








在讲协程的如何切换线程之前，有必要先了解下协程的上下文是什么？它的结构是什么样的？以及我们如何使用它？今天带着该问题来认识它。







CoroutineContext

协程上下文都是继承自CoroutineContext，它是一个接口，内部方法以及内部类如下：

```
▼  CoroutineContext
   get(CoroutineContext.Key<E>): E?
   fold(R, (R, CoroutineContext.Element) -> R): R
   plus(CoroutineContext): CoroutineContext
   minusKey(CoroutineContext.Key<*>): CoroutineContext
  >  Key
  >  Element
```

掘金技术社区 @ xiangcman

它的实现子类有如下：

```
▼ *  CoroutineContext (kotlin.coroutines)
   EmptyCoroutineContext (kotlin.coroutines)
   CombinedContext (kotlin.coroutines)
  ▼  Element (kotlin.coroutines.CoroutineContext)
     ContinuationInterceptor (kotlin.coroutines)
    >  AbstractCoroutineContextElement (kotlin.coroutines)
```

掘金技术社区 @ xiangcman

比如我们常见的EmptyCoroutineContext，它的内部实现如下：

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

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

掘金技术社区 @ xiangcman

可以看到它的get、fold、plus、minusKey几个方法都是默认实现，你可以理解它就是个空壳子的context。

Element

在讲CoroutineContext内部结构之前，先来认识下Element，它也实现了CoroutineContext接口：

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

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

```
public interface Element : CoroutineContext {
```

A key of this coroutine context element.

```
public val key: Key<*>
```

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

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

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

```
}
```

掘金技术社区 @ xiangcman

Element中有一个key的属性，这里可以理解key就是当前Element的唯一标识。实现一个context的时候需要指明它的key是啥，此处就是用该key来标识

get：如果传进来的key和自己的key相等，则返回自己，否则返回null

fold：将初始值和当前element返回给lambda，让lambda自己去处理

minusKey：如果传进来的key和自己相同，则返回EmptyCoroutineContext，否则返回自己，其实是删除对应key的context.

写了3个context，然后用"+"拼接：

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val context=One()+Two()+Three()
    val contextName = context.javaClass.name
    val oneName = context.get(One)?.javaClass?.name
    Log.d(TAG, msg: "contextName:$contextName")
    Log.d(TAG, msg: "oneName:$oneName")
}

```

```

class One :AbstractCoroutineContextElement(Key){
    companion object Key : CoroutineContext.Key<One>
}
class Two :AbstractCoroutineContextElement(Key){
    companion object Key : CoroutineContext.Key<Two>
}

```

```

class Three :AbstractCoroutineContextElement(Key){
    companion object Key : CoroutineContext.Key<Three>
}

```

掘金技术社区 @ xiangcman

自定义context的时候，需要继承自AbstractCoroutineContextElement，它是继承自Element，因为它强制要求需要一个key作为context的标识，一般key的element标识是当前context，看上面的One这个context，它的key拥有的element是One。

输出日志如下：

```

D contextName:kotlin.coroutines.CombinedContext
D oneName:com.example.coroutinesdemo.CoroutineContextActivity$One

```

掘金技术社区 @ xiangcman

One()+Two()+Three()得到的是一个CombinedContext，get方法通过One这个key取到了One这个取对应的Context

```

val context1=One()+EmptyCoroutineContext
val context1Name = context1.javaClass.name
Log.d(TAG, msg: "context1Name:$context1Name")

```

日志如下：

```

D context1Name:com.example.coroutinesscopedemo.CoroutineContextActivity$One

```

可以看到我给One的context拼接了一个EmptyCoroutineContext时候，得到的是它自己，"+"是重载了context的plus方法，看下plus方法的实现：

```

public operator fun plus(context: CoroutineContext): CoroutineContext =
    //①
    if (context === EmptyCoroutineContext) this else // fast path -- avoid lambda
creation
    //②
    context.fold(this) { acc, element ->
        //③
        val removed = acc.minusKey(element.key)
        //③.1
        if (removed === EmptyCoroutineContext) element else {
            // make sure interceptor is always last in the context (and thus is
fast to get when present)
            //④
            val interceptor = removed[ContinuationInterceptor]
            //⑤
            if (interceptor == null) CombinedContext(removed, element) else {
                //⑥
                val left = removed.minusKey(ContinuationInterceptor)
                //⑦
                if (left === EmptyCoroutineContext) CombinedContext(element,
interceptor) else
                    //⑧
                    CombinedContext(CombinedContext(left, element), interceptor)
            }
        }
    }
}

```

- 1.如果传进来的context是EmptyCoroutineContext，则返回自己，所以上面的One()+EmptyCoroutineContext，得到的是One这个context
- 2.context.fold，会把初始值和context传给闭包，所以acc是当前context，element是传进来的context
- 3.acc.minuskey(element.key)，如果传进来的context的key和当前context的key相等，则返回传进来的context，所以新的context会把旧的context给覆盖掉了

- 4.如果传进来的context的key和当前context的key不相等, removed则是当前context, 查看当前context中是否有ContinuationInterceptor类型的context, 我们的dispatcher都是属于该类型, 需要单独处理
- 5.如果context中没有ContinuationInterceptor类型的context, 则初始化出一个CombinedContext的context, 所以上面的One()+Two()+Three()是一个CombinedContext的context
- 6.如果当前context中存在ContinuationInterceptor类型的context, 则继续判断当前context是不是ContinuationInterceptor类型的context
- 7.如果是ContinuationInterceptor类型的context, 则把传进来的context和当前的context组合成CombinedContext的context
- 8.如果当前的context不是一个ContinuationInterceptor类型的context, 则把当前当前的context和传进来的context新组合成一个CombinedContext的context, 再和前面的ContinuationInterceptor组合成一个新的CombinedContext的context

CombinedContext

```
internal class CombinedContext(  
    private val left: CoroutineContext,  
    private val element: Element  
) : CoroutineContext, Serializable {  
  
    override fun <E : Element> get(key: Key<E>): E? {  
        var cur = this  
        while (true) {  
            cur.element[key]?.let { return it }  
            val next = cur.left  
            if (next is CombinedContext) {  
                cur = next  
            } else {  
                return next[key]  
            }  
        }  
    }  
  
    public override fun <R> fold(initial: R, operation: (R, Element) -> R): R =  
        operation(left.fold(initial, operation), element)  
  
    public override fun minusKey(key: Key<*>): CoroutineContext {  
        //①  
        element[key]?.let { return left }  
        //②  
        val newLeft = left.minusKey(key)  
        return when {  
            //③  
            newLeft === left -> this  
            //④  
            newLeft === EmptyCoroutineContext -> element  
            //⑤  
            else -> CombinedContext(newLeft, element)  
        }  
    }  
}
```



```

override fun toString(): String =
    "[" + fold("") { acc, element ->
        if (acc.isEmpty()) element.toString() else "$acc, $element"
    } + "]"
}

```

它是直接继承自CoroutineContext，有两个比较重要的属性：

left：CoroutineContext，它是左边的节点

element：Element，当前节点

其实和链表的结构有点类似，left相当于next节点。

get：递归节点，直到left节点不是CombinedContext类型的

fold：先把left和初始值组成一个初始值，然后再把这个初始值和当前节点传给闭包

minusKey：

1.如果当前节点中找到了该key，则返回left节点

2.如果找不到，则继续在left节点中找

3.如果找不到返回this

4.如果找到了则返回当前节点

5.否则继续往左边再找

整个分析来看，协程中的context如果是多个context拼接的时候如果传进来的是EmptyCoroutineContext，则只保存自己。如果传进来的context的key和当前context的key一样，则会覆盖掉原来的context。如果都不满足，则采用链表的形式插入到原来的context头节点上，如果传进来的是ContinuationInterceptor类型的，则会把该类型放到头节点。

总结

CoroutineContext是协程重要的对象，它通过重载了plus方法，轻松的将每一个CoroutineContext拼接成一个新的CoroutineContext，一般的CoroutineContext会分为EmptyCoroutineContext，它是直接实现了CoroutineContext接口，可以理解它是没有任何信息的CoroutineContext。而Element是带有key的CoroutineContext，所以如果往CoroutineContext添加一个Element会通过它的key来找到对应的Element，Element下面常见的子类有ContinuationInterceptor，它是我们协程切换线程的CoroutineContext，还有常见的CoroutineExceptionHandler，它是我们协程捕捉异常的CoroutineContext，还有常见的Job，它是我们协程构建结构化的CoroutineContext，还有不常用的CoroutineName，它是用来给协程起名字的CoroutineContext。在通过+(plus)拼接CoroutineContext的时候，如果发现传进来的是EmptyCoroutineContext，则还是返回自己，如果不是EmptyCoroutineContext则会拼接成一个CombinedContext，它是一个链表结构的CoroutineContext，如果在+过程中发现要添加的CoroutineContext已经存在于原有的CoroutineContext中，则会用新的覆盖掉原有的CoroutineContext。但是在CombinedContext中会将ContinuationInterceptor类型的CoroutineContext放到链表的最前面。

再来一张本次讲解的context类图：

```

@startuml
skin rose
interface CoroutineContext{

```

```

+operator fun <E : Element> get(key: Key<E>): E?
+fun <R> fold(initial: R, operation: (R, Element) -> R): R
+operator fun plus(context: CoroutineContext): CoroutineContext
+fun minusKey(key: Key<*>): CoroutineContext
}
interface Key<E : Element>
interface Element implements CoroutineContext{
    +val key: Key<*>
    +operator fun <E : Element> get(key: Key<E>): E?
    +fun <R> fold(initial: R, operation: (R, Element) -> R): R
    +fun minusKey(key: Key<*>): CoroutineContext
}
class AbstractCoroutineContextElement implements Element
class One extends AbstractCoroutineContextElement
class Two extends AbstractCoroutineContextElement
class Three extends AbstractCoroutineContextElement
class CombinedContext implements CoroutineContext{
    -val left: CoroutineContext
    -val element: Element
    +operator fun <E : Element> get(key: Key<E>): E?
    +fun <R> fold(initial: R, operation: (R, Element) -> R): R
    +fun minusKey(key: Key<*>): CoroutineContext
}
class EmptyCoroutineContext implements CoroutineContext
Element-up-*Key
@enduml

```