Spark
編程指南繁體中文版

TaiwanSparkUserGroup

Published
with GitBook

目錄

Spark 編程指南繁體中文版

========== 如果你是個讀者，這邊有更容易閱讀的Gitbook版本

貢獻方式

請有意願加入的同好參考(https://github.com/TaiwanSparkUserGroup/spark-programming-guide-zh-tw/blob/master/CONTRIBUTING.rst)

大綱

Copyright

License

快速上手

本節課程提供一個使用 Spark 的快速介绍，首先我们使用 Spark 的交互式 shell (用 Python 或 Scala) 介绍它的 API。當示範如何在 Java, Scala 和 Python 寫獨立的程式時，可以參考編程指南裡完整的範例。

依照這個指南，首先從 Spark 網站下載一個 Spark 發行套件。因為我們不會使用 HDFS，你可以下載任何 Hadoop 版本的套件。

- Spark Shell
- 獨立應用程序
- 開始翻滾吧

使用 Spark Shell

基礎

Spark 的 shell 作為一個强大的交互式數據分析工具，提供了一個簡單的方式來學習 API。它可以使用 Scala(在 Java 虛擬機上運行現有的 Java 庫的一个很好方式) 或 Python。在 Spark 目錄裡使用下面的方式開始運行：

```
./bin/spark-shell
```

Spark 最主要的抽象是叫Resilient Distributed Dataset(RDD) 的彈性分布式資料集。RDDs 可以使用 Hadoop InputFormats(例如 HDFS 文件)創建，也可以從其他的 RDDs 轉換。讓我們在 Spark 源代碼目錄從 README 文本文件中創建一個新的 RDD。

```
scala> val textFile = sc.textFile("README.md")
textFile: spark.RDD[String] = spark.MappedRDD@2ee9b6e3
```

RDD 的 actions 從 RDD 中返回值，transformations 可以轉換成一個新 RDD 並返回它的引用。讓我們開始使用幾個操作：

```
scala> textFile.count() // RDD 的數據行數
res0: Long = 126

scala> textFile.first() // RDD 的第一行數據
res1: String = # Apache Spark
```

現在讓我們使用一個 transformation，我們將使用 filter 在這個文件裡返回一個包含子數據集的新 RDD。

```
scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))
linesWithSpark: spark.RDD[String] = spark.FilteredRDD@7dd4af09
```

我們可以把 actions 和 transformations 連接在一起：

```
scala> textFile.filter(line => line.contains("Spark")).count() // 有多少行包括 "Spark"?
res3: Long = 15
```

更多 RDD 操作

RDD actions 和 transformations 能被用在更多的複雜計算中。比方說，我們想要找到一行中最多的單詞數量：

```
scala> textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
res4: Long = 15
```

首先將行映射(map)成一個整數產生一個新 RDD。 在這個新的 RDD 上調用 `reduce` 找到行中最大的個數。 `map` 和 `reduce` 的參數是 Scala 的函數串(closures)，並且可以使用任何語言特性或者 Scala/Java 函式庫。例如，我们可以很方便地調用其他的函數。 我們使用 `Math.max()` 函數讓代碼更容易理解：

```
scala> import java.lang.Math
import java.lang.Math

scala> textFile.map(line => line.split(" ").size).reduce((a, b) => Math.max(a, b))
res5: Int = 15
```

Hadoop 流行的一個通用的數據流(data flow)模式是 MapReduce。Spark 能很容易地實現 MapReduce：

```
scala> val wordCounts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a + b)
wordCounts: spark.RDD[(String, Int)] = spark.ShuffledAggregatedRDD@71f027b8
```

這裡，我們結合 flatMap, map 和 reduceByKey 來計算文件裡每個單詞出現的數量，它的結果是包含一組(String, Int) 鍵值對的 RDD。我们可以使用 [collect] 操作在我們的 shell 中收集單詞的數量：

```
scala> wordCounts.collect()
res6: Array[(String, Int)] = Array((means,1), (under,2), (this,3), (Because,1), (Python,2), (agree,1), (cluster.,1), ..
```

暫存(Caching)

Spark 支持把數據集拉到集群內的暫存記憶體中。當要重複取用資料時非常有用．例如當我们在一個小的熱(hot)數據集中查詢，或者運行一個像網頁搜索排序這樣的迭代演算法。作為一個簡單的例子，讓我们把 `linesWithSpark` 數據集標記在暫存中：

```
scala> linesWithSpark.cache()
res7: spark.RDD[String] = spark.FilteredRDD@17e51082

scala> linesWithSpark.count()
res8: Long = 15

scala> linesWithSpark.count()
res9: Long = 15
```

暫存 100 行的文本文件來研究 Spark 這看起来很傻。真正讓人感興趣的部分是我们可以在非常大型的數據集中使用同樣的函數，甚至在 10 個或者 100 個節點中交叉計算。你同樣可以使用 `bin/spark-shell` 連接到一个 cluster 來繼亂掉编程指南中的方法進行交互操作。

獨立應用程序

下一步？

恭喜你成功運行你的第一個 Spark 應用程式!

- 要深入了解 API，可以從Spark编程指南開始，或者從其他的元件開始，例如：Spark Streaming。
- 要讓程式運行在集群(cluster)上，前往部署概論。
- 最後，Spark 在 `examples` 文件目錄里包含了 Scala, Java 和 Python 的幾個簡單的例子，你可以直接運行它们：

```
# For Scala and Java, use run-example:
./bin/run-example SparkPi

# For Python examples, use spark-submit directly:
./bin/spark-submit examples/src/main/python/pi.py
```

概論

在結構中，每隻 Spark 應用程式都由一隻驅動程式*(driver program)*構成，驅動程序在集群上運行用户的 `main` 函數来執行各式各樣的併行操作*(parallel operations)*。Spark 的主要抽象是提供一個彈性分布式資料庫*(RDD)*，RDD 是指能横跨集群所有節點進行併行計算的分區元素集合。RDDs 從 Hadoop 的文件系统中的一個文件中產生而來(或其他 Hadoop 支持的文件系统)，或者從一個已有的 Scala 集合轉換得到。用戶可以將 Spark RDD 持久化*(persist)*到記憶體中，讓它在併行計算中有效率的被重複使用。而且，RDDs 能在節點失敗中自動恢復。

Spark 的第二個抽象是共享變數*(shared variables)*，共享變數被運行在併行運算中。默認情況下，當 Spark 運行一個併行函數時，這個併行函數會作為一個任務集在不同的節點上運行，它會把函數裡使用到的每個變數都複製移動到每個任務中。有時，一個變數需被共享到交叉任務中或驅動程式和任務之間。Spark 支持 2 種類型的共享變數：廣播變數*(broadcast variables)*，使用在所有節點的記憶體中快取一個值；累加器(accumulators)，只能執行“增加(added)”操作，例如：計數器(counters)和加總(sums)。

這個指南會在 Spark 支持的所有語言中展示它的每一個特性。簡單的操作 Spark 互動式 shell - `bin/spark-shell` 操作 Scala shell，或 `bin/pyspark` 啟動一個 Python shell。

引入 **Spark**

Spark 1.2.0 使用 Scala 2.10 撰寫應用程式，因此你的Scala 版本必須相容(例如：2.10.X)。

撰寫 Spark 應用程式時，你需要添加 Spark 的 Maven 依賴，Spark 可以透過 Maven 中心庫來取得：

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.2.0
```

如果你希望連結 HDFS 集群，需要根據你的 HDFS 版本設定 `hadoop-client` 的相依性。你可以透過第三方發行頁面找到相對應的版本

```
groupId = org.apache.hadoop
artifactId = hadoop-client
version = <your-hdfs-version>
```

最後，你需要匯入一些 Spark 的類別(class) 和隱式轉換 (implicit conversions) 到你的程式，增加下面幾行即可：

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```

初始化 Spark

使用 Spark 的第一步是創建SparkContext 物件，讓 Spark 知道如何找到集群。而建立 `SparkContext` 之前，還需建立 SparkConf 物件，而這個物件則包含你的應用程式資訊。

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
new SparkContext(conf)
```

`appName` 參數是你自定的程式名稱，會顯示在 cluster UI 上。`master` 是Spark, Mesos 或 YARN 集群的 URL，或是運行本地模式(standalone) 時可使用 “local”。實際上，當你的程式在集群上運作時，你不需要把 `master` 放在程式中，因此可以用 spark-submit 啟動你的應用程式。當然，你也可以在 Spark 程式中使用 “local” 做本地測試或是單元測試。

使用 Shell

在 Spark shell 裡，有一個內建的 SparkContext，名為 `sc`。你可以用 `--master` 設定 SparkContext 欲連結的集群，用 `--jars` 來指定需要加到 classpath 中的 JAR 包，倘若有多個 JAR，可使用逗號分隔符號來連結他們。例如：想在一個擁有 4 個CPU 的環境上執行 `bin/spark-shell`：

```
$ ./bin/spark-shell --master local[4]
```

或是想在 classpath 中增加 `code.jar`，你可以這樣寫：

```
$ ./bin/spark-shell --master local[4] --jars code.jar
```

此外，你可以執行 `spark-shell --help` 得到完整的參數列表。目前，使用 `spark-submit` 會比 spark-shell 普遍。

彈性分布式資料集 (RDDs)

Spark 核心概念是 *Resilient Distributed Dataset (RDD)*，你可以將它視為一個可以併型操作、有容錯機制的資料集和。目前有 2 種方式可以建立 RDDs：第一種是在你執行的驅動程式中併行化一個已經存在集合；另外一個方式是引用外部儲存系統的資料集，例如共享文件系統，HDFS，HBase或其他 Hadoop 資料格式的資料來源。

平行集合

建立平行集合 (*Parallelized collections*) 的方法是利用一個已存在的序列(Scala `Seq`)上使用 SparkContext 的 `parallelize`。序列中的元素會被複製到一個併行操作的分布式資料集合裡。例如：如何建立一個包含數字1 到5 的併行集合：

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

當平行集合建立完成，這樣的分布式資料集(`distData`) 就可以被操作。例如，可以使用 `distData.reduce((a, b) => a + b)` 將裡面的元素數字加總。後續會再對這部份進行操作上的介紹。

值得注意，在併行集合裡有一個重要的概念，就是切片數(*slices*)，即一個資料集被切的份數。Spark 會在集群上指派每一個切片執行任務。你也可以在集群上替每個 CPU 設定 2-4 個切片(slices)。一般情況下， Spark 會常識基於你的集群狀態自動設定切片的數量。當然，你也可以利用 `parallelize` 參數的第二個位置做設定，例如 `sc.parallelize(data, 10)`。

外部資料集

Spark 支援任何一個 Hadoop 的文件系統建立分布式資料集，例如，HDFS，Cassandra，HBase，Amazon S3等。此外，Spark 也支援文字文件(text files)，SequenceFiles 和其他 Hadoop InputFormat。

文字文件 RDDs 可以由 SparkContext 的 `textFile` 函數建立。只需要在這函數中標示文件的 URI (機器上的本地路徑或是 `hdfs://`，`s3n://` 等)， Spark 會將文件讀取寫入成一個集合。以下是一個範例：

```
scala> val distFile = sc.textFile("data.txt")
distFile: RDD[String] = MappedRDD@1d4cee08
```

當上述步驟建立完成後，`distFiile` 就可以針對資料集做操作。例如，使用下面的方法使用 `map` 和 `reduce` 加總所有行的長度：`distFile.map(s => s.length).reduce((a, b) => a + b)`。

注意，Spark 讀取資料時：

- 如果使用本地文件系統路徑，文件必須能在 work 節點上同一個路徑中找到，所以你需要複製文件到所有的 workers，或者使用網路的方法分享文件。
- 所有 Spark 內關於連結文件的方法，除了 `textFile`，還有針對壓縮過的檔案，例如 `textFile("/my/文件目錄")`，`textFile("/my/文件目錄/*.txt")` 和 `textFile("/my/文件目錄/*.gz")`。
- `textFile` 函數中，有第二個選擇參數來設定切片(*slices*) 的數目。預設情況下，Spark 為每一塊文件(HDFS 預設文件大小是 64M) 建立一個切片(*slice*)。但是你也可以修改成一個更大的值來設定一個更高的切片數目。值得注意，你不能設定一個小於文件數目的切片值。

除了文本文件，Spark 的 Scala API 支持其他幾種資料格式：

- `SparkContext.sholeTextFiles` 可以讀取一個包含多個檔案的文件目錄，並且返回每一個(filename, content)。和 `textFile` 的差異是：它記錄的是每一個檔案中的每一行。
- 關於 SequenceFiles，你可以使用 SparkContext 的 `sequenceFile[K, V]` 方法產生，K 和 V 分別就是 key 和 values 。像 IntWritable 與 Text，他們必須是Hadoop 的Writable 的子類。另外對於幾種通用的 Writables，Spark 允許你指定原本類型來替代。例如： `sequenceFile[Int, String]` 會自動讀取 IntWritables 和 Text。

- 至於其他的 Hadoop InputFormats，可以使用 `SparkContext.hadoopRDD` 方法，它可以指定任意的 `JobConf`，輸入格式(InputFormat)，key 類型，values 類型。你可以跟設定 Hadoop job 一樣的方法設定輸入來源。你還可以在新的的 MapReduce 接口(org.apache.hadoop.mapreduce) 基礎上使用 `SparkContext.newAPIHadoopRDD` (譯者提醒：原生的接口是 `SparkContext.newHadoopRDD`)。

- `RDD.saveAsObjectFile` 和 `SparkContext.objectFile` 可以保存一個RDD，保存格式是一個簡單的 Java 物件序列化格式。這是一種效率不高的特有格式，如 Avro，它提供簡單的方法來保存任何一個 RDD。

RDD 操作

RDDs 支持兩種類型的操作：轉換*(transformations)* 從已經存在的資料集裡面建立一個新的資料集：動作*(actions)* 在資料集上做運算之後返回一個值到驅動程式。例如， `map` 是一個轉換操作，它會將每一個資料集內的元素傳遞給函數並且返回新的 RDD。另一方面， `reduce` 是一個動作，它會使用相同的函數來對 RDD 內所有的元素做整合，並且將最後的結果送回驅動程式(不過也有一個函數 `reduceByKey` 功能是返回一個分布式資料集)。

在 Spark 中，所有的转换(transformations)都是惰性(lazy)的，它们不会马上计算它们的结果。相反的，它们仅仅记录转换操作是应用到哪些基础数据集(例如一个文件)上的。转换仅仅在这个时候计算：当动作(action) 需要一个结果返回给驱动程序的时候。这个设计能够让 Spark 运行得更加高效。例如，我们可以实现：通过 `map` 创建一个新数据集在 `reduce` 中使用，并且仅仅返回 `reduce` 的结果给 driver，而不是整个大的映射过的数据集。

預設情況，每一個轉換過後的 RDD 會在每次執行動作(action) 的時候重新計算。當然，你也可以使用 `persist` (或 `cache`) 方式做持久化(`persist`) 一個 RDD 到緩存裡。在這情況之下， Spark 會在集群上保存相關的訊息，在你下次查詢時，縮短運行時間，同時也支援 RDD 持久化到硬碟，或是在其他節點上做複製動作。

基礎

為了理解 RDD 基本知識，可以先了解下面的簡單程式：

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

第一行是定義外部文件的 RDD。這個資料集並沒有收到緩存或做其他的操作： `lines` 單單只是一個指向文件位置的動作。第二行是定義 `lineLengths` ，它是 `map` 轉換(transformation) 的結果。同樣，因為 Spark 的 lazy 模式， `lineLengths` 不會被立刻計算。最後，當我們執行 `reduce` ，因為這是一個動作(action)， Spark 會把計算分成多個任務(task)，並且讓他們運作在多個機器上。每台機器都會執行自己的 map 或是本地 reduce ，最後將結果送回給驅動程式。

如果想再次使用 `lineLengths` ，我們可以這樣做：

```
lineLengths.persist()
```

在 `reduce` 之前，它會把 `lineLengths` 在第一次運算結束後將內容保存到緩存中，以利後續再次使用。

傳遞函數到 Spark

Spark 的 API 大多數是依靠在驅動程式裡傳遞函數到集群上運作，目前有兩種推薦方式：

- 匿名函數 (Anonymous function syntax)，可在較短的程式碼中使用。
- 全局單例物件裡的靜態方法。例如，定義 `object MyFunctions` 然後傳遞 `MyFounctions.func1`，例如：

```
object MyFunctions {
  def func1(s: String): String = { ... }
}

myRdd.map(MyFunctions.func1)
```

注意，它可能傳遞的是一個類別實例裡面的一個方法(非一個單例物件)，在這裡，必須傳送包含方法的整個物件。例如：

```
class MyClass {
  def func1(s: String): String = { ... }
  def doStuff(rdd: RDD[String]): RDD[String] = { rdd.map(func1) }
}
```

如果我們建立一個 `new MyClass` 物件，並且使用它的 `doStuff`， `map` 裡面引用了這個 `MyClass` 中的 `func1` 方法，所以這個物件必須也傳送到集群上。類似寫成 `rdd.map(x => this.func1(x))`。

以同樣的方式，存取外部物件的變數將會引用整個物件：

```
class MyClass {
  val field = "Hello"
  def doStuff(rdd: RDD[String]): RDD[String] = { rdd.map(x => field + x) }
}
```

等於寫成 `rdd.map(x => this.field + x)`，引用整個 `this` 物件。為了避免這個問題，最簡單的方式是複製 `field` 到一個本地變數而不是從外部取用：

```
def doStuff(rdd: RDD[String]): RDD[String] = {
  val field_ = this.field
  rdd.map(x => field_ + x)
}
```

使用鍵值對

即使很多 Spark 的操作視在任意類型物件的 RDDs 上，仍有少數幾個特殊操作僅在鍵值(key-value) 對 RDDs 上使用。最常見的是分布式 "shuffle" 操作，例如根據一個 key 對一組資料進行分組跟整合。

在 Scala 中，這些操作包含二元組(Tuple2)(在語言的內建元祖中，簡單的寫 (a, b) 即可建立) 的 RDD 上自動變成可用元素，只要在你的程式中匯入 `org.apache.spark.SparkContext._` 來啟動 Spark 的隱式轉換(implicit conversions)。在 PairRDDFunctions 的類別鍵值對操作中可以使用，如果你匯入隱式轉換，它會自動包成元組 (tuple) RDD。

舉例，下面的程式在鍵值對上使用 `reduceByKey` 來統計一個文件裡面每一行內容出現的次數：

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

也可以使用 `counts.sortByKey()`，例如，將鍵值對按照字母做排序，最後用 `counts.collect()` 輸出成結果陣列送回驅動程式。

注意：使用一個自行定義物件作為 key 在使用鍵值對操作的時候，需要確保定義的 `equals()` 方法和 `hashCode()` 方法是匹配的。細節請看 Object.hashCode() 文件) 內的描述。

Transformations

下面的表格列了 Sparkk 支援且常用 transformations。細節請參考 RDD API 手冊(Scala, Java, Python) 和 PairRDDFunctions 文档(Scala, Java)。

Transformation	Meaning
map(*func*)	回傳一個新的分布式資料集，將資料來源的每一個元素傳遞給函數 *func* 映射組成。
filter(*func*)	回傳一個新的資料集，從資料來源中篩選元素通過函數 *func* 回傳 true。
flatMap(*func*)	類似 map，但是每個輸入元素能被映射成多個輸出選項(所以 *func* 必須回傳一個 Seq，而非單一 item)。
mapPartitions(*func*)	類似 map，但分別運作在 RDD 的每個分區上，火以 *func* 的類型必須為 `Iterator<T> => Iterator<U>` 當執行在類型是 T 的 RDD 上。
mapPartitionsWithIndex(*func*)	類似 mapPartitions，但 *func* 需要提供一個 integer 值描述索引(index)，所以 *func* 的類型必須是 (Int, Iterator) => Iterator 當執行在類型為 T 的 RDD 上。
sample(withReplacement, fraction, seed)	對資料做抽樣。
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([numTasks]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or combineByKey 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 numTasks argument to set a different number of tasks.
reduceByKey(func, [numTasks])	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, [numTasks])	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], [numTasks])	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, [numTasks])	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 also supported through leftOuterJoin and rightOuterJoin.
cogroup(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Iterable, Iterable) 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.
	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.

Actions

下面的表格列出 Sparkk 支援且常用的 actions。細節請參考 RDD API 手冊(Scala, Java, Python) 和 PairRDDFunctions 手冊(Scala, Java)。

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. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
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 either 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 using SparkContext.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 variable (see below) or interacting with external storage systems.

RDD 永續儲存

Spark 最重要的一個功能是它可以通過各種操作（operations）永續儲存（或者緩存）一個集合到記憶體中。當你想儲存一個 RDD 的時候，每一個節點都會參與計算的所有分區資料儲存到記憶體中，而且這些資料可以被這個集合（以及這個集合衍生出的其他集合）的動作（ action ）來重複使用。這樣的設計會使後續的動作速度加快（通常快10 倍以上）。對迭代算法與快速的交互操作來說，記憶體是一個關鍵點。

你可以用 `persist()` 或 `cache()` 方法來儲存一個RDD。首先，在action 中運算後取得RDD；接著，將它保存在每個節點的記憶體裡。Spark 的緩存是一個容錯的機制-如果RDD 的任何一個分區內不見，它可以透過原本的（transformations ）操作，自動重複計算並且建立到這個分區內補足。

此外，可以利用不同的儲存機制來儲存每一個被永續化的 RDD 。例如，Spark 允許使用者將RDD 儲存在硬碟上、將集合序列化的 Java 物件永續儲存到記憶體、在節點之間複製或是儲存到Tachyon中。我們可以傳遞 `StorageLevel` 物件給 `persist()` 方法來設定這些選項。 `cache()` 方法則是預設儲存位置為— `StorageLevel.MEMORY_ONLY` 。完整的設定說明如下：

Storage Level	Meaning
MEMORY_ONLY	把RDD 作為非序列化的Java 物件儲存在在jvm中。如果RDD 不適合放在記憶體，一些分區將不會被儲存在記憶體內，而是在每次需要在分區內時重新計算。這是系統預設的儲存方式。
MEMORY_AND_DISK	將RDD 作為非序列化的Java 物件儲存在jvm中。如果RDD 不適合放在記憶體，將這些分區儲存在硬碟，需要再取出。
MEMORY_ONLY_SER	將RDD 作為序列化的Java 對象儲存（每個分區一個byte 數組）。這種方式比非序列化方式更加節省空間，特別是用快速序列化方式，只是更耗費cpu 資源—密集的讀取操作。
MEMORY_AND_DISK_SER	和MEMORY_ONLY_SER类似，但不是在每次需要时重复计算这些不适合存储到内存中的分区，而是将这些分区存储到磁盘中。
DISK_ONLY	僅僅將RDD 分區儲存在硬碟內
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	和上面的儲存方式類似，但是複製每個分區到集群的兩個節點上面
OFF_HEAP (experimental)	以序列化的格式儲存RDD 到Tachyon中。相較於MEMORY_ONLY_SER，OFF_HEAP 減少回收垃圾的耗損，允許更小的執行者分享記憶體。這使得在擁有大量記憶體的環境下或者多開發空間的環境中更具威力。

NOTE:在python中，儲存的物件都是通過Pickle 做序列化，所以是否選擇序列化並不重要。

Spark 也會自動永續儲存一些shuffle 操作（如 `reduceByKey` ）的中間資料，即使用戶沒有使用 `persist` 方法。好處是避免在 shuffle 發生錯誤情況下，需要重新計算整個輸入。如果使用者計畫重算過程中的RDD ， 建議使用 `persist` 。

如何選擇儲存方式

Spark 提供多種儲存方式意味在記憶體利用率和 cpu 利用率之間的平衡。我們推薦透過下列的步驟選擇一個合適的儲存方式：

- 如果你的RDD 適合預設的儲存方式（MEMORY_ONLY），就使用預設方式。因為這是cpu 利用率最好的的選擇，RDD 上的操作會比較快。
- 如果不適用系統預設的方式，選擇MEMORY_ONLY_SER。這是一個更快的序列化物件的空間使用率，速度也不錯。
- 除非計算 RDD 耗損資源多，或是資料量過於龐大，不要將RDD 儲存在硬碟上，否則，重新計算一個分區就會和讀取硬碟資料一樣慢。

- 如果希望錯誤恢復速度加快，可以利用重複(replicated) 儲存方式。所有的儲存方式都可以通過重複計算遺失的資料來支援容錯機制。
- 在擁有大量記憶體的環境或者多應用程式的環境，OFF_HEAP 擁有下列優勢：
    - 它運行多個執行者共享Tachyon 中相同的記憶體池 (memory pool)
    - 它明顯減少收垃圾的花費
    - 如果單個執行者毀損，記憶體的數據不會遺失

刪除資料

Spark 自動監控每個節點記憶體使用情況，利用最近最少使用原則來刪除老舊資料。如果你想手動刪除 RDD ， 可以用`RDD.unpersist()`

共享變數

一般情況，當傳遞一個操作函數(例如 map 或 reduce) 給 Spark時，Spark 實際上是操作這個函數變數的副本。這些變數被複製到每台機器上，而且這些變數在遠端機器上的更新都不會傳送回驅動程式。一般來說，跨任務的讀寫操作變數效率不高，但 Spark 還是為了兩種常用模式提供共享變數：廣播變數 (broadcast variable) 與累加器 (accumulator)

廣播變數

廣播變數允許程式將一個可讀變數存在每台機器的記憶體裡，而不是每個任務都存有一份副本。例如，利用廣播變數，我們能將一個大資料量輸入的集合副本發送到每個節點上。(Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.They can be used, for example, to give every node a copy of a large input dataset in an efficient manner.) Spark 也會試著利用有效率的廣播算法來分配廣播變數，以減少傳遞之間的資源成本。

一個廣播變數可以利用 `SparkContext.broadcast(v)` 方法從一個初始化變數v 中產生。廣播變數是v的一個包裝變數，它的值可以透過 `value` 方法取得，可參考下列程式：

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: spark.Broadcast[Array[Int]] = spark.Broadcast(b5c40191-a864-4c7d-b9bf-d87e1a4e787c)
scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

广播变量创建以后，我们就能够在集群的任何函数中使用它来代替变量v，这样我们就不需要再次传递变量v到每个节点上。另外，为了保证所有的节点得到广播变量具有相同的值，对象v不能在广播之后被修改。

廣播變數建好以後，就能夠在集群的任何函數中使用它，來代替變數v，這樣一來，就不用再次傳送變數v 到其他節點上。此外，為了保障所有節點得到的廣播變數值相同，廣播之後就不能對物件v再做修改。

累加器

顧名思義，累加器是一種只能利用關連操作做“加” 操作的變數，因此他能夠快速的執行併行操作。而且他們能夠操作 `counters` 和 `sums` 。Spark 原本支援數值類型的累加器，開發人員可以自行增加可被支援的類型。

如果建立一個具名的累加器，它可在 Spark UI 上顯示。這對理解運作階段 (running stages) 的過程很有幫助。(注意：python 中已支援 AccumulatorPython)

一个累加器可以通过调用 `SparkContext.accumulator(v)` 方法从一个初始变量v中创建。运行在集群上的任务可以通过 `add` 方法或者使用 `+=` 操作来给它加值。然而，它们无法读取这个值。只有驱动程序可以使用 `value` 方法来读取累加器的值。 如下的代码，展示了如何利用累加器将一个数组里面的所有元素相加：

```
scala> val accum = sc.accumulator(0, "My Accumulator")
accum: spark.Accumulator[Int] = 0
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
scala> accum.value
res2: Int = 10
```

```
>>> accum = sc.accumulator(0)
Accumulator<id=0, value=0>
>>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
```

```
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
scala> accum.value
10
```

這個例子利用內建的整數類型累加器，開發者還可以利用AccumulatorParam 建立自己的累加器。

AccumulatorParam 接口有兩個方法：`zero` 方法替你的資料類型提供一個“0 值” (zero value)；`addInPlace` 方法計算兩個值的和。例如，我们有一個 `Vector` 物件代表數學向量，累加器可以透過下列方式進行定義：

```
object VectorAccumulatorParam extends AccumulatorParam[Vector] {
  def zero(initialValue: Vector): Vector = {
    Vector.zeros(initialValue.size)
  }
  def addInPlace(v1: Vector, v2: Vector): Vector = {
    v1 += v2
  }
}
// Then, create an Accumulator of this type:
val vecAccum = sc.accumulator(new Vector(...))(VectorAccumulatorParam)
```

在scala 中，Spark 支援一般Accumulable 接口來累計數值-結果類型和用於累加的元素類型不同(例如收集的元素建立列表)。Spark 支援 `SparkContext.accumulableCollection` 方法累加一般的scala 集合。

從這裡開始

你可以從Spark 官網上尋找一些Spark 執行範例。另外，Spark的example 目錄提供幾個Spark 例子，你可以利用下列方式執行Java 或是scala範例：

```
./bin/run-example SparkPi
```

為了優化你的專案， configuration和tuning說明提供許多實用的資訊。因為，確認你RDDs 中的資料是有效格式是非常重要的事情。此外，為了提昇部署效能，[集群模式介紹] (https://spark.apache.org/docs/latest/cluster-overview.html) 介紹了許多分布式操作的集群管理說明。

最後，完整的API 文件可以從以下網址連結scala, java, python 中查詢。

Spark Streaming

Spark streaming是Spark核心API的一個擴充，它對即時資料串流的處理具有可擴充性、高吞吐量、可容錯性等特點。我們可以從kafka、flume、Twitter、 ZeroMQ、Kinesis等來源取得資料，也可以通過由 高階函式如map、reduce、join、window等組成的複雜演算法計算出資料。最後，處理後的資料可以推送到檔案系統、資料庫、即時儀表板中。事實上，你可以將處理後的資料應用到Spark的機器學習演算法、 圖形處理演算法中去。

在內部，它的工作原理如下圖所示。Spark Streaming接收即時的輸入資料串流，然後將這些資料切分為批次資料供Spark引擎處理，Spark引擎將資料生成最終的結果資料。

Spark Streaming支援一個高層的抽象類別類別，叫做離散化串流(`discretized stream`)或者`DStream`，它代表連續的資料串流。DStream既可以利用從Kafka, Flume和Kinesis等來源取得的輸入資料串流創建，也可以在其他DStream的基礎上藉由高階函式獲得。在內部，DStream是由一系列RDDs組成。

本指南指導使用者開始利用DStream編寫Spark Streaming程式。使用者能夠利用scala、java或者Python來編寫Spark Streaming程式。

注意：Spark 1.2已經為Spark Streaming導入了Python API。它的所有DStream transformations和幾乎所有的輸出操作可以在scala和java介面中使用。然而，它只支援基本的來源如純文字文件或者socket上 的文字資料。諸如flume、kafka等外部的來源的API會在將來導入。

一個快速的例子

在我們進入如何編寫Spark Streaming程式的細節之前，讓我們快速地瀏覽一個簡單的例子。在這個例子中，程式從監聽TCP Socket的資料伺服器取得文字資料，然後計算文字中包含的單字數。做法如下：

首先，我們導入Spark Streaming的相關類別以及一些從StreamingContext獲得的隱式轉換到我們的環境中，為我們所需的其他類別（如DStream）提供有用的函數。StreamingContext 是Spark所有串流操作的主要入口。然後，我們創建了一個具有兩個執行執行緒以及1秒批次間隔時間(即以秒為單位分割資料串流)的本地StreamingContext。

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
// Create a local StreamingContext with two working thread and batch interval of 1 second
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

利用這個StreamingContext，我們能夠創建一個DStream，它表示從TCP來源（主機位址localhost，port為9999）取得的資料串流。

```
// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)
```

這個 `lines` 變數是一個DStream，表示即將從資料伺服器獲得的資料串流。這個DStream的每條紀錄都代表一行文字。下一步，我們需要將DStream中的每行文字都切分為單字。

```
// Split each line into words
val words = lines.flatMap(_.split(" "))
```

`flatMap` 是一個一對多的DStream操作，它通過把原DStream的每條紀錄都生成多條新紀錄來創建一個新的DStream。在這個例子中，每行文字都被切分成了多個單字，我們把切分 的單字串流用 `words` 這個DStream表示。下一步，我們需要計算單字的個數。

```
import org.apache.spark.streaming.StreamingContext._
// 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()
```

`words` 這個DStream被mapper(一對一轉換操作)成了一個新的DStream，它由（word，1）對組成。然後，我們就可以用這個新的DStream計算每批次資料的單字頻率。最後，我們用 `wordCounts.print()` 印出每秒計算的單字頻率。

需要注意的是，當以上這些程式碼被執行時，Spark Streaming僅僅準備好了它要執行的計算，實際上並沒有真正開始執行。在這些轉換操作準備好之後，要真正執行計算，需要調用以下的函數

```
ssc.start()             // Start the computation
ssc.awaitTermination()  // Wait for the computation to terminate
```

完整的例子可以在NetworkWordCount中找到。

如果你已經下載和建構了Spark環境，你就能夠用以下的函數執行這個例子。首先，你需要運行Netcat作為資料伺服器

```
$ nc -lk 9999
```

然後，在不同的terminal，你能夠用如下方式執行例子

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

基本概念

在了解了簡單的例子的基礎後，接著將介紹編寫Spark Streaming應用程式必需的一些基本概念。

- 連接
- 初始化StreamingContext
- 離散化串流
- 輸入DStreams
- DStream中的轉換
- DStream的輸出操作
- 暫存或持續化
- Checkpointing
- 部署應用程式
- 監控應用程式

連接

與Spark類似，Spark Streaming也可以利用maven函式庫。編寫你自己的Spark Streaming程式，你需要引入下面的Dependencies到你的SBT或者Maven項目中

```
    org.apache.spark
    spark-streaming_2.10
    1.2
```

為了從Kafka, Flume和Kinesis這些不在Spark核心API中提供的來源獲取資料，我們需要添加相關的模區塊 `spark-streaming-xyz_2.10` 到Dependencies中。例如，一些通用的組件如下表所示：

Source	Artifact
Kafka	spark-streaming-kafka_2.10
Flume	spark-streaming-flume_2.10
Kinesis	spark-streaming-kinesis-asl_2.10
Twitter	spark-streaming-twitter_2.10
ZeroMQ	spark-streaming-zeromq_2.10
MQTT	spark-streaming-mqtt_2.10

為了獲取最新的列表，請訪問Apache repository

初始化StreamingContext

為了初始化Spark Streaming程式，一個StreamingContext對象必需被創建，它是Spark Streaming所有串流操作的主要入口。一個StreamingContext 對象可以用SparkConf對象創建。

```
import org.apache.spark._
import org.apache.spark.streaming._
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

`appName`表示你的應用程式顯示在集群UI上的名字，`master`是一個Spark、Mesos、YARN集群URL 或者一個特殊字串“local[*]”，它表示程式用本地模式運行。當程式運行在集群中時，你並不希望在程式中硬編碼`master`，而是希望用`spark-submit`啟動應用程式，並從`spark-submit`中得到`master`的值。對於本地測試或者單元測試，你可以傳遞“local”字串在同一個進程內運行Spark Streaming。需要注意的是，它在內部創建了一個SparkContext對象，你可以藉由`ssc.sparkContext`訪問這個SparkContext對象。

批次時間片需要根據你的程式的潛在需求以及集群的可用資源來設定，你可以在性能調教那一節獲取詳細的訊息。

可以利用已經存在的`SparkContext`對象創建`StreamingContext`對象。

```
import org.apache.spark.streaming._
val sc = ...                // existing SparkContext
val ssc = new StreamingContext(sc, Seconds(1))
```

當一個StreamingContext（context）定義之後，你必須按照以下幾步進行操作

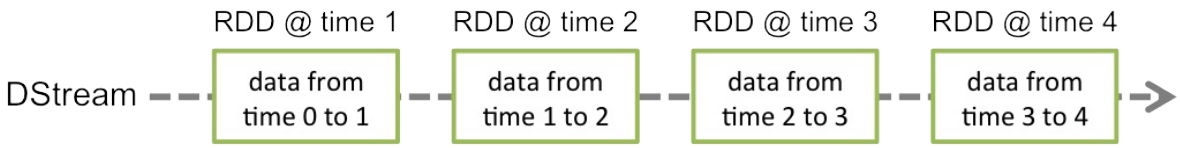
- 定義輸入來源；
- 準備好串流計算指令；
- 利用`streamingContext.start()`函數接收和處理資料；
- 處理過程將一直持續，直到`streamingContext.stop()`函數被調用。

幾點需要注意的地方：

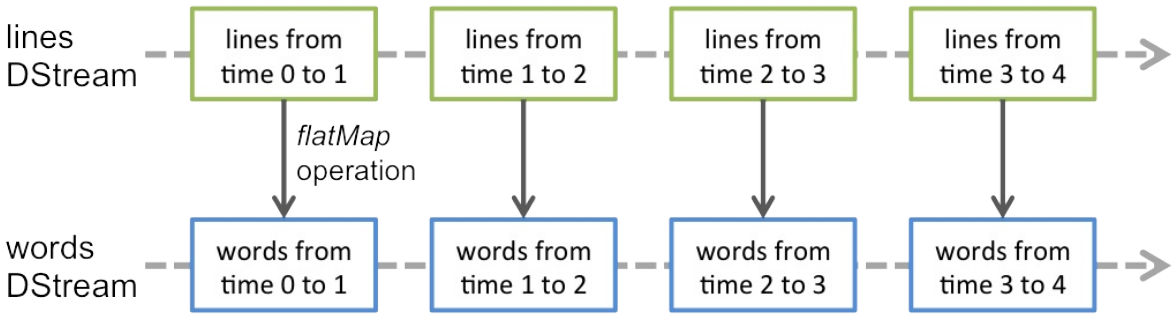
- 一旦一個context已經啟動，就不能有新的串流操作建立或者是添加到context中。
- 一旦一個context已經停止，它就不能再重新啟動
- 在JVM中，同一時間只能有一個StreamingContext处於活躍狀態
- 在StreamingContext上調用`stop()`函數，也會關閉SparkContext對象。如果只想僅關閉StreamingContext對象，設定`stop()`的可選參數為false
- 一個SparkContext對象可以重複利用去創建多個StreamingContext對象，前提條件是前面的StreamingContext在後面StreamingContext創建之前關閉（不關閉SparkContext）。

離散化串流（DStreams）

離散化串流（Discretized Streams）或者DStreams是Spark Streaming提供的基本的抽象類別，它代表一個連續的資料串流。它要麼是從來源中獲取的輸入串流，要麼是輸入串流藉由轉換操作生成的處理後的資料串流。在內部，DStreams由一系列連續的 RDD組成。DStreams中的每個RDD都包含特定時間間隔內的資料，如下圖所示：



任何對DStreams的操作都轉換成了對DStreams隱含的RDD的操作。在前面的例子中，`flatMap`操作應用於`lines`這個DStreams的每個RDD，生成`words`這個DStreams的 RDD。過程如下圖所示：



藉由Spark引擎計算這些隱含RDD的轉換操作。DStreams操作隱藏了大部分的細節，並且為了更便捷，為開發者提供了更高層的API。下面幾節將具體討論這些操作的細節。

輸入DStreams和receivers

輸入DStreams表示從資料來源獲取輸入資料串流的DStreams。在快速例子中，`lines`表示輸入DStream，它代表從netcat服務器獲取的資料串流。每一個輸入串流DStream 和一個`Receiver`對象相連接，這個`Receiver`從來源中獲取資料，並將資料存入記憶體中用於處理。

輸入DStreams表示從資料來源獲取的原始資料串流。Spark Streaming擁有兩類資料來源

- 基本來源（Basic sources）：這些來源在StreamingContext API中直接可用。例如檔案系統、socket連接、Akka的actor等。
- 高級來源（Advanced sources）：這些來源包括Kafka,Flume,Kinesis,Twitter等等。它們需要藉由額外的類來使用。我們在連接那一節討論了類Dependencies。

需要注意的是，如果你想在一個串流應用中平行地創建多個輸入DStream來接收多個資料串流，你能夠創建多個輸入串流（這將在性能調教那一節介紹）。它將創建多個Receiver同時接收多個資料串流。但是，`receiver`作為一個長期運行的任務運行在Spark worker或executor中。因此，它占有一個核，這個核是分配給Spark Streaming應用程式的所有 核中的一個（it occupies one of the cores allocated to the Spark Streaming application）。所以，為Spark Streaming應用程式分配足夠的核（如果是本地運行，那麼是執行緒） 用以處理接收的資料並且運行`receiver`是非常重要的。

幾點需要注意的地方：

- 如果分配給應用程式的核的數量少於或者等於輸入DStreams或者receivers的數量，系統只能夠接收資料而不能處理它們。
- 當運行在本地，如果你的master URL被設定成了“local”，這樣就只有一個核運行任務。這對程式來說是不足的，因為作為`receiver`的輸入DStream將會占用這個核，這樣就沒有剩餘的核來處理資料了。

基本來源

我們已經在快速例子中看到，`ssc.socketTextStream(...)`函數用來把從TCPsocket獲取的文本資料創建成DStream。除了socket，StreamingContext API也支援把文件 以及Akka actors作為輸入來源創建DStream。

- 檔案串流（File Streams）：從任何與HDFS API兼容的檔案系統中讀取資料，一個DStream可以藉由如下方式創建

```
streamingContext.fileStream[keyClass, valueClass, inputFormatClass](dataDirectory)
```

Spark Streaming將會監控`dataDirectory`目錄，並且處理目錄下生成的任何文件（嵌套目錄不被支援）。需要注意一下三點：

```
1 所有文件必須具有相同的資料格式
2 所有文件必須在`dataDirectory`目錄下創建，文件是自動的移動和重命名到資料目錄下
3 一旦移動，文件必須被修改。所以如果文件被持續的附加資料，新的資料不會被讀取。
```

對於簡單的文本文件，有一個更簡單的函數`streamingContext.textFileStream(dataDirectory)`可以被調用。檔案串流不需要運行一個receiver，所以不需要分配核。

在Spark1.2中，`fileStream`在Python API中不可用，只有`textFileStream`可用。

- 基於自定義actor的串流：DStream可以調用`streamingContext.actorStream(actorProps, actor-name)`函數從Akka actors獲取的資料串流來創建。具體的訊息見自定義receiver指南 `actorStream`在Python API中不可用。
- RDD佇列作為資料串流：為了用測試資料測試Spark Streaming應用程式，人們也可以調

用 `streamingContext.queueStream(queueOfRDDs)` 函數基於RDD佇列創建DStreams。每個push到佇列的RDD都被 當做DStream的批次資料，像串流一樣處理。

關於從socket、文件和actor中獲取串流的更多細節，請看StreamingContext和 JavaStreamingContext

高級來源

這類來源需要非Spark函式庫介面，並且它們中的部分還需要複雜的Dependencies（例如kafka和flume）。為了減少Dependencies的版本冲突问题，從這些來源創建DStream的功能已經被移到了獨立的函式庫中，你能在連接查看 細節。例如，如果你想用來自推特的流資料創建DStream，你需要按照如下步骤操作：

- 連接：添加 `spark-streaming-twitter_2.10` 到SBT或maven項目的Dependencies中
- 編寫：導入 `TwitterUtils` 類，用 `TwitterUtils.createStream` 函數創建DStream,如下所示

```
import org.apache.spark.streaming.twitter._
TwitterUtils.createStream(ssc)
```

- 部署：將編寫的程式以及其所有的Dependencies（包括spark-streaming-twitter_2.10的Dependencies以及它的傳遞Dependencies）包裝為jar檔，然後部署。這在部署章節將會作更進一步的介紹。

需要注意的是，這些高級的來源在 `spark-shell` 中不能被使用，因此基於這些來源的應用程式無法在shell中測試。

下面將介紹部分的高級來源：

- Twitter：Spark Streaming利用 `Twitter4j 3.0.3` 獲取公共的推文流，這些推文藉由推特流API獲得。認證訊息可以藉由Twitter4J函式庫支援的 任何函數提供。你既能夠得到公共串流，也能夠得到基於關鍵字過濾後的串流。你可以查看API文件（scala和java） 和例子（TwitterPopularTags和TwitterAlgebirdCMS）
- Flume：Spark Streaming 1.2能夠從flume 1.4.0中獲取資料，可以查看flume整合指南了解詳細訊息
- Kafka：Spark Streaming 1.2能夠從kafka 0.8.0中獲取資料，可以查看kafka整合指南了解詳細訊息
- Kinesis：查看Kinesis整合指南了解詳細訊息

自定義來源

在Spark 1.2中，這些來源不被Python API支援。 輸入DStream也可以藉由自定義來源創建，你需要做的是實作用戶自定義的 `receiver`，這個 `receiver` 可以從自定義來源接收資料以及將資料推到Spark中。藉由自定義receiver指南了解詳細訊息

Receiver可靠性

基於可靠性有兩類資料來源。來源(如kafka、flume)允許。如果從這些可靠的來源獲取資料的系统能夠正確的響應所接收的資料，它就能夠確保在任何情況下不丢失資料。這樣，就有兩種類型的receiver：

- Reliable Receiver：一個可靠的receiver正確的響應一個可靠的來源，資料已經收到並且被正確地複製到了Spark中。
- Unreliable Receiver ：這些receivers不支援響應。即使對於一個可靠的來源，開發者可能實作一個非可靠的receiver，這個receiver不會正確響應。

怎樣編寫可靠的Receiver的細節在自定義receiver中有詳細介紹。

DStream中的轉換（transformation）

和RDD類似，transformation允許從輸入DStream來的資料被修改。DStreams支援很多在RDD中可用的transformation操作。一些常用的操作如下所示：

Transformation	Meaning
map(func)	利用函數 `func` 處理原DStream的每個元素，返回一個新的DStream
flatMap(func)	與map相似，但是每個輸入項可用被映射為0個或者多個輸出項
filter(func)	返回一個新的DStream，它僅僅包含來源DStream中滿足函數func的項
repartition(numPartitions)	藉由創建更多或者更少的partition改變這個DStream的平行級別(level of parallelism)
union(otherStream)	返回一個新的DStream,它包含來源DStream和otherStream的聯合元素
count()	藉由計算來源DStream中每個RDD的元素數量，返回一個包含單元素(single-element)RDDs的新DStream
reduce(func)	利用函數func聚集來源DStream中每個RDD的元素，返回一個包含單元素(single-element)RDDs的新DStream。函數應該是相連接的，以使計算可以平行化
countByValue()	這個操作應用於元素類型為K的DStream上，返回一個（K,long）對的新DStream，每個鍵的值是在原DStream的每個RDD中的頻率。
reduceByKey(func, [numTasks])	當在一個由(K,V)對組成的DStream上調用這個操作，返回一個新的由(K,V)對組成的DStream，每一個key的值均由給定的reduce函數聚集起來。注意：在預設情況下，這個操作利用了Spark預設的並發任務數去分組。你可以用 `numTasks` 參數設定不同的任務數
join(otherStream, [numTasks])	當應用於兩個DStream（一個包含（K,V）對,一個包含(K,W)對），返回一個包含(K, (V, W))對的新DStream
cogroup(otherStream, [numTasks])	當應用於兩個DStream（一個包含（K,V）對,一個包含(K,W)對），返回一個包含(K, Seq[V], Seq[W])的元組
transform(func)	藉由對來源DStream的每個RDD應用RDD-to-RDD函數，創建一個新的DStream。這個可以在DStream中的任何RDD操作中使用
updateStateByKey(func)	利用給定的函數更新DStream的狀態，返回一個新"state"的DStream。

最後兩個transformation操作需要重點介紹一下：

UpdateStateByKey操作

updateStateByKey操作允許不斷用新訊息更新它的同時保持任意狀態。你需要藉由兩步來使用它

- 定義狀態-狀態可以是任何的資料類型
- 定義狀態更新函數-怎樣利用更新前的狀態和從輸入串流裡面獲取的新值更新狀態

讓我們舉個例子說明。在例子中，你想保持一個文本資料串流中每個單字的運行次數，運行次數用一個state表示，它的類型是整數

```
def updateFunction(newValues: Seq[Int], runningCount: Option[Int]): Option[Int] = {
    val newCount = ...  // add the new values with the previous running count to get the new count
    Some(newCount)
}
```

這個函數被用到了DStream包含的單字上

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
// Create a local StreamingContext with two working thread and batch interval of 1 second
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(" "))
// Count each word in each batch
val pairs = words.map(word => (word, 1))
val runningCounts = pairs.updateStateByKey[Int](updateFunction _)
```

更新函數將會被每個單字調用， `newValues` 擁有一系列的1（從 (词, 1)對而來），runningCount擁有之前的次數。要看完整的程式碼，見例子

Transform操作

`transform` 操作（以及它的變化形式如 `transformWith` ）允許在DStream運行任何RDD-to-RDD函數。它能夠被用來應用任何沒在DStream API中提供的RDD操作（It can be used to apply any RDD operation that is not exposed in the DStream API）。 例如，連接資料串流中的每個批（batch）和另外一個資料集的功能並沒有在DStream API中提供，然而你可以簡單的利用 `transform` 函數做到。如果你想藉由連接帶有預先計算的垃圾郵件訊息的輸入資料串流 來清理即時資料，然後過了它們，你可以按如下函數來做：

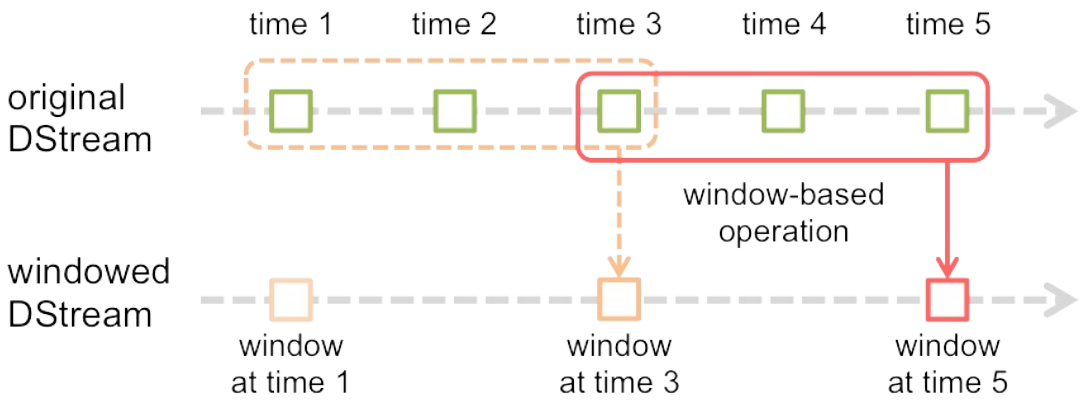
```
val spamInfoRDD = ssc.sparkContext.newAPIHadoopRDD(...) // RDD containing spam information

val cleanedDStream = wordCounts.transform(rdd => {
  rdd.join(spamInfoRDD).filter(...) // join data stream with spam information to do data cleaning
  ...
})
```

事實上，你也可以在 `transform` 函數中用機器学习和圖計算算法

窗口(window)操作

Spark Streaming也支援窗口計算，它允許你在一個滑動窗口資料上應用transformation操作。下圖闡明了這個滑動窗口。



如上圖顯示，窗口在來源DStream上滑動，合並和操作落入窗內的來源RDDs，產生窗口化的DStream的RDDs。在這個具體的例子中，程式在三個時間單元的資料上進行窗口操作，並且每兩個時間單元滑動一次。 這說明，任何一個窗口操作都需要指定兩個參數：

- 窗口長度：窗口的持續時間(圖中為3)
- 滑動時間間隔：窗口操作執行的時間間隔(圖中為2)

這兩個參數必須是來源DStream的批時間間隔的倍數(圖中為1)。

下面舉例說明窗口操作。例如，你想擴充前面的例子用來計算過去30秒的词频，間隔時間是10秒。為了達到這個目的，我們必須在過去30秒的 `pairs` DStream上應用 `reduceByKey` 操作。用函數 `reduceByKeyAndWindow` 實作。

```
// Reduce last 30 seconds of data, every 10 seconds
val windowedWordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b), Seconds(30), Seconds(10))
```

一些常用的窗口操作如下所示，這些操作都需要用到上文提到的兩個參數：窗口長度和滑動的時間間隔

Transformation	Meaning
window(windowLength, slideInterval)	基於來源DStream產生的窗口化的批次資料計算一個新的DStream
countByWindow(windowLength, slideInterval)	返回流中元素的一個滑動窗口數
reduceByWindow(func, windowLength, slideInterval)	返回一個單元素流。利用函數func聚集滑動時間間隔的流的元素創建這個單元素流。函數必須是相連接的以使計算能夠正確的平行計算。
reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])	應用到一個(K,V)對組成的DStream上，返回一個由(K,V)對組成的新的DStream。每一個key的值均由给定的reduce函數聚集起來。注意：在預設情況下，這個操作利用了Spark預設的並發任務數去分組。你可以用 `numTasks` 參數設定不同的任務數
reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])	A more efficient version of the above reduceByKeyAndWindow() where the reduce value of each window is calculated incrementally using the reduce values of the previous window. This is done by reducing the new data that enter the sliding window, and "inverse reducing" the old data that leave the window. An example would be that of "adding" and "subtracting" counts of keys as the window slides. However, it is applicable to only "invertible reduce functions", that is, those reduce functions which have a corresponding "inverse reduce" function (taken as parameter invFunc. Like in reduceByKeyAndWindow, the number of reduce tasks is configurable through an optional argument.
countByValueAndWindow(windowLength, slideInterval, [numTasks])	應用到一個(K,V)對組成的DStream上，返回一個由(K,V)對組成的新的DStream。每個key的值都是它們在滑動窗口中出現的频率。

DStreams上的輸出操作

輸出操作允許DStream的操作推到如資料函式庫、檔案系統等外部系统中。因為輸出操作實际上是允許外部系统消費轉換後的資料，它們触發的實际操作是DStream轉換。目前，定義了下面幾種輸出操作：

Output Operation	Meaning
print()	在DStream的每個批次資料中印出前10条元素，這個操作在開發和調试中都非常有用。在Python API中調用 `pprint()`。
saveAsObjectFiles(prefix, [suffix])	保存DStream的內容為一個序列化的文件 `SequenceFile`。每一個批間隔的文件的文件名基於 `prefix` 和 `suffix` 生成。"prefix-TIME_IN_MS[.suffix]"，在Python API中不可用。
saveAsTextFiles(prefix, [suffix])	保存DStream的內容為一個文本文件。每一個批間隔的文件的文件名基於 `prefix` 和 `suffix` 生成。"prefix-TIME_IN_MS[.suffix]"
saveAsHadoopFiles(prefix, [suffix])	保存DStream的內容為一個hadoop文件。每一個批間隔的文件的文件名基於 `prefix` 和 `suffix` 生成。"prefix-TIME_IN_MS[.suffix]"，在Python API中不可用。
foreachRDD(func)	在從流中生成的每個RDD上應用函數 `func` 的最通用的輸出操作。這個函數應該推送每個RDD的資料到外部系统，例如保存RDD到文件或者藉由網路寫到資料函式庫中。需要注意的是， `func` 函數在驅動程式中執行，並且通常都有RDD action在裡面推動RDD流的計算。

利用foreachRDD的設計模式

dstream.foreachRDD是一個强大的原语，發送資料到外部系统中。然而，明白怎樣正確地、有效地用這個原语是非常重要的。下面幾點介紹了如何避免一般錯誤。

- 经常寫資料到外部系统需要建一個連接對象（例如到远程服務器的TCP連接），用它發送資料到远程系统。為了達到這個目的，開發人员可能不经意的在Spark驱動中創建一個連接對象，但是在Spark worker中 嘗試調用這個連接對象保存紀錄到RDD中，如下：

```
dstream.foreachRDD(rdd => {
    val connection = createNewConnection()  // executed at the driver
    rdd.foreach(record => {
        connection.send(record) // executed at the worker
    })
})
```

這是不正確的，因為這需要先序列化連接對象，然後將它從driver發送到worker中。這樣的連接對象在機器之間不能传送。它可能表现為序列化錯誤（連接對象不可序列化）或者初始化錯誤（連接對象應該 在worker中初始化）等等。正確的解决办法是在worker中創建連接對象。

- 然而，這會造成另外一個常見的錯誤-為每一個紀錄創建了一個連接對象。例如：

```
dstream.foreachRDD(rdd => {
    rdd.foreach(record => {
        val connection = createNewConnection()
        connection.send(record)
        connection.close()
    })
})
```

通常，創建一個連接對象有資源和時間的開支。因此，為每個紀錄創建和销毁連接對象會導致非常高的開支，明顯的減少系统的整體吞吐量。一個更好的解决办法是利用 `rdd.foreachPartition` 函數。 為RDD的partition創建一個連接對象，用這個兩件

對象發送partition中的所有紀錄。

```
dstream.foreachRDD(rdd => {
     rdd.foreachPartition(partitionOfRecords => {
         val connection = createNewConnection()
         partitionOfRecords.foreach(record => connection.send(record))
         connection.close()
     })
 })
```

這就將連接對象的創建開销分摊到了partition的所有紀錄上了。

- 最後，可以藉由在多個RDD或者批次資料間重用連接對象做更進一步的優化。開發者可以保有一個静态的連接對象池，重複使用池中的對象將多批次的RDD推送到外部系统，以進一步節省開支。

```
dstream.foreachRDD(rdd => {
    rdd.foreachPartition(partitionOfRecords => {
        // ConnectionPool is a static, lazily initialized pool of connections
        val connection = ConnectionPool.getConnection()
        partitionOfRecords.foreach(record => connection.send(record))
        ConnectionPool.returnConnection(connection)  // return to the pool for future reuse
    })
})
```

需要注意的是，池中的連接對象應該根據需要延遲創建，並且在空闲一段時間後自動超時。這樣就獲取了最有效的方式發生資料到外部系统。

其它需要注意的地方：

- 輸出操作藉由懒執行的方式操作DStreams，正如RDD action藉由懒執行的方式操作RDD。具體地看，RDD actions和DStreams輸出操作接收資料的處理。因此，如果你的應用程式没有任何輸出操作或者 用於輸出操作 `dstream.foreachRDD()`，但是没有任何RDD action操作在 `dstream.foreachRDD()` 裡面，那麼什麼也不會執行。系统僅僅會接收輸入，然後丢棄它們。
- 預設情況下，DStreams輸出操作是分時執行的，它們按照應用程式的定義順序按序執行。

暫存或持續化

和RDD相似，DStreams也允許開發者持續化串流資料到記憶體中。在DStream上使用 `persist()` 函數可以自動地持續化DStream中的RDD到記憶體中。如果DStream中的資料需要計算多次，這是非常有用的。
像 `reduceByWindow` 和 `reduceByKeyAndWindow` 這種窗口操作、 `updateStateByKey` 這種基於狀態的操作，持續化是預設的，不需要開發者調用 `persist()` 函數。

例如藉由網路（如kafka，flume等）獲取的輸入資料串流，預設的持續化策略是複製資料到兩個不同的節點以容錯。

注意，與RDD不同的是，DStreams預設持續化級別是儲存序列化資料到記憶體中，這將在性能調教章節介紹。更多的訊息請看rdd持續化

Checkpointing

一個串流應用程式必須全天候運行，所有必須能夠解決應用程式邏輯無關的故障（如系統錯誤，JVM崩潰等）。為了使這成為可能，Spark Streaming需要checkpoint足夠的訊息到容錯儲存系統中， 以使系統從故障中恢復。

- Metadata checkpointing：保存流計算的定義訊息到容錯儲存系統如HDFS中。這用來恢復應用程式中運行worker的節點的故障。元資料包括
  - Configuration ：創建Spark Streaming應用程式的配置訊息
  - DStream operations ：定義Streaming應用程式的操作集合
  - Incomplete batches：操作存在佇列中的未完成的批次
- Data checkpointing ：保存生成的RDD到可靠的儲存系統中，這在有狀態transformation（如結合跨多個批次的資料）中是必須的。在這樣一個transformation中，生成的RDDDependencies於之前 批的RDD，隨着時間的推移，這個依賴鏈的長度會持續增長。在恢復的過程中，為了避免這種無限增長。有狀態的transformation的中間RDD將會定時地儲存到可靠儲存系統中，以截斷這個依賴鏈。

元資料checkpoint主要是為了從driver故障中恢復資料。如果transformation操作被用到了，資料checkpoint即使在簡單的操作中都是必須的。

何時checkpoint

應用程式在下面兩種情況下必須開啟checkpoint

- 使用有狀態的transformation。如果在應用程式中用到了 `updateStateByKey` 或者 `reduceByKeyAndWindow`，checkpoint目錄必需提供用以定期checkpoint RDD。
- 從運行應用程式的driver的故障中恢復過來。使用元資料checkpoint恢復處理訊息。

注意，沒有前述的有狀態的transformation的簡單串流應用程式在運行時可以不開啟checkpoint。在這種情況下，從driver故障的恢復將是部分恢復（接收到了但是還沒有處理的資料將會丟失）。 這通常是可以接受的，許多運行的Spark Streaming應用程式都是這種方式。

怎樣配置Checkpointing

在容錯、可靠的檔案系統（HDFS、s3等）中設定一個目錄用於保存checkpoint訊息。可以藉由 `streamingContext.checkpoint(checkpointDirectory)` 函數來做。這運行之前介紹的 有狀態transformation。另外，如果你想從driver故障中恢復，你應該以下面的方式重寫你的Streaming應用程式。

- 當應用程式是第一次啟動，新建一個StreamingContext，啟動所有Stream，然後調用 `start()` 函數
- 當應用程式因為故障重新啟動，它將會從checkpoint目錄checkpoint資料重新創建StreamingContext

```
// Function to create and setup a new StreamingContext
def functionToCreateContext(): StreamingContext = {
    val ssc = new StreamingContext(...)   // new context
    val lines = ssc.socketTextStream(...) // create DStreams
    ...
    ssc.checkpoint(checkpointDirectory)   // set checkpoint directory
    ssc
}

// Get StreamingContext from checkpoint data or create a new one
val context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContext _)

// Do additional setup on context that needs to be done,
// irrespective of whether it is being started or restarted
context. ...
```

```
  // Start the context
  context.start()
  context.awaitTermination()
```

如果 `checkpointDirectory` 存在，StreamingContext將會利用checkpoint資料重新創建。如果這個目錄不存在，將會調用 `functionToCreateContext` 函數創建一個新的StreamingContext，建立DStreams。 請看RecoverableNetworkWordCount例子。

除了使用 `getOrCreate`，開發者必須保證在故障發生時，driver處理自動重啟。只能藉由部署運行應用程式的基礎設施來達到該目的。在部署章節將有更進一步的討論。

注意，RDD的checkpointing有儲存成本。這會導致批次資料（包含的RDD被checkpoint）的處理時間增加。因此，需要小心的設定批次處理的時間間隔。在最小的批次大小(包含1秒的資料)情況下，checkpoint每批次資料會顯著的減少 操作的吞吐量。相反，checkpointing太少會導致lineage以及任務大小增大，這會產生有害的影響。因為有狀態的transformation需要RDD checkpoint。預設的間隔時間是批次間隔時間的倍數，最少10秒。它可以藉由 `dstream.checkpoint` 來設定。典型的情況下，設定checkpoint間隔是DStream的滑動間隔的5-10大小是一個好的嘗試。

部署應用程式

Requirements

運行一個Spark Streaming應用程式，有下面一些步驟

- 有管理器的集群-這是任何Spark應用程式都需要的需求，詳見部署指南
- 將應用程式包裝為jar檔-你必須編譯你的應用程式為jar檔。如果你用spark-submit啟動應用程式，你不需要將Spark和Spark Streaming包裝進這個jar檔。 如果你的應用程式用到了高級來源（如kafka，flume），你需要將它們連接的外部artifact以及它們的Dependencies打包進需要部署的應用程式jar檔中。例如，一個應用程式用到了 `TwitterUtils`，那麼就需要將 `spark-streaming-twitter_2.10` 以及它的所有Dependencies打包到應用程式jar中。
- 為executors配置足夠的記憶體-因為接收的資料必須儲存在記憶體中，executors必須配置足夠的記憶體用來保存接收的資料。注意，如果你正在做10分鐘的窗口操作，系統的記憶體要至少能保存10分鐘的資料。所以，應用程式的記憶體需求Dependencies於使用 它的操作。
- 配置checkpointing-如果stream應用程式需要checkpointing，然後一個與Hadoop API兼容的容錯儲存目錄必須配置為檢查點的目錄，串流應用程式將checkpoint訊息寫入該目錄用於錯誤恢復。
- 配置應用程式driver的自動重啟-為了自動從driver故障中恢復，運行串流應用程式的部署設施必須能監控driver進程，如果失敗了能夠重啟它。不同的集群管理器，有不同的工具得到該功能
  - Spark Standalone：一個Spark應用程式driver可以提交到Spark獨立集群運行，也就是說driver運行在一個worker節點上。進一步來看，獨立的集群管理器能夠被指示用來監控driver，並且在driver失敗（或者是由於非零的退出程式碼如exit(1)， 或者由於運行driver的節點的故障）的情況下重啟driver。
  - YARN：YARN為自動重啟應用程式提供了類似的機制。
  - Mesos： Mesos可以用Marathon提供該功能
- 配置write ahead logs-在Spark 1.2中，為了獲得極強的容錯保證，我們引入了一個新的實驗性的特性-預寫日誌（write ahead logs）。如果該特性開啟，從receiver獲取的所有資料會將預寫日誌寫入配置的checkpoint目錄。 這可以防止driver故障丟失資料，從而保證零資料丟失。這個功能可以藉由設定配置參數 `spark.streaming.receiver.writeAheadLogs.enable` 為true來開啟。然而，這些較強的語意可能以receiver的接收吞吐量為代价。這可以藉由 平行運行多個receiver增加吞吐量來解決。另外，當預寫日誌開啟時，Spark中的複製資料的功能推薦不用，因為該日誌已經儲存在了一個副本在儲存系統中。可以藉由設定輸入DStream的儲存級別為 `StorageLevel.MEMORY_AND_DISK_SER` 獲得該功能。

升級應用程式程式碼

如果運行的Spark Streaming應用程式需要升級，有兩種可能的函數

- 啟動升級的應用程式，使其與未升級的應用程式平行運行。一旦新的程式（與就程式接收相同的資料）已經準備就緒，舊的應用程式就可以關閉。這種函數支援將資料發送到兩個不同的目的地（新程式一個，舊程式一個）
- 首先，平滑的關閉（ `StreamingContext.stop(...)` 或 `JavaStreamingContext.stop(...)` ）現有的應用程式。在關閉之前，要保證已經接收的資料完全處理完。然後，就可以啟動升級的應用程式，升級 的應用程式會接着舊應用程式的點開始處理。這種函數僅支援具有來源端暫存功能的輸入來源（如flume，kafka），這是因為當舊的應用程式已經關閉，升級的應用程式還沒有啟動的時候，資料需要被暫存。

監控應用程式

除了Spark的監控功能，Spark Streaming增加了一些专有的功能。應用StreamingContext的時候，Spark web UI 顯示添加的 `Streaming` 菜單，用以顯示運行的receivers（receivers是否是存活狀態、接收的紀錄數、receiver錯誤等）和完成的批次的統計訊息（批次處理時間、佇列等待等等）。這可以用來監控 串流應用程式的處理過程。

在WEB UI中的 `Processing Time` 和 `Scheduling Delay` 兩個度量指標是非常重要的。第一個指標表示批次資料處理的時間，第二個指標表示前面的批次處理完畢之後，當前批在佇列中的等待時間。如果 批次處理時間比批次間隔時間持續更長或者佇列等待時間持續增加，這就預示系統無法以批次資料產生的速度處理這些資料，整個處理過程滯後了。在這種情況下，考慮減少批次處理時間。

Spark Streaming程式的處理過程也可以藉由StreamingListener介面來監控，這 個介面允許你獲得receiver狀態和處理時間。注意，這個介面是開發者API，它有可能在未來提供更多的訊息。

性能調教

集群中的Spark Streaming應用程式獲得最好的性能需要一些調整。這章將介紹幾個參數和配置，提高Spark Streaming應用程式的性能。你需要考慮兩件事情：

- 高效地利用集群資料來源減少批次資料的處理時間
- 設定正確的批次大小（size），使資料的處理速度能夠赶上資料的接收速度

- 減少批次資料的執行時間

- 設定正確的批次大小
- 記憶體調教

減少批次資料的執行時間

在Spark中有幾個優化可以減少批次處理的時間。這些可以在優化指南中作了討論。這節重點討論幾個重要的。

資料接收的平行化程度

藉由網路(如kafka，flume，socket等)接收資料需要這些資料反序列化並被保存到Spark中。如果資料接收成為系統的瓶頸，就要考慮平行地接收資料。注意，每個輸入DStream創建一個 `receiver` （運行在worker機器上） 接收單個資料串流。創建多個輸入DStream並配置它們可以從來源中接收不同分區的資料串流，從而實作多資料串流接收。例如，接收兩個topic資料的單個輸入DStream可以被切分為兩個kafka輸入串流，每個接收一個topic。這將 在兩個worker上運行兩個 `receiver` ，因此允許資料平行接收，提高整體的吞吐量。多個DStream可以被合並生成單個DStream，這樣運用在單個輸入DStream的transformation操作可以運用在合並的DStream上。

```
val numStreams = 5
val kafkaStreams = (1 to numStreams).map { i => KafkaUtils.createStream(...) }
val unifiedStream = streamingContext.union(kafkaStreams)
unifiedStream.print()
```

另外一個需要考慮的參數是 `receiver` 的阻塞時間。對於大部分的 `receiver` ，在存入Spark記憶體之前，接收的資料都被合並成了一個大資料區塊。每批次資料中區塊的個數決定了任務的個數。這些任務是用類 似map的transformation操作接收的資料。阻塞間隔由配置參數 `spark.streaming.blockInterval` 決定，預設的值是200毫秒。

多輸入串流或者多 `receiver` 的可選的函數是明確地重新分配輸入資料串流（利用 `inputStream.repartition(<number of partitions>)` ），在進一步操作之前，藉由集群的機器數分配接收的批次資料。

資料處理的平行化程度

如果運行在計算stage上的並發任務數不足夠大，就不會充分利用集群的資料來源。例如，對於分散式reduce操作如 `reduceByKey` 和 `reduceByKeyAndWindow` ，預設的並發任務數藉由配置屬性來確定（configuration.html#spark-properties） `spark.default.parallelism` 。你可以藉由參數（ `PairDStreamFunctions` (api/scala/index.html#org.apache.spark.streaming.dstream.PairDStreamFunctions)）傳遞平行度，或者設定參數 `spark.default.parallelism` 修改預設值。

資料序列化

資料序列化的總開銷是平常大的，特別是當sub-second級的批次資料被接收時。下面有兩個相關點：

- Spark中RDD資料的序列化。關於資料序列化請參照Spark優化指南。注意，與Spark不同的是，預設的RDD會被持續化為序列化的位元組陣列，以減少與垃圾回收相關的暫停。
- 輸入資料的序列化。從外部獲取資料存到Spark中，獲取的byte資料需要從byte反序列化，然後再按照Spark的序列化格式重新序列化到Spark中。因此，輸入資料的反序列化花費可能是一個瓶頸。

任務的啟動開支

每秒鐘啟動的任務數是非常大的（50或者更多）。發送任務到slave的花費明顯，這使請求很難獲得亞秒（sub-second）級別的反應。藉由下面的改變可以減小開支

- 任務序列化。運行kyro序列化任何可以減小任務的大小，從而減小任務發送到slave的時間。

- 執行模式。在Standalone模式下或者粗粒度的Mesos模式下運行Spark可以在比細粒度Mesos模式下運行Spark獲得更短的任務啟動時間。可以在在Mesos下運行Spark中獲取更多訊息。

These changes may reduce batch processing time by 100s of milliseconds, thus allowing sub-second batch size to be viable.

設定正確的批次大小

為了Spark Streaming應用程式能夠在集群中穩定運行，系統應該能夠以足夠的速度處理接收的資料（即處理速度應該大於或等於接收資料的速度）。這可以藉由串流的網路UI觀察得到。批次處理時間應該小於批次間隔時間。

根據串流計算的性質，批次間隔時間可能顯著的影響資料處理速率，這個速率可以藉由應用程式維持。可以考慮 `WordCountNetwork` 這個例子，對於一個特定的資料處理速率，系統可能可以每2秒印出一次單字計數 （批次間隔時間為2秒），但無法每500毫秒印出一次單字計數。所以，為了在生產環境中維持期望的資料處理速率，就應該設定合適的批次間隔時間(即批次資料的容量)。

找出正確的批次大小的一個好的辦法是用一個保守的批次間隔時間（5-10,秒）和低資料速率來測試你的應用程式。為了驗證你的系統是否能滿足資料處理速率，你可以藉由檢查點到點的延遲值來判斷（可以在 Spark驅動程式的log4j日誌中查看"Total delay"或者利用StreamingListener介面）。如果延遲維持穩定，那麼系統是穩定的。如果延遲持續增長，那麼系統無法跟上資料處理速率，是不穩定的。 你能夠嘗試着增加資料處理速率或者減少批次大小來作進一步的測試。注意，因為瞬間的資料處理速度增加導致延遲瞬間的增長可能是正常的，只要延遲能重新回到了低值（小於批次大小）。

記憶體調優

調整記憶體的使用以及Spark應用程式的垃圾回收行為已經在Spark優化指南中詳細介紹。在這一節，我們重點介紹幾個强烈推薦的自定義選項，它們可以 減少Spark Streaming應用程式垃圾回收的相關暫停，獲得更穩定的批次處理時間。

- Default persistence level of DStreams：和RDDs不同的是，預設的持續化級別是序列化資料到記憶體中（DStream是`StorageLevel.MEMORY_ONLY_SER`，RDD是`StorageLevel.MEMORY_ONLY`）。 即使保存資料為序列化形态會增加序列化/反序列化的開销，但是可以明顯的減少垃圾回收的暫停。
- Clearing persistent RDDs：預設情況下，藉由Spark內置策略（LUR），Spark Streaming生成的持續化RDD將會從記憶體中清理掉。如果spark.cleaner.ttl已經設定了，比這個時間存在更老的持續化 RDD將會被定時的清理掉。正如前面提到的那樣，這個值需要根據Spark Streaming應用程式的操作小心設定。然而，可以設定配置選項`spark.streaming.unpersist`為true來更智能的去持續化（unpersist）RDD。這個 配置使系统找出那些不需要经常保有的RDD，然後去持續化它們。這可以減少Spark RDD的記憶體使用，也可能改善垃圾回收的行為。
- Concurrent garbage collector：使用並發的標記-清除垃圾回收可以進一步減少垃圾回收的暫停時間。盡管並發的垃圾回收會減少系统的整體吞吐量，但是仍然推薦使用它以獲得更穩定的批次處理時間。

容錯語意

這一節，我們將討論在節點錯誤事件時Spark Streaming的行為。為了理解這些，讓我們先記住一些Spark RDD的基本容錯語意。

- 一個RDD是不可變的、確定可重複計算的、分散式資料集。每個RDD記住一個確定性操作的lineage(lineage)，這個lineage用在容錯的輸入資料集上來創建該RDD。
- 如果任何一個RDD的分區因為節點故障而丟失，這個分區可以藉由操作lineage從來源容錯的資料集中重新計算得到。
- 假定所有的RDD transformations是確定的，那麼最終轉換的資料是一樣的，不論Spark機器中發生何種錯誤。

Spark運行在像HDFS或S3等容錯系統的資料上。因此，任何從容錯資料而來的RDD都是容錯的。然而，這不是在Spark Streaming的情況下，因為Spark Streaming的資料大部分情況下是從 網路中得到的。為了獲得生成的RDD相同的容錯屬性，接收的資料需要重複保存在worker node的多個Spark executor上（預設的複製因子是2），這導致了當出現錯誤事件時，有兩類資料需要被恢復

- Data received and replicated ：在單個worker節點的故障中，這個資料會幸存下來，因為有另外一個節點保存有這個資料的副本。
- Data received but buffered for replication：因為沒有重複保存，所以為了恢復資料，唯一的办法是從來源中重新讀取資料。

有兩種錯誤我們需要關心

- worker節點故障：任何運行executor的worker節點都有可能出故障，那樣在這個節點中的所有記憶體資料都會丟失。如果有任何receiver運行在錯誤節點，它們的暫存資料將會丟失
- Driver節點故障：如果運行Spark Streaming應用程式的Driver節點出現故障，很明顯SparkContext將會丟失，所有執行在其上的executors也會丟失。

作為輸入來源的文件語意（**Semantics with files as input source**）

如果所有的輸入資料都存在於一個容錯的檔案系統如HDFS，Spark Streaming總可以從任何錯誤中恢復並且執行所有資料。這给出了一個恰好一次(exactly-once)語意，即無論發生什麼故障， 所有的資料都將會恰好處理一次。

基於**receiver**的輸入來源語意

對於基於receiver的輸入來源，容錯的語意既Dependencies於故障的情形也Dependencies於receiver的類型。正如之前討論的，有兩種類型的receiver

- Reliable Receiver：這些receivers只有在確保資料複製之後才會告知可靠來源。如果這樣一個receiver失敗了，緩衝區（非複製）資料不會被來源所承認。如果receiver重啟，來源會重發數 据，因此不會丟失資料。
- Unreliable Receiver：當worker或者driver節點故障，這種receiver會丟失資料

選擇哪種類型的receiverDependencies於這些語意。如果一個worker節點出現故障，Reliable Receiver不會丟失資料，Unreliable Receiver會丟失接收了但是沒有複製的資料。如果driver節點 出現故障，除了以上情況下的資料丟失，所有過去接收並複製到記憶體中的資料都會丟失，這會影響有狀態transformation的结果。

為了避免丟失過去接收的資料，Spark 1.2引入了一個實驗性的特徵 `write ahead logs`，它保存接收的資料到容錯儲存系統中。有了 `write ahead logs` 和Reliable Receiver，我們可以 做到零資料丟失以及exactly-once語意。

下面的表格總结了錯誤語意：

Deployment Scenario	Worker Failure	Driver Failure
Spark 1.1 或者更早, 没有write ahead log 的Spark 1.2	在Unreliable Receiver情況下緩衝區資料丟失；在Reliable Receiver和文件的情況下，零資料丟失	在Unreliable Receiver情況下緩衝區資料丟失；在所有receiver情況下，過去的資料丟失；在文件的情況下，零資料丟失
帶有write ahead log 的Spark 1.2	在Reliable Receiver和文件的情況下，零資料丟失	在Reliable Receiver和文件的情況下，零資料丟失

輸出操作的語意

根據其確定操作的lineage，所有資料都被建模成了RDD，所有的重新計算都會產生同樣的結果。所有的DStream transformation都有exactly-once語意。那就是說，即使某個worker節點出現 故障，最終的轉換結果都是一樣。然而，輸出操作（如 `foreachRDD`）具有 `at-least once` 語意，那就是說，在有worker事件故障的情況下，變換後的資料可能被寫入到一個外部實體不止一次。 利用 `saveAs***Files` 將資料保存到HDFS中的情況下，以上寫多次是能夠被接受的（因為文件會被相同的資料覆盖）。

Spark SQL

Spark SQL允許Spark執行用SQL, HiveQL或者Scala表示的關係查詢。這個模組的核心是一個新類型的RDD-SchemaRDD。SchemaRDDs由行物件組成，行物件用有一個模式（scheme） 來描述行中每一列的資料類型。SchemaRDD與關聯式資料庫中的表(table)很相似。可以通過存在的RDD、一個Parquet文件、一個JSON資料庫或者對儲存在Apache Hive中的資料執行HiveSQL查詢中創建。

本章的所有例子都利用了Spark分布式系統中的樣本資料，可以在 `spark-shell` 中運行它們。

開始

Spark中所有相關功能的入口點是SQLContext物件或者它的子物件，創建一個SQLContext僅僅需要一個SparkContext。

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// createSchemaRDD is used to implicitly convert an RDD to a SchemaRDD.
import sqlContext.createSchemaRDD
```

除了一個基本的 SQLContext，你也能夠創建一個 HiveContext，它支持基本 SQLContext 所支持功能的一個超集(superset)。它額外的功能包括用更完整的 HiveQL 分析器寫查詢去訪問 HiveUDFs 的能力、 從 Hive Table 讀取資料的能力。用 HiveContext 你不需要開啟一個已經存在的 Hive， SQLContext 可用的資料來源 HiveContext 也可使用。HiveContext 分開打包是為了避免在 Spark 構建時包含了所有的 Hive 依賴。如果對你的應用程式來說，這些依賴不存在問題， Spark 1.2推薦使用 HiveContext。以後的穩定版本將專注於為 SQLContext 提供與 HiveContext 等價的功能。

用來解析查詢語句的特定SQL變種語言可以通過 `spark.sql.dialect` 選項來選擇。這個參數可以通過兩種方式改變，一種方式是通過 `setConf` 方法設定，另一種方式是在SQL命令中通過 `SET key=value` 來設定。對於 SQLContext，唯一可用的方言是 “sql”，它是 Spark SQL 提供的一个個簡單的SQL解析器。在 HiveContext 中，雖然也支持 "sql"，但預設的語言是 “hiveql”。這是因為 HiveQL解析器更更完整。在很多實例中推薦使用 “hiveql”。

資料來源

Spark SQL 支持通過 SchemaRDD 接口操作各種資料來源。一個 SchemaRDD 能夠作為一個一般的 RDD 被操作，也可以被註冊為一個臨時的表。註冊一個 SchemaRDD 為一個表就可以允許你在其數據上運行 SQL 查詢。這節描述了將資料讀取為 SchemaRDD 的多種方法。

- RDDs
- parquet文件
- JSON資料集
- Hive表

RDDs

Spark 支持兩種方法將存在的 RDDs 轉換為 SchemaRDDs 。第一種方法使用映射來推斷包含特定對象類型的RDD模式(schema)。在你寫 spark 程序的同時，當你已經知道了模式，這種基於映射的方法可以使代碼更簡潔並且程序工作得更好。

創建 SchemaRDDs 的第二種方法是通過一個編程接口來實現，這個接口允许你建構一個模式，然後在存在的 RDDs 上使用它。雖然這種方法更冗長，但是它允許你在運行期之前不知道列以及列的類型的情況下構造 SchemaRDDs。

利用映射推斷(inffering)模式(scheme)

Spark SQL的Scala接口支持將包含樣本類(case class)的 RDDs 自動轉換為 SchemaRDD。這個樣本類(case class)定義了表的模式。

給樣本類的參數名字通過映射來讀取，然後作為列的名字。樣本類可以嵌套或者包含複雜的類型如序列或者數組。這個 RDD 可以隱式轉化為一個 SchemaRDD ，然後註冊為一個表。表可以在後續的 sql 語句中使用。

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// createSchemaRDD is used to implicitly convert an RDD to a SchemaRDD.
import sqlContext.createSchemaRDD

// Define the schema using a case class.
// Note: Case classes in Scala 2.10 can support only up to 22 fields. To work around this limit,
// you can use custom classes that implement the Product interface.
case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile("examples/src/main/resources/people.txt").map(_.split(",")).map(p => Person(p(0), p(1).trim.to
people.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

// The results of SQL queries are SchemaRDDs and support all the normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

编程指定模式

當樣本類不能提前確定（例如，記錄的結構是經過編碼的字串，或者一個文本集合將會被解析，不同的字段投射给不同的用戶），一个 SchemaRDD 可以通过三步来创建。

- 從原來的 RDD 創建一個行的 RDD
- 創建由一个 `StructType` 表示的模式與第一步創建的 RDD 的行結構相匹配
- 在行 RDD 上通過 `applySchema` 方法應用模式

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// Create an RDD
val people = sc.textFile("examples/src/main/resources/people.txt")

// The schema is encoded in a string
val schemaString = "name age"

// Import Spark SQL data types and Row.
import org.apache.spark.sql._
```

```
// Generate the schema based on the string of schema
val schema =
  StructType(
    schemaString.split(" ").map(fieldName => StructField(fieldName, StringType, true)))

// Convert records of the RDD (people) to Rows.
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))

// Apply the schema to the RDD.
val peopleSchemaRDD = sqlContext.applySchema(rowRDD, schema)

// Register the SchemaRDD as a table.
peopleSchemaRDD.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val results = sqlContext.sql("SELECT name FROM people")

// The results of SQL queries are SchemaRDDs and support all the normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
results.map(t => "Name: " + t(0)).collect().foreach(println)
```

Parquet檔案

Parquet是一欄位(columnar)格式，可以被許多其它的資料處理系统支援。 Spark SQL 提供支援讀和寫 Parquet 檔案的功能，這些檔案可以自動地保留原始資料的模式。

讀取資料

```
// sqlContext from the previous example is used in this example.
// createSchemaRDD is used to implicitly convert an RDD to a SchemaRDD.
import sqlContext.createSchemaRDD

val people: RDD[Person] = ... // An RDD of case class objects, from the previous example.

// The RDD is implicitly converted to a SchemaRDD by createSchemaRDD, allowing it to be stored using Parquet.
people.saveAsParquetFile("people.parquet")

// Read in the parquet file created above.  Parquet files are self-describing so the schema is preserved.
// The result of loading a Parquet file is also a SchemaRDD.
val parquetFile = sqlContext.parquetFile("people.parquet")

//Parquet files can also be registered as tables and then used in SQL statements.
parquetFile.registerTempTable("parquetFile")
val teenagers = sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

配置

可以在 SQLContext 上使用 setConf 方法配置 Parquet． 或者在用 SQL 時使用 `SET key=value` 指令來配置 Parquet。

Property Name	Default	Meaning
spark.sql.parquet.binaryAsString	false	一些其它的Parquet-producing系统，特别是Impala和其它版本的Spark SQL，當寫出 Parquet 模式的時候，二進位資料和字串之間無法分區分。這個標記告訴Spark SQL 將二進位資料解釋為字串來提供這些系统的相容性。
spark.sql.parquet.cacheMetadata	true	打開 parquet 中繼資料的暫存，可以提高靜態數據的查詢速度
spark.sql.parquet.compression.codec	gzip	設置寫 parquet 文件時的壓縮演算法，可以接受的值包括：uncompressed, snappy, gzip, lzo
spark.sql.parquet.filterPushdown	false	打開 Parquet 過濾器的 pushdown 優化。因為已知的 Paruet 錯誤，這個選項預設是關閉的。如果你的表不包含任何空的字串或者二進位的列，開啟這個選項仍是安全的
spark.sql.hive.convertMetastoreParquet	true	當設置為 false 時，Spark SQL 將使用 Hive SerDe 代替內建的支援

JSON資料集

Spark SQL能夠自動推斷 JSON 資料集的模式，讀取為 SchemaRDD。這種轉換可以透過下面兩種方法來實現.

- jsonFile ：從一個包含 JSON 文件的目錄中讀取。文件中的每一行是一個 JSON 物件
- jsonRDD ：從存在的 RDD 讀取資料，這些 RDD 的每个元素是一個包含 JSON 物件的字串

注意作為*jsonFile*的文件不是一個典型的 JSON 文件，每行必須是獨立的並且包含一個有效的JSON物件。一個多行的JSON物件經常會失敗.

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// A JSON dataset is pointed to by path.
// The path can be either a single text file or a directory storing text files.
val path = "examples/src/main/resources/people.json"
// Create a SchemaRDD from the file(s) pointed to by path
val people = sqlContext.jsonFile(path)

// The inferred schema can be visualized using the printSchema() method.
people.printSchema()
// root
//  |-- age: integer (nullable = true)
//  |-- name: string (nullable = true)

// Register this SchemaRDD as a table.
people.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

// Alternatively, a SchemaRDD can be created for a JSON dataset represented by
// an RDD[String] storing one JSON object per string.
val anotherPeopleRDD = sc.parallelize(
  """{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: Nil)
val anotherPeople = sqlContext.jsonRDD(anotherPeopleRDD)
```

Hive

Spark SQL也支援從 Apache Hive 中讀取和寫入資料。然而，Hive 有大量的相依套件，所以它不包含在 Spark 工具中。可以透過 `-Phive` 和 `-Phive-thriftserver` 參數建構 Spark，使其 支持Hive。注意這個重新建構的 jar 檔必須存在於所有的worker節點中，因為它們需要透過 Hive 的序列化和反序列化來存取儲存在 Hive 中的資料。

當和 Hive 一起工作時，開發者需要提供 HiveContext。 HiveContext 從 SQLContext 繼承而来，它增加了在 MetaStore 中發現表以及利用 HiveSql 寫查詢的功能。沒有 Hive 部署的用戶也 可以創建 HiveContext。當沒有通過 `hive-site.xml` 配置，上下文將會在當前目錄自動地創建 `metastore_db` 和 `warehouse`。

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)

sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

// Queries are expressed in HiveQL
sqlContext.sql("FROM src SELECT key, value").collect().foreach(println)
```

性能優化

對於某些工作負載，可以在通過在記憶體中緩存數據或者打開一些實驗選項來提高性能。

在記憶體中緩存數據

Spark SQL可以通過調用 `sqlContext.cacheTable("tableName")` 方法來緩存使用欄位格式的表。然後， Spark 將會僅僅瀏覽需要的欄位並且自動壓縮數據以減少記憶體的使用以及垃圾回收的 壓力。你可以通過調用 `sqlContext.uncacheTable("tableName")` 方法在記憶體中刪除表。

注意，如果你調用 `schemaRDD.cache()` 而不是 `sqlContext.cacheTable(...)` ,表將不會用欄位格式來緩存。在這種情況下，`sqlContext.cacheTable(...)` 是强烈推薦的用法。

可以在 SQLContext 上使用 setConf 方法或者在用SQL時運行 `SET key=value` 命令來配置記憶體緩存。

Property Name	Default	Meaning
spark.sql.inMemoryColumnarStorage.compressed	true	當設置為 true 時， Spark SQL 將為基於數據統計信息的每列自動選擇一個壓縮算法。
spark.sql.inMemoryColumnarStorage.batchSize	10000	控制每一批(batches)資料給欄位緩存的大小。更大的資料可以提高記憶體的利用率以及壓縮效率，但有OOMs的風險

其它的配置選項

以下的選項也可以用來條整查詢執行的性能。有可能這些選項會在以後的版本中放棄使用，這是因為更多的最佳化會自動執行。

Property Name	Default	Meaning
spark.sql.autoBroadcastJoinThreshold	10485760(10m)	配置一個表的最大大小(byte)。當執行 join 操作時，這個表將會廣播到所有的 worker 節點。可以將值設置為 -1 來禁用廣播。注意，目前的統計數據只支持 Hive Metastore 表，命令 `ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan` 已經在這個表中運行。
spark.sql.codegen	false	當為 true 時，特定查詢中的表達式求值的代碼將會在運行時動態生成。對於一些用有複雜表達式的查詢，此選項可導致顯著速度提升。然而，對於簡單的查詢，這各選項會減慢查詢的執行
spark.sql.shuffle.partitions	200	設定當操作 join 或者 aggregation 而需要 shuffle 資料時分區的数量

其它SQL接口

Spark SQL 也支持直接運行 SQL 查詢的接口，不用寫任何代碼。

運行 Thrift JDBC/ODBC 伺服器

這裡實現的 Thrift JDBC/ODBC 伺服器與 Hive 0.12中的HiveServer2一致。你可以用在 Spark 或者 Hive 0.12 附帶的 beeline 腳本測試 JDBC 伺服器。

在 Spark 目錄中，運行下面的命令啟動 JDBC/ODBC 伺服器。

```
./sbin/start-thriftserver.sh
```

這個腳本接受任何的 `bin/spark-submit` 命令行參數，加上一個 `--hiveconf` 參數用來指明 Hive 屬性。你可以運行 `./sbin/start-thriftserver.sh --help` 來獲得所有可用選項的完整列表。預設情況下，伺服器監聽 `localhost:10000` 。你可以用環境變數覆蓋這些變數。

```
export HIVE_SERVER2_THRIFT_PORT=
export HIVE_SERVER2_THRIFT_BIND_HOST=
./sbin/start-thriftserver.sh \
  --master  \
  ...
```

或者透過系統變數覆蓋。

```
./sbin/start-thriftserver.sh \
  --hiveconf hive.server2.thrift.port= \
  --hiveconf hive.server2.thrift.bind.host= \
  --master
  ...
```

現在你可以用 `beeline` 測試 Thrift JDBC/ODBC 伺服器。

```
./bin/beeline
```

連接到 Thrift JDBC/ODBC 伺服器的方式如下：

```
beeline> !connect jdbc:hive2://localhost:10000
```

Beeline 將會詢問你的用戶名稱和密碼。在非安全的模式，簡單地輸入你機器的用戶名稱和空密碼就行了。對於安全模式，你可以按照Beeline文檔的說明來執行。

運行 Spark SQL CLI

Spark SQL CLI 是一個便利的工具，它可以在本地運行 Hive 元儲存(metastore)服務、執行命令行輸入的查詢。注意，Spark SQL CLI不能與 Thrift JDBC 伺服器通信。

在 Spark 目錄運行下面的命令可以啟動 Spark SQL CLI。

```
./bin/spark-sql
```

編寫語言整合(Language-Integrated)的關聯性查詢

語言整合的關聯性查詢是實驗性的，現在暫時只支援**scala**。

Spark SQL也支持用特定領域的語法編寫查詢語句，請參考使用下列範例的資料：

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// Importing the SQL context gives access to all the public SQL functions and implicit conversions.
import sqlContext._
val people: RDD[Person] = ... // An RDD of case class objects, from the first example.

// 下述等同於 'SELECT name FROM people WHERE age >= 10 AND age <= 19'
val teenagers = people.where('age >= 10).where('age <= 19).select('name)
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

DSL使用Scala的符號來表示在潛在表(underlying table)中的列，這些列以前綴(')標示。將這些符號透過隱式轉成由SQL執行引擎計算的表達式。你可以在ScalaDoc 中了解詳情。

Spark SQL資料類型

- 數字類型
  - ByteType：代表一個位元的整數。範圍是-128到127
  - ShortType：代表兩個位元的整數。範圍是-32768到32767
  - IntegerType：代表4個位元的整數。範圍是-2147483648到2147483647
  - LongType：代表8個位元的整數。範圍是-9223372036854775808到9223372036854775807
  - FloatType：代表4位元的單精度浮點數
  - DoubleType：代表8位元的雙精度浮點數
  - DecimalType：代表任意精度的10進位資料。透過內部的java.math.BigDecimal支援。BigDecimal由一個任意精度的不定長度(unscaled value)的整數和一個32位元整數組成
  - StringType：代表一個字串
  - BinaryType：代表個byte序列值
  - BooleanType：代表boolean值
  - Datetime類型
    - TimestampType：代表包含年，月，日，時，分，秒的值
    - DateType：代表包含年，月，日的值
  - 複雜類型
    - ArrayType(elementType, containsNull)：代表由elementType類型元素組成的序列值。`containsNull` 用來指明 `ArrayType` 中的值是否有null值
    - MapType(keyType, valueType, valueContainsNull)：表示包括一組鍵 - 值組合的值。透過 keyType 表示 key 資料的類型，透過 valueType 表示 value 資料的類型。`valueContainsNull` 用來指明 `MapType` 中的值是否有null值
    - StructType(fields):表示一個擁有 `StructFields (fields)` 序列結構的值
      - StructField(name, dataType, nullable):代表 `StructType` 中的域(field)，field的名字通过 `name` 指定，`dataType` 指定field的資料類型， `nullable` 表示filed的值是否有null值。

Spark的所有資料類型都定義在 `org.apache.spark.sql` 中，你可以透過 `import org.apache.spark.sql._` 取用它們。

資料類型	**Scala**中的值類型	取用或者創建資料類型的**API**
ByteType	Byte	ByteType
ShortType	Short	ShortType
IntegerType	Int	IntegerType
LongType	Long	LongType
FloatType	Float	FloatType
DoubleType	Double	DoubleType
DecimalType	scala.math.BigDecimal	DecimalType
StringType	String	StringType
BinaryType	Array[Byte]	BinaryType
BooleanType	Boolean	BooleanType
TimestampType	java.sql.Timestamp	TimestampType
DateType	java.sql.Date	DateType
ArrayType	scala.collection.Seq	ArrayType(elementType, [containsNull]) 注意 containsNull 預設為 true
MapType	scala.collection.Map	MapType(keyType, valueType, [valueContainsNull]) 注意 valueContainsNull 預設為 true

StructType	org.apache.spark.sql.Row	StructType(fields)，注意 fields 是一個 StructField 序列，不能使用兩個相同名字的 StructField
StructField	The value type in Scala of the data type of this field (For example, Int for a StructField with the data type IntegerType)	StructField(name, dataType, nullable)

MLlib

MLlib is Spark’s scalable machine learning library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as underlying optimization primitives, as outlined below: MLlib是Spark分擴展的機器學習庫，由常見的學習算法及應用程序 所構成，其中包含分類(classification)，迴歸(regression)，群集(clustering)，協同過濾(collaborative filtering)，降維(dimensionality reduction)以及底層最佳化算法。

數據類型

MLlib支持本地向量以及矩陣貯存在一台機器上，並且通過一或多個RDD備份分布式的矩陣。本地向量及本地矩陣是作為簡易的數據模型的公用接口。底層的學習算法操作是由Breeze以及jbloas所提供的。在MLlib中，所謂的"labeled point"（標記點）是指監督式學習中的一個訓練例子。

本地向量

一個本地向量有intereage, 0-based indices及double類型，貯存在單一機器上。MLlib 支持兩種類型的本地向量: dense及sparse。

dense向量：透過輸入double陣列回傳。

sparse向量：透過兩個平行陣列：indices及values回傳

舉個例子，向量(1.0, 0.0, 3.0)可以被表示成:

dense格式－[1.0, 0.0, 3.0]

sparse格式－(3, [0, 2], [1.0, 3.0])，3表示此vector的大小。

本地向量的基類是Vector，而且提供DenseVector及SparseVector兩個實作。以下介紹使用工廠方法在Vectors中去創建本地向量。

```
import org.apache.spark.mllib.linalg.{Vector, Vectors}

// Create a dense vector (1.0, 0.0, 3.0).
val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)
// Create a sparse vector (1.0, 0.0, 3.0) by specifying its indices and values corresponding to nonzero entries.
val sv1: Vector = Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0))
// Create a sparse vector (1.0, 0.0, 3.0) by specifying its nonzero entries.
val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))
```

Note: Scala會預設匯入scala.collection.immutable.Vector，所以你必須自行匯入org.apache.spark.mllib.linalg.Vector，才能使用MLlib的Vector

---

DenseVector 與 SparseVector源碼

```
@SQLUserDefinedType(udt = classOf[VectorUDT])
class DenseVector(val values: Array[Double]) extends Vector {

  override def size: Int = values.length

  override def toString: String = values.mkString("[", ",", "]")

  override def toArray: Array[Double] = values

  private[mllib] override def toBreeze: BV[Double] = new BDV[Double](values)

  override def apply(i: Int) = values(i)

  override def copy: DenseVector = {
    new DenseVector(values.clone())
  }

  private[spark] override def foreachActive(f: (Int, Double) => Unit) = {
    var i = 0
    val localValuesSize = values.size
    val localValues = values

    while (i < localValuesSize) {
      f(i, localValues(i))
      i += 1
    }
  }
}
```

```
/**
 * A dense vector represented by a value array.
 */
@SQLUserDefinedType(udt = classOf[VectorUDT])
class DenseVector(val values: Array[Double]) extends Vector {

  override def size: Int = values.length

  override def toString: String = values.mkString("[", ",", "]")

  override def toArray: Array[Double] = values

  private[mllib] override def toBreeze: BV[Double] = new BDV[Double](values)

  override def apply(i: Int) = values(i)

  override def copy: DenseVector = {
    new DenseVector(values.clone())
  }

  private[spark] override def foreachActive(f: (Int, Double) => Unit) = {
    var i = 0
    val localValuesSize = values.size
    val localValues = values

    while (i < localValuesSize) {
      f(i, localValues(i))
      i += 1
    }
  }
}
```

標記點(Labeled point)

標記點是一個本地向量，無論密集(dense)或稀疏(sparse)均會與一一個標記/響應相關。在MLlib中，標記點被使用在監督式學習算法中。我們使用一個double去儲存一個標記(label)，那麼我們將可以在迴歸(regression)及分類(classification)中使用標記點。在二元分類中，標記應該是0或1；而在多類分類中，標記應為從0開始的的索引:0, 1, 2, ...

一個標記點用 LabeledPoint來表示（Scala中它屬於一個case class）

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint

// Create a labeled point with a positive label and a dense feature vector.
val pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))

// Create a labeled point with a negative label and a sparse feature vector.
val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0)))
```

- Sparse data

擁有sparse訓練資料是很常見的做法。MLlib支持讀取為LIBSVM格式的訓練實例，它使用LIBSVM與LIBLINEAR做為預設格式。它是一種文本格式，每一行表示成一個標記稀疏特徵向量(labeled sparse feature vector)，使用以下的格式：

```
label index1:value1 index2:value2 ...
```

其中索引是以1為基索引（one-based)並升序。在讀取後，這些特徵索引會被轉換成以0為基索引。

MLUtils.loadLibSVMFile 讀取LIBSVM格式的訓練實例

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.rdd.RDD

val examples: RDD[LabeledPoint] = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
```

本地矩陣

本地矩陣儲存在單一台機器上，它有整數類型行列索引，以及浮點類型的值。MLlib支持密度矩陣(dense matrices)，其輸入值是被儲存在單一個以列為主的浮點陣列。舉例來說，以下這個矩陣

$$\begin{pmatrix} 1.0 & 2.0\ 3.0 & 4.0\ 5.0 & 6.0\ \end{pmatrix}$$

被儲存在一個一維陣列[1.0, 3.0, 5.0, 2.0, 4.0, 6.0]以及大小為(3, 2)的矩陣中。

本地矩陣的基類為Matrix，並且提供一個實作類：DenseMatrix。我吃建議使用Matrices中的工廠方法去創建本地矩陣。

```
import org.apache.spark.mllib.linalg.{Matrix, Matrices}

// Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))
```

分布矩陣

一個分布矩陣有長整數類型(long-typed)的行列索引以及浮點類型的值, 被分散地儲存在一或多個RDD中。選擇一個正確的格式去儲存成巨大且分布的矩陣是非常重要的。將一個分布矩陣轉換成一個不同的格式可能需要請求一個全域洗牌(global shuffle)，這是相當昂貴的。目前為止，已經實作三種類型的分布矩陣。

此基本類型被稱為RowMatrix。RowMatrix是一個面向行的分布矩陣，而不是具有意義的行索引，例如：特徵向量的集合。透過一個RDD來表示所有的行，其中每一行都是一個本地向量。 我們假設RowMatrix的列數量並不巨大，所以單一本地向量可以合理的傳達給driver，也可以儲存/操作正在使用的單一節點上。

IndexedRowMatrix類似於RowMatrix，但具有行索引，可以被使用在識別行以及執行關聯(join)。

CoordinateMatrix是儲存在座標列表格式(coordinate list (COO) format)中的分布矩陣，其實體集合是一個RDD。

Note

由於我們緩存矩陣的大小，所以在分布矩陣的底層RDD必須是確定的，。一般使用非確定的RDD可能會導致錯誤。

RowMatrix

一个 RowMatrix 是一个面向行的分布式矩阵,其行索引是没有具体的含義。例如：一系列特征向量的一个集合。通過一个 RDD 来代表所有的行,每一行就是一个本地向量。既然 每一行由一个本地向量表示,所以其列数就被整型数据大小所限制,其實實作中列数是一個很小的数值。

RowMatrix可以由RDD[Vector] 實例被創建。接著我們可以計算列的摘要統計量。

```
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.mllib.linalg.distributed.RowMatrix


val rows: RDD[Vector] = ... // an RDD of local vectors
// Create a RowMatrix from an RDD[Vector].
val mat: RowMatrix = new RowMatrix(rows)

// Get its size.
val m = mat.numRows()
val n = mat.numCols()
```

IndexedRowMatrix

IndexedRowMatrix與RowMatrix是相似的，但其行索引具有特定定含義，本質上是一個含有索引訊息的行數據集合(an RDD of indexed rows)。 每一行由long類型索引和一個本地向量組成。

IndexedRowMatrix可以由一個RDD[IndexedRow]實例被創建，其中IndexedRow是一個被包裝過的(Long, Vcetor)。 IndexedRowMatrix可以透過刪除行索引被轉換成RowMatrix。

```
import org.apache.spark.mllib.linalg.distributed.{IndexedRow, IndexedRowMatrix, RowMatrix}
import util.Random

val rows: RDD[IndexedRow] = ... // an RDD of indexed rows
// Create an IndexedRowMatrix from an RDD[IndexedRow].
val mat: IndexedRowMatrix = new IndexedRowMatrix(rows)

// Get its size.
val m = mat.numRows()
val n = mat.numCols()

// Drop its row indices.
val rowMat: RowMatrix = mat.toRowMatrix()
```

CoordinateMatrix

CoordinateMatrix是一个分布式矩陣,其實體集合是一個 RDD。每一个實體是一个(i:Long, j:Long, value:Double)座標 ,其中i代表行索引，j代表列索引,value代 表實体的值。只有當矩陣的行和列都很巨大,並且矩陣很稀疏时才使用 CoordinateMatrix。

一个CoordinateMatrix可從一個RDD[MatrixEntry]實例創建,这里的 MatrixEntry是的(Long, Long, Double)的封裝尸類。通過調用toIndexedRowMatrix可以將一个CoordinateMatrix 轉變为一個IndexedRowMatrix(但其行是稀疏的)。目前暫不支持其他計算操作。

```
import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix, MatrixEntry}

val entries: RDD[MatrixEntry] = ... // an RDD of matrix entries
// Create a CoordinateMatrix from an RDD[MatrixEntry].
val mat: CoordinateMatrix = new CoordinateMatrix(entries)

// Get its size.
val m = mat.numRows()
val n = mat.numCols()

// Convert it to an IndexRowMatrix whose rows are sparse vectors.
val indexedRowMatrix = mat.toIndexedRowMatrix()
```

基本統計分析

此章節共有以下幾個主題

- 概述統計量(Summary statistics)
- 關聯(Correlations)
- 分層抽樣(Stratified sampling)
- 假設檢定(Hypothesis testing)
- 隨機數據生成(Random data generation)

概述統計量(Summary Statistics)

对 RDD[Vector]格式數據的概述統計量,我们提供 Statistics 中的 colStats 方法来實现。

colStats()方法返回一個MultivariateStatisticalSummary實例，其中包括面向列的最大值，最小值，平均，變異數，非零值個數以及總數量。

```
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}

val observations: RDD[Vector] = sc.textFile("data/mllib/sample_lda_data.txt").map(s=>Vectors.dense(s.split(" ").map(_.t
... // an RDD of Vectors

// Compute column summary statistics.
val summary: MultivariateStatisticalSummary = Statistics.colStats(observations)
println(summary.mean) // a dense vector containing the mean value for each column
println(summary.variance) // column-wise variance
println(summary.numNonzeros) // number of nonzeros in each column
```

相關性(Correlations)

在統計分析中,計算兩系列數據之間的相關性很常見。在 MLlib 中,我們提供了用於計算多系列數據之間兩兩關系的靈活性。目前支持的相關性算法是 Perarson 和 Spearsman 相關。

Statistics提供了計算系列間相關性的方法。根據輸入類型，兩個RDD[Double] 或一個RDD[Vector] ,輸出相應的一个Double或相關矩陣。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.stat.Statistics

val sc: SparkContext = ...

val seriesX: RDD[Double] = ... // a series
val seriesY: RDD[Double] = ... // must have the same number of partitions and cardinality as seriesX

// compute the correlation using Pearson's method. Enter "spearman" for Spearman's method. If a
// method is not specified, Pearson's method will be used by default.
val correlation: Double = Statistics.corr(seriesX, seriesY, "pearson")

val data: RDD[Vector] = ... // note that each Vector is a row and not a column

// calculate the correlation matrix using Pearson's method. Use "spearman" for Spearman's method.
// If a method is not specified, Pearson's method will be used by default.
val correlMatrix: Matrix = Statistics.corr(data, "pearson")
```

分層抽樣

在 MLlib 中,不同於其他统計方法,分層抽样方法如 sampleByKey 和 sampleByKeyExact，運行在鍵值對格式的 RDD 上。對分層抽样来说，keys是一个標簽, 值是特定的屬性。比如，key 可以是男人或女人、文檔 ID，其相應的值可以是人口數據中的年齡列表或者文檔中的詞列表。sampleByKey 方法對每一个觀測擲幣決定是否抽中它, 所以需要對數據進行一次遍歷,也需要輸入期望抽样的大小。而 sampleByKeyExact方法並不是簡單地在每一層中使用sampleByKey 方法隨機抽样，它需要更多資源，但將提供信心度高達 99.99%的精確抽样大小。在 Python 中，目前不支持 sampleByKeyExact。

sampleByKeyExact()充許使用者準確抽取 $$[f_k \cdot n_k] \quad\forall k\in\mathbb K $$個元素，這裡的$$f_k$$是從鍵k中期望抽取的比例，$$n_k$$是從鍵k中抽取的鍵值對數量，而$$\mathbb K$$是鍵的集合。為了確保抽樣大小，無放回抽樣對數據會多一次遍歷；然而，有放回的抽樣會多兩次遍歷。

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.rdd.PairRDDFunctions

val sc: SparkContext = ...

val data = ... // an RDD[(K, V)] of any key value pairs
val fractions: Map[K, Double] = ... // specify the exact fraction desired from each key

// Get an exact sample from each stratum
val approxSample = data.sampleByKey(withReplacement = false, fractions)
val exactSample = data.sampleByKeyExact(withReplacement = false, fractions)
```

假設檢定

在統計分析中，假設檢定是一個強大的工具，用來判斷結果的統計量是否充分，以及結果是否隨機。MLlib目前支持Pearson卡方檢定($$\chi^2$$) 來檢定適配度和獨立性。輸入數據類型決定了是否產生適配度或獨立性，適配度檢定需要Vector輸入類型，而獨立性檢定需要一個Matrix矩陣輸入。

MLlib也支持RDD[LabeledPoint]輸入類型，然後使用卡方獨立性檢定來進行特徵選擇。

Statistics提供了進行Pearson卡方檢定的方法。下面示例演示了怎樣運行和解釋假設定。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.stat.Statistics._

val sc: SparkContext = ...

val vec: Vector = ... // a vector composed of the frequencies of events

// compute the goodness of fit. If a second vector to test against is not supplied as a parameter,
// the test runs against a uniform distribution.
val goodnessOfFitTestResult = Statistics.chiSqTest(vec)
println(goodnessOfFitTestResult) // summary of the test including the p-value, degrees of freedom,
                                 // test statistic, the method used, and the null hypothesis.

val mat: Matrix = ... // a contingency matrix

// conduct Pearson's independence test on the input contingency matrix
val independenceTestResult = Statistics.chiSqTest(mat)
println(independenceTestResult) // summary of the test including the p-value, degrees of freedom...

val obs: RDD[LabeledPoint] = ... // (feature, label) pairs.

// The contingency table is constructed from the raw (feature, label) pairs and used to conduct
// the independence test. Returns an array containing the ChiSquaredTestResult for every feature
// against the label.
val featureTestResults: Array[ChiSqTestResult] = Statistics.chiSqTest(obs)
var i = 1
featureTestResults.foreach { result =>
    println(s"Column $i:\n$result")
    i += 1
} // summary of the test
```

隨機數據生成

隨機數據生成對隨機算法，原型及性能測式來說是有用的。MLlib支持指定分類型來生成隨機的RDD，如均勻，標準常態，Possion分布。

RandomRDDs提供工廠方法來生成隨機double RDD或vector RDDs。下面示例生成一個隨機的dobule RDD，其值服從標準常態分布$$N(0,1)$$，然後將其映射為$$N(0,1)$$。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.random.RandomRDDs._

val sc: SparkContext = ...

// Generate a random double RDD that contains 1 million i.i.d. values drawn from the
// standard normal distribution `N(0, 1)`, evenly distributed in 10 partitions.
val u = normalRDD(sc, 1000000L, 10)
// Apply a transform to get a random double RDD following `N(1, 4)`.
val v = u.map(x => 1.0 + 2.0 * x)
```

Classification and Regression

MLlib支特二元分類，多元分類，回歸分析等多種算表，下表列出問題類別及其相關的算法:

問題類別	支持算法
二元分類	線性支持向量機(linear SVMs)，邏輯斯迴歸(logistic regression)，決策樹(decision trees)， 單純貝氏(naive Bayes)
多元分類	決策樹，單純貝氏
迴歸	線性最小平方法，Lasso，脊迴歸(ridge regression)

這些方法的更多細節請參照下面內容:

- 線性模型
  - 二元分類(支特向量機，邏輯斯迴歸)
  - 線性迴歸(最小平方法，Lasso，ridge)
- 決策樹
- 單純貝氏

Linear Methods

- 數學公式
  - 損失函數(Loss functions)
  - 正則化(Regularizers)
  - 最佳化
- 二元分類
  - 線性支持向量機(Linear Support Vector Machines, SVMs)
  - 邏輯斯迴歸(Logistic regression)
  - 評估指標(Evaluation metrics)
  - 示例
- 最小平方法(Linear least squares), Lasso, 及 脊迴歸(ridge regression)
  - 示例
- 流式線性迴歸(Streaming linear regression)
  - 示例
- 實作 (開發者)

數學公式(Mathematical formulation)

許多標準機器學習方法可以被轉換為凸優化問題(convex optimization problem), 即一個找到凸函數$$f$$最小值的任務，這個函數$$f$$依賴於一個有d個值的向量變量$$w$$(代碼中的weights)。更正式點，這是一個$$min_{w \in R^d} f(x)$$優化問題，其目標函數$$f$$具有下面形式：

$$ f(x):=\lambda R(w)+\frac{1}n \sum_{i=1}^{n}L(w; x_i, y_i)

$$ 向量$$x_i \in R^d$$是訓練數據樣本，其中$$1\leq i\leq n$$。 $$y_i \in R$$是相對應的類標簽，也是我們想要預測的目標。如果$$L(w; x, y)$$能被表述為$$w^Tx$$和$$y$$的一個函數，我們稱該方法為線性的，有機個MLlib分類和迴歸算法屬於該范疇，我們在此一一討論。

目標函數$$f$$包括兩部份：控制模型複雜度的正則化因子和度量模型誤差的損失函數。損失函數$$L(w;.)$$是典型與$$w$$相關的凸函數。事先鎖定正則化參數$$\lambda \geq 0$$(代碼中的regParam)承載了我們在最小化損失量(訓練誤差)和最小化模型複雜度(避免過渡擬合)兩個目標之間的權衡取捨。

損失函數

下表概述了MLlib支持的損失函數及其梯度和子梯度：

	損失函數**$$L(w;x,y)$$**	梯度或子梯度
hinge loss	$$max{0, 1-yw^Tx} $$	$$ \left{ \begin{array}{ll} -yx ~~若~yw^Tx \lt1, \ 0 ~~~otherwise \end{array} \right.$$
logistic loss	$$log(1+exp(-yw^Tx)),~y \in {-1,+1}$$	$$-y(1-\frac{1}{1+exp(-yw^Tx)})x$$
squared loss	$$\frac{1}{2}(x^Tx-y)^2,~y \in R$$	$$(w^Tx-y)x$$

正則化

正則化的目標是獲得簡單模型和避免過度似合。在MLilb中，支持下面正則化因子：

	正則化因子**$$R(w)$$**	梯度或子梯度
零（未正則化）	0	0
L2范數	$$\frac{1}{2} \Vert w\Vert_2^2$$	$$w$$
L1范數	$$\Vert w\Vert_1 $$	$$sign(w)$$

在這裡，$$sign(w)$$是表示$$w$$中所有實體的類標簽($$sign(\pm 1)$$)的向量。

與L1正則化問題比較，由於L2的平滑性，L2的正則化問題一般較容易解決。但是由於可以強化權重的稀疏性，L1正則化更能產生較小及更容易解釋的模型，而後者在特徵選擇是非常有用的。不正則化而去訓練模型是不恰當的，尤其是在訓練樣本數量較小的時候。

最佳化

事實上，線性方法使用凸優化方法去最佳化目標函數。MLlib使用SGD及L-BFGS兩個方法，它們將在最佳化的章節被介紹。目前，大多數的算法API支持隨機梯法下降(Stochastic Gradient Descent, SGD)，也有一些支持 L-BFGS。在最佳化方法做選擇，可以參照最佳化章節的指南。

二元分類

二元分類將數據項分成兩類： 正例及負例。MLlib支持兩種二元分類的線性方法：線性支持向量機與邏輯斯迴歸。對這兩種方法來說，MLlib都支持L1, L2正則化。在MLlib中，訓練數据集用一個LabeledPoint格式的RDD來表示。需要注意，本指南中的數學公式裡，約定訓練標簽$$y$$為+1(正例)或-1(反例)，但在MLlib中，為了與多煩標簽保持一致，反例標簽是0, 而不是-1。

線性支持向量機(SVMs)

對於大規模的分類任務來說，線性支持向量機是標準的方法。它是之前"數學公式"一節中所描述的線性方法，廿金金損失函數是hinge loss:

$$ L(w;x,y)=\max{0,1-yw^Tx}.

$$ 預設配置下，線性SVM使用L2正則化訓練。我們支持L1正則化。通過這種方式，問題變為線性規劃問題。

線性SVM算法是產出一個SVM模型。給定新數據點$$x$$，該模型基於$$w^Tx$$的值來預測。默認情形下，$$w^Tx\geq0$$時為正例，否則為反例。

邏輯斯迴歸(Logistic regression)

邏輯斯迴歸廣泛被運用於二元變量預測。它是之前"數學公式"一節中竹竹尸一中描述的線性方法，其損失函數是logistic losss:

$$ L(w;x,y)=log(1+exp(-yx^Tx))

$$ 迴輯斯迴歸算法的產出是一個卜口田中土輯斯迴歸模型。給定新數據點$$x$$，該模型運用下面的邏輯函數來預測：

$$ f(z)=\frac{1}{1+e^{-z}}

$$ 在這裡，$$z=w^Tx$$。預設情況下，若$$f(w^Tx)\gt0.5$$輸出是正例，否則是反例。與線性SVM不同之處在於，線性迴歸模型$$f(z)$$輸出含有一個機率解釋(即$$x$$)是正例的機率。

評估指標(Evaluation metrics)

MLlib支持常用的二元分類評估指標方法（在PySpark中不可用）。包括精度，召回率，F度量(F-measure)，接收者特徵操作曲線(receiver operating characteristic, ROC)，精度-召回率曲線以及曲面下面積(area under the curves, AUC)。AUC常用來比較不同模型的性能，精度/召回率/F度量用來決定閾值(threshold)時為預測指定恰當閾值。

示例

下面代碼片段演示了如何加載數據集，運用算法對象的靜態方法執行訓練算法，以及運用模型預測來計算訓練誤差。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.classification.SVMWithSGD
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.util.MLUtils

// Load training data in LIBSVM format.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")

// Split data into training (60%) and test (40%).
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache()
val test = splits(1)

// Run training algorithm to build the model
val numIterations = 100
val model = SVMWithSGD.train(training, numIterations)

// Clear the default threshold.
model.clearThreshold()

// Compute raw scores on the test set.
val scoreAndLabels = test.map { point =>
  val score = model.predict(point.features)
  (score, point.label)
}

// Get evaluation metrics.
val metrics = new BinaryClassificationMetrics(scoreAndLabels)
val auROC = metrics.areaUnderROC()

println("Area under ROC = " + auROC)
```

預設配置下，SVMWithSGD.train()將正則化參數設置為1.0來進行L2正則化。如果我們想配置算法參數，我們可以直接生一個SVMWithSGD對象，然後調用setter方法。所有其他的MLlib算法都支持這 種自定義方法。舉例來說，下面代碼生了一個用於SVM的L1正則化變量，其正則化參數為0.1，且迭代次數為200。

```
import org.apache.spark.mllib.optimization.L1Updater

val svmAlg = new SVMWithSGD()
svmAlg.optimizer.
  setNumIterations(200).
  setRegParam(0.1).
  setUpdater(new L1Updater)
val modelL1 = svmAlg.run(training)
```

LogisticRegressionWithSGD的使用方法與SVMWithSGD相似。

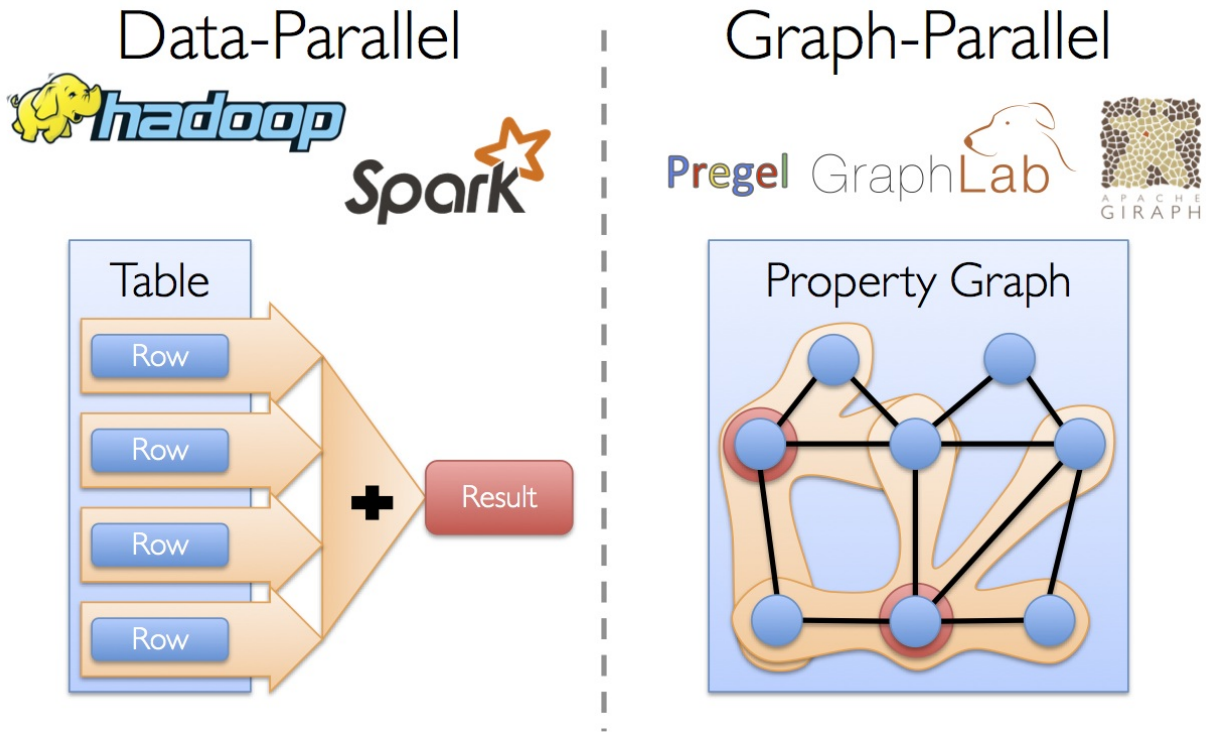
Spark GraphX

概觀

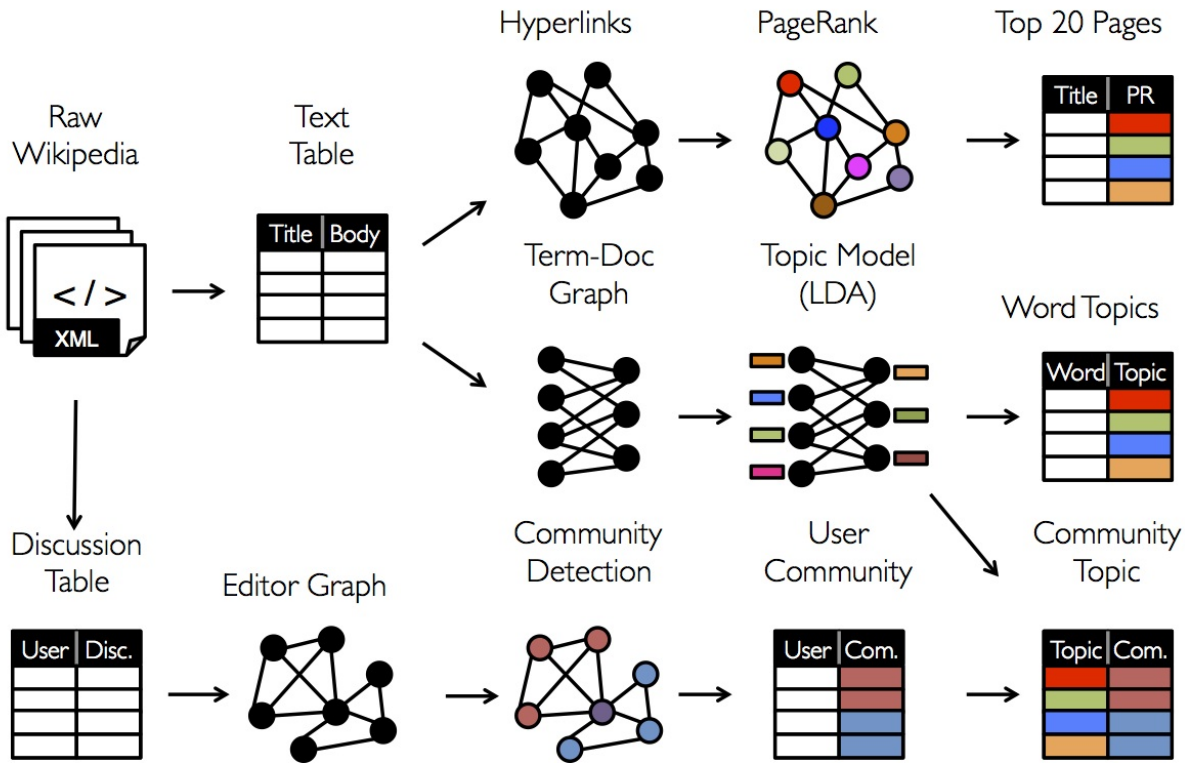
GraphX是一個新的（alpha） Spark API，它用於圖形（Graph）和平行圖形（Graph-parallel）的計算。GraphX透過引入Resilient Distributed Property Graph：一種帶有頂點和邊屬性的有向多重圖，來擴展Spark RDD。為了支援圖形的運算，GraphX公開一系列基本運算子（例如：subGraph、joinVertices、aggregateMessages）和Pregel API的優化。此外，GraphX也持續增加圖形演算法還有簡化分析圖形的工具（Builder）。

動機

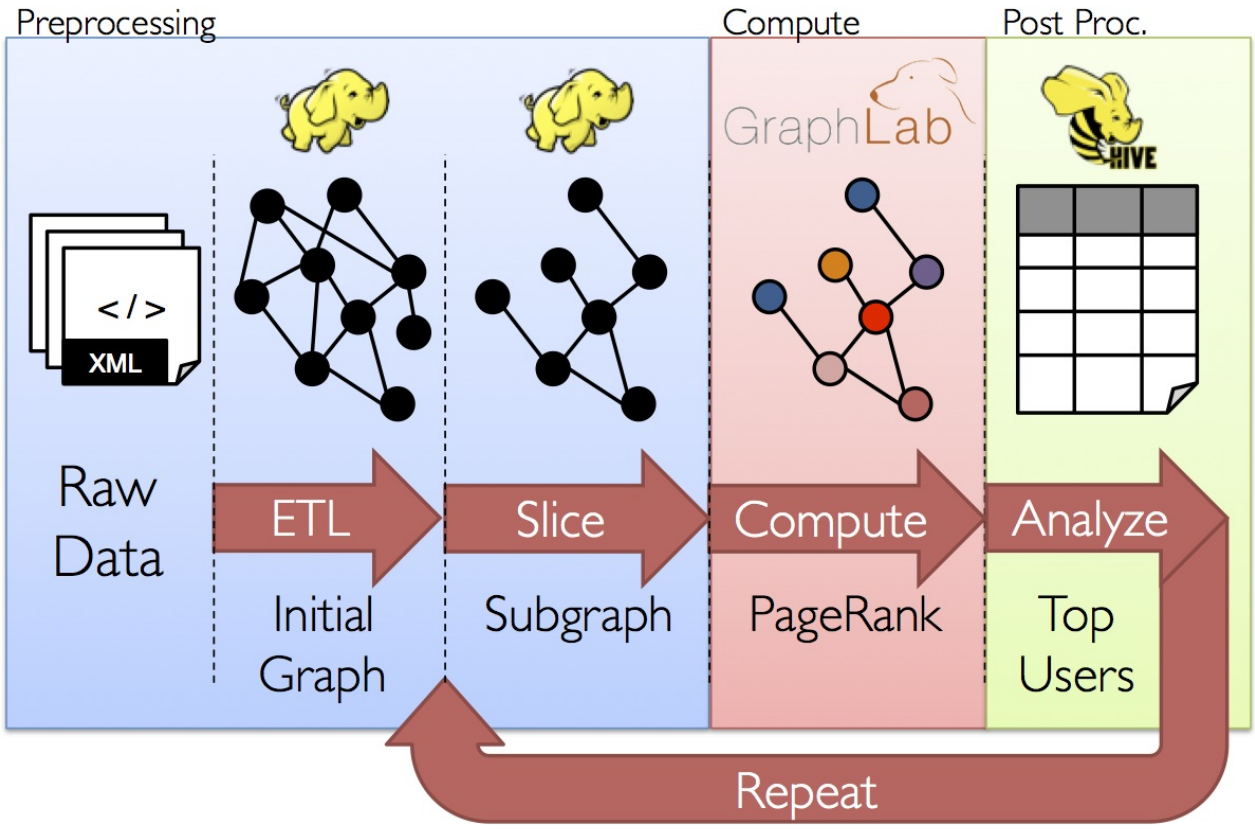
從社群媒體到語言模型，數量和重要性不斷成長的圖形結構資料推動了許多`graph-parallel`系統（例如：Giraph和GraphLab）的發展。 藉由限制可表示的運算型別和帶入新的技術來劃分和分配圖形，這些系統能夠有效率地執行複雜的圖形演算法，比一般的`data-parallel`的系統快很多。



然而，透過這種限制可以大量的提高效能，但是很難表現典型圖形分析流程：建構圖形、修改結構或是表達橫跨多個圖形的運算中很多的重要階段。另外，如何看待資料取決於我們的目標，且相同的原始資料可能有許多不同的表格和圖形。



總結來講，圖形和表格之間經常需要夠夠互相轉換。然而，現存的圖形分析流程必須撰寫 `graph-parallel` 和 `data-parallel` 系統，導致大量資料的搬移和重複還有複雜的程式模型。



GraphX的目的就是將 `graph-parallel` 和 `data-parallel` 整合成一個系統中，而且只有一個整合後的API。GraphX允許使用者將資料視為一個圖形和集合（例如: RDDs），而不需要任何的資料搬移和複製。最新的 `graph-parallel` 系統，使得GraphX能夠優化圖形指令的執行。

- 入門

- 圖形特性
- 圖形運算子
- Pregel API
- 圖形建構式
- 頂點和邊的RDDs
- 圖形演算法
- 範例

入門

第一步你需要先引入Spark和GraphX到你的專案中，如下面所示

```
import org.apache.spark._
import org.apache.spark.graphx._
// To make some of the examples work we will also need RDD
import org.apache.spark.rdd.RDD
```

如果你没有用到Spark shell，你將會需要SparkContext。若想學習更多Spark的入門知識，可以參考Spark Quick Start Guide。

屬性圖

屬性圖是一個有向多重圖，它帶有連接到每個節點和邊的使用者定義的對象。 有向多重圖中多個平行(parallel)的邊共享相同的來源和目的地節點。支持平行邊的能力簡化了建模場景，這個場景中，相同的節點存在多種關係(例如co-worker和friend)。每個節點由一个 唯一的64位元的辨識碼（VertexID）作為key。GraphX並沒有對節點辨識碼限制任何的排序。同樣，節點擁有相應的來源和目的節點辨識碼。

屬性圖通過vertex(VD)和edge(ED)類型参数化，這些類型是分別與每個節點和邊相關聯的物件類型。

在某些情况下，在相同的圖形中，可能希望節點擁有不同的屬性類型。這可以通過繼承完成。例如，將用戶和產品視為一個二分圖，我們可以用以下方式

```
class VertexProperty()
case class UserProperty(val name: String) extends VertexProperty
case class ProductProperty(val name: String, val price: Double) extends VertexProperty
// The graph might then have the type:
var graph: Graph[VertexProperty, String] = null
```

和RDD一樣，屬性圖是不可變的、分布式的、容錯的。圖的值或者結構的改變需要按期望的生成一個新的圖来實現。注意，原始圖的實質的部分(例如:不受影響的結構，屬性和索引)都可以在新圖中重用，用來減少這種內在的功能數據結構的成本。執行者使用一系列節點分區試探法來對圖進行分區。如RDD一樣，圖中的每個分區可以在發生故障的情況下被重新創建在不同的機器上。

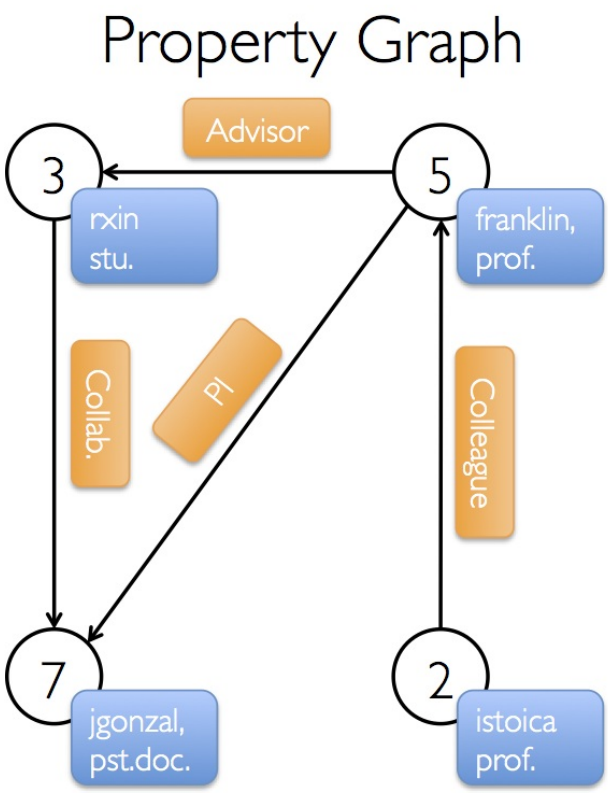
邏輯上的屬性圖對應於一對類型化的集合(RDD),這個集合編碼了每一個節點和邊的屬性。因此，圖類別包含訪問圖中的節點和邊的成員。

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

`VertexRDD[VD]` 和 `EdgeRDD[ED]` 類別分別繼承和最佳化自 `RDD[(VertexID, VD)]` 和 `RDD[Edge[ED]]`。 `VertexRDD[VD]` 和 `EdgeRDD[ED]` 都支持額外的功能來建立在圖計算和利用內部最佳化。

屬性圖的例子

在GraphX項目中，假設我們想建構以下包括不同合作者的屬性圖。節點屬性可能包含用戶名和職業．我們可以用字串在邊上標記個合作者間的關係。



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

所得的圖形將具有類型簽名

```
val userGraph: Graph[(String, String), String]
```

有很多方式從一個原始文件、RDDs建構一個屬性圖。最一般的方法是利用Graph object。 下面的程式從RDDs產生屬性圖。

```
// Assume the SparkContext has already been constructed
val sc: SparkContext
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
```

在上面的例子中，我們用到了以下Edge案例類別(case class)。邊有一個 `srcId` 和 `dstId` 分別對應於來源和目標節點的辨識碼。另外， `Edge` 類別有一個 `attr` 成員用来存儲存邊的屬性。

我們可以透過 `graph.vertices` 和 `graph.edges` 成員將一圖圖解構為相應的節點和邊。

```
val graph: Graph[(String, String), String] // Constructed from above
// Count all users which are postdocs
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count
// Count all the edges where src > dst
graph.edges.filter(e => e.srcId > e.dstId).count
```

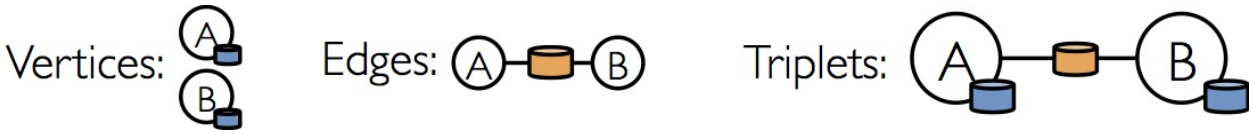
注意，graph.vertices 返回一個 VertexRDD[(String, String)]，它繼承於 RDD[(VertexID, (String, String))]。所以我們可以用以下scala
graph.edges 返回一個包含 Edge[String] 物件的 EdgeRDD。我們也可以使用案例類別建構子，如下例所示。

```
graph.edges.filter { case Edge(src, dst, prop) => src > dst }.count
```

除了屬性圖的節點和邊的檢視表，GraphX也包含了一個三元組(triplet)的檢視表，三元檢視表邏輯上將節點和邊的屬性保存為一個 `RDD[EdgeTriplet[VD, ED]]` ，它包含EdgeTriplet類別的實例。 可以通過下面的Sql表達式表示這個連接(join)。

```
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```

或者通過下面的圖來表示。



`EdgeTriplet` 類別繼承於 `Edge` 類別，並且加入了 `srcAttr` 和 `dstAttr` 成員，這兩個成員分別包含來源和目的的屬性。我們可以用以下三元組檢視表產生字串集合用來描述用戶之間的關係。

```
val graph: Graph[(String, String), String] // Constructed from above
// Use the triplets view to create an RDD of facts.
val facts: RDD[String] =
  graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1)
facts.collect.foreach(println(_)變
```

圖形操作（Graph Operators）

就像RDDs基本的操作map、filter和reduceByKey一樣，屬性圖形也具備一些基本的運算子，這些運算子採用使用者自訂義的函數並產生新轉換後的特徵和結構的新圖形。在Graph中實作了優化後的核心運算子以及GraphOps中定義的表示為核心運算子組合的快捷運算子。由於Scala中有隱式轉換，故 `GraphOps` 中的運算子可以作為 `Graph` 的成員直接使用。例如，我們可以透過下方的例子來計算每個頂點（定義在 `GraphOps` ）的內分支度。

```
val graph: Graph[(String, String), String]
// Use the implicit GraphOps.inDegrees operator
val inDegrees: VertexRDD[Int] = graph.inDegrees
```

區分核心圖形操作和 `GraphOps` 的原因是為了能在未來支援不同的圖形表示。每個圖形表示都必須提供核心操作的實作和重複使用定義在 `GraphOps` 中有用的操作。

運算子的摘要清單（Summary List Of Operators）

以下一些定義在 `Graph` 和 `GraphOps` 中的函數摘要，為了簡單起見，用 `Graph` 的成員做表示。注意，某些函數是已經經過刪簡後的（如預設參數和型別限制皆沒有列出），還有一些較為進階的函數也沒有列出，若是希望了解更多，請閱讀官方的API文件。

```
/** Summary of the functionality in the property graph */
class Graph[VD, ED] {
  // Information about the Graph ===================================================================
  val numEdges: Long
  val numVertices: Long
  val inDegrees: VertexRDD[Int]
  val outDegrees: VertexRDD[Int]
  val degrees: VertexRDD[Int]
  // Views of the graph as collections =============================================================
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
  val triplets: RDD[EdgeTriplet[VD, ED]]
  // Functions for caching graphs ==================================================================
  def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
  def cache(): Graph[VD, ED]
  def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
  // Change the partitioning heuristic  ============================================================
  def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
  // Transform vertex and edge attributes ==========================================================
  def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]): Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) => Iterator[ED2])
    : Graph[VD, ED2]
  // Modify the graph structure ====================================================================
  def reverse: Graph[VD, ED]
  def subgraph(
      epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
      vpred: (VertexID, VD) => Boolean = ((v, d) => true))
    : Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
  // Join RDDs with the graph ======================================================================
  def joinVertices[U](table: RDD[(VertexID, U)])(mapFunc: (VertexID, VD, U) => VD): Graph[VD, ED]
  def outerJoinVertices[U, VD2](other: RDD[(VertexID, U)])
      (mapFunc: (VertexID, VD, Option[U]) => VD2)
    : Graph[VD2, ED]
  // Aggregate information about adjacent triplets =================================================
  def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexID]]
  def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[Array[(VertexID, VD)]]
  def aggregateMessages[Msg: ClassTag](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
```

```
      mergeMsg: (Msg, Msg) => Msg,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[A]
  // Iterative graph-parallel computation ========================================================
  def pregel[A](initialMsg: A, maxIterations: Int, activeDirection: EdgeDirection)(
      vprog: (VertexID, VD, A) => VD,
      sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexID,A)],
      mergeMsg: (A, A) => A)
    : Graph[VD, ED]
  // Basic graph algorithms ======================================================================
  def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double]
  def connectedComponents(): Graph[VertexID, ED]
  def triangleCount(): Graph[Int, ED]
  def stronglyConnectedComponents(numIter: Int): Graph[VertexID, ED]
}
```

屬性運算子（Property Operators）

像是RDD的`map`運算子一樣，如下列所示：

```
class Graph[VD, ED] {
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
}
```

每個運算子執行後都會產生一個新的圖形，其頂點或邊的屬性都會經過使用者所定義的`map`函數而改變。

> 注意，在經過這些操作下，是不會影響到圖形的結構。這些運算子有一個重要特色，就是它會重複利用原始圖形結構的索引值。下面的兩段程式碼目的上是相同的，但是第一段並不會保存結構的索引值，這樣將無法讓GraphX系統優化。
>
> ```
> val newVertices = graph.vertices.map { case (id, attr) => (id, mapUdf(id, attr)) }
> val newGraph = Graph(newVertices, graph.edges)
> ```
>
> 另一種方法是透過mapVertices⇒VD2)(ClassTag[VD2]):Graph[VD2,ED])來保存索引。
>
> ```
> val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
> ```

這些運算子經常用來初始化作為特定計算或處理不必要的屬性的圖形。例如，給一個具有外分支度（Out-degree）屬性頂點的圖形，用於PageRank。

```
// Given a graph where the vertex property is the out degree
val inputGraph: Graph[Int, String] =
  graph.outerJoinVertices(graph.outDegrees)((vid, _, degOpt) => degOpt.getOrElse(0))
// Construct a graph where each edge contains the weight
// and each vertex is the initial PageRank
val outputGraph: Graph[Double, Double] =
  inputGraph.mapTriplets(triplet => 1.0 / triplet.srcAttr).mapVertices((id, _) => 1.0)
```

結構性運算子（Structural Operators）

目前的GraphX只有支援一組簡單的結構性運算子，我們希望未來能夠增加。下面列出了基本的結構性運算子。

```
class Graph[VD, ED] {
  def reverse: Graph[VD, ED]
  def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
               vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
}
```

reverse：此運算子將會反轉圖形內所有邊的方向並回傳反轉後的圖形。例如，這個操作可以用來計算反轉後的PageRank。由於這個操作並不會修改到頂點或是邊，也不會改變邊的數量，所以能夠在不搬移或複製資料的情況下有效率地實現。

subgraph⇒Boolean,(VertexId,VD)⇒Boolean):Graph[VD,ED])：此運算子會利用使用者給予的頂點和邊的條件（predicateds），回傳的是圖形是滿足條件的頂點和邊，以及滿足頂點條件的相連頂點。`subgraph`運算子可以在許多情況上限制有興趣的頂點和邊或刪除受損的連結。下面的範例就是說明如何刪除受損的連結。

```
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof")),
                       (4L, ("peter", "student"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi"),
                       Edge(4L, 0L, "student"),   Edge(5L, 0L, "colleague")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
// Notice that there is a user 0 (for which we have no information) connected to users
// 4 (peter) and 5 (franklin).
graph.triplets.map(
    triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1
  ).collect.foreach(println(_))
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
// The valid subgraph will disconnect users 4 and 5 by removing user 0
validGraph.vertices.collect.foreach(println(_))
validGraph.triplets.map(
    triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1
  ).collect.foreach(println(_))
```

> 注意，上面的範例中，只有提供頂點的條件。如果沒有給予頂點或邊的條件，`subgraph`運算子預設為`True`，代表不會做任何限制。

mask(ClassTag[VD2],ClassTag[ED2]):Graph[VD,ED])：此運算子會建造一個子圖形，這個子圖形具備輸入圖形的頂點和邊。可以利用`subgraph`運算子限制圖形，然後將其結果作為`mask`的遮罩來限制結果。例如，我們可以先利用有遺失的頂點來運行連通分量演算法，然後再結合`subgraph`及`mask`來取得正確的結果。

```
/ Run Connected Components
val ccGraph = graph.connectedComponents() // No longer contains missing field
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
// Restrict the answer to the valid subgraph
val validCCGraph = ccGraph.mask(validGraph)
```

groupEdges⇒ED):Graph[VD,ED])：此運算子會合併平行的邊（如一對頂點之前重複的邊）。在許多應用上，會藉由將平行的邊合併（權值合併）為一條來降低圖形的大小。

Join運算子（Join Operators）

在許多情況下，必須將外部的資料合併到圖中。例如，我們可能會想將額外的使用者資訊合併到現有的圖中或是想從一個圖中取出資訊加到另一個圖中。這些任務都可以藉由 `join` 運算子來完成。以下列出 `join` 運算子主要的功能。

```
class Graph[VD, ED] {
  def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD)
    : Graph[VD, ED]
  def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId, VD, Option[U]) => VD2)
    : Graph[VD2, ED]
}
```

joinVertices])((VertexId,VD,U)⇒VD)(ClassTag[U]):Graph[VD,ED])：此運算子會將輸入的RDD和頂點作結合，回傳一個透過使用者定義的 `map` 函數所轉換後的頂點的圖。若頂點沒有匹配值則會保留其原始值。

> 注意，對於給定的一個頂點，如果RDD有超過一個的值，而只能使用其中一個。因此建議用下列的方法來將結果預設索引值，來保證RDD的唯一性，來大量加速後續的 `join` 運算。

```
val nonUniqueCosts: RDD[(VertexID, Double)]
val uniqueCosts: VertexRDD[Double] =
  graph.vertices.aggregateUsingIndex(nonUnique, (a,b) => a + b)
val joinedGraph = graph.joinVertices(uniqueCosts)(
  (id, oldCost, extraCost) => oldCost + extraCost)
```

除了將使用者自定義的map函數套用到所有的頂點和改變頂點屬性類型外，更一般的outerJoinVertices])((VertexId,VD,Option[U])⇒VD2)(ClassTag[U],ClassTag[VD2]):Graph[VD2,ED])的用法與 `joinVertices` 類似。因為並非所有頂點在RDD中都有匹配值，map函數需要一個option型別參數。

```
val outDegrees: VertexRDD[Int] = graph.outDegrees
val degreeGraph = graph.outerJoinVertices(outDegrees) { (id, oldAttr, outDegOpt) =>
  outDegOpt match {
    case Some(outDeg) => outDeg
    case None => 0 // No outDegree means zero outDegree
  }
}
```

> 你可能已經注意到，上面的例子中用到了 `curry` 函數的多參數清單。我們能夠將f(a)(b)寫成f(a,b)，但f(a,b)表示b的型別將不會依賴於a。因此，使用者需要為自定義的函數提供型別的宣告。

```
val joinedGraph = graph.joinVertices(uniqueCosts,
  (id: VertexID, oldCost: Double, extraCost: Double) => oldCost + extraCost)
```

相鄰聚合（Neighborhood Aggregation）

圖形分析中最關鍵的步驟就是匯集每個頂點周圍的資訊。例如，我們可能想知道每個使用者的追隨者數量或是平均年齡。許多的迭代圖形演算法（如PageRank、最短路徑（Shortest Path）和連通分量（Connected Components））重複的匯集相鄰頂點（如PageRank的值、到來源的最短路徑、最小可到達的頂點id）的資訊。

> 為了改善效能，將主要的聚合運算子從 `graph.mapReduceTriplets` 改成新的 `graph.AggregateMessages`。雖然API的變化不大，但是我們仍然提高轉換的指南。

聚合訊息(aggregateMessages)

GraphX中的核心聚合運算是aggregateMessages⇒Unit,(A,A)⇒A,TripletFields)(ClassTag[A]):VertexRDD[A])。這個運算子在

圖形的每個edge triplet應用一個使用者自定義的 `sendMsg` 函數，然後也應用 `mergeMsg` 函數去匯集目標頂點的資訊。

```
class Graph[VD, ED] {
  def aggregateMessages[Msg: ClassTag](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
      mergeMsg: (Msg, Msg) => Msg,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[Msg]
}
```

使用者自定義的 `sendMsg` 函數接受一個EdgeContext型別， `EdgeContext` 透露了起始和目標的屬性以及傳送訊息給起始和目標屬性的函數（sendToSrc:Unit)和sendToDst:Unit)）。可以將 `sendMsg` 視作 `map-reduce` 中的 `map` 函數。而使用者自定義的 `mergeMsg` 函數接受兩個指定的訊息到相同的頂點並產生一個訊息，可以將 `mergeMsg` 視作 `map-reduce` 中的 `reduce` 函數。aggregateMessages⇒Unit,(A,A)⇒A,TripletFields)(ClassTag[A]):VertexRDD[A])運算子會回傳一個包含匯集訊息（Msg型別）到指定的每一個頂點的 `VertexRDD[Msg]` 。沒有接收到訊息的頂點不會包含在回傳的 `VertexRDD` 中。

另外，aggregateMessages⇒Unit,(A,A)⇒A,TripletFields)(ClassTag[A]):VertexRDD[A])接受一個可選的參數 `tripletFields` ，它顯示出在 `EdgeContext` 中，哪些資料是可被存取的（如起始頂點的屬性，而目標頂點的屬性無法）。 `tripletsFields` 可能的值都定義在TripletFields中，預設值為 `TripleetFields.All` ，其說明使用者自定義的 `sendMsg` 可存取 `EdgeContext` 的任何欄位。 `tripletFields` 參數可用來通知GraphX只有部分的 `EdgeContext` 需要允許GraphX選擇一個優化的 `Join` 策略。舉例，如果我們想要計算每個使用者的追隨者平均年齡，我們只需要起始的欄位，所以我們只需要用 `TripletFields.Src` 來表示我們只需要起始的欄位。

> 在早期GraphX的版本，我們利用位元碼檢測，作為 `TripletFields.Src` 的值，然而我們發現這樣有點不太可靠，所以挑選了更明確的用法。

在以下的範例中，我們用 `aggregateMessages` 運算子來計算每個使用者年長的追隨者的平均年齡。

```
// Import random graph generation library
import org.apache.spark.graphx.util.GraphGenerators
// Create a graph with "age" as the vertex property.  Here we use a random graph for simplicity.
val graph: Graph[Double, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapVertices( (id, _) => id.toDouble )
// Compute the number of older followers and their total age
val olderFollowers: VertexRDD[(Int, Double)] = graph.aggregateMessages[(Int, Double)](
  triplet => { // Map Function
    if (triplet.srcAttr > triplet.dstAttr) {
      // Send message to destination vertex containing counter and age
      triplet.sendToDst(1, triplet.srcAttr)
    }
  },
  // Add counter and age
  (a, b) => (a._1 + b._1, a._2 + b._2) // Reduce Function
)
// Divide total age by number of older followers to get average age of older followers
val avgAgeOfOlderFollowers: VertexRDD[Double] =
  olderFollowers.mapValues( (id, value) => value match { case (count, totalAge) => totalAge / count } )
// Display the results
avgAgeOfOlderFollowers.collect.foreach(println(_))
```

> 當訊息（或是訊息的總數）是固定常數（如福點數和加法，而不是串列和串接）時， `aggregateMessages` 的效果會最好。

Map Reduce三元組轉換指南

在早期GraphX的版本中，利用mapReduceTriplets],reduceFunc:
(A,A)=>A,activeSetOpt:Option[(org.apache.spark.graphx.VertexRDD[_],org.apache.spark.graphx.EdgeDirection)])
(implicitevidence$10:scala.reflect.ClassTag[A]):org.apache.spark.graphx.VertexRDD[A])運算子來完成相鄰聚合（Neighborhood Aggregation）。

```
class Graph[VD, ED] {
  def mapReduceTriplets[Msg](
      map: EdgeTriplet[VD, ED] => Iterator[(VertexId, Msg)],
      reduce: (Msg, Msg) => Msg)
    : VertexRDD[Msg]
}
```

mapReduceTriplets],reduceFunc:
(A,A)=>A,activeSetOpt:Option[(org.apache.spark.graphx.VertexRDD[_],org.apache.spark.graphx.EdgeDirection)])
(implicitevidence$10:scala.reflect.ClassTag[A]):org.apache.spark.graphx.VertexRDD[A])運算子接受每個三元組應用於使用者自定義的 `map` 函數，且能夠產生利用使用者自定義的 `reduce` 函數來匯集訊息。然而，我們發現使用者返回的迭代器是昂貴的，且它禁止我們添加額外的優化功能（如區域頂點的重新編號）。在aggregateMessages⇒Unit,(A,A)⇒A,TripletFields)
(ClassTag[A]):VertexRDD[A])中，我們介紹了 `EdgeContext`，它透露三元組欄位和函數來更明確的傳送訊息給起始和目標的頂點。因此，我們移除了位元碼檢測，而要求使用者明確的指出三元組的哪些欄位是實際上使用的。

以下是利用了 `mapReduceTriplets` 的範例：

```
val graph: Graph[Int, Float] = ...
def msgFun(triplet: Triplet[Int, Float]): Iterator[(Int, String)] = {
  Iterator((triplet.dstId, "Hi"))
}
def reduceFun(a: Int, b: Int): Int = a + b
val result = graph.mapReduceTriplets[String](msgFun, reduceFun)
```

也等效於以下使用 `aggregateMessages` 的範例：

```
val graph: Graph[Int, Float] = ...
def msgFun(triplet: EdgeContext[Int, Float, String]) {
  triplet.sendToDst("Hi")
}
def reduceFun(a: Int, b: Int): Int = a + b
val result = graph.aggregateMessages[String](msgFun, reduceFun)
```

計算分支度（degree）資訊

最一般的聚合任務就是計算每一個頂點的分支度數，就是每個頂點相鄰邊的數量。在有向圖中，經常需要知道頂點的內分支度、外分支度及分支度的總數。GraphOps類別中，有一系列的運算子來計算每個頂點的分支度。例如，以下的範例是計算最大的內分支度、外分支度和分支度的總數。

```
// Define a reduce operation to compute the highest degree vertex
def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
  if (a._2 > b._2) a else b
}
// Compute the max degrees
val maxInDegree: (VertexId, Int)  = graph.inDegrees.reduce(max)
val maxOutDegree: (VertexId, Int) = graph.outDegrees.reduce(max)
val maxDegrees: (VertexId, Int)   = graph.degrees.reduce(max)
```

Collecting Neighbors

在某些情形下，透過收集每個頂點相鄰的頂點及他們的屬性來代替計算會更容易。這可以透過collectNeighborIds:VertexRDD[Array[VertexId]])運算子完成。

```
class GraphOps[VD, ED] {
  def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexId]]
  def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[ Array[(VertexId, VD)] ]
}
```

> 這些運算子是非常昂貴的，因為需要複製資訊及大量的通訊。如果可能，盡量使用`aggregateMessages`來直接替代相同的計算。

暫存與否

在Spark中，RDDs在預設下是不會一直存在記憶體中。為了避免重複運算，當要多次使用它們，則必須明確的將它們暫存起來。而Graphs在GraphX中的行為就像是RDDs一樣。當Graphs需要被多次使用，記得先呼叫`Graph.cache()`。

在迭代運算中，為了得到最佳的效能，不暫存是必須的。在預設情況下，暫存的RDDs和Graphs會一直保留在記憶體中，直到記憶體將它們釋放（利用LRU演算法）。對於迭代運算中，先前計算的結果也會暫存在記憶體中。雖然最終都會被釋放，但是暫存不需要的資料在記憶體中會減慢垃圾回收（Garbage collection）速度。若中間產生出來的結果不暫存，則會提升整體的效率。這牽扯到每次迭代中實體化一個Graph或者RDD，且不暫存其他的資料集，在未來的迭代中僅僅使用實體化的資料集。對於迭代的計算，我們推薦**Pregel API**，它能適時的將中間結果釋放。

Pregel API

圖本身是遞迴型的資料結構，因為頂點的屬性依賴於其鄰居的屬性，這些鄰居的屬性又依賴於其鄰居的屬性。結論是，許多重要的圖形演算法，都需要重複的重新計算每個頂點的屬性，直到滿足某個確定的條件。一系列的Graph-parallel抽象體已經背提出來代表這些迭代型演算法。GraphX公開了一個Pregel-like的運算子，它是廣泛使用Pregel和GraphLab抽象的一個融合。

在GraphX中，更高階層的Pregel運算子是一個限制到圖拓僕的批量同步（bulk-synchronous）。Pregel運算子執行一系列的super-steps，再這些步驟中，頂點從之前的super-steps中接收進入訊息的總和，為頂點的屬性計算一個新值，然後在下一個super-step中發送訊息到相鄰的頂點。不像Pregel而更像GraphLab，訊息被作為一個邊三元組的函數平行的運算，且訊息運算會存取來源和目標頂點的特徵。在super-step中，未收到訊息的頂點會被跳過。當沒有任何訊息遺留時，Pregel運算子會停止迭代且回傳最後的圖。

> 注意，不像標準的Pregel實作，GraphX中的頂點只能夠發送訊息給相鄰頂點，且利用使用者自訂的通知函數來平行完成訊息的建立。這些限制允許了GraphX進行額外的優化。

以下是Pregel操作((VertexId,VD,A)⇒VD,(EdgeTriplet[VD,ED])⇒Iterator[(VertexId,A)],(A,A)⇒A)(ClassTag[A]):Graph[VD,ED])的型別簽章（signature）以及實做的草圖（注意，graph.cache呼叫已經移除了）

```
class GraphOps[VD, ED] {
  def pregel[A]
      (initialMsg: A,
       maxIter: Int = Int.MaxValue,
       activeDir: EdgeDirection = EdgeDirection.Out)
      (vprog: (VertexId, VD, A) => VD,
       sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
       mergeMsg: (A, A) => A)
    : Graph[VD, ED] = {
    // Receive the initial message at each vertex
    var g = mapVertices( (vid, vdata) => vprog(vid, vdata, initialMsg) ).cache()
    // compute the messages
    var messages = g.mapReduceTriplets(sendMsg, mergeMsg)
    var activeMessages = messages.count()
    // Loop until no messages remain or maxIterations is achieved
    var i = 0
    while (activeMessages > 0 && i < maxIterations) {
      // Receive the messages: -----------------------------------------------------------------------
      // Run the vertex program on all vertices that receive messages
      val newVerts = g.vertices.innerJoin(messages)(vprog).cache()
      // Merge the new vertex values back into the graph
      g = g.outerJoinVertices(newVerts) { (vid, old, newOpt) => newOpt.getOrElse(old) }.cache()
      // Send Messages: ------------------------------------------------------------------------------
      // Vertices that didn't receive a message above don't appear in newVerts and therefore don't
      // get to send messages.  More precisely the map phase of mapReduceTriplets is only invoked
      // on edges in the activeDir of vertices in newVerts
      messages = g.mapReduceTriplets(sendMsg, mergeMsg, Some((newVerts, activeDir))).cache()
      activeMessages = messages.count()
      i += 1
    }
    g
  }
}
```

注意，Pregel接受兩個參數列表（graph.pregel(list1)(list2)）。第一個參數列表包含了配置參數，如初始訊息、最大的迭代數、訊息發送邊的方向（預設向外）。第二個參數列表包含了用來接收訊息（vprog）、計算訊息（sendMsg）、合併訊息（mergeMsg）。

以下範例是我們可以使用Pregel運算子來表示單源最短路徑（Single source shortest path）的運算。

```
import org.apache.spark.graphx._
// Import random graph generation library
```

```
import org.apache.spark.graphx.util.GraphGenerators
// A graph with edge attributes containing distances
val graph: Graph[Int, Double] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapEdges(e => e.attr.toDouble)
val sourceId: VertexId = 42 // The ultimate source
// Initialize the graph such that all vertices except the root have distance infinity.
val initialGraph = graph.mapVertices((id, _) => if (id == sourceId) 0.0 else Double.PositiveInfinity)
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
  triplet => {  // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  },
  (a,b) => math.min(a,b) // Merge Message
  )
println(sssp.vertices.collect.mkString("\n"))
```

圖形建構式

GraphX提供了幾種方式從RDD或是硬碟上的頂點和邊建立圖。在預設情況下，圖形建構式不會為圖的邊重新分割，而是把邊保留在預設的區塊中（例如HDFS的原始區塊）。 Graph.groupEdges⇒ED):Graph[VD,ED])要求重新分割圖形，因為它假定相同的邊會被分配到同一個區塊，所以你必須在使用 `groupEdges` 前使用Graph.partitionBy:Graph[VD,ED])

```
object GraphLoader {
  def edgeListFile(
      sc: SparkContext,
      path: String,
      canonicalOrientation: Boolean = false,
      minEdgePartitions: Int = 1)
    : Graph[Int, Int]
}
```

GraphLoader.edgeListFile:Graph[Int,Int])提供了一個從硬碟上邊的清單讀取一個圖形的方式。格式如下面範例（起始頂點ID，目標頂點ID）， `#` 表示註解行。

```
# This is a comment
2 1
4 1
1 2
```

從指定的邊建立一個圖，自動建立所有邊提及的所有頂點。所有的頂點和邊的屬性預設都是1。 `canonicalOrientation` 參數允許重新導向正向（srcId < dstId）的邊。這在connected components演算法中需要用到。 `minEdgePartitions` 參數用來規定邊的分區生成的最小數量。邊分區可能比指定的分區還要多。例如，一個HDFS檔案有更多的區塊。

```
object Graph {
  def apply[VD, ED](
      vertices: RDD[(VertexId, VD)],
      edges: RDD[Edge[ED]],
      defaultVertexAttr: VD = null)
    : Graph[VD, ED]
  def fromEdges[VD, ED](
      edges: RDD[Edge[ED]],
      defaultValue: VD): Graph[VD, ED]
  def fromEdgeTuples[VD](
      rawEdges: RDD[(VertexId, VertexId)],
      defaultValue: VD,
      uniqueEdges: Option[PartitionStrategy] = None): Graph[VD, Int]
}
```

Graph.apply],RDD[Edge[ED]],VD)(ClassTag[VD],ClassTag[ED]):Graph[VD,ED])允許從頂點和邊的RDD上建立一個圖。重複的頂點會被任意地挑出，而只有從邊RDD出來的頂點才會有預設屬性，頂點RDD並不會有。

Graph.fromEdges(ClassTag[VD],ClassTag[ED]):Graph[VD,ED])只允許從一個邊的RDD上建立一個圖，且自動地建立邊提及的頂點，並給予這些頂點預設值。

Graph.fromEdgeTuples],VD,Option[PartitionStrategy])(ClassTag[VD]):Graph[VD,Int])只允許一個edge tuple組成的RDD上建立一個圖，並給予邊的值為1。自動地建立邊所提及的頂點，並給予這些頂點預設值。它還支援刪除邊，為了刪除邊，需要傳遞一個PartitionStrategy值為 `Some` 作為參數 `uniqueEdges` 的值（如uniqueEdges = some(PartitionStrategy.RandomVertexCut)），要刪除同一分區相同的邊，一個分割策略是必須的。

頂點和邊的RDDs

GraphX提供了儲存在圖中的頂點和邊的RDD。因為GraphX將頂點和邊保存在優化過的資料結構中，這些資料結構提供了額外的功能，分別傳回 `VertexRDD` 和 `EdgeRDD` 。這一章節，我們將學習它們一些有用的功能。

VertexRDDs

`VertexRDD[A]` 繼承了 `RDD[(VertexID, A)]` 並且新增了額外的限制條件，那就是每個 `VertexID` 只能出現一次。此外， `VertexRDD[A]` 代表一組具有型別A特性的頂點。在程式內部，透過將頂點屬性儲存到一個可重複使用的hash-map的資料結構來達成。所以，若兩個 `VertexRDDs` 是從相同的 `VertexRDD` (如藉由 `filter` 或 `mapValues`)基底產生的，它們就能夠在常數時間內完成合併，而避免了hash的計算。為了利用索引式的資料結構， `VertexRDD` 提供了下列的附加功能：

```
class VertexRDD[VD] extends RDD[(VertexID, VD)] {
  // Filter the vertex set but preserves the internal index
  def filter(pred: Tuple2[VertexId, VD] => Boolean): VertexRDD[VD]
  // Transform the values without changing the ids (preserves the internal index)
  def mapValues[VD2](map: VD => VD2): VertexRDD[VD2]
  def mapValues[VD2](map: (VertexId, VD) => VD2): VertexRDD[VD2]
  // Remove vertices from this set that appear in the other set
  def diff(other: VertexRDD[VD]): VertexRDD[VD]
  // Join operators that take advantage of the internal indexing to accelerate joins (substantially)
  def leftJoin[VD2, VD3](other: RDD[(VertexId, VD2)])(f: (VertexId, VD, Option[VD2]) => VD3): VertexRDD[VD3]
  def innerJoin[U, VD2](other: RDD[(VertexId, U)])(f: (VertexId, VD, U) => VD2): VertexRDD[VD2]
  // Use the index on this RDD to accelerate a `reduceByKey` operation on the input RDD.
  def aggregateUsingIndex[VD2](other: RDD[(VertexId, VD2)], reduceFunc: (VD2, VD2) => VD2): VertexRDD[VD2]
}
```

舉例， `filter` 運算子是如何回一個 `VertexRDD` 。 `filter` 實際上是由 `BitSet` 實作，因此重複使用索引值以及保留快速與其他 `VertexRDDs` 合併的能力。相同的， `mapValues` 運算子不允許 `map` 函數改變 `VertexID` ，確保相同的 `hashMap` 的資料結構被重複使用。當合併兩個從相同 `hashMap` 得到的 `VertexRDDs` 且利用線性搜尋（linear scan）而非花費時間較長的點查詢（point lookups）來實現合併時， `leftJoin` 和 `innerJoin` 都能夠使用。

`aggregateUsingIndex` 運算子能夠有效率地將一個 `RDD[(VertexID, A)]` 建造成一個新的 `VertexRDD` 。概念上，我透過一組為一些 `VertexRDD[A]` 的 `super-set` 頂點建造了 `VertexRDD[B]` ，那麼我們就能夠在聚合（aggregate）和往後查詢 `RDD[(VertexID, A)]` 時重複使用索引。例如：

```
val setA: VertexRDD[Int] = VertexRDD(sc.parallelize(0L until 100L).map(id => (id, 1)))
val rddB: RDD[(VertexId, Double)] = sc.parallelize(0L until 100L).flatMap(id => List((id, 1.0), (id, 2.0)))
// There should be 200 entries in rddB
rddB.count
val setB: VertexRDD[Double] = setA.aggregateUsingIndex(rddB, _ + _)
// There should be 100 entries in setB
setB.count
// Joining A and B should now be fast!
val setC: VertexRDD[Double] = setA.innerJoin(setB)((id, a, b) => a + b)
```

EdgeRDDs

`EdgeRDD[ED]` 繼承了 `RDD[Edge[ED]]` ，使用定義在PartitionStrategy眾多分割方法其中一種，將邊作區塊性的分割。在每個分區中，邊的屬性和周遭結構會被個別的儲存，能夠在屬性改變時，最大化重用。

`EdgeRDD` 揭示了三個額外的函數：

```
// Transform the edge attributes while preserving the structure
```

```
  def mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2]
  // Revere the edges reusing both attributes and structure
  def reverse: EdgeRDD[ED]
  // Join two `EdgeRDD`s partitioned using the same partitioning strategy.
  def innerJoin[ED2, ED3](other: EdgeRDD[ED2])(f: (VertexId, VertexId, ED, ED2) => ED3): EdgeRDD[ED3]
```

在多數的應用中，我們會發現 `EdgeRDD` 的操作可以透過圖形運算子（graph operators）或是定義在基本RDD中的操作來完成。

Optimized Representation

圖形演算法

GraphX具備一系列的圖形演算法來簡化圖形分析的任務。這些演算法都在`org.apache.spark.graphx.lib`，可以直接透過`Graph`中的GraphOps來取得。這章節將會描述如何使用這些圖形演算法。

PageRank

PageRank是用來衡量一個圖形中每個頂點的重要程度，假設有一條從u到v的邊，這條邊稱為u給v的重要性指標。例如，一個Twitter使用者有許多追隨者，如此一來，可以認為這名使用者相當重要。

在GraphX中的PageRank object實作了靜態和動態PageRank的方法。靜態的PageRank會在固定的次數內運行，而動態的PageRank則會一直運行，直到收斂。GraphOps允許直接呼叫這些方法。

GraphX內有一個範例，可以讓我們直接在社群媒體資料集上運行PageRank演算法。 使用者的資料在`graphx/data/users.txt`，使用者之間的關係在`graphx/data/followers.txt`中。我們可以透過以下來計算出每個使用者的PageRank。

```
// Load the edges as a graph
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")
// Run PageRank
val ranks = graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
val users = sc.textFile("graphx/data/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}
// Print the result
println(ranksByUsername.collect().mkString("\n"))
```

連通分量演算法

連通分量演算法利用連通分量中編號最小的頂點的ID來作為其的標籤。例如，在社群媒體中，連通分量可以近似為一個群聚。在GraphX中的ConnectedComponents object有這個演算法實作，我們可以透過下面的範例來完成。

```
// Load the graph as in the PageRank example
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")
// Find the connected components
val cc = graph.connectedComponents().vertices
// Join the connected components with the usernames
val users = sc.textFile("graphx/data/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ccByUsername = users.join(cc).map {
  case (id, (username, cc)) => (username, cc)
}
// Print the result
println(ccByUsername.collect().mkString("\n"))
```

三角形計數演算法

若一個頂點有兩個相鄰的頂點且和它們有邊相連，那麼這個頂點就是三角形的一部分。GraphX中的TriangleCount object實

作了演算法，它計算通過每個頂點的三角形數量，用來衡量群聚。需要注意的 `TriangleCount` 要求邊的方向是按照規定的方向（srcId < dstId）並且圖形是利用 `Graph.partitionBy` 所切開的。

```
// Load the edges in canonical order and partition the graph for triangle count
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt", true).partitionBy(PartitionStrategy.RandomVertexC
// Find the triangle count for each vertex
val triCounts = graph.triangleCount().vertices
// Join the triangle counts with the usernames
val users = sc.textFile("graphx/data/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map { case (id, (username, tc)) =>
  (username, tc)
}
// Print the result
println(triCountByUsername.collect().mkString("\n"))
```

範例

假設我們想要利用一些檔案來建構一個圖形，利用重要的關係和使用者來限制圖形，並且在其上面執行page-rank，最後回傳與Top使用者相關的屬性。我們可以透過如下方式實現。

```
// Connect to the Spark cluster
val sc = new SparkContext("spark://master.amplab.org", "research")

// Load my user data and parse into tuples of user id and attribute list
val users = (sc.textFile("graphx/data/users.txt")
  .map(line => line.split(",")).map( parts => (parts.head.toLong, parts.tail) ))

// Parse the edge data which is already in userId -> userId format
val followerGraph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")

// Attach the user attributes
val graph = followerGraph.outerJoinVertices(users) {
  case (uid, deg, Some(attrList)) => attrList
  // Some users may not have attributes so we set them as empty
  case (uid, deg, None) => Array.empty[String]
}

// Restrict the graph to users with usernames and names
val subgraph = graph.subgraph(vpred = (vid, attr) => attr.size == 2)

// Compute the PageRank
val pagerankGraph = subgraph.pageRank(0.001)

// Get the attributes of the top pagerank users
val userInfoWithPageRank = subgraph.outerJoinVertices(pagerankGraph.vertices) {
  case (uid, attrList, Some(pr)) => (pr, attrList.toList)
  case (uid, attrList, None) => (0.0, attrList.toList)
}

println(userInfoWithPageRank.vertices.top(5)(Ordering.by(_._2._1)).mkString("\n"))
```

提交應用程式

在Spark bin 目錄下的 `spark-submit` 讓你在集群上啟動應用程式。它可以通過統一接口使用Spark 支援的所有 集群管理器，所有你不必為每個管理器作相對應的設定。

用spark-submit 啟動應用程式

`bin/spark-submit` 指令負責建立包含Spark 以及他所相依的類別路徑(classpath)，他支援不同的集群管理器以及Spark 支持的加載模式。

```
./bin/spark-submit \
  --class
  --master  \
  --deploy-mode  \
  --conf = \
  ... # other options
   \
  [application-arguments]
```

常用選項如下：

- `--class`：你的應用程式入口點(如org.apache.spark.examples.SparkPi)
- `--master`：集群的master URL(如spark://23.195.26.187:7077)
- `--deploy-mode`：在worker 節點部署你的driver(cluster) 或者本地作為外部客戶端(client)。預設是client。
- `--conf`：自定的Spark 配置屬性，格式是key=value。
- `application-jar`：包含應用程式以及其相依的jar 包路徑。這個URL 必須在集群中全域可用，例如，存放在所有節點的 `hdfs://` 路徑或是 `file://` 路徑
- `application-arguments`：傳遞給主類別的參數

常見的部署策略是從網路提交你的應用程式。這種設定之下，適合 `client` 模式。在 `client` 模式，driver直接在 `spark-submit` 中啟動。這樣的方式直接作為集群客戶端。由於應用程式的輸入與輸出都與控制台相連，所以也適合與 REPL 的應用程式。

另一種選擇，如果你的應用程式從一個與worker 機器距離很遠的機器上提交，一般情況下，用 `cluster` 模式可減少drivers和executors 的網路延遲。注意，`cluster` 模式目前不支援獨立集群、mesos集群以及python應用程式。

有幾個我們使用的集群管理特有的選項。例如，在Spark讀立即群的 `cluster` 模式下，你也可以指定 `--supervise` 以確保driver 自動重新啟動(如果它因為發生錯誤而退出失敗)。 為了列出spark-submit 所有可用參數，用 `--help` 執行。

```
# Run application locally on 8 cores
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  /path/to/examples.jar \
  100

# Run on a Spark Standalone cluster in client deploy mode
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000

# Run on a Spark Standalone cluster in cluster deploy mode with supervise
./bin/spark-submit \
```

```
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster
  --supervise
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000

# Run on a YARN cluster
export HADOOP_CONF_DIR=XXX
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn-cluster \  # can also be `yarn-client` for client mode
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000

# Run a Python application on a Spark Standalone cluster
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000
```

Master URLs

傳送給Spark 的url 可用下列模式

Master URL	Meaning
local	用一個worker 本地執行Spark
local[K]	用k 個worker 本地執行Spark (理想情況下，數值為機器CPU 的數量)
local[*]	有多少worker 就用多少，以本地執行Spark
spark://HOST:PORT	連結到指定Spark 獨立集群master。端口必須是master 配置的端口，預設是7077
mesos://HOST:PORT	連結到指定的mesos 集群
yarn-client	以 `client` 模式連結到Yarn 集群。集群位置設定在變數HADOOP_CONF_DIR
yarn-cluster	以 `cluster` 模式連結到Yarn 集群。集群位置設定在變數HADOOP_CONF_DIR

Spark独立部署模式

安装Spark独立模式集群

安装Spark独立模式，你只需要将Spark的编译版本简单的放到集群的每个节点。你可以获得每个稳定版本的预编译版本，也可以自己编译。

手动启动集群

你能够通过下面的方式启动独立的master服务器。

```
./sbin/start-master.sh
```

一旦启动，master将会为自己打印出 `spark://HOST:PORT` URL，你能够用它连接到workers或者作为"master"参数传递给 `SparkContext` 。你也可以在master web UI上发现这个URL， master web UI默认的地址是 `http://localhost:8080` 。

相同的，你也可以启动一个或者多个workers或者将它们连接到master。

```
./bin/spark-class org.apache.spark.deploy.worker.Worker spark://IP:PORT
```

一旦你启动了一个worker，查看master web UI。你可以看到新的节点列表以及节点的CPU数以及内存。

下面的配置参数可以传递给master和worker。

Argument	Meaning
-h HOST, --host HOST	监听的主机名
-i HOST, --ip HOST	同上，已经被淘汰
-p PORT, --port PORT	监听的服务的端口（master默认是7077，worker随机）
--webui-port PORT	web UI的端口(master默认是8080，worker默认是8081)
-c CORES, --cores CORES	Spark应用程序可以使用的CPU核数（默认是所有可用）；这个选项仅在worker上可用
-m MEM, --memory MEM	Spark应用程序可以使用的内存数（默认情况是你的机器内存数减去1g）；这个选项仅在worker上可用
-d DIR, --work-dir DIR	用于暂存空间和工作输出日志的目录（默认是SPARK_HOME/work）；这个选项仅在worker上可用
--properties-file FILE	自定义的Spark配置文件的加载目录（默认是conf/spark-defaults.conf）

集群启动脚本

为了用启动脚本启动Spark独立集群，你应该在你的Spark目录下建立一个名为 `conf/slaves` 的文件，这个文件必须包含所有你要启动的Spark worker所在机器的主机名，一行一个。如果 `conf/slaves` 不存在，启动脚本默认为单个机器（localhost），这台机器对于测试是有用的。注意，master机器通过ssh访问所有的worker。在默认情况下，SSH是并行运行，需要设置无密码（采用私有密钥）的访问。 如果你没有设置为无密码访问，你可以设置环境变量 `SPARK_SSH_FOREGROUND` ，为每个worker提供密码。

一旦你設置了這个文件，你就可以通过下面的shell脚本启动或者停止你的集群。

- sbin/start-master.sh：在机器上启动一个master实例
- sbin/start-slaves.sh：在每台机器上启动一个slave实例
- sbin/start-all.sh：同时启动一个master实例和所有slave实例
- sbin/stop-master.sh：停止master实例
- sbin/stop-slaves.sh：停止所有slave实例
- sbin/stop-all.sh：停止master实例和所有slave实例

注意，这些脚本必须在你的Spark master运行的机器上执行，而不是在你的本地机器上面。

你可以在`conf/spark-env.sh`中设置环境变量进一步配置集群。利用`conf/spark-env.sh.template`创建这个文件，然后将它复制到所有的worker机器上使设置有效。下面的设置可以起作用：

Environment Variable	Meaning
SPARK_MASTER_IP	绑定master到一个指定的ip地址
SPARK_MASTER_PORT	在不同的端口上启动master（默认是7077）
SPARK_MASTER_WEBUI_PORT	master web UI的端口（默认是8080）
SPARK_MASTER_OPTS	应用到master的配置属性，格式是 "-Dx=y"（默认是none），查看下面的表格的选项以组成一个可能的列表
SPARK_LOCAL_DIRS	Spark中暂存空间的目录。包括map的输出文件和存储在磁盘上的RDDs(including map output files and RDDs that get stored on disk)。这必须在一个快速的、你的系统的本地磁盘上。它可以是一个逗号分隔的列表，代表不同磁盘的多个目录
SPARK_WORKER_CORES	Spark应用程序可以用到的核心数（默认是所有可用）
SPARK_WORKER_MEMORY	Spark应用程序用到的内存总数（默认是内存总数减去1G）。注意，每个应用程序个体的内存通过`spark.executor.memory`设置
SPARK_WORKER_PORT	在指定的端口上启动Spark worker(默认是随机)
SPARK_WORKER_WEBUI_PORT	worker UI的端口（默认是8081）
SPARK_WORKER_INSTANCES	每台机器运行的worker实例数，默认是1。如果你有一台非常大的机器并且希望运行多个worker，你可以设置这个数大于1。如果你设置了这个环境变量，确保你也设置了`SPARK_WORKER_CORES`环境变量用于限制每个worker的核数或者每个worker尝试使用所有的核。
SPARK_WORKER_DIR	Spark worker运行目录，该目录包括日志和暂存空间（默认是SPARK_HOME/work）
SPARK_WORKER_OPTS	应用到worker的配置属性，格式是 "-Dx=y"（默认是none），查看下面表格的选项以组成一个可能的列表
SPARK_DAEMON_MEMORY	分配给Spark master和worker守护进程的内存（默认是512m）
SPARK_DAEMON_JAVA_OPTS	Spark master和worker守护进程的JVM选项，格式是"-Dx=y"（默认为none）
SPARK_PUBLIC_DNS	Spark master和worker公共的DNS名（默认是none）

注意，启动脚本还不支持windows。为了在windows上启动Spark集群，需要手动启动master和workers。

`SPARK_MASTER_OPTS`支持一下的系统属性：

Property Name	Default	Meaning
spark.deploy.retainedApplications	200	展示完成的应用程序的最大数目。老的应用程序会被删除以满足该限制
spark.deploy.retainedDrivers	200	展示完成的drivers的最大数目。老的应用程序会被删除以满足该限

		制
spark.deploy.spreadOut	true	这个选项控制独立的集群管理器是应该跨节点传递应用程序还是应努力将程序整合到尽可能少的节点上。在HDFS中，传递程序是数据本地化更好的选择，但是，对于计算密集型的负载，整合会更有效率。
spark.deploy.defaultCores	(infinite)	在Spark独立模式下，给应用程序的默认核数（如果没有设置 `spark.cores.max` ）。如果没有设置，应用程序总数获得所有可用的核，除非设置了 `spark.cores.max` 。在共享集群上设置较低的核数，可用防止用户默认抓住整个集群。
spark.worker.timeout	60	独立部署的master认为worker失败（没有收到心跳信息）的间隔时间。

`SPARK_WORKER_OPTS` 支持的系统属性：

Property Name	Default	Meaning
spark.worker.cleanup.enabled	false	周期性的清空worker/应用程序目录。注意，这仅仅影响独立部署模式。不管应用程序是否还在执行，用于程序目录都会被清空
spark.worker.cleanup.interval	1800 (30分)	在本地机器上，worker清空老的应用程序工作目录的时间间隔
spark.worker.cleanup.appDataTtl	7 *24* 3600 (7天)	每个worker中应用程序工作目录的保留时间。这个时间依赖于你可用磁盘空间的大小。应用程序日志和jar包上传到每个应用程序的工作目录。随着时间的推移，工作目录会很快的填满磁盘空间，特别是如果你运行的作业很频繁。

连接一个应用程序到集群中

为了在Spark集群中运行一个应用程序，简单地传递 `spark://IP:PORT` URL到SparkContext

为了在集群上运行一个交互式的Spark shell，运行一下命令：

```
./bin/spark-shell --master spark://IP:PORT
```

你也可以传递一个选项 `--total-executor-cores <numCores>` 去控制spark-shell的核数。

启动**Spark**应用程序

spark-submit脚本支持最直接的提交一个Spark应用程序到集群。对于独立部署的集群，Spark目前支持两种部署模式。在 `client` 模式中，driver启动进程与 客户端提交应用程序所在的进程是同一个进程。然而，在 `cluster` 模式中，driver在集群的某个worker进程中启动，只有客户端进程完成了提交任务，它不会等到应用程序完成就会退出。

如果你的应用程序通过Spark submit启动，你的应用程序jar包将会自动分发到所有的worker节点。对于你的应用程序依赖的其它jar包，你应该用 `--jars` 符号指定（如 `--jars jar1,jar2` ）。

另外， `cluster` 模式支持自动的重启你的应用程序（如果程序一非零的退出码退出）。为了用这个特征，当启动应用程序时，你可以传递 `--supervise` 符号到 `spark-submit` 。如果你想杀死反复失败的应用， 你可以通过如下的方式：

```
./bin/spark-class org.apache.spark.deploy.Client kill
```

你可以在独立部署的Master web UI（http://:8080）中找到driver ID。

资源调度

独立部署的集群模式仅仅支持简单的FIFO调度器。然而，为了允许多个并行的用户，你能够控制每个应用程序能用的最大资源数。在默认情况下，它将获得集群的所有核，这只有在某一时刻只 允许一个应用程序才有意义。你可以通过 `spark.cores.max` 在SparkConf中设置核数。

```
val conf = new SparkConf()
             .setMaster(...)
             .setAppName(...)
             .set("spark.cores.max", "10")
val sc = new SparkContext(conf)
```

另外，你可以在集群的master进程中配置 `spark.deploy.defaultCores` 来改变默认的值。在 `conf/spark-env.sh` 添加下面的行：

```
export SPARK_MASTER_OPTS="-Dspark.deploy.defaultCores="
```

这在用户没有配置最大核数的共享集群中是有用的。

高可用

默认情况下，独立的调度集群对worker失败是有弹性的（在Spark本身的范围内是有弹性的，对丢失的工作通过转移它到另外的worker来解决）。然而，调度器通过master去执行调度决定， 这会造成单点故障：如果master死了，新的应用程序就无法创建。为了避免这个，我们有两个高可用的模式。

用ZooKeeper的备用master

利用ZooKeeper去支持领导选举以及一些状态存储，你能够在你的集群中启动多个master，这些master连接到同一个ZooKeeper实例上。一个被选为“领导”，其它的保持备用模式。如果当前 的领导死了，另一个master将会被选中，恢复老master的状态，然后恢复调度。整个的恢复过程大概需要1到2分钟。注意，这个恢复时间仅仅会影响调度新的应用程序-运行在失败master中的 应用程序不受影响。

配置

为了开启这个恢复模式，你可以用下面的属性在 `spark-env` 中设置 `SPARK_DAEMON_JAVA_OPTS` 。

System property	Meaning
spark.deploy.recoveryMode	设置ZOOKEEPER去启动备用master模式（默认为none）
spark.deploy.zookeeper.url	zookeeper集群url(如192.168.1.100:2181,192.168.1.101:2181)
spark.deploy.zookeeper.dir	zookeeper保存恢复状态的目录（默认是/spark）

可能的陷阱：如果你在集群中有多个masters，但是没有用zookeeper正确的配置这些masters，这些masters不会发现彼此，会认为它们都是leaders。这将会造成一个不健康的集群状态（因为所有的master都会独立的调度）。

细节

zookeeper集群启动之后，开启高可用是简单的。在相同的zookeeper配置（zookeeper URL和目录）下，在不同的节点上简单地启动多个master进程。master可以随时添加和删除。

为了调度新的应用程序或者添加worker到集群，它需要知道当前leader的IP地址。这可以通过简单的传递一个master列表来完成。例如，你可能启动你的SparkContext指向 `spark://host1:port1,host2:port2` 。 这将造成你的SparkContext同时注册这

两个master-如果 `host1` 死了，这个配置文件将一直是正确的，因为我们将找到新的leader- `host2` 。

"registering with a Master"和正常操作之间有重要的区别。当启动时，一个应用程序或者worker需要能够发现和注册当前的leader master。一旦它成功注册，它就在系统中了。如果 错误发生，新的leader将会接触所有之前注册的应用程序和worker，通知他们领导关系的变化，所以它们甚至不需要事先知道新启动的leader的存在。

由于这个属性的存在，新的master可以在任何时候创建。你唯一需要担心的问题是新的应用程序和workers能够发现它并将它注册进来以防它成为leader master。

用本地文件系统做单节点恢复

zookeeper是生产环境下最好的选择，但是如果你想在master死掉后重启它， `FILESYSTEM` 模式可以解决。当应用程序和worker注册，它们拥有足够的状态写入提供的目录，以至于在重启master 进程时它们能够恢复。

配置

为了开启这个恢复模式，你可以用下面的属性在 `spark-env` 中设置 `SPARK_DAEMON_JAVA_OPTS` 。

System property	Meaning
spark.deploy.recoveryMode	设置为FILESYSTEM开启单节点恢复模式（默认为none）
spark.deploy.recoveryDirectory	用来恢复状态的目录

细节

- 这个解决方案可以和监控器/管理器（如monit）相配合，或者仅仅通过重启开启手动恢复。
- 虽然文件系统的恢复似乎比没有做任何恢复要好，但对于特定的开发或实验目的，这种模式可能是次优的。特别是，通过 `stop-master.sh` 杀掉master不会清除它的恢复状态，所以，不管你何时启动一个新的master，它都将进入恢复模式。这可能使启动时间增加到1分钟。
- 虽然它不是官方支持的方式，你也可以创建一个NFS目录作为恢复目录。如果原始的master节点完全死掉，你可以在不同的节点启动master，它可以正确的恢复之前注册的所有应用程序和workers。未来的应用程序会发现这个新的master。

在YARN上執行Spark

配置

大部分是 `Spark on YARN` 模式提供的配置與其它部署模式提供的配置相同。下面這些是 `Spark on YARN` 模式提供的配置選擇。

Spark 屬性

Property Name	Default	Meaning
spark.yarn.applicationMaster.waitTries	10	ApplicationMaster等待Spark master的次數以及Spa 次數
spark.yarn.submit.file.replication	HDFS 預設的複製次數（3）	上傳到HDFS的文件的HDFS複製水準。這些文件包 任何分布式記憶體文件/檔案
spark.yarn.preserve.staging.files	false	預設為true，在作業結束時保留階段性文件（Spark 布式記憶體文件）而非刪除他們
spark.yarn.scheduler.heartbeat.interval-ms	5000	Spark application master給YARN ResourceManage （ms）
spark.yarn.max.executor.failures	numExecutors * 2,最小為3	失敗應用程式之前最大的執行失敗次數
spark.yarn.historyServer.address	(none)	Spark歷史服務器（如host.com:18080）的位置。這 （http://）。預設情況下下沒有設定值，是因為該選 Spark應用程式完成以ResourceManager UI到Spark 這個位址可從YARN ResourceManager得到
spark.yarn.dist.archives	(none)	抓取逗號分隔的檔案列表到每一個執行器的工作目錄
spark.yarn.dist.files	(none)	放置逗號分隔的文件列表到每個執行器的工作目錄
spark.yarn.executor.memoryOverhead	executorMemory * 0.07,最小384	配給每個執行器的記憶體大小（以MB為單位）。它 符串或者其他本地消耗用的記憶體。這往往隨著執行 情況下是6%-10%）
spark.yarn.driver.memoryOverhead	driverMemory * 0.07,最小384	分配給每個driver的記憶體大小（以MB為單位）。它 符串或者其他本地消耗用的記憶體。這會隨著執行器 況下是6%-10%）
spark.yarn.queue	default	應用程式傳送到的YARN 隊列的名稱
spark.yarn.jar	(none)	Spark jar文件的位置，會覆蓋預設的位置。預設情況 會用到本地安裝的Spark jar。但是Spark jar也可以是 置。這讓YARN記憶體它到節點上，而不用在每次運 配。指向HDFS中的jar包，可以這個參數為"hdfs:///s
spark.yarn.access.namenodes	(none)	你的Spark應用程式訪問的HDFS namenode列表。 如， `spark.yarn.access.namenodes=hdfs://nn1.com:80` Spark應用程式必須訪問namenode列表，Kerberos 們。Spark獲得namenode的安全通過，這樣Spark應 端的HDFS集群。
spark.yarn.containerLauncherMaxThreads	25	為了啟動執行者容器，應用程式master用到的最大線
spark.yarn.appMasterEnv.[EnvironmentVariableName]	(none)	增加通過 `EnvironmentVariableName` 指定的環境變量到 YARN上的啟動。用戶可以指定多個設定，從而設定 cluster模式下，這控制著Spark driver的環境。在yar 控制執行器啟動者的環境。

在YARN上啟動Spark

確保 `HADOOP_CONF_DIR` 或 `YARN_CONF_DIR` 指向的目錄包含Hadoop集群的（客戶端）配置文件。這些配置用於寫資料到dfs與YARN ResourceManager。

有兩種佈署模式可以用在YARN上啟動Spark應用程式。在yarn-cluster模式下，Spark driver執行在application master中，這個程序被集群中的YARN所管理，客戶端會在初始化應用程式 之后关闭。在yarn-client模式下，driver运行在客户端进程中，application master仅仅用来向YARN请求资源。

和Spark獨立模式以及Mesos模式不同，這些模式中，master的位置由"master"參數指定，而在YARN模式下，ResourceManager的位置從Hadoop配置取得。因此master參數是簡單的 `yarn-client` 和 `yarn-cluster`。

在yarn-cluster模式下啟動Spark應用程式。

```
./bin/spark-submit --class path.to.your.Class --master yarn-cluster [options]  [app options]
```

例如：

```
$ ./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master yarn-cluster \
    --num-executors 3 \
    --driver-memory 4g \
    --executor-memory 2g \
    --executor-cores 1 \
    --queue thequeue \
    lib/spark-examples*.jar \
    10
```

以上啟動了一個YARN客戶端程式用來啟動預設的 Application Master，然後SparkPi會作為Application Master的子線程運作。客戶端會定期訪問Application Master作狀態更新並且江更新顯示在控制台上。一旦應用程式執行完畢，客戶端就會退出。

在yarn-client模式下啟動Spark應用程式，執行下面的shell脚本

```
$ ./bin/spark-shell --master yarn-client
```

增加其他的**jar**

在yarn-cluster模式下，driver執行在不同機器上，所以離開了保存在本地客戶端的文件，`SparkContext.addJar` 將不會作事。為了讓 `SparkContext.addJar` 用到保存在客戶端的文件， 可以在啟動命令列上加 `--jars` 選項。

```
$ ./bin/spark-submit --class my.main.Class \
    --master yarn-cluster \
    --jars my-other-jar.jar,my-other-other-jar.jar
    my-main-jar.jar
    app_arg1 app_arg2
```

注意事項

- 在Hadoop 2.2之前，YARN不支持容器核的資源請求。因此，當執行早期版本時，通過命令行參數指定CPU的數量無法傳遞到YARN。在調度決策中，CPU需求是否成功兌現取決於用哪個調度器以及如何配置他
- Spark executors使用的本地目錄將會YARN配置（yarn.nodemanager.local-dirs）的本地目錄。如果用戶指定了 `spark.local.dir`，他將被忽略。
- `--files` 和 `--archives` 選項支援指定帶有 # 符號的文件名稱。例如，你能夠指定 `--files localtest.txt#appSees.txt`，它

上傳你在本地命名為 `localtest.txt` 的文件到HDFS，但是將會連結道明稱為 `appSees.txt` 。當你的應用程式執行在YARN上時，你應該用 `appSees.txt` 去引用該文件。

- 如果你在yarn-cluster模式下執行 `SparkContext.addJar` ，並且用到了本地文件， `--jars` 選項允許 `SparkContext.addJar` 函數能夠工作。如果你正在使用 HDFS, HTTP, HTTPS或FTP，你不需要用到該選項

Spark配置

Spark提供三个位置用来配置系统：

- Spark properties控制大部分的应用程序参数，可以用SparkConf对象或者java系统属性设置
- Environment variables可以通过每个节点的 `conf/spark-env.sh` 脚本设置每台机器的设置。例如IP地址
- Logging可以通过log4j.properties配置

Spark属性

Spark属性控制大部分的应用程序设置，并且为每个应用程序分别配置它。这些属性可以直接在SparkConf上配置，然后传递给 `SparkContext`。 `SparkConf` 允许你配置一些通用的属性（如master URL、应用程序明）以及通过 `set()` 方法设置的任意键值对。例如，我们可以用如下方式创建一个拥有两个线程的应用程序。注意，我们用 `local[2]` 运行，这意味着两个线程-表示最小的 并行度，它可以帮助我们检测当在分布式环境下运行的时才出现的错误。

```
val conf = new SparkConf()
             .setMaster("local[2]")
             .setAppName("CountingSheep")
             .set("spark.executor.memory", "1g")
val sc = new SparkContext(conf)
```

注意，我们在本地模式中拥有超过1个线程。和Spark Streaming的情况一样，我们可能需要一个线程防止任何形式的饥饿问题。

动态加载Spark属性

在一些情况下，你可能想在 `SparkConf` 中避免硬编码确定的配置。例如，你想用不同的master或者不同的内存数运行相同的应用程序。Spark允许你简单地创建一个空conf。

```
val sc = new SparkContext(new SparkConf())
```

然后你在运行时提供值。

```
./bin/spark-submit --name "My app" --master local[4] --conf spark.shuffle.spill=false
  --conf "spark.executor.extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps" myApp.jar
```

Spark shell和 `spark-submit` 工具支持两种方式动态加载配置。第一种方式是命令行选项，例如 `--master`，如上面shell显示的那样。 `spark-submit` 可以接受任何Spark属性，用 `--conf` 标记表示。但是那些参与Spark应用程序启动的属性要用特定的标记表示。运行 `./bin/spark-submit --help` 将会显示选项的整个列表。

`bin/spark-submit` 也会从 `conf/spark-defaults.conf` 中读取配置选项，这个配置文件中，每一行都包含一对以空格分开的键和值。例如：

```
spark.master            spark://5.6.7.8:7077
spark.executor.memory   512m
spark.eventLog.enabled  true
spark.serializer        org.apache.spark.serializer.KryoSerializer
```

任何标签（flags）指定的值或者在配置文件中的值将会传递给应用程序，并且通过 `SparkConf` 合并这些值。在 `SparkConf` 上设

置的属性具有最高的优先级，其次是传递给 `spark-submit` 或者 `spark-shell` 的属性值，最后是 `spark-defaults.conf` 文件中的属性值。

查看Spark属性

在 `http://<driver>:4040` 上的应用程序web UI在“Environment”标签中列出了所有的Spark属性。这对你确保设置的属性的正确性是很有用的。注意，只有通过spark-defaults.conf, SparkConf以及 命令行直接指定的值才会显示。对于其它的配置属性，你可以认为程序用到了默认的值。

可用的属性

控制内部设置的大部分属性都有合理的默认值，一些最通用的选项设置如下：

应用程序属性

Property Name	Default	Meaning
spark.app.name	(none)	你的应用程序的名字。这将在UI和日志数中出现
spark.master	(none)	集群管理器连接的地方
spark.executor.memory	512m	每个executor进程使用的内存数。和JVM存串拥有相同的格式（如512m,2g）
spark.driver.memory	512m	driver进程使用的内存数
spark.driver.maxResultSize	1g	每个Spark action(如collect)所有分区的序化结果的总大小限制。设置的值应该不小1m，0代表没有限制。如果总大小超过此限制，工作将会终止。大的限制值可能导driver出现内存溢出错误（依赖于spark.driver.memory和JVM中对象的内存耗）。设置合理的限制，可以避免出现内溢出错误。
spark.serializer	org.apache.spark.serializer.JavaSerializer	序列化对象使用的类。默认的java序列化可以序列化任何可序列化的java对象但是很慢。所有我们建议用org.apache.spark.serializer.KryoSeria
spark.kryo.classesToRegister	(none)	如果你用Kryo序列化，给定的用逗号分隔自定义类名列表表示要注册的类
spark.kryo.registrator	(none)	如果你用Kryo序列化，设置这个类去注册的自定义类。如果你需要用自定义的方式册你的类，那么这个属性是有用的。否则 `spark.kryo.classesToRegister` 会更简它应该设置一个继承自KryoRegistrator的
spark.local.dir	/tmp	Spark中暂存空间的使用目录。在Spark1以及更高的版本中，这个属性被SPARK_LOCAL_DIRS(Standalone, Me和LOCAL_DIRS(YARN)环境变量覆盖。
spark.logConf	false	当SparkContext启动时，将有效的SparkConf记录为INFO。

运行环境

Property Name	Default	Meaning
		传递给executors的JVM选项字符串。例如GC设置或者其它日志设置。注意，在这个选项中设置Spark属性或者堆大小是不合法的。Spark属性

		需要用SparkConf对象或者 `spark-submit` 脚本用到的 `spark-defaults.conf` 文件设置。堆内存可以通过 `spark.executor.memory` 设置
spark.executor.extraClassPath	(none)	附加到executors的classpath的额外的classpath实体。这个设置存在的主要目的是Spark与旧版本的向后兼容问题。用户一般不用设置这个选项
spark.executor.extraLibraryPath	(none)	指定启动executor的JVM时用到的库路径
spark.executor.logs.rolling.strategy	(none)	设置executor日志的滚动(rolling)策略。默认情况下没有开启。可以配置为 `time` （基于时间的滚动）和 `size` (基于大小的滚动)。对于 `time`，用 `spark.executor.logs.rolling.time.interval` 设置滚动间隔；对于 `size`，用 `spark.executor.logs.rolling.size.maxBytes` 设置最大的滚动大小
spark.executor.logs.rolling.time.interval	daily	executor日志滚动的时间间隔。默认情况下没有开启。合法的值是 `daily`，`hourly`，`minutely` 以及任意的秒。
spark.executor.logs.rolling.size.maxBytes	(none)	executor日志的最大滚动大小。默认情况下没有开启。值设置为字节
spark.executor.logs.rolling.maxRetainedFiles	(none)	设置被系统保留的最近滚动日志文件的数量。更老的日志文件将被删除。默认没有开启。
spark.files.userClassPathFirst	false	(实验性)当在Executors中加载类时，是否用户添加的jar比Spark自己的jar优先级高。这个属性可以降低Spark依赖和用户依赖的冲突。它现在还是一个实验性的特征。
spark.python.worker.memory	512m	在聚合期间，每个python worker进程使用的内存数。在聚合期间，如果内存超过了这个限制，它将会将数据塞进磁盘中
spark.python.profile	false	在Python worker中开启profiling。通过 `sc.show_profiles()` 展示分析结果。或者在driver退出前展示分析结果。可以通过 `sc.dump_profiles(path)` 将结果dump到磁盘中。如果一些分析结果已经手动展示，那么在driver退出前，它们再不会自动展示
spark.python.profile.dump	(none)	driver退出前保存分析结果的dump文件的目录。每个RDD都会分别dump一个文件。可以通过 `ptats.Stats()` 加载这些文件。如果指定了这个属性，分析结果不会自动展示
spark.python.worker.reuse	true	是否重用python worker。如果是，它将使用固定数量的Python workers，而不需要为每个任务fork()一个Python进程。如果有一个非常大的广播，这个设置将非常有用。因为，广播不需要为每个任务从JVM到Python worker传递一次
spark.executorEnv.[EnvironmentVariableName]	(none)	通过 `EnvironmentVariableName` 添加指定的环境变量到executor进程。用户可以指定多个 `EnvironmentVariableName`，设置多个环境变量
spark.mesos.executor.home	driver side SPARK_HOME	设置安装在Mesos的executor上的Spark的目录。默认情况下，executors将使用driver的Spark本地（home）目录，这个目录对它们不可见。注意，如果没有通过 `spark.executor.uri` 指定Spark的二进制包，这个设置才起作用
spark.mesos.executor.memoryOverhead	executor memory * 0.07, 最小384m	这个值是 `spark.executor.memory` 的补充。它用来计算mesos任务的总内存。另外，有一个7%的硬编码设置。最后的值将选择 `spark.mesos.executor.memoryOverhead` 或者 `spark.executor.memory` 的7%二者之间的大者

Shuffle行为(Behavior)

Property Name	Default	Meaning
spark.shuffle.consolidateFiles	false	如果设置为"true"，在shuffle期间，合并的中间文件将会被创建。创建更少的文件可以提供文件系统的shuffle的效率。这些shuffle都伴随着大量递归任务。当用ext4和dfs文件系统时，推荐设置为"true"。在ext3中，因为文件系统的限制，这个选项可能机器（大于8核）降低效率
spark.shuffle.spill	true	如果设置为"true"，通过将多出的数据写入磁盘来限制内存数。通过 spark.shuffle.memoryFraction 来指定spilling的阈值
spark.shuffle.spill.compress	true	在shuffle时，是否将spilling的数据压缩。压缩算法通过 spark.io.compression.codec 指定。
spark.shuffle.memoryFraction	0.2	如果 spark.shuffle.spill 为“true”，shuffle中聚合和合并组操作使用的java堆内存占总内存的比重。在任何时候，shuffles使用的所有内存内maps的集合大小都受这个限制的约束。超过这个限制，spilling数据将会保存到磁盘上。如果spilling太过频繁，考虑增大这个值
spark.shuffle.compress	true	是否压缩map操作的输出文件。一般情况下，这是一个好的选择。
spark.shuffle.file.buffer.kb	32	每个shuffle文件输出流内存内缓存的大小，单位是kb。这个缓存减少了创建只中间shuffle文件中磁盘搜索和系统访问的数量
spark.reducer.maxMbInFlight	48	从递归任务中同时获取的map输出数据的最大大小（mb）。因为每一个输出都需要我们创建一个缓存用来接收，这个设置代表每个任务固定的内存上限，所以除非你有更大的内存，将其设置小一点
spark.shuffle.manager	sort	它的实现用于shuffle数据。有两种可用的实现： sort 和 hash 。基于sort的shuffle有更高的内存使用率
spark.shuffle.sort.bypassMergeThreshold	200	(Advanced) In the sort-based shuffle manager, avoid merge-sorting data if there is no map-side aggregation and there are at most this many reduce partitions
spark.shuffle.blockTransferService	netty	实现用来在executor直接传递shuffle和缓存块。有两种可用的实现： netty 和 nio 。基于netty的块传递在具有相同的效率情况下更简单

Spark UI

Property Name	Default	Meaning
spark.ui.port	4040	你的应用程序dashboard的端口。显示内存和工作量数据
spark.ui.retainedStages	1000	在垃圾回收之前，Spark UI和状态API记住的stage数
spark.ui.retainedJobs	1000	在垃圾回收之前，Spark UI和状态API记住的job数
spark.ui.killEnabled	true	运行在web UI中杀死stage和相应的job
spark.eventLog.enabled	false	是否记录Spark的事件日志。这在应用程序完成后，重新构造web UI是有用的
spark.eventLog.compress	false	是否压缩事件日志。需要 spark.eventLog.enabled 为true
spark.eventLog.dir	file:///tmp/spark-events	Spark事件日志记录的基本目录。在这个基本目录下，Spark为每个应用程序创建一个子目录。各个应用程序记录日志到直到的目录。用户可能想设置这为统一的地点，像HDFS一样，所以历史文件可以通过历史服务器读取

压缩和序列化

Property Name	Default	
spark.broadcast.compress	true	在发送广播变量之前是否压缩
spark.rdd.compress	true	是否压缩序列化的RDD分区。
spark.io.compression.codec	snappy	压缩诸如RDD分区、广播变量 以用完整的类名来制 定。 `org.apache.spark.io.LZ4C`
spark.io.compression.snappy.block.size	32768	Snappy压缩中用到的块大小。
spark.io.compression.lz4.block.size	32768	LZ4压缩中用到的块大小。降
spark.closure.serializer	org.apache.spark.serializer.JavaSerializer	闭包用到的序列化类。目前只
spark.serializer.objectStreamReset	100	当用 `org.apache.spark.seriali` 收停止。通过请求'reset',你从 况下，每一百个对象reset一次
spark.kryo.referenceTracking	true	当用Kryo序列化时，跟踪是否 效率是有用的。如果你知道不
spark.kryo.registrationRequired	false	是否需要注册为Kyro可用。如 每个对象和其非注册类名。写
spark.kryoserializer.buffer.mb	0.064	Kyro序列化缓存的大小。这样 那么大。
spark.kryoserializer.buffer.max.mb	64	Kryo序列化缓存允许的最大值

Networking

Property Name	Default	Meaning
spark.driver.host	(local hostname)	driver监听的主机名或者IP地址。这用于和executors以及独立的 master通信
spark.driver.port	(random)	driver监听的接口。这用于和executors以及独立的master通信
spark.fileserver.port	(random)	driver的文件服务器监听的端口
spark.broadcast.port	(random)	driver的HTTP广播服务器监听的端口
spark.replClassServer.port	(random)	driver的HTTP类服务器监听的端口
spark.blockManager.port	(random)	块管理器监听的端口。这些同时存在于driver和executors
spark.executor.port	(random)	executor监听的端口。用于与driver通信
spark.port.maxRetries	16	当绑定到一个端口，在放弃前重试的最大次数
spark.akka.frameSize	10	在"control plane"通信中允许的最大消息大小。如果你的任务需要发 送大的结果到driver中，调大这个值
spark.akka.threads	4	通信的actor线程数。当driver有很多CPU核时，调大它是有用的
spark.akka.timeout	100	Spark节点之间的通信超时。单位是s
spark.akka.heartbeat.pauses	6000	This is set to a larger value to disable failure detector that comes inbuilt akka. It can be enabled again, if you plan to use this feature (Not recommended). Acceptable heart beat pause in seconds for akka. This can be used to control sensitivity to gc pauses. Tune this in combination of `spark.akka.heartbeat.interval` and `spark.akka.failure-detector.threshold` if you need to.
spark.akka.failure-detector.threshold	300.0	This is set to a larger value to disable failure detector that comes inbuilt akka. It can be enabled again, if you plan to use this feature (Not recommended). This maps to akka's `akka.remote.transport-failure-detector.threshold` . Tune this in combination of `spark.akka.heartbeat.pauses` and `spark.akka.heartbeat.interval` if

		you need to.
spark.akka.heartbeat.interval	1000	This is set to a larger value to disable failure detector that comes inbuilt akka. It can be enabled again, if you plan to use this feature (Not recommended). A larger interval value in seconds reduces network overhead and a smaller value (~ 1 s) might be more informative for akka's failure detector. Tune this in combination of `spark.akka.heartbeat.pauses` and `spark.akka.failure-detector.threshold` if you need to. Only positive use case for using failure detector can be, a sensistive failure detector can help evict rogue executors really quick. However this is usually not the case as gc pauses and network lags are expected in a real Spark cluster. Apart from that enabling this leads to a lot of exchanges of heart beats between nodes leading to flooding the network with those.

Security

Property Name	Default	Meaning
spark.authenticate	false	是否Spark验证其内部连接。如果不是运行在YARN上，请看 `spark.authenticate.secret`
spark.authenticate.secret	None	设置Spark两个组件之间的密匙验证。如果不是运行在YARN上，但是需要验证，这个选项必须设置
spark.core.connection.auth.wait.timeout	30	连接时等待验证的实际。单位为秒
spark.core.connection.ack.wait.timeout	60	连接等待回答的时间。单位为秒。为了避免不希望的超时，你可以设置更大的值
spark.ui.filters	None	应用到Spark web UI的用于过滤类名的逗号分隔的列表。过滤器必须是标准的javax servlet Filter。通过设置java系统属性也可以指定每个过滤器的参数。 `spark.<class name of filter>.params='param1=value1,param2=value2'` 。例如 `-Dspark.ui.filters=com.test.filter1` 、 `-Dspark.com.test.filter1.params='param1=foo,param2=testing'`
spark.acls.enable	false	是否开启Spark acls。如果开启了，它检查用户是否有权限去查看或修改job。 Note this requires the user to be known, so if the user comes across as null no checks are done。UI利用使用过滤器验证和设置用户
spark.ui.view.acls	empty	逗号分隔的用户列表，列表中的用户有查看(view)Spark web UI的权限。默认情况下，只有启动Spark job的用户有查看权限
spark.modify.acls	empty	逗号分隔的用户列表，列表中的用户有修改Spark job的权限。默认情况下，只有启动Spark job的用户有修改权限
spark.admin.acls	empty	逗号分隔的用户或者管理员列表，列表中的用户或管理员有查看和修改所有Spark job的权限。如果你运行在一个共享集群，有一组管理员或开发者帮助debug，这个选项有用

Spark Streaming

Property Name	Default	Meaning
spark.streaming.blockInterval	200	在这个时间间隔（ms）内，通过Spark Streaming receivers接收的数据在保存到Spark之前，chunk为数据块。推荐的最小值为50ms
spark.streaming.receiver.maxRate	infinite	每秒钟每个receiver将接收的数据的最大记录数。有效的情况下，每个流将消耗至少这个数目的记录。设置这个配置为0或者-1将会不作限制
spark.streaming.receiver.writeAheadLogs.enable	false	Enable write ahead logs for receivers. All the input data received through receivers will be saved to write ahead logs that will allow it to be recovered after driver failures

spark.streaming.unpersist	true	强制通过Spark Streaming生成并持久化的RDD自动从Spark内存中非持久化。通过Spark Streaming接收的原始输入数据也将清除。设置这个属性为false允许流应用程序访问原始数据和持久化RDD，因为它们没有被自动清除。但是它会造成更高的内存花费

环境变量

通过环境变量配置确定的Spark设置。环境变量从Spark安装目录下的 `conf/spark-env.sh` 脚本读取（或者windows的 `conf/spark-env.cmd` ）。在独立的或者Mesos模式下，这个文件可以给机器 确定的信息，如主机名。当运行本地应用程序或者提交脚本时，它也起作用。

注意，当Spark安装时， `conf/spark-env.sh` 默认是不存在的。你可以复制 `conf/spark-env.sh.template` 创建它。

可以在 `spark-env.sh` 中设置如下变量：

Environment Variable	Meaning
JAVA_HOME	java安装的路径
PYSPARK_PYTHON	PySpark用到的Python二进制执行文件路径
SPARK_LOCAL_IP	机器绑定的IP地址
SPARK_PUBLIC_DNS	你Spark应用程序通知给其他机器的主机名

除了以上这些，Spark standalone cluster scripts也可以设置一些选项。例如 每台机器使用的核数以及最大内存。

因为 `spark-env.sh` 是shell脚本，其中的一些可以以编程方式设置。例如，你可以通过特定的网络接口计算 `SPARK_LOCAL_IP` 。

配置Logging

Spark用log4j logging。你可以通过在conf目录下添加 `log4j.properties` 文件来配置。一种方法是复制 `log4j.properties.template` 文件。

RDD 永續儲存

Spark 有一個最重要的功能是在記憶體中永續儲存 (或 緩存) 一個資料集。