自定义Spark Streaming接收器(Receivers)

Spark Streaming除了可以使用内置的接收器（Receivers，比如Flume、Kafka、Kinesis、files和sockets等）来接收流数据，还可以自定义接收器来从任意的流中接收数据。开发者们可以自己实现org.apache.spark.streaming.receiver.Receiver类来从其他的数据源中接收数据。本文将介绍如何实现自定义接收器，并且在Spark Streaming应用程序中使用。我们可以用Scala或者Java来实现自定义接收器。

实现自定义接收器

一切从实现Receiver开始。所有的自定义接收器必须扩展Receiver抽象类，并且实现其中两个方法：

1、onStart():这个函数主要负责启动数据接收相关的事情；
2、onStop():这个函数主要负责停止数据接收相关的事情。

不管是onStart()和onStop()方法都不可以无限期地阻塞。通常情况下，onStart()会启动相关线程负责接收数据，而onStop()会保证接收数据的线程被终止。接收数据的线程也可以使用Receiver类提供的isStopped()方法来检测是否可以停止接收数据。

数据一旦被接收，这些数据可以通过调用store(data)方法存储在Spark中，store(data)方法是由Receiver类提供的。Receiver类提供了一系列的store()方法，使得我们可以把一条一条地或者多条记录存储在Spark中。值得注意的是，使用某个store()方法实现接收器将会影响其可靠性和容错性，这将在后面详细地讨论。

接收线程中可能会出现任何异常，这些异常都需要被捕获，并且恰当地处理来避免接收器挂掉。restart()将通过异步地调用onStop()和onStart()

方法来重启接收器。stop()将会调用onStop()方法并且中止接收器。同样， reportError()将会向Driver发送错误信息（这些错误信息可以在logs和UI中看到），而不停止或者重启接收器。

下面是一个从socket接收文本的自定义接收器。它将₩n作为一条记录的标志，并且将这些数据存储在Spark中，如果这些接收线程在连接或者接收数据的过程中出现错误，接收器将重启并尝试再次连接，详细的代码如下：

```
//////////////////////////////////////////////////////////////////////
 User: 过往记忆
 Date: 2016年03月03日
 Time: 22:52:23
 bolg:
 本文地址：/archives/1594
 过往记忆博客，专注于hadoop、hive、spark、shark、flume的技术博客，大量的干货
 过往记忆博客微信公共帐号：iteblog_hadoop
//////////////////////////////////////////////////////////////////////

class CustomReceiver(host: String, port: Int)
  extends Receiver[String](StorageLevel.MEMORY_AND_DISK_2) with Logging {

  def onStart() {
    // Start the thread that receives data over a connection
    new Thread("Socket Receiver") {
      override def run() { receive() }
    }.start()
  }

  def onStop() {
   // There is nothing much to do as the thread calling receive()
   // is designed to stop by itself if isStopped() returns false
  }

  /** Create a socket connection and receive data until receiver is stopped */
  private def receive() {
    var socket: Socket = null
    var userInput: String = null
    try {
     // Connect to host:port
     socket = new Socket(host, port)

     // Until stopped or connection broken continue reading
     val reader = new BufferedReader(new InputStreamReader(socket.getInputStream(), "UTF-8"
))
     userInput = reader.readLine()
     while(!isStopped && userInput != null) {
       store(userInput)
```

```
      userInput = reader.readLine()
    }
    reader.close()
    socket.close()

    // Restart in an attempt to connect again when server is active again
    restart("Trying to connect again")
   } catch {
    case e: java.net.ConnectException =>
      // restart if could not connect to server
      restart("Error connecting to " + host + ":" + port, e)
    case t: Throwable =>
      // restart if there is any other error
      restart("Error receiving data", t)
   }
  }
}
```

在Spark Streaming应用程序中使用自定义接收器

自定义的接收器可以通过使用streamingContext.receiverStream()方法来在Spark Streaming应用程序中使用。这将使用自定义接收器接收到的数据来创建input DStream，如下：

```
// Assuming ssc is the StreamingContext
val lines = ssc.receiverStream(new CustomReceiver(host, port))
val words = lines.flatMap(_.split(" "))
...
```

完整的代码如下：

```
package org.apache.spark.examples.streaming

import java.io.{BufferedReader, InputStream, InputStreamReader}
import java.net.Socket

import org.apache.spark.{Logging, SparkConf}
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.receiver.Receiver
```

```
/**
 * Custom Receiver that receives data over a socket. Received bytes is interpreted as
 * text and ₩n delimited lines are considered as records. They are then counted and printed.
 *
 * To run this on your local machine, you need to first run a Netcat server
 *    `$ nc -lk 9999`
 * and then run the example
 *    `$ bin/run-
example org.apache.spark.examples.streaming.CustomReceiver localhost 9999`
 */
object CustomReceiver {
  def main(args: Array[String]) {
    if (args.length < 2) {
      System.err.println("Usage: CustomReceiver <hostname> <port>")
      System.exit(1)
    }

    StreamingExamples.setStreamingLogLevels()

    // Create the context with a 1 second batch size
    val sparkConf = new SparkConf().setAppName("CustomReceiver")
    val ssc = new StreamingContext(sparkConf, Seconds(1))

    // Create a input stream with the custom receiver on target ip:port and count the
    // words in input stream of ₩n delimited text (eg. generated by 'nc')
    val lines = ssc.receiverStream(new CustomReceiver(args(0), args(1).toInt))
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
    wordCounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}


class CustomReceiver(host: String, port: Int)
  extends Receiver[String](StorageLevel.MEMORY_AND_DISK_2) with Logging {

  def onStart() {
    // Start the thread that receives data over a connection
    new Thread("Socket Receiver") {
      override def run() { receive() }
    }.start()
  }

  def onStop() {
```

```
  // There is nothing much to do as the thread calling receive()
  // is designed to stop by itself isStopped() returns false
 }

 /** Create a socket connection and receive data until receiver is stopped */
 private def receive() {
  var socket: Socket = null
  var userInput: String = null
  try {
   logInfo("Connecting to " + host + ":" + port)
   socket = new Socket(host, port)
   logInfo("Connected to " + host + ":" + port)
   val reader = new BufferedReader(new InputStreamReader(socket.getInputStream(), "UTF-8"
))
   userInput = reader.readLine()
   while(!isStopped && userInput != null) {
    store(userInput)
    userInput = reader.readLine()
   }
   reader.close()
   socket.close()
   logInfo("Stopped receiving")
   restart("Trying to connect again")
  } catch {
   case e: java.net.ConnectException =>
    restart("Error connecting to " + host + ":" + port, e)
   case t: Throwable =>
    restart("Error receiving data", t)
  }
 }
}
```

接收器的可靠性

基于接收器的可靠性和容错性可以将接收器分为两大类：

1、可靠的接收器：对于可靠的数据源，它可以对发送出的数据进行确认，可靠的接收器可以正确地通知数据源数据已经被接收并且已经可靠地存储到Spark中。通常情况下，实现这类接收器需要谨慎地考虑数据源确认语义。

2、不可靠的接收器：不可靠的接收器并不发送确认信息到数据源。这可以在哪些不支持确认的数据源中使用，甚至在可靠的数据源中使用。

为了实现可靠的接收器，你必须使用store(multiple-records)来存储数据。调用这个store方法将会阻塞，只有当所有的数据都被存储到Spark里面才会返回。如果这个接收器配置的存储级别使用了副本机制（默认就启用了），这种情况下只有所有的副本都存储完成store才会返回。这样才能保证数据被可靠地保存了，然后接收器才可以通知数据源数据已经被接收。这可以保证接收器在接收到一半数据的情况下也不会丢失数据，因为接收器不会通知数据源，因此数据源可以再次发送这些数据。

不可靠的接收器不需要实现这些逻辑。它仅仅从数据源接收数据并且调用store(single-record)方法来一条一条地将数据存储在Spark中。因为它并没有实现store(multiple-records)方法的可靠性保证，所以它有以下几点优势：

1、系统来考虑如何恰当地将数据生成块；
2、如果速度限制被指定，那么系统来考虑如何控制数据的接收速率；
3、因为上面2点优势，实现不可靠的接收器比可靠的接收器要简单地多。

下面对两类接收器的各自特点进行总结：

1、不可靠的接收器：（1）、比较容易实现；（2）、系统来提供块的生成和速率控制；不需要提供容错保证，但是可能在接收器挂掉的情况下丢失数据。
2、可靠的接收器：（1）、需要提供强容错保证，能够保证零数据丢失；（2）、块生成和速率必须在接收器中实现；（3）、实现复杂度依赖于数据源确认机制的复杂度。

实现并使用基于Actor的自定义接收器

用户自定义的Akka Actor同样可以被用于数据的接收。ActorHelper trait可以使用在任何的Akka actor中，并且可以使用store(...)方法把接收到的数据存储在Spark中。actor的监管策略做相关的配置来处理异常等，如下：

```
//////////////////////////////////////////////////////////////////////
 User: 过往记忆
 Date: 2016年03月03日
 Time: 22:52:23
 bolg:
 本文地址：/archives/1594
 过往记忆博客，专注于hadoop、hive、spark、shark、flume的技术博客，大量的干货
 过往记忆博客微信公共帐号：iteblog_hadoop
//////////////////////////////////////////////////////////////////////

class CustomActor extends Actor with ActorHelper {
  def receive = {
    case data: String => store(data)
  }
}
```

然后使用下面的方法来使用这个自定义的actor：

```
// Assuming ssc is the StreamingContext
val lines = ssc.actorStream[String](Props(new CustomActor()), "CustomReceiver")
```

完整的代码如下：

```
package org.apache.spark.examples.streaming

import scala.collection.mutable.LinkedHashSet
import scala.reflect.ClassTag
import scala.util.Random

import akka.actor._
import com.typesafe.config.ConfigFactory

import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.akka.{ActorReceiver, AkkaUtils}

case class SubscribeReceiver(receiverActor: ActorRef)
case class UnsubscribeReceiver(receiverActor: ActorRef)

/**
 * Sends the random content to every receiver subscribed with 1/2
 *  second delay.
 */
class FeederActor extends Actor {

  val rand = new Random()
  val receivers = new LinkedHashSet[ActorRef]()

  val strings: Array[String] = Array("words ", "may ", "count ")

  def makeMessage(): String = {
    val x = rand.nextInt(3)
    strings(x) + strings(2 - x)
  }

  /*
   * A thread to generate random messages
   */
  new Thread() {
```

```
    override def run() {
      while (true) {
        Thread.sleep(500)
        receivers.foreach(_ ! makeMessage)
      }
    }
  }.start()

  def receive: Receive = {
    case SubscribeReceiver(receiverActor: ActorRef) =>
      println("received subscribe from %s".format(receiverActor.toString))
      receivers += receiverActor

    case UnsubscribeReceiver(receiverActor: ActorRef) =>
      println("received unsubscribe from %s".format(receiverActor.toString))
      receivers -= receiverActor
  }
}

/**
 * A sample actor as receiver, is also simplest. This receiver actor
 * goes and subscribe to a typical publisher/feeder actor and receives
 * data.
 *
 * @see [[org.apache.spark.examples.streaming.FeederActor]]
 */
class SampleActorReceiver[T](urlOfPublisher: String) extends ActorReceiver {

  lazy private val remotePublisher = context.actorSelection(urlOfPublisher)

  override def preStart(): Unit = remotePublisher ! SubscribeReceiver(context.self)

  def receive: PartialFunction[Any, Unit] = {
    case msg => store(msg.asInstanceOf[T])
  }

  override def postStop(): Unit = remotePublisher ! UnsubscribeReceiver(context.self)

}

/**
 * A sample feeder actor
 *
 * Usage: FeederActor <hostname> <port>
 *   <hostname> and <port> describe the AkkaSystem that Spark Sample feeder would start on
.
```

```
 */
object FeederActor {

  def main(args: Array[String]) {
    if (args.length < 2){
      System.err.println("Usage: FeederActor <hostname> <port>₩n")
      System.exit(1)
    }
    val Seq(host, port) = args.toSeq

    val akkaConf = ConfigFactory.parseString(
      s"""akka.actor.provider = "akka.remote.RemoteActorRefProvider"
         |akka.remote.enabled-transports = ["akka.remote.netty.tcp"]
         |akka.remote.netty.tcp.hostname = "$host"
         |akka.remote.netty.tcp.port = $port
         |""".stripMargin)
       val actorSystem = ActorSystem("test", akkaConf)
    val feeder = actorSystem.actorOf(Props[FeederActor], "FeederActor")

    println("Feeder started as:" + feeder)

    actorSystem.awaitTermination()
  }
}

/**
 * A sample word count program demonstrating the use of plugging in
 *
 * Actor as Receiver
 * Usage: ActorWordCount <hostname> <port>
 *   <hostname> and <port> describe the AkkaSystem that Spark Sample feeder is running on.
 *
 * To run this example locally, you may run Feeder Actor as
 *    `$ bin/run-example org.apache.spark.examples.streaming.FeederActor localhost 9999`
 * and then run the example
 *    `$ bin/run-
example org.apache.spark.examples.streaming.ActorWordCount localhost 9999`
 */
object ActorWordCount {
  def main(args: Array[String]) {
    if (args.length < 2) {
      System.err.println(
        "Usage: ActorWordCount <hostname> <port>")
      System.exit(1)
    }
```

```
    StreamingExamples.setStreamingLogLevels()

    val Seq(host, port) = args.toSeq
    val sparkConf = new SparkConf().setAppName("ActorWordCount")
    // Create the context and set the batch size
    val ssc = new StreamingContext(sparkConf, Seconds(2))

    /*
     * Following is the use of AkkaUtils.createStream to plug in custom actor as receiver
     *
     * An important point to note:
     * Since Actor may exist outside the spark framework, It is thus user's responsibility
     * to ensure the type safety, i.e type of data received and InputDStream
     * should be same.
     *
     * For example: Both AkkaUtils.createStream and SampleActorReceiver are parameterized
     * to same type to ensure type safety.
     */
    val lines = AkkaUtils.createStream[String](
      ssc,
      Props(classOf[SampleActorReceiver[String]],
        "akka.tcp://test@%s:%s/user/FeederActor".format(host, port.toInt)),
      "SampleReceiver")

    // compute wordcount
    lines.flatMap(_.split("₩₩s+")).map(x => (x, 1)).reduceByKey(_ + _).print()

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

本文翻译自：《Spark Streaming Custom Receivers》：http://spark.apache.org/docs/latest/streaming-custom-receivers.html