

## 重磅加餐 | 带你快速入门Scala语言

2020-05-02 胡夕

Kafka核心源码解读

[进入课程 >](#)



讲述：胡夕

时长 14:22 大小 13.17M



你好，我是胡夕。最近，我在留言区看到一些同学反馈说“Scala 语言不太容易理解”，于是，我决定临时加一节课，给你讲一讲 Scala 语言的基础语法，包括变量和函数的定义、元组的写法、函数式编程风格的循环语句的写法、它独有的 case 类和强大的 match 模式匹配功能，以及 Option 对象的用法。

学完这节课以后，相信你能够在较短的时间里掌握这些实用的 Scala 语法，特别是 Kafka 源码中用到的 Scala 语法特性，彻底扫清源码阅读路上的编程语言障碍。



**Java 函数式编程**


就像我在开篇词里面说的，你不熟悉 Scala 语言其实并没有关系，但你至少要对 Java 8 的函数式编程有一定的了解，特别是要熟悉 Java 8 Stream 的用法。

倘若你之前没有怎么接触过 Lambda 表达式和 Java 8 Stream，我给你推荐一本好书：

《Java 8 实战》。这本书通过大量的实例深入浅出地讲解了 Lambda 表达式、Stream 以及函数式编程方面的内容，你可以去读一读。

现在，我就给你分享一个实际的例子，借着它开始我们今天的所有讨论。


TopicPartition 是 Kafka 定义的主题分区类，它建模的是 Kafka 主题的分区对象，其关键代码如下：

 复制代码

```
1 public final class TopicPartition implements Serializable {
2     private final int partition;
3     private final String topic;
4     // 其他字段和方法.....
5 }
```

对于任何一个分区而言，一个 TopicPartition 实例最重要的就是 **topic 和 partition 字段**，即 **Kafka 的主题和分区号**。假设给定了一组分区对象 List < TopicPartition >，我想要找出分区数大于 3 且以 “test” 开头的所有主题列表，我应该怎么写这段 Java 代码呢？你可以先思考一下，然后再看下面的答案。

我先给出 Java 8 Stream 风格的答案：

 复制代码

```
1 // 假设分区对象列表变量名是list
2 Set<String> topics = list.stream()
3     .filter(tp -> tp.topic().startsWith("test-"))
4     .collect(Collectors.groupingBy(TopicPartition::topic, Collectors.counting()))
5     .entrySet().stream()
6     .filter(entry -> entry.getValue() > 3)
7     .map(entry -> entry.getKey()).collect(Collectors.toSet());
```

这是典型的 Java 8 Stream 代码，里面大量使用了诸如 filter、map 等操作算子，以及 Lambda 表达式，这让代码看上去一气呵成，而且具有很好的可读性。

我从第 3 行开始解释下每一行的作用：第 3 行的 filter 方法调用实现了筛选以 “test” 开头主题的功能；第 4 行是运行 collect 方法，同时指定使用 groupingBy 统计分区数并按照主题进行分组，进而生成一个 Map 对象；第 5~7 行是提取出这个 Map 对象的所有 <K, V> 对，然后再次调用 filter 方法，将分区数大于 3 的主题提取出来；最后是将这些主题做成一个集合返回。

其实，给出这个例子，我只是想说明，**Scala 语言的编写风格和 Java 8 Stream 有很多相似之处**：一方面，代码中有大量的 filter、map，甚至是 flatMap 等操作算子；另一方面，代码的风格也和 Java 中的 Lambda 表达式写法类似。

如果你不信的话，我们来看下 Kafka 中计算消费者 Lag 的 getLag 方法代码：

 复制代码

```
1 private def getLag(offset: Option[Long], logEndOffset: Option[Long]): Option[Long] = {
2   offset.filter(_ != -1).flatMap(offset => logEndOffset.map(_ - offset))
3 }
```

你看，这里面也有 filter 和 map。是不是和上面的 Java 代码有异曲同工之妙？

如果你现在还看不懂这个方法的代码是什么意思，也不用着急，接下来我会带着你一步一步来学习。我相信，学完了这节课以后，你一定能自主搞懂 getLag 方法的源码含义。

getLag 代码是非常典型的 Kafka 源码，一旦你熟悉了这种编码风格，后面一定可以举一反三，一举攻克其他的源码阅读难题。

我们先从 Scala 语言中的变量 (Variable) 开始说起。毕竟，不管是学习任何编程语言，最基础的就是先搞明白变量是如何定义的。

## 定义变量和函数

Scala 有两类变量：**val** 和 **var**。**val** 等同于 Java 中的 **final** 变量，一旦被初始化，就不能再被重新赋值了。相反地，**var** 是非 **final** 变量，可以重复被赋值。我们看下这段代码：

 复制代码

```
1 scala> val msg = "hello, world"
2 msg: String = hello, world
3 
```

```
4 scala> msg = "another string"
5 <console>:12: error: reassignment to val
6     msg = "another string"
7
8 scala> var a:Long = 1L
9 a: Long = 1
10
11 scala> a = 2
12 a: Long = 2
```

很直观，对吧？msg 是一个 val，a 是一个 var，所以 msg 不允许被重复赋值，而 a 可以。我想提醒你的是，**变量后面可以跟“冒号 + 类型”，以显式标注变量的类型**。比如，这段代码第 6 行的 “: Long”，就是告诉我们变量 a 是一个 Long 型。当然，如果你不写 “: Long”，也是可以的，因为 Scala 可以通过后面的值 “1L” 自动判断出 a 的类型。

不过，很多时候，显式标注上变量类型，可以让代码有更好的可读性和可维护性。

下面，我们来看下 Scala 中的函数如何定义。我以获取两个整数最大值的 Max 函数为例，进行说明，代码如下：

```
1 def max(x: Int, y: Int): Int = {
2   if (x > y) x
3   else y
4 }
```

 复制代码

首先，def 关键字表示这是一个函数。max 是函数名，括号中的 x 和 y 是函数输入参数，它们都是 Int 类型的值。结尾的 “Int =” 组合表示 max 函数返回一个整数。

其次，max 代码使用 if 语句比较 x 和 y 的大小，并返回两者中较大的值，但是它没有使用所谓的 return 关键字，而是直接写了 x 或 y。**在 Scala 中，函数体具体代码块最后一行的值将被作为函数结果返回**。在这个例子中，if 分支代码块的最后一行是 x，因此，此路分支返回 x。同理，else 分支返回 y。

讲完了 max 函数，我再用 Kafka 源码中的一个真实函数，来帮你进一步地理解 Scala 函数：

```

1 def deleteIndicesIfExist(
2   // 这里参数suffix的默认值是"", 即空字符串
3   // 函数结尾处的Unit类似于Java中的void关键字, 表示该函数不返回任何结果
4   baseFile: File, suffix: String = ""): Unit = {
5   info(s"Deleting index files with suffix $suffix for baseFile $baseFile")
6   val offset = offsetFromFile(baseFile)
7   Files.deleteIfExists(Log.offsetIndexFile(dir, offset, suffix).toPath)
8   Files.deleteIfExists(Log.timeIndexFile(dir, offset, suffix).toPath)
9   Files.deleteIfExists(Log.transactionIndexFile(dir, offset, suffix).toPath)
10  }

```

和上面的 max 函数相比, 这个函数有两个额外的语法特性需要你了解。

第一个特性是**参数默认值**, 这是 Java 不支持的。这个函数的参数 suffix 默认值是空字符串, 因此, 以下两种调用方式都是合法的:

```

1 deleteIndicesIfExist(baseFile) // OK
2 deleteIndicesIfExist(baseFile, ".swap") // OK

```

第二个特性是**该函数的返回值 Unit**。Scala 的 Unit 类似于 Java 的 void, 因此, deleteIndicesIfExist 函数的返回值是 Unit 类型, 表明它仅仅是执行一段逻辑代码, 不需要返回任何结果。

## 定义元组 (Tuple)

接下来, 我们来看下 Scala 中的元组概念。**元组是承载数据的容器, 一旦被创建, 就不能再被更改了**。元组中的数据可以是不同数据类型的。定义和访问元组的方法很简单, 请看下面的代码:


```

1 scala> val a = (1, 2.3, "hello", List(1,2,3)) // 定义一个由4个元素构成的元组, 每个元
2 a: (Int, Double, String, List[Int]) = (1,2.3,hello,List(1, 2, 3))
3
4 scala> a._1 // 访问元组的第一个元素
5 res0: Int = 1
6
7 scala> a._2 // 访问元组的第二个元素
8 res1: Double = 2.3

```

```
9 scala> a._3 // 访问元组的第三个元素
10 res2: String = hello
11
12 scala> a._4 // 访问元组的第四个元素
13 res3: List[Int] = List(1, 2, 3)
14
```

总体上而言，元组的用法简单而优雅。Kafka 源码中也有很多使用元组的例子，比如：

 复制代码


```
1 def checkEnoughReplicasReachOffset(requiredOffset: Long): (Boolean, Errors) =
2     .....
3     if (minIsr <= curInSyncReplicaIds.size) {
4         .....
5         (true, Errors.NONE)
6     } else
7         (false, Errors.NOT_ENOUGH_REPLICAS_AFTER_APPEND)
8 }
```

checkEnoughReplicasReachOffset 方法返回一个 (Boolean, Errors) 类型的元组，即元组的第一个元素或字段是 Boolean 类型，第二个元素是 Kafka 自定义的 Errors 类型。

该方法会判断某分区 ISR 中副本的数量，是否大于等于所需的最小 ISR 副本数，如果是，就返回 (true, Errors.NONE) 元组，否则返回 (false, Errors.NOT\_ENOUGH\_REPLICAS\_AFTER\_APPEND)。目前，你不必理会代码中 minIsr 或 curInSyncReplicaIds 的含义，仅仅掌握 Kafka 源码中的元组用法就够了。

## 循环写法

下面我们来看下 Scala 中循环的写法。我们常见的循环有两种写法：**命令式编程方式**和**函数式编程方式**。我们熟悉的是第一种，比如下面的 for 循环代码：

 复制代码

```
1 scala> val list = List(1, 2, 3, 4, 5)
2 list: List[Int] = List(1, 2, 3, 4, 5)
3
4 scala> for (element <- list) println(element)
5 1
6 2
7 3
8 4
```

Scala 支持的函数式编程风格的循环，类似于下面的这种代码：

[复制代码](#)

```
1 scala> list.foreach(e => println(e))
2 // 省略输出.....
3 scala> list.foreach(println)
4 // 省略输出.....
```

特别是代码中的第二种写法，会让代码写得异常简洁。我用一段真实的 Kafka 源码再帮你加强下记忆。它取自 SocketServer 组件中 stopProcessingRequests 方法，主要目的是让 Broker 停止请求和新入站 TCP 连接的处理。SocketServer 组件是实现 Kafka 网络通信的重要组件，后面我会花 3 节课的时间专门讨论它。这里，咱们先来学习下这段明显具有函数式风格的代码：

[复制代码](#)

```
1 // dataPlaneAcceptors:ConcurrentHashMap<Endpoint, Acceptor>对象
2 dataPlaneAcceptors.asScala.values.foreach(_.initiateShutdown())
```

这一行代码首先调用 asScala 方法，将 Java 的 ConcurrentHashMap 转换成 Scala 语言中的 concurrent.Map 对象；然后获取它保存的所有 Acceptor 线程，通过 foreach 循环，调用每个 Acceptor 对象的 initiateShutdown 方法。如果这个逻辑用命令式编程来实现，至少要几行甚至是十几行才能完成。

## case 类

在 Scala 中，case 类与普通类是类似的，只是它具有一些非常重要的不同点。Case 类非常适合用来表示不可变数据。同时，它最有一个特点是，case 类自动地为所有类字段定义 Getter 方法，这样能省去很多样板代码。我举个例子说明一下。

如果我们要编写一个类表示平面上的一个点，Java 代码大概长这个样子：

[复制代码](#)

```
1 public final class Point {
```



```
2  private int x;
3  private int y;
4  public Point(int x, int y) {
5      this.x = x;
6      this.y = y;
7  }
8  // setter methods.....
9  // getter methods.....
10 }
```

我就不列出完整的 Getter 和 Setter 方法了，写过 Java 的你一定知道这些样本代码。但如果用 Scala 的 case 类，只需要写一行代码就可以了：

 复制代码


```
1  case class Point(x:Int, y: Int) // 默认写法。不能修改x和y
2  case class Point(var x: Int, var y: Int) // 支持修改x和y
```

Scala 会自动地帮你创建出 x 和 y 的 Getter 方法。默认情况下，x 和 y 不能被修改，如果要支持修改，你要采用上面代码中第二行的写法。

## 模式匹配

有了 case 类的基础，接下来我们就可以学习下 Scala 中强大的模式匹配功能了。

和 Java 中 switch 仅仅只能比较数值和字符串相比，Scala 中的 match 要强大得多。我先来举个例子：

 复制代码

```
1  def describe(x: Any) = x match {
2      case 1 => "one"
3      case false => "False"
4      case "hi" => "hello, world!"
5      case Nil => "the empty list"
6      case e: IOException => "this is an IOException"
7      case s: String if s.length > 10 => "a long string"
8      case _ => "something else"
9  }
```



这个函数的 x 是 Any 类型，这相当于 Java 中的 Object 类型，即所有类的父类。注意倒数第二行的 “case \_” 的写法，它是用来兜底的。如果上面的所有 case 分支都不匹配，那就进入到这个分支。另外，它还支持一些复杂的表达式，比如倒数第三行的 case 分支，表示 x 是字符串类型，而且 x 的长度超过 10 的话，就进入到这个分支。


要知道，Java 在 JDK 14 才刚刚引入这个相同的功能，足见 Scala 语法的强大和便捷。

## Option 对象

最后，我再介绍一个小的语法特性或语言特点：**Option 对象**。

实际上，Java 也引入了类似的类：Optional。根据我的理解，不论是 Scala 中的 Option，还是 Java 中的 Optional，都是用来帮助我们更好地规避 NullPointerException 异常的。

Option 表示一个容器对象，里面可能装了值，也可能没有装任何值。由于是容器，因此一般都是这样的写法：Option[Any]。中括号里面的 Any 就是上面说到的 Any 类型，它能是任何类型。如果值存在的话，就可以使用 Some(x) 来获取值或给值赋值，否则就使用 None 来表示。我用一段代码帮助你理解：

 复制代码

```
1 scala> val keywords = Map("scala" -> "option", "java" -> "optional") // 创建一个
2 keywords: scala.collection.immutable.Map[String,String] = Map(scala -> option,
3
4 scala> keywords.get("java") // 获取key值为java的value值。由于值存在故返回Some(option
5 res24: Option[String] = Some(optional)
6
7 scala> keywords.get("C") // 获取key值为C的value值。由于不存在故返回None
8 res23: Option[String] = None
```

Option 对象还经常与模式匹配语法一起使用，以实现不同情况下的处理逻辑。比如，Option 对象有值和没有值时分别执行什么代码。具体写法你可以参考下面这段代码：

 复制代码

```
1 def display(game: Option[String]) = game match {
2   case Some(s) => s
3   case None => "unknown"
4 }
```

```
5 scala> display(Some("Heroes 3"))
6 res26: String = Heroes 3
7
8 scala> display(Some("StarCraft"))
9 res27: String = StarCraft
10
11 scala> display(None)
12 res28: String = unknown
13
```

## 总结

今天，我们专门花了些时间快速地学习了一下 Scala 语言的语法，这些语法能够帮助你更快速地上手 Kafka 源码的学习。现在，让我们再来看下这节课刚开始时我提到的 getLag 方法源码，你看看现在是否能够说出它的含义。我再次把它贴出来：

 复制代码

```
1 private def getLag(offset: Option[Long], logEndOffset: Option[Long]): Option[Long] = {
2   offset.filter(_ != -1).flatMap(offset => logEndOffset.map(_ - offset))
3 }
```

现在，你应该知道了，它是一个函数，接收两个类型为 Option[Long] 的参数，同时返回一个 Option[Long] 的结果。代码逻辑很简单，首先判断 offset 是否有值且不能是 -1。这些都是在 filter 函数中完成的，之后调用 flatMap 方法计算 logEndOffset 值与 offset 的差值，最后返回这个差值作为 Lag。

这节课结束以后，语言问题应该不再是你学习源码的障碍了，接下来，我们就可以继续专心地学习源码了。借着这个机会，我还想跟你多说几句。

很多时候，我们都以为，要有足够强大的毅力才能把源码学习坚持下去，但实际上，毅力是在你读源码的过程中培养起来的。

考虑到源码并不像具体技术本身那样容易掌握，我力争用最清晰易懂的方式来讲这门课。所以，我希望你每天都能花一点点时间跟着我一起学习，我相信，到结课的时候，你不仅可以搞懂 Kafka Broker 端源码，还能提升自己的毅力。而毅力和执行力的提升，可能比技术本身的提升还要弥足珍贵。

另外，我还想给你分享一个小技巧：想要养成每天阅读源码的习惯，你最好把目标拆解得足够小。人的大脑都是有惰性的，比起“我每天要读 1000 行源码”，它更愿意接受“每天只读 20 行”。你可能会说，每天读 20 行，这也太少了吧？其实不是的。只要你读了 20 行源码，你就一定能再多读一些，“20 行”这个小目标只是为了促使你愿意开始去做这件事情。而且，即使你真的只读了 20 行，那又怎样？读 20 行总好过 1 行都没有读，对吧？

当然了，阅读源码经常会遇到一种情况，那就是读不懂某部分的代码。没关系，读不懂的代码，你可以选择先跳过。

如果你是个追求完美的人，那么对于读不懂的代码，我给出几点建议：

1. **多读几遍**。不要小看这个朴素的建议。有的时候，我们的大脑是很任性的，只让它看一遍代码，它可能“傲娇地表示不理解”，但你多给它看几遍，也许就恍然大悟了。
2. **结合各种资料来学习**。比如，社区或网上关于这部分代码的设计文档、源码注释或源码测试用例等。尤其是搞懂测试用例，往往是让我们领悟代码精神最快捷的办法了。

总之，阅读源码是一项长期的工程，不要幻想有捷径或一蹴而就，微小积累会引发巨大改变，我们一起加油。

## 学习计划

五一计划 📅

晒学习姿势  
「免费」领课程



【点击】图片，立即参加 >>>

上一篇 [导读 | 构建Kafka工程和源码阅读环境、Scala语言热身](#)

下一篇 [01 | 日志段：保存消息文件的对象是怎么实现的？](#)

## 精选留言 (3)

写留言



每天晒白牙

2020-05-02

很及时的加餐

展开 ▾

作者回复: 哈哈哈，一起加油！



1



许童童

2020-05-02

很好的入门指南

展开 ▾

作者回复: 谢谢，一起加油！



小崔

2020-05-02

Option的好处没看出来，该判断的地方并没有少。

展开 ▾

作者回复: 好处在于显式地告诉程序员这是一个可能为空的变量，需要小心谨慎处理：)



