### Spark Streaming工作原理

Spark Core

RDD

sc =new SparkContext(conf)

Spark SQL

DataFrame/DataSet

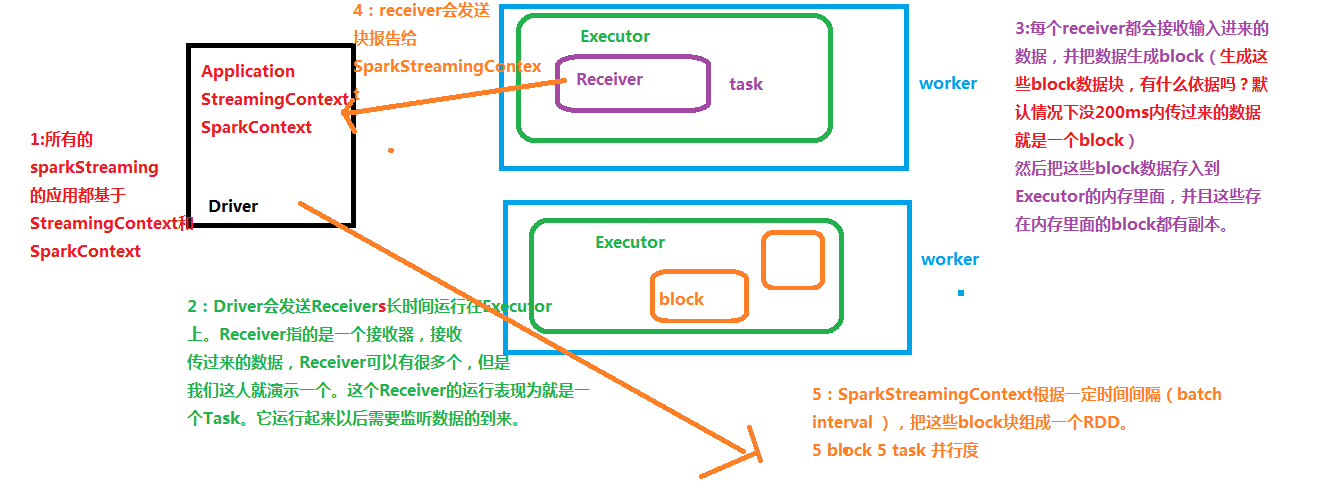
sqlContext=new SQLContext(sc) /HiveContext

Spark Streaming

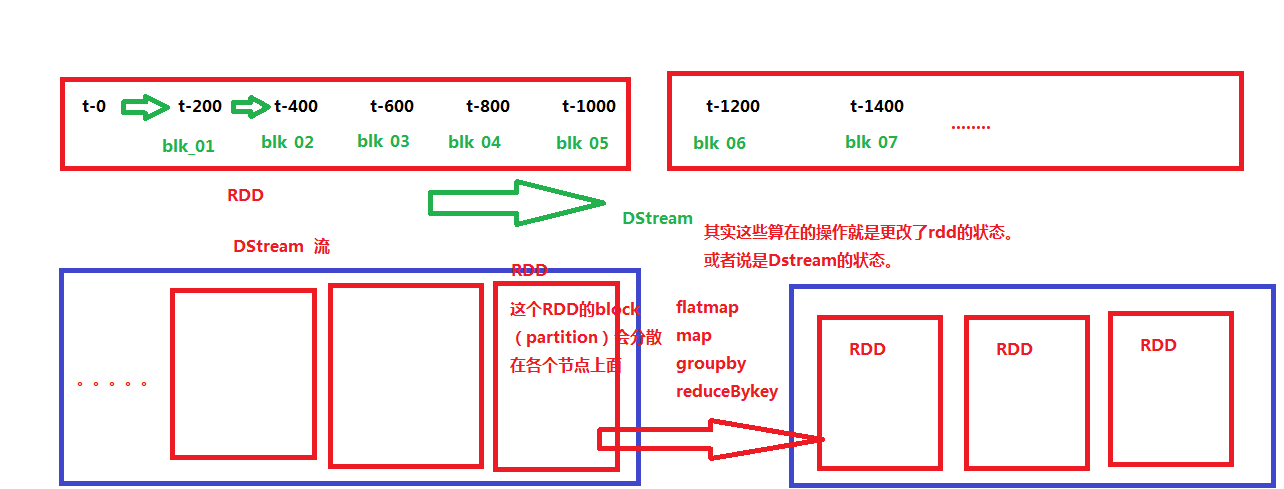
DStream

ssc=new StreamingContext(sc,**Seconds(1)**);

SparkStreaming的运行的原理



DStream分割的原理



Block interval

默认是200ms ，这个值是可以通过配置区修改的(针对一个receiver)

Batch inverval

这个值的就是我们每次处理多长时间范围内的数据。那么实在new StreamingContext的时候就指定。

### 入门案例演示

Spark Streaming is an extension of the core Spark API that **enables scalable, high-throughput, fault-tolerant** stream processing of live data streams

添加pom.xml文件

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-streaming\_2.10</artifactId>

<version>1.6.2</version>

</dependency>

首先需要安装一个nc工具。Yum install -y nc 即可

object NetWordCount {

def main(args: Array[String]): Unit = {

/\*\*

\* local[1] 中括号里面的数字都代表的是启动几个工作线程

\* 默认情况下是一个工作线程。那么做为sparkstreaming 我们至少要开启

\* 两个线程，因为其中一个线程用来接收数据，这样另外一个线程用来处理数据。

\*/

val conf=new SparkConf().setMaster("local[2]").setAppName("NetWordCount")

/\*\* Seconds 指的是每次数据数据的时间范围 （bacth interval）\*/

val ssc=new StreamingContext(conf,Seconds(2))

val fileDS=ssc.socketTextStream("hadoop1", 9999)

val wordcount=fileDS.flatMap { line => line.split("\t") }

.map { word => (word,1) }

.reduceByKey(\_+\_)

/\*\*

\* 打印RDD里面前十个元素 \*/

wordcount.print()

ssc.start() //启动应用

ssc.awaitTermination() //等待任务结束

}

}

### Streaming VS Mapreduce VS Storm

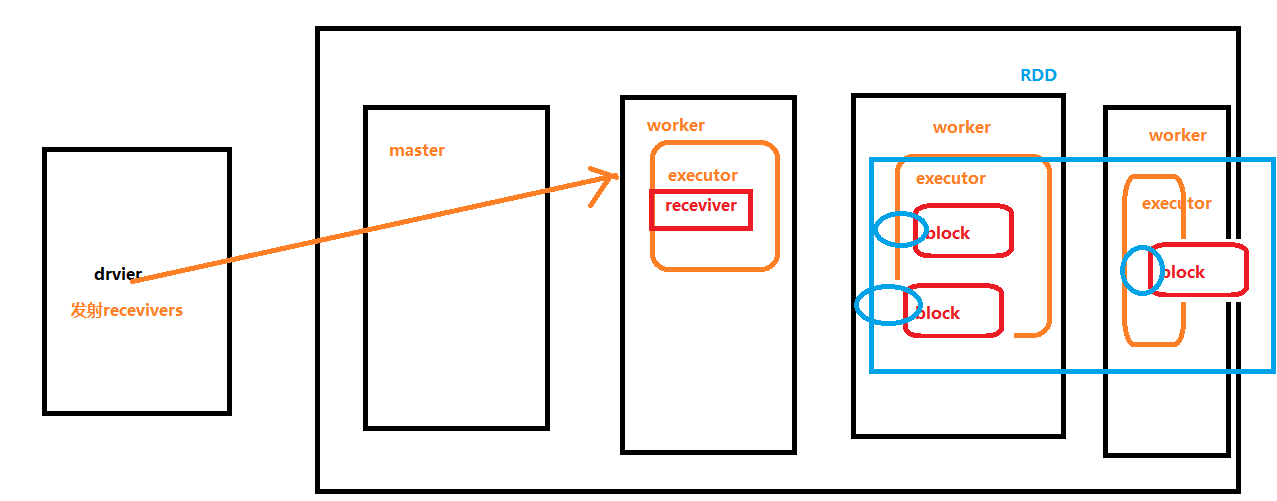
SparkStreaming VS Mapreduce：

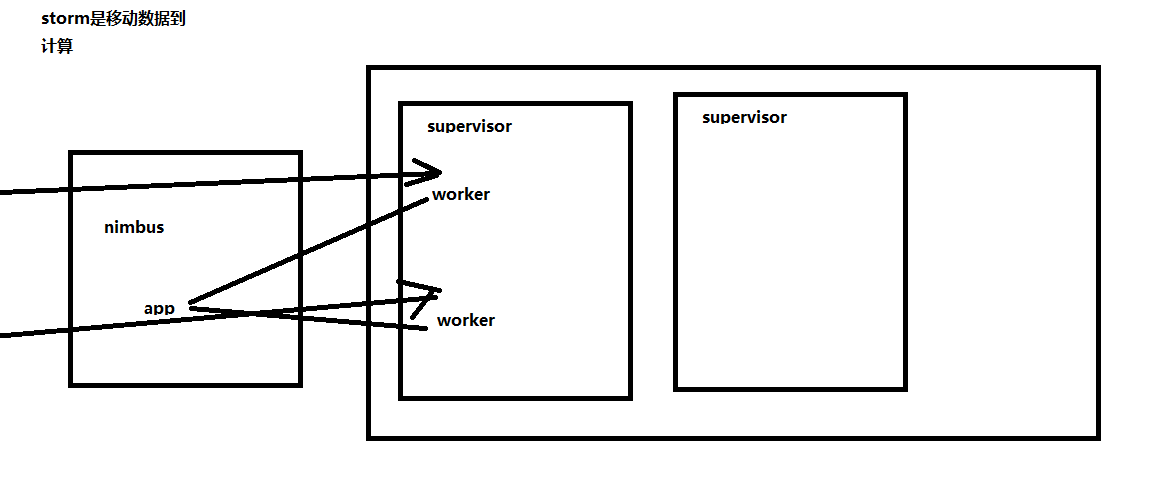
Spark Streaming是一个**准实时**流处理框架，而Hadoop MR是一个离线、批处理框架，主要处理的是**历史的**数据；很显然，在数据的实时性角度，Spark Streaming优于Hadoop MR。

SparkStreaming VS Storm：

1. 数据处理方式

Spark streaming是构建在spark上的实时流计算框架，利用时间批量窗口生成spark的计算输入源RDD，后对该RDD生成Job，进行排队调度到spark计算框架中执行，底层是基于spark资源调度和任务计算框架的；Spark streaming是基于数据的批处理方式，针对数据形成任务进行计算，**是移动计算而不移动数据，而Storm恰恰相反**，storm在处理[**架构**](http://lib.csdn.net/base/16)上是数据流入到计算节点，移动的是数据而不是计算，对于时间窗口的批量数据处理，需要用户自己来实现，这个在之前的storm系列的相关章节中有介绍。





B、 生态体系

Spark streaming是基于spark的，可以和spark其他的组件结合，可以跟spark core,SparkSQL,图计算，机器学习，进行无缝的切换，实现交互式的查询adhoc，[**机器学习**](http://lib.csdn.net/base/2)MLib等。Storm相对来讲，只是作为一个流式计算框架，缺乏现有的[**Hadoop**](http://lib.csdn.net/base/20)生态体系的融合。

C、 延迟以及吞吐量

Spark streaming基于对批量数据的处理，准实时的，依赖spark的调度和计算框架，在延迟方面比storm要高，一般最小的延迟在2s左右，而storm可以达到100ms以内，storm是真正意义上的纯实时的流式处理框架。正因为spark streaming是批处理的方式处理数据，整体的吞吐量比较高。

D、 容错性

Spark streaming通过lineage（RDD 血统 ）以及在内存维护两份数据备份进行容错，通过lineage记录之前对RDD的操作，若某节点在运行时候出现故障，则可以通过备份数据在其他节点重新计算得到。

Storm通过ack组件进行数据流的跟踪，开销比sparking streaming要大。

E、  事务性

Spark streaming保证数据只被处理一次，并且是在批处理的层次级别。

Storm通过跟踪机制能保证每个记录至少被处理一次，如果需要保证状态只更新一次的话，需要由用户自己来实现。

所以对于statefull的计算，对事务性比较高的话，spark streaming要更好一些。

在流式计算方面Storm和spark streaming 是平分天下。其实是各有千秋。

### Spark streaming-- UpdateStateBykey

之前简单的WordCount例子不保存历史数据，都是只统计当前数据的条数。UpdateStateBykey把统计数据保存到中间状态，可以接着上一次的结果继续计算。

报错：（需要设置一个checpoint保存中间结果到目录，而不是内存）

Exception in thread "main" java.lang.IllegalArgumentException: requirement failed: The checkpoint directory has not been set. Please set it by StreamingContext.checkpoint().

解决：需要设置保存中间变量

package com.xtwy.streaming

import org.apache.spark.streaming.StreamingContext

import org.apache.spark.SparkConf

import org.apache.spark.streaming.Seconds

object NetWordCountUpdateStateByKey {

def main(args: Array[String]): Unit = {

/\*\*

\* local[1] 中括号里面的数字都代表的是启动几个工作线程

\* 默认情况下是一个工作线程。那么做为sparkstreaming 我们至少要开启

\* 两个线程，因为其中一个线程用来接收数据，这样另外一个线程用来处理数据。

\*/

val conf=new SparkConf().setMaster("local[2]").setAppName("NetWordCountUpdateStateByKey")

/\*\*

\* Seconds 指的是每次数据数据的时间范围 （bacth interval）

\*/

val ssc=new StreamingContext(conf,Seconds(2));

val fileDS=ssc.socketTextStream("hadoop1", 9999)

/\*\*

\* 需要设置一个checkpoint的目录

\* 因为我们的计算结果有中间状态，这些中间状态需要存储

\*/

ssc.checkpoint(".")

val wordDS=fileDS.flatMap { line => line.split("\t") }

.map { word => (word,1) }

/\*\*

\* updateFunc: (Seq[Int], Option[S]) => Option[S]

\* updateFunc 这是一个匿名函数

\* (Seq[Int], Option[S]) 两个参数

\* Option[S] 返回值

\* 首先我们考虑一个问题

\* wordDS 做的bykey的计算，说明里面的内容是tuple类型，是键值对的形式，说白了是不是

\* 就是【K V】

\* wordDS[K,V]

\* (Seq[Int], Option[S])

\* 参数一：Seq[Int] Seq代表的是一个集合，int代表的是V的数据类型

\* ---分组的操作，key相同的为一组 (hadoop,{1,1,1,1})

\* 参数二：Option[S] S代表的是中间状态State的数据类型，S对于我们的这个wordcount例子来讲，应该是

\* int类型。中间状态存储的是单词出现的次数。 hadoop -> 4

\*

\* Option[S] 返回值 应该跟中间状态一样吧。

\* Option Some/None

\*

\*/

val wordcountDS=wordDS.updateStateByKey((values:Seq[Int],state:Option[Int]) =>{

val currentCount= values.sum; //获取此次本单词出现的次数

val count=state.getOrElse(0);//获取上一次的结果 也就是中间状态

Some(currentCount+count);

})

wordcountDS.print()

//启动应用

ssc.start()

//等待任务结束

ssc.awaitTermination()

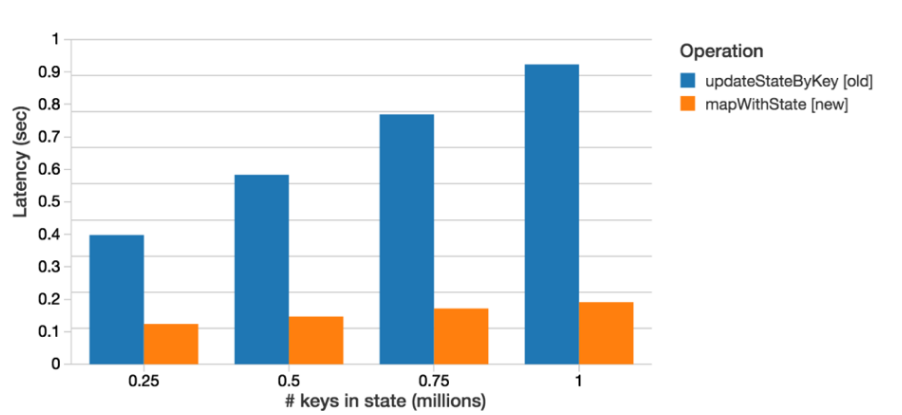
}

}

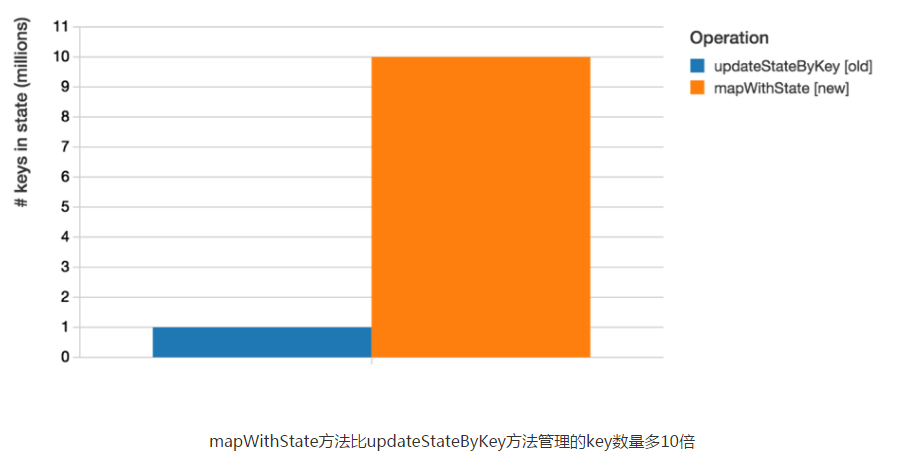
### mapWithState（比updateStateByKey更高级）

与updateStateByKey方法相比，使用mapWithState方法能够得到6倍的低延迟同时维护的key状态数量要多10倍，这一性能提升和扩展性可从后面的基准测试结果得到验证，所有的结果全部在时间间隔为1秒的batch和相同大小的集群中生成。

下图比较的是mapWithState 方法和updateStateByKey 方法处理1秒的batch所消耗的平均时间，在本例中，我们为同样数量（从0.25~1百万）的key保存其状态，然后以同样的速率（30k个更新/s）对其进行更新，如下图所示，mapWithState方法比updateStateByKey方法的处理时间快8倍，从而允许更低的端到端延迟。



此外，更快的处理速度使得mapWithState 方法能够比updateStateByKey 方法管理多10倍的key（批处理间隔、集群大小、更新频率全部相同）。



package com.xtwy.streaming

import org.apache.spark.streaming.StreamingContext

import org.apache.spark.SparkConf

import org.apache.spark.streaming.Seconds

import org.apache.spark.streaming.StateSpec

import org.apache.spark.streaming.State

object MapWithStateDemo {

def main(args: Array[String]): Unit = {

/\*\*

\* local[1] 中括号里面的数字都代表的是启动几个工作线程

\* 默认情况下是一个工作线程。那么做为sparkstreaming 我们至少要开启

\* 两个线程，因为其中一个线程用来接收数据，这样另外一个线程用来处理数据。

\*/

val conf=new SparkConf().setMaster("local[2]").setAppName("MapWithStateDemo")

/\*\*

\* Seconds 指的是每次数据数据的时间范围 （bacth interval）

\*/

val ssc=new StreamingContext(conf,Seconds(2));

ssc.checkpoint(".")

val fileDS=ssc.socketTextStream("hadoop1", 9999)

val wordDstream =fileDS.flatMap { line => line.split("\t") }

.map { word => (word,1) }

/\*\*

\* word: String, one: Option[Int], state: State[Int]

\* 这个函数里面有三个参数

\* 第一个参数：word: String 代表的就是key

\* 第二个参数：one: Option[Int] 代表的就是value

\* 第三个参数：state: State[Int] 代表的就是状态（历史状态，也就是上次的结果）

\*

\* hello,4

\*

\* hello,1

\*

\* hello,5

\*/

val mappingFunc = (word: String, one: Option[Int], state: State[Int]) => {

val sum = one.getOrElse(0) + state.getOption.getOrElse(0)

val output = (word, sum)

state.update(sum)

output

}

val initialRDD = ssc.sparkContext.parallelize(List(("hello", 1), ("world", 1)))

/\*\*

\* hello,1

\* hello,2

\* world,2

\*/

val stateDstream = wordDstream.mapWithState(

StateSpec.function(mappingFunc).initialState(initialRDD)) //可加入一个初始RDD,作为初始值再累加

/\*\*

\* 打印RDD里面前十个元素

\*/

// wordcount.print()

stateDstream.print();

//启动应用

ssc.start()

//等待任务结束

ssc.awaitTermination()

}

}

# transform操作

transform操作，应用在DStream上时，可以用于执行任意的RDD到RDD的转换操作。它可以用于实现，DStream API中所没有提供的操作。比如说，DStream API中，并没有提供将一个DStream中的每个batch，与一个特定的RDD进行join的操作。但是我们自己就可以使用transform操作来实现该功能。

码表RDD

val fileDS=ssc.socketTextStream("hadoop1", 9999);

fileDS.transform( rdd ->

Rdd.join(码表RDD)

)

黑名单过滤，当然这个是来源于一个真实的项目。只不过为了讲课方面，所以需求经过了改装，但是处理问题的代码思路都是一样的。

单词计数，但是单词里面，不需要统计 ？ ， ！ 。也就是说

在单词计数的时候过滤黑名单里面的标点符号。

package com.xtwy.streaming

import org.apache.spark.streaming.StreamingContext

import org.apache.spark.SparkConf

import org.apache.spark.streaming.Seconds

object TransformaDemo {

def main(args: Array[String]): Unit = {

val conf=new SparkConf().setMaster("local[2]").setAppName("TransformaDemo")

val ssc=new StreamingContext(conf,Seconds(5));

val fileDS=ssc.socketTextStream("192.168.32.110", 9999);

val wordcountDS=fileDS.flatMap { line => line.split("\t") }

.map { word => (word,1) }

val fillter=ssc.sparkContext.parallelize(List(",","?","!",".")).map { param => (param,true) }

val needwordDS= wordcountDS.transform( rdd =>{

val leftRDD= rdd.leftOuterJoin(fillter)

//leftRDD [String,(int,option[boolean])]

val needword=leftRDD.filter( tuple =>{

val x= tuple.\_1;

val y=tuple.\_2;

if(y.\_2.isEmpty){

true;

}else{

false;

}

})

needword.map(tuple =>(tuple.\_1,1))

})

val wcDS= needwordDS.reduceByKey(\_+\_);

wcDS.print();

ssc.start();

ssc.awaitTermination();

}

}

# Windows操作

SparkStreaming 的用途(流式计算的应用场景)

-1,实时统计，累加（淘宝的大屏）

kafka + sparkstreaming(updateStatByKey,mapwithState)

-2,实时统计，最近一段时间指标

实时查看最近一个小时之内的用户点击量，各省或者重点城市

Window 窗口

**reduceByKeyAndWindow**

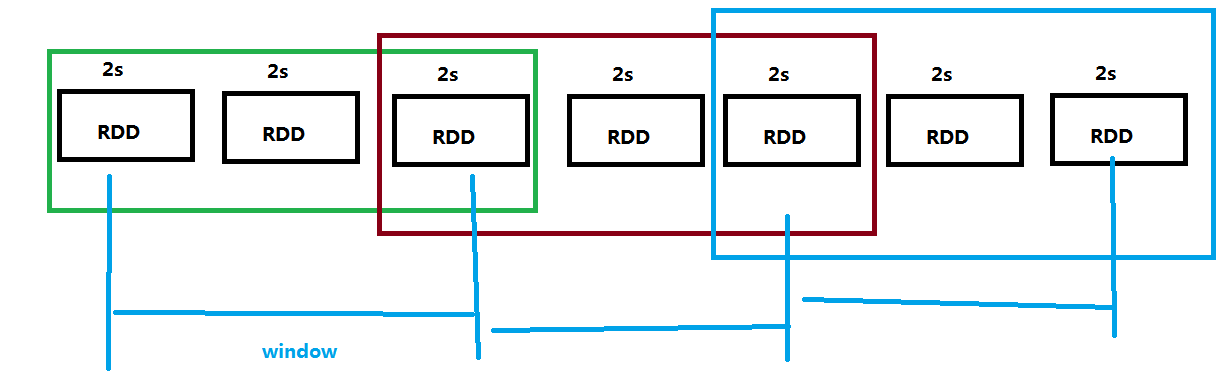
**案例：每隔4秒实时统计前6秒的单词计数情况.(每2秒生成一个RDD)**

**注：如上（2）必须是（4和6的倍数）**

**val** conf=**new** SparkConf().setMaster("local[2]").setAppName("TransformaDemo")

**val** ssc=**new** StreamingContext(conf,Seconds(2));

**val** fileDS=ssc.socketTextStream("192.168.32.110", 9999);



object reduceByKeyAndWindowDemo {

def main(args: Array[String]): Unit = {

val conf=new SparkConf().setMaster("local[2]").setAppName("TransformaDemo")

val ssc=new StreamingContext(conf,Seconds(2));

val fileDS=ssc.socketTextStream("192.168.32.110", 9999);

val wordcountDS=fileDS.flatMap { line => line.split("\t") }

.map { word => (word,1) } //(\_+\_)

.reduceByKeyAndWindow((x:Int,y:Int) =>{x+ y},Seconds(6),Seconds(4))

wordcountDS.print();

ssc.start();

ssc.awaitTermination();

/\*

\* hadoop hadoop 不在统计范围之内

\* hadoop hadoop

\* hadoop hadoop hadoop,6

\* hadoop hadoop hadoop,6

\* hadoop hadoop

\* hadoop hadoop

\*

\* \*/

}

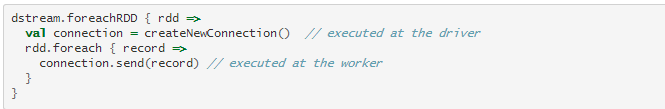
}

### Spark Streaming之foreachRDD

通常在foreachRDD中，都会创建一个Connection，比如JDBC Connection，然后通过Connection将数据写入外部存储。

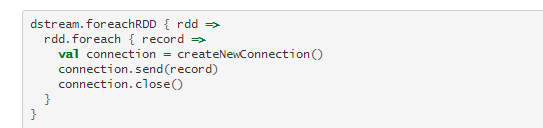
**误区一**：在RDD的foreach操作外部，创建Connection

这种方式是错误的，因为它会导致Connection对象被序列化后传输到每个Task中。而这种Connection对象，实际上一般是不支持序列化的，也就无法被传输。

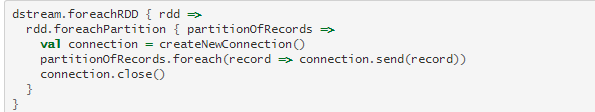


**误区二**：在RDD的foreach操作内部，创建Connection

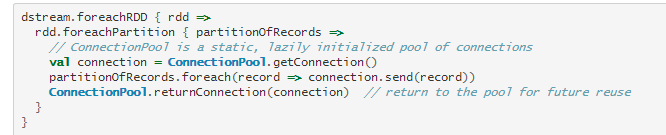
这种方式是可以的，但是效率低下。因为它会导致对于RDD中的每一条数据，都创建一个Connection对象。而通常来说，Connection的创建，是很消耗性能的。



使用方式一：



使用方式二：（创建一个连接池，最完美的方案）



<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.6</version>

</dependency>

做实时的单词计数，然后把单词计数结果存进mysql数据库。

1：首先创建好连接池

2：开发代码把统计结果存进去。

create table wordcount(countTime datetime ,word varchar(30),count int);

package com.xtwy.streaming

import org.apache.spark.streaming.StreamingContext

import org.apache.spark.SparkConf

import org.apache.spark.streaming.Seconds

object ForEachRDDOperation {

def main(args: Array[String]): Unit = {

val conf=new SparkConf().setMaster("local[2]").setAppName("ForEachRDDOperation")

val ssc=new StreamingContext(conf,Seconds(2));

val fileDS=ssc.socketTextStream("192.168.32.110", 9999);

val wordcountDS=fileDS.flatMap { line => line.split("\t") }

.map { word => (word,1) }

.reduceByKey(\_+\_)

wordcountDS.foreachRDD( partitionOfRecords =>{

partitionOfRecords.foreachPartition( records =>{

val connect= MysqlPool.getJdbcCoon();

while(records.hasNext){

val tuple= records.next();

val sql="insert into wordcount values( now(),'"+tuple.\_1+"', "+tuple.\_2.toInt+")";

val statement=connect.createStatement();

statement.executeUpdate(sql);

print(sql);

}

MysqlPool.releaseConn(connect)

})

})

ssc.start()

ssc.awaitTermination()

}

}

package com.xtwy.streaming

import java.sql.DriverManager

import java.sql.Connection

object MysqlPool {

private val max=8 ;//连接池的连接总数

private val connectionNum=10;//每次产生的连接数

private var conNum=0;//当前连接池已经产生的连接数

import java.util

private val pool=new util.LinkedList[Connection]();//连接池

{

Class.forName("com.mysql.jdbc.Driver")

}

/\*\*

\* 释放连接

\*/

def releaseConn(conn:Connection):Unit={

pool.push(conn);

}

/\*\*

\* 获取连接

\*/

def getJdbcCoon():Connection={

//同步代码块

AnyRef.synchronized({

if(pool.isEmpty()){

for( i <- 1 to connectionNum){

val conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/xtwy","root","root");

pool.push(conn);

conNum+1;

}

}

pool.poll();

})

}

}