kotlin协程是什么？

核心就是一段程序能够被挂起，并在稍后再在挂起的位置恢复。

Kotlin协程是依赖于线程池API的，在一个线程可以创建多个协程，并且协程运行时并不会阻塞当前线程。

其最大的特点就是能够以阻塞（同步）方式写出非阻塞（异步）的代码。

挂起：表示协程从当前执行的线程中暂时离开，在协程任务执行完成后，再恢复原来执行的线程继续执行。

kotlin协程怎么实现挂起？

suspend挂起函数：使用suspend关键字修饰的函数就被称为挂起函数。这个suspend关键字的作用就是告诉编译器，这是一个挂起函数，仅作一个标记，提醒这可能是一个异步耗时操作，会挂起协程。

挂起函数被限制在只能在协程作用域CoroutineScope、协程内、另一个suspend挂起函数内被调用。

比如官方API中的delay函数就是个顶层的suspend函数，不会阻塞线程，将协程挂起延迟指定时间，然后恢复协程。

 public suspend fun delay(time: Long) {

     if (time <= 0) return

     return suspendCancellableCoroutine {

         //由调度器具体实现，通常在线程内部sleep指定时间后恢复

         scheduleResumeAfterDelay(time, it)

    }

 }

在编译器内，suspend修饰的函数，以及调用suspend函数的地方都还会在左侧有个挂起点的标识

但suspend修饰的函数就并不一定会有挂起操作，协程的挂起是由内部框架执行的，suspend关键字本身只是个提示作用。

如果suspend函数内部没有挂起逻辑，编译器也会提示redundant suspend modifier警告，表示这个suspend关键字是多余的。

kotlin协程怎么使用？

 val coroutine = suspend {

     //模拟异步任务

    ...

     println("create coroutine")

     "result"

 }.createCoroutine(object : Continuation<String>{

     override val context: CoroutineContext = Job()

 ​

     override fun resumeWith(result: Result<String>) {

         println("Coroutine Result $result")

    }

 })

 //启动协程

 coroutine.resumeWith(Result.success(Unit))

suspend修饰函数作为接收者的拓展函数createCoroutine，创建协程，调用resultWith启动协程。

 public fun <T> (suspend () -> T).createCoroutine(

     completion: Continuation<T>

 ): Continuation<Unit> =

  //只是一个马甲，把协程回调值替换为unit

     SafeContinuation(

         //创建协程，并调用拦截器方法

         createCoroutineUnintercepted(completion).intercepted(),

         COROUTINE\_SUSPENDED

    )

 ​

而通常我们创建协程都是要让协程直接启动的，所有官方还提供了另一种拓展函数。

 public fun <T> (suspend () -> T).startCoroutine(

     completion: Continuation<T>

 ) {

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

 }

最常用的方法还是使用CoroutineScope的拓展函数launch与async来创建、启动协程。

launch函数会创建新协程，立即执行，且不会阻塞当前线程

通常是用于启动不需要返回值的新任务。

 //Builders.common.kt

 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

 }

该函数会返回一个Job对象是协程的句柄，可用于管理协程的开启与关闭。

 fun testCoroutineLaunch() = runBlocking {

     val job = launch{

         println("running launch")

        ...

    }

 }

参数block就是以CoroutineScope为接收者的函数类型。

async函数同样会创建新协程并立即执行，不会阻塞当前线程。

 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

 }

与launch的唯一区别是，该函数会返回一个Deferred对象（Job子类），可以通过调用await函数挂起协程，直到协程执行完成，返回执行结果。（不阻塞当前线程）

async方法构建的协程，支持并发

async协程不调用await函数也会开始执行协程

 fun testAsync() = runBlocking {

     val startTime = System.currentTimeMillis()

     val task1 = async {

         delay(100)

         println("task1 running print")

         1

    }

     val task2 = async {

         delay(150)

         println("task2 running print")

         2

    }

     task1.await()

     task2.await()

     val endTime = System.currentTimeMillis()

     println("async task over ${endTime - startTime}")

 }

 ​

 task1 running print

 task2 running print

 async task over 164 //总计时间表明两个协程为并发关系

此外官方还提供了awaitAll函数，可以将相同类型返回值的协程同步挂起并返回List

 val taskList = listOf(task1,task2)

 taskList.awaitAll() ​

 awaitAll(task1,task2)

任何创建的协程都会被该 scope 追踪，Kotlin禁止创建不能够被追踪的协程，从而避免协程泄漏

还有一种仅在单元测试时使用的协程创建方式runBlocking。

runBlock以阻塞当前线程的方式创建一个新协程作用域，直到协程体内执行完成

 public fun <T> runBlocking(

     context: CoroutineContext = EmptyCoroutineContext,

     block: suspend CoroutineScope.() -> T

 ): T {

    ...

 }

Android如何执行回调操作？

 val handler = Handler()

 getTestData(object : CallBack{

  override fun onSuccess(result : String){

          ...

          handler.post(...)

      }

 })

在线程池开启一个线程，然后在异步任务结束后，要有个恢复的动作，将线程切换回UI主线程更新UI。而这种切换线程恢复的动作，在以前都会演变成回调的方式。

如果此时需要顺序触发下一个异步任务，叠加一层回调，依次类推，也就演变成了俗称的"回调地狱"。

官方提供了suspendCoroutine和suspendCancellableCoroutine来将回调API转化为suspend的函数。

suspendCoroutine

 //Continuation.kt

 public suspend inline fun <T> suspendCoroutine(

     crossinline block: (Continuation<T>) -> Unit

 ): T {

    ...

     return suspendCoroutineUninterceptedOrReturn { c: Continuation<T> ->

         val safe = SafeContinuation(c.intercepted())

         block(safe)

         safe.getOrThrow()

    }

 }

suspendCancellableCoroutine

 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()

    }

其中suspendCancellableCoroutine是最为常用的，也是官方推荐使用转换回调API的方式。相比suspendCoroutine新增了 可取消的能力。

内部调用的suspendCoroutineUninterceptedOrReturn函数是编译器内部字节码的函数，没有源码，从注释上来看，作用就是拿到Continuation实例

这两个函数的参数block都是以Continuation（CancellableContinuation是其子类）作为参数的高阶函数。

 public interface Continuation<in T> {

     public val context: CoroutineContext

     public fun resumeWith(result: Result<T>)

 }

在函数体内调用resumeWith函数，接收Result类型参数，即恢复协程挂起。

官方还提供了拓展函数resume和resumeWithException来更方便的恢复协程

Retrofit如何支持kotlin协程？

 //KotlinExtensions.kt

 suspend fun <T : Any> Call<T>.await(): T {

    ...

     return suspendCancellableCoroutine { continuation ->

      ...

         enqueue(object : Callback<T> {

            ...

             if (response.isSuccessful) {

               val body = response.body()

               if (body == null) {

                  ...

                   continuation.resumeWithException(e)

              }else{

                   continuation.resume(body)

              }

            }

            ...

        }

    }

 }

其实同样也是利用suspendCancellableCoroutine，调用resume方法进行网络请求回调。

观察launch、async、suspendCoroutine、suspendCancellableCoroutine函数，其实都可以算作是对标准协程构建的封装，最终会调用Continuation的resumeWith方法来恢复协程。

kotlin协程的挂起后恢复本质上还是回调，只是把这部分代码隐藏在内部编译器（字节码），实现了一个有限状态机，让外部的异步代码看起来像是同步一样。

CorroutineContext 是什么？

launch、async、runBlocking等函数都接收CorroutineContext类型的参数。

 public interface CoroutineContext {

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

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

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

     public interface Key<E : Element>

     public interface Element : CoroutineContext {

      public val key: Key<\*>

        ...

    }

 }

CorroutineContext其实是一个包含了用户定义的一些各种不同元素的Element对象集合

内部实现为单链表。

每一种Element都一个唯一key。

作为协程的持久上下文， 其允许定义协程的行为：

Job：控制协程的生命周期。

CoroutineDispatcher：将工作分派到适当的线程。

CoroutineName：协程的名称，可用于调试。

CoroutineExceptionHandler：处理未捕获的异常。

Job 是什么？

Job 作为协程的句柄，能够管理协程的生命周期

对于每一个您所创建的协程 (通过launch或者 async)，它会返回一个 Job 实例

 public interface Job : CoroutineContext.Element {

    ...

     //启动协程，如果已经启动则返回false

     public fun start(): Boolean

     //取消协程

     public fun cancel(cause: CancellationException? = null)

     //挂起协程，直到此job执行完成

     public suspend fun join()

     //设置协程执行完成后的通知回调

     public fun invokeOnCompletion(

         handler: CompletionHandler

    ): DisposableHandle

 }

每个Job内部都包含一系列生命周期状态机制

New ：新创建，未开始执行

Active ：调用start开启协程，使协程处于活跃状态

Completing ：当前协程已完成，等待子协程的执行完成

Completed : 协程已完成已结束

Cancelling : 处于取消中状态，出现异常或者调用cancel时，等待子协程结束

Canceled：已取消

这些生命周期状态并不能直接访问，但Job提供了间接访问属性

 //是否处于活跃状态（Active）

 public val isActive: Boolean

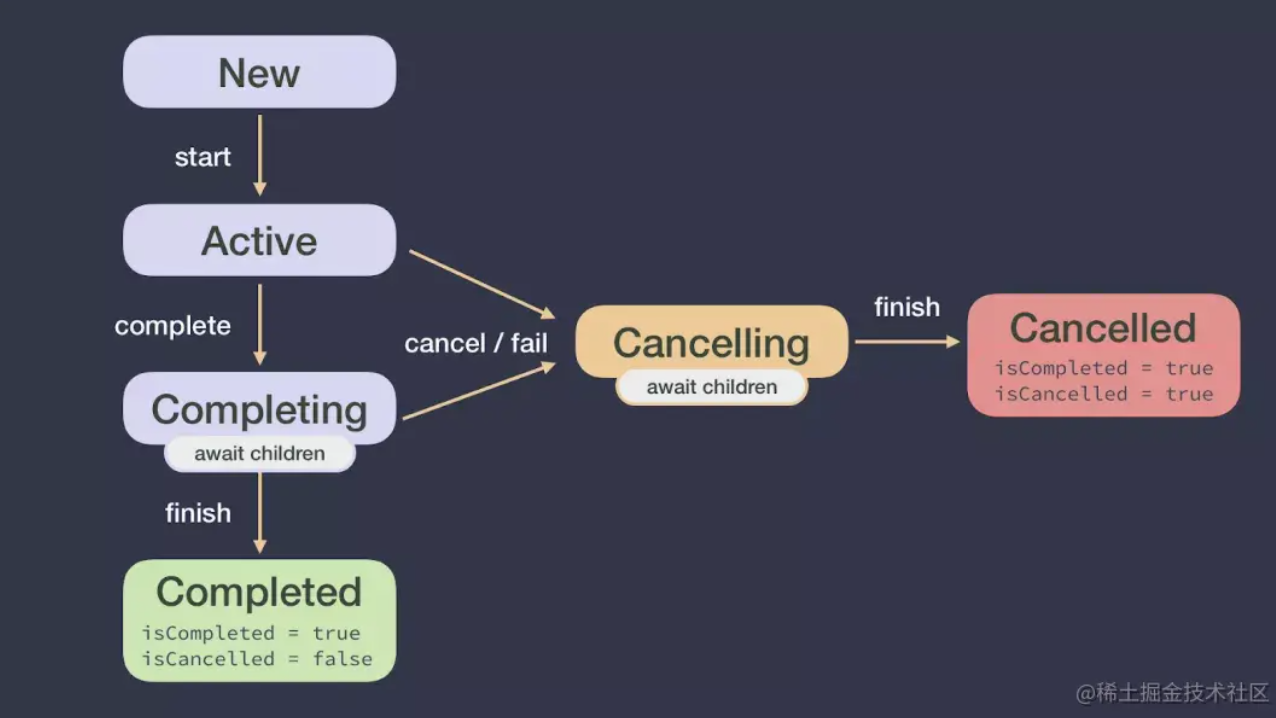
 //是否已完成

 public val isCompleted: Boolean

 //是否已取消

 public val isCancelled: Boolean

Job的生命周期状态流转图（来自官网）



async方法创建协程返回的Deferre继承自Job。拥有相同的状态机制， 除此之外，Deferre能够在协程执行完后调用await函数获取返回值。

 public interface Deferred<out T> : Job {

     //启动协程，并等待协程执行完成返回结果

     //如果异常结束则会抛出异常；如果协程尚未完成，则挂起直到协程执行完成。

     public suspend fun await(): T

 }

CoroutineName是用户用来指定的协程名称的，方便调试和定位问题

 //CoroutineName.kt

 public data class CoroutineName(

     //定义协程的名字

     val name: String

 ) : AbstractCoroutineContextElement(CoroutineName) {

     //CoroutineName实例在协程上下文中的key

     public companion object Key : CoroutineContext.Key<CoroutineName>

 }

协程调度器顾名思义就是调度协程在哪个线程上执行，底层是基于线程池，而Android的UI线程就是基于Handler。

 //CoroutineDispatcher.kt

 public abstract class CoroutineDispatcher :

     AbstractCoroutineContextElement(ContinuationInterceptor), ContinuationInterceptor {

    ...

     //执行调度任务

     public abstract fun dispatch(context: CoroutineContext, block: Runnable)

    ...

     //在协程恢复时拦截，加上个continuation马甲

     public final override fun <T> interceptContinuation(

  continuation: Continuation<T>

  ): Continuation<T> = DispatchedContinuation(this, continuation)

 }

官方API提供了四种默认调度器

Dispatchers

Default ：默认的后台计算线程池

IO : 适合 IO 密集型的任务线程池，比如：读写文件，操作数据库以及网络请求

Main : 主线程

Unconfined ：当前默认的协程中运行。（但是在遇到第一个暂停点之后，恢复的线程是不确定的。所以对于 Unconfined 其实是无法保证全都在当前线程中调用的。）

其中Default和IO在内部会优化共享同一个线程池,避免频繁切换线程造成的资源损耗。

同时还提供了个顶层suspend函数withContext用于切换协程内部运行的调度器

 public suspend fun <T> withContext(

     //协程上下文，通常可用于设置调度器

     context: CoroutineContext,

     //协程体

     block: suspend CoroutineScope.() -> T

 ): T {

...

 }

可以只在外部调用 withContext只切换一次线程，这样可以在多次调用的情况下，以尽可能避免了线程切换所带来的性能损失。

自定义调度器

官方提供了ExecutorService的拓展函数，可以很方便的将线程池转化为Dispatcher

 //Executors.kt

 public fun ExecutorService.asCoroutineDispatcher(): ExecutorCoroutineDispatcher

  =  ExecutorCoroutineDispatcherImpl(this)

CoroutineDispatcher是ContinuationInterceptor的子类

 //ContinuationInterceptor.kt

 public interface ContinuationInterceptor : CoroutineContext.Element {

     //拦截器的key,用于CoroutionContext获取拦截器

     companion object Key : CoroutineContext.Key<ContinuationInterceptor>

     public fun <T> interceptContinuation(

         continuation: Continuation<T>

    ): Continuation<T>

 }

 ​

 //Continuationlmpl.kt

 internal abstract class ContinuationImpl(

     completion: Continuation<Any?>?,

    ...

 ) : BaseContinuationImpl(completion){

    ...

     public fun intercepted(): Continuation<Any?> =

         intercepted?: (

             //获取拦截器对象

     context[ContinuationInterceptor]

  ?.interceptContinuation(this)

  ?: this

        ).also { intercepted = it }

 }

 ​

 //IntrinsicsJvm.kt

 //在创建协程时，协程体恢复时被调用

 public actual fun <T> Continuation<T>.intercepted()

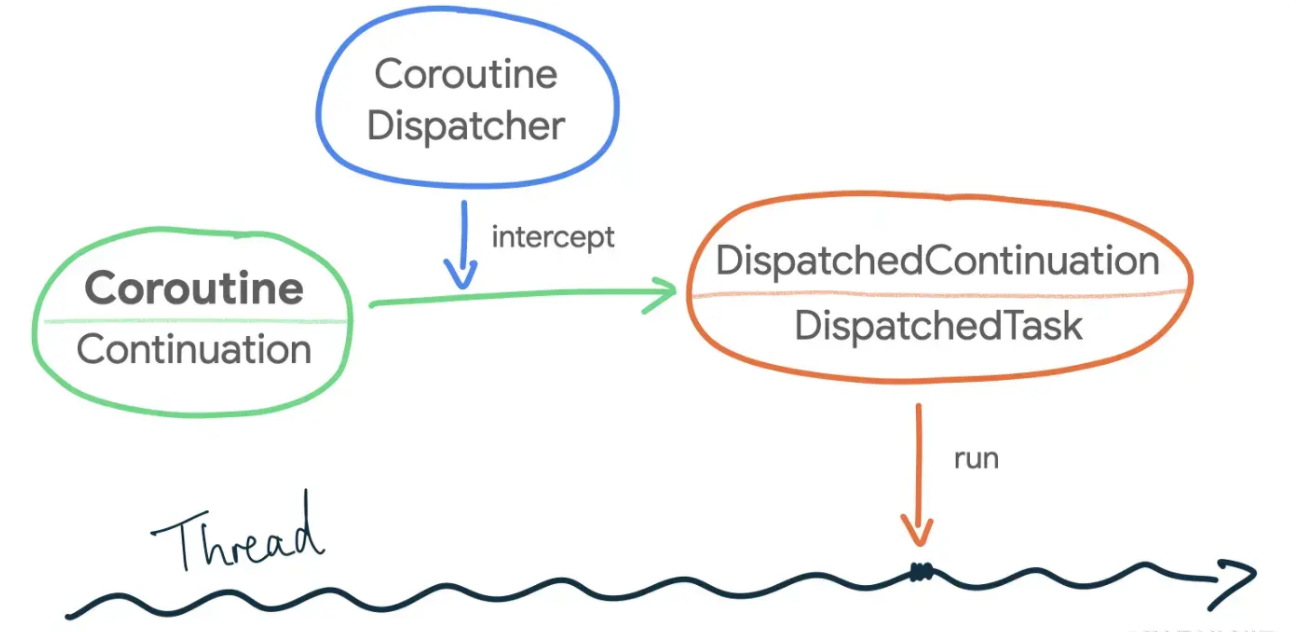
 : Continuation<T> = (this as? ContinuationImpl)?.intercepted() ?: this

也就是前面创建协程时，其内部马甲Coroutine 类中获取到Coroutine对象都是经过拦截器拦截后的Coroutine对象。

其就类似于okHttp中的拦截器的作用。所不同的是，拦截器在CoroutineContext中只能存在一个。

调度器的原理是什么？

引用官方的一张图，清晰表述了协程的线程调度流程



前面提到，调度器实现的interceptContinuation函数，将原始Coroutine对象包装成DispatchedContinuation。

 //DispatchedContinuation.kt

 internal class DispatchedContinuation<in T>(

     @JvmField val dispatcher: CoroutineDispatcher,

     @JvmField val continuation: Continuation<T>

 ) : DispatchedTask<T>(MODE\_UNINITIALIZED), CoroutineStackFrame, Continuation<T> by continuation {

    ...

     override fun resumeWith(result: Result<T>) {

        ...

         //判断是否需要进行线程调度

         if (dispatcher.isDispatchNeeded(context)) {

             dispatcher.dispatch(context, this)

        }

        ...

    }

 }

 ​

 // SchedulerTask.kt

 //继承自Runnable

 internal actual typealias SchedulerTask = Task

 //DispatchedTask.kt

 internal abstract class DispatchedTask<in T>(

     @JvmField public var resumeMode: Int

 ) : SchedulerTask() {

     internal open fun <T> getSuccessfulResult(state: Any?): T =

         state as T

     //task实际上就是的runnable

  public final override fun run(){

        ...

         //实际的continuation

         val continuation = delegate.continuation

        ...

         //恢复协程

         continuation.resume(getSuccessfulResult(state))

    }

 }

实际上这就是个马甲类，对原始continuation判断是否需要进行线程调度。

以DEFAULT默认调度器来说，其可以与IO公用一个公共线程池

 internal object CommonPool : ExecutorCoroutineDispatcher() {

     @Volatile

     private var pool: Executor? = null

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

         try {

            (pool ?: getOrCreatePoolSync()).execute(wrapTask(block))

        } catch (e: RejectedExecutionException) {

             unTrackTask()

             DefaultExecutor.enqueue(block)

        }

    }

     @Synchronized

     private fun getOrCreatePoolSync(): Executor =

         pool ?: createPool().also { pool = it }

 }

所以协程对于线程的调度，底层依然是依赖线程池API。

至于Dispatchers.Main也是同理，使用主线程的Handler，这里就不多赘述了。

如何组合上下文元素？

如果需要为协程定义多个元素，则可以使用+运算符进行合并。比如同时设置Job、调度器、协程名称

 val context = Job() + Dispatchers.Main + CoroutineName("name")

CoroutineScope 是什么？

CoroutineScope定义了协程所运行在的作用域，所有协程都必须在作用域内启动。

 public interface CoroutineScope {

     public val coroutineContext: CoroutineContext

 }

可管理作用域内运行的任务，通过调用 scope.cancel()来取消正在进行的任务

 public fun CoroutineScope.cancel(cause: CancellationException? = null) {

     //实际还是调用context内的Job对象管理

     val job = coroutineContext[Job] ?: ...

     job.cancel(cause)

 }

官方提供了 KTX 库在一些类的生命周期里提供了CoroutineScope，比如 在ViewModel内的viewModelScope 和 LifecycleOwner的lifecycleScope，都会在各自生命周期结束时取消作用域内的协程。

还有个全局协程作用域GlobalScope，但不推荐使用，不会继承外部作用域且无法自定义CoroutineContext。

如果需要全局的驻留任务，最好还是在Application内构建自定义的CoroutineScope。

作用域层级是什么？

在CoroutineScope中可以创建协程，而在协程体内也隐含了当前协程所处的CorroutinScope。

 val scope = CoroutineScope(Job() + Dispatchers.Main)

 ​

 val job = scope.launch {

     // 新的协程会将 CoroutineScope 作为父级

    ...

     val result = launch {

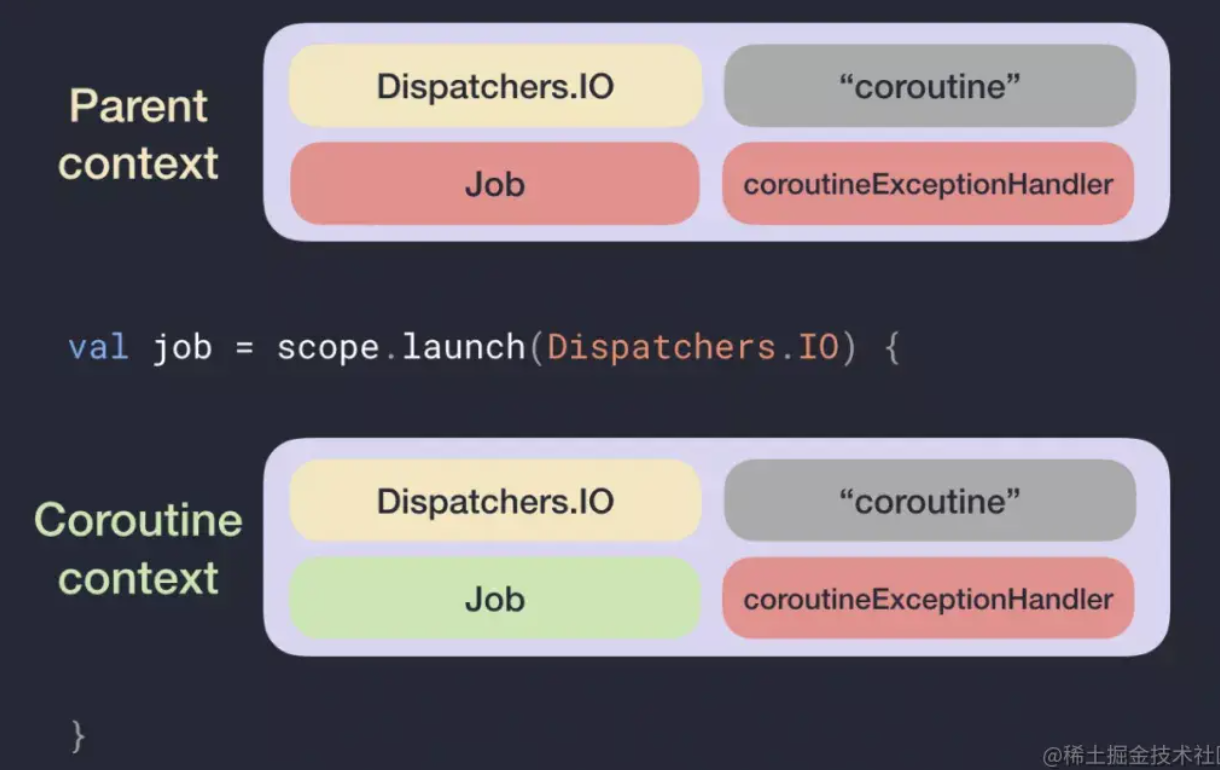
         // launch创建的新协程会将当前协程作为父级

        ...

    }

 }

以最开始的CoroutineScope为根节点层级，在哪个CoroutineScope中创建新协程，就是新的协程的父级。



但使用最根节点CoroutinesScop的创建的新协程的CoroutineContext实际是

新的 CoroutineContext = 父级 CoroutineContext + 新context （默认只创建Job）

父级 CoroutineContext 里的 Job 是 scope 对象的 Job (红色) ，

而新的 Job 实例 (绿色) 会赋值给新的协程的 CoroutineContext。

在新协程的范围内，会覆盖父级CoroutineContext的Job对象。

父协程只能在全部子协程执行完成后才会进去完成状态，即使父协程本身的任务已经执行完成。

顶级作用域 ：

没有父协程的协程所在的作用域称之为顶级作用域。

协同作用域 ：

在协程中启动一个协程，新协程为所在协程的子协程。子协程所在的作用域默认为协同作用域。此时子协程抛出未捕获的异常时，会将异常传递给父协程处理，如果父协程被取消，则所有子协程同时也会被取消。

主从作用域 ：

与协同作用域在协程的父子关系上一致，区别在于，处于该作用域下的协程出现未捕获的异常时，不会将异常向上传递给父协程。

官方还提供了两种在协程内部创建作用域的API：

coroutineScope

coroutineScope是顶层suspend函数，创建一个新的协程作用域，并调用指定的协程代码块，等待内部协程结束后再结束作用域，属于协同作用域。

 public suspend fun <R> coroutineScope(

     block: suspend CoroutineScope.() -> R

 ): R {

    ...

 }

supervisorScope

supervisorScope是顶层suspend函数，与coroutineScope的区别就是协程作用域在取消\异常时不会自动传递到父协程层级，属于主从作用域。

 public suspend fun <R> supervisorScope(block: suspend CoroutineScope.() -> R): R {

    ...

 }

协程的启动模式分类

在launch和async函数的start参数中，允许接收枚举类CoroutineStart

 public enum class CoroutineStart {

     DEFAULT,

     LAZY,

     ATOMIC,

     UNDISPATCHED;

 }

有可能在执行前被取消。

ATOMIC ：创建协程后，立即调度执行，协程执行到第一个挂起点之前，不响应取消，协程一定会被执行（执行途中可能会被取消）。

LAZY：如果调度前被取消了，直接进入异常结束状态，且不调用start、await等方法是不会执行的。

UNDISPATCHED : 协程在这种模式下会直接开始在当前线程下执行，直到运行到第一个挂起点。这听起来有点像 ATOMIC，不同之处在于UNDISPATCHED是不经过任何调度器就开始执行的。当然遇到挂起点之后的执行，将取决于挂起点本身的逻辑和协程上下文中的调度器。

还记得launch和async函数内部调用的Coroutine.start吗？

其内部最终会调用到CoroutineStart的invoke方法

 //AbstractCoroutine.kt

 public fun <R> start(

     start: CoroutineStart,

     receiver: R,

     block: suspend R.() -> T

 ) {

     start(block, receiver, this)

 }

 ​

 //CoroutineStart.kt

 @InternalCoroutinesApi

 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

 }

以默认的DEFAULT模式为例，调用startCoroutineCancellable方法来启动协程

 //Cancellable.kt

 public fun <T> (suspend () -> T).startCoroutineCancellable(

     completion: Continuation<T>

 ): Unit = runSafely(completion) {

     //创建Continuation

  createCoroutineUnintercepted(completion)

    //拦截器拦截

        .intercepted()

    //调用resumeWith启动协程

        .resumeCancellableWith(Result.success(Unit))

 }

是不是很眼熟？其实就是对于标准协程的拓展封装，其内部依然是围绕Continuation来启动协程。

协程怎么取消？

前面提到CoroutineScope能统一管理作用域内的协程，最终是协程上下文的job对象，调用cancel方法来取消协程。

 //Job.kt

 public fun cancel(cause: CancellationException? = null)

该方法会使协程抛出一个特殊的CancellationException异常来结束协程操作。

同时作为参数cause可以传递指定结束原因，默认为null，使用默认的异常defaultCancellationException

 //JobSupport.kt

 internal inline fun defaultCancellationException(

  message: String? = null,

  cause: Throwable? = null

 ) = JobCancellationException(

  message ?: cancellationExceptionMessage(),

  cause, this

 )

 protected open fun cancellationExceptionMessage(): String

  = "Job was cancelled"

 ​

协程在取消时会轮询，所有子协程都将被取消

而被取消的协程作用域是不能再创建新协程的

被取消的协程并不会影响到相同层级的其他协程

 val scopeJob = CoroutineScope().launch{

     val job1 = launch{

        ...

    }

     val job2 = launch{

        ...

    }

     //只会取消job1协程

     job1.cancel()

 }

 ...

 //会取消作用域内的所有协程

 scopeJob.cancel()

虽说调用cancel方法会使协程进入取消流程。

但就与线程中执行Runnable类似，在协程开始运行后，取消协程并不意味着协程执行的任务也会随之停止。

 fun test() = runBlocking{

     val startTime = System.currentTimeMillis()

     val job = launch(Dispatchers.Default) {

         var nextPrintTime = startTime

         var i = 0

         while (i <= 5){

             if (System.currentTimeMillis() >= nextPrintTime){

                 println("job running print ${i++}")

                 nextPrintTime += 500

            }

        }

    }

     //程序执行1s

     delay(1000)

     println("coroutine scope delay done,wait cancel job")

     job.cancel()

     println("job canceled")

 }

 ​

 job running print 0

 job running print 1

 job running print 2

 coroutine scope delay done,wait cancel job

 job canceled

 job running print 3

 job running print 4

 job running print 5

就像上面的代码，cancel执行完成后，协程依然在运行。

如果需要让上面的代码按预期执行，则需要在协程体中，定时检查协程是否已被取消。

isActive

还记得Job提供的isActive属性吗？可以通过该属性检查协程是否处于活跃状态。

同时官方还提供了CoroutineScope的拓展函数可以检查协程活跃状态

 public val CoroutineScope.isActive: Boolean

     get() = coroutineContext[Job]?.isActive ?: true

另外Job还提供了更方便的拓展方法

 public fun Job.ensureActive(): Unit {

     if (!isActive) throw getCancellationException()

 } ​

 public fun CoroutineContext.ensureActive() {

     get(Job)?.ensureActive()

 }

yield

这个函数会挂起当前协程，让出线程资源去执行其他协程任务。

其内部首先会调用ensureActive方法检查协程的活跃状态

 //Yield.kt

 public suspend fun yield(): Unit = suspendCoroutineUninterceptedOrReturn { uCont ->

     val context = uCont.context

  //检查协程活跃状态

     context.ensureActive()

  ...                                                             }

join

Jop.join方法会挂起当前协程，直到协程执行完成

如果在调用cancel方法后再调用join方法，协程会处于挂起直到协程执行完成

如果先调用join再调用cancel，则不会产生影响，因为join执行后，协程就已经结束了。

await

Deferred.await方法挂起当前协程，直到协程执行完成后，返回协程结果内容。

如果在调用cancel方法后，再调用await方法，会抛出CancellationException异常（表示正常结束），结束协程。

如果先调用await再调用cancel，则不会产生影响，因为await执行后，协程就已经结束了。

协程的cancel 能否成功 仅仅取决于是否在协程体中加入了检查点，比如 isActive 、yield、delay等， 如果协程没有加入检查点，那么cancel 一定是无效的！

在协程被结束后，可以在finally代码块中，执行清理资源操作

 fun test() = runBlocking{

     val job = launch{

         try{

             //执行任务

             doSomeWork()

        }finally{

             //在协程结束后，执行的操作

             doCleanWork()

        }

    }

     println("协程已结束")

 }

但由于finally代码块是在协程结束后才会执行的，此时不能继续挂起协程。

此时如果还需要挂起，则可以指定context 为NonCancellable。

 finally{

     withContext(NonCancellable){

         println("job NonCancellable suspend finally")

         doCleanWork()

    }

 }

注意不能滥用NonCancellable，这会导致协程无法被取消，造成内存泄漏。

最好只用于处理一些资源回收操作

拓展

还记得在前面提到转换回调API的suspendCancellableCoroutine函数吗？

该函数的block参数类型是(CancellableContinuation<T>) -> Unit。

作为Continuation子类的CancellableContinuation，其内部提供了invokeOnCancellation函数，可以在协程被取消（或出现异常） 时，执行一些资源回收操作。

 //CancellableContinuation.kt

 public interface CancellableContinuation<in T> : Continuation<T> {

    ...

     public fun invokeOnCancellation(handler: CompletionHandler)

    ...

 }

 ​

 //CompletionHandler.common.kt

 public typealias CompletionHandler = (cause: Throwable?) -> Unit

比如在Retrofit中，就是在invokeOnCancellation中，出现异常时尝试结束Call。

 //KotlinExtensions.kt

suspend fun <T : Any> Call<T?>.await(): T? {

   return suspendCancellableCoroutine { continuation ->

     continuation.invokeOnCancellation {

       //取消网络请求

       cancel()

    }

    ...

  }

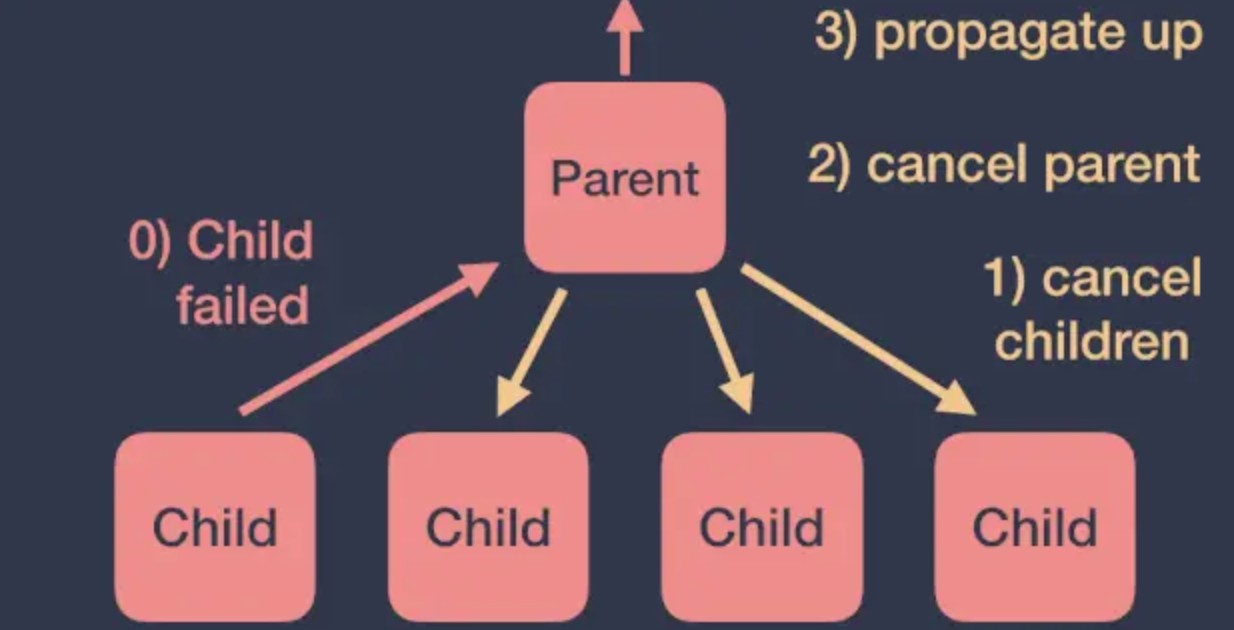
 }

协程异常传递

默认情况下，调用launch、或者async方法，会使用Job来创建协程。

而在协程内抛出异常结束时，且协程内部未捕获，会抛出异常给父协程，同时父协程内的所有子协程都将被取消，并且进一步向上一层级传递，并最终将异常传递给顶层的作用域

官方的示例图很形象的展示了这一过程



取消它自己的子级；

取消它自己；

将异常传递给它的父级。

 CoroutineScope().launch{

  ...

     lauch(CoroutineName("coroutine\_1")){

         //协程1出现异常

         throw Exception("...")

    }

     launch(CoroutineName("coroutine\_2")){

         //协程2在异常出现后，会被取消

        ...

    }

 }

而传递给顶层作用域CoroutineScope，将会把作用域内所有协程全部取消。

 //JobSupport.kt

 public open class JobSupport constructor(active: Boolean) : Job,...{

    ...

     //是否传递给父协程，结束同层级协程

     public open fun childCancelled(cause: Throwable): Boolean {

         if (cause is CancellationException) return true

         return cancelImpl(cause) && handlesException

    }

 }

协程作用域被取消后，无法继续开启新协程

SupervisorJob是什么？

如果我们不想因为一个任务的失败而影响其他任务，子协程运行失败不影响其他子协程和父协程，在协程创建时可以使用SupervisorJob

 //Supervisor.kt

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

 ​

 private class SupervisorJobImpl(parent: Job?) : JobImpl(parent) {

     //不会传递给父级

     override fun childCancelled(cause: Throwable): Boolean = false

 }

作为Job的子类，其重写了chileCanceled方法为false，表示不会传递异常到父类协程，只会在协程内部处理（结束当前协程），并结束内部子协程。

 lifecycleScope.launch(SupervisorJob()){

    ...

 }

当子协程任务出错或失败时，SupervisorJob 只会取消它和它自己的子级，也不会传播异常给它的父级，它会让子协程自己处理异常

而使用挂起函数supervisorScope创建的协程作用域，同样有SupervisorJob的作用。

 //Supervisor.kt

 public suspend fun <R> supervisorScope(block: suspend CoroutineScope.() -> R): R {

    ...

     return suspendCoroutineUninterceptedOrReturn { uCont ->

         val coroutine = SupervisorCoroutine(uCont.context, uCont)

         coroutine.startUndispatchedOrReturn(coroutine, block)

    }

 }

 ​

 private class SupervisorCoroutine<in T>(

     context: CoroutineContext,

     uCont: Continuation<T>

 ) : ScopeCoroutine<T>(context, uCont) {

     override fun childCancelled(cause: Throwable): Boolean = false

 }

但不论是Job还是SupervisorJob，只要内部没有捕获的异常都会被抛出，最终导致程序崩溃。

唯有协程取消时抛出的CancellationException异常是会被忽略的，这对于协程而言就意味着协程是正常结束或者退出。

launch

对于launch创建的协程，异常会在第一时间被抛出，可以直接使用try catch来捕获。

 fun test() = runBlocking{

     launch(CoroutineName("coroutine\_1")){

         try{

             throw NullPointException("...")

        }catch(e : Exception){

             //直接捕获异常

        }

    }

 }

但对于父协程内部开启的子协程，在父协程如果使用默认Job创建的子协程，会在子协程抛出异常后，直接将异常抛出到最顶层作用域，内部无法拦截。

 fun test() = runBlocking{

     launch(CoroutineName("coroutine\_1")){

         try{

             launch(CoroutineName("inner\_coroutine\_1")){

                 throw NullPointerException("...")

            }

        }catch(e : Exception){

             //无法捕获内部协程异常，会直接越过当前协程范围传播到父级作用域

        }

    }

 }

如果coroutine\_1使用SupervisorJob或者supervisorScope，则异常会被拦截在coroutine\_1范围内，不会延伸传播到runBlocking，只会在协程内部处理。

async

对于async创建的协程，若协程内没有捕获异常

如果没有调用await，外部try-catch是无法捕获到异常。

 fun test() = runBlocking{

     try{

         val task = async{

        throw NullPointerException("...")

    }

    }catch(e : Exception){

         //无法捕获到异常

    }

 }

而在调用了await后，async代码块本身是不会抛出异常，await本身会抛出异常。

 fun test() = runBlocking{

     val task = async{

         throw NullPointerException("...")

    }

     try{

         task.await()

    }catch(e : Exception){

         //能够捕获到异常

    }

 }

在线程中有Thread.setDefaultUncaughtExceptionHandler 方法来处理未知异常。Kotlin协程中同样有相同的机制。

CoroutineExceptionHandler作为CoroutineContext中的一种，可以添加到CoroutineContext中，处理协程中未捕获的异常。

仅针对launch创建协程自动抛出的异常生效。

对于async创建协程是不会有任何效果，无法捕获。

CoroutineExceptionHandler是兜底机制，无法从异常中恢复协程！

通常用作记录异常、资源回收等操作。

对于默认的Job而言，在子协程内设置CoroutineExceptionHandler是无意义的，异常会被向上传递到父协程，由设置的CoroutineExceptionHandler处理，如果没有设置，则继续向上传递，根作用域。

 fun test() = runBlocking {

     val scope = CoroutineScope(Job())

     val handler = CoroutineExceptionHandler { coroutineContext, throwable ->

         println("handler cast $throwable")

    }

     scope.launch(handler){ //只有作用域的根协程，能拦截处理异常

         launch(handler){ //子协程无法拦截

             throw NullPointerException("...")

        }

         async(handler){ //async抛出的异常无法被handler捕获

             throw NullPointerException("...")

        }

    }

 }

如果内部子协程使用SupervisorJob，则异常就不会向上传递，直接在内部协程的CoroutineExceptionHandler捕获处理

 scope.launch(handler){

  launch(SupervisorJob()+handler){ //拦截在子协程内部的handler

    throw NullPointerException("...")

    }

 }

而使用supervisorScope创建的作用域，也有相同的作用。作用域根协程不会将异常向上传递，而是在协程内部的CoroutineExceptionHandler捕获处理

 supervisorScope{

     launch(handler){ //在协程内部handler捕获

         throw NullPointerException("...")

    }

 }

总结

在基于JVM的Android平台上，本质上还是通过调度器把协程任务挂起（切换）线程，将阻塞转移到另一个线程，然后在协程任务执行完成后恢复，回调到原来的线程。

只是Kotlin协程将实现细节隐藏在框架内部与编译器字节码中，把它”拍平“了，这才能直接把异步任务写成同步的方式。

使用CoroutineScope的launch和async来启动协程，便于在CoroutineScope统一管理内部协程。

launch 适合启动外部不需要返回值的协程

async 适合启动外部需要返回值的协程

在CoroutineContext中设置Dispatcher来让协程运行在指定线程。

可通过withContext来切换协程内代码块所运行的线程。

在Android中可以使用拓展库lifecycleScope和viewModelScope作用域管理协程。

作用域（父级协程）取消（异常）时，会取消所有子协程

作用域取消后无法创建新协程

父级协程需等待所有子协程执行完才能完成

默认情况下，子协程未捕获的异常会传递到父协程

对于异步任务，内部最好使用isActive、yield等检查协程状态，使其能够被取消。

对于回调API转化的协程，最好使用suspendCancellableCoroutine来创建可取消的协程（内部会检查协程状态）。

而当我们不希望协程出现异常时，自动传递到父级协程（无法拦截），造成同层级的其他协程被取消，可以在CoroutineContext中设置SupervisorJob，或者使用supervisorScope创建子协程作用域，将异常拦截在协程体内部。

最后，可以给CoroutineContext设置CoroutineExceptionHandler来作为最后的异常拦截器，处理一些出现异常后的资源回收操作。