JobGenerator 详解

[酷玩 Spark] Spark Streaming 源码解析系列，返回目录请 猛戳这里

「腾讯广告」技术团队（原腾讯广点通技术团队）荣誉出品

```
本系列内容适用范围：

* 2018.11.02 update, Spark 2.4 全系列 √ (已发布：2.4.0)
* 2018.02.28 update, Spark 2.3 全系列 √ (已发布：2.3.0 ~ 2.3.2)
* 2017.07.11 update, Spark 2.2 全系列 √ (已发布：2.2.0 ~ 2.2.3)
```

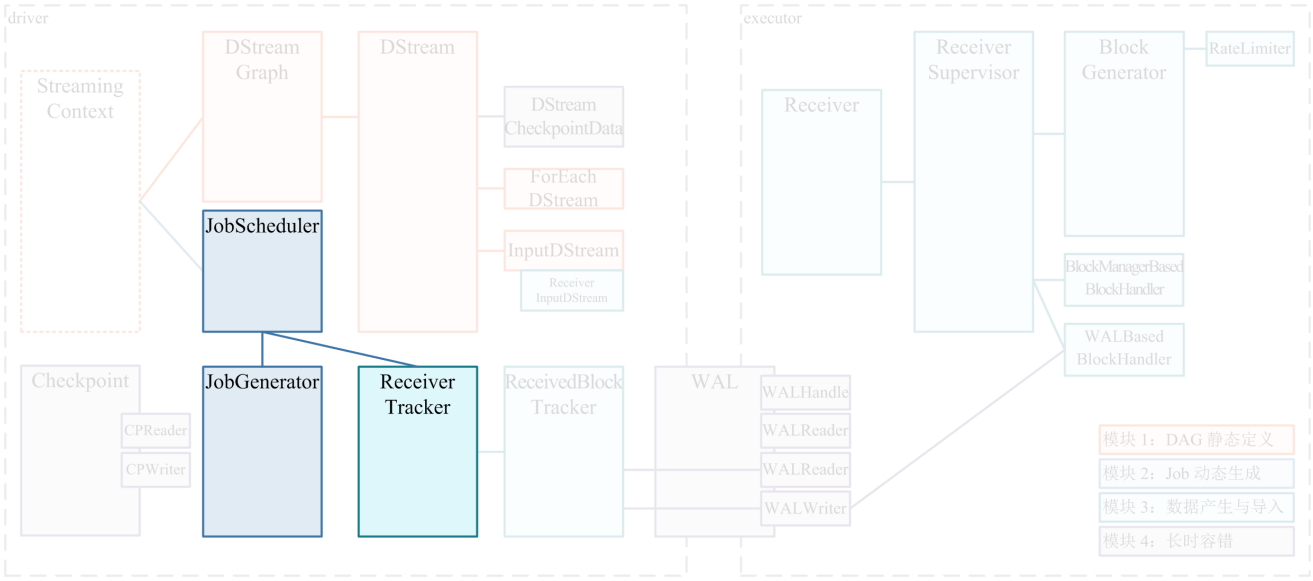
阅读本文前，请一定先阅读 Spark Streaming 实现思路与模块概述 一文，其中概述了 Spark Streaming 的 4 大模块的基本作用，有了全局概念后再看本文对 `模块 2：Job 动态生成` 细节的解释。

引言

前面在 Spark Streaming 实现思路与模块概述 和 DStream 生成 RDD 实例详解 里我们分析了 `DStream` 和 `DStreamGraph` 具有能够实例化 `RDD` 和 `RDD` DAG 的能力，下面我们来看 Spark Streaming 是如何将其动态调度的。

在 Spark Streaming 程序的入口，我们都会定义一个 `batchDuration`，就是需要每隔多长时间就比照静态的 `DStreamGraph` 来动态生成一个 RDD DAG 实例。在 Spark Streaming 里，总体负责动态作业调度的具体类是 `JobScheduler`，

`JobScheduler` 有两个非常重要的成员：`JobGenerator` 和 `ReceiverTracker`。`JobScheduler` 将每个 batch 的 RDD DAG 具体生成工作委托给 `JobGenerator`，而将源头输入数据的记录工作委托给 `ReceiverTracker`。




```
JobScheduler    的全限定名是：org.apache.spark.streaming.scheduler.JobScheduler
JobGenerator    的全限定名是：org.apache.spark.streaming.scheduler.JobGenerator
ReceiverTracker 的全限定名是：org.apache.spark.streaming.scheduler.ReceiverTracker
```

本文我们来详解 `JobScheduler`。

JobGenerator 启动

在用户 code 最后调用 `ssc.start()` 时，将隐含的导致一系列模块的启动，其中对我们 `JobGenerator` 这里的启动调用关系如下：

```
// 来自 StreamingContext.start(), JobScheduler.start(), JobGenerator.start()

ssc.start()                                 //  【用户 code：StreamingContext.start()】
    -> scheduler.start()                    //  【JobScheduler.start()】
                -> jobGenerator.start()     //  【JobGenerator.start()】
```

具体的看，`JobGenerator.start()` 的代码如下：

```
// 来自 JobGenerator.start()

def start(): Unit = synchronized {
  ...
  eventLoop.start()                          //  【启动 RPC 处理线程】

  if (ssc.isCheckpointPresent) {
    restart()                                //  【如果不是第一次启动，就需要从 checkpoint
恢复】
  } else {
    startFirstTime()                         //  【第一次启动，就 startFirstTime()】
  }
}
```

可以看到，在启动了 RPC 处理线程 `eventLoop` 后，就会根据是否是第一次启动，也就是是否存在 checkpoint，来具体的决定是 `restart()` 还是 `startFirstTime()`。

后面我们会分析失效后重启的 `restart()` 流程，这里我们来关注 `startFirstTime()`：

```
// 来自 JobGenerator.startFirstTime()

private def startFirstTime() {
  val startTime = new Time(timer.getStartTime())
  graph.start(startTime - graph.batchDuration)
  timer.start(startTime.milliseconds)
  logInfo("Started JobGenerator at " + startTime)
}
```

可以看到，这里首次启动时做的工作，先是通过 `graph.start()` 来告知了 `DStreamGraph` 第 1 个 batch 的启动时间，然后就是 `timer.start()` 启动了关键的定时器。

当定时器 `timer` 启动以后，`JobGenerator` 的 `startFirstTime()` 就完成了。

RecurringTimer

通过之前几篇文章的分析我们知道，**`JobGenerator` 维护了一个定时器**，周期就是用户设置的 `batchDuration`，**定时为每个 batch 生成 RDD DAG 的实例。**

具体的，这个定时器实例就是：

```
// 来自 JobGenerator

private[streaming]
class JobGenerator(jobScheduler: JobScheduler) extends Logging {
...
  private val timer = new RecurringTimer(clock,
ssc.graph.batchDuration.milliseconds,
      longTime => eventLoop.post(GenerateJobs(new Time(longTime))),
"JobGenerator")
...
}
```

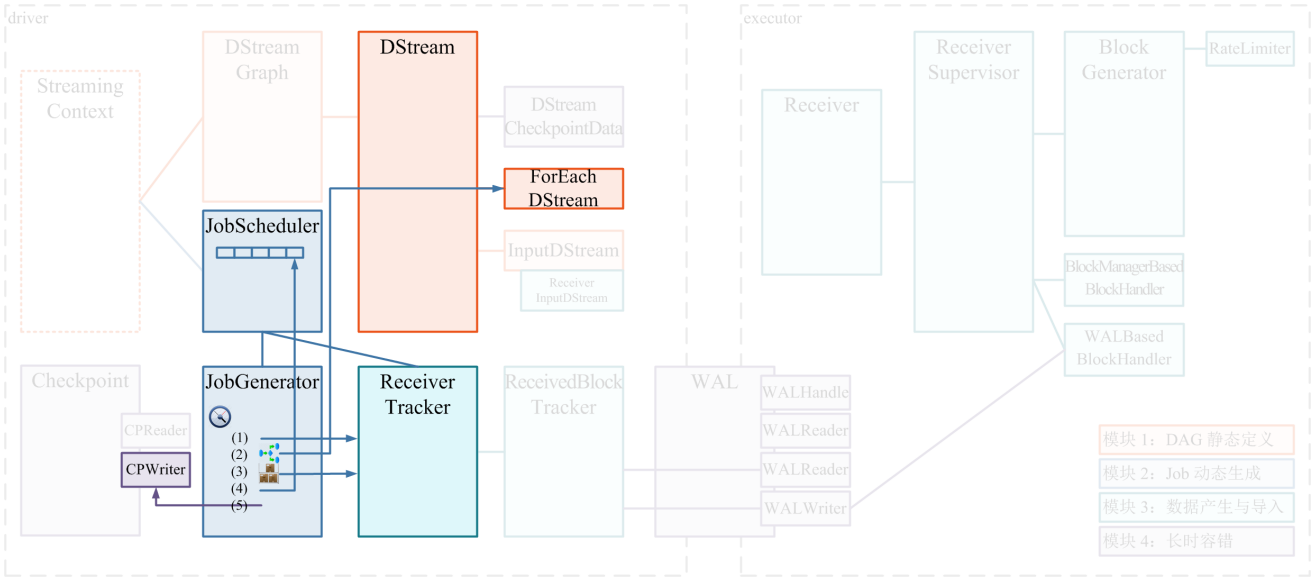
通过代码也可以看到，整个 `timer` 的调度周期就是 `batchDuration`，每次调度起来就是做一个非常简单的工作：往 `eventLoop` 里发送一个消息 —— 该为当前 batch (`new Time(longTime)`) GenerateJobs 了！

GenerateJobs

接下来，`eventLoop` 收到消息时，会在一个消息处理的线程池里，执行对应的操作。在这里，处理 `GenerateJobs(time)` 消息的对应操作是 `generateJobs(time)`：

```
private def generateJobs(time: Time) {
  SparkEnv.set(ssc.env)
  Try {
    jobScheduler.receiverTracker.allocateBlocksToBatch(time)                //
【步骤 (1)】
    graph.generateJobs(time)                                                //
【步骤 (2)】
  } match {
    case Success(jobs) =>
      val streamIdToInputInfos = jobScheduler.inputInfoTracker.getInfo(time) //
【步骤 (3)】
      jobScheduler.submitJobSet(JobSet(time, jobs, streamIdToInputInfos))    //
【步骤 (4)】
    case Failure(e) =>
      jobScheduler.reportError("Error generating jobs for time " + time, e)
  }
  eventLoop.post(DoCheckpoint(time, clearCheckpointDataLater = false))       //
【步骤 (5)】
}
```

这段代码异常精悍，包含了 `JobGenerator` 主要工作 —— 如下图所示 —— 的 5 个步骤！



- (1) **要求 `ReceiverTracker` 将目前已收到的数据进行一次 allocate**，即将上次 batch 切分后的数据切分到到本次新的 batch 里
  - 这里 `ReceiverTracker` 对已收到数据的 meta 信息进行 `allocateBlocksToBatch(time)`，与 `ReceiverTracker` 自己接收 `ReceiverSupervisorImpl` 上报块数据 meta 信息的过程，是相互独立的，但通过 `synchronized` 关键字来互斥同步
  - 即是说，不管 `ReceiverSupervisorImpl` 形成块数据的时间戳 `t1`、`ReceiverSupervisorImpl` 发送块数据的时间戳 `t2`、`ReceiverTracker` 收到块数据的时间戳 `t3` 分别是啥，最终块数据划入哪个 batch，还是由 `ReceiverTracker.allocateBlocksToBatch(time)` 方法获得 `synchronized` 锁的那一刻，还有未划入之前任何一个 batch 的块数据 meta，将被划分入最新的 batch
  - 所以，每个块数据的 meta 信息，将被划入一个、且只被划入一个 batch

- (2) **要求 `DStreamGraph` 复制出一套新的 RDD DAG 的实例**，具体过程是：`DStreamGraph` 将要求图里的尾 `DStream` 节点生成具体的 RDD 实例，并递归的调用尾 `DStream` 的上游 `DStream` 节点……以此遍历整个 `DStreamGraph`，遍历结束也就正好生成了 RDD DAG 的实例
  - 这个过程的详解，请参考前面的文章 DStream 生成 RDD 实例详解
  - 精确的说，整个 `DStreamGraph.generateJobs(time)` 遍历结束的返回值是 `Seq[Job]`

- (3) **获取第 1 步 `ReceiverTracker` 分配到本 batch 的源头数据的 meta 信息**
  - 第 1 步中 `ReceiverTracker` 只是对 batch 的源头数据 meta 信息进行了 batch 的分配，本步骤是按照 batch 时间来向 `ReceiverTracker` 查询得到划分到本 batch 的块数据 meta 信息

- (4) 将第 2 步生成的本 batch 的 RDD DAG，和第 3 步获取到的 meta 信息，**一同提交给 `JobScheduler` 异步执行**

    - 这里我们提交的是将 (a) `time` (b) `Seq[job]` (c) `块数据的 meta 信息` 这三者包装为一个 `JobSet`，然后调用 `JobScheduler.submitJobSet(JobSet)` 提交给 `JobScheduler`
    - 这里的向 `JobScheduler` 提交过程与 `JobScheduler` 接下来在 `jobExecutor` 里执行过程是异步分离的，因此本步将非常快即可返回

- (5) 只要提交结束（不管是否已开始异步执行），就**马上对整个系统的当前运行状态做一个 checkpoint**
    - 这里做 checkpoint 也只是异步提交一个 `DoCheckpoint` 消息请求，不用等 checkpoint 真正写完成即可返回
    - 这里也简单描述一下 checkpoint 包含的内容，包括已经提交了的、但尚未运行结束的 JobSet 等实际运行时信息。

（本文完，参与本文的讨论请 猛戳这里，返回目录请 猛戳这里）