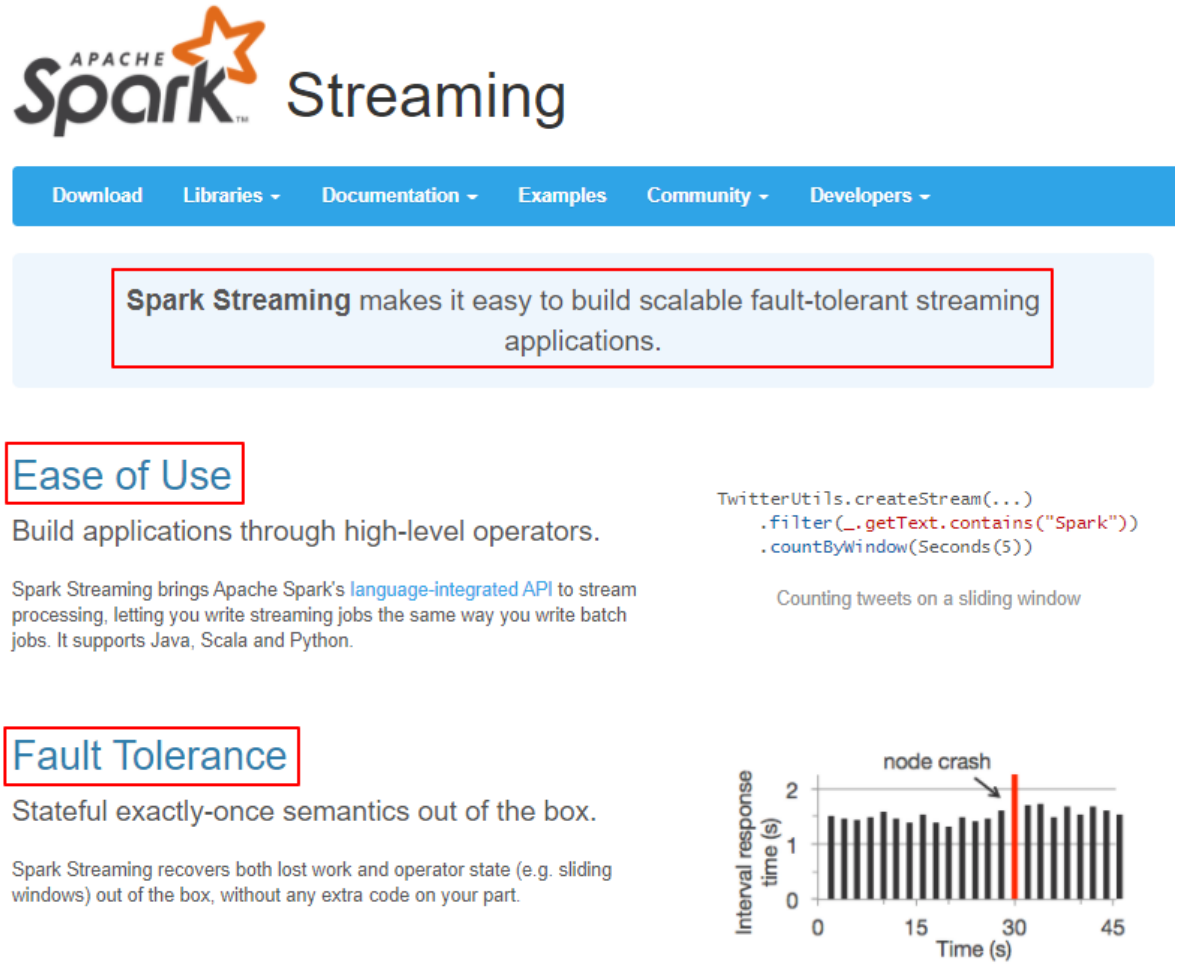
大数据高速计算引擎Spark（下）【讲师：回灯】

第三部分 Spark Streaming

随着大数据技术的不断发展，人们对于大数据的实时性处理要求也在不断提高，传统的 MapReduce 等批处理框架在某些特定领域，例如实时用户推荐、用户行为分析这些应用场景上逐渐不能满足人们对实时性的需求，因此诞生了一批如 S3、Samza、Storm、Flink等流式分析、实时计算框架。

Spark 由于其内部优秀的调度机制、快速的分布式计算能力，能够以极快的速度进行迭代计算。正是由于具有这样的优势，Spark 能够在某些程度上进行实时处理，Spark Streaming 正是构建在此之上的流式框架。

第1节 Spark Streaming概述



1.1 什么是Spark Streaming

Spark Streaming类似于Apache Storm（来一条数据处理一条，延迟低，响应快，低吞吐量），用于流式数据的处理；

Spark Streaming具有有高吞吐量和容错能力强等特点；

Spark Streaming支持的数据输入源很多，例如：Kafka（最重要的数据源）、Flume、Twitter 和 TCP 套接字等；

数据输入后可用高度抽象API，如：map、reduce、join、window等进行运算；

处理结果能保存在很多地方，如HDFS、数据库等；

Spark Streaming 能与 MLlib 以及 Graphx 融合。

Spark Streaming 与 Spark 基于 RDD 的概念比较类似；

Spark Streaming使用离散化流（Discretized Stream）作为抽象表示，称为DStream。

DStream是随时间推移而收到的数据的序列。在内部，每个时间区间收到的数据都作为 RDD 存在，DStream 是由这些 RDD 所组成的序列。

DStream 可以从各种输入源创建，比如 Flume、Kafka 或者 HDFS。创建出来的 DStream 支持两种操作：

- 转化操作，会生成一个新的DStream
- 输出操作(output operation)，把数据写入外部系统中

DStream 提供了许多与 RDD 所支持的操作相类似的操作支持，还增加了与时间相关的新操作，比如滑动窗口。

1.2 Spark Streaming架构

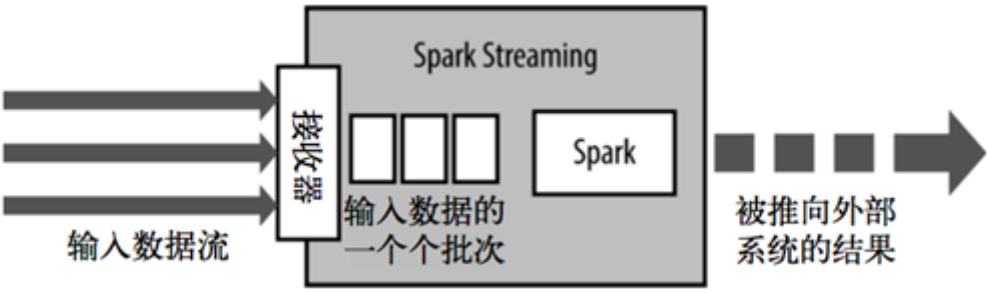
Spark Streaming使用 **mini-batch** 的架构，**把流式计算当作一系列连续的小规模批处理来对待。**

Spark Streaming从各种输入源中读取数据，并把数据分组为小的批次。新的批次按均匀的时间间隔创建出来。

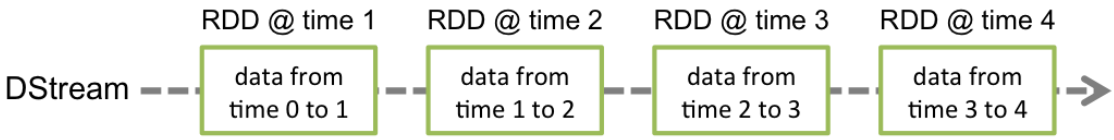
在每个时间区间开始的时候，一个新的批次就创建出来，在该区间内收到的数据都会被添加到这个批次中。在时间区间结束时，批次停止增长。

时间区间的大小是由批次间隔这个参数决定的。批次间隔一般设在500毫秒到几秒之间，由开发者配置。

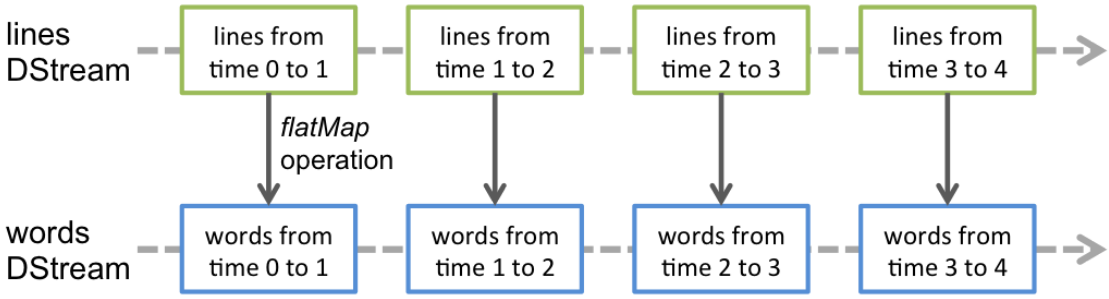
每个输入批次都形成一个RDD，以 Spark 作业的方式处理并生成其他的 RDD。 处理的结果可以以批处理的方式传给外部系统。



Spark Streaming的编程抽象是离散化流，也就是DStream。它是一个 RDD 序列，每个RDD代表数据流中一个时间片内的数据。



应用于 DStream 上的转换操作都会转换为底层RDD上的操作。如对行 DStream 中的每个RDD应用flatMap操作以生成单词 DStream 的RDD。



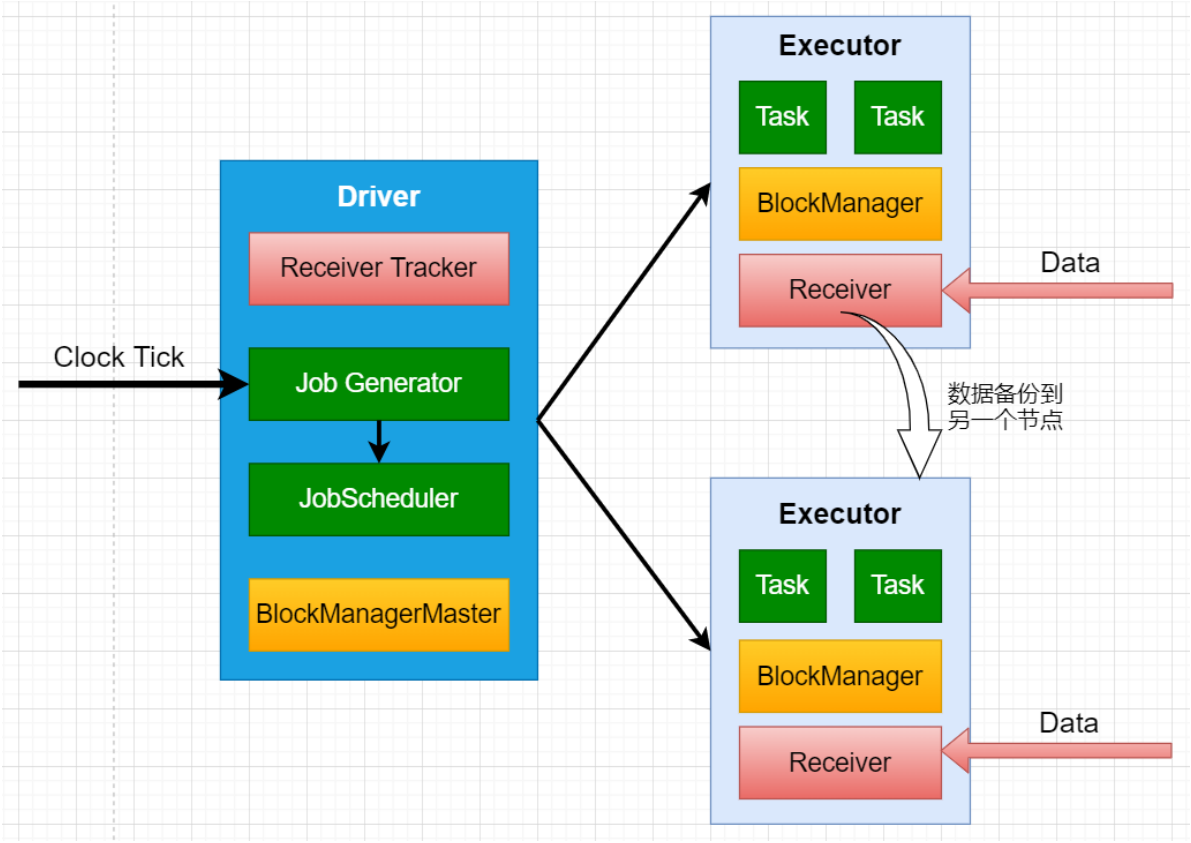
这些底层的RDD转换是由Spark引擎完成的。DStream操作隐藏了大部分这些细节，为开发人员提供了更高级别的API以方便使用。

```scala
 * A DStream internally is characterized by a few basic properties:
 *  - A list of other DStreams that the DStream depends on
 *  - A time interval at which the DStream generates an RDD
 *  - A function that is used to generate an RDD after each time interval
 */

abstract class DStream[T: ClassTag] (
    @transient private[streaming] var ssc: StreamingContext
  ) extends Serializable with Logging {

  validateAtInit()

  /.../

  /** Time interval after which the DStream generates an RDD */
  def slideDuration: Duration

  /** List of parent DStreams on which this DStream depends on */
  def dependencies: List[DStream[_]]

  /** Method that generates an RDD for the given time */
  def compute(validTime: Time): Option[RDD[T]]

  // ========================================================================
  // Methods and fields available on all DStreams
  // ========================================================================

  // RDDs generated, marked as private[streaming] so that testsuites can access it
  @transient
  private[streaming] var generatedRDDs = new HashMap[Time, RDD[T]]()
```

Spark Streaming为每个输入源启动对应的接收器。接收器运行在Executor中，从输入源收集数据并保存为 RDD

默认情况下接收到的数据后会复制到另一个Executor中，进行容错；

Driver 中的 StreamingContext 会周期性地运行 Spark 作业来处理这些数据。



SparkStreaming运行流程：

1、客户端提交Spark Streaming作业后启动Driver，Driver启动Receiver，Receiver接收数据源的数据

2、每个作业包含多个Executor，每个Executor以线程的方式运行task，Spark Streaming至少包含一个receiver task（一般情况下）

3、Receiver接收数据后生成Block，并把BlockId汇报给Driver，然后备份到另外一个 Executor 上

4、ReceiverTracker维护 Reciver 汇报的BlockId

5、Driver定时启动JobGenerator，根据Dstream的关系生成逻辑RDD，然后创建Jobset，交给JobScheduler

6、JobScheduler负责调度Jobset，交给DAGScheduler，DAGScheduler根据逻辑RDD，生成相应的Stages，每个stage包含一到多个Task，将TaskSet提交给TaskSchedule

7、TaskScheduler负责把 Task 调度到 Executor 上，并维护 Task 的运行状态

1.3 Spark Streaming 优缺点

与传统流式框架相比，Spark Streaming 最大的不同点在于它对待数据是粗粒度的处理方式，即一次处理一小批数据，而其他框架往往采用细粒度的处理模式，即依次处理一条数据。Spark Streaming 这样的设计实现既为其带来了显而易见的优点，又引入了不可避免的缺点。

优点

- Spark Streaming 内部的实现和调度方式高度依赖 Spark 的 DAG 调度器和 RDD，这就决定了 Spark Streaming 的设计初衷必须是粗粒度方式的。同时，由于 Spark 内部调度器足够快速和高效，可以快速地处理小批量数据，这就获得准实时的特性
- Spark Streaming 的粗粒度执行方式使其确保 ”处理且仅处理一次” 的特性（EOS），同时也可以更方便地实现容错恢复机制
- 由于 Spark Streaming 的 DStream 本质是 RDD 在流式数据上的抽象，因此基于 RDD 的各种操作也有相应的基于 DStream 的版本，这样就大大降低了用户对于新框架的学习成本，在了解 Spark 的情况下用户将很容易使用 Spark Streaming
- 由于 DStream 是在 RDD 上的抽象，那么也就更容易与 RDD 进行交互操作，在需要将流式数据和批处理数据结合进行分析的情况下，将会变得非常方便

缺点

- Spark Streaming 的粗粒度处理方式也造成了不可避免的延迟。在细粒度处理方式下，理想情况下每一条记录都会被实时处理，而在 Spark Streaming 中，数据需要汇总到一定的量后再一次性处理，这就增加了数据处理的延迟，这种延迟是由框架的设计引入的，并不是由网络或其他情况造成的

1.4 Structured Streaming

Spark Streaming计算逻辑是把数据按时间划分为DStream，存在以下问题：

- 框架自身只能根据 Batch Time 单元进行数据处理，很难处理基于event time（即时间戳）的数据，很难处理延迟，乱序的数据
- 流式和批量处理的 API 不完全一致，两种使用场景中，程序代码还是需要一定的转换
- 端到端的数据容错保障逻辑需要用户自己构建，难以处理增量更新和持久化存储等一致性问题

基于以上问题，提出了下一代 Structure Streaming 。将数据源映射为一张无界长度的表，通过表的计算，输出结果映射为另一张表。

以结构化的方式去操作流式数据，简化了实时计算过程，同时还复用了 Catalyst 引擎来优化SQL操作。此外还能支持增量计算和基于event time的计算。

第2节 DStream基础数据源

基础数据源包括：文件数据流、socket数据流、RDD队列流；这些数据源主要用于测试。

引入依赖：

```
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_2.12</artifactId>
            <version>${spark.version}</version>
        </dependency>
```

2.1 文件数据流

文件数据流：通过 textFileStream(directory) 方法进行读取 HDFS 兼容的文件系统文件

```
def textFileStream(directory: String): DStream[String] = withNamedScope( name = "text file stream") {
  fileStream[LongWritable, Text, TextInputFormat](directory).map(_._2.toString)
}
```

Spark Streaming 将会监控 directory 目录，并不断处理移动进来的文件

- 不支持嵌套目录
- 文件需要有相同的数据格式
- 文件进入 directory 的方式需要通过移动或者重命名来实现
- 一旦文件移动进目录，则不能再修改，即便修改了也不会读取新数据
- 文件流不需要接收器（receiver），不需要单独分配CPU核

```
import org.apache.log4j.{Level, Logger}
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}

object FileDStream {
  def main(args: Array[String]): Unit = {
    Logger.getLogger("org").setLevel(Level.ERROR)
    val conf = new
SparkConf().setAppName(this.getClass.getCanonicalName).setMaster(
"local[*]")
    // 创建StreamingContext
    // StreamingContext是所有流功能函数的主要访问点，这里使用多个执行线程和
2秒的批次间隔来创建本地的StreamingContext
    // 时间间隔为2秒,即2秒一个批次
    val ssc = new StreamingContext(conf, Seconds(5))

    // 这里采用本地文件，也可以采用HDFS文件
    val lines = ssc.textFileStream("data/log/")
    val words = lines.flatMap(_.split("\\s+"))
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)

    // 打印单位时间所获得的计数值
    wordCounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
```

2.2 Socket数据流

Spark Streaming可以通过Socket端口监听并接收数据，然后进行相应处理；

新开一个命令窗口，启动 nc 程序：

```
nc -lk 9999
# yum install nc
```

随后可以在nc窗口中随意输入一些单词，监听窗口会自动获得单词数据流信息，在监听窗口每隔x秒就会打印出词频统计信息，可以在屏幕上出现结果。

备注：使用local[*]，可能存在问题。

如果给虚拟机配置的cpu数为1，使用local[*]也只会启动一个线程，该线程用于receiver task，此时没有资源处理接收达到的数据。

【现象：程序正常执行，不会打印时间戳，屏幕上也不会有其他有效信息】

```
297     def socketTextStream(
298         hostname: String,
299         port: Int,
300         storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2
301       ): ReceiverInputDStream[String] = withNamedScope( name = "socket text stream") {
302       socketStream[String](hostname, port, SocketReceiver.bytesToLines, storageLevel)
303     }
```

注意：DStream的 StorageLevel 是 MEMORY_AND_DISK_SER_2；

```
package cn.lagou.streaming

import org.apache.log4j.{Level, Logger}
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}

object SocketDStream {
  def main(args: Array[String]): Unit = {
    Logger.getLogger("org").setLevel(Level.ERROR)
    val conf = new
SparkConf().setAppName(this.getClass.getCanonicalName).setMaster(
"local[*]")

    // 创建StreamingContext
    val ssc = new StreamingContext(conf, Seconds(1))

    val lines = ssc.socketTextStream("linux122", 9999)
    val words = lines.flatMap(_.split("\\s+"))
    val wordCounts = words.map(x => (x.trim, 1)).reduceByKey(_ +
_)

    // 打印单位时间所获得的计数值
    wordCounts.print()

    ssc.start()
    ssc.awaitTermination()
  }
}
```

SocketServer程序（单线程），监听本机指定端口，与socket连接后可发送信息：

```
package cn.lagou.streaming

import java.io.PrintWriter
import java.net.{ServerSocket, Socket}

import scala.util.Random

object SocketLikeNC {
  def main(args: Array[String]): Unit = {
    val words: Array[String] = "Hello World Hello Hadoop Hello
spark kafka hive zookeeper hbase flume sqoop".split("\\s+")
    val n: Int = words.length
    val port: Int = 9999
    val random: Random = scala.util.Random

    val server = new ServerSocket(port)
    val socket: Socket = server.accept()
    println("成功连接到本地主机: " + socket.getInetAddress)
    while (true) {
      val out = new PrintWriter(socket.getOutputStream)
      out.println(words(random.nextInt(n)) + " "+
words(random.nextInt(n)))
      out.flush()
      Thread.sleep(100)
    }
  }
}
```

SocketServer程序（多线程）

```
package cn.lagou.streaming

import java.net.ServerSocket

object SocketServer {
  def main(args: Array[String]): Unit = {
    val server = new ServerSocket(9999)
    println(s"Socket Server 已启动:
${server.getInetAddress}:${server.getLocalPort}")
```

```
    while (true) {
        val socket = server.accept()
        println("成功连接到本地主机: " + socket.getInetAddress)
        new ServerThread(socket).start()
    }
  }
}
```

```
package cn.lagou.streaming

import java.io.DataOutputStream
import java.net.Socket

class ServerThread(sock: Socket) extends Thread {
  val words = "hello world hello spark hello word hello java
hello hadoop hello kafka"
    .split("\\s+")
  val length = words.length

  override def run(): Unit = {
    val out = new DataOutputStream(sock.getOutputStream)
    val random = scala.util.Random

    while (true) {
      val (wordx, wordy) = (words(random.nextInt(length)),
words(random.nextInt(length)))
      out.writeUTF(s"$wordx $wordy")
      Thread.sleep(100)
    }
  }
}
```

2.3 RDD队列流

调试Spark Streaming应用程序的时候，可使用 `streamingContext.queueStream(queueOfRDD)` 创建基于RDD队列的DStream；

```
456     def queueStream[T: ClassTag](
457         queue: Queue[RDD[T]],
458         oneAtATime: Boolean = true
459       ): InputDStream[T] = {
460       queueStream(queue, oneAtATime, sc.makeRDD(Seq.empty[T], numSlices = 1))
461     }
```

备注：

- oneAtATime：缺省为true，一次处理一个RDD；设为false，一次处理全部RDD
- RDD队列流可以使用local[1]
- 涉及到同时出队和入队操作，所以要做同步

每秒创建一个RDD（RDD存放1-100的整数），Streaming每隔1秒就对数据进行处理，计算RDD中数据除10取余的个数。

```
package cn.lagou.streaming

import org.apache.log4j.{Level, Logger}
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.{Seconds, StreamingContext}

import scala.collection.mutable.Queue

object RDDQueueDStream {
  def main(args: Array[String]) {
    Logger.getLogger("org").setLevel(Level.WARN)
    val sparkConf = new
SparkConf().setAppName(this.getClass.getCanonicalName).setMaster(
"local[2]")
    // 每隔1秒对数据进行处理
    val ssc = new StreamingContext(sparkConf, Seconds(1))

    val rddQueue = new Queue[RDD[Int]]()
    val queueStream = ssc.queueStream(rddQueue)
    val mappedStream = queueStream.map(r => (r % 10, 1))
    val reducedStream = mappedStream.reduceByKey(_ + _)

    reducedStream.print()
    ssc.start()

    // 每秒产生一个RDD
    for (i <- 1 to 5){
      rddQueue.synchronized {
```

```
        val range = (1 to 100).map(_*i)
        rddQueue += ssc.sparkContext.makeRDD(range, 2)
      }
      Thread.sleep(2000)
    }
    ssc.stop()
  }
}
```

第3节 DStream转换操作

DStream上的操作与RDD的类似，分为 Transformations（转换）和 Output Operations（输出）两种，此外转换操作中还有一些比较特殊的方法，如：updateStateByKey、transform 以及各种 Window 相关的操作。

Transformation	Meaning
map(func)	将源DStream中的每个元素通过一个函数func从而得到新的DStreams
flatMap(func)	和map类似，但是每个输入的项可以被映射为0或更多项
filter(func)	选择源DStream中函数func判为true的记录作为新DStreams
repartition(numPartitions)	通过创建更多或者更少的partition来改变此DStream的并行级别
union(otherStream)	联合源DStreams和其他DStreams来得到新DStream
count()	统计源DStreams中每个RDD所含元素的个数得到单元素RDD的新DStreams
reduce(func)	通过函数func(两个参数一个输出)来整合源DStreams中每个RDD元素得到单元素RDD的DStreams。这个函数需要关联从而可以被并行计算
countByValue()	对于DStreams中元素类型为K调用此函数，得到包含(K,Long)对的新DStream，其中Long值表明相应的K在源DStream中每个RDD出现的频率
reduceByKey(func, [numTasks])	对(K,V)对的DStream调用此函数，返回同样(K,V)的新DStream，新DStream中的对应V为使用reduce函数整合而来。默认情况下，这个操作使用Spark默认数量的并行任务（本地模式为2，集群模式中的数量取决于配置参数spark.default.parallelism）。也可以传入可选的参数numTasks来设置不同数量的任务
join(otherStream, [numTasks])	两DStream分别为(K,V)和(K,W)对，返回(K,(V,W))对的新DStream
cogroup(otherStream, [numTasks])	两DStream分别为(K,V)和(K,W)对，返回(K,(Seq[V],Seq[W])对新DStreams
transform(func)	将RDD到RDD映射的函数func作用于源DStream中每个RDD上得到新DStream。这个可用于在DStream的RDD上做任意操作

Transformation	Meaning
updateStateByKey(func)	得到”状态”DStream，其中每个key状态的更新是通过将给定函数用于此key的上一个状态和新值而得到。这个可用于保存每个key值的任意状态数据

备注：

- 在DStream与RDD上的转换操作非常类似（无状态的操作）
- DStream有自己特殊的操作（窗口操作、追踪状态变化操作）
- 在DStream上的转换操作比RDD上的转换操作少

DStream 的转化操作可以分为 无状态(stateless) 和 有状态(stateful) 两种：

- 无状态转化操作。每个批次的处理不依赖于之前批次的数据。常见的 RDD 转化操作，例如 map、filter、reduceByKey 等
- 有状态转化操作。需要使用之前批次的数据 或者是 中间结果来计算当前批次的数据。有状态转化操作包括：基于滑动窗口的转化操作 或 追踪状态变化的转化操作

3.1 无状态转换

无状态转化操作就是把简单的 RDD 转化操作应用到每个批次上，也就是转化 DStream 中的每一个 RDD。

常见的无状态转换包括：map、flatMap、filter、repartition、reduceByKey、groupByKey；直接作用在DStream上

重要的转换操作：**transform**。通过对源DStream的每个RDD应用RDD-to-RDD函数，创建一个新的DStream。支持在新的DStream中做任何RDD操作。

```
/**
 * Return a new DStream in which each RDD is generated by applying a function
 * on each RDD of 'this' DStream.
 */
def transform[U: ClassTag](transformFunc: RDD[T] => RDD[U]): DStream[U] = ssc.withScope {
  // because the DStream is reachable from the outer object here, and because
  // DStreams can't be serialized with closures, we can't proactively check
  // it for serializability and so we pass the optional false to SparkContext.clean
  val cleanedF = context.sparkContext.clean(transformFunc, checkSerializable = false)
  transform((r: RDD[T], _: Time) => cleanedF(r))
}
```

这是一个功能强大的函数，它可以允许开发者直接操作其内部的RDD。也就是说开发者，可以提供任意一个RDD到RDD的函数，这个函数在数据流每个批次中都被调用，生成一个新的流。

示例：黑名单过滤

```
假设：arr1为黑名单数据(自定义)，true表示数据生效，需要被过滤掉；false表示数据未生效
val arr1 = Array(("spark", true), ("scala", false))

假设：流式数据格式为"time word"，需要根据黑名单中的数据对流式数据执行过滤操作。如"2 spark"要被过滤掉
1 hadoop
2 spark
3 scala
4 java
5 hive

结果："2 spark" 被过滤
```

方法一：使用外连接

```
package cn.lagou.streaming

import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.dstream.ConstantInputDStream
import org.apache.spark.streaming.{Seconds, StreamingContext}

object BlackListFilter1 {
  def main(args: Array[String]) {
    // 初始化
    val conf = new
SparkConf().setAppName(this.getClass.getCanonicalName).setMaster(
"local[2]")
    val ssc = new StreamingContext(conf, Seconds(10))
    ssc.sparkContext.setLogLevel("WARN")

    // 黑名单数据
    val blackList = Array(("spark", true), ("scala", true))
    val blackListRDD = ssc.sparkContext.makeRDD(blackList)
```

```
    // 生成测试DStream。使用ConstantInputDStream
    val strArray: Array[String] = "spark java scala hadoop kafka
hive hbase zookeeper"
      .split("\\s+")
      .zipWithIndex
      .map { case (word, idx) => s"$idx $word" }
    val rdd = ssc.sparkContext.makeRDD(strArray)
    val clickStream = new ConstantInputDStream(ssc, rdd)

    // 流式数据的处理
    val clickStreamFormatted = clickStream.map(value =>
(value.split(" ")(1), value))
    clickStreamFormatted.transform(clickRDD => {
      // 通过leftOuterJoin操作既保留了左侧RDD的所有内容，又获得了内容是否在
黑名单中
      val joinedBlackListRDD: RDD[(String, (String,
Option[Boolean]))] = clickRDD.leftOuterJoin(blackListRDD)

      joinedBlackListRDD.filter { case (word, (streamingLine,
flag)) =>
        if (flag.getOrElse(false)) false
        else true
      }.map { case (word, (streamingLine, flag)) => streamingLine
}
    }).print()

    // 启动流式作业
    ssc.start()
    ssc.awaitTermination()
  }
}
```

方法二：使用SQL

```
package cn.lagou.streaming

import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.{DataFrame, SparkSession}
import org.apache.spark.streaming.dstream.ConstantInputDStream
import org.apache.spark.streaming.{Seconds, StreamingContext}

object BlackListFilter2 {
```

```scala
  def main(args: Array[String]) {
    // 初始化
    val conf = new
SparkConf().setAppName(this.getClass.getCanonicalName).setMaster(
"local[2]")
    val ssc = new StreamingContext(conf, Seconds(10))
    ssc.sparkContext.setLogLevel("WARN")

    // 黑名单数据
    val blackList = Array(("spark", true), ("scala", true))
    val blackListRDD = ssc.sparkContext.makeRDD(blackList)

    // 生成测试DStream。使用ConstantInputDStream
    val strArray: Array[String] = "spark java scala hadoop kafka
hive hbase zookeeper"
      .split("\\s+")
      .zipWithIndex
      .map { case (word, idx) => s"$idx $word" }
    val rdd = ssc.sparkContext.makeRDD(strArray)
    val clickStream = new ConstantInputDStream(ssc, rdd)

    // 流式数据的处理
    val clickStreamFormatted = clickStream.map(value =>
(value.split(" ")(1), value))
    clickStreamFormatted.transform{clickRDD =>
      val spark = SparkSession
        .builder()
        .config(rdd.sparkContext.getConf)
        .getOrCreate()

      import spark.implicits._
      val clickDF: DataFrame = clickRDD.toDF("word", "line")
      val blackDF: DataFrame = blackListRDD.toDF("word", "flag")
      clickDF.join(blackDF, Seq("word"), "left")
        .filter("flag is null or flag = false")
        .select("line")
        .rdd
    }.print()

    // 启动流式作业
    ssc.start()
    ssc.awaitTermination()
  }
}
```

方法三：直接过滤

```
package cn.lagou.streaming

import org.apache.spark.SparkConf
import org.apache.spark.broadcast.Broadcast
import org.apache.spark.streaming.dstream.ConstantInputDStream
import org.apache.spark.streaming.{Seconds, StreamingContext}

object BlackListFilter3 {
  def main(args: Array[String]) {
    // 初始化
    val conf = new
SparkConf().setAppName(this.getClass.getCanonicalName).setMaster(
"local[2]")
    val ssc = new StreamingContext(conf, Seconds(10))
    ssc.sparkContext.setLogLevel("WARN")

    // 黑名单数据
    val blackList = Array(("spark", true), ("scala", true))
    val blackListBC: Broadcast[Array[String]] =
ssc.sparkContext.broadcast(blackList.filter(_._2).map(_._1))

    // 生成测试DStream。使用ConstantInputDStream
    val strArray: Array[String] = "spark java scala hadoop kafka
hive hbase zookeeper"
      .split("\\s+")
      .zipWithIndex
      .map { case (word, idx) => s"$idx $word" }
    val rdd = ssc.sparkContext.makeRDD(strArray)
    val clickStream = new ConstantInputDStream(ssc, rdd)

    // 流式数据的处理
    clickStream.map(value => (value.split(" ")(1), value))
        .filter{case (word, _) =>
!blackListBC.value.contains(word)}
        .map(_._2)
        .print()

    // 启动流式作业
    ssc.start()
    ssc.awaitTermination()
  }
```

```
}
```

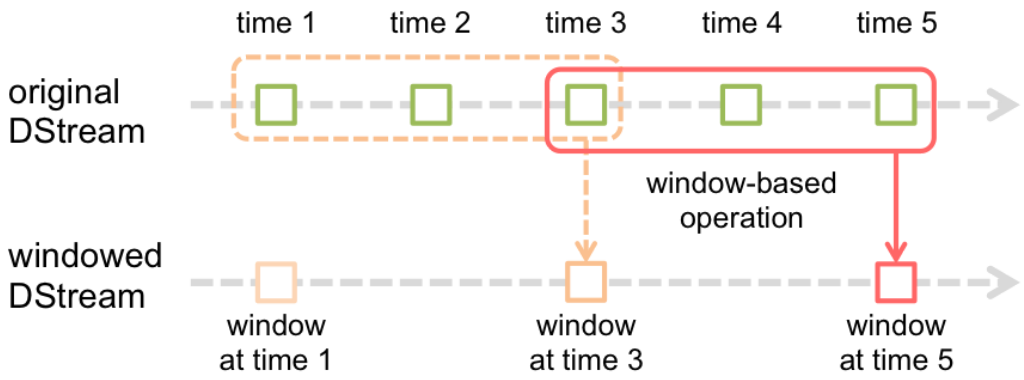
3.2 有状态转换

有状态的转换主要有两种：窗口操作、状态跟踪操作

3.2.1 窗口操作

Window Operations可以设置**窗口大小**和**滑动窗口间隔**来动态的获取当前Streaming的状态。

基于窗口的操作会在一个比 StreamingContext 的 batchDuration（批次间隔）更长的时间范围内，通过整合多个批次的结果，计算出整个窗口的结果。



基于窗口的操作需要两个参数：

- 窗口长度(windowDuration)。控制每次计算最近的多少个批次的数据
- 滑动间隔(slideDuration)。用来控制对新的 DStream 进行计算的间隔

两者都必须是 StreamingContext 中批次间隔(batchDuration)的整数倍。

每秒发送1个数字：

```
package cn.lagou.streaming

import java.io.PrintWriter
import java.net.{ServerSocket, Socket}

object SocketLikeNCWithWindow {
  def main(args: Array[String]): Unit = {
    val port = 1521
    val ss = new ServerSocket(port)
    val socket: Socket = ss.accept()
```

```
    println("connect to host : " + socket.getInetAddress)
    var i = 0
    // 每秒发送1个数

    while(true) {
      i += 1
      val out = new PrintWriter(socket.getOutputStream)
      out.println(i)
      out.flush()
      Thread.sleep(1000)
    }
  }
}
```

案例一：

- 观察窗口的数据；
- 观察 batchDuration、windowDuration、slideDuration 三者之间的关系；
- 使用窗口相关的操作；

```
package cn.lagou.streaming

import org.apache.spark.SparkConf
import org.apache.spark.streaming.dstream.{DStream,
ReceiverInputDStream}
import org.apache.spark.streaming.{Seconds, StreamingContext}

object WindowDemo {
  def main(args: Array[String]): Unit = {
    val conf = new
SparkConf().setMaster("local[*]").setAppName(this.getClass.getCan
onicalName)

    // 每 5s 生成一个RDD（mini-batch）
    val ssc = new StreamingContext(conf, Seconds(5))
    ssc.sparkContext.setLogLevel("error")

    val lines: ReceiverInputDStream[String] =
ssc.socketTextStream("localhost", 1521)
    lines.foreachRDD{ (rdd, time) =>
      println(s"rdd = ${rdd.id}; time = $time")
      rdd.foreach(value => println(value))
    }
```

```
    // 20s 窗口长度(ds包含窗口长度范围内的数据)；10s 滑动间隔(多次时间处理一次数据)
    val res1: DStream[String] = lines.reduceByWindow(_ + " " + _, Seconds(20), Seconds(10))
    res1.print()

    val res2: DStream[String] = lines.window(Seconds(20), Seconds(10))
    res2.print()

    // 求窗口元素的和
    val res3: DStream[Int] = lines.map(_.toInt).reduceByWindow(_+_, Seconds(20), Seconds(10))
    res3.print()

    // 求窗口元素的和
    val res4 = res2.map(_.toInt).reduce(_+_)
    res4.print()

    ssc.start()
    ssc.awaitTermination()
  }
}
```

案例二：热点搜索词实时统计。每隔 10 秒，统计最近20秒的词出现的次数

```
package cn.lagou.streaming

import org.apache.spark.SparkConf
import org.apache.spark.streaming.dstream.{DStream, ReceiverInputDStream}
import org.apache.spark.streaming.{Seconds, StreamingContext}

object HotWordStats {
  def main(args: Array[String]): Unit = {
    val conf: SparkConf = new SparkConf()
      .setMaster("local[2]")
      .setAppName(this.getClass.getCanonicalName)
    val ssc = new StreamingContext(conf, Seconds(2))
    ssc.sparkContext.setLogLevel("ERROR")

    //设置检查点，检查点具有容错机制。生产环境中应设置到HDFS
```

```
    ssc.checkpoint("data/checkpoint/")
    val lines: ReceiverInputDStream[String] =
ssc.socketTextStream("localhost", 9999)
    val words: DStream[String] = lines.flatMap(_.split("\\s+"))
    val pairs: DStream[(String, Int)] = words.map(x => (x, 1))

    // 通过reduceByKeyAndWindow算子，每隔10秒统计最近20秒的词出现的次数
    // 后 3个参数：窗口时间长度、滑动窗口时间、分区
    val wordCounts1: DStream[(String, Int)] =
pairs.reduceByKeyAndWindow(
      (a: Int, b: Int) => a + b,
      Seconds(20),
      Seconds(10), 2)
    wordCounts1.print

    // 这里需要checkpoint的支持
    val wordCounts2: DStream[(String, Int)] =
pairs.reduceByKeyAndWindow(
      _ + _,
      _ - _,
      Seconds(20),
      Seconds(10), 2)
    wordCounts2.print

    ssc.start()
    ssc.awaitTermination()
  }
}
```

3.2.2 updateStateByKey（状态追踪操作）

UpdateStateByKey的主要功能：

- 为Streaming中每一个Key维护一份state状态，state类型可以是任意类型的，可以是自定义对象；更新函数也可以是自定义的
- 通过更新函数对该key的状态不断更新，对于每个新的batch而言，Spark Streaming会在使用updateStateByKey 的时候为已经存在的key进行state的状态更新
- 使用 updateStateByKey 时要开启 checkpoint 功能

```
398     def updateStateByKey[S: ClassTag](
399         updateFunc: (Seq[V], Option[S]) => Option[S]
400       ): DStream[(K, S)] = ssc.withScope {
401       updateStateByKey(updateFunc, defaultPartitioner())
402     }
```

流式程序启动后计算wordcount的累计值，将每个批次的结果保存到文件

```
package cn.lagou.streaming

import org.apache.spark.SparkConf
import org.apache.spark.streaming.dstream.{DStream,
ReceiverInputDStream}
import org.apache.spark.streaming.{Seconds, StreamingContext}

object StateTracker1 {
  def main(args: Array[String]) {
    val conf: SparkConf = new SparkConf().setMaster("local[2]")
      .setAppName(this.getClass.getCanonicalName)
    val ssc = new StreamingContext(conf, Seconds(5))
    ssc.sparkContext.setLogLevel("ERROR")

    ssc.checkpoint("data/checkpoint/")
    val lines: ReceiverInputDStream[String] =
ssc.socketTextStream("localhost", 9999)
    val words: DStream[String] = lines.flatMap(_.split("\\s+"))
    val wordDstream: DStream[(String, Int)] = words.map(x => (x,
1))

    // 定义状态更新函数
    // 函数常量定义，返回类型是Some(Int)，表示的含义是最新状态
    // 函数的功能是将当前时间间隔内产生的Key的value集合，加到上一个状态中，得
到最新状态
    val updateFunc = (currValues: Seq[Int], prevValueState:
Option[Int]) => {
      //通过Spark内部的reduceByKey按key规约，然后这里传入某key当前批次的
Seq，再计算当前批次的总和
      val currentCount = currValues.sum
      // 已累加的值
      val previousCount = prevValueState.getOrElse(0)
      Some(currentCount + previousCount)
    }
```

```
    val stateDstream: DStream[(String, Int)] =
wordDstream.updateStateByKey[Int](updateFunc)
    stateDstream.print()

    // 把DStream保存到文本文件中，会生成很多的小文件。一个批次生成一个目录
    val outputDir = "data/output1"
    stateDstream.repartition(1)
      .saveAsTextFiles(outputDir)

    ssc.start()
    ssc.awaitTermination()
  }
}
```

统计全局的key的状态，但是就算没有数据输入，也会在每一个批次的时候返回之前的key的状态。

这样的缺点：如果数据量很大的话，checkpoint 数据会占用较大的存储，而且效率也不高。

mapWithState：也是用于全局统计key的状态。如果没有数据输入，便不会返回之前的key的状态，有一点增量的感觉。

这样做的好处是，只关心那些已经发生的变化的key，对于没有数据输入，则不会返回那些没有变化的key的数据。即使数据量很大，checkpoint也不会像updateStateByKey那样，占用太多的存储。

```
package cn.lagou.streaming

import org.apache.spark.SparkConf
import org.apache.spark.streaming.dstream.{DStream,
ReceiverInputDStream}
import org.apache.spark.streaming.{Seconds, State, StateSpec,
StreamingContext}

object StateTracker2 {
  def main(args: Array[String]) {
    val conf: SparkConf = new SparkConf()
      .setMaster("local[*]")
      .setAppName(this.getClass.getCanonicalName)
    val ssc = new StreamingContext(conf, Seconds(2))
    ssc.sparkContext.setLogLevel("ERROR")

    ssc.checkpoint("data/checkpoint/")
```

```
    val lines: ReceiverInputDStream[String] =
ssc.socketTextStream("localhost", 9999)
    val words: DStream[String] = lines.flatMap(_.split("\\s+"))
    val wordDstream: DStream[(String, Int)] = words.map(x => (x,
1))

    // 函数返回的类型即为 mapWithState 的返回类型
    // (KeyType, Option[ValueType], State[StateType]) =>
MappedType
    def mappingFunction(key: String, one: Option[Int], state:
State[Int]): (String, Int) = {
      val sum: Int = one.getOrElse(0) +
state.getOption.getOrElse(0)
      state.update(sum)
      (key, sum)
    }

    val spec = StateSpec.function(mappingFunction _)
    val resultDStream: DStream[(String, Int)] =
pairsDStream.mapWithState[Int, (String, Int)](spec)
    resultDStream.cache()

    // 把DStream保存到文本文件中，会生成很多的小文件。一个批次生成一个目录
    val outputDir = "data/output2/"
    stateDStream.repartition(1)
      .saveAsTextFiles(outputDir)

    ssc.start()
    ssc.awaitTermination()
  }
}
```

第4节 DStream输出操作

输出操作定义 DStream 的输出操作。

与 RDD 中的惰性求值类似，如果一个 DStream 及其派生出的 DStream 都没有被执行输出操作，那么这些 DStream 就都不会被求值。

如果 StreamingContext 中没有设定输出操作，整个流式作业不会启动。

Output Operation	Meaning
print()	在运行流程序的Driver上，输出DStream中每一批次数据的最开始10个元素。用于开发和调试
saveAsTextFiles(*prefix*, [*suffix*])	以text文件形式存储 DStream 的内容。每一批次的存储文件名基于参数中的prefix和suffix
saveAsObjectFiles(*prefix*, [*suffix*])	以 Java 对象序列化的方式将Stream中的数据保存为 Sequence Files。每一批次的存储文件名基于参数中的为"prefix-TIME_IN_MS[.suffix]"
saveAsHadoopFiles(*prefix*, [*suffix*])	将Stream中的数据保存为 Hadoop files。每一批次的存储文件名基于参数中的为"prefix-TIME_IN_MS[.suffix]"
foreachRDD(*func*)	**最通用的输出操作。**将函数 func 应用于 DStream 的每一个RDD上

通用的输出操作 foreachRDD，用来对 DStream 中的 RDD 进行任意计算。在 foreachRDD中，可以重用 Spark RDD 中所有的 Action 操作。需要注意的：

- 连接不要定义在 Driver 中
- 连接定义在 RDD的 foreach 算子中，则遍历 RDD 的每个元素时都创建连接，得不偿失
- 应该在 RDD的 foreachPartition 中定义连接，每个分区创建一个连接
- 可以考虑使用连接池

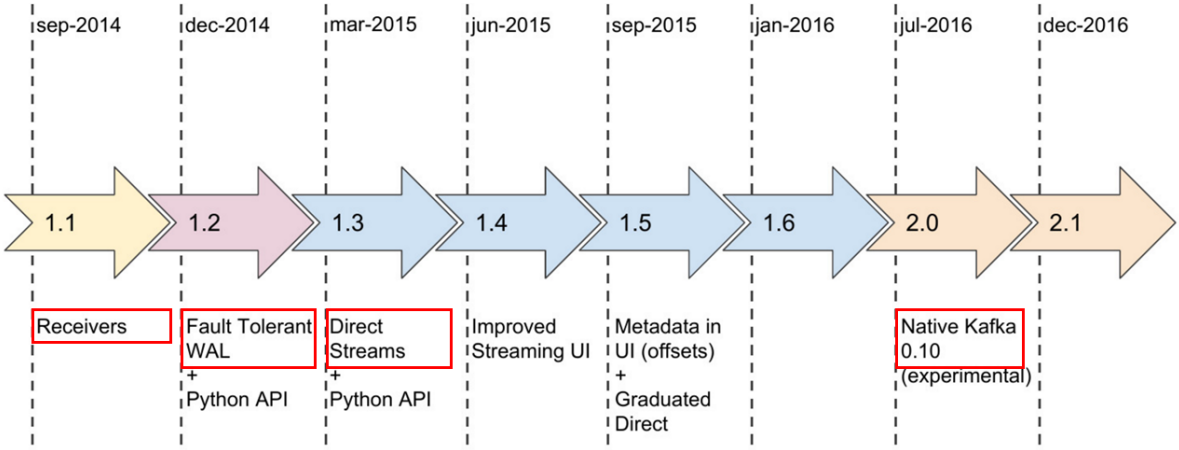
第5节 与Kafka整合

官网：http://spark.apache.org/docs/2.4.5/streaming-kafka-integration.html

针对不同的spark、kafka版本，集成处理数据的方式分为两种：Receiver Approach 和Direct Approach，不同集成版本处理方式的支持，可参考下图：

Note: Kafka 0.8 support is deprecated as of Spark 2.3.0.

	spark-streaming-kafka-0-8	spark-streaming-kafka-0-10
Broker Version	0.8.2.1 or higher	0.10.0 or higher
API Maturity	Deprecated	Stable
Language Support	Scala, Java, Python	Scala, Java
Receiver DStream	Yes	No
Direct DStream	Yes	Yes
SSL / TLS Support	No	Yes
Offset Commit API	No	Yes
Dynamic Topic Subscription	No	Yes



对Kafka的支持分为两个版本08（在高版本中将被废弃）、010，两个版本不兼容。

5.1 Kafka-08 接口

1、Receiver based Approach

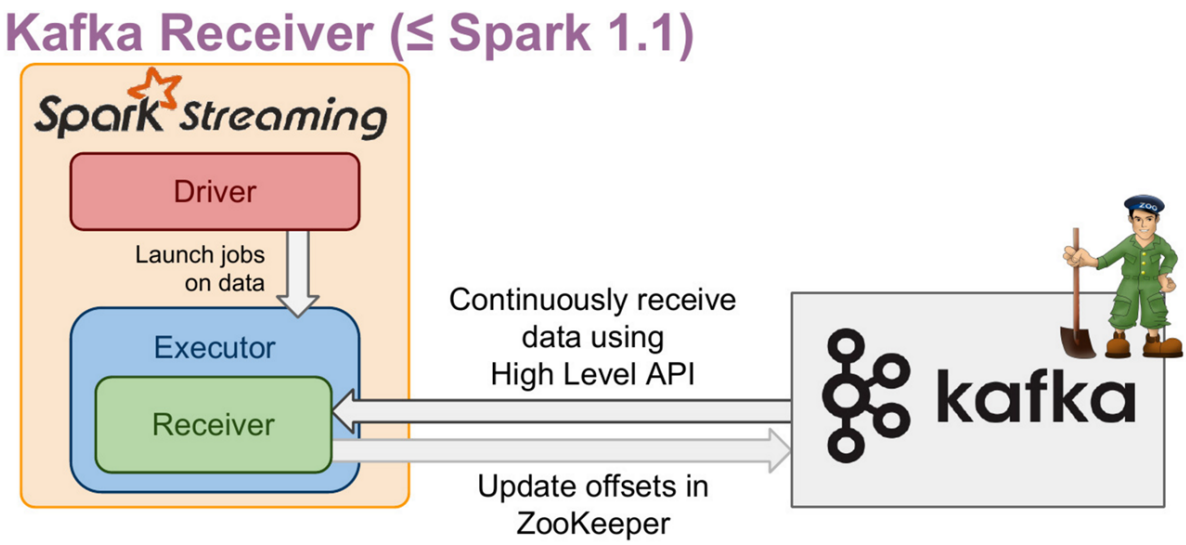
基于 Receiver 的方式使用 Kafka 旧版消费者高阶API实现。

对于所有的 Receiver，通过 Kafka 接收的数据被存储于 Spark 的 Executors上，底层是写入BlockManager中，默认200ms生成一个block（spark.streaming.blockInterval）。然后由 Spark Streaming 提交的 job 构建BlockRDD，最终以 Spark Core任务的形式运行。对应 Receiver方式，有以下几点需要注意：

- Receiver 作为一个常驻线程调度到 Executor上运行，占用一个cpu
- Receiver 个数由KafkaUtils.createStream调用次数决定，一次一个 Receiver
- kafka中的topic分区并不能关联产生在spark streaming中的rdd分区。增加在KafkaUtils.createStream()中的指定的topic分区数，仅仅增加了单个receiver消费的topic的线程数，它不会增加处理数据中的并行的spark的数量。【即：topicMap[topic,num_threads]中，value对应的数值是每个topic对应的消费线

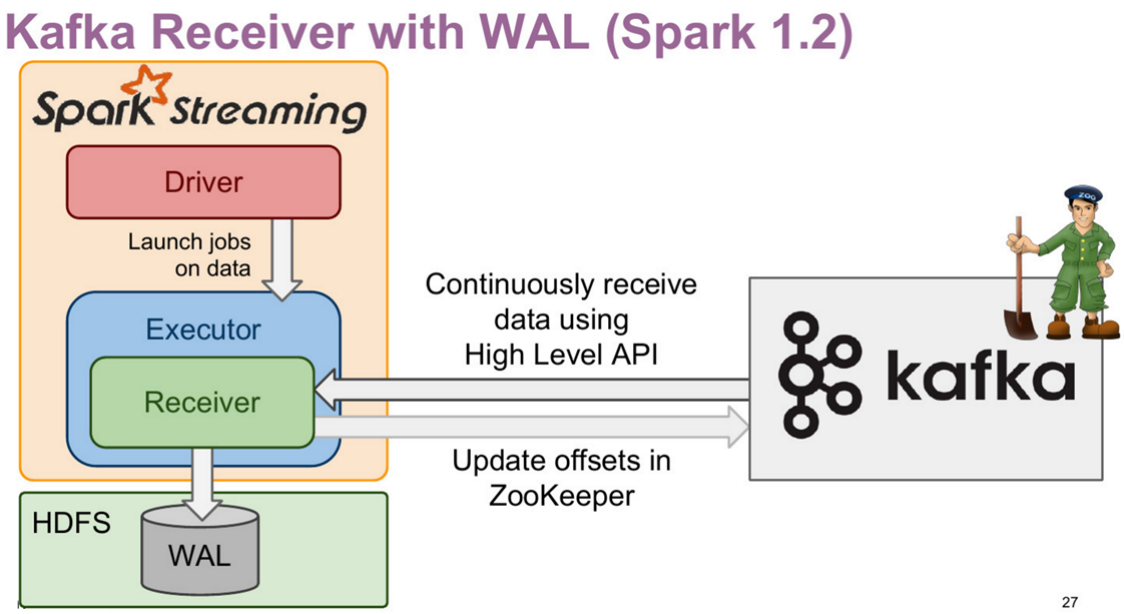
程数】

- receiver默认200ms生成一个block，可根据数据量大小调整block生成周期。一个block对应RDD一个分区。
- receiver接收的数据会放入到BlockManager，每个 Executor 都会有一个BlockManager实例，由于数据本地性，那些存在 Receiver 的 Executor 会被调度执行更多的 Task，就会导致某些executor比较空闲
- 默认情况下，Receiver是可能丢失数据的。可以通过设置spark.streaming.receiver.writeAheadLog.enable为true开启预写日志机制，将数据先写入一个可靠地分布式文件系统(如HDFS)，确保数据不丢失，但会损失一定性能



Kafka-08 接口（Receiver方式）：

- Offset保存在ZK中，系统管理
- 对应Kafka的版本 0.8.2.1+
- 接口底层实现使用 Kafka 旧版消费者高阶API
- DStream底层实现为BlockRDD



Kafka-08 接口（Receiver with WAL）：

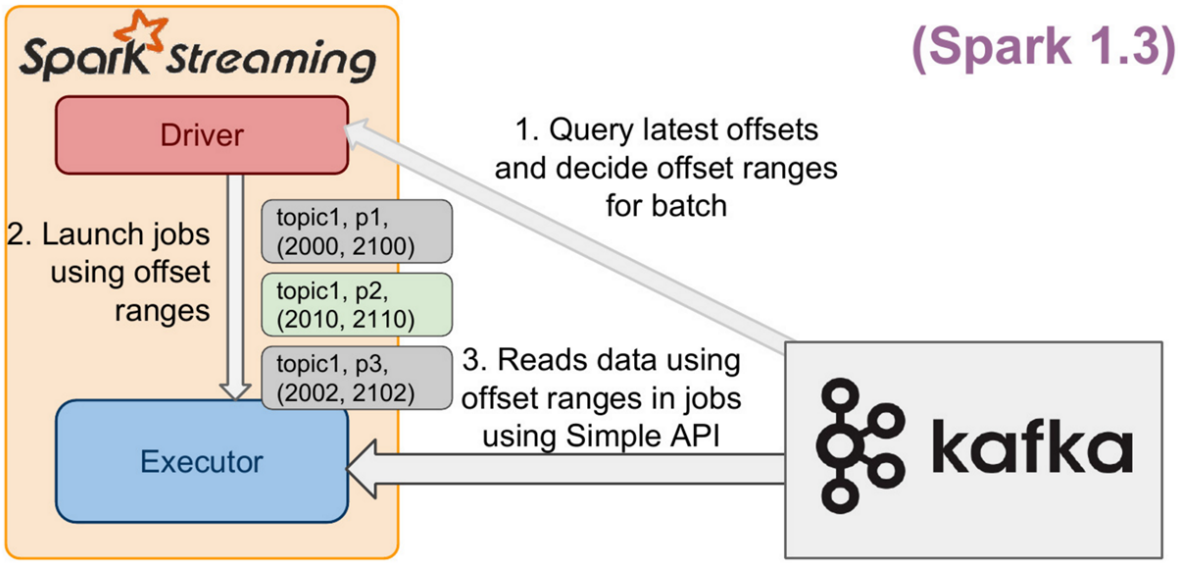
- 增强了故障恢复的能力
- 接收的数据与Dirver的元数据保存到HDFS
- 增加了流式应用处理的延迟

2、Direct Approach

Direct Approach是 Spark Streaming不使用Receiver集成kafka的方式，在企业生产环境中使用较多。相较于Receiver，有以下特点：

- 不使用 Receiver。减少不必要的CPU占用；减少了 Receiver接收数据写入BlockManager，然后运行时再通过blockId、网络传输、磁盘读取等来获取数据的整个过程，提升了效率；无需WAL，进一步减少磁盘IO；
- Direct方式生的RDD是KafkaRDD，它的分区数与 Kafka 分区数保持一致，便于把控并行度

  注意：在 Shuffle 或 Repartition 操作后生成的RDD，这种对应关系会失效
- 可以手动维护offset，实现 Exactly Once 语义



5.2 Kafka-010 接口

Spark Streaming与kafka 0.10的整合，和0.8版本的 Direct 方式很像。Kafka的分区和Spark的RDD分区是一一对应的，可以获取 offsets 和元数据，API 使用起来没有显著的区别。

添加依赖：

```
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming-kafka-0-
10_2.12</artifactId>
            <version>${spark.version}</version>
        </dependency>
```

不要手动添加 org.apache.kafka 相关的依赖，如kafka-clients。spark-streaming-kafka-0-10已经包含相关的依赖了，不同的版本会有不同程度的不兼容。

使用kafka010接口从 Kafka 中获取数据：

- Kafka集群
- kafka生产者发送数据
- Spark Streaming程序接收数

```
package cn.lagou.Streaming.kafka

import java.util.Properties
```

```
import org.apache.kafka.clients.producer.{KafkaProducer,
ProducerConfig, ProducerRecord}
import org.apache.kafka.common.serialization.StringSerializer

object KafkaProducer {
  def main(args: Array[String]): Unit = {
    // 定义 kafka 参数
    val brokers = "linux121:9092,linux122:9092,linux123:9092"
    val topic = "topicB"
    val prop = new Properties()

    prop.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, brokers)
    prop.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
classOf[StringSerializer])
    prop.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
classOf[StringSerializer])

    // KafkaProducer
    val producer = new KafkaProducer[String, String](prop)

    for (i <- 1 to 1000000){
      val msg = new ProducerRecord[String, String](topic,
i.toString, i.toString)
      // 发送消息
      producer.send(msg)
      println(s"i = $i")
      Thread.sleep(100)
    }

    producer.close()
  }
}
```

```
package cn.lagou.Streaming.kafka

import org.apache.kafka.clients.consumer.{ConsumerConfig,
ConsumerRecord}
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.log4j.{Level, Logger}
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.dstream.InputDStream
```

```
import org.apache.spark.streaming.kafka010.{ConsumerStrategies,
KafkaUtils, LocationStrategies}

object KafkaDStream1 {
  def main(args: Array[String]): Unit = {
    // 初始化
    Logger.getLogger("org").setLevel(Level.ERROR)
    val conf: SparkConf = new SparkConf()
      .setMaster("local[2]")
      .setAppName(this.getClass.getCanonicalName)
    val ssc = new StreamingContext(conf, Seconds(2))

    // 定义kafka相关参数
    val kafkaParams: Map[String, Object] =
getKafkaConsumerParams()
    val topics: Array[String] = Array("topicB")

    // 从 kafka 中获取数据
    val dstream: InputDStream[ConsumerRecord[String, String]] =
KafkaUtils.createDirectStream(
      ssc,
      LocationStrategies.PreferConsistent,
      ConsumerStrategies.Subscribe[String, String](topics,
kafkaParams)
    )

    // DStream输出
    dstream.foreachRDD{(rdd, time) =>
      if (!rdd.isEmpty()) {
        println(s"********** rdd.count = ${rdd.count()}; time =
$time **********")
      }
    }

    ssc.start()
    ssc.awaitTermination()
  }

  def getKafkaConsumerParameters(groupId: String): Map[String,
Object] = {
    Map[String, Object](
      ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG ->
"linux121:9092,linux122:9092,linux123:9092",
      ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG ->
classOf[StringDeserializer],
```

```
      ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG ->
classOf[StringDeserializer],
      ConsumerConfig.GROUP_ID_CONFIG -> groupId,
      ConsumerConfig.AUTO_OFFSET_RESET_CONFIG -> "earliest",
      ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG -> (false:
java.lang.Boolean)
    )
  }
}
```

LocationStrategies(本地策略)

- LocationStrategies.PreferBrokers：如果 Executor 在 kafka 集群中的某些节点上，可以使用这种策略。此时Executor 中的数据会来自当前broker节点
- LocationStrategies.PreferConsistent：大多数情况下使用的策略，将Kafka分区均匀的分布在Spark集群的 Executor上
- LocationStrategies.PreferFixed：如果节点之间的分区有明显的分布不均，使用这种策略。通过一个map指定将 topic 分区分布在哪些节点中

ConsumerStrategies(消费策略)

- ConsumerStrategies.Subscribe，用来订阅一组固定topic
- ConsumerStrategies.SubscribePattern，使用正则来指定感兴趣的topic
- ConsumerStrategies.Assign，指定固定分区的集合

这三种策略都有重载构造函数，允许指定特定分区的起始偏移量；使用 Subscribe 或 SubscribePattern 在运行时能实现分区自动发现。

Kafka相关命令：

```
# 创建 topic
kafka-topics.sh --zookeeper
linux121:2181,linux122:2181,linux123:2181 \
--create --topic topicB --replication-factor 2 --partitions 3

# 显示 topic 信息
kafka-topics.sh --zookeeper
linux121:2181,linux122:2181,linux123:2181 --topic topicA --
describe

# 检查 topic 的最大offset
```

```
kafka-run-class.sh  kafka.tools.GetOffsetShell \
--broker-list linux121:9092,linux122:9092,linux123:9092 --topic
topicA --time -1

# 列出所有的消费者
kafka-consumer-groups.sh --bootstrap-server linux121:9092  --list

# 检查消费者的offset
kafka-consumer-groups.sh --bootstrap-server
linux121:9092,linux122:9092,linux123:9092 \
--describe --group mygroup01

# 重置消费者offset
kafka-consumer-groups.sh --bootstrap-server
linux121:9092,linux122:9092,linux123:9092 \
--group group01 --reset-offsets --execute --to-offset 0 --topic
topicA
```

5.3 Offset 管理

Spark Streaming集成Kafka，允许从Kafka中读取一个或者多个 topic 的数据。一个Kafka Topic包含一个或多个分区，每个分区中的消息顺序存储，并使用 offset 来标记消息的位置。开发者可以在 Spark Streaming 应用中通过 offset 来控制数据的读取位置。

Offsets 管理对于保证流式应用在整个生命周期中数据的连贯性是非常重要的。如果在应用停止或报错退出之前没有将 offset 持久化保存，该信息就会丢失，那么Spark Streaming就没有办法从上次停止或报错的位置继续消费Kafka中的消息。

1、获取偏移量(Obtaining Offsets)

Spark Streaming与kafka整合时，允许获取其消费的 offset ，具体方法如下：

```
stream.foreachRDD { rdd =>
  val offsetRanges =
rdd.asInstanceOf[HasOffsetRanges].offsetRanges
  rdd.foreachPartition { iter =>
    val o: OffsetRange =
offsetRanges(TaskContext.get.partitionId)
    println(s"${o.topic} ${o.partition} ${o.fromOffset}
${o.untilOffset}")
  }
}
```

注意：对HasOffsetRanges的类型转换只有在对 createDirectStream 调用的第一个方法中完成时才会成功，而不是在随后的方法链中。RDD分区和Kafka分区之间的对应关系在 shuffle 或 重分区后会丧失，如reduceByKey 或 window。

2、存储偏移量(Storing Offsets)

在Streaming程序失败的情况下，Kafka交付语义取决于**如何以及何时**存储偏移量。Spark输出操作的语义为 at-least-once。

如果要实现EOS语义(Exactly Once Semantics)，必须**在幂等的输出之后存储偏移量 或者 将存储偏移量与输出放在一个事务中。**可以按照增加可靠性（和代码复杂度）的顺序使用以下选项来存储偏移量：

- **Checkpoint**

Checkpoint是对Spark Streaming运行过程中的元数据和每RDDs的数据状态保存到一个持久化系统中，当然这里面也包含了offset，一般是HDFS、S3，如果应用程序或集群挂了，可以迅速恢复。

如果Streaming程序的代码变了，重新打包执行就会出现反序列化异常的问题。

这是因为Checkpoint首次持久化时会将整个 jar 包序列化，以便重启时恢复。重新打包之后，新旧代码逻辑不同，就会报错或仍然执行旧版代码。

要解决这个问题，只能将HDFS上的checkpoint文件删除，但这样也会同时删除 Kafka 的offset信息。

- **Kafka**

默认情况下，消费者定期自动提交偏移量，它将偏移量存储在一个特殊的Kafka主题中（__consumer_offsets）。但在某些情况下，这将导致问题，因为消息可能已经被消费者从Kafka拉去出来，但是还没被处理。

可以将 enable.auto.commit 设置为 false ，在 Streaming 程序输出结果之后，手动提交偏移到kafka。

与检查点相比，使用Kafka保存偏移量的优点是无论应用程序代码如何更改，偏移量仍然有效。

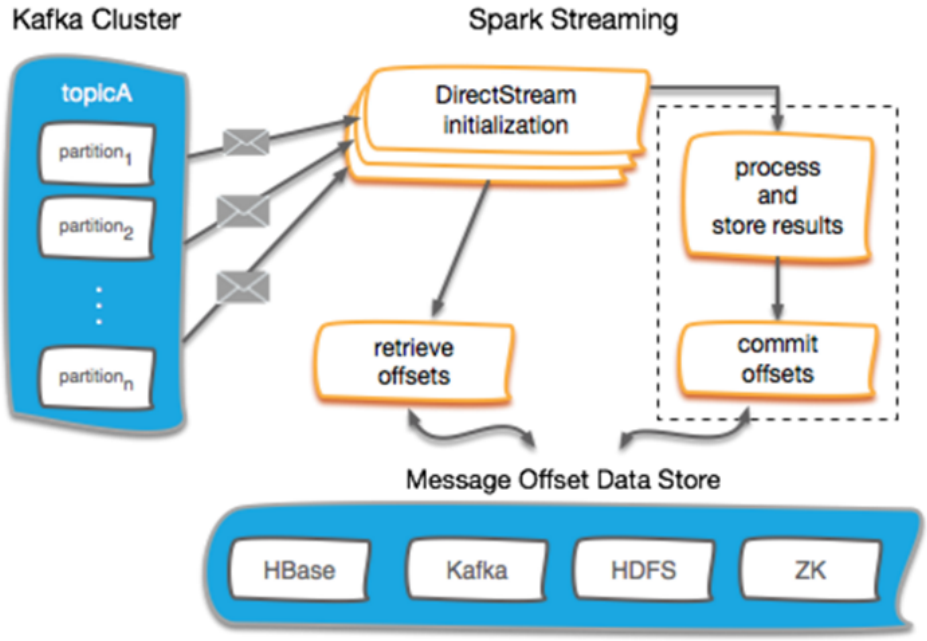
```
stream.foreachRDD { rdd =>
  val offsetRanges =
rdd.asInstanceOf[HasOffsetRanges].offsetRanges

  // 在输出操作完成之后，手工提交偏移量；此时将偏移量提交到 Kafka 的消息队列中
  stream.asInstanceOf[CanCommitOffsets].commitAsync(offsetRanges)
}
```

与HasOffsetRanges一样，只有在createDirectStream的结果上调用时，转换到CanCommitOffsets才会成功，而不是在转换之后。commitAsync调用是线程安全的，但必须在输出之后执行。

- **自定义存储**

Offsets可以通过多种方式来管理，但是一般来说遵循下面的步骤:

- 在 DStream 初始化的时候，需要指定每个分区的offset用于从指定位置读取数据
- 读取并处理消息
- 处理完之后存储结果数据
- 用虚线圈存储和提交offset，强调用户可能会执行一系列操作来满足他们更加严格的语义要求。这包括幂等操作和通过原子操作的方式存储offset
- 将 offsets 保存在外部持久化数据库如 HBase、Kafka、HDFS、ZooKeeper、Redis、MySQL ... ...



可以将 Offsets 存储到HDFS中，但这并不是一个好的方案。因为HDFS延迟有点高，此外将每批次数据的offset存储到HDFS中还会带来小文件问题；

可以将 Offset 存储到保存ZK中，但是将ZK作为存储用，也并不是一个明智的选择，同时ZK也不适合频繁的读写操作；

3、Redis管理的Offset

要想将Offset保存到外部存储中，关键要实现以下几个功能：

- Streaming程序启动时，从外部存储获取保存的Offsets（执行一次）
- 在foreachRDD中，每个批次数据处理之后，更新外部存储的offsets（多次执行）

Redis管理的Offsets：

```
1、数据结构选择：Hash；key、field、value
Key：kafka:topic:TopicName:groupid
Field：partition
Value：offset

2、从 Redis 中获取保存的offsets

3、消费数据后将offsets保存到redis
```

案例一：使用自定义的offsets，从kafka读数据；处理完数据后打印offsets

```
package cn.lagou.Streaming.kafka

import org.apache.kafka.clients.consumer.{ConsumerConfig,
ConsumerRecord}
import org.apache.kafka.common.TopicPartition
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.log4j.{Level, Logger}
import org.apache.spark.{SparkConf, TaskContext}
import org.apache.spark.streaming.dstream.InputDStream
import org.apache.spark.streaming.kafka010.{ConsumerStrategies,
HasOffsetRanges, KafkaUtils, LocationStrategies, OffsetRange}
import org.apache.spark.streaming.{Seconds, StreamingContext}

// 使用自定义的offsets，从kafka读数据；处理完数据后打印offsets
object KafkaDStream2 {
  def main(args: Array[String]): Unit = {
    // 初始化
    Logger.getLogger("org").setLevel(Level.ERROR)
    val conf: SparkConf = new SparkConf()
      .setMaster("local[2]")
      .setAppName(this.getClass.getCanonicalName)
    val ssc = new StreamingContext(conf, Seconds(2))

    // 定义kafka相关参数
    val kafkaParams: Map[String, Object] =
getKafkaConsumerParams()
    val topics: Array[String] = Array("topicB")

    // 从指定的位置获取kafka数据
    val offsets: collection.Map[TopicPartition, Long] = Map(
      new TopicPartition("topicB", 0) -> 100,
      new TopicPartition("topicB", 1) -> 200,
      new TopicPartition("topicB", 2) -> 300
    )

    // 从 kafka 中获取数据
    val dstream: InputDStream[ConsumerRecord[String, String]] =
KafkaUtils.createDirectStream[String, String](
      ssc,
      LocationStrategies.PreferConsistent,
      ConsumerStrategies.Subscribe[String, String](topics,
kafkaParams, offsets)
    )

    // DStream输出
```

```
    dstream.foreachRDD{(rdd, time) =>
      // 输出结果
      println(s"********** rdd.count = ${rdd.count()}; time =
$time **********")

      // 输出offset
      val offsetRanges: Array[OffsetRange] =
rdd.asInstanceOf[HasOffsetRanges].offsetRanges

      rdd.foreachPartition { iter =>
        val o: OffsetRange =
offsetRanges(TaskContext.get.partitionId)
        // 输出kafka消费的offset
        println(s"${o.topic} ${o.partition} ${o.fromOffset}
${o.untilOffset}")
      }
    }

    ssc.start()
    ssc.awaitTermination()
  }

  def getKafkaConsumerParams(groupId: String = "mygroup1"):
Map[String, Object] = {
    Map[String, Object](
      ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG ->
"linux121:9092,linux122:9092,linux123:9092",
      ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG ->
classOf[StringDeserializer],
      ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG ->
classOf[StringDeserializer],
      ConsumerConfig.GROUP_ID_CONFIG -> groupId,
      ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG -> (false:
java.lang.Boolean))
  }
}

// kafka命令，检查 topic offset的值
// kafka-run-class.sh  kafka.tools.GetOffsetShell --broker-list
linux121:9092,linux122:9092,linux123:9092 --topic topicB --time
-1
```

案例二：根据 key 从 Redis 获取offsets，根据该offsets从kafka读数据；处理完数据后将offsets保存到 Redis

```
1、数据结构选择：Hash；key、field、value
Key：kafka:topic:TopicName:groupid
Field：partition
Value：offset

2、从 Redis 中获取保存的offsets

3、消费数据后将offsets保存到redis
```

引入依赖

```
        <!-- jedis -->
        <dependency>
            <groupId>redis.clients</groupId>
            <artifactId>jedis</artifactId>
            <version>2.9.0</version>
        </dependency>
```

主程序（从kafka获取数据，使用 Redis 保存offsets）

```
package cn.lagou.Streaming.kafka

import cn.lagou.sparksql.kafka.OffsetsRedisUtils
import org.apache.kafka.clients.consumer.{ConsumerConfig,
ConsumerRecord}
import org.apache.kafka.common.TopicPartition
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.log4j.{Level, Logger}
import org.apache.spark.streaming.dstream.InputDStream
import org.apache.spark.streaming.kafka010._
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.{SparkConf, TaskContext}

object KafkaDStream3 {
  def main(args: Array[String]): Unit = {
    // 初始化
    Logger.getLogger("org").setLevel(Level.ERROR)
```

```
    val conf = new
SparkConf().setAppName("FileDStream").setMaster("local[*]")
    val ssc = new StreamingContext(conf, Seconds(5))

    // 定义kafka相关参数
    val groupId: String = "mygroup01"
    val topics: Array[String] = Array("topicB")
    val kafkaParams: Map[String, Object] =
getKafkaConsumerParameters(groupId)

    // 从kafka获取offsets
    val offsets: Map[TopicPartition, Long] =
OffsetsRedisUtils.getOffsetFromRedis(topics, groupId)

    // 创建DStream
    val dstream: InputDStream[ConsumerRecord[String, String]] =
KafkaUtils.createDirectStream(
      ssc,
      LocationStrategies.PreferConsistent,
      ConsumerStrategies.Subscribe[String, String](topics,
kafkaParams, offsets)
    )

    // DStream转换&输出
    dstream.foreachRDD{ (rdd, time) =>
      if (!rdd.isEmpty()) {
        // 处理消息
        println(s"********** rdd.count = ${rdd.count()}; time =
$time **************")

        // 将offsets保存到redis
        val offsetRanges: Array[OffsetRange] =
rdd.asInstanceOf[HasOffsetRanges].offsetRanges
        OffsetsRedisUtils.saveOffsetsToRedis(offsetRanges)
      }
    }

    // 启动作业
    ssc.start()
    ssc.awaitTermination()
  }

  def getKafkaConsumerParameters(groupid: String): Map[String,
Object] = {
    Map[String, Object](
```

```
      ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG ->
"linux121:9092,linux122:9092,linux123:9092",
      ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG ->
classOf[StringDeserializer],
      ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG ->
classOf[StringDeserializer],
      ConsumerConfig.GROUP_ID_CONFIG -> groupid,
      ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG -> (false:
java.lang.Boolean)
//      ConsumerConfig.AUTO_OFFSET_RESET_CONFIG -> "earliest"
    )
  }
}
```

工具类（Redis读取/保存offsets）

```
package cn.lagou.Streaming.kafka

import org.apache.kafka.common.TopicPartition
import org.apache.spark.streaming.kafka010.OffsetRange
import redis.clients.jedis.{Jedis, JedisPool, JedisPoolConfig}

import scala.collection.mutable

object OffsetsRedisUtils {
  private val config = new JedisPoolConfig
  private val redisHost = "192.168.80.123"
  private val redisPort = 6379
  // 最大连接
  config.setMaxTotal(30)
  // 最大空闲
  config.setMaxIdle(10)

  private val pool = new JedisPool(config, redisHost, redisPort,
10000)

  private val topicPrefix = "kafka:topic"

  // key的格式为 => prefix : topic : groupId
  private def getKey(topic: String, groupId: String, prefix:
String = topicPrefix): String = s"$prefix:$topic:$groupId"

  private def getRedisConnection: Jedis = pool.getResource
```

```
  // 从 redis 中获取offsets
  def getOffsetFromRedis(topics: Array[String], groupId: String):
Map[TopicPartition, Long] = {
    val jedis: Jedis = getRedisConnection

    val offsets: Array[mutable.Map[TopicPartition, Long]] =
topics.map { topic =>
      import scala.collection.JavaConverters._
      jedis.hgetAll(getKey(topic, groupId))
        .asScala
        .map { case (partition, offset) => new
TopicPartition(topic, partition.toInt) -> offset.toLong }
    }
    // println(s"offsets = ${offsets.toBuffer}")

    // 归还资源
    jedis.close()

    offsets.flatten.toMap
  }

  // 将 offsets 保存到 redis
  def saveOffsetsToRedis(ranges: Array[OffsetRange], groupId:
String): Unit = {
    val jedis: Jedis = getRedisConnection

    ranges.map(range => (range.topic, range.partition ->
range.untilOffset))
      .groupBy(_._1)
      .map { case (topic, buffer) => (topic, buffer.map(_._2)) }
      .foreach { case (topic, partitionAndOffset) =>
        val offsets: Array[(String, String)] =
partitionAndOffset.map(elem => (elem._1.toString,
elem._2.toString))

        import scala.collection.JavaConverters._

        jedis.hmset(getKey(topic, groupId), offsets.toMap.asJava)
      }

    // 归还资源
    jedis.close()
  }
```

```
  def main(args: Array[String]): Unit = {
    val topics = Array("topicB")
    val groupId = "group01"
    val offsets: Map[TopicPartition, Long] =
getOffsetFromRedis(topics, groupId)
    println(offsets)
    offsets.foreach(println)

    val jedis: Jedis = getRedisConnection
    import scala.collection.JavaConverters._
    jedis.hgetAll(getKey("topicB",
groupId)).asScala.foreach(println)
  }
}
```

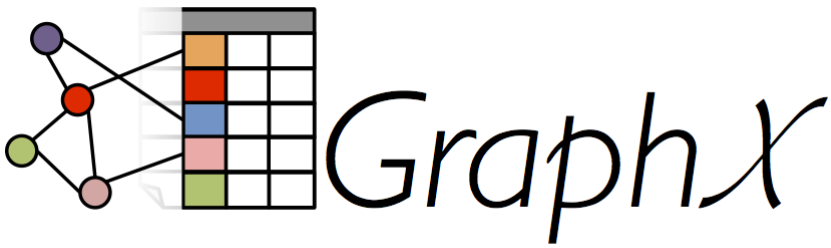
第四部分 Spark GraphX

第1节 Spark GraphX概述

GraphX 是 Spark 一个组件，专门用来表示图以及进行图的并行计算。GraphX 通过重新定义了图的抽象概念来拓展了 RDD： 定向多图，其属性附加到每个顶点和边。

为了支持图计算， GraphX 公开了一系列基本运算符（比如：mapVertices、mapEdges、subgraph）以及优化后的 Pregel API 变种。此外，还包含越来越多的图算法和构建器，以简化图形分析任务。

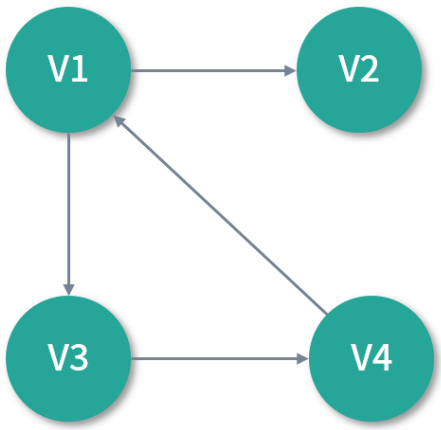
GraphX在图顶点信息和边信息存储上做了优化，使得图计算框架性能相对于原生RDD实现得以较大提升，接近或到达 GraphLab 等专业图计算平台的性能。GraphX最大的贡献是，在Spark之上提供一栈式数据解决方案，可以方便且高效地完成图计算的一整套流水作业。



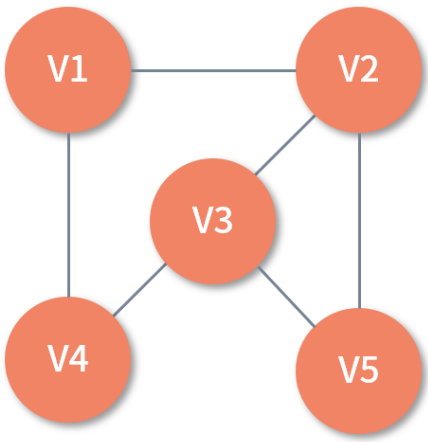
1.1 图的相关术语

图是一种较线性表和树更为复杂的数据结构，图表达的是多对多的关系。

如下图所示，G1 是一个简单的图，其中 V1、V2、V3、V4 被称作**顶点（Vertex）**，任意两个顶点之间的通路被称为**边（Edge）**，它可以由（V1、V2）有序对来表示，这时称 G1 为有向图，意味着边是有方向的，若以无序对来表示图中一条边，则该图为无向图，如 G2。



G 1



G 2

在 G1 中，与顶点相关联的边的数量被称为**顶点的度**（Degree）。其中，以顶点为起点的边的数量被称为该顶点的**出度**（OutDegree），以顶点为终点的边的数量被称为该顶点的**入度**（InDegree）。

以 G1 中的 V1 举例，V1 的度为 3，其中出度为 2，入度为 1。在无向图 G2 中，如果任意两个顶点之间是连通的，则称 G2 为连通图（Connected Graph）。在有向图中 G1 中，如果任意两个顶点 Vm、Vn 且 m ≠ n，**从 Vm 到 Vn 以及从 Vn 到 Vm 之间都存在通路，则称 G1 为强连通图（Strongly Connected Graph）**。任意两个顶点之间若存在通路，则称为路径（Path），用一个顶点序列表示，若第一个顶点和最后一个顶点相同，则称为回路或者环（Cycle）。

1.2 图数据库与图计算

Neo4j 是一个比较老牌的开源图数据库，目前在业界的使用也较为广泛，它提供了一种简单易学的查询语言 Cypher。

Neo4j 支持交互式查询，查询效率很高。能够迅速从整网中找出符合特定模式的子网，供随后分析之用，适用于OLTP 场景。

Neo4j 是图数据库，偏向于存储和查询。能存储关联关系比较复杂，实体之间的连接丰富。比如社交网络、知识图谱、金融风控等领域的数据。擅长从某个点或某些点出发，根据特定条件在复杂的关联关系中找到目标点或边。如在社交网络中找到某个点三步以内能认识的人，这些人可以认为是潜在朋友。数据量限定在一定范围内，能短时完成的查询就是所谓的OLTP操作。

Neo4j 查询与插入速度较快，没有分布式版本，容量有限，而且一旦图变得非常大，如数十亿顶点，数百亿边，查询速度将变得缓慢。Neo4j 分为社区版和企业版，企业版有一些高级功能，需要授权，价格昂贵。

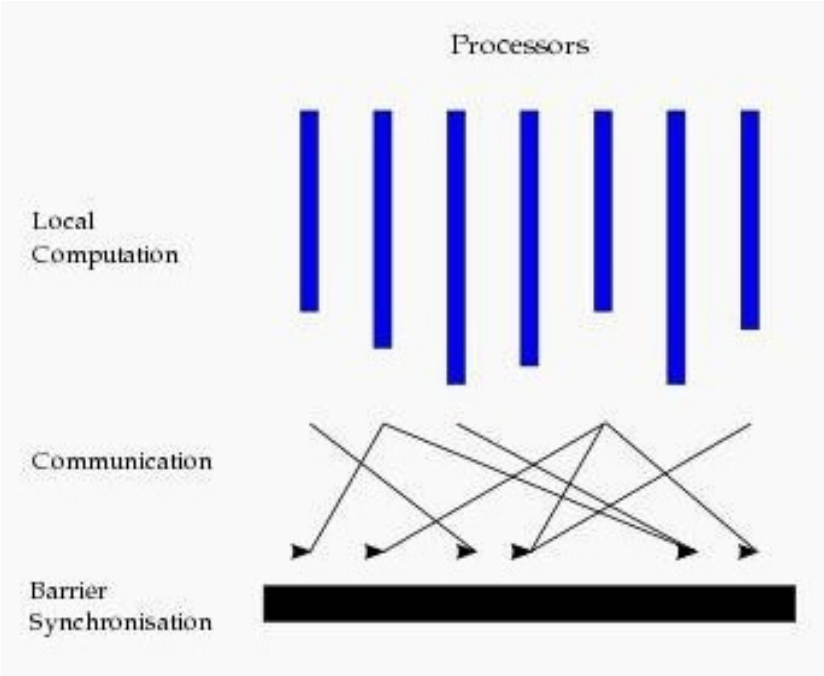
比较复杂的分析和算法，如基于图的聚类，PageRank 算法等，这类计算任务对于图数据库来说就很难胜任了，主要由一些图挖掘技术来负责。

Pregel 是 Google 于 2010 年在 SIGMOD 会议上发表的《Pregel: A System for Large-Scale Graph Processing》论文中提到的海量并行图挖掘的抽象框架，Pregel 与 Dremel 一样，是 Google 新三驾马车之一，它基于 BSP 模型（Bulk Synchronous Parallel，整体同步并行计算模型），将计算分为若干个超步（super step），在超步内，通过消息来传播顶点之间的状态。Pregel 可以看成是同步计算，即等所有顶点完成处理后再进行下一轮的超步，Spark 基于 Pregel 论文实现的海量并行图挖掘框架 GraphX。

1.3 图计算模式

目前基于图的并行计算框架已经有很多，比如来自Google的Pregel、来自Apache开源的图计算框架Giraph / HAMA以及最为著名的GraphLab，其中Pregel、HAMA和Giraph都是非常类似的，都是基于BSP模式。

BSP即整体同步并行，它将计算分成一系列超步的迭代。从纵向上看，它是一个串行模式，而从横向上看，它是一个并行的模式，每两个超步之间设置一个栅栏（barrier），即整体同步点，确定所有并行的计算都完成后再启动下一轮超步。



每一个超步包含三部分内容：

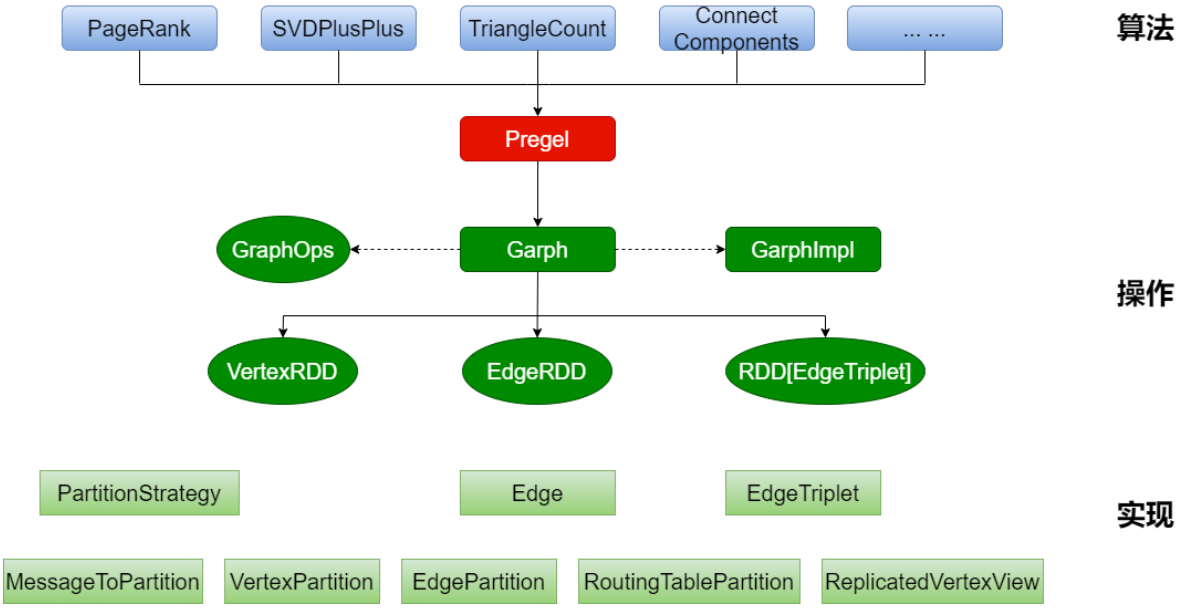
- **计算compute**：每一个processor利用上一个超步传过来的消息和本地的数据进行本地计算
- **消息传递**：每一个processor计算完毕后，将消息传递个与之关联的其它processors
- **整体同步点**：用于整体同步，确定所有的计算和消息传递都进行完毕后，进入下一个超步

第2节 Spark GraphX 基础

```
架构
存储模式
核心数据结构
```

GraphX 与 Spark 其他组件相比相对独立，拥有自己的核心数据结构与算子。

2.1 GraphX 架构



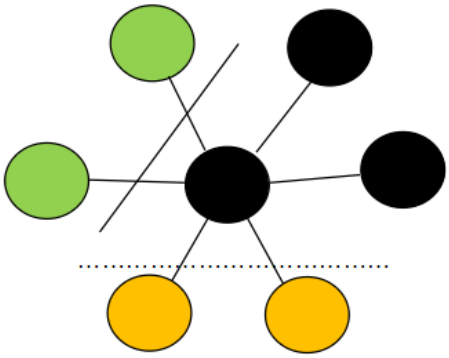
GraphX的整体架构可以分为三个部分：

- 算法层。基于 Pregel 接口实现了常用的图算法。包括 PageRank、SVDPlusPlus、TriangleCount、 ConnectedComponents、StronglyConnectedConponents 等算法
- 接口层。在底层 RDD 的基础之上实现了 Pregel 模型 BSP 模式的计算接口
- 底层。图计算的核心类，包含：VertexRDD、EdgeRDD、RDD[EdgeTriplet]

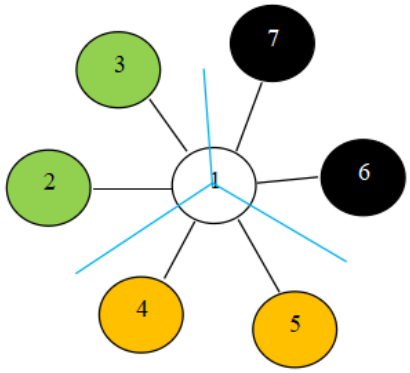
2.2 存储模式

巨型图的存储总体上有边分割和点分割两种存储方式。2013年，GraphLab2.0将其存储方式由边分割变为点分割，在性能上取得重大提升，目前基本上被业界广泛接受并使用。

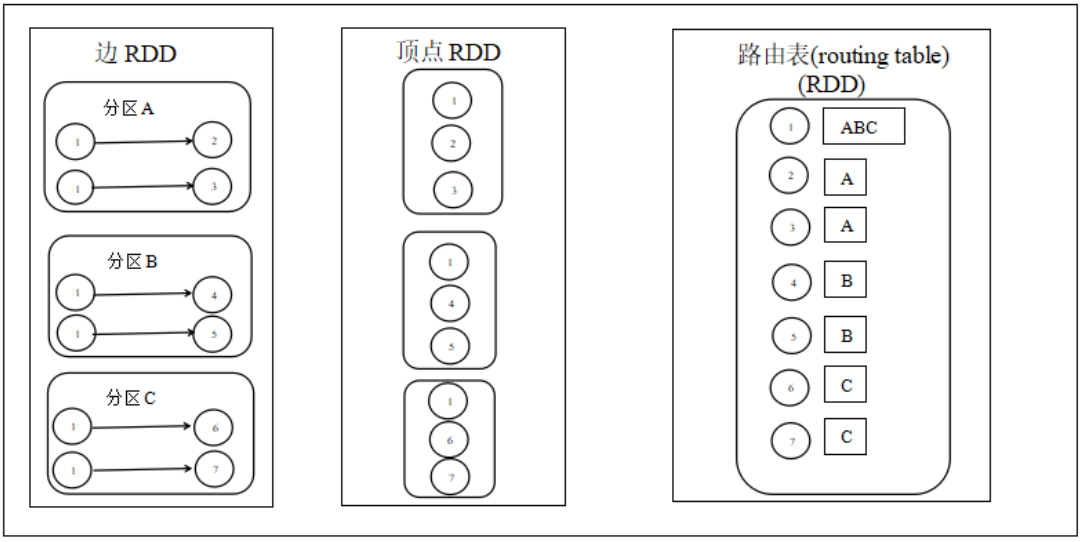
- **边分割（Edge-Cut）**：每个顶点都存储一次，但有的边会被打断分到两台机器上。这样做的好处是节省存储空间；坏处是对图进行基于边的计算时，对于一条两个顶点被分到不同机器上的边来说，要跨机器通信传输数据，内网通信流量大
- **点分割（Vertex-Cut）**：每条边只存储一次，都只会出现在一台机器上。邻居多的点会被复制到多台机器上，增加了存储开销，同时会引发数据同步问题。好处是可以大幅减少内网通信量



Edge Cut



Vertex Cut



按顶点分割图

虽然两种方法互有利弊，但现在是**点分割占上风**，各种分布式图计算框架都将自己底层的存储形式变成了点分割。主要原因有以下两个：

- 磁盘价格下降，存储空间不再是问题，而内网的通信资源没有突破性进展，集群计算时内网带宽是宝贵的，时间比磁盘更珍贵。这点就类似于常见的空间换时间的策略；
- 在当前的应用场景中，绝大多数网络都是“无尺度网络”，遵循幂律分布，不同点的邻居数量相差非常悬殊。而边分割会使那些多邻居的点所相连的边大多数被分到不同的机器上，这样的数据分布会使得内网带宽更加捉襟见肘，于是边分割存储方式被渐渐抛弃了；

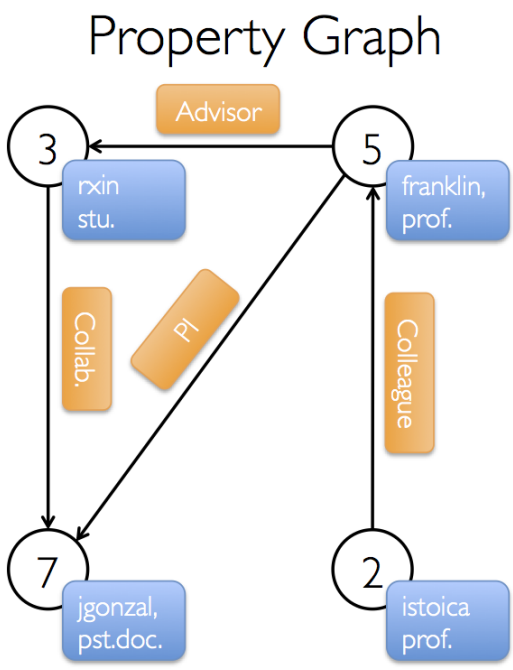
2.3 核心数据结构

核心数据结构包括：`graph`、`vertices`、`edges`、`triplets`

GraphX API 的开发语言目前仅支持 Scala。GraphX 的核心数据结构 Graph 由 RDD 封装而成。

1、Graph

GraphX 用属性图的方式表示图，顶点有属性，边有属性。存储结构采用边集数组的形式，即一个顶点表，一个边表，如下图所示：



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

顶点 ID 是非常重要的字段，它不光是顶点的唯一标识符，也是描述边的唯一手段。

顶点表与边表实际上就是 RDD，它们分别为 VertexRDD 与 EdgeRDD。在 Spark 的源码中，Graph 类如下：

```
41  abstract class Graph[VD: ClassTag, ED: ClassTag] protected () extends Serializable {
42
43    /**...*/
49    val vertices: VertexRDD[VD]
50
51    /**...*/
62    val edges: EdgeRDD[ED]
63
64    /**...*/
80    val triplets: RDD[EdgeTriplet[VD, ED]]
81
82    /**...*/
90    def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
91
92    /**...*/
97    def cache(): Graph[VD, ED]
98
99    /**...*/
105   def checkpoint(): Unit
```

- vertices 为顶点表，VD 为顶点属性类型
- edges 为边表，ED 为边属性类型
- 可以通过 Graph 的 vertices 与 edges 成员直接得到顶点 RDD 与边 RDD
- 顶点 RDD 类型为 VerticeRDD，继承自 RDD[(VertexId, VD)]
- 边 RDD 类型为 EdgeRDD，继承自 RDD[Edge[ED]]

2、vertices

vertices对应着名为 VertexRDD 的RDD。这个RDD由顶点id和顶点属性两个成员变量。

```
abstract class VertexRDD[VD](
    sc: SparkContext,
    deps: Seq[Dependency[_]]) extends RDD[(VertexId, VD)](sc, deps) {
```

VertexRDD继承自 RDD[(VertexId, VD)]，这里VertexId表示顶点id，VD表示顶点所带的属性的类别。

VertexId 实际上是一个Long类型的数据；

```
package object graphx {
  /**
   * A 64-bit vertex identifier that uniquely identifies a vertex within a graph. It does not need
   * to follow any ordering or any constraints other than uniqueness.
   */
  type VertexId = Long

  /** Integer identifier of a graph partition. Must be less than 2^30. */
  // TODO: Consider using Char.
  type PartitionID = Int

  private[graphx] type VertexSet = OpenHashSet[VertexId]
}
```

3、edges

edges对应着EdgeRDD。这个RDD拥有三个成员变量，分别是源顶点id、目标顶点id以及边属性。

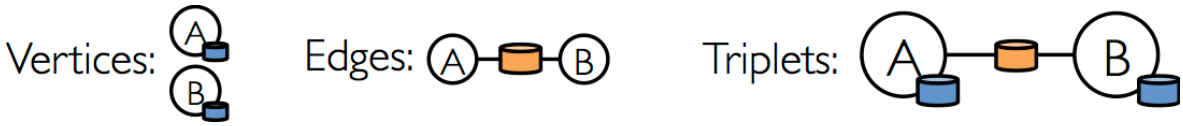
```
abstract class EdgeRDD[ED](
    sc: SparkContext,
    deps: Seq[Dependency[_]]) extends RDD[Edge[ED]](sc, deps) {
```

Edge代表边，由 源顶点id、目标顶点id、以及边的属性构成。

```
case class Edge[@specialized(Char, Int, Boolean, Byte, Long, Float, Double) ED] (
    var srcId: VertexId = 0,
    var dstId: VertexId = 0,
    var attr: ED = null.asInstanceOf[ED])
  extends Serializable {
```

4、triplets

triplets 表示边点三元组，如下图所示（其中圆柱形分别代表顶点属性与边属性）：



通过 triplets 成员，用户可以直接获取到起点顶点、起点顶点属性、终点顶点、终点顶点属性、边与边属性信息。triplets 的生成可以由边表与顶点表通过 ScrId 与 DstId 连接而成。

triplets对应着EdgeTriplet。它是一个三元组视图，这个视图逻辑上将顶点和边的属性保存为一个RDD[EdgeTriplet[VD, ED]]。

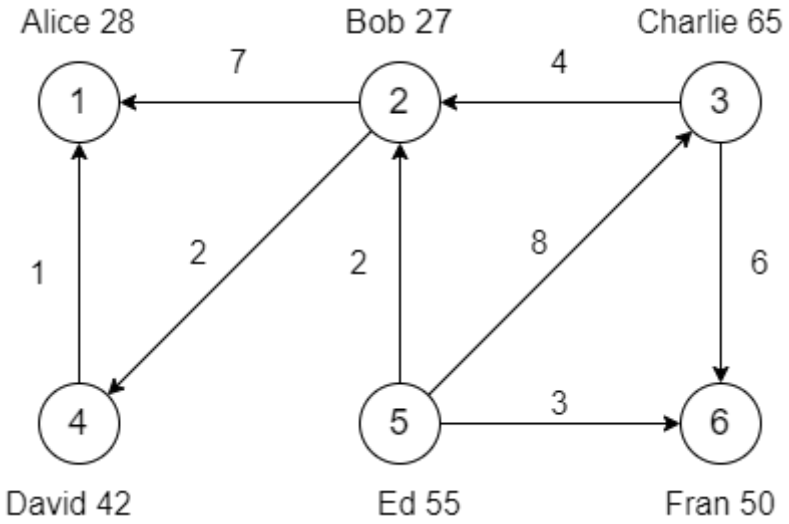
```
class EdgeTriplet[VD, ED] extends Edge[ED] {
  /**...*/
  var srcAttr: VD = _ // nullValue[VD]

  /**...*/
  var dstAttr: VD = _ // nullValue[VD]

  /**...*/
  protected[spark] def set(other: Edge[ED]): EdgeTriplet[VD, ED] = {
    srcId = other.srcId
    dstId = other.dstId
    attr = other.attr
    this
  }
```

第3节 Spark GraphX计算

图的定义
属性操作
转换操作
结构操作
关联操作
聚合操作
Pregel API

引入依赖：

```
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-graphx_2.12</artifactId>
            <version>${spark.version}</version>
        </dependency>
```



案例一：图的基本操作

```
package graphx

import org.apache.spark.graphx.{Edge, Graph, VertexId, VertexRDD}
import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}

object GraphXExample1 {
  def main(args: Array[String]): Unit = {
    // 初始化
    val conf: SparkConf = new
SparkConf().setAppName(this.getClass.getCanonicalName).setMaster(
"local[*]")
    val sc = new SparkContext(conf)
    sc.setLogLevel("warn")

    // 初始化数据
    // 定义顶点 (Long, info)
    val vertexArray: Array[(VertexId, (String, Int))] = Array(
      (1L, ("Alice", 28)),
      (2L, ("Bob", 27)),
      (3L, ("Charlie", 65)),
      (4L, ("David", 42)),
      (5L, ("Ed", 55)),
      (6L, ("Fran", 50))
    )
```

```
    // 定义边 (Long, Long, attr)
    val edgeArray: Array[Edge[Int]] = Array(
      Edge(2L, 1L, 7),
      Edge(2L, 4L, 2),
      Edge(3L, 2L, 4),
      Edge(3L, 6L, 3),
      Edge(4L, 1L, 1),
      Edge(5L, 2L, 2),
      Edge(5L, 3L, 8),
      Edge(5L, 6L, 3)
    )

    // 构造vertexRDD和edgeRDD
    val vertexRDD: RDD[(Long, (String, Int))] =
sc.makeRDD(vertexArray)
    val edgeRDD: RDD[Edge[Int]] = sc.makeRDD(edgeArray)

    // 构造图Graph[VD,ED]
    val graph: Graph[(String, Int), Int] = Graph(vertexRDD,
edgeRDD)

    // 属性操作示例
    // 找出图中年龄大于30的顶点
    graph.vertices
      .filter { case (_, (_, age)) => age > 30 }
      .foreach(println)

    // 找出图中属性大于5的边
    graph.edges
        .filter{edge => edge.attr>5}
        .foreach(println)

    // 列出边属性>5的tripltes
    graph.triplets
        .filter(t => t.attr > 5)
        .foreach(println)

    // degrees操作
    // 找出图中最大的出度、入度、度数
    println("********** 出度 **********")
    graph.outDegrees.foreach(println)
    val outDegress: (VertexId, Int) = graph.outDegrees
        .reduce{(x, y) => if (x._2 > y._2) x else y}
    println(s"outDegress = $outDegress")
```

```
    println("********** 入度 **********")
    graph.inDegrees.foreach(println)
    val inDegress: (VertexId, Int) = graph.inDegrees
      .reduce{(x, y) => if (x._2 > y._2) x else y}
    println(s"inDegress = $inDegress")

    println("********** 度数 **********")
    graph.degrees.foreach(println)
    val degress: (VertexId, Int) = graph.degrees
        .reduce{(x, y) => if (x._2 > y._2) x else y}
    println(s"degress = $degress")

    // 转换操作
    // 顶点的转换操作。所有人的年龄加 10 岁
    graph.mapVertices{case (id, (name, age)) => (id, (name,
age+10))}
        .vertices
        .foreach(println)

    // 边的转换操作。边的属性*2
    graph.mapEdges(e => e.attr*2)
        .edges
        .foreach(println)

    // 结构操作
    // 顶点年龄 > 30 的子图
     val subGraph: Graph[(String, Int), Int] =
graph.subgraph(vpred = (id, vd) => vd._2 >= 30)
    println("*************** 子图 ***************")
    subGraph.edges.foreach(println)
    subGraph.vertices.foreach(println)

    // 连接操作
    println("********************** 连接操作
**********************")
    // 创建一个新图，顶点VD的数据类型为User，并从graph做类型转换
    val initialUserGraph: Graph[User, Int] = graph.mapVertices {
case (_, (name, age)) => User(name, age, 0, 0) }

    // initialUserGraph与inDegrees、outDegrees 进行 join，修改
inDeg、outDeg
    val userGraph: Graph[User, Int] =
initialUserGraph.outerJoinVertices(initialUserGraph.inDegrees) {
```

```
      case (id, u, inDegOpt) => User(u.name, u.age,
inDegOpt.getOrElse(0), u.outDeg)
    }.outerJoinVertices(initialUserGraph.outDegrees) {
      case (id, u, outDegOpt) => User(u.name, u.age, u.inDeg,
outDegOpt.getOrElse(0))
    }

    userGraph.vertices.foreach(println)

    // 找到 出度=入度 的人员
    userGraph.vertices.filter { case (id, u) => u.inDeg ==
u.outDeg }
        .foreach(println)

    // 聚合操作
    // 找出5到各顶点的最短距离
    val sourceId: VertexId = 5L // 定义源点
    val initialGraph: Graph[Double, Int] = graph.mapVertices((id,
_) => if (id == sourceId) 0.0 else Double.PositiveInfinity)

    val sssp: Graph[Double, Int] =
initialGraph.pregel(Double.PositiveInfinity)(
      // 两个消息来的时候，取它们当中路径的最小值
      (id, dist, newDist) => math.min(dist, newDist),

      // Send Message函数
      // 比较 triplet.srcAttr + triplet.attr和 triplet.dstAttr。如果
小于，则发送消息到目的顶点
      triplet => { // 计算权重
        if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
          Iterator((triplet.dstId, triplet.srcAttr +
triplet.attr))
        } else {
          Iterator.empty
        }
      },

      // mergeMsg
      (a, b) => math.min(a, b) // 最短距离
    )

    println("找出5到各顶点的最短距离")
    println(sssp.vertices.collect.mkString("\n"))

    sc.stop
```

```
  }
}

case class User(name: String, age: Int, inDeg: Int, outDeg: Int)
```

Pregel API

图本身是递归数据结构，顶点的属性依赖于它们邻居的属性，这些邻居的属性又依赖于自己邻居的属性。

所以许多重要的图算法都是迭代的重新计算每个顶点的属性，直到满足某个确定的条件。

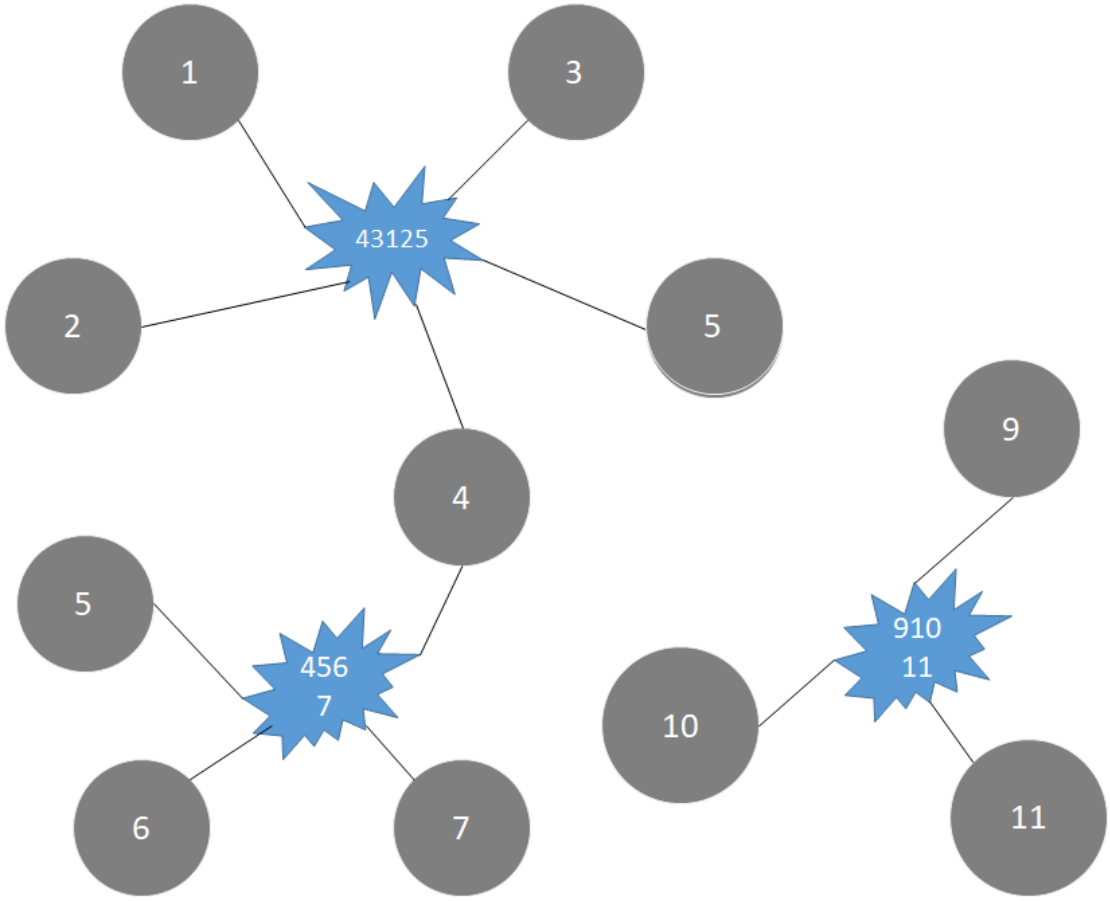
一系列的图并发抽象被提出来用来表达这些迭代算法。

GraphX公开了一个类似Pregel的操作。

```
def pregel[A: ClassTag](
    initialMsg: A,
    maxIterations: Int = Int.MaxValue,
    activeDirection: EdgeDirection = EdgeDirection.Either)(
    vprog: (VertexId, VD, A) => VD,
    sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
    mergeMsg: (A, A) => A)
  : Graph[VD, ED] = {
  Pregel(graph, initialMsg, maxIterations, activeDirection)(vprog, sendMsg, mergeMsg)
}
```

- vprog：用户定义的顶点运行程序。它作用于每一个顶点，负责接收进来的信息，并计算新的顶点值
- sendMsg：发送消息
- mergeMsg：合并消息

案例二：连通图算法



给定数据文件，找到存在的连通体

```
package cn.lagou.graphx

import org.apache.spark.graphx.{Graph, GraphLoader, VertexRDD}
import org.apache.spark.{SparkConf, SparkContext}

object GraphXExample2 {
  def main(args: Array[String]): Unit = {
    val conf: SparkConf = new
SparkConf().setAppName(this.getClass.getCanonicalName)
      .setMaster("local[*]")
    val sc = new SparkContext(conf)
    sc.setLogLevel("warn")

    // 从数据文件中加载，生成图
    val graph: Graph[Int, Int] = GraphLoader.edgeListFile(sc,
"data/graph.dat")
    graph.vertices
        .foreach(println)

    graph.edges
      .foreach(println)
```

```
    // 生成连通图
    graph.connectedComponents()
      .vertices
      .sortBy(_._2)
      .foreach(println)

    sc.stop()
  }
}
```

案例三：寻找相同的用户，合并信息

假设：

- 假设有五个不同信息可以作为用户标识，分别为：1X、2X、3X、4X、5X；
- 每次可以选择使用若干为字段作为标识
- 部分标识可能发生变化，如：12 => 13 或 24 => 25

根据以上规则，判断以下标识是否代表同一用户：

- 11-21-32、12-22-33 (X)
- 11-21-32、11-21-52 (OK)
- 21-32、11-21-33 (OK)
- 11-21-32、32-48 (OK)

问题：在以下数据中，找到同一用户，合并相同用户的数据

- 对于用户标识(id)：合并后去重
- 对于用户的信息：key相同，合并权重

```
List(11L, 21L, 31L), List("kw$北京" -> 1.0, "kw$上海" -> 1.0,
"area$中关村" -> 1.0)
List(21L, 32L, 41L), List("kw$上海" -> 1.0, "kw$天津" -> 1.0,
"area$回龙观" -> 1.0)
List(41L), List("kw$天津" -> 1.0, "area$中关村" -> 1.0)

List(12L, 22L, 33L), List("kw$大数据" -> 1.0, "kw$spark" -> 1.0,
"area$西二旗" -> 1.0)
List(22L, 34L, 44L), List("kw$spark" -> 1.0, "area$五道口" -> 1.0)
List(33L, 53L), List("kw$hive" -> 1.0, "kw$spark" -> 1.0, "area$西
二旗" -> 1.0)
```

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.graphx.{Edge, Graph, VertexId, VertexRDD}
import org.apache.spark.rdd.RDD

object GraphXExample3 {
  def main(args: Array[String]): Unit = {
    // 初始化
    val conf: SparkConf = new
SparkConf().setAppName("GraphXDemo").setMaster("local")
    val sc = new SparkContext(conf)
    sc.setLogLevel("error")

    // 定义数据
    val dataRDD: RDD[(List[Long], List[(String, Double)])] =
sc.makeRDD(List(
      (List(11L, 21L, 31L), List("kw$北京" -> 1.0, "kw$上海" ->
1.0, "area$中关村" -> 1.0)),
      (List(21L, 32L, 41L), List("kw$上海" -> 1.0, "kw$天津" ->
1.0, "area$回龙观" -> 1.0)),
      (List(41L), List("kw$天津" -> 1.0, "area$中关村" -> 1.0)),
      (List(12L, 22L, 33L), List("kw$大数据" -> 1.0, "kw$spark" ->
1.0, "area$西二旗" -> 1.0)),
      (List(22L, 34L, 44L), List("kw$spark" -> 1.0, "area$五道口"
-> 1.0)),
      (List(33L, 53L), List("kw$hive" -> 1.0, "kw$spark" -> 1.0,
"area$西二旗" -> 1.0))
    ))

    // 1 将标识信息中的每一个元素抽取出来，作为id
    // 备注1、这里使用了flatMap，将元素压平；
```

```
    // 备注2、这里丢掉了标签信息，因为这个RDD主要用于构造顶点、边，tags信息用不
    // 备注3、顶点、边的数据要求Long，这个程序修改后才能用在我们的程序中
    val dotRDD: RDD[(VertexId, VertexId)] = dataRDD.flatMap { case (allids, _) =>
      allids.map(id => (id, allids.mkString.hashCode.toLong))
    }

    // 2 定义顶点
    val vertexesRDD: RDD[(VertexId, String)] = dotRDD.map { case (id, _) => (id, "") }

    // 3 定义边(id: 单个的标识信息；ids: 全部的标识信息)
    val edgesRDD: RDD[Edge[Int]] = dotRDD.map { case (id, ids) => Edge(id, ids, 0) }

    // 4 生成图
    val graph = Graph(vertexesRDD, edgesRDD)

    // 5 找到强连通体
    val connectRDD: VertexRDD[VertexId] = graph.connectedComponents()
      .vertices

    // 6 定义中心点的数据
    val centerVertexRDD: RDD[(VertexId, (List[VertexId], List[(String, Double)]))] = dataRDD.map { case (allids, tags) =>
      (allids.mkString.hashCode.toLong, (allids, tags))
    }

    // 7 步骤5、6的数据做join，获取需要合并的数据
    val allInfoRDD = connectRDD.join(centerVertexRDD)
      .map { case (_, (id2, (allIds, tags))) => (id2, (allIds, tags)) }

    // 8 数据聚合（即将同一个用户的标识、标签放在一起）
    val mergeInfoRDD: RDD[(VertexId, (List[VertexId], List[(String, Double)]))] = allInfoRDD.reduceByKey { case ((bufferList, bufferMap), (allIds, tags)) =>
      val newList = bufferList ++ allIds

      // map 的合并
      val newMap = bufferMap ++ tags
      (newList, newMap)
    }
```

```
    // 9 数据合并（allIds：去重；tags：合并权重）
    val resultRDD: RDD[(List[VertexId], Map[String, Double])] =
mergeInfoRDD.map { case (key, (allIds, tags)) =>
      val newIds = allIds.distinct
      // 按照key做聚合；然后对聚合得到的lst第二个元素做累加
      val newTags = tags.groupBy(x => x._1).mapValues(lst =>
lst.map(x => x._2).sum)
      (newIds, newTags)
    }

    resultRDD.foreach(println)

    sc.stop()
  }
}
```