协程和线程的差异

- 线程的目的是提高CPU资源使用率， 使多个任务得以并行的运行，是为了服务于机器的.
- 协程的目的是为了让多个任务之间更好的协作，主要体现在代码逻辑上，是为了服务开发者 (能提升资源的利用率, 但并不是原始目的)

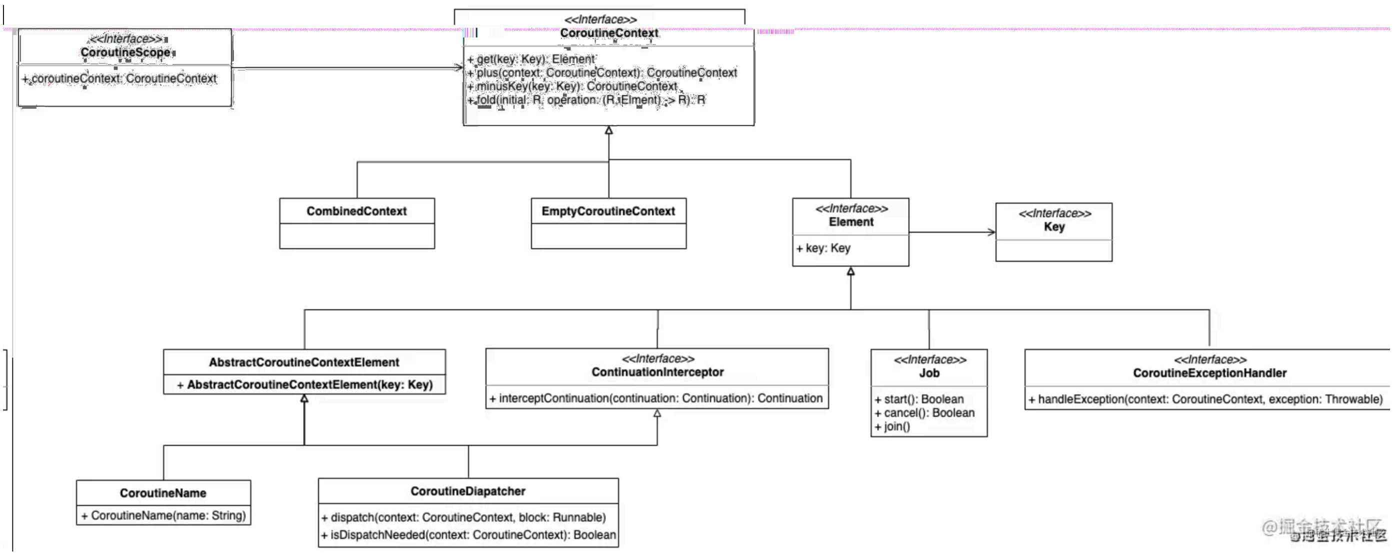
协程的核心竞争力

> 简化异步并发任务。

协程上下文 CoroutineContext

- 协程总是运行在一些以 `CoroutineContext` 类型为代表的上下文中 ， 协程上下文是各种不同元素的集合
- 集合内部的元素`Element`是根据`key`去对应（`Map`特点），但是不允许重复（`Set`特点）
- `Elemen`t之间可以通过+号进行组合
- `Element`有如下四类，共同组成了`CoroutineContext`
    - `Job`：协程的唯一标识，用来控制协程的生命周期(`new`、`active`、`completing`、`completed`、`cancelling`、`cancelled`)
    - `CoroutineDispatcher`：指定协程运行的线程(`IO`、`Default`、`Main`、`Unconfined`)
    - `CoroutineName`：指定协程的名称，默认为`coroutine`
    - `CoroutineExceptionHandler`：指定协程的异常处理器，用来处理未捕获的异常

它们的关系如图所示：



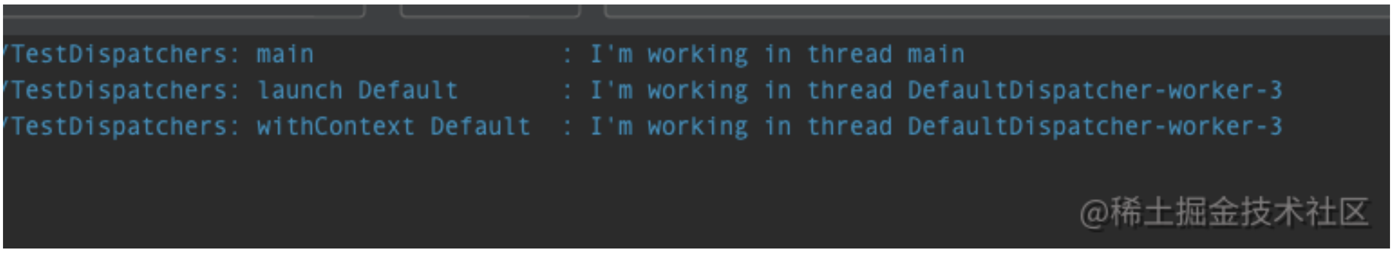
协程切换线程源码分析

我们在协程体内，可能通过`withContext`与`launch`方法简单便捷的切换线程，用同步的方式写异步代码，这也是`kotin`协程的主要优势之一

示例：

```kotlin
private fun testDispatchers() = runBlocking {

    Log.d(TAG, "main                 : I'm working in thread 
${Thread.currentThread().name}")

    launch(Dispatchers.Default) {
        Log.d(TAG, "launch Default       : I'm working in thread 
${Thread.currentThread().name}")
    }

    withContext(Dispatchers.Default) {
        Log.d(TAG, "withContext Default  : I'm working in thread 
${Thread.currentThread().name} ")
    }
}
```

输出结果为：



从输出结果可以看出，调用`Dispatch.Default`会由主线程切换到`DefaultDispatcher-worker-3`线程,而且`launch`和`withContext`切换的线程是相同的。

`launch`方法解析

协程的发起方式如下

```kotlin
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job {
    //创建协程上下文Context
    val newContext = newCoroutineContext(context)
    val coroutine = if (start.isLazy)
        LazyStandaloneCoroutine(newContext, block) else
        StandaloneCoroutine(newContext, active = true)
    //创建一个独立协程并启动
    coroutine.start(start, coroutine, block)
    return coroutine
}
```

`launch`方法主要作用：
1、是创建新的上下文`Context`
2、创建并启动协程

组合一个新的`Context`

```
public actual fun CoroutineScope.newCoroutineContext(context: CoroutineContext): CoroutineContext {
    //根据传入的Context 组合成新的上下文
    val combined = coroutineContext + context
    val debug = if (DEBUG) combined + CoroutineId(COROUTINE_ID.incrementAndGet()) else combined
    //如果发起的时候没有传入调度器，则使用默认的Default
    return if (combined !== Dispatchers.Default && combined[ContinuationInterceptor] == null)
        debug + Dispatchers.Default else debug
}
```

从上述方法中能够得出，此方法主要是
1、将`launch`方法传入的`context`与`CoroutineScope`中的`context`组合起来
2、若`combined`中没传入一个调度器，则会默认使用`Dispatchers.Default`调度器

创建一个独立协程`Coroutine`

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

//继承抽象协程类
private open class StandaloneCoroutine(
    parentContext: CoroutineContext,
    active: Boolean
) : AbstractCoroutine<Unit>(parentContext, active) {
   //省略......
}

//AbstractCoroutine类核心源码
public fun <R> start(start: CoroutineStart, receiver: R, block: suspend R.() -> T){
    initParentJob()
    start(block, receiver, this)
}

// CoroutineStart类核心源码
public operator fun <T> invoke(block: suspend R.() -> T, receiver: R, completion: Continuation<T>)
    when (this) {
        //launch 默认为DEFAULT
```

```
        CoroutineStart.DEFAULT -> block.startCoroutineCancellable(completion)
        CoroutineStart.ATOMIC -> block.startCoroutine(completion)
        CoroutineStart.UNDISPATCHED -> block.startCoroutineUndispatched(completion)
        CoroutineStart.LAZY -> Unit // will start lazily
    }
```

创建一个协程体 `Continuation`

```
internal fun <R, T> (suspend (R) -> T).startCoroutineCancellable(receiver: R,
completion: Continuation<T>) =
    runSafely(completion) {
        createCoroutineUnintercepted(receiver, completion)
            //如果需要则进行拦截处理
            .intercepted()
            //调用 resumeWith 方法
            .resumeCancellableWith(Result.success(Unit))
    }
```

调用`createCoroutineUnintercepted`,会把我们的协程体即`suspend block`转换成`Continuation`

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

//ContinuationImpl类核心源码
public fun intercepted(): Continuation<Any?> =
        intercepted
            ?: (context[ContinuationInterceptor]?.interceptContinuation(this) ?: this)
                .also { intercepted = it }

//CoroutineDispatcher类核心源码
public final override fun <T> interceptContinuation(continuation: Continuation<T>):
Continuation<T> =
      DispatchedContinuation(this, continuation)
```

从上述方法可以得出
1.`interepted`是个扩展方法，最后会调用到`ContinuationImpl.intercepted`方法
2.在`intercepted`会利用`CoroutineContext`，获取当前的调度器
3.当前调度器是`CoroutineDispatcher`，最终会返回一个`DispatchedContinuation`，我们也是利用它来实现线程切换的

调度处理

```
//DispatchedContinuation
public fun <T> Continuation<T>.resumeCancellableWith(result: Result<T>) = when (this) {
    is DispatchedContinuation -> resumeCancellableWith(result)
    else -> resumeWith(result)
}


@Suppress("NOTHING_TO_INLINE")
inline fun resumeCancellableWith(result: Result<T>) {
    val state = result.toState()
    //判断是否需要切换线程
    if (dispatcher.isDispatchNeeded(context)) {
        _state = state
        resumeMode = MODE_CANCELLABLE
        //调用器进行切换线程
        dispatcher.dispatch(context, this)
    } else {
        //Unconfined，会执行该方法
        executeUnconfined(state, MODE_CANCELLABLE) {
            if (!resumeCancelled()) {
                resumeUndispatchedWith(result)
            }
        }
    }
}
```

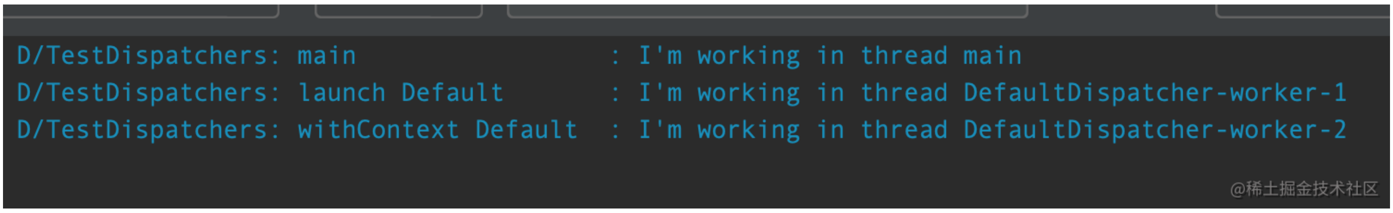
上述分析可得出
1、判断是否需要切换线程，如果需要则调用`dispatcher.dispatch()`方法进行切换线程
2、如果不需要切换线程， 则直接在原有线程执行。

`withContext`方法解析

```
public suspend fun <T> withContext(
    context: CoroutineContext,
    block: suspend CoroutineScope.() -> T
): T = suspendCoroutineUninterceptedOrReturn sc@ { uCont ->

    //创建新的content
    val oldContext = uCont.context
    val newContext = oldContext + context

    ......

    //创建新的调度协程
    val coroutine = DispatchedCoroutine(newContext, uCont)
```

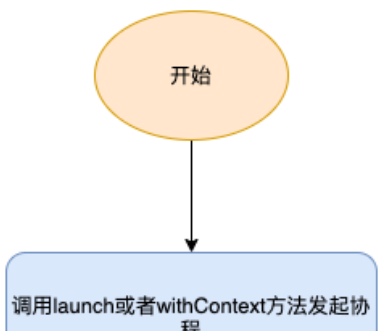
```
    //初始化父类Job
    coroutine.initParentJob()
    //开始一个可以取消的协程
    block.startCoroutineCancellable(coroutine, coroutine)
    coroutine.getResult()
}

private class DispatchedCoroutine<in T>(
    context: CoroutineContext,
    uCont: Continuation<T>
) : ScopeCoroutine<T>(context, uCont) {

    //在complete时会会回调
    override fun afterCompletion(state: Any?) {
        afterResume(state)
    }

    override fun afterResume(state: Any?) {
        //uCont就是父协程, context仍是老版context,因此可以切换回原来的线程上
        uCont.intercepted().resumeCancellableWith(recoverResult(state, uCont))
    }
}
```

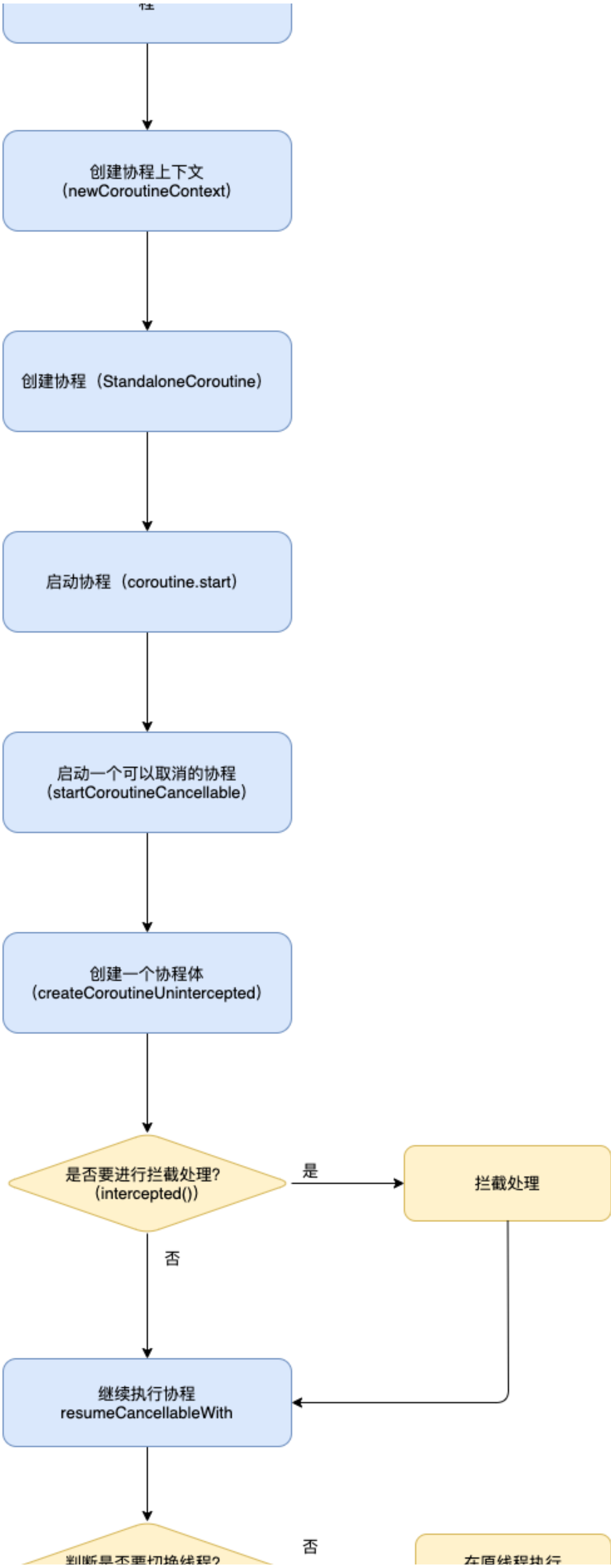
从上述方法可以得出，调用`withContext`方法最终也是调用`uCont.intercepted().resumeCancellableWith`方法与`launch`方法最后切换线程是相同的，
这里也说明了上面输出结果，为什么二者调用同一调度器切换的线程是相同的。
也有不相同的时候，就是当线程`DefaultDispatcher-worker-1`还没创建成功的时候,`withContext`已经需要切换线程时，会再创建一个新的线程，如下图所示



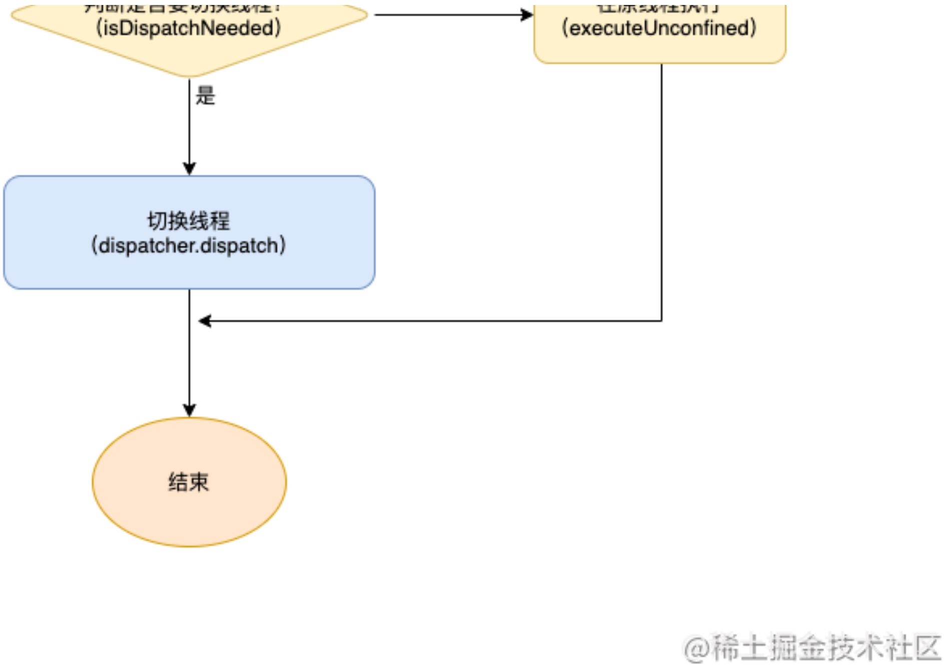
```
D/TestDispatchers: main                  : I'm working in thread main
D/TestDispatchers: launch Default        : I'm working in thread DefaultDispatcher-worker-1
D/TestDispatchers: withContext Default   : I'm working in thread DefaultDispatcher-worker-2
```


其切换线程的流程图为：





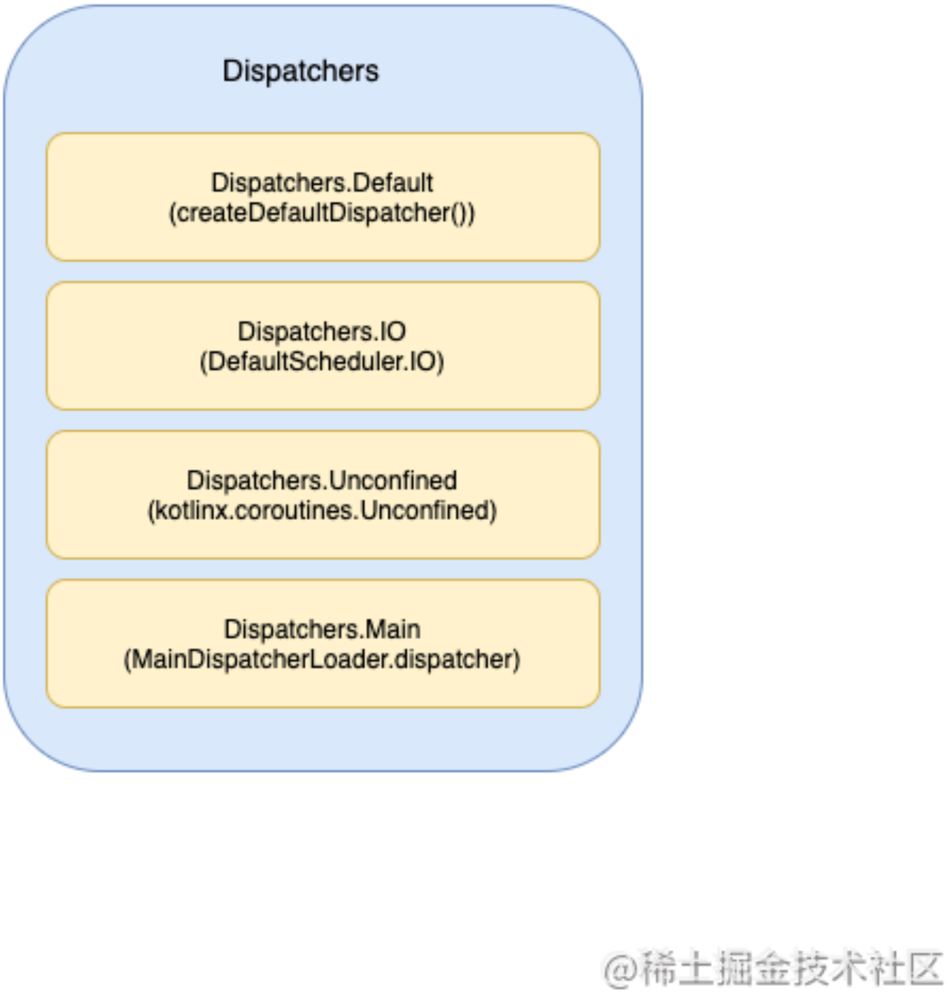
创建协程上下文
（newCoroutineContext）
创建协程（StandaloneCoroutine）
启动协程（coroutine.start）
启动一个可以取消的协程
（startCoroutineCancellable）
创建一个协程体
（createCoroutineUnintercepted）
是否要进行拦截处理?
（intercepted()）
是
拦截处理
否
继续执行协程
resumeCancellableWith
否



CoroutineDispatcher 作用

- 用于指定协程的运行线程
- `kotlin`已经内置了`CoroutineDispatcher`的4个实现，分别为 `Dispatchers`的 `Default`、`IO`、`Main`、`Unconfined`字段

```
public actual object Dispatchers {

    @JvmStatic
    public actual val Default: CoroutineDispatcher = createDefaultDispatcher()

    @JvmStatic
    public val IO: CoroutineDispatcher = DefaultScheduler.IO

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

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



Dispatchers.Default

`Default`根据`useCoroutinesScheduler`属性（默认为`true`） 去获取对应的线程池

- `DefaultScheduler ：Kotlin`内部自己实现的线程池逻辑
- `CommonPool`： `Java`类库中的`Executor`实现的线程池逻辑

```
internal actual fun createDefaultDispatcher(): CoroutineDispatcher =
    if (useCoroutinesScheduler) DefaultScheduler else CommonPool
internal object DefaultScheduler : ExperimentalCoroutineDispatcher() {
    .....
}

open class ExperimentalCoroutineDispatcher(
    private val corePoolSize: Int,
    private val maxPoolSize: Int,
    private val idleWorkerKeepAliveNs: Long,
```

```kotlin
    private val schedulerName: String = "CoroutineScheduler"
) : ExecutorCoroutineDispatcher() {
    constructor(
        corePoolSize: Int = CORE_POOL_SIZE,
        maxPoolSize: Int = MAX_POOL_SIZE,
        schedulerName: String = DEFAULT_SCHEDULER_NAME
    ) : this(corePoolSize, maxPoolSize, IDLE_WORKER_KEEP_ALIVE_NS, schedulerName)

    ......
}
//java类库中的Executor实现线程池逻辑
internal object CommonPool : ExecutorCoroutineDispatcher() {}
```

如果想使用java类库中的线程池该如何使用呢？也就是修改`useCoroutinesScheduler`属性为`false`

```kotlin
internal const val COROUTINES_SCHEDULER_PROPERTY_NAME = "kotlinx.coroutines.scheduler"

internal val useCoroutinesScheduler =
systemProp(COROUTINES_SCHEDULER_PROPERTY_NAME).let { value ->
    when (value) {
        null, "", "on" -> true
        "off" -> false
        else -> error("System property '$COROUTINES_SCHEDULER_PROPERTY_NAME' has
unrecognized value '$value'")
    }
}

internal actual fun systemProp(
    propertyName: String
): String? =
    try {
       //获取系统属性
        System.getProperty(propertyName)
    } catch (e: SecurityException) {
        null
    }
```

从源码中可以看到,使用过获取系统属性拿到的值， 那我们就可以通过修改系统属性 去改变`useCoroutinesScheduler`的值,
具体修改方法为

```kotlin
 val properties = Properties()
 properties["kotlinx.coroutines.scheduler"] = "off"
 System.setProperties(properties)
```

`DefaultScheduler`的主要实现都在其父类 `ExperimentalCoroutineDispatcher` 中

```kotlin
open class ExperimentalCoroutineDispatcher(
    private val corePoolSize: Int,
    private val maxPoolSize: Int,
    private val idleWorkerKeepAliveNs: Long,
    private val schedulerName: String = "CoroutineScheduler"
) : ExecutorCoroutineDispatcher() {
    public constructor(
        corePoolSize: Int = CORE_POOL_SIZE,
        maxPoolSize: Int = MAX_POOL_SIZE,
        schedulerName: String = DEFAULT_SCHEDULER_NAME
    ) : this(corePoolSize, maxPoolSize, IDLE_WORKER_KEEP_ALIVE_NS, schedulerName)

    //省略......

    //创建CoroutineScheduler实例
    private fun createScheduler() = CoroutineScheduler(corePoolSize, maxPoolSize,
idleWorkerKeepAliveNs, schedulerName)

    override val executor: Executorget() = coroutineScheduler

    //此方法也就是上文说到切换线程的方法
    override fun dispatch(context: CoroutineContext, block: Runnable): Unit =
        try {
            //dispatch方法委托到CoroutineScheduler的dispatch方法
            coroutineScheduler.dispatch(block)
        } catch (e: RejectedExecutionException) {
            ....
        }

    //省略......

    //实现请求阻塞，执行IO密集型任务
    public fun blocking(parallelism: Int = BLOCKING_DEFAULT_PARALLELISM):
CoroutineDispatcher {
        require(parallelism > 0) { "Expected positive parallelism level, but have
$parallelism" }
        return LimitingDispatcher(this, parallelism, null, TASK_PROBABLY_BLOCKING)
    }
    //实现并发数量限制，执行CPU密集型任务
    public fun limited(parallelism: Int): CoroutineDispatcher {
        require(parallelism > 0) { "Expected positive parallelism level, but have
$parallelism" }
        require(parallelism <= corePoolSize) { "Expected parallelism level lesser than
core pool size ($corePoolSize), but have $parallelism" }
        return LimitingDispatcher(this, parallelism, null, TASK_NON_BLOCKING)
    }

   //省略......
}
```

从上文代码可以提炼出
1、在`ExperimentalCoroutineDispatcher`类中创建协程调度线程池`coroutineScheduler`,通过该线程池来管理线程。
2、该类中的`dispatch（）`方法,在协程切换线程中`dispatcher.dispatch(context, this)`调用。
3、其中`blocking（）`方法是执行IO密集型任务，`limited（）`方法执行CPU密集型任务,
实现请求数量限制是调用`LimitingDispatcher`类，其类实现为

```
private class LimitingDispatcher(
    private val dispatcher: ExperimentalCoroutineDispatcher,
    private val parallelism: Int,
    private val name: String?,
    override val taskMode: Int
) : ExecutorCoroutineDispatcher(), TaskContext, Executor {
    //同步阻塞队列
    private val queue = ConcurrentLinkedQueue<Runnable>()
    //cas计数
    private val inFlightTasks = atomic(0)

    override fun dispatch(context: CoroutineContext, block: Runnable) = dispatch(block,
false)

    private fun dispatch(block: Runnable, tailDispatch: Boolean) {
        var taskToSchedule = block
        while (true) {

            if (inFlight <= parallelism) {
                //LimitingDispatcher的dispatch方法委托给了DefaultScheduler的
dispatchWithContext方法
                dispatcher.dispatchWithContext(taskToSchedule, this, tailDispatch)
                return
            }
            ......
        }
    }
}
```

Dispatchers.IO

先看下`Dispatchers.IO`的定义

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


    Internal object DefaultScheduler : ExperimentalCoroutineDispatcher() {
    val IO = blocking(systemProp(IO_PARALLELISM_PROPERTY_NAME,
64.coerceAtLeast(AVAILABLE_PROCESSORS)))
```

`IO`在`DefaultScheduler`中的实现 是调用`blacking()`方法，而`blacking ()`方法最终实现是`LimitingDispatcher`类，
所以 从源码可以看出 `Dispatchers.Default`和`IO` 是在同一个线程中运行的，也就是共用相同的线程池。

而`Default`和`IO` 都是共享`CoroutineScheduler`线程池，`kotlin`内部实现了一套线程池两种调度策略，主要是通过`dispatch`方法中的`Mode`区分的

Type	Mode
Default	NON_BLOCKING
IO	PROBABLY_BLOCKING

```
internal enum class TaskMode {

    //执行CPU密集型任务
    NON_BLOCKING,

    //执行IO密集型任务
    PROBABLY_BLOCKING,
}

//CoroutineScheduler类核心源码
fun dispatch(block: Runnable, taskContext: TaskContext = NonBlockingContext,
tailDispatch: Boolean = false) {

      ......

     if (task.mode == TaskMode.NON_BLOCKING) {
            signalCpuWork() //Dispatchers.Default
     } else {
            signalBlockingWork() // Dispatchers.IO
     }
}
```

从上述代码中可以提炼出的是：
1、`signalCpuWork()`方法处理CPU密集任务，在该方法中根据CPU密集型任务处理策略，创建并管理线程以及执行任务
2、`signalBlockingWork()`方法处理IO密集任务，在该方法中根据IO密集型任务处理策略，创建并管理线程以

及执行任务

其处理策略如下图所示：

Type	处理策略	适合场景	特点
Default	1、CoroutineScheduler最多有corePoolSize个线程被创建; 2、corePoolSize它的取值为max(2, CPU核心数)，即它会尽量的等于CPU核心数	复杂计算、视频解码等	1、CPU密集型任务特点会消耗大量的CPU资源。 2、因为线程本身也有栈等空间，同时线程过多，频繁的线程切换带来的消耗也会影响线程池的性能 3.对于CPU密集型任务，线程池并发线程数等于CPU核心数才能让CPU的执行效率最大化
IO	创建线程数不能大于maxPoolSize，公式：max(corePoolSize, min(CPU核心数 * 128, 2^21 - 2))	网络请求、IO操作等	1、IO密集型 执行任务时CPU会处于闲置状态，任务不会消耗大量的CPU资源。 2、线程执行IO密集型任务时大多数处于阻塞状态，处于阻塞状态的线程是不占用CPU的执行时间。 3、Dispatchers.IO构造时通过LimitingDispatcher默认限制了最大线程并发数parallelism为max(64, CPU核心数)，剩余的任务被放进队列中等待。

Text

Dispatchers.Unconfined

任务执行在默认的启动线程。之后由调用resume的线程决定恢复协程的线程

```
internal object Unconfined : CoroutineDispatcher() {
    //为false为不需要dispatch
    override fun isDispatchNeeded(context: CoroutineContext): Boolean = false

    override fun dispatch(context: CoroutineContext, block: Runnable) {
        // 只有当调用yield方法时，Unconfined的dispatch方法才会被调用
        // yield() 表示当前协程让出自己所在的线程给其他协程运行
        val yieldContext = context[YieldContext]
        if (yieldContext != null) {
            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 code, make sure you properly delegate " +
            "isDispatchNeeded and dispatch calls.")
    }
}
```

每一个协程都有对应的`Continuation`实例，其中的`resumeWith`用于协程的恢复，存在于`DispatchedContinuation`，重点看`resumeWith`的实现以及类委托

```
internal class DispatchedContinuation<in T>(
    @JvmField val dispatcher: CoroutineDispatcher,
    @JvmField val continuation: Continuation<T>//协程suspend挂起方法产生的Continuation
) : DispatchedTask<T>(MODE_UNINITIALIZED), CoroutineStackFrame, Continuation<T> by continuation {
    .....
```

```
    override fun resumeWith(result: Result<T>) {
        val context = continuation.context
        val state = result.toState()
        if (dispatcher.isDispatchNeeded(context)) {
            _state = state
            resumeMode = MODE_ATOMIC
            dispatcher.dispatch(context, this)
        } else {
            executeUnconfined(state, MODE_ATOMIC) {
                withCoroutineContext(this.context, countOrElement) {
                    continuation.resumeWith(result)
                }
            }
        }
    }
    ....
}
```

通过`isDispatchNeeded`（是否需要`dispatch`，`Unconfined`=`false`，`default`，`IO`=`true`）判断做不同处理

- `true`：调用协程的`CoroutineDispatcher`的`dispatch`方法
- `false`：调用`executeUnconfined`方法

```
private inline fun DispatchedContinuation<*>.executeUnconfined(
    contState: Any?, mode: Int, doYield: Boolean = false,
    block: () -> Unit
): Boolean {
    assert { mode != MODE_UNINITIALIZED }
    val eventLoop = ThreadLocalEventLoop.eventLoop
    if (doYield && eventLoop.isUnconfinedQueueEmpty) return false
    return if (eventLoop.isUnconfinedLoopActive) {
        _state = contState
        resumeMode = mode
        eventLoop.dispatchUnconfined(this)
        true
    } else {
        runUnconfinedEventLoop(eventLoop, block = block)
        false
    }
}
```

从`threadlocal`中取出`eventLoop`（`eventLoop`和当前线程相关），判断是否在执行`Unconfined`任务

1. 如果在执行则调用`EventLoop`的`dispatchUnconfined`方法把`Unconfined`任务放进`EventLoop`中
2. 如果没有在执行则直接执行

```kotlin
internal inline fun DispatchedTask<*>.runUnconfinedEventLoop(
    eventLoop: EventLoop,
    block: () -> Unit
) {
    eventLoop.incrementUseCount(unconfined = true)
    try {
        block()
        while (true) {
            if (!eventLoop.processUnconfinedEvent()) break
        }
    } catch (e: Throwable) {
        handleFatalException(e, null)
    } finally {
        eventLoop.decrementUseCount(unconfined = true)
    }
}
```

1. 执行`block()`代码块，即上文提到的`resumeWith()`
2. 调用`processUnconfinedEvent()`方法实现执行剩余的`Unconfined`任务，直到全部执行完毕跳出循环
   `EventLoop`是`CoroutineDispatcher`的一个子类

```kotlin
internal abstract class EventLoop : CoroutineDispatcher() {
    .....
    //双端队列实现存放Unconfined任务
    private var unconfinedQueue: ArrayQueue<DispatchedTask<*>>? = null
    //从队列的头部移出Unconfined任务执行
    public fun processUnconfinedEvent(): Boolean {
        val queue = unconfinedQueue ?: return false
        val task = queue.removeFirstOrNull() ?: return false
        task.run()
        return true
    }
    //把Unconfined任务放进队列的尾部
    public fun dispatchUnconfined(task: DispatchedTask<*>) {
        val queue = unconfinedQueue ?:
            ArrayQueue<DispatchedTask<*>>().also { unconfinedQueue = it }
        queue.addLast(task)
    }
    .....
}
```

内部通过双端队列实现存放`Unconfined`任务

1. `EventLoop`的`dispatchUnconfined`方法用于把`Unconfined`任务放进队列的尾部
2. `processUnconfinedEvent`方法用于从队列的头部移出`Unconfined`任务执行

Dispatchers.Main

`kotlin`在`JVM`上的实现 `Android`就需要引入`kotlinx-coroutines-android`库，它里面有`Android`对应的`Dispatchers.Main`实现，

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

     @JvmField
    val dispatcher: MainCoroutineDispatcher = loadMainDispatcher()

    private fun loadMainDispatcher(): MainCoroutineDispatcher {
        return try {
            val factories = if (FAST_SERVICE_LOADER_ENABLED) {
                FastServiceLoader.loadMainDispatcherFactory()
            } else {
                ServiceLoader.load(
                        MainDispatcherFactory::class.java,
                        MainDispatcherFactory::class.java.classLoader
                ).iterator().asSequence().toList()
            }
            factories.maxBy { it.loadPriority }?.tryCreateDispatcher(factories)
                ?: MissingMainCoroutineDispatcher(null)
        } catch (e: Throwable) {
            // Service loader can throw an exception as well
            MissingMainCoroutineDispatcher(e)
        }
    }

    internal fun loadMainDispatcherFactory(): List<MainDispatcherFactory> {
        val clz = MainDispatcherFactory::class.java
        if (!ANDROID_DETECTED) {
            return load(clz, clz.classLoader)
        }

        return try {
            val result = ArrayList<MainDispatcherFactory>(2)
            createInstanceOf(clz,
"kotlinx.coroutines.android.AndroidDispatcherFactory")?.apply { result.add(this) }
            createInstanceOf(clz,
"kotlinx.coroutines.test.internal.TestMainDispatcherFactory")?.apply { result.add(this)
}
            result
        } catch (e: Throwable) {
            // Fallback to the regular SL in case of any unexpected exception
            load(clz, clz.classLoader)
        }
    }
```

从上文代码中主要功能是通过反射获取`AndroidDispatcherFactory` 然后根据加载的优先级 去创建`Dispatcher`

```
internal class AndroidDispatcherFactory : MainDispatcherFactory {

    override fun createDispatcher(allFactories: List<MainDispatcherFactory>) =
        HandlerContext(Looper.getMainLooper().asHandler(async = true), "Main")

    override fun hintOnError(): String? = "For tests Dispatchers.setMain from kotlinx-
coroutines-test module can be used"

    override val loadPriority: Int
        get() = Int.MAX_VALUE / 2
}
internal class HandlerContext private constructor(
    private val handler: Handler,
    private val name: String?,
    private val invokeImmediately: Boolean
) : HandlerDispatcher(), Delay {

    public constructor(
        handler: Handler,
        name: String? = null
    ) : this(handler, name, false)

   ......

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

    ......
}
```

从上文代码中可以提炼出以下信息：`createDispatcher`调用`HandlerContext`类,通过调用`Looper.getMainLooper()`获取`handler`，最终通过`handler`来实现在主线程中运行.
可以得出`Dispatchers.Main`其实就是把任务通过`Handler`运行在`Android`主线程中的。

总结

1、Dispatchers.Default，切换线程执行CPU密集型任务
2、Dispatchers.IO，切换线程执行IO密集型任务
3、Dispatchers.Unconfined，任务执行在默认的启动线程
4、Dispatchers.Main，切换线程到主线程