
            channel.send(x)
        }
        channel.close()
    }

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

    println("Done")
}
```

producer/consumer

```
fun main(): Unit = runBlocking {
    val receiveChannel = produce<Int> {
        repeat(10) { send(it) }
    }

    receiveChannel.consumeEach {
        println(it)
    }
}
```

带有缓冲通道的通道

这里设置bufferSize为10也就是说发送的数据会先进入这个缓冲队列中，如果缓存队列没有满就不会把它给阻塞掉。（如果把bufferSize改成9个这就会直接挂起，或者发送数据数目改成11个。）

```
fun main(): Unit = runBlocking {
    val channel = Channel<Int>(10)
    repeat(10) {
        channel.send(it)
    }
    channel.close()
    println("send finish")
    for (i in channel) {
        println(i)
    }
}
```

ticker

计时器

```
fun main(): Unit = runBlocking {
    val ticker = ticker(100, 0)
    var nextElement = withTimeoutOrNull(1) { println(ticker.receive()) }
    nextElement = withTimeoutOrNull(50) { println(ticker.receive()) }
    withTimeoutOrNull(50) { println(ticker.receive()) }
}
```

这个计时器，先发送一个初始值，然后依据间隔发送值（Unit）。初始值得发送时间和间隔得发送时间都是可以自定义的。

Channel——Source

测试代码

```
fun main():Unit = runBlocking {
    val chan = Channel<Int>()
    launch {
        for (i in 0..9){
            delay(100)
            chan.send(i)
        }
        chan.close()
    }
    launch {
        repeat(10){
            println(chan.receive())
        }
    }
}
```

流程分析

channel属于是一个接口

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

一个具有sendChannel实现和ReceiveChannel实现的接口。也就是说既可以接受数据，也可以发送数据。（其实分析的核心也就这两个函数）

channel创建

测试代码是使用Channel来创建的 。

```
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)
        }
    }
```

肉眼可见它进行了一些的适配

这里我们使用了默认，所以会直接返回一个RendezvousChannel

这玩意只是一个AbstractChannel的实现类。代码呢都还在抽象类里面。

```
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
}
```

send

```
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)
}
```

send的代码不多，这段是拿取任务队列的数据，如果拿到了，就把element传入然后resume了receive。

```
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
        }
    }
}
```

没在任务队列里面拿到东西就就把自己挂起，放入任务队列。

```
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")
        }
    }
}
```

receive

```
public final override suspend fun receive(): E {
    // fast path -- try poll non-blocking
    val result = pollInternal()

    @Suppress("UNCHECKED_CAST")
    if (result !== POLL_FAILED && result !is Closed<*>) return result as
E
    // slow-path does suspend
    return receiveSuspend(RECEIVE_THROWS_ON_CLOSE)
}
```

代码不多，先调用了pollInternal去拿队列里面的元素。

```
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
        }

        send.undeliveredElement()
    }
}
```

如果没有调用send的话这里会返回一个null，就直接返回了，然后如果不为空，就会在这个send里面加receive，等send去invoke suspend。

好吧先回到队列为空的情况看看，它退出了函数。调用了receiveSuspend(RECEIVE_THROWS_ON_CLOSE)

```
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
        }
    }
}
```

干了啥？也就是直接new了一个element入队列，会发现好像send和receive的逻辑是一样的。

```
protected open fun enqueueReceiveInternal(receive: Receive<E>): Boolean
= if (isBufferAlwaysEmpty)
    queue.addLastIfPrev(receive) { it !is Send } else
    queue.addLastIfPrevAndIf(receive, { it !is Send }, { isBufferEmpty
})
```

小结

- Channel内有两个基础的概念一个就是消费着，一个是生产者。分别对应receive和send。它的实现其实很类似于一个阻塞队列（不阻塞的阻塞队列。。
- Channel内部有一个queue来保存生产者的生产信号和消费者的消费信号。
- 对于send，他会去拿队列里面的receive（消费信号），分两种情况一种是拿到了，一种没拿到。
    - 拿到了就把需要生产的值直接resume receive。
    - 如果没拿到那就把消费信号放入到队列中，等消费者来拿。然后挂起。
- 对于receive，类似的，他去拿send信号，也分两种情况
    - 如果拿到了就resume send
    - 如果没拿到就把receive信号放入到队列中，等待生产者来取。
- 所以对于send和receive来说都是先拿取信号。然后分两种情况，拿到了就invoke suspend，没拿到就把自己放入任务队列等待对方invoke自己。就相互invoke suspend 呗

Kotlin stdlib-coroutine

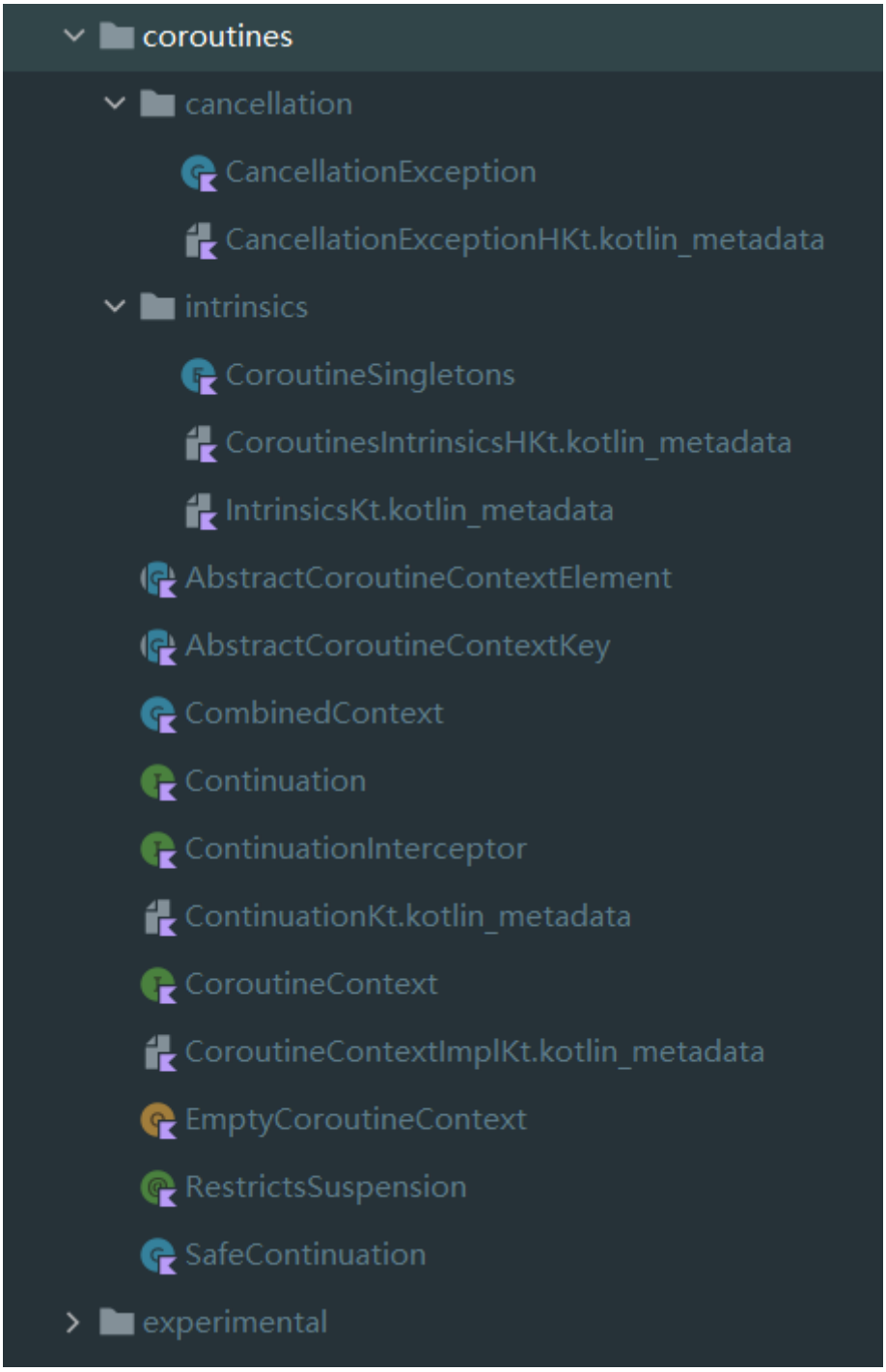
> *Time*： *2022 1-22*

Gradle: org.jetbrains.kotlin:kotlin-stdlib-common:1.5.31

为什么是分析stdlib-common因为kt是跨平台的，common是他们所具有的共性。（common库后又有jvm，js，native）

content

其实内容没有想的那么多（毕竟是common库嘛）



cancellation

这个包里面的文件为协程提供取消的支持

只有一个kt文件，CancellationException。

```
public expect open class CancellationException : IllegalStateException {
    public constructor()
    public constructor(message: String?)
}
public expect fun CancellationException(message: String?, cause:
Throwable?): CancellationException
public expect fun CancellationException(cause: Throwable?):
CancellationException
```

这是一个自定义的异常类。（协程取消是依靠抛出这个异常或者是这个异常的子类，在处理异常的过程中就把协程取消了。）比如这下面两个示例。

```
fun main() {
    runBlocking {
       val job = launch {
           try {
               delay(1000)
           } catch (e: Exception) {
               e.printStackTrace()
           }
       }
        delay(500)
        job.cancel()
    }
}
```

> *kotlinx.coroutines.JobCancellationException: StandaloneCoroutine was cancelled; job=StandaloneCoroutine{Cancelling}@2ac1fdc4*

```
fun main() {
    runBlocking {
        val job = async {
            delay(1000)
            throw CancellationException("我就是想取消协程")
            delay(1000)
            println("欸我没被取消")
        }
        try {
            job.await()
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }
}
```

> *java.util.concurrent.CancellationException:* 我就是想取消协程
> *at TestKt$main$1job1.invokeSuspend(Test.kt:14)*

intrinsics

这个包里面的内容是协程的一些特性

有一个kt文件

CoroutineSingletons

```
public suspend inline fun <T>
suspendCoroutineUninterceptedOrReturn(crossinline block:
(Continuation<T>) -> Any?): T {
    contract { callsInPlace(block, InvocationKind.EXACTLY_ONCE) }
    throw NotImplementedError("Implementation of
suspendCoroutineUninterceptedOrReturn is intrinsic")
}

public val COROUTINE_SUSPENDED: Any get() =
CoroutineSingletons.COROUTINE_SUSPENDED

internal enum class CoroutineSingletons { COROUTINE_SUSPENDED,
UNDECIDED, RESUMED }
```

协程的三种状态：未定，挂起，恢复就是对应的这个CoroutineSingletons枚举类。

suspendCoroutineUninterceptedOrReturn实力抢眼，你会发现它的实现是空的，他的实现是kt编译器利用asm插桩生成的。

它的主要作用就是获取挂起函数的continuation，要么立即挂起，要么立即返回值。

> *Obtains the current continuation instance inside suspend functions and either suspends currently running coroutine or returns result immediately without suspension*

其他

CoroutineContext

概述

CoroutineContext是一个数据类型（介于Map和Set之间），他在kt里面就是一个接口，提供了了一些基本的如add以及get以及remove方法。

其中

- public operator fun get —— get
- public fun minusKey(key: Key<*>) ——remove
- public operator fun plus(context: CoroutineContext) —— add

Element

此外CoroutineContext还为自己的元素定义了一个接口。
kotlin.coroutines.CoroutineContext.Element。

只要实现了这个接口就会CoroutineContext这个数据结构内的元素，element内聚合了一个Key（用于索引元素），对CoroutineContext的抽象方法给出了默认的实现。

```
public interface Element : CoroutineContext {
    public val key: Key<*>

    public override operator fun <E : Element> get(key: Key<E>): E? =
        @Suppress("UNCHECKED_CAST")
        if (this.key == key) this as E else null

    public override fun <R> fold(initial: R, operation: (R, Element) ->
R): R =
        operation(initial, this)

    public override fun minusKey(key: Key<*>): CoroutineContext =
        if (this.key == key) EmptyCoroutineContext else this
}
```

Key

```
public interface Key<E : Element>
```

Key是一个标志性接口，实现类表明它就可以作为索引CoroutineContext元素的‘下标’。（类似于数组的下标，hashMap的key）

值得注意的是每个Key需要绑定一个Element（每个Element相应的也绑定了一个Key）。双向绑定。

数据结构的抽象表示就这样完成了。

实现类

如果说实现的话其实kotlin.coroutines.CoroutineContext.Element也算是一种，虽然是接口but是有实现的接口。（和有理想的咸鱼，一个道理）

EmptyCoroutineContext

这是一个空的CoroutineContext，和EmptyList其实是一个道理。

CombinedContext

这个其实是大多数CoroutineContext的真正形态，EmptyCoroutineContext在加了两个独立的Element以后就成为了CombinedContext。

其他

工具

为了方便我们自己在必要的时候自己定义上下文做一些sao的操作，它提供了两个抽象

- AbstractCoroutineContextElement
- AbstractCoroutineContextKey

第一个是用于元素的自定义。

第二个嘛是用于创建一个特殊的Key。

一个好用的Element

什么Element需要在common里面抽象出来?

拦截器，准确来说是ContinuationInterceptor再准确来说是CoroutineContext.Element。

kotlin协程的‘协’字就体现在这里，他需要协作式的任务调度，也就是说，在执行任务的时候是这样的。

> 任务*A*：我执行了一会，有点累了，摸一下🐵，任务*B*你去吧。！
>
> 任务*B*：好的老哥~，我知道了，我去了。

传统的线程的执行就是抢占式的，如下

> 线程*A*：*CPU*给我执行，给我执行权！！
>
> 线程*B*：别给他，让我来！！

> 线程C：我抢到执行权了，你说气不气？

这不是说要让线程和协程做什么比较，其实他们就是一家子。前面分析到了协程的调度器其实就是一个线程池，kt的协程很大程度上是‘想’要替换对于开发者来说难用的线程池。（也不是说替换吧毕竟业务是变化的，but对于客户端来说基本上所有的任务使用协程都能优雅解决，叫做kotlin线程池更为贴切吧。）。

回归正题，ContinuationInterceptor实现了任务的拦截，相应的协作式的任务调度就可以通过拦截Continuation来实现。所以这玩意比较强大，所以在common库里面就抽离出来了

```
public interface ContinuationInterceptor : CoroutineContext.Element {

    companion object Key : CoroutineContext.Key<ContinuationInterceptor>

    public fun <T> interceptContinuation(continuation: Continuation<T>):
Continuation<T>

    public fun releaseInterceptedContinuation(continuation:
Continuation<*>)

    public override operator fun <E : CoroutineContext.Element> get(key:
CoroutineContext.Key<E>): E?

    public override fun minusKey(key: CoroutineContext.Key<*>):
CoroutineContext
}
```

Continuation

协程最大的优势就是挂起和恢复，挂起和恢复需要保存执行的一些状态，而Continuation为此诞生的。

who are you

```
public interface Continuation<in T> {

    public val context: CoroutineContext

    public fun resumeWith(result: Result<T>)

}
```

Continuation很简单，一个CoroutineContext就包含了所有的运行环境，然后resumeWith就类似于一个Thread.start(),这样他就在运行环境跑起来了，或者专业点说就是恢复执行。

实现

匿名的实现

```
public inline fun <T> Continuation(
    context: CoroutineContext,
    crossinline resumeWith: (Result<T>) -> Unit
): Continuation<T> =
    object : Continuation<T> {
        override val context: CoroutineContext
            get() = context

        override fun resumeWith(result: Result<T>) =
            resumeWith(result)
    }
```

SafeContinuation

之所以有SafeContinuation是因为Continuation不Safe，嗯，就是这样。

```
internal expect class SafeContinuation<in T> : Continuation<T> {
    internal constructor(delegate: Continuation<T>, initialResult: Any?)

    @PublishedApi
    internal constructor(delegate: Continuation<T>)

    @PublishedApi
    internal fun getOrThrow(): Any?

    override val context: CoroutineContext
    override fun resumeWith(result: Result<T>): Unit
}
```

不safe的原因是你可以任意次恢复恢复

```
fun main() {
    val continuation =  object : Continuation<Unit>{
         override val context: CoroutineContext
             get() = EmptyCoroutineContext

         override fun resumeWith(result: Result<Unit>) {
             println("我被恢复了")
         }
     }
     continuation.resume(Unit)
     continuation.resume(Unit)

}
```

不会报错的。

也就是说只要拿到了Continuation任务就是不确定的，比如执行了一个任务，在某个挂起点挂起了，你可以在这个挂起点任意的恢复，这样任务并发就会出现一些不可预知的问题，比如一个支付软件，就因为多次在支付处挂起了恢复扣了多次钱？这显然是荒谬的。拿SafeContinuation呢？怎么个Safe法。

```
suspend fun main() {
     test()
     println("我恢复了")
}

suspend fun test() = suspendCoroutine<Unit> {
     println("挂起了")
     it.resume(Unit)
     it.resume(Unit)
}
```

> *Exception in thread "main" java.lang.IllegalStateException: Already resumed*

第二次恢复会直接crash。所以安全了。

对于最为常见的挂起函数来说，挂起恢复多少次都是安全的，因为同一个挂起点只恢复了一次，程序是安全的。

但是如果continuation给你了，你resume了多次这就真的不安全了，因为同一挂起点挂起恢复了多次。程序的逻辑被完全打破了。

RestrictsSuspension

这是一个受限制的协程，是用于一些比价特殊的协程的作用域。

比如sequence，使用比较小众。

总结

协程的基础层只给出了如下简单的实现。

- suspend function/挂起，恢复
- CoroutineContext
- Continuation

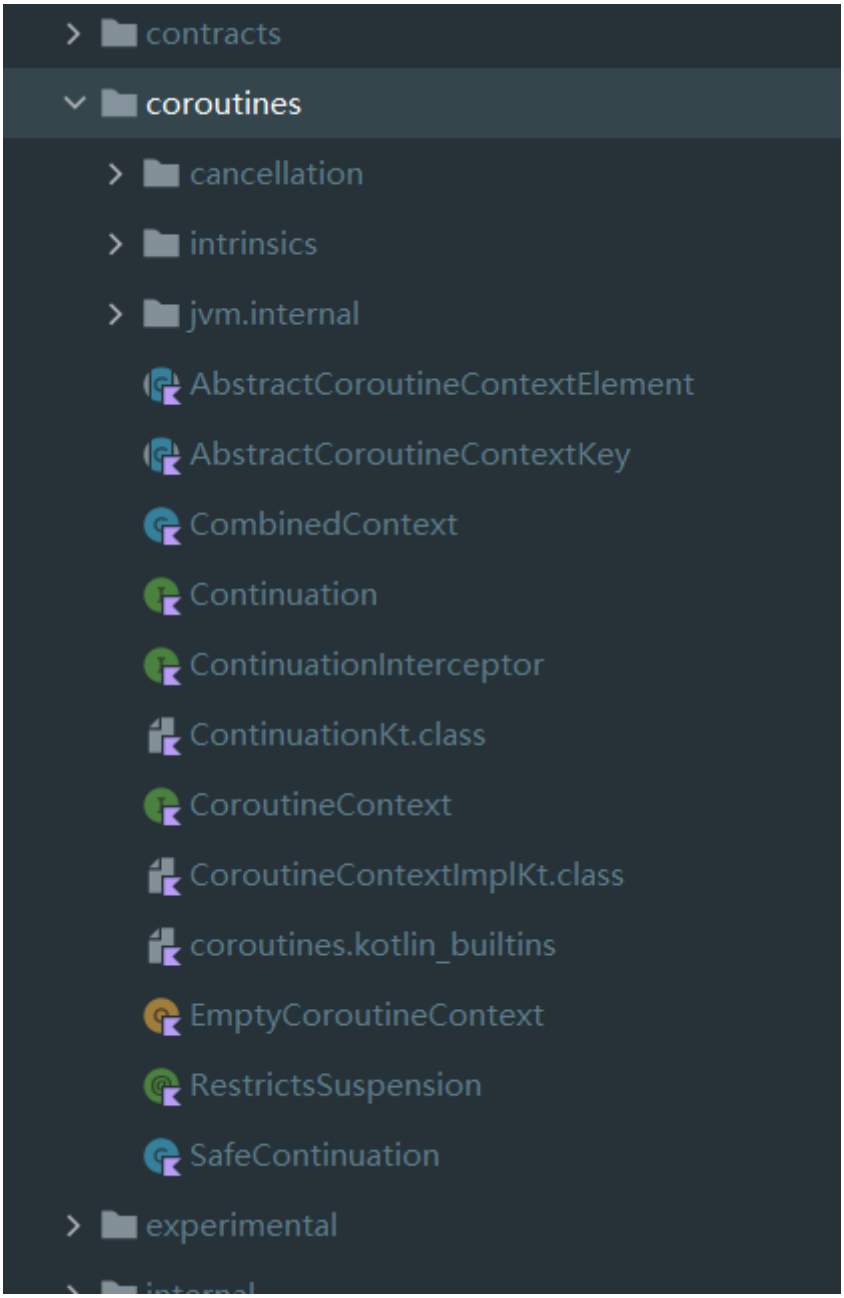
协程的具体实现都是围绕着上面的进行扩展的。

Kotlin stdlib coroutine JVM

> *Time : 2022 -1-22*

Gradle: org.jetbrains.kotlin:kotlin-stdlib:1.5.31

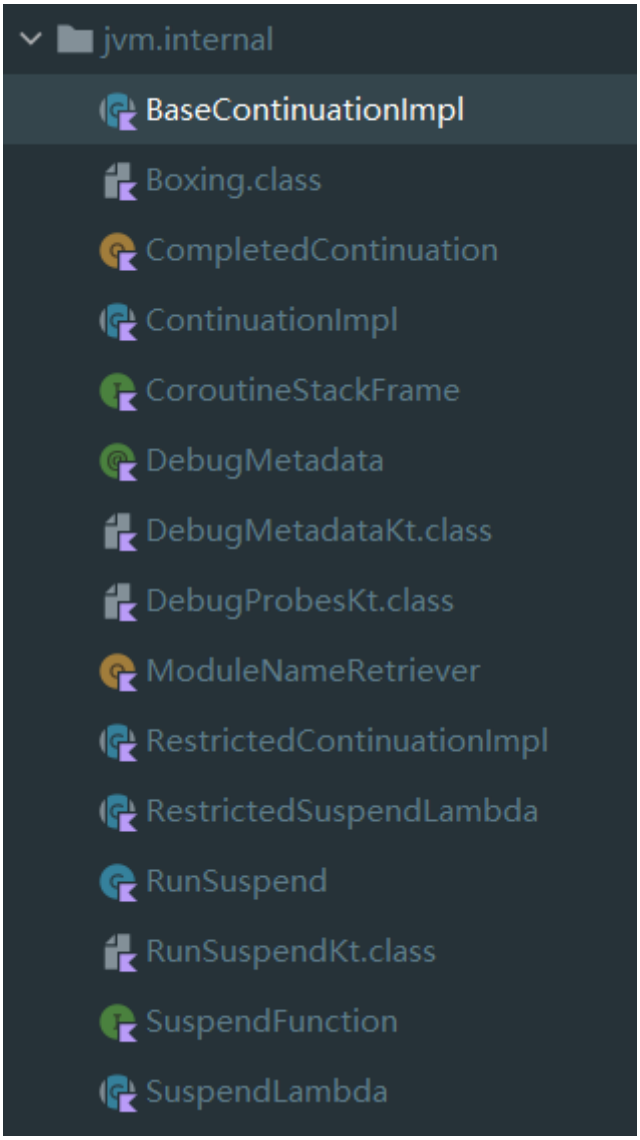
Content



乍一看好像和前面的base好像是差不多的。

简述

看上去有很多包，but相比于之前的stdlib-common只增加了一个jvm.internal,所以我们只需要分析jvm.internal 即可。



所以实际上JVM的实现只是这点东西。

初步分析以后发现JVM的实现主要是加了几种Continuation的实例。

RunSuspend

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

private class RunSuspend : Continuation<Unit> {
    override val context: CoroutineContext
        get() = EmptyCoroutineContext

    var result: Result<Unit>? = null
```

```
    override fun resumeWith(result: Result<Unit>) = synchronized(this) {
        this.result = result
        @Suppress("PLATFORM_CLASS_MAPPED_TO_KOTLIN") (this as
Object).notifyAll()
    }

    fun await() = synchronized(this) {
        while (true) {
            when (val result = this.result) {
                null -> @Suppress("PLATFORM_CLASS_MAPPED_TO_KOTLIN")
(this as Object).wait()
                else -> {
                    result.getOrThrow() // throw up failure
                    return
                }
            }
        }
    }
}
```

这个代码之前已经分析过了，主要是为suspend main提供的，因为JVM平台上没有挂起函数的概念的，所以更不会有Continuation这种东西。然则挂起函数只是普通函数+Continuation参数，所以为了兼容JVM平台，它想到的方法就是在编译期间new一个Continuation，那就是RunSupend了。

BaseContinuationImpl

common当时只是抽离了Continuation这个东西，but没有给出具体的实现，Continuation还是一个接口呢。

所以BaseContinuatonImpl是JVM平台上对于协程的Continuation的最基础的实现

```
internal abstract class BaseContinuationImpl(
    public val completion: Continuation<Any?>?
) : Continuation<Any?>, CoroutineStackFrame, Serializable {
    public final override fun resumeWith(result: Result<Any?>) {
        var current = this
        var param = result
        while (true) {
            probeCoroutineResumed(current)
            with(current) {
                val completion = 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()
                if (completion is BaseContinuationImpl) {
                    current = completion
                    param = outcome
                } else {
                    completion.resumeWith(outcome)
                    return
                }
            }
        }
    }

    protected abstract fun invokeSuspend(result: Result<Any?>): Any?
}
```

代码最基础的功能就是能跑就够了，Continuation既然系协程，那肯定得满足挂起和恢复，由于挂起是编译器自动生成的，所以不用管，那就只有恢复逻辑了，也就是resumeWith，resumeWith之前已经分析过了，也就是一层层的恢复。（continuation在执行的时候其实是类似于链表的串，所以得一层层地恢复）。but你会发现resumeWith并不是真的恢复了。真的恢复逻辑在invokeSuspend.然鹅invokeSuspend也是空的，所以挂起和恢复的最底层逻辑都是编译器生成的。

这里只有对于的调度逻辑。

ContinuationImpl\RestrictedContinuationImpl

```
internal abstract class ContinuationImpl(
    completion: Continuation<Any?>?,
    private val _context: CoroutineContext?
) : BaseContinuationImpl(completion) {
    constructor(completion: Continuation<Any?>?) : this(completion,
completion?.context)

    public override val context: CoroutineContext
        get() = _context!!

    @Transient
    private var intercepted: Continuation<Any?>? = null

    public fun intercepted(): Continuation<Any?> =
        intercepted
            ?:
(context[ContinuationInterceptor]?.interceptContinuation(this) ?: this)
                .also { intercepted = it }
```

```
    protected override fun releaseIntercepted() {
        val intercepted = intercepted
        if (intercepted != null && intercepted !== this) {

 context[ContinuationInterceptor]!!.releaseInterceptedContinuation(inter
cepted)
        }
        this.intercepted = CompletedContinuation // just in case
    }
}
```

```
internal abstract class RestrictedContinuationImpl(
    completion: Continuation<Any?>?
) : BaseContinuationImpl(completion) {
    init {
        completion?.let {
            require(it.context === EmptyCoroutineContext) {
                "Coroutines with restricted suspension must have
EmptyCoroutineContext"
            }
        }
    }

    public override val context: CoroutineContext
        get() = EmptyCoroutineContext
}
```

这两爷子都是继承自BaseContinuationImpl。

先看看ContinuationImpl，可以发现它只是在Base的基础上添加了拦截的操作。

如有context里面有ContinuationInterceptor那就调用ContinuationInterceptor的interceptContinuation方法，然后再返回，否则就返回自身。

发现了嘛加入了上下文拦截功能。

高大上的功能就这几行代码。

这个RestrictedContinuationImpl的实现就有些好笑了，之前没看出来Restricted究竟是那里受限制，没想到只是上下文只能是Empty。大道至简啊

协程拦截器使用

```
suspend fun main(){
    val y = withContext(MyContinuationInterceptor()){
        println("before delay")
        delay(1000)
        println("after delay ")
    }
}



class  MyContinuationInterceptor : ContinuationInterceptor{
    companion object Key :
CoroutineContext.Key<MyContinuationInterceptor>
    override val key: CoroutineContext.Key<*>
        get() = Key

    override fun <T> interceptContinuation(continuation:
Continuation<T>): Continuation<T> {
        println("这谁啊？  $continuation")
        return continuation
    }

}
```

SuspendFunction/SuspendLambda/RestrictedSuspendLambda

```
internal interface SuspendFunction
```

挂起函数和普通函数好像长得差不多啊是吧，所以怎么区分他们呢？这不就有一个标记接口嘛。只要实现你就是挂起函数

> *To distinguish suspend function types from ordinary function types all suspend function types shall implement this interface*

```
internal abstract class SuspendLambda(
    public override val arity: Int,
    completion: Continuation<Any?>?
) : ContinuationImpl(completion), FunctionBase<Any?>, SuspendFunction {
    constructor(arity: Int) : this(arity, null)

    public override fun toString(): String =
        if (completion == null)
            Reflection.renderLambdaToString(this) // this is lambda
        else
            super.toString() // this is continuation
}
```

这个就是挂起函数。

具有挂起函数标记，同时也是ContinuationImpl，还是FunctionBase的实现类

```
interface FunctionBase<out R> : Function<R> {
    val arity: Int
}
```

Suspension lambdas inherit from this class

这里又是kotlin编译器的魔法了

```
internal abstract class SuspendLambda(
    public override val arity: Int,
    completion: Continuation<Any?>?
) : ContinuationImpl(completion), FunctionBase<Any?>, SuspendFunction {
    constructor(arity: Int) : this(arity, null)

    public override fun toString(): String =
        if (completion == null)
            Reflection.renderLambdaToString(this) // this is lambda
        else
            super.toString() // this is continuation
}
```

是suspend lambda的标记

除此之外还有RestrictedSuspendLambda

```
internal abstract class RestrictedSuspendLambda(
    public override val arity: Int,
    completion: Continuation<Any?>?
) : RestrictedContinuationImpl(completion), FunctionBase<Any?>,
SuspendFunction {
    constructor(arity: Int) : this(arity, null)

    public override fun toString(): String =
        if (completion == null)
            Reflection.renderLambdaToString(this) // this is lambda
        else
            super.toString() // this is continuation
}
```

和前面的RestrictedContinuationImpl差不多。

其他

除了上面的内容之外的就是一些用于调试的东西了。

总结

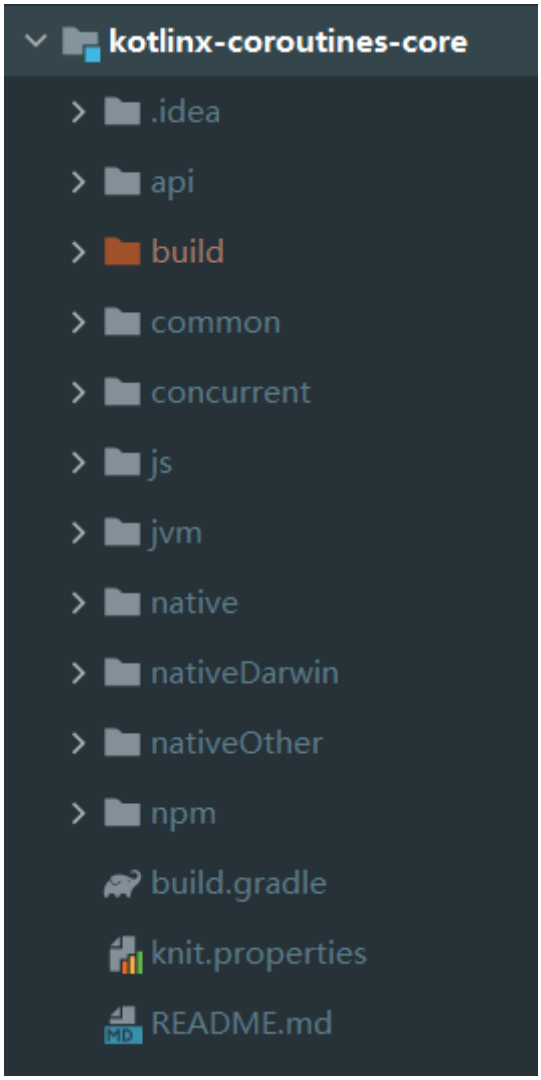
kotlin stdlib coroutine-JVM只是在common的基础上加上了几个Continuation使得协程‘完整’了，可以用了。but可以用还是不够的，要好用才行，而stdlib里给出的东西都太过于基层了，和C语言相较于Java一样，如果直接用于生产环境会有一些并发问题难以解决。（所以就有了kotlinx-coroutines-core也就是基于协程基础设施开发的框架。）

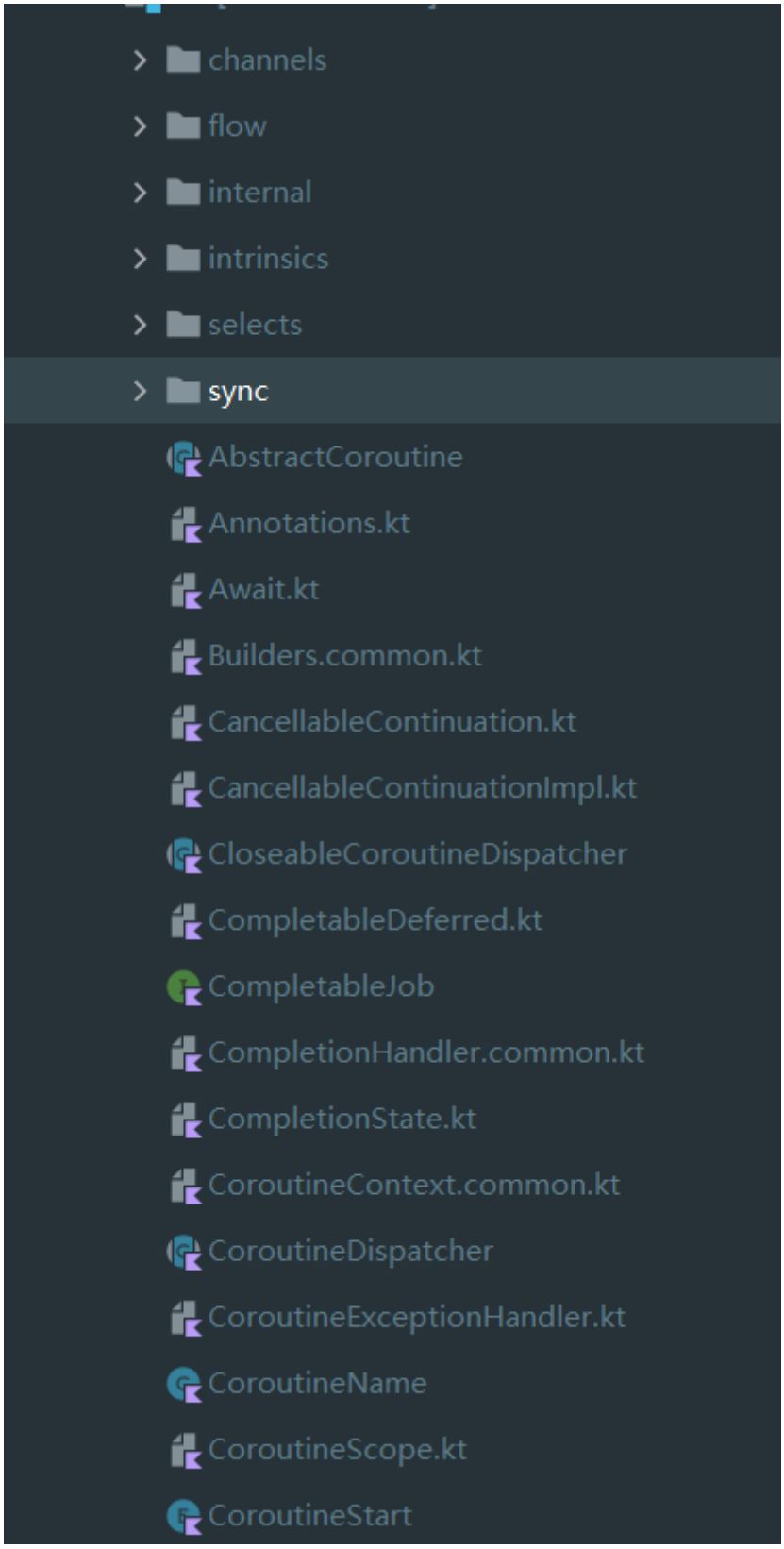
kotlinx-coroutines-core-common

> 先试试吧——试试就逝世

> 最近修改时间: 2022-1-23

千万不要尝试点开整个库。相信我你会崩溃的，内容是真的多。所以我clone了一份官方库。


kotlinx-coroutines-core
.idea
api
build
common
concurrent
js
jvm
native
nativeDarwin
nativeOther
npm
build.gradle
knit.properties
README.md


channels
flow
internal
intrinsics
selects
sync
AbstractCoroutine
Annotations.kt
Await.kt
Builders.common.kt
CancellableContinuation.kt
CancellableContinuationImpl.kt
CloseableCoroutineDispatcher
CompletableDeferred.kt
CompletableJob
CompletionHandler.common.kt
CompletionState.kt
CoroutineContext.common.kt
CoroutineDispatcher
CoroutineExceptionHandler.kt
CoroutineName
CoroutineScope.kt
CoroutineStart

没截完，剩下的就不截了。

Content

intrisics

里面有两个kt文件

Cancellable

```
@InternalCoroutinesApi
public fun <T> (suspend () -> T).startCoroutineCancellable(completion:
Continuation<T>): Unit = runSafely(completion) {

 createCoroutineUnintercepted(completion).intercepted().resumeCancellabl
eWith(Result.success(Unit))
}


internal fun <R, T> (suspend (R) -> T).startCoroutineCancellable(
    receiver: R, completion: Continuation<T>,
    onCancellation: ((cause: Throwable) -> Unit)? = null
) =
    runSafely(completion) {
        createCoroutineUnintercepted(receiver,
completion).intercepted().resumeCancellableWith(Result.success(Unit),
onCancellation)
    }

internal fun
Continuation<Unit>.startCoroutineCancellable(fatalCompletion:
Continuation<*>) =
    runSafely(fatalCompletion) {
        intercepted().resumeCancellableWith(Result.success(Unit))
    }

private inline fun runSafely(completion: Continuation<*>, block: () ->
Unit) {
    try {
        block()
    } catch (e: Throwable) {
        dispatcherFailure(completion, e)
    }
}

private fun dispatcherFailure(completion: Continuation<*>, e: Throwable)
{
```

```
        completion.resumeWith(Result.failure(e))
        throw e
    }
```

为协程提供了取消的支持

不过值得注意的是，这是个internal api，也就是说适合内部调用，库外调用有风险。需谨慎

Undispatched

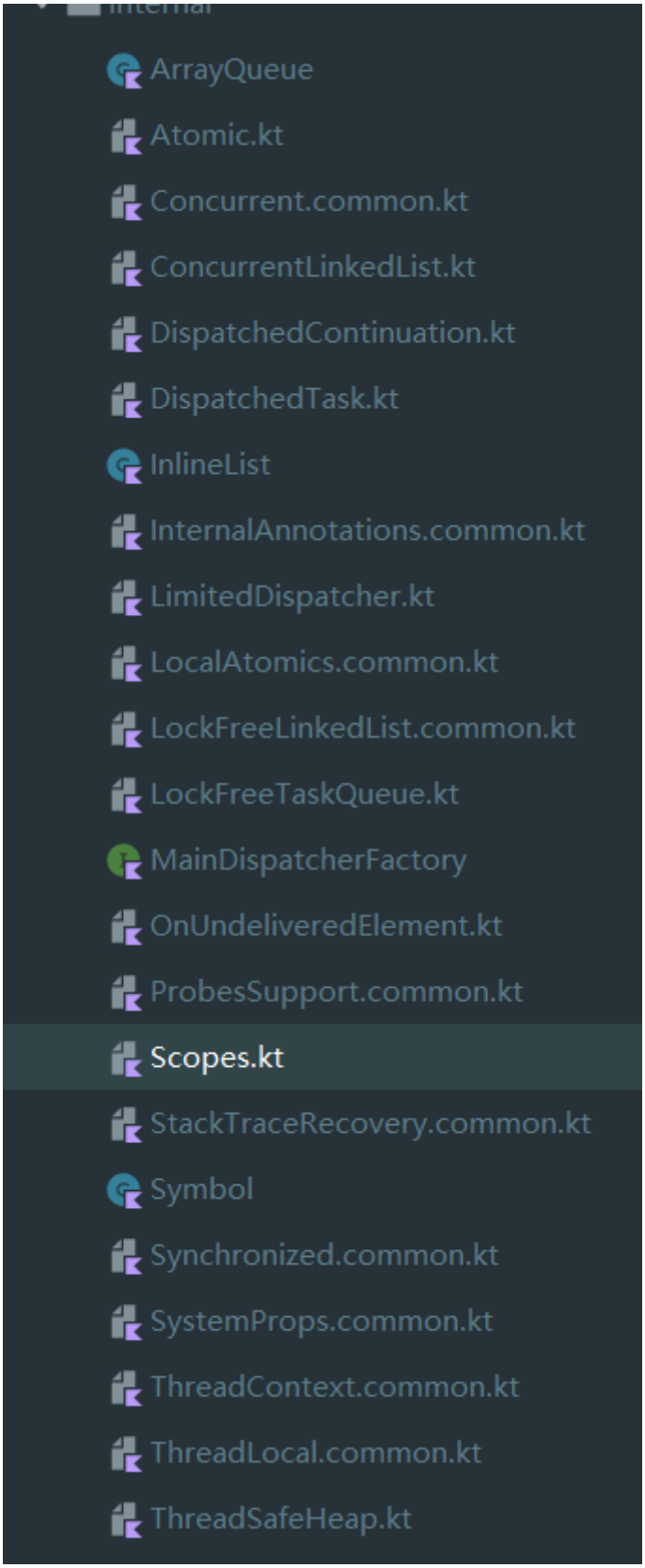
Undispatched就是不被调度的意思。也就是说直接在当前线程开始run。

也是写了几个扩展函数。

这里不展开讲了。

internal

内部的东西就比较多了



光从名称上来看很多都是与线程并发相关的一些东西，为解决线程安全的一些数据结构，或是工具类。

不过这里面有一些我们可能稍微接触较多的东西

- DispatchedTask/DispatchedContinuation

> 线程池嘛总是得要有*Task*是吧，协程有没有可能是一个线程池，如果是那么对应的*Task*又是什么？要想知道答案，请看*DispatchedTask*的父类。***internal actual typealias SchedulerTask = Task***。

- LimitedDispatcher/MainDispatcherFactory/MissingMainCoroutineDispatcher

> *Dispachers*我们是比较常用的东西，点看*Dispatchers.IO*,*Dispatcher.Default*,*Dispachers.Unconfinded*你会发现他们是*CoroutineDispatcher*的子类，然而*LimitedDispatcher*也是,欸你说巧不巧，*MainDispatcherFactory*/*MissingMainCoroutineDispatcher*,*Main*欸你说熟不熟悉，*Dispatches*也有个*Main*来着，我是说有没有可能他们存在一定联系呢？

- ScopeCoroutine/ContextScope

> 我隐约还记得*stdlib-common*库里面有几个比较重要的接口*Continuation*？*CoroutineContext*？如果你也记得那真是太棒了。
>
> 对于一个挂起函数而言，我们的整个函数体就相当于是单线程的，虽然他们是异步的操作。这解决了异步的问题（我是说挂起函数又或者说*Continuation*），*but*并发呢？任务的调度呢？任务的取消？异常的处理？好像还是空白呢。所以就会有*CoroutineScope*，他是一个域在这个域里面你可以对一切的任务进行管理，说停就停，说开始就开始，任务怎么并发，调度开发者可以进行有效的管理。而作为他们的子类有没有可能有一定的参考价值呢？说实话还真不一定有，*but*也真不一定没有。（我反正是知道了*CoroutineContext*，*CoroutineScope*，*Continuation*就*CoroutineContext*地位最低*hh*。）

- CommonThreadLocal

> *ThreadLocal*我懂*CommonThreadLocal*是啥我还真不懂，不过说不定有点关系呢。你说这是啥呢。
>
> ```
> internal actual typealias CommonThreadLocal<T> = ThreadLocal<T>
> ```

其他

CoroutineScope

- AbstractCoroutine/JobSupport/JobImpl

> 前面分析很多关于协程的东西，在*Java*中线程有一个类代指。*Thread*，相似的协程也有，*AbstractCoroutine*就是抽象概念上的协程。也就是说抽象概念上的协程是*Job+Continuation+CoroutineScope*的集合体。

-

- DeferredCoroutine/LazyDeferredCoroutine/StandaloneCoroutine/LazyStandaloneCoroutine

> 不点破的话很难说这两个是什么东西，点破又很简单，*DeferredCoroutine/LazyDeferredCoroutine*是*async*创建的协程，相应的*StandaloneCoroutine/LazyStandaloneCoroutine*就是*launch*对应创建的协程。

- UndispatchedCoroutine/DispatchedCoroutine/ScopeCoroutine

> 这两个是*withContext*所需要用到的协程。*UndispatchedCoroutine*不会改变*Dispatcher*，另一个就会改变*Dispatcher*。都是*ScopeCoroutine*的子类，而*ScopeCoroutine*又是*AbstractCoroutine*的实现类，所以也就相当于是一种偶作用域的协程实例而已。源码给的注释是这样的
>
> ***This is a coroutine instance that is created by [coroutineScope] builder.***

- ContextScope/MainScope

> *ContextScope*就很通俗易懂，包含*CoroutineContext*的*Scope*，不像*AbstractCoroutine*还实现了*Job*以及*CoroutineContext*。简单来说就算是删减版的*AbstractCoroutine*。

- GlobalScope

> 这是一个顶层的协程，*CoroutineScope*的实现类

- SupervisorCoroutine

> 我们知道*coroutineScope()*这个函数是可以创建协程的，他又一套异常处理机制，子*Job*异常往父丢，相应的*supervisorScope()*异常处理就相反，出现异常自己处理，这里借助的就是*SupervisorCoroutine*。

- TimeoutCoroutine

> 如果用过*withTimeout*那就更好解释了，这就是*withTimeout*内部会创建的协程。

Continuation

- CancellableContinuation/CancellableContinuationImpl

> 标准库的*Continuation*对于取消的支持不够所以就有了*CancellableContinuation*。

CoroutineContext

- CoroutineDispatcher
    - EventLoop
    - Dispatchers

> 协程调度器管理协程的调度执行，如果要说的直白一点就是一个线程池。

- CoroutineExceptionHandler

> 异常处理器

- CoroutineName

> 代表了协程的名字。

- NonCancellable

> 一种不可以取消的*Job*

- CoroutineId

> 用于调试协程的时候指定的协程的*Id*和*Name*其实有点相似。

Job

- JobNode/NodeList/InactiveNodeList/JobCancellingNode

> *Job*实现的的是任务的管理，他是怎么管理的？因为他把每个任务都当成是一个节点，然后形成了类似于树的层级结构。

- Job/CompletableJob/ChildJob/ParentJob/Deferred/CompletableDeferred

> 这基本上就是所有的*Job*类型了

Exception

- CompletionHandlerException
- CancellationException
    - JobCancellationException
- CoroutinesInternalError

> 几乎所有的异常类型。

builder

- CoroutineScope.launch
- CoroutineScope.async
- runBlocking

> 协程的构建器，在协程作用域下构建协程。

标准库

- delay
- withContext
- withTimeout
- yield

> *kotlinx-coroutines-core*中的标准库。

高级框架

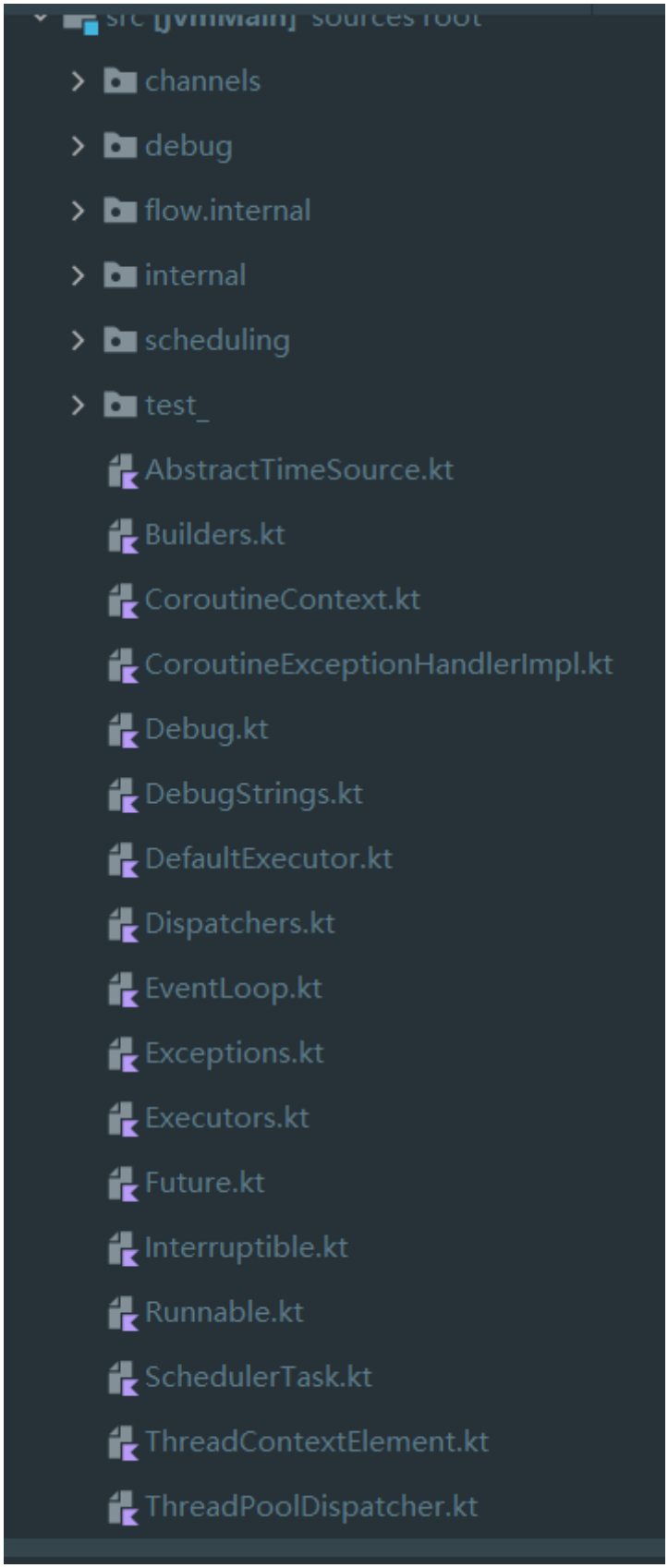
- channel
- flow
- select
- sync（Synchronization primitives (mutex)同步原语或者说叫锁）

总结

kotlinx-coroutines引入了协程的作用域的概念，然后在stdlib协程的的基础框架层的支持下写了一套方便易用的框架。总的来说这就是一个协程扩展库。

kotlinx-coroutines-core-JVM

> *Time： 2022-1-23*



JVM的具体实现其实没有我们想得那么难，似乎并没有那么多。

Content

框架

- flow.internal
- channels

internal

内部主要是一些关于并发的工具类，于并发安全相关，除此之外就是一些common的实现。

稍微能看懂的也就是MainDispatcherLoader（加载不同平台的Main线程）。

scheduling

- CoroutineScheduler.kt
- Deprecated.kt
- Dispatcher.kt
- Tasks.kt
- WorkQueue.kt

线程池以及Task，以及调度器，以及任务队列数据结构。

other

- runBlocking/BlockingCoroutine

  > *runBlocking*的具体实现

- CoroutineContext.kt

  > 一些*Continuation*以及*CoroutineContext*的扩展和*common*中*AbstractCoroutine*的实现类。

- CoroutineExceptionHandlerImpl.kt

  > 异常处理处理的实现类

- DefaultExecutor.kt

  > *EventLoop*的一个实现类*DefaultExecutor*，比如*Delay*默认使用的就是*DefaultExecutor*。

- Dispatchers.kt

  > *JVM*平台的*Dispachers*的实现类。如：
  >
  > *Default=DefaultScheduler*
  >
  > *Main=MainDispatcherLoader.dispatcher*
  >
  > *Unconfined= kotlinx.coroutines.Unconfined*
  >
  > *IO = DefaultIoScheduler*

- EventLoop.kt

  > *EventLoop*简单来说就是线程池的一个封装，这个文件里面也就是实现了一个*BlockingEventLoop*

- Executors.kt

  > 提供了线程池到*CoroutineDispacher*的转化。

- ThreadPoolDispatcher.kt

  > 前面只是提供了线程池到*Dispatcher*的转化方法，线程池还是得自己*new*的，而这个提供了更近一步进行了封装，*new*线程池到转化*ExecutorCoroutineDispatcher(CoroutineDispatcher*的子类*)*。一条龙服务。