# Android—kotlin-Channel超详细讲解

hqk LV.4

2021年12月15日 16:00 · 阅读 3038

关注

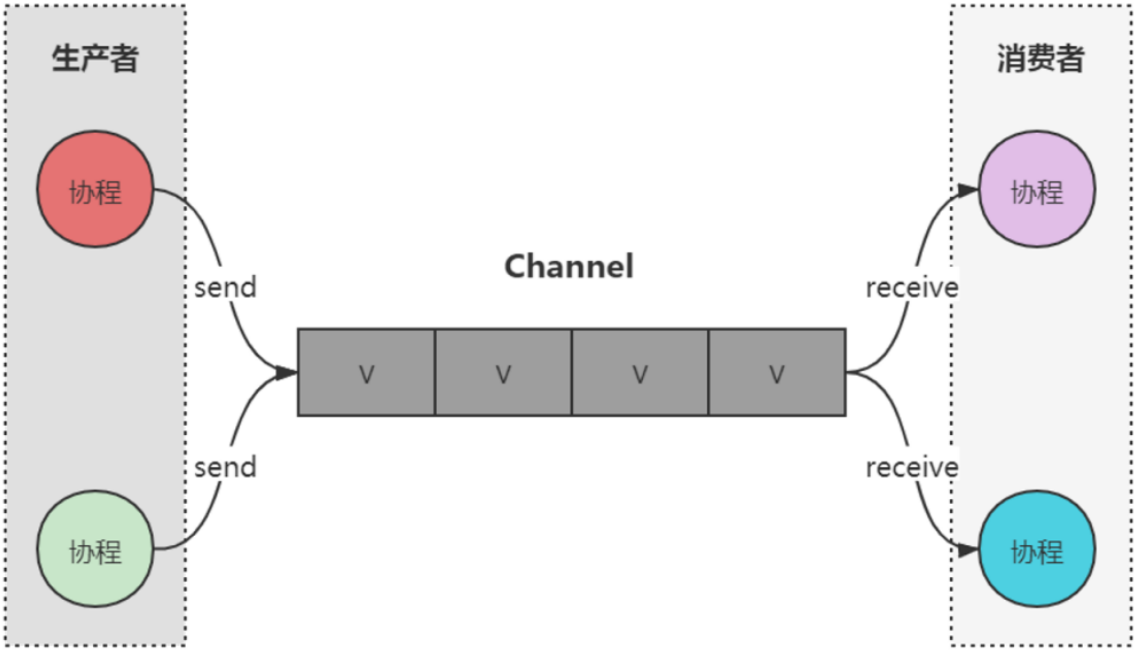

## 前言

在上一篇，主要讲解了关于Flow异步冷流相关的知识点。在本篇中将会讲解Channel通道（热流）相关的知识点！

那么Channel是什么呢?

## 1、Channel通道

### 1.1 认识Channel




**如图所示**

Channel实际上是一个**并发安全的队列**，它可以用来连接协程，实现不同协程之间的通信。

既然如此，来个小demo试试手：

kotlin 复制代码

```kotlin
@Test
fun `test know channel`() = runBlocking<Unit> {
    val channel = Channel<Int>()
    //生产者
    val producer = GlobalScope.launch {
        var i = 0
        while (true) {
            delay(1000)
            channel.send(++i)
            println("send $i")
        }
    }

    //消费者
    val consumer = GlobalScope.launch {
        while (true) {
            val element = channel.receive()
            println("receive $element")
        }
    }
    joinAll(producer, consumer)
}
```

这里很简单，就两个协程，分别代表：生产者和消费者

### 来看看运行效果

复制代码

```
receive 1
send 1
send 2
receive 2
....略
send 999
receive 999
```

这个就很简单，就直接进入下一专题了！

## 1.2 Channel的容量

Channel实际上就是一个队列，队列中一定存在缓存区，那么一旦这个缓冲区满了，并且也一直没有人调用receive并取走函数，send就需要挂起。故意让接收端的节奏放慢，发现send总是会挂起，直到receive之后才会继续往下执行。

概念一大堆，来个Demo试试手：

kotlin 复制代码

```kotlin
@Test
fun `test know channel2`() = runBlocking<Unit> {
    val channel = Channel<Int>()
    //生产者
    val producer = GlobalScope.launch {
        var i = 0
        while (true) {
            delay(1000)
            channel.send(++i)
            println("send $i")
        }
    }

    //消费者
    val consumer = GlobalScope.launch {
        while (true) {
            delay(2000)
            val element = channel.receive()
            println("receive $element")
        }
```

```
        }
        joinAll(producer, consumer)

    }
```

这里我们看到：消费者用时比生产者用时高，那么

### 来看看运行效果

复制代码

```
receive 1
send 1
receive 2 //等了2秒打印
send 2
receive 3 //这里又等了2秒打印
send 3
```

通过这个运行效果也验证了：一旦这个缓冲区满了，并且也一直没有人调用receive并取走函数，send就需要挂起。

通俗点就是：当消费者处理元素用时大于生产者生产元素用时，并且缓存区也满了时，生产者就会偷会懒，等待消费者处理缓冲区的数据。

这样理解，相信很容易吧，接着下一个专题

## 1.3 迭代Channel

Channel本身确实像序列，所以我们在读取的时候可以直接获取一个Channel的iterator。

kotlin 复制代码

```
    @Test
    fun `test iterate channel`() = runBlocking<Unit> {
        val channel = Channel<Int>(Channel.UNLIMITED)
        //生产者
        val producer = GlobalScope.launch {
            for (x in 1..5) {
                channel.send(x * x)
                println("send ${x * x}")
            }
        }

        //消费者
        val consumer = GlobalScope.launch {
            /*val iterator = channel.iterator()
            while (iterator.hasNext()){
```

```
                val element = iterator.next()
                println("receive $element")
                delay(2000)
            }*/

                        //上下两种写法都可以
            for (element in channel) {
                println("receive $element")
                delay(2000)
            }
        }
        joinAll(producer, consumer)

    }
```

一切尽在注释中。

### 先来看运行效果

复制代码

```
send 1
send 4
send 9
send 16
send 25  //前5条消息几乎瞬间出来
receive 1 //往后的每条消息间隔2秒
receive 4
receive 9
receive 16
receive 25
```

我们可以看到，这运行效果和1.2的完全不一样！这里的生产者根本就没有等待对应的消费者处理完成就提前完成了所有工作！

上面我们提到过：生产者"偷懒"的条件：**一是消费者处理时间大于生产者；二是缓存区必须满了！**

但这里在定义Channel通道时，使用了： `val channel = Channel<Int>(Channel.UNLIMITED)` ，**将缓存区改成了无限大**，因此生产者才不管消费者能不能处理过来，一梭哈全生成完了！

## 1.4 produce与actor

- 构造生产者与消费者的便捷方法

- 我们可以**通过produce方法启动一个生产者协程**，并返回一个ReceiveChannel，其他协程就可以用这个Channel来接收数据了。反过来，**我们可以用actor启动一个消费协程！**

概念说完了，该开始上手了

### 1.4.1 使用produce

kotlin 复制代码

```kotlin
@Test
fun `test fast producer channel`() = runBlocking<Unit> {
    //生产者,
    val receiveChannel: ReceiveChannel<Int> = GlobalScope.produce<Int> {
        repeat(100) {
            delay(1000)
            send(it)
        }
    }
            //消费者
    val consumer = GlobalScope.launch {
        for (i in receiveChannel) {
            println("received: $i")
        }
    }
    consumer.join()
}
```

### 来看看运行效果

复制代码

```
received: 0  //每隔一秒打印
received: 1
received: 2
received: 3
...略
```

这里我们可以看到通过 `GlobalScope.produce` 返回了 `ReceiveChannel` 生产者协程，在消费者里就可以通过 `ReceiveChannel` 来接收对应生产者产生的数据。接下来看下一个！

### 1.4.2 使用actor

kotlin 复制代码

```kotlin
@Test
fun `test fast consumer channel`() = runBlocking<Unit> {
```

```
    val sendChannel: SendChannel<Int> = GlobalScope.actor<Int> {
        while (true) {
            val element = receive()
            println(element)
        }
    }

    val producer = GlobalScope.launch {
        for (i in 0..3) {
            sendChannel.send(i)
        }
    }

    producer.join()
}
```

### 来看看运行效果

复制代码

```
0
1
2
3
```

这里我们看到通过 `GlobalScope.actor` 产生了对应的消费者 `sendChannel` ，在对应的生产者里面通过 `sendChannel.send(i)` 向对应的消费者发送数据！

接着看下一个！

## 1.5 Channel的关闭

- produce和actor返回的Channel都会随着对应的协程执行完毕而关闭，也正是这样，**Channel才被称为热数据流**；
- 对于一个Channel，如果我们调用了它的close方法，它会立即停止接收新元素，也就是说这时它的 `isClosedForSend` 会立即返回true；
    - 而由于Channel缓冲区的存在，这时候可能还有一些元素没有被处理完，因此要等所有的元素都被读取之后 `isClosedForSend` 才会返回true；
- Channel的生命周期最好由主导方来维护，建议**由主导的一方实现关闭**。

- 因为可能会存在一个生产者对应多个消费者，就好比如，一个老师讲课，有多个学生听课，是否上下课的信号由老师来负责，而不是学生！

老规矩，概念完了，就开始Demo上手：

kotlin 复制代码

```kotlin
@Test
fun `test close channel`() = runBlocking<Unit> {
    val channel = Channel<Int>(3)
    //生产者
    val producer = GlobalScope.launch {
        List(3) {
            channel.send(it)
            println("send $it")
        }
                //由生产者主导生命周期，执行关闭！
        channel.close()
        println("""close channel.
            |  - ClosedForSend: ${channel.isClosedForSend}
            |  - ClosedForReceive: ${channel.isClosedForReceive}""".trimMargin())
    }

    //消费者
    val consumer = GlobalScope.launch {
        for (element in channel){
            println("receive $element")
            delay(1000)
        }
        println("""After Consuming.
            |   - ClosedForSend: ${channel.isClosedForSend}
            |   - ClosedForReceive: ${channel.isClosedForReceive}""".trimMargin())
    }

    joinAll(producer, consumer)
}
```

这里我们看到，就仅仅是在生产者里面主导了生命周期，其他的都是状态打印！

## 来看看运行效果

复制代码

```
send 0
receive 0
send 1
send 2
close channel.
 - ClosedForSend: true
 - ClosedForReceive: false
```

```
receive 1
receive 2
After Consuming.
 - ClosedForSend: true
 - ClosedForReceive: true
```

从这个运行效果可以看出：

- 当生产者执行完毕时：对应 `ClosedForSend` 为true；
- 当消费者执行完毕时：对应 `ClosedForReceive` 为true。

### 1.6 BroadcastChannel

上面提到，生产者和消费者在Channel中存在一对多的情形，从数据处理本身来讲，虽然有多个接收端，但是同一个元素只会被一个接收端读到。广播则不然，**多个接收端不存在互斥行为。**

来看看这个广播如何使用：

kotlin 复制代码

```kotlin
@Test
fun `test broadcast`() = runBlocking<Unit> {
    //val broadcastChannel = BroadcastChannel<Int>(Channel.BUFFERED)
    val channel = Channel<Int>() //这里使用默认缓存区大小
    //初始化三个消费者
    val broadcastChannel = channel.broadcast(3)
    val producer = GlobalScope.launch {
        List(3){
            delay(100)
            broadcastChannel.send(it)
        }
        //由主导方管理生命周期
        broadcastChannel.close()
    }

        //创建三个消费者
    List(3){ index ->
        GlobalScope.launch {
            val receiveChannel = broadcastChannel.openSubscription()
            for (i in receiveChannel){
                println("[#$index] received: $i")
            }
        }
    }.joinAll()
}
```

一切尽在注释中，

### 来看看运行效果

复制代码

```
[#0] received: 0
[#1] received: 0
[#2] received: 0
[#0] received: 1
[#1] received: 1
[#2] received: 1
[#0] received: 2
[#2] received: 2
[#1] received: 2
```

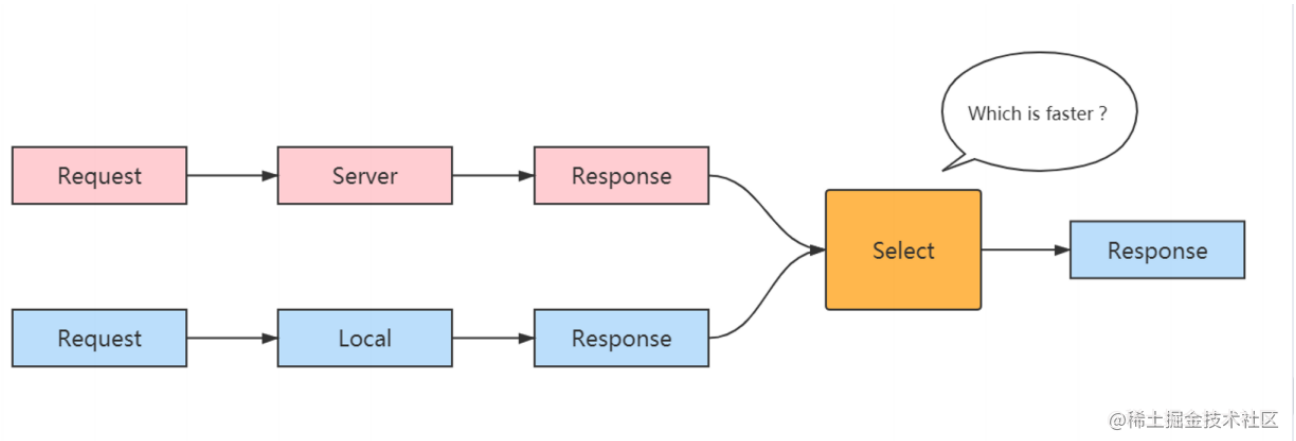
从这个运行效果可以看出：多个消费者，能够同时接收同一个生成者相同的信息，并没有互斥性！

## 2、select-多路复用

### 什么是多路复用

数据通信系统或计算机网络系统中，传输媒体的宽带或容量往往会大于传输单一信号的需求，为了有效的利用通信线路，希望**一个信道同时传输多路信息**，这就是所谓的多路复用技术(Multiplexing)

### 2.1 复用多个await




### 如图所示

两个API分别从网络和本地缓存获取数据，期望哪个先返回就先用哪个做展示。

#### 2.1.1 开始实战

### 服务端

java 复制代码

```java
public class UserServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws Se
        String user = request.getParameter("user");
        if(user != null){
            System.out.println(user);
        }
        System.out.println("doGet");
        PrintWriter out = response.getWriter();
        JsonObject jsonObject = new JsonObject();
        jsonObject.addProperty("name","jason");
        jsonObject.addProperty("address","California");
        out.write(jsonObject.toString());
        System.out.println(jsonObject.toString());
        out.close();
    }
}
```

服务端使用的是最原始的 HttpServlet+TomCat 方式，没有用现在的SpringBoot，代码也很简单，就不过多说明了。

### 客户端

kotlin 复制代码

```kotlin
private val cachePath = "E://coroutine.cache" //该文件里面内容为：{"name":"hqk","address":"成都
private val gson = Gson()

data class Response<T>(val value: T, val isLocal: Boolean)

//通过本地加载用户信息
fun CoroutineScope.getUserFromLocal(name: String) = async(Dispatchers.IO) {
    delay(10000) //故意的延迟  挂起10秒
    File(cachePath).readText().let { gson.fromJson(it, User::class.java) }
}

//通过网络加载用户信息
fun CoroutineScope.getUserFromRemote(name: String) = async(Dispatchers.IO) {
    userServiceApi.getUser(name)
}

class CoroutineTest02 {
    @Test
    fun `test select await`() = runBlocking<Unit> {
```

```
            GlobalScope.launch {
                val localRequest = getUserFromLocal("xxx")
                val remoteRequest = getUserFromRemote("yyy")

                val userResponse = select<Response<User>> {
                    localRequest.onAwait { Response(it, true) }
                    remoteRequest.onAwait { Response(it, false) }
                }

                userResponse.value?.let { println(it) }
            }.join()
        }

    }

    //定义用户数据类
    data class User(val name: String, val address: String)

    //Retrofit 网络数据请求
    val userServiceApi: UserServiceApi by lazy {
        val retrofit = retrofit2.Retrofit.Builder()
                .client(OkHttpClient.Builder().addInterceptor {
                    it.proceed(it.request()).apply {
                        Log.d("hqk", "request:${code()}")
                        //Log.d("hqk", "boy:${body()?.string()}")
                    }
                }.build())
                .baseUrl("http://10.0.0.130:8080/kotlinstudyserver/")
                .addConverterFactory(GsonConverterFactory.create())
                .build()
        retrofit.create(UserServiceApi::class.java)
    }


    interface UserServiceApi {

            //获取用户信息
        @GET("user")
        suspend fun getUser(@Query("name") name: String) : User
    }
```

这里可以看到 `@Test` 测试类里面分别调用了获取本地、网络用户的方法，并在 `select{}` 里面分别调用了对应方法的 `onAwait` ，返回 `userResponse` 对象

## 来看看运行效果

复制代码

```
User(name=jason, address=California)
```

因为获取本地用户那里挂起了10秒，而网络请求的数据的时间小于本地加载时间，因此这里，加载的是网络数据。

**那如果说将本地挂起10给注释掉，再次运行看看效果：**

复制代码

```
User(name=hqk, address=成都)
```

很明显，这里加载是本地数据，而非网络数据。

由此，可以得出：**当复用多个await时，谁先返回，那就先用哪个做展示**

## 2.2 复用多个Channel

跟await类似，会接收到最快的那个Channel消息。

kotlin 复制代码

```kotlin
@Test
fun `test select channel`() = runBlocking<Unit> {
    val channels = listOf(Channel<Int>(), Channel<Int>())
    GlobalScope.launch {
        delay(100)
        channels[0].send(200)
    }

    GlobalScope.launch {
        delay(50)
        channels[1].send(100)
    }

    val result = select<Int?> {
        channels.forEach { channel ->
            channel.onReceive { it }
        }
    }
    println(result)
}
```

**先来看看运行效果：**

复制代码

```
100
```

这里我们看到，通过 `listOf` 将对应通道整合成一个list集合，然后分别开了两个协程，在对应协程里分别挂起的不同的时间。最后我们看到接收了执行了耗时较短的通道信息！

### 2.3 SelectClause

我们怎么知道哪些事件可以被select呢？其实所有能够被select的时间都是SelectClauseN类型，包括：

- SelectClause0：对应事件没有返回值，例如join没有返回值，那么onJoin就是SelectClauseN类型。使用时，onJoin的参数是一个无参函数。
- SelectClause1：对应事件有返回值，上面的onAwait和onReceive都是此类情况（下面就不举该例）
- SelectClause2：对应事件有返回值，此外还需要一个额外的参数，例如Channel.onSend有两个参数，第一个是Channel数据类型的值，表示即将发送的值；第二个是发送成功时的回调函数。

如果我们想要确认挂起函数是否支持select，只需要查看其**是否存在对应的SelectClauseN类型**可回调即可。

概念说了一大堆，分别实战看看效果：

#### 2.3.1 示例一（SelectClause0）

kotlin 复制代码

```kotlin
@Test
fun `test SelectClause0`() = runBlocking<Unit> {
    val job1 = GlobalScope.launch {
        delay(100)
        println("job 1")
    }

    val job2 = GlobalScope.launch {
        delay(10)
        println("job 2")
    }

    select<Unit> {
        job1.onJoin { println("job 1 onJoin") }
        job2.onJoin { println("job 2 onJoin") }
    }

    delay(1000)
```

```
}
```

### 来看看运行效果：

复制代码

```
job 2
job 2 onJoin
job 1
```

这是一个非常标准的协程，对应事件没有任何返回值的，这个就是上面所说的 `SelectClause0` 类型。

### 2.3.2 示例二（SelectClause2）

kotlin 复制代码

```kotlin
@Test
fun `test SelectClause2`() = runBlocking<Unit> {
    val channels = listOf(Channel<Int>(), Channel<Int>())
    println(channels)
    launch(Dispatchers.IO) {
         select<Unit?> {
            launch {
                delay(10)
                channels[1].onSend(200) { sentChannel ->
                    println("sent 1 on $sentChannel")
                }
            }
            launch {
                delay(100)
                channels[0].onSend(100) { sentChannel ->
                    println("sent 0 on $sentChannel")
                }
            }
        }
    }
    GlobalScope.launch {
        println(channels[0].receive())
    }
    GlobalScope.launch {
        println(channels[1].receive())
    }
    delay(1000)
}
```

### 来看看运行效果

复制代码

```
[RendezvousChannel@2a084b4c{EmptyQueue}, RendezvousChannel@42b93f6b{EmptyQueue}]
200
sent 1 on RendezvousChannel@42b93f6b{EmptyQueue} //回调成功执行业务逻辑—打印
```

这里我们看到使用了 channels.onSend 方式，上面所说，第一个参数为对应类型，第二个参数就会回调函数，也就是说，后面大括号里面的内容就会回调成功的业务逻辑处理。

## 2.4 使用Flow实现多路复用

多数情况下，我们可以通过构造合适的Flow来实现多路复用的效果。

kotlin 复制代码

```kotlin
private val cachePath = "E://coroutine.cache" //该文件里面内容为：{"name":"hqk","address":"成都
private val gson = Gson()

data class Response<T>(val value: T, val isLocal: Boolean)

//通过本地获取用户信息
fun CoroutineScope.getUserFromLocal(name: String) = async(Dispatchers.IO) {
//    delay(10000) //故意的延迟
    File(cachePath).readText().let { gson.fromJson(it, User::class.java) }
}

//通过网络获取用户信息
fun CoroutineScope.getUserFromRemote(name: String) = async(Dispatchers.IO) {
    userServiceApi.getUser(name)
}



class CoroutineTest02 {
    @Test
    fun `test select flow`() = runBlocking<Unit> {
        // 函数 -> 协程 -> Flow -> Flow合并
        val name = "guest"
        coroutineScope {
                //通过作用域，将对应方法调用添加至list集合里
            listOf(::getUserFromLocal, ::getUserFromRemote)
                //遍历集合每个方法，function 就为对应的某个方法
                .map { function ->
                    function.call(name) //这里调用对应方法后，将返回的结果传至下个map里
                }.map { deferred -> //这里对应deferred 表示对应方法返回的结果
                    flow { emit(deferred.await()) }//这里表示，得到谁，就通过flow 发射值
```

```
                }.merge() //流 合并
                .collect { user -> println(user) } //这里只管接收flow对应发射值

        }
    }
}
```

一切尽在注释中，

### 来看看运行效果

复制代码

```
User(name=hqk, address=成都)
User(name=jason, address=California)
```

这里我们看到，本地和网络都成功的收到了！

## 3、并发安全

### 3.1 不安全的并发访问

我们使用线程在解决并发问题的时候总是会遇到线程安全的问题，而Java平台上的Kotlin协程实现免不了存在并发调度的情况，因此线程安全同样值得留意。

比如说：

kotlin 复制代码

```kotlin
@Test
fun `test not safe concurrent`() = runBlocking<Unit> {
    var count = 0
    List(1000) {
        GlobalScope.launch { count++ }
    }.joinAll()
    println(count)
}
```

我们可以看到，这里开启了1000个协程并发，每个协程都对 `count` 自加一，理想情况下应该为1000

### 来看看具体效果如何

复制代码

```
973 //每次重新运行值都不一样
```

现在我们看到真实效果值，并非理想情况，因此我们需要重视并发情况！

### 3.2 协程的并发工具

除了我们在线程中常用的解决并发问题的手段之外，协程框架也提供了一些并发的安全工具，包括：

- `Channel`： 并发安全的消息通道，我们已经非常熟悉
- `Mutex`： 轻量级锁，它的 `lock和unlock` 从语义上与线程锁比较类似，之所以轻量是因为它在获取不到锁时不会阻塞线程，而是挂起等待锁的释放；
- `Semaphore`： 轻量级信号量，信号量可以有多个，协程在获取到信号量后即可执行并发操作。
    - 当 `Semaphore` 的参数为1时，效果等价于 `Mutex`

说了那么多，上手试试！

#### 3.2.1 示例一（使用AtomicXXX）

kotlin 复制代码

```kotlin
@Test
fun `test safe concurrent`() = runBlocking<Unit> {
    var count = AtomicInteger(0)
    List(1000) {
        GlobalScope.launch { count.incrementAndGet() }
    }.joinAll()
    println(count.get())
}
```

这个是比较Java常规的解决方案：通过原子操作类解决

运行效果就1000，效果就不贴了。

#### 3.2.2 示例二（使用Mutex）

kotlin 复制代码

```kotlin
@Test
 fun `test safe concurrent tools`() = runBlocking<Unit> {
    var count = 0
    val mutex = Mutex()
```

```
        List(1000) {
            GlobalScope.launch {
                mutex.withLock {
                    count++
                }
            }
        }.joinAll()
        println(count)
    }
```

我们可以看到，在协程开始前初始化了 `Mutex` 对象，在对应协程自加操作前通过 `mutex.withLock` 将对应逻辑上锁。

接下来看下一个！

### 3.2.3 示例三（使用Semaphore）

kotlin 复制代码

```
@Test
fun `test safe concurrent tools2`() = runBlocking<Unit> {
    var count = 0
    val semaphore = Semaphore(1)
    List(1000) {
        GlobalScope.launch {
            semaphore.withPermit {
                count++
            }
        }
    }.joinAll()
    println(count)
}
```

这里我们可以看到通过 `Semaphore(1)` 得到了对应对象，然后在并发逻辑处额外用 `semaphore.withPermit` 解决了并发安全问题。

## 3.3 避免访问外部可变状态

kotlin 复制代码

```
@Test
fun `test avoid access outer variable`() = runBlocking<Unit> {
    var count = 0
    val result = count + List(1000){
        GlobalScope.async { 1 }
```

```
        }.map { it.await() }.sum()
        println(result)
    }
```

编写函数时要求它不得访问外部状态，只能基于参数做运算，通过返回值提供运算结果

## 结束语

好了，本篇到这里就结束了！相信看到这的小伙伴应该对Channel有所了解！在下一篇中，将会详解**协程Flow的综合应用**

分类： Android 标签： Android Kotlin