# RDD

## shared variables 共享参数

Spark supports two types of shared variables: *broadcast variables*, which can be used to cache a value in memory on all nodes, and *accumulators*, which are variables that are only “added” to, such as counters and sums.

broadcast variables 类似于在各个机器上传输一个一致的不被修改的副本【方便broadcast到新的节点】

accumulator 类似保存一个一致的参数，其他excutor都可以修改，但是最终的修改都要作用在这个唯一的参数上 结构类似于，一个准确的variable，其他的是variable\_stage1、variable\_stage2…..这些都会反馈给variable【允许节点在保证一致性的情况下安全地处理参数】

## Broadcast variables

keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. In “shuffle” operations, Spark automatically broadcasts the common data needed by tasks within each stage.

explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

**val** broadcastVar **=** sc.broadcast(**Array**(1, 2, 3))

broadcastVar.value

so that v is not shipped to the nodes more than once.

the object v should not be modified after it is broadcast in order to ensure that all nodes get the same value of the broadcast variable (e.g. if the variable is shipped to a new node later).

## closures --》 Accumulators

Accumulators in Spark are used specifically to provide a mechanism for safely updating a variable when execution is split up across worker nodes in a cluster.

**var** counter **=** 0

**var** rdd **=** sc.parallelize(data)

*// Wrong: Don't do this!!*

rdd.foreach(x **=>** counter += x)

println("Counter value: " + counter)

单机和分布式可能会得到不同的结果。因为如果是在不同的JVM中执行是，counter会一直是0，因为在分布式集群中：The variables within the closure sent to each executor are now copies and thus, when **counter** is referenced within the foreach function, it’s no longer the **counter** on the driver node. There is still a **counter** in the memory of the driver node but this is no longer visible to the executors! The executors only see the copy from the serialized closure. Thus, the final value of **counter** will still be zero since all operations on **counter** were referencing the value within the serialized closure.

例子：

**val** accum **=** sc.longAccumulator("My Accumulator")

sc.parallelize(**Array**(1, 2, 3, 4)).foreach(x **=>** accum.add(x))

accum.value //10

While this code used the built-in support for accumulators of type Long, programmers can also create their own types by subclassing [AccumulatorV2](http://spark.apache.org/docs/2.3.0/api/scala/index.html#org.apache.spark.util.AccumulatorV2). 【可以使用built-in的long类型的accumulator，也可以通过AccumulatorV2创建自己的类】

**class** **VectorAccumulatorV2** **extends** **AccumulatorV2**[MyVector, MyVector] {

**private** **val** myVector**:** MyVector = **MyVector**.createZeroVector

**def** reset()**:** Unit = {

myVector.reset()

}

**def** add(v**:** MyVector)**:** Unit = {

myVector.add(v)

}

...

}

*// Then, create an Accumulator of this type:*

**val** myVectorAcc **=** **new** **VectorAccumulatorV2**

*// Then, register it into spark context:*

sc.register(myVectorAcc, "MyVectorAcc1")

其他例子：

**val** accum **=** sc.longAccumulator

data.map { x **=>** accum.add(x); x }

*// Here, accum is still 0 because no actions have caused the map operation to be computed.*

## JVM

Only one SparkContext may be active per JVM. You must stop() the active SparkContext before creating a new one.

## parallelize  并行化

SparkContext’s parallelize method:

**val** data **=** **Array**(1, 2, 3, 4, 5)

**val** distData **=** sc.parallelize(data)

we might call distData.reduce((a, b) => a + b)to add up the elements of the array

Normally, Spark tries to set the number of partitions automatically based on your cluster. However, you can also set it manually by passing it as a second parameter to parallelize (e.g. sc.parallelize(data, 10)).

## File Format 文件格式

Spark supports text files, [SequenceFiles](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/mapred/SequenceFileInputFormat.html), and any other Hadoop [InputFormat](http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapred/InputFormat.html)

Some notes on reading files with Spark:

If using a path on the local filesystem, the file must also be accessible at the same path on worker nodes. Either copy the file to all workers or use a network-mounted shared file system.

All of Spark’s file-based input methods, including textFile, support running on directories, compressed files, and wildcards as well. For example, you can use textFile("/my/directory"), textFile("/my/directory/\*.txt"), and textFile("/my/directory/\*.gz").

The textFile method also takes an optional second argument for controlling the number of partitions of the file. By default, Spark creates one partition for each block of the file (blocks being 128MB by default in HDFS), but you can also ask for a higher number of partitions by passing a larger value. Note that you cannot have fewer partitions than blocks.

Apart from text files, Spark’s Scala API also supports several other data formats:

SparkContext.wholeTextFiles lets you read a directory containing multiple small text files, and returns each of them as (filename, content) pairs. This is in contrast with textFile, which would return one record per line in each file. Partitioning is determined by data locality which, in some cases, may result in too few partitions. For those cases, wholeTextFiles provides an optional second argument for controlling the minimal number of partitions.

For [SequenceFiles](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/mapred/SequenceFileInputFormat.html), use SparkContext’s sequenceFile[K, V] method where K and V are the types of key and values in the file. These should be subclasses of Hadoop’s [Writable](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/Writable.html) interface, like [IntWritable](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/IntWritable.html) and [Text](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/Text.html). In addition, Spark allows you to specify native types for a few common Writables; for example, sequenceFile[Int, String] will automatically read IntWritables and Texts.

For other Hadoop InputFormats, you can use the SparkContext.hadoopRDD method, which takes an arbitrary JobConf and input format class, key class and value class. Set these the same way you would for a Hadoop job with your input source. You can also use SparkContext.newAPIHadoopRDD for InputFormats based on the “new” MapReduce API (org.apache.hadoop.mapreduce).

RDD.saveAsObjectFile and SparkContext.objectFile support saving an RDD in a simple format consisting of serialized Java objects. While this is not as efficient as specialized formats like Avro, it offers an easy way to save any RDD.

## 例子

**val** lines **=** sc.textFile("data.txt")

**val** lineLengths **=** lines.map(s **=>** s.length)

**val** totalLength **=** lineLengths.reduce((a, b) **=>** a + b)

The first line defines a base RDD from an external file. This dataset is not loaded in memory or otherwise acted on: lines is merely a pointer to the file. The second line defines lineLengths as the result of a map transformation. Again, lineLengths is *not* immediately computed, due to laziness. Finally, we run reduce, which is an action. At this point Spark breaks the computation into tasks to run on separate machines, and each machine runs both its part of the map and a local reduction, returning only its answer to the driver program.

If we also wanted to use lineLengths again later, we could add:

lineLengths.persist()

before the reduce, which would cause lineLengths to be saved in memory after the first time it is computed.

## Passing Functions to Spark

* [Anonymous function syntax](http://docs.scala-lang.org/tour/basics.html#functions), which can be used for short pieces of code.

例1、val addOne = (x: Int) => x + 1

println(addOne(1)) // 2

例2、val add = (x: Int, y: Int) => x + y

println(add(1, 2)) // 3

* Static methods in a global singleton object. For example, you can define object MyFunctions and then pass MyFunctions.func1, as follows:
* **object** **MyFunctions** {
* **def** func1(s**:** String)**:** String = { ... }
* }
* myRdd.map(**MyFunctions**.func1)

accessing fields of the outer object will reference the whole object:

**class** **MyClass** {

**val** field **=** "Hello"

**def** doStuff(rdd**:** RDD[String])**:** RDD[String] **=** { rdd.map(x **=>** field + x) }

}

is equivalent to writing rdd.map(x => this.field + x), which references all of this.

【会把第一段的定义也传到cluster里，而其实并不需要。因此如非必要，尽量做到各段功能解耦】 To avoid this issue, the simplest way is to copy field into a local variable instead of accessing it externally:

**def** doStuff(rdd**:** RDD[String])**:** RDD[String] **=** {

**val** field\_ **=** **this**.field

rdd.map(x **=>** field\_ + x)

}

## Print

rdd.foreach(println) or rdd.map(println) ，in cluster mode, the output to stdout being called by the executors is now writing to the executor’s stdout instead, not the one on the driver, so stdout on the driver won’t show these! You can use:

rdd.collect().foreach(println)

This can cause the driver to run out of memory, though, because collect() fetches the entire RDD to a single machine; if you only need to print a few elements of the RDD, a safer approach is to use the take(): rdd.take(100).foreach(println).

## Key-Value Pairs

“shuffle” operations are only available on RDDs of key-value pairs.

uses the reduceByKey operation on key-value pairs to count how many times each line of text occurs in a file:

**val** lines **=** sc.textFile("data.txt")

**val** pairs **=** lines.map(s **=>** (s, 1))

**val** counts **=** pairs.reduceByKey((a, b) **=>** a + b)

We could also use counts.sortByKey(), for example, to sort the pairs alphabetically, and finally counts.collect() to bring them back to the driver program as an array of objects.

## Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc ([Scala](http://spark.apache.org/docs/2.3.0/api/scala/index.html#org.apache.spark.rdd.RDD), [Java](http://spark.apache.org/docs/2.3.0/api/java/index.html?org/apache/spark/api/java/JavaRDD.html), [Python](http://spark.apache.org/docs/2.3.0/api/python/pyspark.html#pyspark.RDD), [R](http://spark.apache.org/docs/2.3.0/api/R/index.html)) and pair RDD functions doc ([Scala](http://spark.apache.org/docs/2.3.0/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions), [Java](http://spark.apache.org/docs/2.3.0/api/java/index.html?org/apache/spark/api/java/JavaPairRDD.html)) for details.

|  |  |
| --- | --- |
| **Transformation** | **Meaning** |
| **map**(*func*) | Return a new distributed dataset formed by passing each element of the source through a function *func*. |
| **filter**(*func*) | Return a new dataset formed by selecting those elements of the source on which *func*returns true. |
| **flatMap**(*func*) | Similar to map, but each input item can be mapped to 0 or more output items (so *func*should return a Seq rather than a single item). |
| **mapPartitions**(*func*) | Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T. |
| **mapPartitionsWithIndex**(*func*) | Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T. |
| **sample**(*withReplacement*, *fraction*, *seed*) | Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed. |
| **union**(*otherDataset*) | Return a new dataset that contains the union of the elements in the source dataset and the argument. |
| **intersection**(*otherDataset*) | Return a new RDD that contains the intersection of elements in the source dataset and the argument. |
| **distinct**([*numPartitions*])) | Return a new dataset that contains the distinct elements of the source dataset. |
| **groupByKey**([*numPartitions*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.  **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.  **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numPartitions argument to set a different number of tasks. |
| **reduceByKey**(*func*, [*numPartitions*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. |
| **aggregateByKey**(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. |
| **sortByKey**([*ascending*], [*numPartitions*]) | When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument. |
| **join**(*otherDataset*, [*numPartitions*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin. |
| **cogroup**(*otherDataset*, [*numPartitions*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith. |
| **cartesian**(*otherDataset*) | When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements). |
| **pipe**(*command*, *[envVars]*) | Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings. |
| **coalesce**(*numPartitions*) | Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset. |
| **repartition**(*numPartitions*) | Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network. |
| **repartitionAndSortWithinPartitions**(*partitioner*) | Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery. |

## Actions

The following table lists some of the common actions supported by Spark. Refer to the RDD API doc ([Scala](http://spark.apache.org/docs/2.3.0/api/scala/index.html#org.apache.spark.rdd.RDD), [Java](http://spark.apache.org/docs/2.3.0/api/java/index.html?org/apache/spark/api/java/JavaRDD.html), [Python](http://spark.apache.org/docs/2.3.0/api/python/pyspark.html#pyspark.RDD), [R](http://spark.apache.org/docs/2.3.0/api/R/index.html))

and pair RDD functions doc ([Scala](http://spark.apache.org/docs/2.3.0/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions), [Java](http://spark.apache.org/docs/2.3.0/api/java/index.html?org/apache/spark/api/java/JavaPairRDD.html)) for details.

|  |  |
| --- | --- |
| **Action** | **Meaning** |
| **reduce**(*func*) | Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel. |
| **collect**() | Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data. |
| **count**() | Return the number of elements in the dataset. |
| **first**() | Return the first element of the dataset (similar to take(1)). |
| **take**(*n*) | Return an array with the first *n* elements of the dataset. |
| **takeSample**(*withReplacement*, *num*, [*seed*]) | Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed. |
| **takeOrdered**(*n*, *[ordering]*) | Return the first *n* elements of the RDD using either their natural order or a custom comparator. |
| **saveAsTextFile**(*path*) | Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file. |
| **saveAsSequenceFile**(*path*)  (Java and Scala) | Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc). |
| **saveAsObjectFile**(*path*)  (Java and Scala) | Write the elements of the dataset in a simple format using Java serialization, which can then be loaded usingSparkContext.objectFile(). |
| **countByKey**() | Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key. |
| **foreach**(*func*) | Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an [Accumulator](http://spark.apache.org/docs/2.3.0/rdd-programming-guide.html#accumulators) or interacting with external storage systems.  **Note**: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See [Understanding closures](http://spark.apache.org/docs/2.3.0/rdd-programming-guide.html#understanding-closures-a-nameclosureslinka)for more details. |

## Shuffle

The shuffle is Spark’s mechanism for re-distributing data so that it’s grouped differently across partitions. This typically involves copying data across executors and machines, making the shuffle a complex and costly operation.

The **Shuffle** is an expensive operation since it involves disk I/O, data serialization, and network I/O.

Certain shuffle operations can consume significant amounts of heap memory since they employ in-memory data structures to organize records before or after transferring them. Specifically, reduceByKey and aggregateByKey create these structures on the map side, and 'ByKey operations generate these on the reduce side. When data does not fit in memory Spark will spill these tables to disk, incurring the additional overhead of disk I/O and increased garbage collection.

Shuffle also generates a large number of intermediate files on disk. As of Spark 1.3, these files are preserved until the corresponding RDDs are no longer used and are garbage collected. Garbage collection may happen only after a long period of time, if the application retains references to these RDDs or if GC does not kick in frequently. This means that long-running Spark jobs may consume a large amount of disk space.(数据倾斜会导致GC时间过长，从而使得资源占用时间因个别任务而过长)

See the ‘Shuffle Behavior’ section within the [Spark Configuration Guide](http://spark.apache.org/docs/2.3.0/configuration.html).

## RDD Persistence

Use persist() can

Caching a RDD is a key tool for iterative algorithms and fast interactive use.

each persisted RDD can be stored using a different *storage level*, allowing you, for example, to persist the dataset on disk, persist it in memory but as serialized Java objects (to save space), replicate it across nodes.

The cache() method is a shorthand for using the default storage level, which is StorageLevel.MEMORY\_ONLY (store deserialized objects in memory).

:

|  |  |
| --- | --- |
| **Storage Level** | **Meaning** |
| MEMORY\_ONLY | Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory,  some partitions will not be cached and will be recomputed on the fly each time  they're needed. This is the default level. |
| MEMORY\_AND\_DISK | Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory,  store the partitions that don't fit on disk, and read them from there when they're  needed. |
| MEMORY\_ONLY\_SER  (Java and Scala) | Store RDD as *serialized* Java objects (one byte array per partition). This is generally  more space-efficient than deserialized objects, especially when using a [fast serializer](http://spark.apache.org/docs/2.3.0/tuning.html),  but more CPU-intensive to read. |
| MEMORY\_AND\_DISK\_SER  (Java and Scala) | Similar to MEMORY\_ONLY\_SER, but spill partitions that don't fit in memory to disk  instead of recomputing them on the fly each time they're needed. |
| DISK\_ONLY | Store the RDD partitions only on disk. |
| MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2, etc. | Same as the levels above, but replicate each partition on two cluster nodes. |
| OFF\_HEAP (experimental) | Similar to MEMORY\_ONLY\_SER, but store the data in [off-heap memory](http://spark.apache.org/docs/2.3.0/configuration.html#memory-management). This  requires off-heap memory to be enabled. |

Spark’s storage levels are meant to provide different trade-offs between memory usage and CPU efficiency.

 How to choice ? Following this process:

* If your RDDs fit comfortably with the default storage level (MEMORY\_ONLY), leave them that way. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.
* If not, try using MEMORY\_ONLY\_SER and [selecting a fast serialization library](http://spark.apache.org/docs/2.3.0/tuning.html) to make the objects much more space-efficient, but still reasonably fast to access. (Java and Scala)
* Don’t spill to disk unless the functions that computed your datasets are expensive, or they filter a large amount of the data. Otherwise, recomputing a partition may be as fast as reading it from disk.
* Use the replicated storage levels if you want fast fault recovery (e.g. if using Spark to serve requests from a web application). *All* the storage levels provide full fault tolerance by recomputing lost data, but the replicated ones let you continue running tasks on the RDD without waiting to recompute a lost partition.

## Rmove Data

least-recently-used (LRU) fashion.[ automatically]

use the RDD.unpersist() method.[manually]

# Spark SQL

One use of Spark SQL is to execute SQL queries. Spark SQL can also be used to read data from an existing Hive installation.

The Dataset API is available in [Scala](http://spark.apache.org/docs/2.3.0/api/scala/index.html#org.apache.spark.sql.Dataset) and [Java](http://spark.apache.org/docs/2.3.0/api/java/index.html?org/apache/spark/sql/Dataset.html). Python does not have the support for the Dataset API.

【Spark Sql 可以作为数据探索的工具。也可以作为基于Hive的交互式查询工具替代Impala】

SparkSession in Spark 2.0 provides builtin support for Hive features.

With a SparkSession, applications can create DataFrames from an [existing RDD](http://spark.apache.org/docs/2.3.0/sql-programming-guide.html#interoperating-with-rdds), from a Hive table, or from [Spark data sources](http://spark.apache.org/docs/2.3.0/sql-programming-guide.html#data-sources).

examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala

JDBC To Other Databases，可以使用Spark Sql方便的导入系统，与其他数据入Hive中的数据一起计算

Trouble shooting

* The JDBC driver class must be visible to the primordial class loader on the client session and on all executors. This is because Java’s DriverManager class does a security check that results in it ignoring all drivers not visible to the primordial class loader when one goes to open a connection. One convenient way to do this is to modify compute\_classpath.sh on all worker nodes to include your driver JARs.
* Some databases, such as H2, convert all names to upper case. You’ll need to use upper case to refer to those names in Spark SQL.

# Structured Streaming

streaming aggregations, event-time windows, stream-to-batch joins, etc.

100 milliseconds . **Continuous Processing(spark 2.3)** can achieve end-to-end latencies as low as 1 millisecond with at-least-once guarantees.

# Spark Streaming

sources like Kafka, Flume, Kinesis, or TCP sockets. high-level functions like map, reduce, join and window.  pushed out to filesystems, databases, and live dashboards.



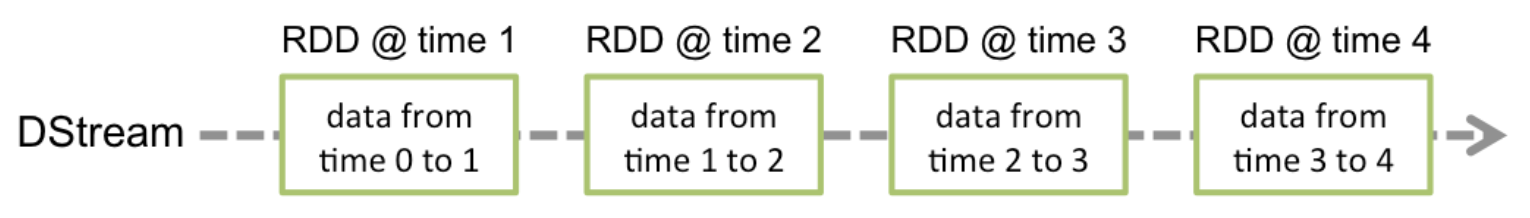
 apply Spark’s [machine learning](http://spark.apache.org/docs/2.3.0/ml-guide.html) and [graph processing](http://spark.apache.org/docs/2.3.0/graphx-programming-guide.html) algorithms on data streams.

After a context is defined, you have to do the following.

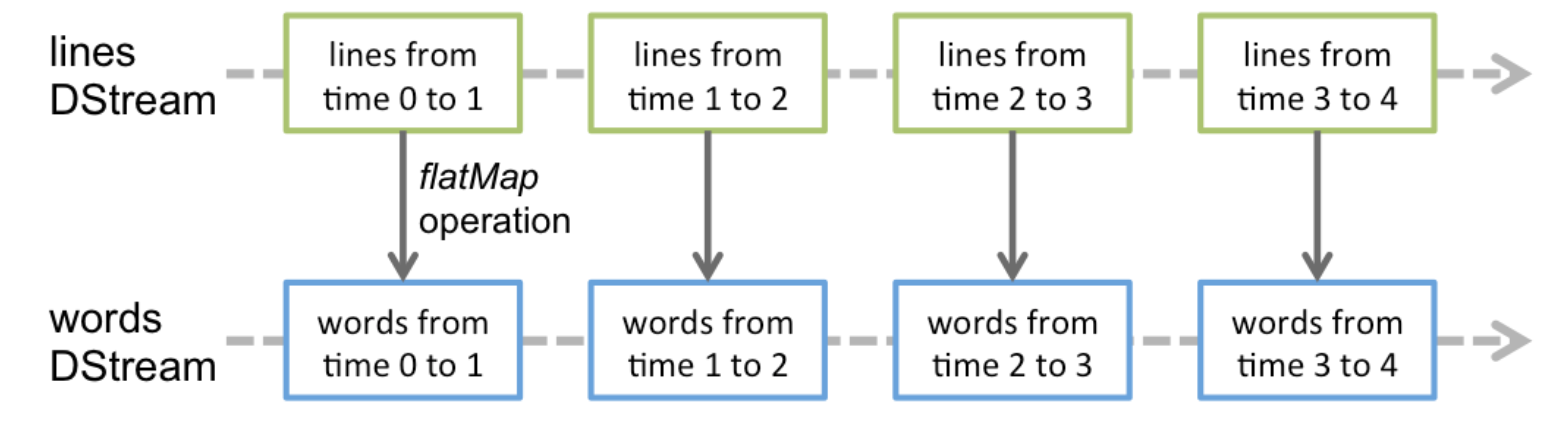
1. Define the input sources by creating input DStreams.
2. Define the streaming computations by applying transformation and output operations to DStreams.
3. Start receiving data and processing it using streamingContext.start().
4. Wait for the processing to be stopped (manually or due to any error) using streamingContext.awaitTermination().
5. The processing can be manually stopped using streamingContext.stop().

Points to remember:

* Once a context has been started, no new streaming computations can be set up or added to it.
* Once a context has been stopped, it cannot be restarted.
* Only one StreamingContext can be active in a JVM at the same time.
* stop() on StreamingContext also stops the SparkContext. To stop only the StreamingContext, set the optional parameter of stop() called stopSparkContext to false.
* A SparkContext can be re-used to create multiple StreamingContexts, as long as the previous StreamingContext is stopped (without stopping the SparkContext) before the next StreamingContext is created.



实例：



**import** **org.apache.spark.\_**

**import** **org.apache.spark.streaming.\_**

**import** **org.apache.spark.streaming.StreamingContext.\_** *// not necessary since Spark 1.3*

*// Create a local StreamingContext with two working thread and batch interval of 1 second.*

*// The master requires 2 cores to prevent a starvation scenario.*

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

**val** ssc **=** **new** **StreamingContext**(conf, **Seconds**(1))

*// Create a DStream that will connect to hostname:port, like localhost:9999*

**val** lines **=** ssc.socketTextStream("localhost", 9999)

*// Split each line into words*

**val** words **=** lines.flatMap(**\_**.split(" "))

**import** **org.apache.spark.streaming.StreamingContext.\_** *// not necessary since Spark 1.3*

*// Count each word in each batch*

**val** pairs **=** words.map(word **=>** (word, 1))

**val** wordCounts **=** pairs.reduceByKey(**\_** + **\_**)

*// Print the first ten elements of each RDD generated in this DStream to the console*

wordCounts.print()

ssc.start() *// Start the computation*

ssc.awaitTermination() *// Wait for the computation to terminate*

You will first need to run Netcat (a small utility found in most Unix-like systems) as a data server by using

$ nc -lk 9999

in a different terminal, you can start the example by using

./bin/run-example streaming.NetworkWordCount localhost 9999

## Basic Sources

**File Streams**

1/ HDFS API

2/simple text files[Directories are Monitored [ changes may be missed, and data omitted from the stream]

**Custom Receivers**

**Queue of RDDs as a Stream**

**Advanced Sources**

Kafka

Flume

**Kinesis**

**Custom Sources**

implement a user-defined **receiver** (see next section to understand what that is) that can receive data from the custom sources and push it into Spark. See the [Custom Receiver Guide](http://spark.apache.org/docs/2.3.0/streaming-custom-receivers.html) for details.

|  |  |
| --- | --- |
| **Transformation** | **Meaning** |
| **map**(*func*) | Return a new DStream by passing each element of the source DStream through a function *func*. |
| **flatMap**(*func*) | Similar to map, but each input item can be mapped to 0 or more output items. |
| **filter**(*func*) | Return a new DStream by selecting only the records of the source DStream on which *func* returns true. |
| **repartition**(*numPartitions*) | Changes the level of parallelism in this DStream by creating more or fewer partitions. |
| **union**(*otherStream*) | Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*. |
| **count**() | Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream. |
| **reduce**(*func*) | Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative and commutative so that it can be computed in parallel. |
| **countByValue**() | When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream. |
| **reduceByKey**(*func*, [*numTasks*]) | When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. **Note:** By default, this uses Spark's defau­lt number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property spark.default.parallelism) to do the grouping. You can pass an optional numTasks argument to set a different number of tasks. |
| **join**(*otherStream*, [*numTasks*]) | When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key. |
| **cogroup**(*otherStream*, [*numTasks*]) | When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples. |
| **transform**(*func*) | Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream. |
| **updateStateByKey**(*func*) | Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key. |
|  |  |

## Transformations on DStreams

Similar to that of RDDs, transformations allow the data from the input DStream to be modified. DStreams support many of the transformations available on normal Spark RDD’s. Some of the common ones are as follows.