# barrenlake

昵称：barrenlake
园龄：1年7个月
粉丝：4
关注：1
+加关注

< 2016年7月 >

| 日 | 一 | 二 | 三 | 四 | 五 | 六 |
|---|---|---|---|---|---|---|
| 26 | 27 | 28 | 29 | 30 | 1 | 2 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 1 | 2 | 3 | 4 | 5 | 6 |

搜索

找找看

谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类

java
mysql(2)
other
scala(2)
spark(8)
uml(1)

随笔档案

2016年4月 (2)
2015年10月 (2)
2015年6月 (1)
2015年4月 (1)
2015年3月 (3)
2015年1月 (1)
2014年12月 (1)
2014年11月 (4)

最新评论

1. Re:还好，我还在路上
赞博主！
--bruce_xu
2. Re:Spark Streaming Backpressure分析
写的很棒，是我看到的写的最好的
--bruce_xu
3. Re:还好，我还在路上
坚持写下去！！！
--meup

阅读排行榜

1. spark Association failed with

博客园 首页 博问 闪存 新随笔 联系 订阅 管理

随笔-15 评论-3 文章-0 trackbacks-0

## Spark Streaming Backpressure分析

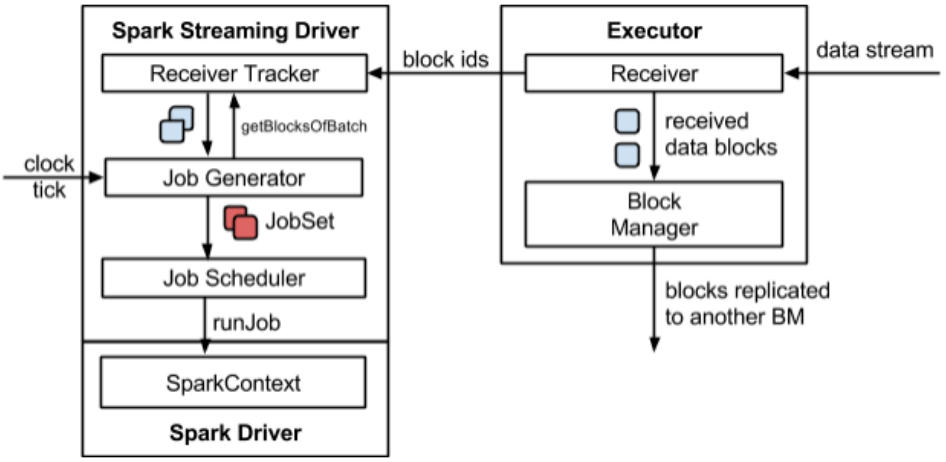
### 1、为什么引入Backpressure

默认情况下，Spark Streaming通过Receiver以生产者生产数据的速率接收数据，计算过程中会出现batch processing time > batch interval的情况，其中batch processing time 为实际计算一个批次花费时间， batch interval为Streaming应用设置的批处理间隔。这意味着Spark Streaming的数据接收速率高于Spark从队列中移除数据的速率，也就是数据处理能力低，在设置间隔内不能完全处理当前接收速率接收的数据。如果这种情况持续过长的时间，会造成数据在内存中堆积，导致Receiver所在Executor内存溢出等问题（如果设置StorageLevel包含disk, 则内存存放不下的数据会溢写至disk, 加大延迟）。Spark 1.5以前版本，用户如果要限制Receiver的数据接收速率，可以通过设置静态配制参数“spark.streaming.receiver.maxRate”的值来实现，此举虽然可以通过限制接收速率，来适配当前的处理能力，防止内存溢出，但也会引入其它问题。比如：producer数据生产高于maxRate，当前集群处理能力也高于maxRate，这就会造成资源利用率下降等问题。为了更好的协调数据接收速率与资源处理能力，Spark Streaming 从v1.5开始引入反压机制（back-pressure）,通过动态控制数据接收速率来适配集群数据处理能力。

### 2、Backpressure

Spark Streaming Backpressure: 根据JobScheduler反馈作业的执行信息来动态调整Receiver数据接收率。通过属性“spark.streaming.backpressure.enabled”来控制是否启用backpressure机制，默认值false，即不启用。

2.1 Streaming架构如下图所示（详见Streaming数据接收过程文档和Streaming 源码解析）




2.2 BackPressure执行过程如下图所示：

在原架构的基础上加上一个新的组件RateController,这个组件负责监听“OnBatchCompleted”事件，然后从中抽取processingDelay 及schedulingDelay信息. Estimator依据这些信息估算出最大处理速度（rate），最后由基于Receiver的Input Stream将rate通过ReceiverTracker与ReceiverSupervisorImpl转发给BlockGenerator（继承自RateLimiter）.

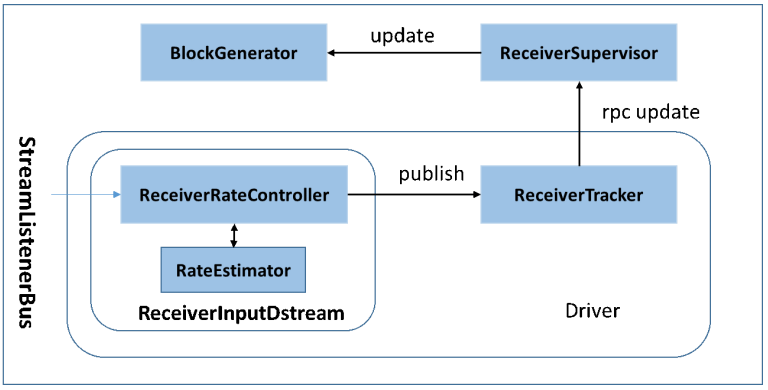
[akka.tcp:sparkMaster@ip:7077] (1386)
2. spark 监控--WebUi、Metrics System(315)
3. MLlib 卡方检验(244)
4. Spark 资源调度及任务调度(180)
5. Spark任务调度流程及调度策略分析(149)

评论排行榜

1. 还好，我还在路上(2)
2. Spark Streaming Backpressure分析(1)

推荐排行榜

1. 还好，我还在路上(2)
2. Spark Streaming Backpressure分析(1)




## 3、BackPressure 源码解析

### 3.1 RateController类体系

RatenController 继承自StreamingListener. 用于处理BatchCompleted事件。核心代码为：

```
**
 * A StreamingListener that receives batch completion updates, and maintains
 * an estimate of the speed at which this stream should ingest messages,
 * given an estimate computation from a `RateEstimator`
 */
private[streaming] abstract class RateController(val streamUID: Int, rateEstimator: Rate
extends StreamingListener with Serializable {
……
……  /**
   * Compute the new rate limit and publish it asynchronously.
   */
  private def computeAndPublish(time: Long, elems: Long, workDelay: Long, waitDelay: Lon
    Future[Unit] {
      val newRate = rateEstimator.compute(time, elems, workDelay, waitDelay)
      newRate.foreach { s =>
        rateLimit.set(s.toLong)
        publish(getLatestRate())
      }
    }
  def getLatestRate(): Long = rateLimit.get()

  override def onBatchCompleted(batchCompleted: StreamingListenerBatchCompleted) {
    val elements = batchCompleted.batchInfo.streamIdToInputInfo
    for {
      processingEnd <- batchCompleted.batchInfo.processingEndTime
      workDelay <- batchCompleted.batchInfo.processingDelay
      waitDelay <- batchCompleted.batchInfo.schedulingDelay
      elems <- elements.get(streamUID).map(_.numRecords)
    } computeAndPublish(processingEnd, elems, workDelay, waitDelay)
  }
}
```

### 3.2 RateController的注册

JobScheduler启动时会抽取在DStreamGraph中注册的所有InputDstream中的rateController，并向ListenerBus注册监听. 此部分代码如下：

```
def start(): Unit = synchronized {
   if (eventLoop != null) return // scheduler has already been started

   logDebug("Starting JobScheduler")
   eventLoop = new EventLoop[JobSchedulerEvent]("JobScheduler") {
     override protected def onReceive(event: JobSchedulerEvent): Unit = processEvent(eve

```

```
    override protected def onError(e: Throwable): Unit = reportError("Error in job sche
  }
  eventLoop.start()

  // attach rate controllers of input streams to receive batch completion updates
 <span style="color: #800000;">  for {
    inputDStream <- ssc.graph.getInputStreams
    rateController <- inputDStream.rateController
  } ssc.addStreamingListener(rateController)</span>

  listenerBus.start()
  receiverTracker = new ReceiverTracker(ssc)
  inputInfoTracker = new InputInfoTracker(ssc)
  receiverTracker.start()
  jobGenerator.start()
  logInfo("Started JobScheduler")
 }
```

### 3.3 BackPressure执行过程分析

BackPressure 执行过程分为BatchCompleted事件触发时机和事件处理两个过程

#### 3.3.1 BatchCompleted触发过程

对BatchedCompleted的分析，应该从JobGenerator入手，因为BatchedCompleted是批次处理结束的标志，也就是JobGenerator产生的作业执行完成时触发的，因此进行作业执行分析。

Streaming 应用中JobGenerator每个Batch Interval都会为应用中的每个Output Stream建立一个Job, 该批次中的所有Job组成一个Job Set.使用JobScheduler的submitJobSet进行批量Job提交。此部分代码结构如下所示

```
 /** Generate jobs and perform checkpoint for the given `time`.  */
 private def generateJobs(time: Time) {
   // Set the SparkEnv in this thread, so that job generation code can access the envir
   // Example: BlockRDDs are created in this thread, and it needs to access BlockManage
   // Update: This is probably redundant after threadlocal stuff in SparkEnv has been r
   SparkEnv.set(ssc.env)

   // Checkpoint all RDDs marked for checkpointing to ensure their lineages are
   // truncated periodically. Otherwise, we may run into stack overflows (SPARK-6847).
   ssc.sparkContext.setLocalProperty(RDD.CHECKPOINT_ALL_MARKED_ANCESTORS, "true")
   Try {
     jobScheduler.receiverTracker.allocateBlocksToBatch(time) // allocate received bloc
     graph.generateJobs(time) // generate jobs using allocated block
   } match {
     case Success(jobs) =>
       val streamIdToInputInfos = jobScheduler.inputInfoTracker.getInfo(time)
      <span style="color: #ff0000;"> jobScheduler.submitJobSet(JobSet(time, jobs, strea
</span>      case Failure(e) =>
       jobScheduler.reportError("Error generating jobs for time " + time, e)
   }
   eventLoop.post(DoCheckpoint(time, clearCheckpointDataLater = false))
 }
```

其中，sumitJobSet会创建固定数量的后台线程（具体由“spark.streaming.concurrentJobs”指定），去处理Job Set中的Job. 具体实现逻辑为：

```
def submitJobSet(jobSet: JobSet) {
  if (jobSet.jobs.isEmpty) {
    logInfo("No jobs added for time " + jobSet.time)
  } else {
    listenerBus.post(StreamingListenerBatchSubmitted(jobSet.toBatchInfo))
    jobSets.put(jobSet.time, jobSet)
    jobSet.jobs.foreach(job => jobExecutor.execute(new JobHandler(job)))
    logInfo("Added jobs for time " + jobSet.time)
  }
```

```
10  }
```

其中JobHandler用于执行Job及处理Job执行结果信息。当Job执行完成时会产生JobCompleted事件. JobHandler的具体逻辑如下面代码所示：

+ View Code

当Job执行完成时，向eventLoop发送JobCompleted事件。EventLoop事件处理器接到JobCompleted事件后将调用handleJobCompletion 来处理Job完成事件。handleJobCompletion使用Job执行信息创建StreamingListenerBatchCompleted事件并通过StreamingListenerBus向监听器发送。实现如下：

```
1  private def handleJobCompletion(job: Job, completedTime: Long) {
2    val jobSet = jobSets.get(job.time)
3    jobSet.handleJobCompletion(job)
4    job.setEndTime(completedTime)
5    listenerBus.post(StreamingListenerOutputOperationCompleted(job.toOutputOperationInfo)
6    logInfo("Finished job " + job.id + " from job set of time " + jobSet.time)
7    if (jobSet.hasCompleted) {
8      jobSets.remove(jobSet.time)
9      jobGenerator.onBatchCompletion(jobSet.time)
10     logInfo("Total delay: %.3f s for time %s (execution: %.3f s)".format(
11       jobSet.totalDelay / 1000.0, jobSet.time.toString,
12       jobSet.processingDelay / 1000.0
13     ))
14     listenerBus.post(StreamingListenerBatchCompleted(jobSet.toBatchInfo))
15   }
16   job.result match {
17     case Failure(e) =>
18       reportError("Error running job " + job, e)
19     case _ =>
20   }
21 }
```

### 3.3.2、BatchCompleted事件处理过程

StreamingListenerBus将事件转交给具体的StreamingListener，因此BatchCompleted将交由RateController进行处理。RateController接到BatchCompleted事件后将调用onBatchCompleted对事件进行处理。

```
1  override def onBatchCompleted(batchCompleted: StreamingListenerBatchCompleted) {
2    val elements = batchCompleted.batchInfo.streamIdToInputInfo
3
4    for {
5      processingEnd <- batchCompleted.batchInfo.processingEndTime
6      workDelay <- batchCompleted.batchInfo.processingDelay
7      waitDelay <- batchCompleted.batchInfo.schedulingDelay
8      elems <- elements.get(streamUID).map(_.numRecords)
9    } computeAndPublish(processingEnd, elems, workDelay, waitDelay)
10 }
```

onBatchCompleted会从完成的任务中抽取任务的执行延迟和调度延迟，然后用这两个参数用RateEstimator（目前存在唯一实现PIDRateEstimator，proportional-integral-derivative (PID) controller， PID控制器）估算出新的rate并发布。代码如下：

```
  /**
   * Compute the new rate limit and publish it asynchronously.
   */
  private def computeAndPublish(time: Long, elems: Long, workDelay: Long, waitDelay:
Long): Unit =
    Future[Unit] {
      val newRate = rateEstimator.compute(time, elems, workDelay, waitDelay)
      newRate.foreach { s =>
        rateLimit.set(s.toLong)
        publish(getLatestRate())
      }
    }
```

其中publish（）由RateController的子类ReceiverRateController来定义。具体逻辑如下（ReceiverInputDStream中定义）：

```
/**
 * A RateController that sends the new rate to receivers, via the receiver tracker.
 */
private[streaming] class ReceiverRateController(id: Int, estimator: RateEstimator)
    extends RateController(id, estimator) {
  override def publish(rate: Long): Unit =
    ssc.scheduler.receiverTracker.sendRateUpdate(id, rate)
}
```

publish的功能为新生成的rate 借助ReceiverTracker进行转发。ReceiverTracker将rate包装成UpdateReceiverRateLimit事交ReceiverTrackerEndpoint

```
/** Update a receiver's maximum ingestion rate */
def sendRateUpdate(streamUID: Int, newRate: Long): Unit = synchronized {
  if (isTrackerStarted) {
    endpoint.send(UpdateReceiverRateLimit(streamUID, newRate))
  }
}
```

ReceiverTrackerEndpoint接到消息后，其将会从receiverTrackingInfos列表中获取Receiver注册时使用的endpoint(实为ReceiverSupervisorImpl)，再将rate包装成UpdateLimit发送至endpoint.其接到信息后，使用updateRate更新BlockGenerators(RateLimiter子类),来计算出一个固定的令牌间隔。

```
+ View Code
```

其中RateLimiter的updateRate实现如下：

```
/**
  * Set the rate limit to `newRate`. The new rate will not exceed the maximum rate confi
  * {{{spark.streaming.receiver.maxRate}}}, even if `newRate` is higher than that.
  *
  * @param newRate A new rate in events per second. It has no effect if it's 0 or negati
  */
 private[receiver] def updateRate(newRate: Long): Unit =
   if (newRate > 0) {
     if (maxRateLimit > 0) {
       rateLimiter.setRate(newRate.min(maxRateLimit))
     } else {
       rateLimiter.setRate(newRate)
     }
   }
```

setRate的实现 如下：

```
public final void setRate(double permitsPerSecond) {
    Preconditions.checkArgument(permitsPerSecond > 0.0
        && !Double.isNaN(permitsPerSecond), "rate must be positive");
    synchronized (mutex) {
      resync(readSafeMicros());
      double stableIntervalMicros = TimeUnit.SECONDS.toMicros(1L) / permitsPerSecond;  /
      this.stableIntervalMicros = stableIntervalMicros;
      doSetRate(permitsPerSecond, stableIntervalMicros);
    }
  }
```

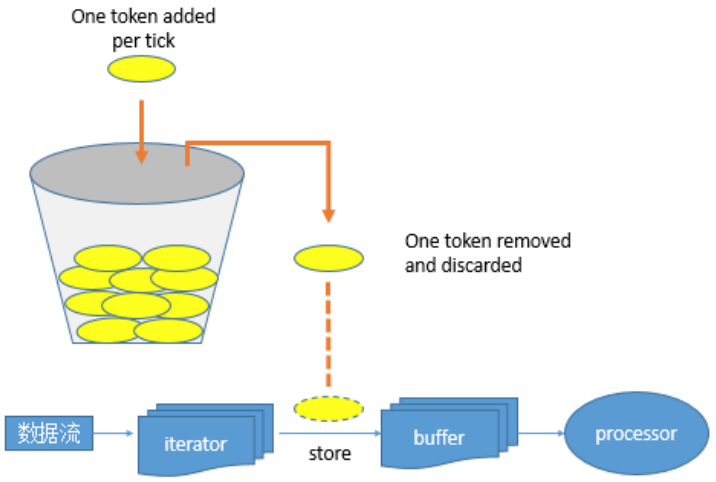
到此，backpressure反压机制调整rate结束。

4. 流量控制点

当Receiver开始接收数据时，会通过supervisor.pushSingle()方法将接收的数据存入currentBuffer等待BlockGenerator定时将数据取走，包装成block. 在将数据存放入currentBuffer之时，要获取许可（令牌）。如果获取到许可就可以将数据存入buffer, 否则将被阻塞，进而阻塞Receiver从数据源拉取数据。

```
  /**
   * Push a single data item into the buffer.
   */
  def addData(data: Any): Unit = {
    if (state == Active) {
     <span style="color: #ff0000;"> waitToPush()  //获取令牌
</span>      synchronized {
        if (state == Active) {
          currentBuffer += data
        } else {
          throw new SparkException(
            "Cannot add data as BlockGenerator has not been started or has been stopped"
        }
      }
    } else {
      throw new SparkException(
        "Cannot add data as BlockGenerator has not been started or has been stopped")
    }
  }
```

其令牌投放采用令牌桶机制进行， 原理如下图所示：




令牌桶机制： 大小固定的令牌桶可自行以恒定的速率源源不断地产生令牌。如果令牌不被消耗，或者被消耗的速度小于产生的速度，令牌就会不断地增多，直到把桶填满。后面再产生的令牌就会从桶中溢出。最后桶中可以保存的最大令牌数永远不会超过桶的大小。当进行某操作时需要令牌时会从令牌桶中取出相应的令牌数，如果获取到则继续操作，否则阻塞。用完之后不用放回。

Streaming 数据流被Receiver接收后，按行解析后存入iterator中。然后逐个存入Buffer，在存入buffer时会先获取token，如果没有token存在，则阻塞；如果获取到则将数据存入buffer. 然后等价后续生成block操作。

转载请注明：http://www.cnblogs.com/barrenlake/p/5349949.html

分类: spark

标签: Spark Streaming Backpressure, Streaming 反压机制

1 0

(请您对文章做出评价)

« 上一篇：Spark Streaming 数据接收过程

posted on 2016-04-03 15:39 barrenlake 阅读(129) 评论(1) 编辑 收藏

评论：

#1楼 2016-05-05 14:20 | bruce_xu
写的很棒，是我看到的写的最好的

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。



最新IT新闻：
· 84.1万元：特斯拉在中国开启入门款Model X 75D预约
· 售价399欧元 Ubuntu版魅族MX6渲染图曝光
· Windows 10 Build 10586.456累积更新日志曝光
· 臭氧层保护公约制定近30年后终见成效：南极臭氧空洞开始修复
· 英国脱欧影响智能手机价格 一加表态要涨价
» 更多新闻...



最新知识库文章：
· 编程同写作，写代码只是在码字
· 遇见程序员男友
· 设计师的视觉设计五项修炼
· 我听到过的最精彩的一个软件纠错故事
· 如何避免软件工程中最昂贵错误的发生
» 更多知识库文章...