{2023}

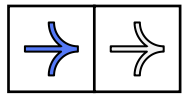
协程实战和原理

分享人：王杰

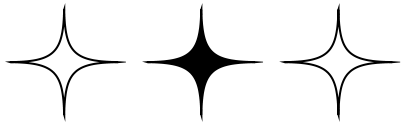
自我介绍

王杰，阅文集团高级安卓开发，目前担任起点读书的研发。社区名九心，长期活跃于技术社区，发表多篇优质文章，Github累计过得2k+star。

目录



01 协程介绍



为什么使用协程?

协程诞生前的局面?

异步操作容易陷入回调地狱

所以协程的特点是用阻塞的方式写出非阻塞的代码

为什么选择协程

减少内存泄漏

安全

Jetpack都支持协程

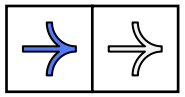
Jetpack

简洁

简化逻辑，用阻塞方式写出非阻塞的代码

取消方便

job取消方便



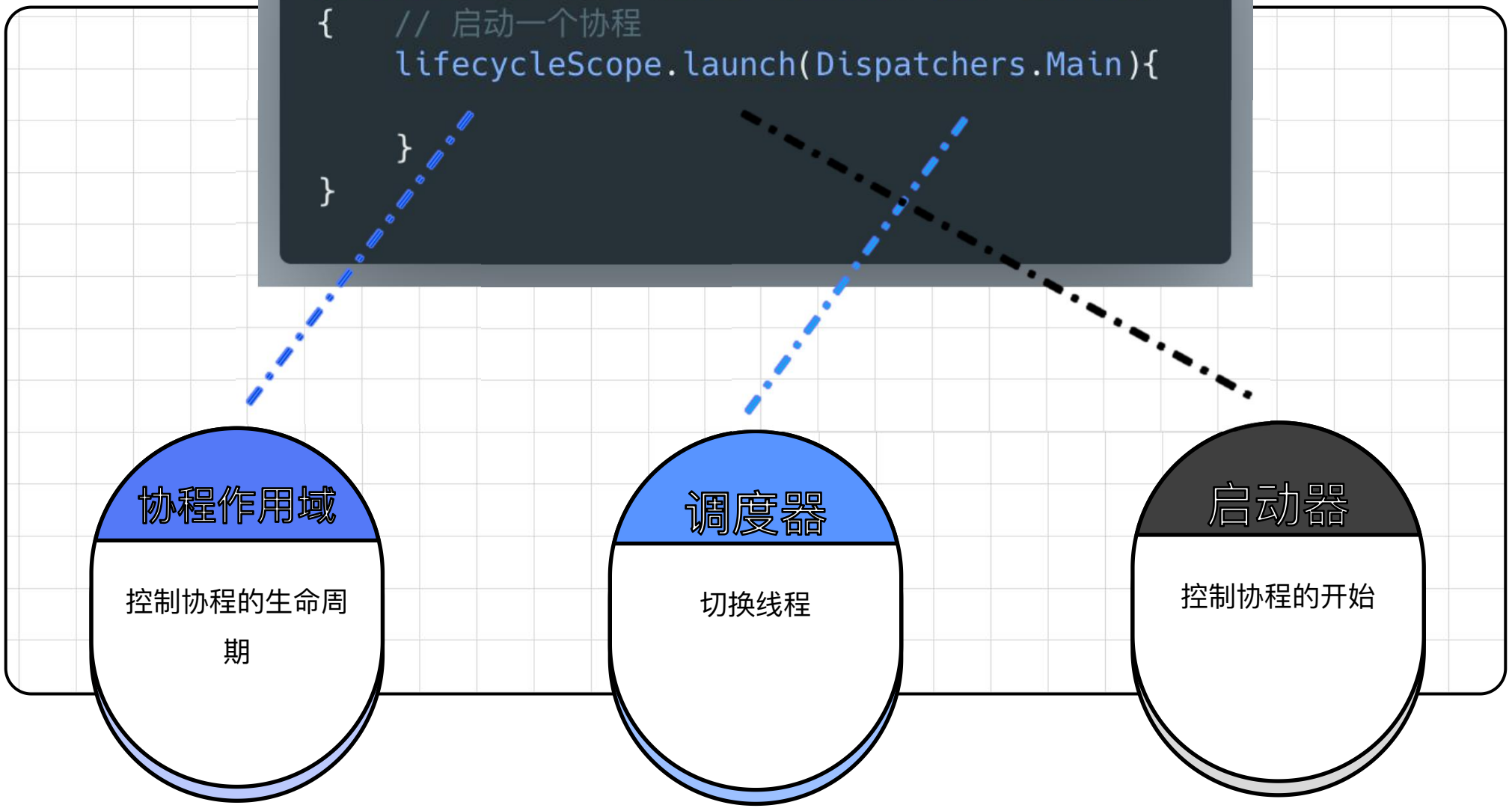
02 快速入门

基本使用

快速入门

```
private fun login(name: String, pwd: String)
{    // 启动一个协程
     lifecycleScope.launch(Dispatchers.Main){

     }
}
```



快速入门

launch方法

```
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
}
```

快速入门

CoroutineScope

协程作用域，控制协程的生命周期

在 Android 中，目前可以使用的作用域有MainScope、LifecycleScope 和ViewModelScope，GlobalScope(不建议使用)，Activity生命周期结束后，没有及时取消。

使用MainScope等作用域启动的协程都是独立的，子协程的取消不会导致当前协程的取消，也不会当前协程下面的其他子协程取消。

快速入门

协程启动模式

1. DEFAULT: 立即执行协程体
2. ATOMIC：立即执行协程体，但在开始运行前无法取消
3. UNDISPATCHED：立即在当前线程执行协程体，直到第一个 suspend 调用
4. LAZY：只有在需要的情况下运行

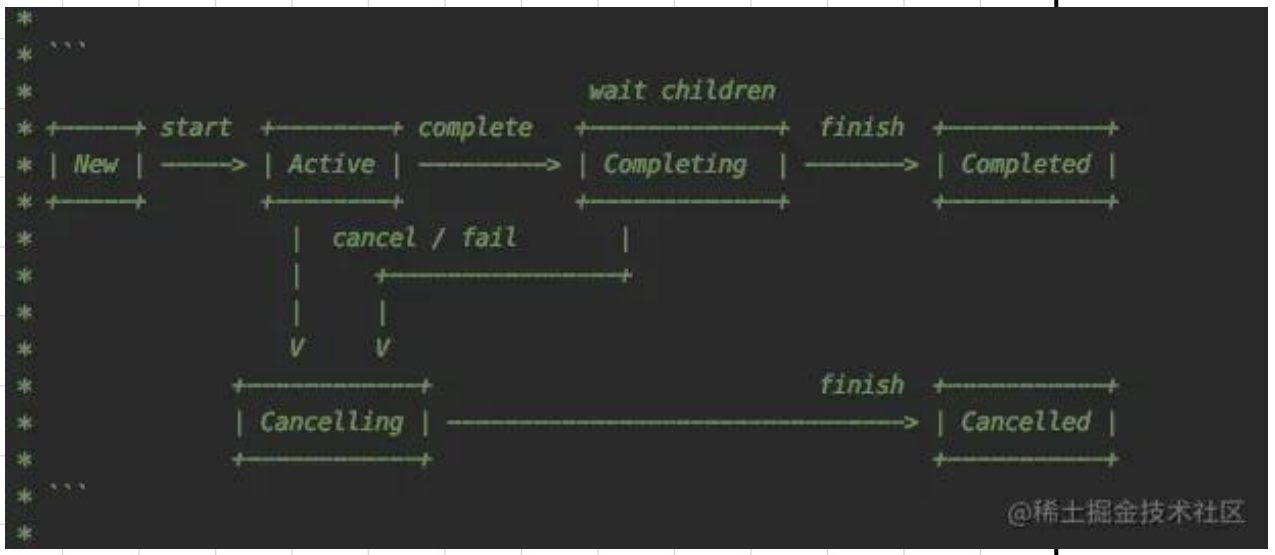
快速入门

Job

```
public interface Job : CoroutineContext.Element {
    ...
    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()
    ...
}
```

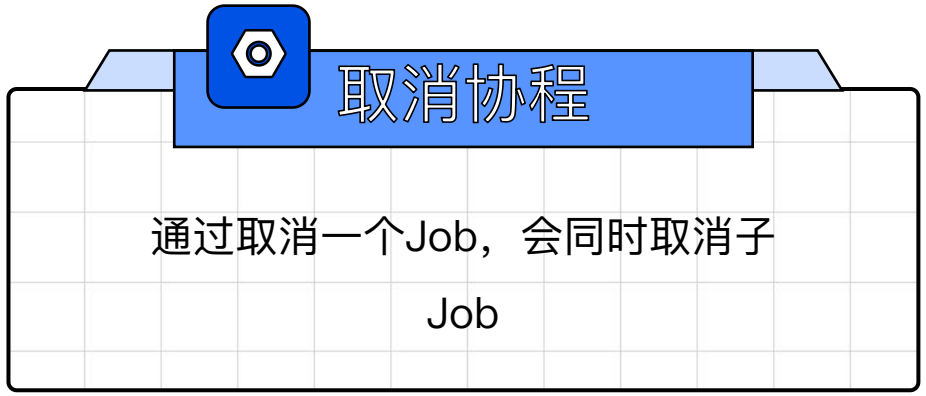
协程是以Job形式存在的

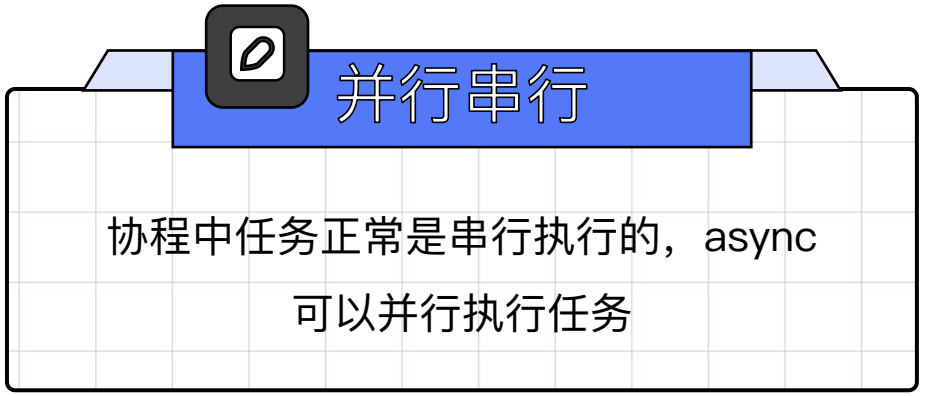


快速入门

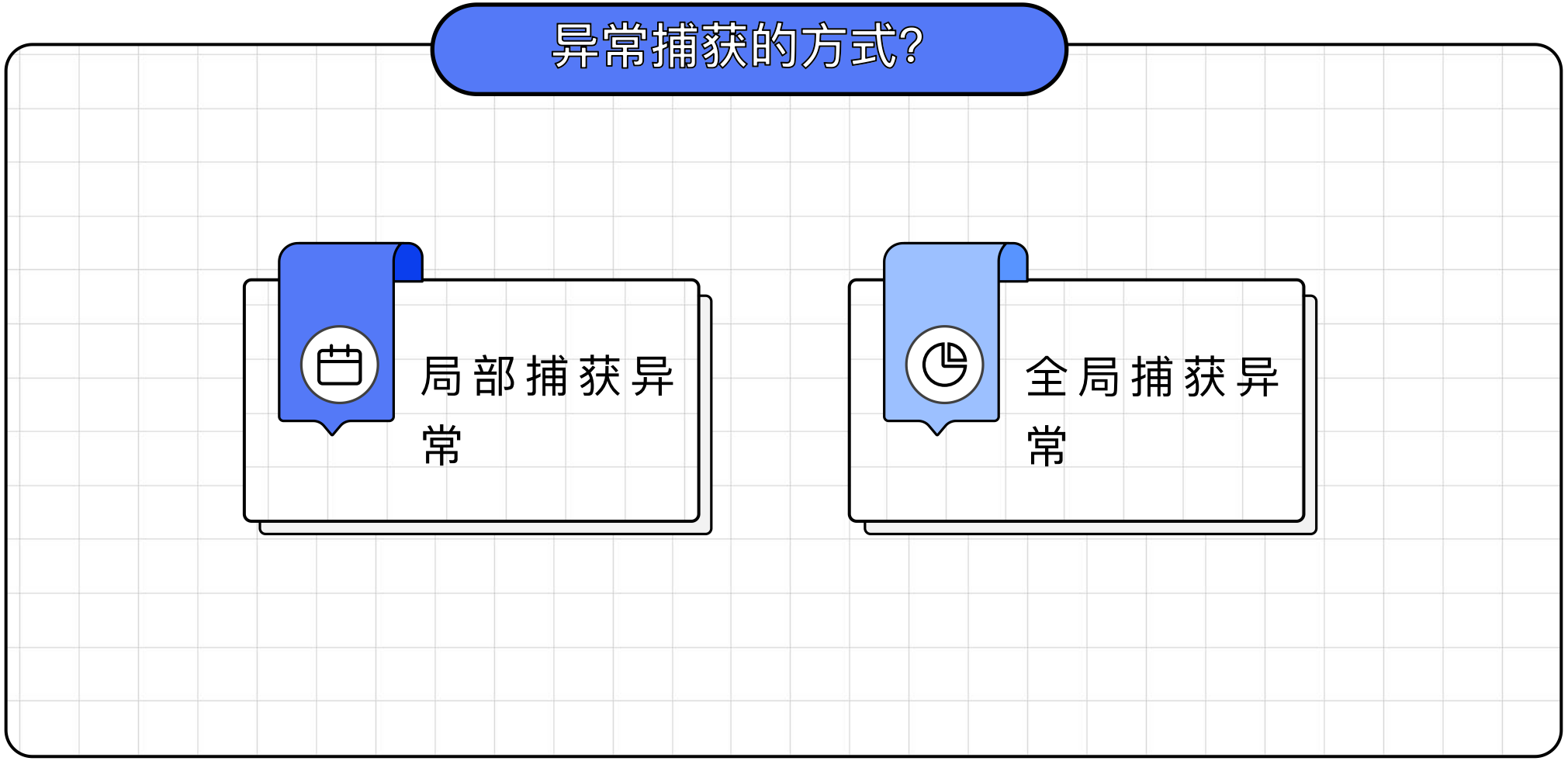

切换线程
使用withContext可以很方便的切换线程


异常捕获
常见的方法是使用try catch去捕获，也有一些特殊方法


取消协程
通过取消一个Job，会同时取消子Job


并行串行
协程中任务正常是串行执行的，async可以并行执行任务

快速入门



03 进阶使用

Flow是什么？

Flow是Kotlin协程与响应式编程模式结合的产物

快速入门

创建Flow

```
private fun createFlow(): Flow<Int> {
    return flow<Int> {
        for(i in 1..10) {
            emit(i)
        }
    }
    /*flowOf(1..10)
    listOf<Int>(1, 2, 3, 4, 5, 6).asFlow()*/
}
```

快速入门

消费Flow

```
btnHello.setOnClickListener {
    lifecycleScope.launch {
        createFlow()
            .collect {

            }
    }
}
```

collect可以消费Flow下发的数据

快速入门

线程切换

```
btnHello.setOnClickListener {
    lifecycleScope.launch(Dispatchers.Main) {
        createFlow()
            .flowOn(Dispatchers.IO)
            .collect {

            }
    }
}
```

- flowOn可以改变Flow产生的线程
- 消费的线程由协程启动处指定

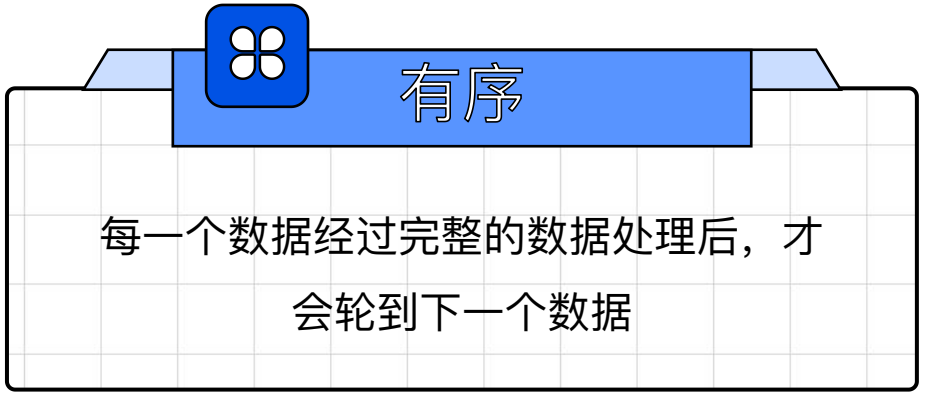
快速入门

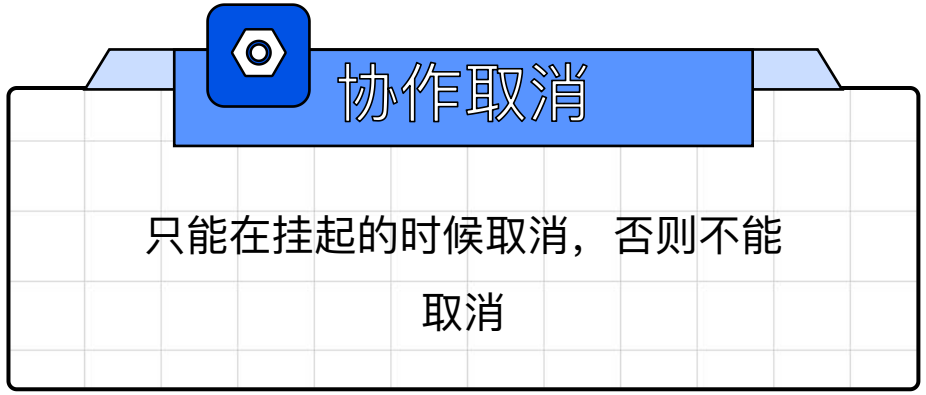
常用的操作符

- 普通操作符：map\take\filter
- 特殊操作符：buffer\conflate\collectLatest
- 组合操作符：zip\combine
- 展平流操作符：flatMapConcat\flatMapMerge\flatMapLatest
- 末端操作符：collect\toList\first\reduce...

Flow的特点

Designed by iSHEJI.com

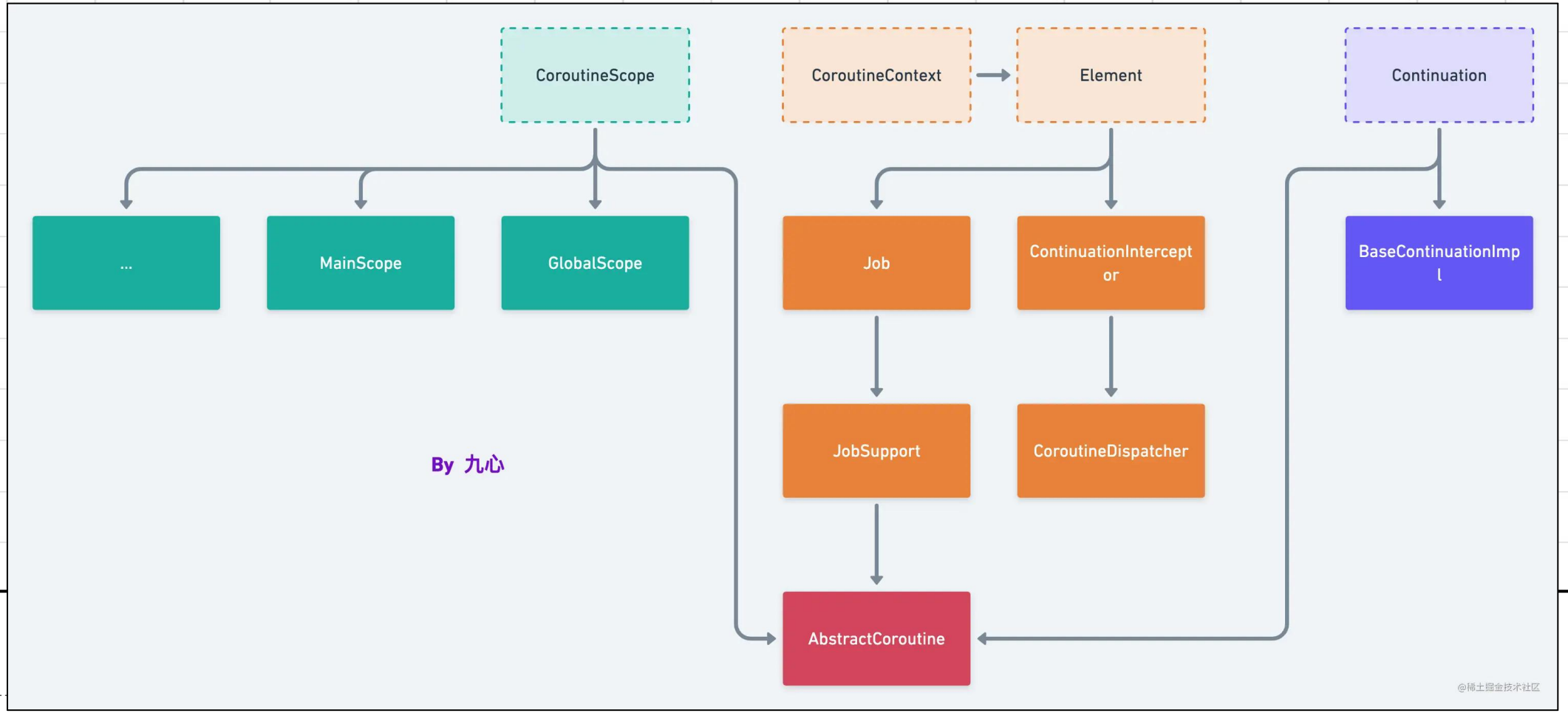




04 基本原理

基本原理

协程的类结构?


CoroutineScope
CoroutineContext
Element
Continuation
...
MainScope
GlobalScope
Job
ContinuationIntercept
or
BaseContinuationImp
l
By 九心
JobSupport
CoroutineDispatcher
AbstractCoroutine
@稀土掘金技术社区

快速入门

Continuation

```kotlin
@SinceKotlin("1.3")
public interface Continuation<in T> {
    public val context: CoroutineContext

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

Continuation: 续体

每次遇到挂起的时候，协程会将剩余的代码包裹成Continuation，等到恢复的时候调用resumeWith方法，执行剩下的内容

快速入门

CoroutineContext

```
public interface CoroutineContext {
    // get 方法，通过 key 获取
    public operator fun <E : Element> get(key: Key<E>): E?
    // 累加操作
    public fun <R> fold(initial: R, operation: (R, Element) -> R): R
    // 操作符 + ，实际的实现调用了 fold 方法
    public operator fun plus(context: CoroutineContext): CoroutineContext
    // 移除操作
    public fun minusKey(key: Key<*>): CoroutineContext

    // CoroutineContext 定义的 Key
    public interface Key<E : Element>

    // CoroutineContext 中元素的定义
    public interface Element : CoroutineContext {
        // key
        public val key: Key<*>
        //...
    }
}
```

从左边的代码中，我们可以看出CoroutineContext被设计成了集合，在这个集合中，有两类比较重要的元素：

- Job：协程的任务呈现方式
- 拦截器：可以完成线程的切换

快速入门

CoroutineScope

```
public interface CoroutineScope {
    public val coroutineContext: CoroutineContext
}
```

协程作用域持有CoroutineContext，其实就相当于持有协程的Job，从而达到管理协程的生命周期

基本原理

协程的基本模型？

```kotlin
internal abstract class BaseContinuationImpl(
    // 完成后调用的 Continuation
    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!! // fail fast when trying to resume continuation without completion
                val outcome: Result<Any?> =
                    try {
                        // 1. 执行 suspend 中的代码块
                        val outcome = invokeSuspend(param)
                        // 2. 如果代码挂起就提前返回
                        if (outcome === COROUTINE_SUSPENDED) return
                        // 3. 返回结果
                        Result.success(outcome)
                    } catch (exception: Throwable) {
                        // 3. 返回失败结果
                        Result.failure(exception)
                    }
                releaseIntercepted() // this state machine instance is terminating
                if (completion is BaseContinuationImpl) {
                    // 4. 如果 completion 中还有子 completion, 递归
                    current = completion
                    param = outcome
                } else {
                    // 5. 结果通知
                    completion.resumeWith(outcome)
                    return
                }
            }
        }
    }
}
```

遇到挂起点就会返回COROUTINE_SUSPENDED，意味着需要等待结果，否则就正常返回结果

简单来说，遇到挂起的方法就等待，无挂起就正常返回结果，如果结果仍然是一个续体，继续执行结果的恢复方法。

基本原理

协程的状态机?

```
private fun showSuspendCode() {
    lifecycleScope.launch {
        Log.d("MainActivity", "1")
        delay(1000)
        Log.d("MainActivity", "2")
    }
}
```

这一段代码如何是拆开成几个续体执行的?

基本原理

协程的状态机?

使用一个Label记录状态，根据不同Label执行不同的代码块

```
public class ContinuationImpl implements Continuation<Object> {
    private int label = 0;
    private final Continuation<Unit> completion;

    public ContinuationImpl(Continuation<Unit> completion) {
        this.completion = completion;
    }

    public CoroutineContext getContext() {
        return EmptyCoroutineContext.INSTANCE;
    }

    public void resumeWith(Object o) {
        try {
            Object result = o;
            switch (label) {
                case 0: {
                    Log.d("MainActivity", "1");
                    this.label = 1;
                    result = DelayKt.delay(1000, this);
                    if (isSuspended(result)) return;
                }
                case 1: {
                    Log.d("MainActivity", "2");
                   break;
                }
            }
            completion.resumeWith(Unit.INSTANCE);
        } catch (Exception e) {
            completion.resumeWith(e);
        }
    }

    private boolean isSuspended(Object result) {
        return result == IntrinsicsKt.getCOROUTINE_SUSPENDED();
    }
}
```

基本原理

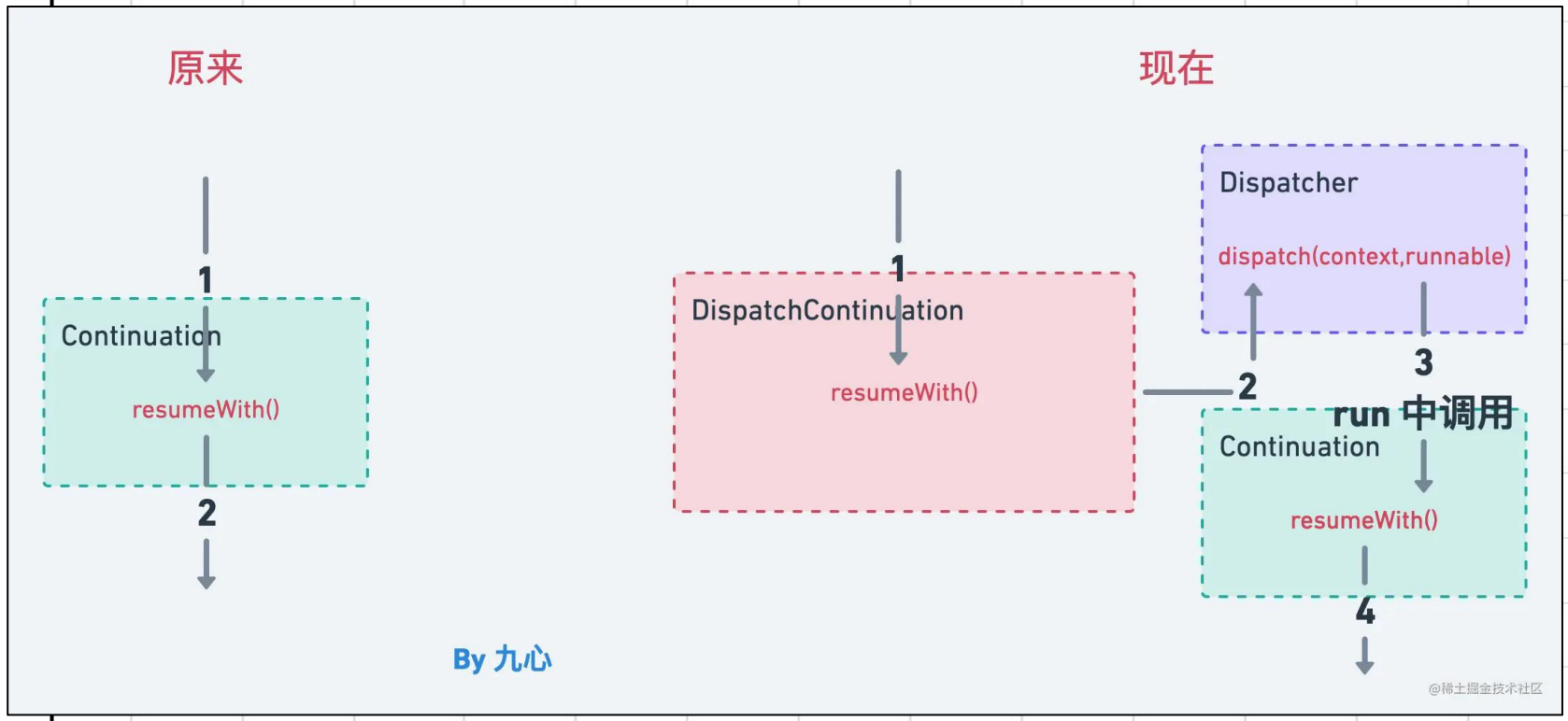
协程切换线程的秘密?

```
internal fun <R, T> (suspend (R) -> T).startCoroutineCancellable(receiver: R, completion: Continuation<T>) =
    runSafely(completion) {
        // 外面再包一层 Coroutine
        createCoroutineUnintercepted(receiver, completion)
            // 如果需要，做拦截处理
            .intercepted()
            // 调用 resumeWith 方法
            .resumeCancellableWith(Result.success(Unit))
    }
```

intercepted()方法会找到拦截器，通过CoroutineContext这个集合去获取

基本原理

协程切换线程的秘密？



1. HandlerContext
2. ExperimentalCoroutine Dispatcher

谢谢大家