副本 Kotlin协程使用和与原理分析

分享：朱锦彪　　指导：黄细闽

协程概述

协程(Coroutines)是**轻量级的线程**，它不需要从**用户态**切换到内核态，Coroutine 是**编译器**级的，Process 和 Thread 是操作系统级的，协程没有直接和操作系统关联，但它也是跑在线程中的，可以是单线程，也可以是多线程。协程设计的初衷是为了解决并发问题，让协作式多任务实现起来更加方便，它可以有效地**消除回调地狱**。

实际上在 Kotlin 中，协程是一套线程 API, 就像 Java 的 `Executors` 和 Android 的 `Handler` 等，是一套比较方便的线程框架，它能够在同一个代码块里进行多次的线程切换，可以用看起来同步的方式写出异步的代码，即**非阻塞式挂起**。

异步编程

异步编程中最为常见的场景是：在后台线程执行一个复杂任务，下一个任务依赖于上一个任务的执行结果，所以必须等待上一个任务执行完成后才能开始执行。看下面代码中的三个函数，后两个函数都依赖于前一个函数的执行结果：

```
fun requestToken(): Token {...}
fun sendPost(token: Token, item: Item) {...}
fun refreshUI(result: Boolean) {...}
```

回调嵌套

常见的做法是使用回调，把之后需要执行的任务封装为回调：

```
fun requestTokenAsync(block: (Token) -> Unit) {...}
fun sendPostAsync(token: Token, item: Item,
  block: (Boolean) -> Unit) {...}
fun refreshUI(result: Boolean) {...}
fun postItemUsingCallBack(item: Item) {
    requestTokenAsync { token ->
        sendPostAsync(token, item) { result ->
            refreshUI(result)
        }
    }
```

```
}
```

层次太多的回调嵌套会大幅提高代码的维护难度，令人难以接受。

CompletableFuture

Java 8 引入的 `CompletableFuture` 可以将多个任务串联起来，可以避免多层嵌套的问题：

```
@RequiresApi(BUild.VERSION_CODES.N)
fun requestTokenAsync(): CompletableFuture<Token> { ... }
@RequiresApi(BUild.VERSION_CODES.N)
fun sendPostAsync(token: Token,
    item: Item): CompletableFuture<Booleam> { ... }
fun refreshUI(post: Post) { ... }
@RequiresApi(BUild.VERSION_CODES.N)
fun postItem(item: Item) {
    requestTokenAsync()
            .thenCompose { token -> createPostAsync(token, item) }
            .thenAccept { result -> processPost(result) }
            .exceptionally { e ->
                e.printStackTrace()
                null
            }
}
```

不过 `CompletableFuture` 只能在 API24 以上使用。

RxJava

RxJava 的方式类似 `CompletableFuture` 系列的链式调用，这也是目前大多数推荐的做法：

```
fun requestToken(): Token { ... }
fun sendPost(token: Token, item: Item) { ... }
fun refreshUI(result: Boolean) { ... }
fun postItem(item: Item) {
    Single.fromCallable { requestToken() }
        .map { token -> sendPost(token, item) }
        .subscribe(
            { result -> refreshUI(result) }
            { e -> e.printStackTrace() }
        )
}
```

使用协程

使用 Kotlin 协程的代码形式上显得非常简洁，以同步的方式书写异步代码，但是不会阻塞当前线程：

```
suspend fun requestToken(): Token { ... }
suspend fun sendPost(token: Token, item: Item) { ... }
fun refreshUI(result: Boolean) { ... }
fun postItem(item: Item) {
    MainScope().launch {
        val token = requestToken()
        val result = sendPost(token, item)
        refreshUI(result)
    }
}
```

概念解析

挂起函数

- `requestToken` 和 `createPost` 函数前面有 suspend 修饰符标记，这表示两个函数都是挂起函数。挂起函数能够以与普通函数相同的方式获取参数和返回。
- 挂起函数挂起协程时，不会阻塞协程所在的线程。
- 挂起函数执行完成后会恢复协程，后面的代码才会继续执行。
- 但是挂起函数只能在协程中或其他挂起函数中调用。

```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job
```

要启动协程，至少要有一个挂起函数，它通常是一个挂起 lambda 表达式。所以 suspend 修饰符可以标记普通函数、扩展函数和 lambda 表达式。

协程调度器

协程上下文包含一个协程调度器 `CoroutineDispatcher` 。协程调度器可以将协程限制在特定的线程环境下执行，或将它分派到一个线程池，亦或是让它不受限地运行：

```
object Dispatchers {
    val Default
    val Main
    val Unconfined
    val IO
}
```

Kotlin 库自带四个预设的协程调度器： `Default` 、 `IO` 、 `Main` 、 `UnConfined`
同时，调用 launch 时也可以不带上下文参数使用默认值，或者使用线程池构造协程调度器。

```
launch(Executors.newSingleThreadExecutor()
```

```
    .asCoroutineDispatcher()) {...}
```

- 缺省上下文参数：协程将运行在父协程的上下文中，即使用父协程的协程调度器。
- `Dispatchers.Default` ：获取默认调度器，根据一个线程数为 2 到 cpu 核心数的线程池构造而来。适合 CPU 密集型任务如排序，压缩，编解码
- `SingleThreadContext` ：新开一个专用的线程来运行这个协程，需要释放或者复用。
- `Dispatchers.IO` ：将协程运行在 IO 专用线程池，这个线程池和 `Default` 线程池共享线程。适合运行网络请求，文件读写，数据库访问等。
- `Dispatchers.Main` ：将协程运行在安卓的 UI 线程。
- `Dispatchers.UnConfined` ：不受限制地调度。

`Dispatchers.Unconfined` 协程调度器在调用它的线程启动了一个协程，但该线程仅仅只是运行到第一个挂起点。挂起后，协程代码继续执行，但是执行环境仍然在刚刚挂起过程的线程中。非受限的调度器非常适用于执行不消耗 CPU 时间的任务，以及不更新局限于特定线程的任何共享数据（如 UI）的协程。

协程启动模式

```
enum class CoroutineStart {
    DEFAULT,
    LAZY,
    ATOMIC,
    UNDISPATCHED
}
```

- `DEFAULT` ：饿汉式启动， `luanch` 调用后协程马上进入待调度状态，等待调度器就绪后就开始执行。
- `LAZY` ：懒汉式启动， `launch` 后并不会有任何调度行为，协程体也不会进入执行状态，直到需要它执行(调用 `job.start` / `join` 等)的时候才会执行。
- `ATOMIC` ：不可在协程 `start` 之前调用 `cancel` 函数，其他与 `DEFAULT` 一致。
- `UNDISPATCHED` ：执行时不进行线程调度，在原有线程上运行，其他与 `ATOMIC` 一致。

协程的启动方法

launch 函数

- `CoroutineScope.launch` 是最常用的 Coroutine builders，不阻塞当前线程，在后台创建一个新协程，也可以指定协程调度器，例如在 Android 中常用的 `MainScope().launch()` ：

```
fun postItem(item: Item) {
    MainScope().launch {
        val token = requestToken()
```

```
        val result = sendPost(token, item)
        refreshUI(result)
    }
}
```

async 函数

首先，我们看看在需要的情况下不用 `async` 函数的情景

```
suspend fun doSomethingOne(): Int {
    delay(1000L)
    return 63
}
suspend fun doSomethingTwo(): Int {
    delay(1500L)
    return 24
}
fun click() {
    MainScope().launch {
        val time = measureTimeMillis {
            val one = doSomethingOne()
            val two = doSomethingTwo()
            println("The result is ${one + two}")
        }
        println("Completed in $time ms")
    }
}
```

运行结果如下：

```
The answer is 87
Completed in 2553 ms
```

两个没有依赖关系的耗时函数如果串行调用会浪费时间。

```
fun click() {
    MainScope().launch {
        val time = measureTimeMillis {
            val one = async { doSomethingOne() }
            val two = async { doSomethingTwo() }
            println("The result is ${one.await() + two.await()}")
        }
        println("Completed in $time ms")
    }
}
```

运行结果如下

```
The answer is 87
Completed in 1613 ms
```

- `async` 函数与 `launch` 类似，启动了一个新的协程。不同的地方在于 `async` 方法返回的是一个 `Deferred` 实例，它扩展了 `Job` 类，调用 `Deferred.await()` 函数可以非阻塞地获取异步结果。

runBlocking 函数

- runBlocking 方法创建一个运行时阻塞当前线程的协程，一般不会在实际环境中使用，主要是为 main 函数和测试函数所设计

```
@JvmStatic
fun main(args: Array<String>) {
    runBlocking {
        launch {
            delay(1000L)
            print("World!")
        }
        print("Hello ")
        delay(1500L)
    }
}
```

运行结果如下

```
Hello World!
```

协程的使用

GlobalScope

GlobalScope表示一个全局的协程作用域，他是一个静态单例对象。不过在GlobalScope中启动的协程并不会保活进程，它运行所在的线程就像守护线程一样：

```
@JvmStatic
fun main(args: Array<String>) {
    GlobalScope.launch {
        delay(500L)
        println("after delay")
    }
}
```

运行结果：

```
Process finished with exit code 0
```

在上面很多例子中，我们都是用 `GlobalScope` 演示协程的启动的。但是实际开发中并不建议使用 `GlobalScope` 启动协程，因为 `GlobalScope` 通常用于启动顶级协程，这些协程在整个应用程序生命周期内运行：

```
fun launchWithGlobalScope() {
    GlobalScope.launch(Dispatchers.Main) {
        val deferred = async {
            delay(3000L)
            "Done"
        }
        Toast.makeText(this@CoroutineActivity, deferred.await(),
Toast.LENGTH_SHORT)
                .show()
    }
}
```

提前退出 Activity 后，Toast 仍然会弹出，造成了内存泄漏。

解决办法：

```
var mGlobalJob: Job? = null

fun launchWithGlobalScope() {
    mGlobalJob = GlobalScope.launch(Dispatchers.Main) { ... }
}
override fun onDestroy() {
    super.onDestroy()
    mGlobalJob?.cancel()
}
```

多个地方都需要启动协程该怎么办?

MainScope

Android 中一般不建议使用 `GlobalScope`，因为它会创建一个顶层协程，需要保持所有对 `GlobalScope` 启动的协程的引用，然后在 Activity destory 等场景的时候 cancel 这些的协程，否则就会造成内存泄露等问题。可以使用 `MainScope`：

```
public fun MainScope(): CoroutineScope = ContextScope(SupervisorJob() +
Dispatchers.Main)
```

MainScope 启动的协程默认在主线程运行

在 Activity 的 onDestroy 回调中取消所有协程：

```
class CoroutineActivity : AppCompatActivity() {
    private val mainScope = MainScope()
    fun request1() {
        mainScope.launch {
            // ...
        }
    }
    // request2, 3, ...
    override fun onDestroy() {
        super.onDestroy()
        mainScope.cancel()
    }
}
```

改进版处理：

```
open class BaseActivity : AppCompatActivity(), CoroutineScope by
MainScope() {
    override fun onDestroy() {
        super.onDestroy()
        cancel()
    }
}
```

子类 Activity 开启协程

```
fun launchWithMainScope() {
    launch {
        val deferred = async(Dispatchers.IO) {
            delay(3000L)
            "Done"
        }
        Toast.makeText(this@CoroutineActivity, deferred.await(),
Toast.LENGTH_SHORT)
                .show()
    }
}
```

ViewModel 协程

在 ViewModel 调用 `clear` 函数后会被取消的协程

```
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.0-beta01"
```

示例代码：

```
class SampleModel : ViewModel() {
    val mMessage: MutableLiveData<String> = MutableLiveData()
    fun getMessage(message: String) {
        viewModelScope.launch() {
            delay(2000)
            mMessage.value = "post $message"
        }
    }
}
```

实现原理：

ViewModel.kt

```
val ViewModel.viewModelScope: CoroutineScope
        get() {
            val scope: CoroutineScope? = this.getTag(JOB_KEY)
            if (scope != null) {
                return scope
            }
            return setTagIfAbsent(JOB_KEY,
                CloseableCoroutineScope(SupervisorJob() +
                Dispatchers.Main.immediate))
        }

internal class CloseableCoroutineScope(context: CoroutineContext) :
    Closeable, CoroutineScope {
    override val coroutineContext: CoroutineContext = context
    override fun close() {
        coroutineContext.cancel()
    }
}
```

ViewModel.java

```
@MainThread
```

```
final void clear() {
    mCleared = true;
    // Since clear() is final, this method is still called on mock objects
    // and in those cases, mBagOfTags is null. It'll always be empty
though
    // because setTagIfAbsent and getTag are not final so we can skip
    // clearing it
    if (mBagOfTags != null) {
        synchronized (mBagOfTags) {
            for (Object value : mBagOfTags.values()) {
                // see comment for the similar call in setTagIfAbsent
                closeWithRuntimeException(value);
            }
        }
    }
    onCleared();
}
```

Lifecycle 协程

- 给每个 `LifeCycle` 对象通过扩展属性定义了协程作用域 `lifecycleScope`。你可以通过 `lfecycle.coroutineScope` 或者 `LifeCycleOwner.lifecycleScope` 进行访问。示例代码如下：

```
implementation "androidx.lifecycle:lifecycle-runtime-ktx:2.3.0-beta01"
```

源码如下：

```
val LifecycleOwner.lifecycleScope: LifecycleCoroutineScope
    get() = lifecycle.coroutineScope
```

示例代码：

```
fun launchWithLifeclcleScope() {
    lifecycleScope.launch {
        val deferred = async(Dispatchers.IO) {
            delay(3000L)
            "Done"
        }
        Toast.makeText(this@CoroutineActivity, deferred.await(),
Toast.LENGTH_SHORT)
                .show()
    }
```

```
}
```

当 `LifeCycle` 回调 `onDestroy()` 时，协程作用域 `lifecycleScope` 会自动取消。在 Activity/Fragment 等生命周期组件中我们可以很方便的使用，但是在 MVVM 中又不会过多的在 View 层进行逻辑处理，`viewModelScope` 基本就可以满足 `ViewModel` 中的需求了，`lifecycleScope` 也显得有点那么食之无味。但是他有三个特殊的用法，**可以指定至少在特定的生命周期之后再执行挂起函数，可以进一步减轻 View 层的负担**：

```
abstract class LifecycleCoroutineScope internal constructor() :
CoroutineScope {
    internal abstract val lifecycle: Lifecycle

    fun launchWhenCreated(block: suspend CoroutineScope.() -> Unit):
    Job = launch {
        lifecycle.whenCreated(block)
    }
    fun launchWhenStarted(block: suspend CoroutineScope.() -> Unit):
    Job = launch {
        lifecycle.whenStarted(block)
    }

    fun launchWhenResumed(block: suspend CoroutineScope.() -> Unit):
    Job = launch {
        lifecycle.whenResumed(block)
    }
}
```

自定义 CoroutineScope 对象

用 CoroutineScope()函数可以创建一个 CoroutineScope 对象，需要传入一个 CoroutineContext 参数：

```
public fun CoroutineScope(context: CoroutineContext): CoroutineScope =
    ContextScope(if (context[Job] != null) context else context + Job())
```

通过自定义 CoroutineScope 对象可以为协程的运行添加各种配置

```
val handler = CoroutineExceptionHandler { _, throwable ->
throwable.printStackTrace() }
val name = CoroutineName("SampleName")
val combinedContext = Dispatchers.IO + handler + name
val scope = CoroutineScope(combinedContext)
scope.launch {
    ...
```

```
}
```

协程控制——Job

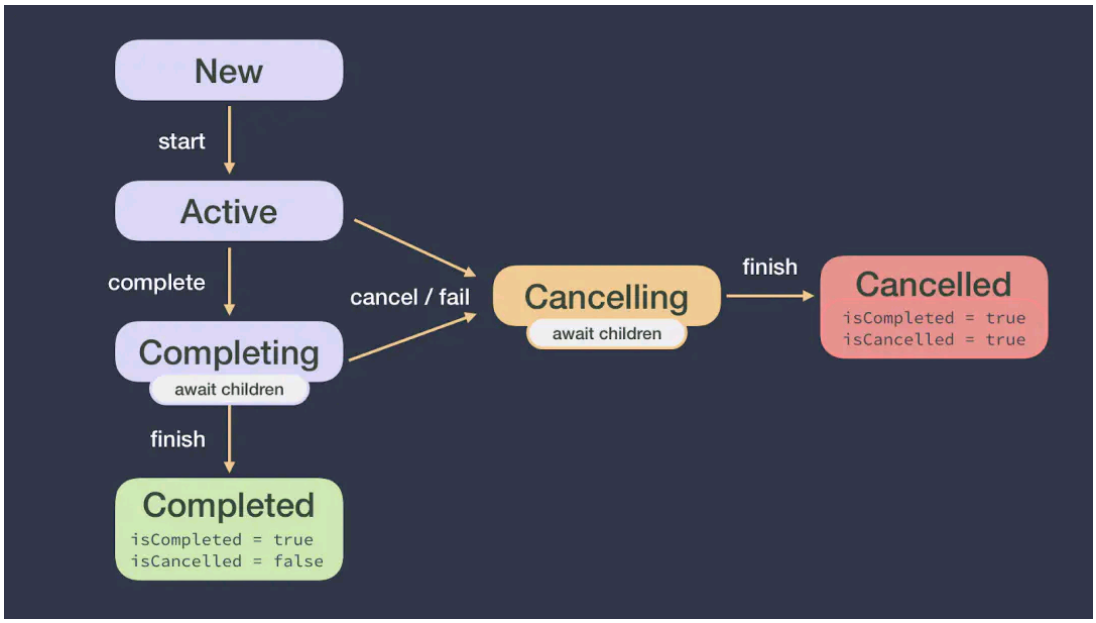
Job 通常由协程创建后返回而来，是协程的唯一标识，负责管理协程的生命周期。同时它还可以存在层级关系，即一个 Job 对象中包含多个子 Job：

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

Job 可以控制协程的执行以及存储了协程的执行状态

```
public val isActive: Boolean
public val isCompleted: Boolean
public val isCancelled: Boolean
public fun start(): Boolean
public fun cancel(cause: CancellationException? = null)
public suspend fun join()
```

协程的生命周期：



协程状态和对应标志位一览：

State	isActive	isCompleted	isCancelled
New	false	false	false
Active	true	false	false
Completing	true	false	false
Cancelling	false	false	true
Cancelled	false	true	true
Completed	false	true	false

调度切换——withContext函数

withContext函数可以在不新开协程的情况下指定上下文来运行一段代码块：

```
@JvmStatic
fun main(args: Array<String>) {
    val singleThreadExecutorContext = Executors.newSingleThreadExecutor()
            .asCoroutineDispatcher()
    runBlocking {
        println("1: ${Thread.currentThread()}")
        withContext(Dispatchers.IO) {
            println("2: ${Thread.currentThread()}")
            withContext(singleThreadExecutorContext) {
                println("3: ${Thread.currentThread()}")
            }
            println("4: ${Thread.currentThread()}")
        }
        println("5: ${Thread.currentThread()}")
    }
    singleThreadExecutorContext.close()
}
```

运行结果：

```
1: Thread[main,5,main]
2: Thread[DefaultDispatcher-worker-1,5,main]
3: Thread[pool-1-thread-1,5,main]
4: Thread[DefaultDispatcher-worker-1,5,main]
5: Thread[main,5,main]
```

协程的父子关系

- 父协程手动调用 `cancel()` 或者异常结束，会立即取消它的所有子协程。
- 父协程必须等待所有子协程完成（处于完成或者取消状态）才能完成。
- 子协程抛出未捕获的异常时，默认情况下会取消其父协程。

```
MainScope().launch {
    launch {
        val avatarUrl = requestAvatarUrl(userId)
        loadImage(mIvAvatar, avatarUrl)
    }
    launch {
        val userName = requestUserName(userId)
        mTvUserName.text = userName
```

```
    }
}
```

协程父子关系的建立

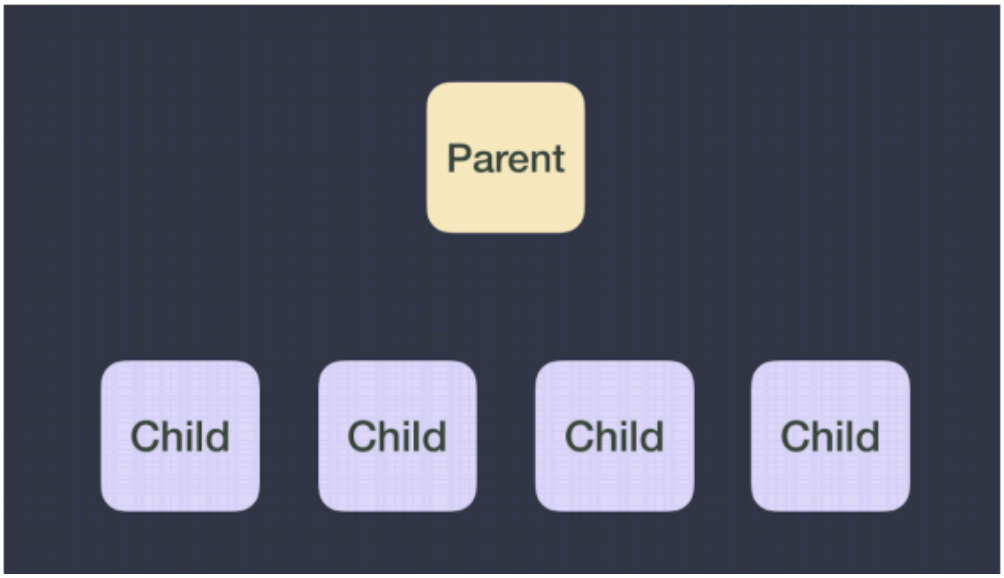
`launch` 和 `async` 新建协程时，首先都是 `newCoroutineContext(context)` 新建协程的 `CoroutineContext` 上下文：

```
public actual fun CoroutineScope.newCoroutineContext(context:
    CoroutineContext): CoroutineContext {
    // 新协程继承了原来 CoroutineScope 的 coroutineContext 即当前父协程环境下的上下文
    val combined = coroutineContext + context
    val debug = if (DEBUG) combined +
        CoroutineId(COROUTINE_ID.incrementAndGet()) else combined
    // 当新协程没有指定线程调度器时，会默认使用 Dispatchers.Default
    return if (combined !== Dispatchers.Default &&
        combined[ContinuationInterceptor] == null)
        debug + Dispatchers.Default else debug
}
```

新协程的 CoroutineContext 都继承了原来 CoroutineScope 的 coroutineContext，然后 `launch` 和 `async` 新建协程最后都会调用 `start(start: CoroutineStart, receiver: R, block: suspend R.() -> T)`，里面第一行是 `initParentJob()`，通过注释可以知道就是这个函数建立父子关系的，下面看其实现：

```
// AbstractCoroutine.kt
internal fun initParentJob() {
    initParentJobInternal(parentContext[Job])
}
// JobSupport.kt
internal fun initParentJobInternal(parent: Job?) {
    ...
    parent.attachChild(this)
}
public final override fun attachChild(child: ChildJob):
    ChildHandle {
    return invokeOnCompletion(onCancelling = true, handler =
        ChildHandleNode(this, child).asHandler) as ChildHandle
        // 将 handler 结点添加到父协程的 state.list 中
}
```

在父协程中设有一个队列专门保存子协程的 Job 方便控制子协程，子协程新建时在 `invokeOnCompletion()` 函数中将新建的子协程的 Job 插入到这个队列中。



协程取消

- 父协程手动调用 `cancel()` ，会立即取消它的所有子协程。
- 子协程调用 cancel 取消时默认不取消父协程。

跟踪父协程的 `cancel()` 调用过程，其中关键过程为 `cancel()` -> `cancel(null)` -> `cancelImpl(null)` -> `makeCancelling(null)` -> `tryMakeCancelling(state, causeException)` -> `notifyCancelling(list, rootCause)` ，下面继续分析 `notifyCancelling(list, rootCause)` 的实现。

```
// JobSupport.kt
private fun notifyCancelling(list: NodeList, cause: Throwable) {
    // first cancel our own children
    onCancellation(cause)
    // 这里会调用所有子协程绑定的 ChildHandleNode.invoke(cause) ->
    //childJob.parentCancelled(parentJob) 来取消所有子协程
    notifyHandlers<JobCancellingNode<*>>(list, cause)
    // then cancel parent
    // cancelParent(cause) 不一定会取消父协程，cancel() 时不会取消父协程，因
    // 为此时产生 cause 的是 JobCancellationException，属于
CancellationException
    cancelParent(cause) // tentative cancellation -- does not matter if
there is no parent
}
public final override fun parentCancelled(parentJob: ParentJob) {
    // 父协程取消时，子协程会通过 parentCancelled 来取消自己
    cancelImpl(parentJob)
}
// returns true if the parent is responsible for handling the exception,
false otherwise
```

```
private fun cancelParent(cause: Throwable): Boolean
    if (cause is CancellationException) return true
    if (!cancelsParent) return false
    // 当 cancelsParent 为 true，且子线程抛出未捕获的异常时，默认情况下
childCancelled() 会取消其父协程。
    return parentHandle?.childCancelled(cause) == true
}
//job.kt
public interface ChildHandle : DisposableHandle {
    public fun childCancelled(cause: Throwable): Boolean
}
// parentHandle 是协程体的一个成员变量，是一个接口，由父协程实现并赋给子协程，供需
要取消父协程时使用
```

对于协程的取消， `cancel()` 只是将协程的状态修改为已取消状态，并不能取消协程的运算逻辑，但是协程库中很多挂起函数都会检测协程状态并响应协程的取消，如delay()函数：

```
@JvmStatic
fun main(args: Array<String>) {
    runBlocking {
        val job = launch {
            var i = 0
            while (i < 5) {
                delay(500L)
                println("job: i am sleeping ${i++}")
            }
        }
        delay(1300L)
        println("main: I am about to cancel")
        job.cancelAndJoin()
        println("main: after cancelled")
    }
}
```

运行结果：

```
job: i am sleeping 0
job: i am sleeping 1
main: I am about to cancel
main: after cancelled
```

协程运行纯计算逻辑时，无法响应协程的取消：

```
@JvmStatic
fun main(args: Array<String>) {
```

```
    runBlocking {
        var startTime = System.currentTimeMillis()
        val job = launch(Dispatchers.Default) {
            var i = 0
            while (i < 5 /*&& isActive*/) {
                if (System.currentTimeMillis() >= startTime){
                    println("job: i am sleeping ${i++}")
                    startTime += 500L
                }
            }
        }
        delay(1300L)
        println("main: I am about to cancel")
        job.cancelAndJoin()
        println("main: after cancelled")
    }
}
```

运行结果：

```
job: i am sleeping 0
job: i am sleeping 1
job: i am sleeping 2
main: I am about to cancel
job: i am sleeping 3
job: i am sleeping 4
main: after cancelled
```

用WithContext函数运行不可取消的代码块：

```
@JvmStatic
fun main(args: Array<String>) {
    runBlocking {
        val job = launch(Dispatchers.Default) {
            withContext(NonCancellable) {
                var i = 0
                while (i < 5) {
                    println("job: i am sleeping ${i++}")
                    delay(500L)
                }
            }
        }
        delay(1300L)
        println("main: I am about to cancel")
```

```
        job.cancelAndJoin()
        println("main: after cancelled")
        println("job.isActive = ${job.isActive}")
        println("job.isCancelled = ${job.isCancelled}")
        println("job.isCompleted = ${job.isCompleted}")
    }
}
```

结果如下：

```
job: i am sleeping 0
job: i am sleeping 1
job: i am sleeping 2
main: I am about to cancel
job: i am sleeping 3
job: i am sleeping 4
main: after cancelled
job.isActive = false
job.isCancelled = true
job.isCompleted = true
```

协程完成

- 父协程必须等待所有子协程的完成。

协程的完成通过 `AbstractCoroutine.resumeWith()` 实现，它会这样调用: `makeCompletingOnce()` -> `tryMakeCompleting()` -> `tryMakeCompletingSlowPath()` -> `tryWaitForChild()` :协程完成（处于完成或者取消状态）才能完成

```
// JobSupport.kt
private tailrec fun tryWaitForChild(state: Finishing, child:
ChildHandleNode, proposedUpdate: Any?): Boolean {
    // 添加 ChildCompletion 节点到子协程的 state.list 末尾，当子协程完成时会
    // 调用 ChildCompletion.invoke()
    val handle = child.childJob.invokeOnCompletion(
        invokeImmediately = false,
        handler = ChildCompletion(this, state, child,
            proposedUpdate).asHandler)
    if (handle !== NonDisposableHandle) return true // child is not
complete and we've started waiting for it
    val nextChild = child.nextChild() ?: return false
    // 继续设置下一个子协程
    return tryWaitForChild(state, nextChild, proposedUpdate)
}
```

```
// ChildCompletion class
override fun invoke(cause: Throwable?) {
    parent.continueCompleting(state, child, proposedUpdate)
}
private fun continueCompleting(state: Finishing, lastChild:
    ChildHandleNode, proposedUpdate: Any?) {
    require(this.state === state) // consistency check -- it cannot
    // change while we are waiting for children
    // figure out if we need to wait for next child
    val waitChild = lastChild.nextChild()
    // try wait for next child
    if (waitChild != null && tryWaitForChild(state, waitChild,
        proposedUpdate)) return // waiting for next child
    // no more children to wait -- try update state
    // 当所有子协程都完成时，才会 tryFinalizeFinishingState() 完成自己
    if (tryFinalizeFinishingState(state, proposedUpdate,
        MODE_ATOMIC_DEFAULT)) return
}
```

异常处理

launch 和 async 的区别

- `launch` 和 `async` 很大的一个区别是异常处理。`launch` 默认处理异常的方式只是打印对战信息，`async` 期望你通过调用 `await` 来获取结果（或异常），所以它默认不会抛出异常。`async` 启动的协程将异常捕获但不抛出，保存在 `Deferred` 对象中，待到调用 `await()` 时再抛出。

```
@JvmStatic
fun main(args: Array<String>) = runBlocking{
    val job = GlobalScope.launch { // launch 根协程
        println("Throwing exception from launch")
        throw IndexOutOfBoundsException()
    }
    job.join()
    println("Joined failed job")
    val deferred = GlobalScope.async { // async 根协程
        println("Throwing exception from async")
        throw ArithmeticException() // 没有打印任何东西，依赖用户去调用等待
    }
//        try {
```

```
//            deferred.await()
//        } catch (e: ArithmeticException) {
//            e.printStackTrace()
//        }
    deferred.await()
    println("end")
}
```

运行结果：

```
Throwing exception from launch
Exception in thread "DefaultDispatcher-worker-2"
java.lang.IndexOutOfBoundsException
at...
Exception in thread "main" java.lang.ArithmeticException
at...
Joined failed job
Throwing exception from async
Process finished with exit code 1
```

恢复代码中被注释掉的 try-catch 块并删除 `deferred.await()` 调用后的结果：

```
Throwing exception from launch
Exception in thread "DefaultDispatcher-worker-2"
java.lang.IndexOutOfBoundsException
at...
java.lang.ArithmeticException
at...
Joined failed job
Throwing exception from async
end
Process finished with exit code 0
```

CoroutineExceptionHandler

- 可以使用 `CoroutineExceptionHandler` 来捕获和处理 `launch` 的异常：

```
@JvmStatic
fun main(args: Array<String>) = runBlocking{
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) {
```

```
        throw AssertionError()
    }
    val deferred = GlobalScope.async(handler) {
        throw ArithmeticException()
    }
    job.join()
    println("after join")
    val await = deferred.await()
    println("after await") // unreachable code
}
```

运行结果：

```
CoroutineExceptionHandler got java.lang.AssertionError
after join
Exception in thread "main" java.lang.ArithmeticException
    at ...
Process finished with exit code 1
```

`CoroutineExceptionHandler` 对 `async` 启动的协程无效。

- `CoroutineExceptionHandler` 仅在未捕获的异常上调用。

```
@JvmStatic
fun main(args: Array<String>) = runBlocking{
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler1 got $exception")
    }
    val job = GlobalScope.launch(handler) {
        launch(CoroutineExceptionHandler { _, throwable ->
            println("CoroutineExceptionHandler2 got $throwable")
        }){
            try {
                throw IllegalStateException()
            } catch (e : Exception) {
                println("Exception catched in try-catch block")
            }
        }
    }
    job.join()
    println("after join")
}
```

运行结果：

```
Exception catched in try-catch block
after join
Process finished with exit code 0
```

异常向上传递

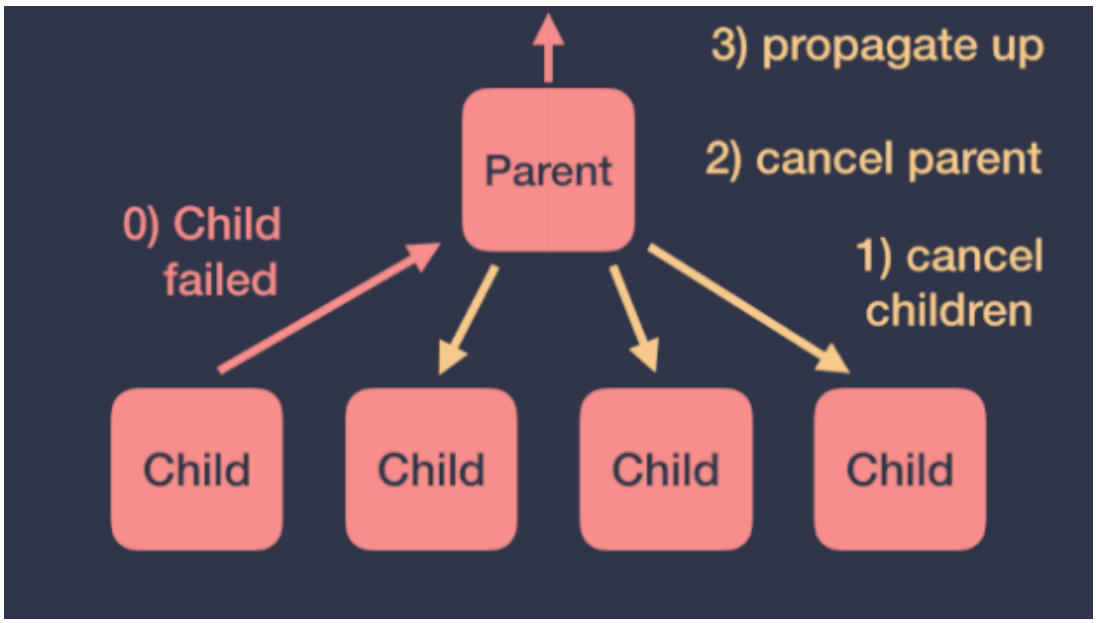
- 所有**子协程**（在另一个 `Job` 上下文中创建的协程）委托它们的父协程处理它们的异常，然后它们也委托给其父协程，以此类推直到根协程， 因此永远不会使用在其上下文中设置的 `CoroutineExceptionHandler` 。

```
@JvmStatic
fun main(args: Array<String>) = runBlocking{
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandlerOne got $exception")
    }
    val job = GlobalScope.launch(handler) {
        launch(CoroutineExceptionHandler { _, throwable ->
            println("CoroutineExceptionHandlerTwo got $throwable")
        }){
            throw IllegalStateException()
        }
    }
    val job2 = GlobalScope.launch() {
        launch(CoroutineExceptionHandler { _, throwable ->
            println("CoroutineExceptionHandlerThree got $throwable")
        }){
            throw ArithmeticException()
        }
    }
    job.join()
    println("after join")
}
```

运行结果如下：

```
CoroutineExceptionHandlerOne got java.lang.IllegalStateException
Exception in thread "DefaultDispatcher-worker-2"
java.lang.ArithmeticException
    at ...
after join
Process finished with
```

- 协程内部使用 `CancellationException` 来进行取消，这个异常会被所有的 `CoroutineExceptionHandler` 忽略。如果一个协程遇到了 `CancellationException` 以外的异常，默认情况下它将使用该异常取消 它的父协程，同时，因为异常而被结束的父协程又会取消它的所有子协程：




```
@JvmStatic
fun main(args: Array<String>) = runBlocking {
    coroutineScope {
        log(0)
        launch {
            delay(50L)
            log(1)
            throw Exception()
        }
        launch {
            log(2)
            throw CancellationException()
        }
        launch {
            delay(51L)
            log(3) // unreachable
        }
        log(4)
        log(isActive)
        delay(100)
        log(5) // unreachable
    }
}
fun log(obj: Any?) {
```

```
    println("${Thread.currentThread()} : $obj")
}
```

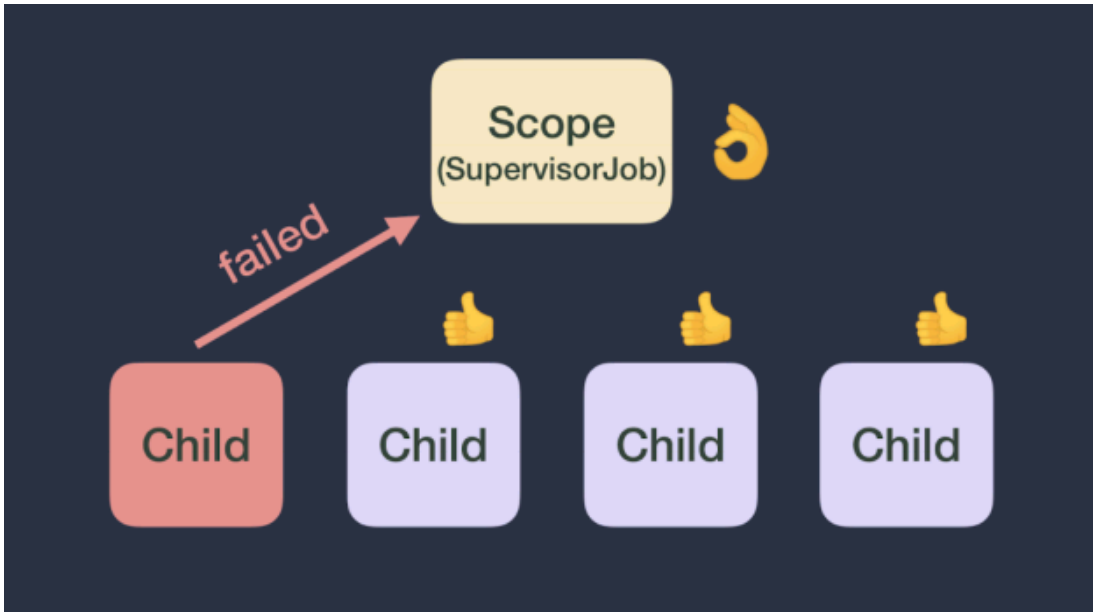
运行结果：

```
Thread[main,5,main] : 0
Thread[main,5,main] : 4
Thread[main,5,main] : true
Thread[main,5,main] : 2
Thread[main,5,main] : 1
Exception in thread "main" java.lang.Exception

Process finished with exit code 1
```

supervisorScope

- `supervisorScope` 不会被子协程的异常所取消：




```
fun main() = runBlocking {
    supervisorScope {
        log(0)
        launch {
            log(1)
            throw Exception()
        }
        log(2)
        delay(100)
        log(3)
        log(isActive)
    }
}
```

运行结果：

```
Thread[main,5,main]: 0
Thread[main,5,main]: 2
Thread[main,5,main]: 1
Exception in thread "main" java.lang.Exception
Thread[main,5,main]: 3
Thread[main,5,main]: true
```

异常处理时机

- 当父协程的所有子协程都结束后，原始的异常才会被父协程处理：

```
@JvmStatic
fun main(args: Array<String>) = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) {
        val innerJob = launch { // 该栈内的协程都将被取消
            launch {
                launch {
                    throw IOException() // 原始异常
                }
                Thread.sleep(100L)
                println("1: $isActive")
            }
            Thread.sleep(100L)
            println("2: $isActive")
        }
        println("before join")
        innerJob.join()
        println("after join") //unreachable
    }
    job.join()
}
```

运行结果：

```
before join
2: false
1: false
CoroutineExceptionHandler got java.io.IOException
Process finished with exit code 0
```

发生多个异常

- 当协程的多个子协程因异常而失败时，一般规则是“取第一个异常”，因此将处理第一个异常。在第一个异常之后发生的所有其他异常都作为被抑制的异常绑定至第一个异常。

```
@JvmStatic
fun main(args: Array<String>) = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception with
            suppressed ${exception.suppressed.contentToString()}")
    }
    val job = GlobalScope.launch(handler) {
        launch {
            try {
                delay(Long.MAX_VALUE) // 当另一个同级的协程因 IOException
                    失败时，它将被取消
            } finally {
                throw ArithmeticException() // 第二个异常
            }
        }
        launch {
            delay(100)
            throw IOException() // 首个异常
        }
        delay(Long.MAX_VALUE)
    }
    job.join()
}
```

运行结果：

```
CoroutineExceptionHandler got java.io.IOException with suppressed
[java.lang.ArithmeticException]
```

协程工作原理

状态机

下面反编译示例 Kotlin 代码生成的字节码探究协程体的工作细节。

示例代码：

```
suspend fun requestToken(): Token {
    delay(123L)
    return Token()
```

```
}
suspend fun sendPost(token: Token, item: Item): Boolean {
    delay(456L)
    return true
}
fun refreshUI(result: Boolean) { }
fun postItem(item: Item) {
    MainScope().launch {
        val token = requestToken()
        println(token)
        val result = sendPost(token, item)
        refreshUI(result)
    }
}
```

代码中，MainScope.launch 启动的协程体在执行到 `requestToken()` 时，协程体会挂起，直到 `requestToken()` 返回可用结果，才会恢复协程，执行到 `sendPost()` 也是同样的过程。协程内部实现使用状态机来处理不同的挂起点，状态机可以表示成一个Java类，每次协程的开启就会实例化这样一个类。每一次的挂起和恢复，都表现为这个状态机实例内部的状态流转，伴随着不同状态执行不同的代码片段，整个过程保证按照协程体的顺序运行。

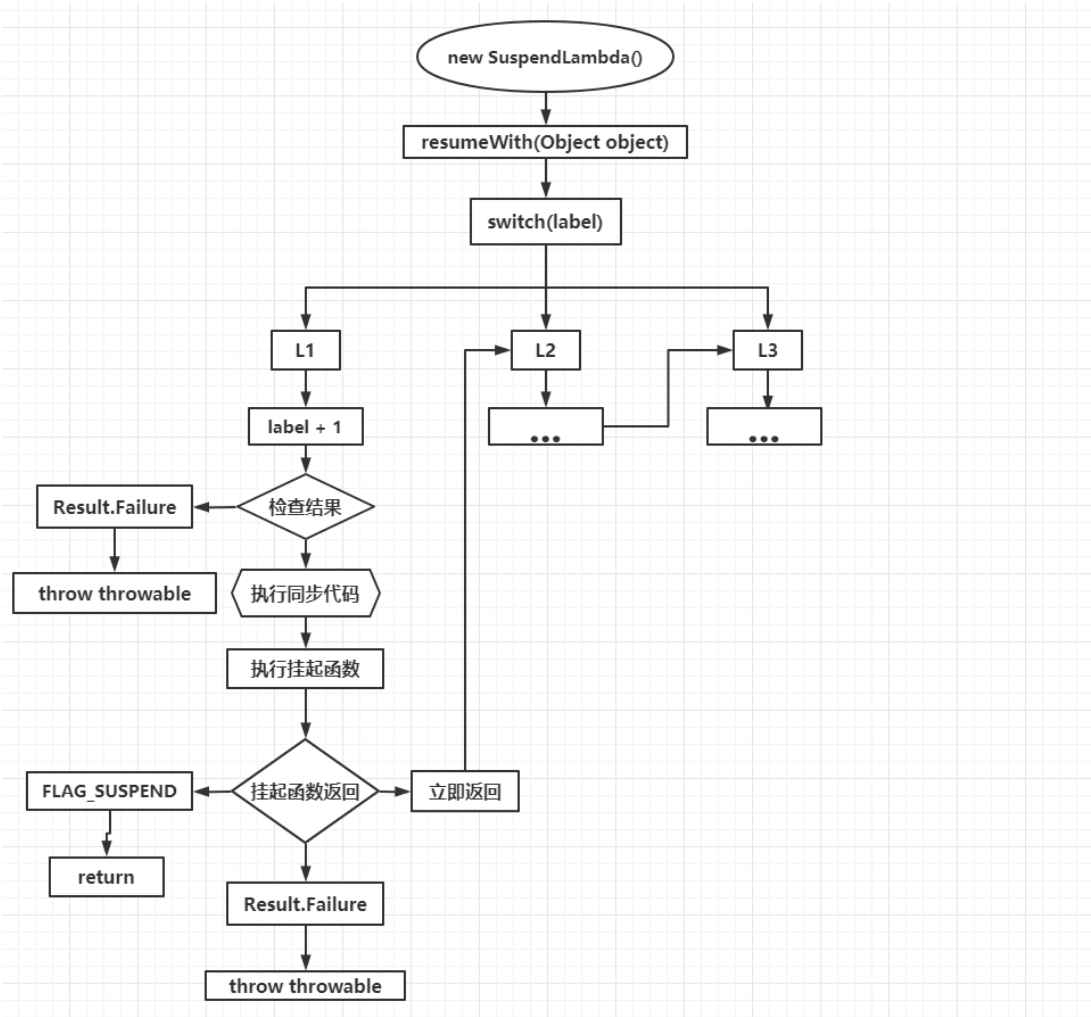
将协程体字节码反编译成 Java 代码，用伪代码方式表示大致如下：

`SuspendLambda` -> `ContinuationImpl` -> `BaseContinuationImpl` -> `Continuation`

```
class CompiledCoroutineBlock extends SuspendLambda {
    Token token = null;
    int label = 0;
    public final void resumeWith(Object result) {
        switch(label) {
            case 0: goto L0; break;
            case 1: goto L1; break;
            case 2: goto L2; break;
            default: throw new IllegalStateException();
        }
        L0: // 首次调用
            label = 1;
            result = requestToken(this); // 自己是个Continuation，传入
            if (result == FLAG_COROUTINE_SUSPEND) return;// 需要等待，结束，协程挂起，等待下一次的 resumeWith函数被调用
        L1:
            label = 2;
            checkException(result);// 检查result是不是异常，如果是则抛出
```

```
            token = (Token)result;
            System.out.println(token);
            result = sendPost(token, item, this);
            if (result == FLAG_COROUTINE_SUSPEND) return;
        L2:
            label = 3;
            checkException(result);
            refreshUI((Boolean)result)
            // 没有其他 suspend 操作了，执行完剩余协程体代码，状态机结束
    }
}
```

状态机运行流程图如下：



反编译 postItem 函数生成的字节码，得到如下 Java 代码：

```
public final void postItem(@NotNull final Item item) {
   Intrinsics.checkNotNullParameter(item, "item");
   BuildersKt.launch$default(CoroutineScopeKt.MainScope(),
(CoroutineContext)null, (CoroutineStart)null, (Function2)(new
Function2((Continuation)null) {
      private CoroutineScope p$;
```

```
      Object L$0;
      Object L$1;
      int label;
      @Nullable
      public final Object invokeSuspend(@NotNull Object $result) {
         Object var10000;
         boolean result;
         label17: {
            Object var5 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
            CoroutineScope $this$launch;
            Token token;
            Test var6;
            switch(this.label) {
            case 0:
               ResultKt.throwOnFailure($result);
               $this$launch = this.p$;
               var6 = Test.this;
               this.L$0 = $this$launch;
               this.label = 1;
               var10000 = var6.requestToken(this);
               if (var10000 == var5) {
                  return var5;
               }
               break;
            case 1:
               $this$launch = (CoroutineScope)this.L$0;
               ResultKt.throwOnFailure($result);
               var10000 = $result;
               break;
            case 2:
               token = (Token)this.L$1;
               $this$launch = (CoroutineScope)this.L$0;
               ResultKt.throwOnFailure($result);
               var10000 = $result;
               break label17;
            default:
               throw new IllegalStateException("call to 'resume' before
'invoke' with coroutine");
            }
            token = (Token)var10000;
            result = false;
            System.out.println(token);
```

```
            var6 = Test.this;
            Item var10002 = item;
            this.L$0 = $this$launch;
            this.L$1 = token;
            this.label = 2;
            var10000 = var6.sendPost(token, var10002, this);
            if (var10000 == var5) {
               return var5;
            }
         }
         result = (Boolean)var10000;
         Test.this.refreshUI(result);
         return Unit.INSTANCE;
      }
      @NotNull
      public final Continuation create(@Nullable Object value, @NotNull
Continuation completion) {
         Intrinsics.checkNotNullParameter(completion, "completion");
         Function2 var3 = new <anonymous constructor>(completion);
         var3.p$ = (CoroutineScope)value;
         return var3;
      }
      public final Object invoke(Object var1, Object var2) {
         return ((<undefinedtype>)this.create(var1,
(Continuation)var2)).invokeSuspend(Unit.INSTANCE);
      }
   }), 3, (Object)null);
}
```

- 可见，suspend 函数不会阻塞线程，且 suspend 函数不一定会挂起协程，如果相关调用的结果已经可用，则继续运行而不挂起。

协程的挂起和恢复

下面查看 suspend 函数经过编译后会得到什么：

```
suspend fun requestToken(): Token {
    delay(123L)
    return Token()
}
suspend fun sendPost(token: Token, item: Item): Boolean {
    delay(456L)
    return true
```

```
}
fun refreshUI(result: Boolean) { }
fun postItem(item: Item) {
    MainScope().launch {
        val token = requestToken()
        println(token)
        val result = sendPost(token, item)
        refreshUI(result)
    }
}
```

编译后：

```
public final Object requestToken(@NotNull Continuation $completion) { ...
}
public final Object createPost(@NotNull Token token, @NotNull Item item,
@NotNull Continuation $completion)
```

Kotlin 协程的内部实现使用了 Kotlin 编译器的一些编译技术，当 suspend 函数被调用时，都有一个隐式的参数额外传入，这个参数是 Continuation 类型，封装了协程 resume 后执行的代码逻辑。

协程的挂起

以上面 `PostItem()` 方法为例，协程体一开始以主线程环境执行，执行到 suspend 函数 `requestToken()` ，进入耗时阶段，此时的代码交付给其他线程执行，主线程不再执行协程体内代码，转去执行其他业务。

- 此过程中，主线程没有发生阻塞，体现了协程的非阻塞性
- 主线程执行代码遇到耗时操作，交付给其他线程，为协程的挂起

协程的恢复

suspend 函数编译后，调用它需要额外传入一个非空的 `Continution` 对象，这个 `Continution` 对象保证了协程的恢复。在耗时函数执行完毕后， `Continution` 的 `resumeWith()` 方法将会被调用。同时，成员变量 `CoroutineContext` 保存了协程的调度器信息，根据调度器的要求 `resumeWith()` 方法可以切换到指定的线程上运行。

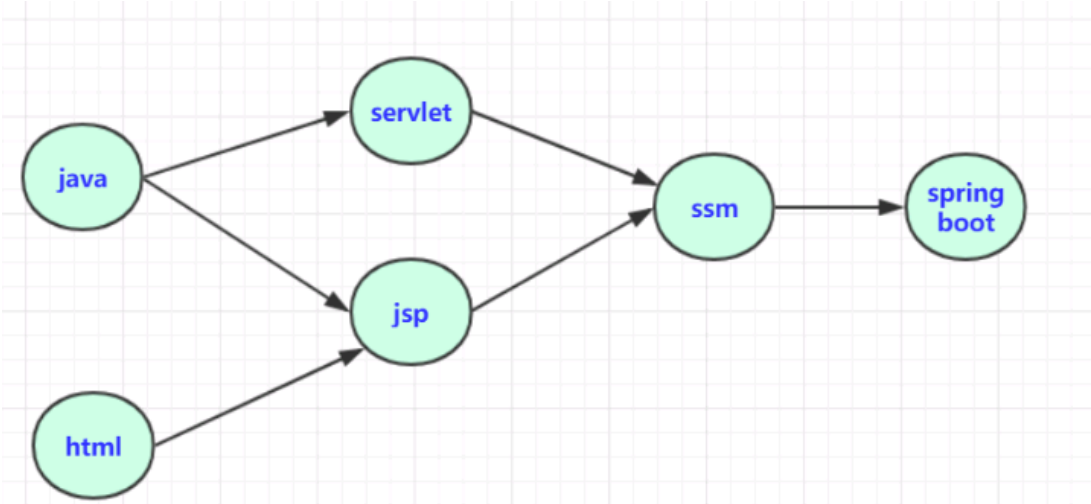
```
/**
 * Interface representing a continuation after a suspension point that
returns a value of type `T`.
 */
public interface Continuation<in T> {
    /**
     * The context of the coroutine that corresponds to this continuation.
```

```
     */
    public val context: CoroutineContext
    /**
     * Resumes the execution of the corresponding coroutine passing a
successful or failed [result] as the
     * return value of the last suspension point.
     */
    public fun resumeWith(result: Result<T>)
}
```

在实际运行中，被传进一个 suspend 函数里的 `Continuation` 实例，它的 `resumeWith()` 方法执行的就是协程体内该 suspend 函数之后的代码。所以 `Continuation` 称为续体或者剩余计算。

协程优势

下面借助拓扑排序中构建学习路线的例子分析协程的优势：




```
// 课程类
class Java
class Html
class Jsp(java: Java, html: Html)
class Servlet(java: Java)
class SSM(servlet: Servlet, jsp: Jsp)
class SpringBoot(ssm: SSM)
```

```
// 各个耗时学习过程
suspend fun learnJava(): Java {
    delay(500L)
    showOff("Java")
```

```
    return Java()
}
suspend fun learnHtml(): Html {
    delay(500L)
    showOff("Html")
    return Html()
}
suspend fun learnJsp(java: Java, html: Html): Jsp {
    delay(500L)
    showOff("Jsp")
    return Jsp(java, html)
}
suspend fun learnServlet(java: Java): Servlet {
    delay(500L)
    showOff("Servlet")
    return Servlet(java)
}
suspend fun learnSSM(servlet: Servlet, jsp: Jsp): SSM {
    delay(500L)
    showOff("SSM")
    return SSM(servlet, jsp)
}
suspend fun learnSpringBoot(ssm: SSM): SpringBoot {
    delay(500L)
    showOff("SpringBoot")
    return SpringBoot(ssm)
}
```

主函数：

```
private var startTime: Long = 0
@JvmStatic
fun main(args: Array<String>) {
    GlobalScope.launch {
        val time = measureTimeMillis {
            startTime = System.currentTimeMillis()
            val javaDeferred = async { learnJava() }
            val htmlDeferred = async { learnHtml() }
            val jspDeferred = async { learnJsp(javaDeferred.await(),
htmlDeferred.await()) }
            val servletDeferred = async {
learnServlet(javaDeferred.await()) }
```

```
            val ssm = learnSSM(servletDeferred.await(), jspDeferred.await())
            val springBoot = learnSpringBoot(ssm)
        }
        println(time)
    }
    Thread.sleep(4000L)
}
fun showOff(obj: Any) {
    println("$obj : ${System.currentTimeMillis() - startTime}")
}
```

运行结果：

```
Java : 561
Html : 561
Jsp : 1093
Servlet : 1093
SSM : 1618
SpringBoot : 2134
Total time cost : 2134 ms
```

不难看出，归功于非阻塞挂起的特性，协程在进行复杂场景的异步请求的时候仍然可以做到逻辑清晰、易于维护。同时，在效率上它也可以高效地并发，减少等待时间。

尝试用onSuccess(result: Result)的写法实现一下？