Spark 2.2.x reference doc Documentation

发布 *2.2.1*

jackiehff

2018 年 01 月 23 日

Contents

CHAPTER 1

编程指南

1.1 快速入门

本教程是对使用 Spark 的一个简单介绍。首先我们会通过 Spark 的交互式 shell 简单介绍一下 (Python 或 Scala) API，然后展示如何使用 Java、Scala 以及 Python 编写一个 Spark 应用程序。

为了方便参照该指南进行学习，请先到 Spark 网站 下载一个 Spark 发布包。由于我们暂时还不会用到 HDFS，所以你可以下载对应任意 Hadoop 版本的 Spark 发布包。

注意： Spark 2.0 版本之前, Spark 的核心编程接口是弹性分布式数据集(RDD)。Spark 2.0 版本之后, RDD 被 Dataset 所取代, Dataset 跟 RDD 一样也是强类型的, 但是底层做了更多的优化。Spark 目前仍然支持 RDD 接口, 你可以在 *RDD 编程指南* 页面获得更完整的参考，但是我们强烈建议你转而使用比 RDD 有着更好性能的 Dataset。想了解关于 Dataset 的更多信息请参考 *Spark SQL, DataFrame 和 Dataset 编程指南*。

1.1.1 使用 Spark Shell 进行交互式分析

基础知识

Spark Shell 提供了一种简单的方式来学习 Spark API，同时它也是一个强大的交互式数据分析工具。Spark Shell 既支持 Scala(Scala 运行在 Java 虚拟机上，所以可以很方便的引用现有的 Java 库)也支持 Python。

Scala

在 Spark 目录下运行以下命令可以启动 Spark Shell:

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

Spark 最主要的抽象概念就是一个叫做 Dataset 的分布式数据集。Dataset 可以从 Hadoop InputFormats(例如 HDFS 文件)创建或者由其他 Dataset 转换而来。下面我们利用 Spark 源码目录下 README 文件中的文本来新建一个 Dataset:

```
scala> val textFile = spark.read.textFile("README.md")
textFile: org.apache.spark.sql.Dataset[String] = [value: string]
```

你可以调用 action 算子直接从 Dataset 获取值，或者转换该 Dataset 以获取一个新的 Dataset。更多细节请参阅 API 文档 。

```
scala> textFile.count() // Number of items in this Dataset
res0: Long = 126 // May be different from yours as README.md will change over time,␣
↪similar to other outputs

scala> textFile.first() // First item in this Dataset
res1: String = # Apache Spark
```

现在我们将该 Dataset 转换成一个新的 Dataset。我们调用 filter 这个 transformation 算子返回一个只包含原始文件数据项子集的新 Dataset。

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

我们可以将 transformation 算子和 action 算子连在一起:

```
scala> textFile.filter(line => line.contains("Spark")).count() // How many lines␣
↪contain "Spark"?
res3: Long = 15
```

Python

在 Spark 目录下运行以下命令可以启动 Spark Shell:

```
./bin/pyspark
```

或者如果在你当前环境已经使用 pip 安装了 PySpark，你也可以直接使用以下命令:

```
pyspark
```

Spark 最主要的抽象概念就是一个叫做 Dataset 的分布式数据集。Dataset 可以从 Hadoop InputFormats(例如 HDFS 文件)创建或者由其他 Dataset 转换而来。由于 Python 语言的动态性, 我们不需要 Dataset 是强类型的。因此 Python 中所有的 Dataset 都是 Dataset[Row], 并且为了和 Pandas 以及 R 中的 data frame 概念保持一致, 我们称其为 DataFrame。下面我们利用 Spark 源码目录下 README 文件中的文本来新建一个 DataFrame:

```
>>> textFile = spark.read.text("README.md")
```

你可以调用 action 算子直接从 DataFrame 获取值，或者转换该 DataFrame 以获取一个新的 DataFrame。更多细节请参阅 API 文档 。

```
>>> textFile.count()  # Number of rows in this DataFrame
126

>>> textFile.first()  # First row in this DataFrame
Row(value=u'# Apache Spark')
```

现在我们将该 DataFrame 转换成一个新的 DataFrame。我们调用 filter 这个 transformation 算子返回一个只包含原始文件数据项子集的新 DataFrame。

```
>>> linesWithSpark = textFile.filter(textFile.value.contains("Spark"))
```

我们可以将 transformation 算子和 action 算子连在一起:

```
>>> textFile.filter(textFile.value.contains("Spark")).count()  # How many lines␣
↪contain "Spark"?
15
```

更多 Dataset 算子

Dataset action 和 transformation 算子可以用于更加复杂的计算。比方说我们想要找到文件中包含单词数最多的行。

Scala

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

首先，使用 map 算子将每一行映射为一个整数值，创建了一个新的 Dataset。然后在该 Dataset 上调用 reduce 算子找出最大的单词计数。map 和 reduce 算子的参数都是 cala 函数字面量(闭包)，并且可以使用任意语言特性或 Scala/Java 库。例如，我们可以很容易地调用其他地方声明的函数。为了使代码更容易理解，下面我们使用Math.max():

```
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
```

因 Hadoop 而广为流行的 MapReduce 是一种通用的数据流模式。Spark 可以很容易地实现 MapReduce 流程：

```
scala> val wordCounts = textFile.flatMap(line => line.split(" ")).
↪groupByKey(identity).count()
wordCounts: org.apache.spark.sql.Dataset[(String, Long)] = [value: string, count(1):␣
↪bigint]
```

这里我们调用 flatMap 这个 transformation 算子将一个行的 Dataset 转换成了一个单词的 Dataset, 然后组合 groupByKey 和 count 算子来计算文件中每个单词出现的次数，生成一个包含(String, Long)键值对的 Dataset。为了在 shell 中收集到单词计数, 我们可以调用 collect 算子:

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

Python

```
>>> from pyspark.sql.functions import *
>>> textFile.select(size(split(textFile.value, "\s+")).name("numWords")).agg(max(col(
↪"numWords"))).collect()
[Row(max(numWords)=15)]
```

首先，使用 map 算子将每一行映射为一个整数值并给其取别名 “numWords”, 创建了一个新的 DataFrame。然后在该 DataFrame 上调用 agg 算子找出最大的单词计数。select 和 agg 的参数都是 Column , 我们可以使用 df.colName 从 DataFrame 上获取一列，也可以引入 pyspark.sql.functions, 它提供了很多方便的函数用来从旧的 Column 构建新的 Column。

因 Hadoop 而广为流行的 MapReduce 是一种通用的数据流模式。Spark 可以很容易地实现 MapReduce 流程：

```
>>> wordCounts = textFile.select(explode(split(textFile.value, "\s+")).alias("word")).
↪groupBy("word").count()
```

这里我们在 select 函数中使用 explode 函数将一个行的 Dataset 转换成了一个单词的 Dataset, 然后组合 groupBy 和 count 算子来计算文件中每个单词出现的次数，生成一个包含 “word” 和 “count” 这 2 列的 DataFrame。为了在 shell 中收集到单词计数, 我们可以调用 collect 算子:

```
>>> wordCounts.collect()
[Row(word=u'online', count=1), Row(word=u'graphs', count=1), ...]
```

缓存

Spark 还支持把数据集拉到集群范围的内存缓存中。当数据需要反复访问时非常有用，比如查询一个小的热门数据集或者运行一个像 PageRank 这样的迭代算法。作为一个简单的示例，我们把 linesWithSpark 这个数据集缓存起来。

Scala

```
scala> linesWithSpark.cache()
res7: linesWithSpark.type = [value: string]

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

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

用 Spark 浏览和缓存一个 100 行左右的文本文件看起来确实有点傻。但有趣的部分是这些相同的函数可以用于非常大的数据集，即使这些数据集分布在数十或数百个节点上。如 *RDD* 编程指南 中描述的那样, 你也可以通过 bin/spark-shell 连接到一个集群，交互式地执行上面那些操作。

Python

```
>>> linesWithSpark.cache()

>>> linesWithSpark.count()
15

>>> linesWithSpark.count()
15
```

用 Spark 浏览和缓存一个 100 行左右的文本文件看起来确实有点傻。但有趣的部分是这些相同的函数可以用于非常大的数据集，即使这些数据集分布在数十或数百个节点上。如 *RDD* 编程指南 中描述的那样, 你也可以通过 bin/pyspark 连接到一个集群，交互式地执行上面那些操作。

1.1.2 自包含的(self-contained)应用程序

假设我们想使用 Spark API 编写一个自包含(self-contained)的 Spark 应用程序。下面我们将快速过一下一个简单的应用程序，分别使用 Scala(sbt编译)，Java(maven编译)和 Python(pip) 编写。

Scala

首先创建一个非常简单的 Spark 应用程序 – 简单到连名字都叫 SimpleApp.scala:

```scala
/* SimpleApp.scala */
import org.apache.spark.sql.SparkSession

object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "YOUR_SPARK_HOME/README.md" // Should be some file on your system
    val spark = SparkSession.builder.appName("Simple Application").getOrCreate()
    val logData = spark.read.textFile(logFile).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println(s"Lines with a: $numAs, Lines with b: $numBs")
    spark.stop()
  }
}
```

注意：应用程序需要定义一个 main 方法，而不是继承 scala.App。scala.App 的子类可能不能正常工作。

这个程序只是统计 Spark README 文件中包含'a'和包含'b'的行数。注意，你需要把 YOUR_SPARK_HOME 替换成 Spark 的安装目录。与之前使用 Spark Shell 的示例不同，Spark Shell 会初始化自己的 SparkSession 对象, 而我们需要初始化 SparkSession 对象作为程序的一部分。

我们调用 SparkSession.builder 来构造一个 [[SparkSession]] 对象, 然后设置应用程序名称, 最后调用 getOrCreate 方法获取 [[SparkSession]] 实例。

我们的应用程序依赖于 Spark API，所以我们需要包含一个 sbt 配置文件，build.sbt，用于配置 Spark 依赖项。这个文件同时也添加了 Spark 本身的依赖库：

```
name := "Simple Project"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.2.1"
```

为了让 sbt 能够正常工作，我们需要根据一个标准规范的 Scala 项目目录结构来放置 SimpleApp.scala 和 build.sbt 文件。一切准备就绪后，我们就可以创建一个包含应用程序代码的 JAR 包，然后使用 spark-submit 脚本运行我们的程序。

```
# Your directory layout should look like this
$ find .
.
./simple.sbt
./src
./src/main
./src/main/scala
./src/main/scala/SimpleApp.scala

# Package a jar containing your application
$ sbt package
...
[info] Packaging {..}/{..}/target/scala-2.11/simple-project_2.11-1.0.jar

# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
  --class "SimpleApp" \
  --master local[4] \
  target/scala-2.11/simple-project_2.11-1.0.jar
```

```
...
Lines with a: 46, Lines with b: 23
```

Java

下面这个示例程序将使用 Maven 来编译一个应用程序 JAR, 但是适用任何类似的构建系统。

我们创建一个非常简单的 Spark 应用程序, SimpleApp.java:

```
/* SimpleApp.java */
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.Dataset;

public class SimpleApp {
  public static void main(String[] args) {
    String logFile = "YOUR_SPARK_HOME/README.md"; // Should be some file on your␣
↪system
    SparkSession spark = SparkSession.builder().appName("Simple Application").
↪getOrCreate();
    Dataset<String> logData = spark.read().textFile(logFile).cache();

    long numAs = logData.filter(s -> s.contains("a")).count();
    long numBs = logData.filter(s -> s.contains("b")).count();

    System.out.println("Lines with a: " + numAs + ", lines with b: " + numBs);

    spark.stop();
  }
}
```

这个程序只是统计 Spark README 文件中包含'a'和包含'b'的行数。注意，你需要把 YOUR_SPARK_HOME 替换成 Spark 的安装目录。与之前使用 Spark Shell 的示例不同，Spark Shell 会初始化自己的 SparkSession 对象, 而我们需要初始化 SparkSession 对象作为程序的一部分。

为了构建程序, 我们还需要编写一个 Maven pom.xml 文件将 Spark 列为依赖项。注意，Spark 构件都附加了 Scala 版本号。

```
<project>
  <groupId>edu.berkeley</groupId>
  <artifactId>simple-project</artifactId>
  <modelVersion>4.0.0</modelVersion>
  <name>Simple Project</name>
  <packaging>jar</packaging>
  <version>1.0</version>
  <dependencies>
    <dependency> <!-- Spark dependency -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.2.1</version>
    </dependency>
  </dependencies>
</project>
```

接着，我们根据标准规范的 Maven 项目目录结构放置这些文件:

```
$ find .
./pom.xml
./src
./src/main
```

```
./src/main/java
./src/main/java/SimpleApp.java
```

现在我们可以使用 Maven 打包应用程序并使用 ./bin/spark-submit 命令执行它。

```
# Package a JAR containing your application
$ mvn package
...
[INFO] Building jar: {..}/{..}/target/simple-project-1.0.jar

# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
  --class "SimpleApp" \
  --master local[4] \
  target/simple-project-1.0.jar
...
Lines with a: 46, Lines with b: 23
```

Python

现在我们将展示如何使用 Python API (PySpark) 来编写一个 Spark 应用程序。

如果你在构建一个打包好的 PySpark 应用程序或者库, 你可以像下面这样将其添加到 setup.py 文件中:

```
install_requires=[
    'pyspark=={site.SPARK_VERSION}'
]
```

我们将创建一个简单的 Spark 应用程序 SimpleApp.py 作为示例程序:

```
"""SimpleApp.py"""
from pyspark.sql import SparkSession

logFile = "YOUR_SPARK_HOME/README.md"  # Should be some file on your system
spark = SparkSession.builder().appName(appName).master(master).getOrCreate()
logData = spark.read.text(logFile).cache()

numAs = logData.filter(logData.value.contains('a')).count()
numBs = logData.filter(logData.value.contains('b')).count()

print("Lines with a: %i, lines with b: %i" % (numAs, numBs))

spark.stop()
```

这个程序只是统计 Spark README 文件中包含'a'和包含'b'的行数。注意，你需要把 YOUR_SPARK_HOME 替换成 Spark 的安装目录。在 Scala 和 Java 编写的示例程序中, 我们使用 SparkSession 来创建 Dataset。对于使用自定义类或第三方库的应用程序，我们还可以将代码依赖打包成 .zip 文件, 然后通过 spark-submit 脚本提供的 –py-files 参数添加到 spark-submit (更多细节参见 spark-submit –help)。SimpleApp 已经足够简单，我们不需要指定任何代码依赖。

我们可以使用 bin/spark-submit 脚本运行这个应用程序:

```
# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
  --master local[4] \
  SimpleApp.py
...
Lines with a: 46, Lines with b: 23
```

如果你已经使用 pip 安装了 PySpark (例如 pip install pyspark), 你可以使用普通的 Python 解释器运行应用程序，或者根据你自己的喜好使用 Spark 提供的 spark-submit 脚本。

```
# Use python to run your application
$ python SimpleApp.py
...
Lines with a: 46, Lines with b: 23
```

1.1.3 下一步

恭喜您成功运行您的第一个 Spark 应用程序！

- 如果想深入了解 Spark API, 可以从 *RDD* 编程指南 和 *Spark SQL, DataFrame* 和 *Dataset* 编程指南，或者在 “Programming Guides” 菜单下查找其它组件。
- 如果想了解如何在集群上运行 Spark 应用程序，请前往：集群模式概述。
- 最后，Spark examples 目录下包含了多个编程语言(Scala, Java, Python, R)版本的示例程序，你可以像下面这样运行它们：

```
# 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

# For R examples, use spark-submit directly:
./bin/spark-submit examples/src/main/r/dataframe.R
```

1.2 RDD 编程指南

1.2.1 概述

总体上来说，每个 Spark 应用程序都包含一个驱动器（driver）程序，驱动器程序运行用户的 main 函数，并在集群上执行各种并行操作。Spark 最重要的一个抽象概念就是弹性分布式数据集（resilient distributed dataset – RDD）, RDD是一个可分区的元素集合，这些元素分布在集群的各个节点上，并且可以在这些元素上执行并行操作。RDD通常是通过HDFS（或者Hadoop支持的其它文件系统）上的文件，或者驱动器中的Scala集合对象来创建或转换得到；其次，用户也可以请求Spark将RDD持久化到内存里，以便在不同的并行操作里复用之；最后，RDD具备容错性，可以从节点失败中自动恢复。

Spark 第二个重要抽象概念是共享变量，共享变量是一种可以在并行操作之间共享使用的变量。默认情况下，当Spark把一系列任务调度到不同节点上运行时，Spark会同时把每个变量的副本和任务代码一起发送给各个节点。但有时候，我们需要在任务之间，或者任务和驱动器之间共享一些变量。Spark 支持两种类型的共享变量：广播变量 和 累加器，广播变量可以用于在各个节点上缓存数据，而累加器则是用来执行跨节点的 “累加” 操作，例如：计数和求和。

本文将会使用 Spark 所支持的所有语言来展示 Spark 的这些特性。如果你能启动 Spark 的交互式shell动手实验一下，效果会更好（对于 Scala shell请使用bin/spark-shell，而对于python，请使用bin/pyspark）。

1.2.2 链接 Spark

Scala

Spark 2.2.1 默认使用 Scala 2.11 版本进行构建和分发的。(Spark 也可以使用其它版本的 Scala 进行构建)如果想用 Scala 写应用程序，你需要使用兼容的 Scala 版本(如：2.11.X)

要编写 Spark 应用程序，你需要添加 Spark 的 Maven 依赖。Spark 依赖可以通过以下 Maven 坐标从 Maven 中央仓库中获得：

```
groupId = org.apache.spark
artifactId = spark-core_2.11
version = 2.2.1
```

另外，如果你想要访问 HDFS 集群，那么需要添加对应 HDFS 版本的 hadoop-client 依赖。

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

最后，你需要在程序中添加下面几行来引入一些 Spark 类：

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

(在 Spark 1.3.0 版本之前，你需要显示地 import org.apache.spark.SparkContext._ 来启用必要的隐式转换)

Java

Spark 2.2.1 对 Lambda 表达式的支持可以让我们很简洁地编写函数，否则的话你可以使用 org.apache.spark.api.java.function 包中的类.

> **注意：** Spark 2.2.0 版本中已经移除对 Java 7 的支持。

要使用 Java 来编写 Spark 应用程序，你需要添加 Spark 的 Maven 依赖。Spark 依赖可以通过以下 Maven 坐标从 Maven 中央仓库中获得:

```
groupId = org.apache.spark
artifactId = spark-core_2.11
version = 2.2.1
```

另外，如果你想要访问 HDFS 集群，那么需要添加对应 HDFS 版本的 hadoop-client 依赖。

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

最后，你需要在程序中添加下面几行来引入一些 Spark 类：

```
import org.apache.spark.api.java.JavaSparkContext
import org.apache.spark.api.java.JavaRDD
import org.apache.spark.SparkConf
```

Python

Spark 2.2.1 适用于 Python 2.7 及以上版本 或 Python 3.4 及以上版本。它可以使用标准的 CPython 解释器, 因此我们可以使用像 NumPy 这样的 C 语言库。它也适用 PyPy 2.3 及以上版本。

Spark 2.2.0 版本中移除了对 Python 2.6 的支持。

使用 Python 编写的 Spark 应用程序既可以使用在运行时包含 Spark 的 bin/spark-submit 脚本运行, 也可以像下面这样通过在 setup.py 文件中包含它:

```
install_requires=[
    'pyspark=={site.SPARK_VERSION}'
]
```

To run Spark applications in Python without pip installing PySpark, use the bin/spark-submit script located in the Spark directory. This script will load Spark's Java/Scala libraries and allow you to submit applications to a cluster. You can also use bin/pyspark to launch an interactive Python shell.

如果你想要访问 HDFS 数据, you need to use a build of PySpark linking to your version of HDFS. Prebuilt packages are also available on the Spark homepage for common HDFS versions.

最后, 你需要添加下面这行来在程序中引入一些 Spark 类:

```
from pyspark import SparkContext, SparkConf
```

PySpark requires the same minor version of Python in both driver and workers. 它使用 PATH 中默认的 Python 版本, 你也可以通过 PYSPARK_PYTHON 指定你想要使用的 Python 版本, 例如:

```
$ PYSPARK_PYTHON=python3.4 bin/pyspark
$ PYSPARK_PYTHON=/opt/pypy-2.5/bin/pypy bin/spark-submit examples/src/main/python/pi.
↪py
```

1.2.3 初始化 Spark

Scala

Spark 程序需要做的第一件事就是创建一个 SparkContext 对象，SparkContext 对象决定了 Spark 如何访问集群。而要新建一个 SparkContext 对象，你还得需要构造一个 SparkConf 对象，SparkConf对象包含了你的应用程序的配置信息。

每个JVM进程中，只能有一个活跃（active）的 SparkContext 对象。如果你非要再新建一个，那首先必须将之前那个活跃的 SparkContext 对象stop()掉。

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

Java

Spark 程序需要做的第一件事就是创建一个 JavaSparkContext 对象, which tells Spark how to access a cluster. To create a SparkContext you first need to build a SparkConf object that contains information about your application.

```
SparkConf conf = new SparkConf().setAppName(appName).setMaster(master);
JavaSparkContext sc = new JavaSparkContext(conf);
```

Python

Spark 程序需要做的第一件事就是创建一个 SparkContext 对象, which tells Spark how to access a cluster. To create a SparkContext you first need to build a SparkConf object that contains information about your application.

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

appName 参数值是你的应用展示在集群UI上的应用名称。master参数值是Spark, Mesos or YARN cluster URL 或者特殊的"local"（本地模式）。实际上，一般不应该将master参数值硬编码到代码中，而是应该用spark-submit脚本的参数来设置。然而，如果是本地测试或单元测试中，你可以直接在代码里给master参数写死一个"local"值。

使用 Shell

Scala

在 Spark Shell 中，默认已经为你新建了一个 SparkContext 对象，变量名为sc。所以 spark-shell 里不能自建SparkContext对象。你可以通过–master参数设置要连接到哪个集群，而且可以给–jars参数传一个逗号分隔的jar包列表，以便将这些jar包加到classpath中。你还可以通过–packages设置逗号分隔的maven工件列表，以便增加额外的依赖项。同样，还可以通过–repositories参数增加maven repository地址。下面是一个示例，在本地4个CPU core上运行的实例：

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

或者，将 code.jar 添加到 classpath 下：

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

通过 maven标识添加依赖：

```
$ ./bin/spark-shell --master local[4] --packages "org.example:example:0.1"
```

spark-shell –help 可以查看完整的选项列表。实际上，spark-shell 是在后台调用 spark-submit 来实现其功能的（spark-submit script.）

Python

In the PySpark shell, a special interpreter-aware SparkContext is already created for you, in the variable called sc. Making your own SparkContext will not work. You can set which master the context connects to using the –master argument, and you can add Python .zip, .egg or .py files to the runtime path by passing a comma-separated list to –py-files. You can also add dependencies (e.g. Spark Packages) to your shell session by supplying a comma-separated list of Maven coordinates to the –packages argument. Any additional repositories where dependencies might exist (e.g. Sonatype) can be passed to the –repositories argument. Any Python dependencies a Spark package has (listed in the requirements.txt of that package) must be manually installed using pip when necessary. For example, to run bin/pyspark on exactly four cores, use:

```
$ ./bin/pyspark --master local[4]
```

Or, to also add code.py to the search path (in order to later be able to import code), use:

```
$ ./bin/pyspark --master local[4] --py-files code.py
```

For a complete list of options, run pyspark –help. Behind the scenes, pyspark invokes the more general spark-submit script.

It is also possible to launch the PySpark shell in IPython, the enhanced Python interpreter. PySpark works with IPython 1.0.0 and later. To use IPython, set the PYSPARK_DRIVER_PYTHON variable to ipython when running bin/pyspark:

```
$ PYSPARK_DRIVER_PYTHON=ipython ./bin/pyspark
```

To use the Jupyter notebook (previously known as the IPython notebook),

```
$ PYSPARK_DRIVER_PYTHON=jupyter PYSPARK_DRIVER_PYTHON_OPTS=notebook ./bin/pyspark
```

You can customize the ipython or jupyter commands by setting PYSPARK_DRIVER_PYTHON_OPTS.

After the Jupyter Notebook server is launched, you can create a new “Python 2” notebook from the “Files” tab. Inside the notebook, you can input the command %pylab inline as part of your notebook before you start to try Spark from the Jupyter notebook.

1.2.4 弹性分布式数据集(RDD)

Spark的核心概念是弹性分布式数据集(RDD)，RDD是一个可容错、可并行操作的分布式元素集合。总体上有两种方法可以创建 RDD 对象：由驱动程序中的集合对象通过并行化操作创建，或者从外部存储系统中数据集加载（如：共享文件系统、HDFS、HBase或者其他Hadoop支持的数据源）。

并行集合

Scala

并行集合是以一个已有的集合对象（例如：Scala Seq）为参数，调用 SparkContext.parallelize() 方法创建得到的 RDD。集合对象中所有的元素都将被复制到一个可并行操作的分布式数据集中。例如，以下代码将一个1到5组成的数组并行化成一个RDD：

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

一旦创建成功，该分布式数据集（上例中的distData）就可以执行一些并行操作。如，distData.reduce((a, b) => a + b)，这段代码会将集合中所有元素加和。后面我们还会继续讨论分布式数据集上的各种操作。

Java

Parallelized collections are created by calling JavaSparkContext's parallelize method on an existing Collection in your driver program. The elements of the collection are copied to form a distributed dataset that can be operated on in parallel. For example, here is how to create a parallelized collection holding the numbers 1 to 5:

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

Once created, the distributed dataset (distData) can be operated on in parallel. For example, we might call distData.reduce((a, b) -> a + b) to add up the elements of the list. We describe operations on distributed datasets later on.

Python

Parallelized collections are created by calling SparkContext's parallelize method on an existing iterable or collection in your driver program. The elements of the collection are copied to form a distributed dataset that can be operated on in parallel. For example, here is how to create a parallelized collection holding the numbers 1 to 5:

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

Once created, the distributed dataset (distData) can be operated on in parallel. For example, we can call distData.reduce(lambda a, b: a + b) to add up the elements of the list. We describe operations on distributed datasets later on.

并行集合的一个重要参数是分区（partition），即这个分布式数据集可以分割为多少片。Spark中每个任务（task）都是基于分区的，每个分区一个对应的任务（task）。典型场景下，一般每个CPU对应2~4个分区。并且一般而言，Spark会基于集群的情况，自动设置这个分区数。当然，你还是可以手动控制这个分区数，只需给parallelize方法再传一个参数即可（如：sc.parallelize(data, 10)）。注意：Spark代码里有些地方仍然使用分片（slice）这个术语，这只不过是分区的一个别名，主要为了保持向后兼容。

外部数据集

Spark 可以通过Hadoop所支持的任何数据源来创建分布式数据集，包括：本地文件系统、HDFS、Cassandra、HBase、Amazon S3 等。Spark 支持的文件格式包括：文本文件（text files）、SequenceFiles，以及其他 Hadoop 支持的输入格式（InputFormat）。

文本文件创建RDD可以用 SparkContext.textFile 方法。这个方法输入参数是一个文件的URI（本地路径，或者 hdfs://，s3n:// 等），其输出RDD是一个文本行集合。以下是一个简单示例：

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

创建后，distFile 就可以执行数据集的一些操作。比如，我们可以把所有文本行的长度加和：distFile.map(s => s.length).reduce((a, b) => a + b)

以下是一些 Spark 读取文件的要点：

- 如果是本地文件系统，那么这个文件必须在所有的 worker 节点上能够以相同的路径访问到。所以要么把文件复制到所有worker节点上同一路径下，要么挂载一个共享文件系统。
- 所有 Spark 基于文件输入的方法（包括textFile）都支持输入参数为：目录，压缩文件，以及通配符。例如：textFile("/my/directory"), textFile("/my/directory/*.txt"),* 以及 *textFile("/my/directory/*.gz")
- textFile 方法同时还支持一个可选参数，用以控制数据的分区个数。默认地，Spark会为文件的每一个block创建一个分区（HDFS上默认block大小为64MB），你可以通过调整这个参数来控制数据的分区数。注意，分区数不能少于block个数。除了文本文件之外，Spark的Scala API还支持其他几种数据格式：
- SparkContext.wholeTextFiles 可以读取一个包含很多小文本文件的目录，并且以 (filename, content) 键值对的形式返回结果。这与textFile 不同，textFile只返回文件的内容，每行作为一个元素。
- 对于 SequenceFiles，可以调用 SparkContext.sequenceFile[K, V]，其中 K 和 V 分别是文件中 key 和 value 的类型。这些类型都应该是 Writable 接口的子类, 如：IntWritable and Text 等。另外，Spark 允许你为一些常用Writable指定原生类型，例如：sequenceFile[Int, String] 将自动读取 IntWritable 和 Text。
- 对于其他的 Hadoop InputFormat，你可以用 SparkContext.hadoopRDD 方法，并传入任意的 JobConf 对象和 InputFormat，以及 key class、value class。这和设置 Hadoop job 的输入源是同样的方法。你还可以使用 SparkContext.newAPIHadoopRDD，该方法接收一个基于新版Hadoop MapReduce API（org.apache.hadoop.mapreduce）的InputFormat作为参数。
- RDD.saveAsObjectFile 和 SparkContext.objectFile 支持将 RDD 中元素以 Java 对象序列化的格式保存成文件。虽然这种序列化方式不如 Avro 效率高，却为保存 RDD 提供了一种简便方式。

RDD 算子

RDD 支持两种类型的算子：transformation 和 action。transformation算子可以将已有RDD转换得到一个新的RDD，而action算子则是基于RDD的计算，并将结果返回给驱动器（driver）。例如，map 是一个 transformation 算子，它将数据集中每个元素传给一个指定的函数，并将该函数返回结果构建为一个新的RDD；而 reduce 是一个 action 算子，它可以将 RDD 中所有元素传给指定的聚合函数，并将最终的聚合结果返回给驱动器（还有一个 reduceByKey 算子，其返回的聚合结果是一个 RDD）。

Spark 中所有 transformation 算子都是懒惰的，也就是说，transformation 算子并不立即计算结果，而是记录下对基础数据集（如：一个数据文件）的转换操作。只有等到某个 action 算子需要计算一个结果返回给驱动器的时候，transformation 算子所记录的操作才会被计算。这种设计使Spark可以运行得更加高效 – 例如，map算子创建了一个数据集，同时该数据集下一步会调用reduce算子，那么Spark将只会返回reduce的最终聚合结果（单独的一个数据）给驱动器，而不是将map所产生的数据集整个返回给驱动器。

默认情况下，每次调用 action 算子的时候，每个由 transformation 转换得到的RDD都会被重新计算。然而，你也可以通过调用 persist（或者cache）操作来持久化一个 RDD，这意味着 Spark 将会把 RDD 的元素都保存在集群中，因此下一次访问这些元素的速度将大大提高。同时，Spark 还支持将RDD元素持久化到内存或者磁盘上，甚至可以支持跨节点多副本。

基础

以下简要说明一下RDD的基本操作，参考如下代码：

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

其中，第一行是从外部文件加载数据，并创建一个基础RDD。这时候，数据集并没有加载进内存除非有其他操作施加于lines，这时候的lines RDD其实可以说只是一个指向 data.txt 文件的指针。第二行，用lines通过map转换得到一个lineLengths RDD，同样，lineLengths也是懒惰计算的。最后，我们使用 reduce算子计算长度之和，reduce是一个action算子。此时，Spark将会把计算分割为一些小的任务，分别在不同的机器上运行，每台机器上都运行相关的一部分map任务，并在本地进行reduce，并将这些reduce结果都返回给驱动器。

如果我们后续需要重复用到 lineLengths RDD，我们可以增加一行：

lineLengths.persist()

这一行加在调用 reduce 之前，则 lineLengths RDD 首次计算后，Spark会将其数据保存到内存中。

将函数传给Spark

Scala

Spark 的 API 很多都依赖于在驱动程序中向集群传递操作函数。以下是两种建议的实现方式：

- 匿名函数（Anonymous function syntax），这种方式代码量比较少。
- 全局单件中的静态方法。例如，你可以按如下方式定义一个 object MyFunctions 并传递其静态成员函数 MyFunctions.func1：

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

myRdd.map(MyFunctions.func1)
```

注意，技术上来说，你也可以传递一个类对象实例上的方法（不是单例对象），不过这回导致传递函数的同时，需要把相应的对象也发送到集群中各节点上。例如，我们定义一个MyClass如下：

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

如果我们 new MyClass 创建一个实例，并调用其 doStuff 方法，同时doStuff中的 map算子引用了该MyClass实例上的 func1 方法，那么接下来，这个MyClass对象将被发送到集群中所有节点上。rdd.map(x => this.func1(x)) 也会有类似的效果。

类似地，如果应用外部对象的成员变量，也会导致对整个对象实例的引用：

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

上面的代码对 field 的引用等价于 rdd.map(x => this.field + x)，这将导致应用整个this对象。为了避免类似问题，最简单的方式就是，将field固执到一个本地临时变量中，而不是从外部直接访问之，如下：

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

Java

Spark's API relies heavily on passing functions in the driver program to run on the cluster. In Java, functions are represented by classes implementing the interfaces in the org.apache.spark.api.java.function package. There are two ways to create such functions:

Implement the Function interfaces in your own class, either as an anonymous inner class or a named one, and pass an instance of it to Spark. Use lambda expressions to concisely define an implementation. While much of this guide uses lambda syntax for conciseness, it is easy to use all the same APIs in long-form. For example, we could have written our code above as follows:

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(new Function<String, Integer>() {
  public Integer call(String s) { return s.length(); }
});
int totalLength = lineLengths.reduce(new Function2<Integer, Integer, Integer>() {
  public Integer call(Integer a, Integer b) { return a + b; }
});
```

Or, if writing the functions inline is unwieldy:

```
class GetLength implements Function<String, Integer> {
  public Integer call(String s) { return s.length(); }
}
class Sum implements Function2<Integer, Integer, Integer> {
  public Integer call(Integer a, Integer b) { return a + b; }
}

JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(new GetLength());
int totalLength = lineLengths.reduce(new Sum());
```

> **注意：** anonymous inner classes in Java can also access variables in the enclosing scope as long as they are marked final. Spark will ship copies of these variables to each worker node as it does for other languages.

Python

Spark's API relies heavily on passing functions in the driver program to run on the cluster. There are three recommended ways to do this:

Lambda 表达式, for simple functions that can be written as an expression. (Lambdas do not support multi-statement functions or statements that do not return a value.) Local defs inside the function calling into Spark, for longer code. Top-level functions in a module.

For example, to pass a longer function than can be supported using a lambda, consider the code below:

```
"""MyScript.py"""
if __name__ == "__main__":
    def myFunc(s):
        words = s.split(" ")
        return len(words)
```

```
    sc = SparkContext(...)
    sc.textFile("file.txt").map(myFunc)
```

Note that while it is also possible to pass a reference to a method in a class instance (as opposed to a singleton object), this requires sending the object that contains that class along with the method. For example, consider:

```
class MyClass(object):
    def func(self, s):
        return s
    def doStuff(self, rdd):
        return rdd.map(self.func)
```

Here, if we create a new MyClass and call doStuff on it, the map inside there references the func method of that MyClass instance, so the whole object needs to be sent to the cluster.

In a similar way, accessing fields of the outer object will reference the whole object:

```
class MyClass(object):
    def __init__(self):
        self.field = "Hello"
    def doStuff(self, rdd):
        return rdd.map(lambda s: self.field + s)
```

为了避免这个问题, 最简单的方式就是将字段拷贝到一个局部变量中, 而不是外部访问:

```
def doStuff(self, rdd):
    field = self.field
    return rdd.map(lambda s: field + s)
```

理解闭包

Spark里一个比较难的事情就是，理解在整个集群上跨节点执行的变量和方法的作用域以及生命周期。Spark里一个频繁出现的问题就是RDD算子在变量作用域之外修改了其值。下面的例子，我们将会以foreach() 算子为例，来递增一个计数器counter，不过类似的问题在其他算子上也会出现。

示例

考虑如下例子，我们将会计算RDD中原生元素的总和，如果不是在同一个 JVM 中执行，其表现将有很大不同。例如，这段代码如果使用Spark本地模式（–master=local[n]）运行，和在集群上运行（例如，用spark-submit提交到YARN上）结果完全不同。

Scala

```
var counter = 0
var rdd = sc.parallelize(data)

// Wrong: Don't do this!!
rdd.foreach(x => counter += x)

println("Counter value: " + counter)
```

Java

```
int counter = 0;
JavaRDD<Integer> rdd = sc.parallelize(data);

// Wrong: Don't do this!!
rdd.foreach(x -> counter += x);

println("Counter value: " + counter);
```

Python

```
counter = 0
rdd = sc.parallelize(data)

# Wrong: Don't do this!!
def increment_counter(x):
    global counter
    counter += x
rdd.foreach(increment_counter)

print("Counter value: ", counter)
```

本地模式 VS 集群模式

上面这段代码其行为是不确定的。在本地模式下运行，所有代码都在运行于单个JVM中，所以RDD的元素都能够被累加并保存到counter变量中，这是因为本地模式下，counter变量和驱动器节点在同一个内存空间中。

然而，在集群模式下，情况会更复杂，以上代码的运行结果就不是所预期的结果了。为了执行这个作业，Spark会将 RDD 算子的计算过程分割成多个独立的任务（task）- 每个任务分发给不同的执行器（executor）去执行。而执行之前，Spark需要计算闭包。闭包是由执行器执行RDD算子（本例中的foreach()）时所需要的变量和方法组成的。闭包将会被序列化，并发送给每个执行器。由于本地模式下，只有一个执行器，所有任务都共享同样的闭包。而在其他模式下，情况则有所不同，每个执行器都运行于不同的worker节点，并且都拥有独立的闭包副本。

在上面的例子中，闭包中的变量会跟随不同的闭包副本，发送到不同的执行器上，所以等到foreach真正在执行器上运行时，其引用的counter已经不再是驱动器上所定义的那个counter副本了，驱动器内存中仍然会有一个counter变量副本，但是这个副本对执行器是不可见的！执行器只能看到其所收到的序列化闭包中包含的counter副本。因此，最终驱动器上得到的counter将会是0。

为了确保类似这样的场景下，代码能有确定的行为，这里应该使用累加器（Accumulator）。累加器是Spark中专门用于集群跨节点分布式执行计算中，安全地更新同一变量的机制。本指南中专门有一节详细说明累加器。

通常来说，闭包（由循环或本地方法组成），不应该改写全局状态。Spark中改写闭包之外对象的行为是未定义的。这种代码，有可能在本地模式下能正常工作，但这只是偶然情况，同样的代码在分布式模式下其行为很可能不是你想要的。所以，如果需要全局聚合，请记得使用累加器（Accumulator）。

打印 RDD 中的元素

另一种常见习惯是，试图用 rdd.foreach(println) 或者 rdd.map(println) 来打印RDD中所有的元素。如果是在单机上，这种写法能够如预期一样，打印出RDD所有元素。然后，在集群模式下，这些输出将会被打印到执行器的标准输出（stdout）上，因此驱动器的标准输出（stdout）上神马也看不到！如果真要在驱动器上把所有RDD元素都打印出来，你可以先调用collect算子，把RDD元素先拉到驱动器上来，代码可能是这样：rdd.collect().foreach(println)。不过如果RDD很大的话，有可能导致驱动器内存溢出，因

为collect会把整个RDD都弄到驱动器所在单机上来；如果你只是需要打印一部分元素，那么take是更安全的选择：rdd.take(100).foreach(println)

使用键值对

大部分Spark算子都能在包含任意类型对象的RDD上工作，但也有一部分特殊的算子要求RDD包含的元素必须是键值对（key-value pair）。这种算子常见于做分布式混洗（shuffle）操作，如：以key分组或聚合。

在Scala中，这种操作在包含 Tuple2 （内建与scala语言，可以这样创建：(a, b) ）类型对象的RDD上自动可用。键值对操作是在 PairRDDFunctions 类上可用，这个类型也会自动包装到包含tuples的RDD上。

例如，以下代码将使用 reduceByKey 算子来计算文件中每行文本出现的次数：

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

同样，我们还可以用 counts.sortByKey() 来对这些键值对按字母排序，最后再用 counts.collect() 将数据以对象数据组的形式拉到驱动器内存中。

注意：如果使用自定义类型对象做键值对中的key的话，你需要确保自定义类型实现了 equals() 方法（通常需要同时也实现hashCode()方法）。完整的细节可以参考：Object.hashCode()文档

转换算子 – transformation

以下是Spark支持的一些常用transformation算子。详细请参考 RDD API doc (Scala, Java, Python, R) 以及 键值对 RDD 函数 (Scala, Java)。

Transformation算子	含义
map(func)	返回一个新的分布式数据集，其中每个元素都是由源RDD中一个元素经func转换得到的。
filter(func)	返回一个新的数据集，其中包含的元素来自源RDD中元素经func过滤后（func返回true时才选中）的结果
flatMap(func)	类似于map，但每个输入元素可以映射到0到n个输出元素（所以要求func必须返回一个Seq而不是单个元素）
mapPartitions(func)	类似于map，但基于每个RDD分区（或者数据block）独立运行，所以如果RDD包含元素类型为T，则 func 必须是 Iterator<T> => Iterator<U> 的映射函数。
mapPartitionsWithIndex(func)	类似于 mapPartitions，只是func 多了一个整型的分区索引值，因此如果RDD包含元素类型为T，则 func 必须是 Iterator<T> => Iterator<U> 的映射函数。
sample(withReplacement, fraction, seed)	采样部分（比例取决于 fraction）数据，同时可以指定是否使用回置采样（withReplacement），以及随机数种子(seed)
union(otherDataset)	返回源数据集和参数数据集（otherDataset）的并集
intersection(otherDataset)	返回源数据集和参数数据集（otherDataset）的交集
distinct([numTasks]))	返回对源数据集做元素去重后的新数据集
groupByKey([numTasks])	只对包含键值对的RDD有效，如源RDD包含 (K, V) 对，则该算子返回一个新的数据集包含(K, Iterable<V>) 对。注意：如果你需要按key分组聚合的话（如sum或average），推荐使用 reduceByKey或者 aggregateByKey 以获得更好的性能。注意：默认情况下，输出计算的并行度取决于源RDD的分区个数。当然，你也可以通过设置可选参数 numTasks 来指定并行任务的个数。
reduceByKey(func, [numTasks])	如果源RDD包含元素类型 (K, V) 对，则该算子也返回包含(K, V) 对的RDD，只不过每个key对应的value是经过func聚合后的结果，而func本身是一个 (V, V) => V 的映射函数。另外，和 groupByKey 类似，可以通过可选参数 numTasks 指定reduce任务的个数。
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	如果源RDD包含 (K, V) 对，则返回新RDD包含 (K, U) 对，其中每个key对应的value都是由 seqOp 函数 和 一个“0”值zeroValue 聚合得到。允许聚合后value类型和输入value类型不同，避免了不必要的开销。和 groupByKey 类似，可以通过可选参数 numTasks 指定reduce任务的个数。
sortByKey([ascending], [numTasks])	如果源RDD包含元素类型 (K, V) 对，其中K可排序，则返回新的RDD包含 (K, V) 对，并按照 K 排序（升序还是降序取决于 ascending 参数）
join(otherDataset, [numTasks])	如果源RDD包含元素类型 (K, V) 且参数RDD（otherDataset）包含元素类型(K, W)，则返回的新RDD中将包含内关联后key对应的 (K, (V, W)) 对。外关联(Outer joins)操作请参考 leftOuterJoin、rightOuterJoin 以及 fullOuterJoin 算子。
cogroup(otherDataset, [numTasks])	如果源RDD包含元素类型 (K, V) 且参数RDD（otherDataset）包含元素类型(K, W)，则返回的新RDD中包含 (K, (Iterable<V>, Iterable<W>))。该算子还有个别名：groupWith
cartesian(otherDataset)	如果源RDD包含元素类型 T 且参数RDD（otherDataset）包含元素类型 U，则返回的新RDD包含前二者的笛卡尔积，其元素类型为 (T, U) 对。
pipe(command, [envVars])	以shell命令行管道处理RDD的每个分区，如：Perl 或者 bash 脚本。RDD中每个元素都将依次写入进程的标准输入（stdin），然后按行输出到标准输出（stdout），每一行输出字符串即成为一个新的RDD元素。
coalesce(numPartitions)	将RDD的分区数减少到numPartitions。当以后大数据集被过滤成小数据集后，减少分区数，可以提升效率。
repartition(numPartitions)	将RDD数据重新混洗（reshuffle）并随机分布到新的分区中，使数据分布更均衡，新的分区个数取决于numPartitions。该算子总是需要通过网络混洗所有数据。
repartitionAndSortWithinPartitions(partitioner)	根据partitioner（spark自带有HashPartitioner和RangePartitioner等）重新分区RDD，并且在每个结果分区中按key做排序。这是一个组合算子，功能上等价于先 repartition 再在每个分区内排序，但这个算子内部做了优化（将排序过程下推到混洗同时进行），因此性能更好。

动作算子 – action

以下是Spark支持的一些常用action算子。详细请参考 RDD API doc (Scala, Java, Python, R) 以及 键值对 RDD 函数 (Scala, Java) 。

Action算子	作用
reduce(func)	将RDD中元素按func进行聚合（func是一个 (T,T) => T 的映射函数，其中T为源RDD元素类型，并且func需要满足 交换律 和 结合律 以便支持并行计算）
collect()	将数据集中所有元素以数组形式返回驱动器（driver）程序。通常用于，在RDD进行了filter或其他过滤操作后，将一个足够小的数据子集返回到驱动器内存中。
count()	返回数据集中元素个数
first()	返回数据集中首个元素（类似于 take(1) ）
take(n)	返回数据集中前 n 个元素
takeSample(withReplacement,num, [seed])	返回数据集的随机采样子集，最多包含 num 个元素，withReplacement 表示是否使用回置采样，最后一个参数为可选参数seed，随机数生成器的种子。
takeOrdered(n, [ordering])	按元素排序（可以通过 ordering 自定义排序规则）后，返回前 n 个元素
saveAsTextFile(path)	将数据集中元素保存到指定目录下的文本文件中（或者多个文本文件），支持本地文件系统、HDFS 或者其他任何Hadoop支持的文件系统。保存过程中，Spark会调用每个元素的toString方法，并将结果保存成文件中的一行。
saveAsSequenceFile(path)(Java and Scala)	将数据集中元素保存到指定目录下的Hadoop Sequence文件中，支持本地文件系统、HDFS 或者其他任何Hadoop支持的文件系统。适用于实现了Writable接口的键值对RDD。在Scala中，同样也适用于能够被隐式转换为Writable的类型（Spark实现了所有基本类型的隐式转换，如：Int，Double，String 等）
saveAsObjectFile(path)(Java and Scala)	将RDD元素以Java序列化的格式保存成文件，保存结果文件可以使用 SparkContext.objectFile 来读取。
countByKey()	只适用于包含键值对(K, V)的RDD，并返回一个哈希表，包含 (K, Int) 对，表示每个key的个数。
foreach(func)	在RDD的每个元素上运行 func 函数。通常被用于累加操作，如：更新一个累加器（Accumulator） 或者 和外部存储系统互操作。

注意：用 foreach 操作出累加器之外的变量可能导致未定义的行为。更详细请参考前面的“理解闭包”（Understanding closures ）这一小节。

混洗(Shuffle)算子

有一些Spark算子会触发众所周知的混洗（Shuffle）事件。Spark中的混洗机制是用于将数据重新分布，其结果是所有数据将在各个分区间重新分组。一般情况下，混洗需要跨执行器（Executor）或跨机器复制数据，这也是混洗操作一般都比较复杂而且开销大的原因。

背景

为了理解混洗阶段都发生了哪些事，我首先以 reduceByKey 算子为例来看一下。reduceByKey算子会生成一个新的RDD，将源RDD中一个key对应的多个value组合进一个tuple - 然后将这些values输入给reduce函数，得到的result再和key关联放入新的RDD中。这个算子的难点在于对于某一个key来说，并非其对应的所

有values都在同一个分区（partition）中，甚至有可能都不在同一台机器上，但是这些values又必须放到一起计算reduce结果。

在Spark中，通常是由于为了进行某种计算操作，而将数据分布到所需要的各个分区当中。而在计算阶段，单个任务（task）只会操作单个分区中的数据 – 因此，为了组织好每个 reduceByKey 中 reduce 任务执行时所需的数据，Spark需要执行一个多对多操作。即，Spark需要读取RDD的所有分区，并找到所有key对应的所有values，然后跨分区传输这些values，并将每个key对应的所有values放到同一分区，以便后续计算各个key对应values的reduce结果 – 这个过程就叫做混洗（Shuffle）。

虽然混洗好后，各个分区中的元素和分区自身的顺序都是确定的，但是分区中元素的顺序并非确定的。如果需要混洗后分区内的元素有序，可以参考使用以下混洗操作：

- mapPartitions 使用 .sorted 对每个分区排序
- repartitionAndSortWithinPartitions 重分区的同时，对分区进行排序，比自行组合repartition和sort更高效
- sortBy 创建一个全局有序的RDD

会导致混洗的算子有：重分区（repartition）类算子，如： repartition 和 coalesce；ByKey 类算子(除了计数类的，如 countByKey) 如：groupByKey 和 reduceByKey；以及Join类算子，如：cogroup 和 join.

性能影响

混洗（Shuffle）之所以开销大，是因为混洗操作需要引入磁盘I/O，数据序列化以及网络I/O等操作。为了组织好混洗数据，Spark需要生成对应的任务集 – 一系列map任务用于组织数据，再用一系列reduce任务来聚合数据。注意这里的map、reduce是来自MapReduce的术语，和Spark的map、reduce算子并没有直接关系。

在Spark内部，单个map任务的输出会尽量保存在内存中，直至放不下为止。然后，这些输出会基于目标分区重新排序，并写到一个文件里。在reduce端，reduce任务只读取与之相关的并已经排序好的blocks。

某些混洗算子会导致非常明显的内存开销增长，因为这些算子需要在数据传输前后，在内存中维护组织数据记录的各种数据结构。特别地，reduceByKey和aggregateByKey都会在map端创建这些数据结构，而ByKey系列算子都会在reduce端创建这些数据结构。如果数据在内存中存不下，Spark会把数据吐到磁盘上，当然这回导致额外的磁盘I/O以及垃圾回收的开销。

混洗还会再磁盘上生成很多临时文件。以Spark-1.3来说，这些临时文件会一直保留到其对应的RDD被垃圾回收才删除。之所以这样做，是因为如果血统信息需要重新计算的时候，这些混洗文件可以不必重新生成。如果程序持续引用这些RDD或者垃圾回收启动频率较低，那么这些垃圾回收可能需要等较长的一段时间。这就意味着，长时间运行的Spark作业可能会消耗大量的磁盘。Spark的临时存储目录，是由spark.local.dir 配置参数指定的。

混洗行为可以由一系列配置参数来调优。参考Spark配置指南中”混洗行为”这一小节。

RDD持久化

Spark的一项关键能力就是它可以持久化（或者缓存）数据集在内存中，从而跨操作复用这些数据集。如果你持久化了一个RDD，那么每个节点上都会存储该RDD的一些分区，这些分区是由对应的节点计算出来并保持在内存中，后续可以在其他施加在该RDD上的action算子中复用（或者从这些数据集派生新的RDD）。这使得后续动作的速度提高很多（通常高于10倍）。因此，缓存对于迭代算法和快速交互式分析是一个很关键的工具。

你可以用persist() 或者 cache() 来标记一下需要持久化的RDD。等到该RDD首次被施加action算子的时候，其对应的数据分区就会被保留在内存里。同时，Spark的缓存具备一定的容错性 – 如果RDD的任何一个分区丢失了，Spark将自动根据其原来的血统信息重新计算这个分区。

另外，每个持久化的RDD可以使用不同的存储级别，比如，你可以把RDD保存在磁盘上，或者以java序列化对象保存到内存里（为了省空间），或者跨节点多副本，或者使用 Tachyon 存到虚拟机以外的内存里。这些存储级别都可以由persist()的参数StorageLevel对象来控制。cache() 方法本身就是一个使用默认存储级别做持

久化的快捷方式，默认存储级别是 StorageLevel.MEMORY_ONLY（以Java序列化方式存到内存里）。完整的存储级别列表如下：

存储级别	含义
MEMORY_ONLY	以未序列化的 Java 对象形式将 RDD 存储在 JVM 内存中。如果RDD不能全部装进内存，那么将一部分分区缓存，而另一部分分区将每次用到时重新计算。这个是Spark的RDD的默认存储级别。
MEMORY_AND_DISK	以未序列化的Java对象形式存储RDD在JVM中。如果RDD不能全部装进内存，则将不能装进内存的分区放到磁盘上，然后每次用到的时候从磁盘上读取。
MEMORY_ONLY_SER	以序列化形式存储 RDD（每个分区一个字节数组）。通常这种方式比未序列化存储方式要更省空间，尤其是如果你选用了一个比较好的序列化协议（fast serializer），但是这种方式也相应的会消耗更多的CPU来读取数据。
MEMORY_AND_DISK_SER	和 MEMORY_ONLY_SER 类似，只是当内存装不下的时候，会将分区的数据吐到磁盘上，而不是每次用到都重新计算。
DISK_ONLY	RDD 数据只存储于磁盘上。
MEMORY_ONLY_2, MEMORY_AND_DISK_2等	和上面没有”_2″ 的级别相对应，只不过每个分区数据会在两个节点上保存两份副本。
OFF_HEAP(实验性的)	将RDD以序列化格式保存到Tachyon。与MEMORY_ONLY_SER相比，OFF_HEAP减少了垃圾回收开销，并且使执行器（executor）进程更小且可以共用同一个内存池，这一特性在需要大量消耗内存和多Spark应用并发的场景下比较吸引人。而且，因为RDD存储于Tachyon中，所以一个执行器挂了并不会导致数据缓存的丢失。这种模式下Tachyon 的内存是可丢弃的。因此，Tachyon并不会重建一个它逐出内存的block。如果你打算用Tachyon做为堆外存储，Spark和Tachyon具有开箱即用的兼容性。请参考这里，有建议使用的Spark和Tachyon的匹配版本对：page。

注意：在Python中存储的对象总是会使用 Pickle 做序列化，所以这时是否选择一个序列化级别已经无关紧要了。

Spark会自动持久化一些混洗操作（如：reduceByKey）的中间数据，即便用户根本没有调用persist。这么做是为了避免一旦有一个节点在混洗过程中失败，就要重算整个输入数据。当然，我们还是建议对需要重复使用的RDD调用其persist算子。

如何选择存储级别?

Spark的存储级别主要可于在内存使用和CPU占用之间做一些权衡。建议根据以下步骤来选择一个合适的存储级别：

- 如果RDD能使用默认存储级别（MEMORY_ONLY），那就尽量使用默认级别。这是CPU效率最高的方式，所有RDD算子都能以最快的速度运行。
- 如果步骤1的答案是否（不适用默认级别），那么可以尝试MEMORY_ONLY_SER级别，并选择一个高效的序列化协议（selecting a fast serialization library），这回大大节省数据对象的存储空间，同时速度也还不错。
- 尽量不要把数据吐到磁盘上，除非：1.你的数据集重新计算的代价很大；2.你的数据集是从一个很大的数据源中过滤得到的结果。否则的话，重算一个分区的速度很可能和从磁盘上读取差不多。
- 如果需要支持容错，可以考虑使用带副本的存储级别（例如：用Spark来服务web请求）。所有的存储级别都能够以重算丢失数据的方式来提供容错性，但是带副本的存储级别可以让你的应用持续的运行，而不必等待重算丢失的分区。
- **在一些需要大量内存或者并行多个应用的场景下，实验性的OFF_HEAP会有以下几个优势：**

- 这个级别下，可以允许多个执行器共享同一个Tachyon中内存池。
- 可以有效地减少垃圾回收的开销。
- 即使单个执行器挂了，缓存数据也不会丢失。

删除数据

Spark能够自动监控各个节点上缓存使用率，并且以LRU（最近经常使用）的方式将老数据逐出内存。如果你更喜欢手动控制的话，可以用RDD.unpersist() 方法来删除无用的缓存。

1.2.5 共享变量

一般而言，当我们给Spark算子（如 map 或 reduce）传递一个函数时，这些函数将会在远程的集群节点上运行，并且这些函数所引用的变量都是各个节点上的独立副本。这些变量都会以副本的形式复制到各个机器节点上，如果更新这些变量副本的话，这些更新并不会传回到驱动器（driver）程序。通常来说，支持跨任务的可读写共享变量是比较低效的。不过，Spark还是提供了两种比较通用的共享变量：广播变量和累加器。

广播变量

广播变量提供了一种只读的共享变量，它是在每个机器节点上保存一个缓存，而不是每个任务保存一份副本。通常可以用来在每个节点上保存一个较大的输入数据集，这要比常规的变量副本更高效（一般的变量是每个任务一个副本，一个节点上可能有多个任务）。Spark还会尝试使用高效的广播算法来分发广播变量，以减少通信开销。

Spark的操作有时会有多个阶段（stage），不同阶段之间的分割线就是混洗操作。Spark会自动广播各个阶段用到的公共数据。这些方式广播的数据都是序列化过的，并且在运行各个任务前需要反序列化。这也意味着，显示地创建广播变量，只有在跨多个阶段（stage）的任务需要同样的数据 或者 缓存数据的序列化和反序列化格式很重要的情况下 才是必须的。

广播变量可以通过一个变量v来创建，只需调用 SparkContext.broadcast(v)即可。这个广播变量是对变量v的一个包装，要访问其值，可以调用广播变量的 value 方法。代码示例如下：

scala> val broadcastVar = sc.broadcast(Array(1, 2, 3)) broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

scala> broadcastVar.value res0: Array[Int] = Array(1, 2, 3)

广播变量创建之后，集群中任何函数都不应该再使用原始变量v，这样才能保证v不会被多次复制到同一个节点上。另外，对象v在广播后不应该再被更新，这样才能保证所有节点上拿到同样的值（例如，更新后，广播变量又被同步到另一新节点，新节点有可能得到的值和其他节点不一样）。

累加器

累加器是一种只支持满足结合律的“累加”操作的变量，因此它可以很高效地支持并行计算。利用累加器可以实现计数（类似MapReduce中的计数器）或者求和。Spark原生支持了数字类型的累加器，开发者也可以自定义新的累加器。如果创建累加器的时候给了一个名字，那么这个名字会展示在Spark UI上，这对于了解程序运行处于哪个阶段非常有帮助（注意：Python尚不支持该功能）。

创捷累加器时需要赋一个初始值v，调用 SparkContext.accumulator(v) 可以创建一个累加器。后续集群中运行的任务可以使用 add 方法 或者 += 操作符 （仅Scala和Python支持）来进行累加操作。不过，任务本身并不能读取累加器的值，只有驱动器程序可以用 value 方法访问累加器的值。

以下代码展示了如何使用累加器对一个元素数组求和：

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

以上代码使用了Spark内建支持的Int型累加器，开发者也可以通过子类化 AccumulatorParam 来自定义累加器。累加器接口（AccumulatorParam ）主要有两个方法：1. zero：这个方法为累加器提供一个“零值”，2.addInPlace 将收到的两个参数值进行累加。例如，假设我们需要为Vector提供一个累加机制，那么可能的实现方式如下：

```
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)
```

如果使用Scala，Spark还支持几种更通用的接口：1.Accumulable，这个接口可以支持所累加的数据类型与结果类型不同（如：构建一个收集元素的list）；2.SparkContext.accumulableCollection 方法可以支持常用的Scala集合类型。

对于在action算子中更新的累加器，Spark保证每个任务对累加器的更新只会被应用一次，例如，某些任务如果重启过，则不会再次更新累加器。而如果在transformation算子中更新累加器，那么用户需要注意，一旦某个任务因为失败被重新执行，那么其对累加器的更新可能会实施多次。

累加器并不会改变Spark懒惰求值的运算模型。如果在RDD算子中更新累加器，那么其值只会在RDD做action算子计算的时候被更新一次。因此，在transformation算子（如：map）中更新累加器，其值并不能保证一定被更新。以下代码片段说明了这一特性：

val accum = sc.accumulator(0) data.map { x => accum += x; f(x) } // 这里，accum任然是0，因为没有action算子，所以map也不会进行实际的计算

1.2.6 部署到集群

提交 *Spark* 应用程序 中描述了如何向集群提交应用。换句话说，就是你需要把你的应用打包成 JAR文件（Java/Scala）或者一系列 .py 或 .zip 文件（Python），然后再用 bin/spark-submit 脚本将其提交给Spark所支持的集群管理器。

1.2.7 从Java/Scala中启动Spark作业

org.apache.spark.launcher 包提供了简明的Java API，可以将Spark作业作为子进程启动。

1.2.8 单元测试

Spark is friendly to unit testing with any popular unit test framework. Simply create a SparkContext in your test with the master URL set to local, run your operations, and then call SparkContext.stop() to tear it down. Make sure you stop the context within a finally block or the test framework’s tearDown method, as Spark does not support two contexts running concurrently in the same program. Spark对所有常见的单元测试框架提供友好的支持。你只需要在测试

中创建一个SparkContext对象，然后吧master URL设为local，运行测试操作，最后调用 SparkContext.stop() 来停止测试。注意，一定要在 finally 代码块或者单元测试框架的 tearDown方法里调用SparkContext.stop()，因为Spark不支持同一程序中有多个SparkContext对象同时运行。

1.2.9 下一步

你可以去 Spark 官网上看看示例程序（example Spark programs）。另外，Spark代码目录下也自带了不少例子，见 examples 目录(Scala,Java, Python, R)。你可以把示例中的类名传给 bin/run-example 脚本来运行这些例子；例如：

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

对于 Python 示例，使用 spark-submit：

```
./bin/spark-submit examples/src/main/python/pi.py
```

对于 R 示例，使用 spark-submit：

```
./bin/spark-submit examples/src/main/r/dataframe.R
```

配置（configuration）和调优（tuning）指南提供了不少最佳实践的信息，可以帮助你优化程序，特别是这些信息可以帮助你确保数据以一种高效的格式保存在内存里。集群模式概览这篇文章描述了分布式操作中相关的组件，以及Spark所支持的各种集群管理器。

最后，完整的API文件见：Scala, Java, Python 以及 R.

1.3 Spark SQL, DataFrame 和 Dataset 编程指南

1.3.1 概述

Spark SQL 是 Spark 用于处理结构化数据的一个模块。不同于基础的 Spark RDD API，Spark SQL 提供的接口提供了更多关于数据和执行的计算任务的结构信息。Spark SQL 内部使用这些额外的信息来执行一些额外的优化操作。有几种方式可以与 Spark SQL 进 行交互，其中包括 SQL 和 Dataset API。当计算一个结果时 Spark SQL 使用的执行引擎是一样的, 它跟你使用哪种 API 或编程语 言来表达计算无关。这种统一意味着开发人员可以很容易地在不同的 API 之间来回切换，基于哪种 API 能够提供一种最自然的方式来表 达一个给定的变换（transformation）。

本文中所有的示例程序都使用 Spark 发行版本中自带的样本数据，并且可以在 spark-shell、pyspark shell 以及 sparkR shell 中运行。

SQL

Spark SQL 的用法之一是执行 SQL 查询，它也可以从现有的 Hive 中读取数据，想要了解更多关于如何配置这个特性的细节, 请参考 Hive表 这节。 如果从其它编程语言内部运行 SQL，查询结果将作为一个 Dataset/DataFrame 返回。你还可以使用命令行或者通过 JDBC/ODBC 来 与 SQL 接口进行交互。

Dataset 和 DataFrame

Dataset 是一个分布式数据集，它是 Spark 1.6 版本中新增的一个接口, 它结合了 RDD（强类型，可以使用强大的 lambda 表达式函数） 和 Spark SQL 的优化执行引擎的好处。Dataset 可以从 JVM 对象构造得到，随后可以使用函数式的变换（map，flatMap，filter 等） 进行操作。Dataset API 目前支持 Scala 和 Java 语言，还

不支持 Python, 不过由于 Python 语言的动态性, Dataset API 的许 多好处早就已经可用了（例如，你可以使用 row.columnName 来访问数据行的某个字段）。这种场景对于 R 语言来说是类似的。

DataFrame 是按命名列方式组织的一个 Dataset。从概念上来讲，它等同于关系型数据库中的一张表或者 R 和 Python 中的一个 data frame， 只不过在底层进行了更多的优化。DataFrame 可以从很多数据源构造得到，比如：结构化的数据文件，Hive 表，外部数据库或现有的 RDD。 DataFrame API 支持 Scala, Java, Python 以及 R 语言。在 Scala 和 Java 语言中, DataFrame 由 Row 的 Dataset 来 表示的。在 Scala API 中, DataFrame 仅仅只是 Dataset[Row] 的一个类型别名，而在 Java API 中, 开发人员需要使用 Dataset<Row> 来 表示一个 DataFrame。

在本文档中, 我们将经常把 Scala/Java Row 的 Dataset 作为 DataFrame。

1.3.2 入门

入口: SparkSession

Scala

Spark 中所有功能的入口是 SparkSession 类。要创建一个基本的 SparkSession 对象, 只需要使用 SparkSession.builder():

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala” 文件。

Java

Spark 中所有功能的入口是 SparkSession 类。要创建一个基本的 SparkSession 对象, 只需要使用 SparkSession.builder():

```
import org.apache.spark.sql.SparkSession;

SparkSession spark = SparkSession
  .builder()
  .appName("Java Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate();
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java” 文件。

Python

Spark 中所有功能的入口是 SparkSession 类。要创建一个基本的 SparkSession 对象, 只需要使用 SparkSession.builder:

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/basic.py” 文件。

R

Spark 中所有功能的入口是 SparkSession 类。要初始化一个基本的 SparkSession 对象，只需要调用 sparkR.session():

```
sparkR.session(appName = "R Spark SQL basic example", sparkConfig = list(spark.some.
↪config.option = "some-value"))
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/r/RSparkSQLExample.R” 文件。

注意： 当第一次调用时, sparkR.session() 会初始化一个全局的 SparkSession 单例实例, 并且总是为后续的调用返回该实例的引用。这样的话, 用户只需要初始化 SparkSession 一次, 然后像 read.df 这样的 SparkR 函数就可以隐式地访问该全局实例, 并且用户不需要传递 SparkSession 实例。

Spark 2.0 中的 SparkSession 提供了对 Hive 特性的内置支持，包括使用 HiveQL 编写查询，访问 Hive UDF 以及从 Hive 表读取数据。要使用这些特性，你不需要预先安装 Hive。

创建 DataFrame

Scala

应用程序可以使用 SparkSession 从一个现有的 RDD，Hive 表或 Spark 数据源创建 DataFrame。

举个例子, 下面基于一个 JSON 文件的内容创建一个 DataFrame:

```
val df = spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala” 文件。

Java

应用程序可以使用 SparkSession 从一个现有的 RDD，Hive 表或 Spark 数据源创建 DataFrame。

举个例子, 下面基于一个 JSON 文件的内容创建一个 DataFrame:

```
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
```

```
Dataset<Row> df = spark.read().json("examples/src/main/resources/people.json");

// Displays the content of the DataFrame to stdout
df.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java” 文件。

Python

应用程序可以使用 SparkSession 从一个现有的 RDD，Hive 表或 Spark 数据源创建 DataFrame。

举个例子, 下面基于一个 JSON 文件的内容创建一个 DataFrame:

```
# spark is an existing SparkSession
df = spark.read.json("examples/src/main/resources/people.json")
# Displays the content of the DataFrame to stdout
df.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/basic.py” 文件。

R

应用程序可以使用 SparkSession 从一个本地的 R data.frame, Hive 表或 Spark 数据源创建 DataFrame。

举个例子, 下面基于一个 JSON 文件的内容创建一个 DataFrame:

```
df <- read.json("examples/src/main/resources/people.json")

# Displays the content of the DataFrame
head(df)
##   age    name
## 1  NA Michael
## 2  30    Andy
## 3  19  Justin

# Another method to print the first few rows and optionally truncate the printing of␣
↪long values
showDF(df)
## +----+-------+
## | age|   name|
## +----+-------+
## |null|Michael|
## |  30|   Andy|
## |  19| Justin|
## +----+-------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/r/RSparkSQLExample.R” 文件。

无类型的 Dataset 操作 (亦即 DataFrame 操作)

DataFrame 为 Scala, Java, Python 以及 R 语言中的结构化数据操作提供了一种领域特定语言。

正如上面所提到的,Spark 2.0 中, Scala 和 Java API 中的 DataFrame 只是 Row 的 Dataset。与使用强类型的 Scala/Java Dataset “强类型转换” 相比，这些操作也被称为 “非强类型转换” 。 These operations are also referred as “untyped transformations” in contrast to “typed transformations” come with strongly typed Scala/Java Datasets.

下面是使用 Dataset 处理结构化数据的几个基础示例：

Scala

```
// This import is needed to use the $-notation
import spark.implicits._
// Print the schema in a tree format
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+

// Select everybody, but increment the age by 1
df.select($"name", $"age" + 1).show()
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+

// Select people older than 21
df.filter($"age" > 21).show()
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+

// Count people by age
df.groupBy("age").count().show()
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
```

```
// +----+-----+
```

完整的示例代码参见 Spark 源码仓库中的“examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala”文件。

Java

```
// col("...") is preferable to df.col("...")
import static org.apache.spark.sql.functions.col;

// Print the schema in a tree format
df.printSchema();
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show();
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+

// Select everybody, but increment the age by 1
df.select(col("name"), col("age").plus(1)).show();
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+

// Select people older than 21
df.filter(col("age").gt(21)).show();
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+

// Count people by age
df.groupBy("age").count().show();
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

完整的示例代码参见 Spark 源码仓库中的“examples/src/main/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java”文件。

Python

In Python it's possible to access a DataFrame's columns either by attribute (df.age) or by indexing (df['age']). While the former is convenient for interactive data exploration, users are highly encouraged to use the latter form, which is future proof and won't break with column names that are also attributes on the DataFrame class.

```
# spark, df are from the previous example
# Print the schema in a tree format
df.printSchema()
# root
# |-- age: long (nullable = true)
# |-- name: string (nullable = true)

# Select only the "name" column
df.select("name").show()
# +-------+
# |   name|
# +-------+
# |Michael|
# |   Andy|
# | Justin|
# +-------+

# Select everybody, but increment the age by 1
df.select(df['name'], df['age'] + 1).show()
# +-------+---------+
# |   name|(age + 1)|
# +-------+---------+
# |Michael|     null|
# |   Andy|       31|
# | Justin|       20|
# +-------+---------+

# Select people older than 21
df.filter(df['age'] > 21).show()
# +---+----+
# |age|name|
# +---+----+
# | 30|Andy|
# +---+----+

# Count people by age
df.groupBy("age").count().show()
# +----+-----+
# | age|count|
# +----+-----+
# |  19|    1|
# |null|    1|
# |  30|    1|
# +----+-----+
```

完整的示例代码参见 Spark 源码仓库中的 "examples/src/main/python/sql/basic.py" 文件。

R

```
# Create the DataFrame
df <- read.json("examples/src/main/resources/people.json")

# Show the content of the DataFrame
head(df)
##   age    name
```

```
## 1  NA Michael
## 2  30    Andy
## 3  19  Justin


# Print the schema in a tree format
printSchema(df)
## root
## |-- age: long (nullable = true)
## |-- name: string (nullable = true)

# Select only the "name" column
head(select(df, "name"))
##      name
## 1 Michael
## 2    Andy
## 3  Justin

# Select everybody, but increment the age by 1
head(select(df, df$name, df$age + 1))
##      name (age + 1.0)
## 1 Michael          NA
## 2    Andy          31
## 3  Justin          20

# Select people older than 21
head(where(df, df$age > 21))
##   age name
## 1  30 Andy

# Count people by age
head(count(groupBy(df, "age")))
##   age count
## 1  19     1
## 2  NA     1
## 3  30     1
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/r/RSparkSQLExample.R” 文件。

Running SQL Queries Programmatically

Scala

The sql function on a SparkSession enables applications to run SQL queries programmatically and returns the result as a DataFrame.

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
```

```
// +----+-------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala” 文件。

Java

The sql function on a SparkSession enables applications to run SQL queries programmatically and returns the result as a Dataset<Row>.

```
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people");

Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
sqlDF.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java” 文件。

Python

The sql function on a SparkSession enables applications to run SQL queries programmatically and returns the result as a DataFrame.

```
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/basic.py” 文件。

R

The sql function enables applications to run SQL queries programmatically and returns the result as a SparkDataFrame.

```
df <- sql("SELECT * FROM table")
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/r/RSparkSQLExample.R” 文件。

Global Temporary View

Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates. If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view. Global temporary view is tied to a system preserved database global_temp, and we must use the qualified name to refer it, e.g. SELECT * FROM global_temp.view1.

Scala

```
// Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala” 文件。

Java

```
// Register the DataFrame as a global temporary view
df.createGlobalTempView("people");

// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java” 文件。

Python

```
# Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

# Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+

# Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/basic.py” 文件。

Sql

```
CREATE GLOBAL TEMPORARY VIEW temp_view AS SELECT a + 1, b * 2 FROM tbl
SELECT * FROM global_temp.temp_view
```

创建 Dataset

Datasets are similar to RDDs, however, instead of using Java serialization or Kryo they use a specialized Encoder to serialize the objects for processing or transmitting over the network. While both encoders and standard serialization are responsible for turning an object into bytes, encoders are code generated dynamically and use a format that allows Spark to perform many operations like filtering, sorting and hashing without deserializing the bytes back into an object.

Scala

```
// 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: Long)

// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS.show()
// +----+---+
// |name|age|
// +----+---+
// |Andy| 32|
```

```
// +----+---+

// Encoders for most common types are automatically provided by importing spark.
↪implicits._
val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)

// DataFrames can be converted to a Dataset by providing a class. Mapping will be␣
↪done by name
val path = "examples/src/main/resources/people.json"
val peopleDS = spark.read.json(path).as[Person]
peopleDS.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala” 文件。

Java

```
import java.util.Arrays;
import java.util.Collections;
import java.io.Serializable;

import org.apache.spark.api.java.function.MapFunction;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Encoder;
import org.apache.spark.sql.Encoders;

public static class Person implements Serializable {
  private String name;
  private int age;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }
}

// Create an instance of a Bean class
Person person = new Person();
```

```
person.setName("Andy");
person.setAge(32);

// Encoders are created for Java beans
Encoder<Person> personEncoder = Encoders.bean(Person.class);
Dataset<Person> javaBeanDS = spark.createDataset(
  Collections.singletonList(person),
  personEncoder
);
javaBeanDS.show();
// +---+----+
// |age|name|
// +---+----+
// | 32|Andy|
// +---+----+

// Encoders for most common types are provided in class Encoders
Encoder<Integer> integerEncoder = Encoders.INT();
Dataset<Integer> primitiveDS = spark.createDataset(Arrays.asList(1, 2, 3),␣
↪integerEncoder);
Dataset<Integer> transformedDS = primitiveDS.map(
    (MapFunction<Integer, Integer>) value -> value + 1,
    integerEncoder);
transformedDS.collect(); // Returns [2, 3, 4]

// DataFrames can be converted to a Dataset by providing a class. Mapping based on␣
↪name
String path = "examples/src/main/resources/people.json";
Dataset<Person> peopleDS = spark.read().json(path).as(personEncoder);
peopleDS.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java” 文件。

与 RDD 互操作

Spark SQL supports two different methods for converting existing RDDs into Datasets. The first method uses reflection to infer the schema of an RDD that contains specific types of objects. This reflection based approach leads to more concise code and works well when you already know the schema while writing your Spark application.

The second method for creating Datasets is through a programmatic interface that allows you to construct a schema and then apply it to an existing RDD. While this method is more verbose, it allows you to construct Datasets when the columns and their types are not known until runtime.

Inferring the Schema Using Reflection

Scala

The Scala interface for Spark SQL supports automatically converting an RDD containing case classes to a DataFrame. The case class defines the schema of the table. The names of the arguments to the case class are read using reflection and become the names of the columns. Case classes can also be nested or contain complex types such as Seqs or Arrays. This RDD can be implicitly converted to a DataFrame and then be registered as a table. Tables can be used in subsequent SQL statements.

```
// For implicit conversions from RDDs to DataFrames
import spark.implicits._

// Create an RDD of Person objects from a text file, convert it to a Dataframe
val peopleDF = spark.sparkContext
  .textFile("examples/src/main/resources/people.txt")
  .map(_.split(","))
  .map(attributes => Person(attributes(0), attributes(1).trim.toInt))
  .toDF()
// Register the DataFrame as a temporary view
peopleDF.createOrReplaceTempView("people")

// SQL statements can be run by using the sql methods provided by Spark
val teenagersDF = spark.sql("SELECT name, age FROM people WHERE age BETWEEN 13 AND 19
↪")

// The columns of a row in the result can be accessed by field index
teenagersDF.map(teenager => "Name: " + teenager(0)).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+

// or by field name
teenagersDF.map(teenager => "Name: " + teenager.getAs[String]("name")).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+

// No pre-defined encoders for Dataset[Map[K,V]], define explicitly
implicit val mapEncoder = org.apache.spark.sql.Encoders.kryo[Map[String, Any]]
// Primitive types and case classes can be also defined as
// implicit val stringIntMapEncoder: Encoder[Map[String, Any]] = ExpressionEncoder()

// row.getValuesMap[T] retrieves multiple columns at once into a Map[String, T]
teenagersDF.map(teenager => teenager.getValuesMap[Any](List("name", "age"))).collect()
// Array(Map("name" -> "Justin", "age" -> 19))
```

完整的示例代码参见 Spark 源码仓库中的“examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala”文件。

Java

Spark SQL supports automatically converting an RDD of JavaBeans into a DataFrame. The BeanInfo, obtained using reflection, defines the schema of the table. Currently, Spark SQL does not support JavaBeans that contain Map field(s). Nested JavaBeans and List or Array fields are supported though. You can create a JavaBean by creating a class that implements Serializable and has getters and setters for all of its fields.

```
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.function.Function;
```

```
import org.apache.spark.api.java.function.MapFunction;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Encoder;
import org.apache.spark.sql.Encoders;

// Create an RDD of Person objects from a text file
JavaRDD<Person> peopleRDD = spark.read()
  .textFile("examples/src/main/resources/people.txt")
  .javaRDD()
  .map(line -> {
    String[] parts = line.split(",");
    Person person = new Person();
    person.setName(parts[0]);
    person.setAge(Integer.parseInt(parts[1].trim()));
    return person;
  });

// Apply a schema to an RDD of JavaBeans to get a DataFrame
Dataset<Row> peopleDF = spark.createDataFrame(peopleRDD, Person.class);
// Register the DataFrame as a temporary view
peopleDF.createOrReplaceTempView("people");

// SQL statements can be run by using the sql methods provided by spark
Dataset<Row> teenagersDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13␣
↪AND 19");

// The columns of a row in the result can be accessed by field index
Encoder<String> stringEncoder = Encoders.STRING();
Dataset<String> teenagerNamesByIndexDF = teenagersDF.map(
    (MapFunction<Row, String>) row -> "Name: " + row.getString(0),
    stringEncoder);
teenagerNamesByIndexDF.show();
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+

// or by field name
Dataset<String> teenagerNamesByFieldDF = teenagersDF.map(
    (MapFunction<Row, String>) row -> "Name: " + row.<String>getAs("name"),
    stringEncoder);
teenagerNamesByFieldDF.show();
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java” 文件。

Python

Spark SQL can convert an RDD of Row objects to a DataFrame, inferring the datatypes. Rows are constructed by passing a list of key/value pairs as kwargs to the Row class. The keys of this list define the column names of the table, and the types are inferred by sampling the whole dataset, similar to the inference that is performed on JSON files.

```
from pyspark.sql import Row

sc = spark.sparkContext

# Load a text file and convert each line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

# Infer the schema, and register the DataFrame as a table.
schemaPeople = spark.createDataFrame(people)
schemaPeople.createOrReplaceTempView("people")

# SQL can be run over DataFrames that have been registered as a table.
teenagers = spark.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

# The results of SQL queries are Dataframe objects.
# rdd returns the content as an :class:`pyspark.RDD` of :class:`Row`.
teenNames = teenagers.rdd.map(lambda p: "Name: " + p.name).collect()
for name in teenNames:
    print(name)
# Name: Justin
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/basic.py” 文件。

Programmatically Specifying the Schema

Scala

When case classes cannot be defined ahead of time (for example, the structure of records is encoded in a string, or a text dataset will be parsed and fields will be projected differently for different users), a DataFrame can be created programmatically with three steps.

Create an RDD of Rows from the original RDD; Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1. Apply the schema to the RDD of Rows via createDataFrame method provided by SparkSession. For example:

```
import org.apache.spark.sql.types._

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

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

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

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

// Apply the schema to the RDD
val peopleDF = spark.createDataFrame(rowRDD, schema)
```

```
// Creates a temporary view using the DataFrame
peopleDF.createOrReplaceTempView("people")

// SQL can be run over a temporary view created using DataFrames
val results = spark.sql("SELECT name FROM people")

// The results of SQL queries are DataFrames and support all the normal RDD operations
// The columns of a row in the result can be accessed by field index or by field name
results.map(attributes => "Name: " + attributes(0)).show()
// +-------------+
// |        value|
// +-------------+
// |Name: Michael|
// |   Name: Andy|
// | Name: Justin|
// +-------------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala” 文件。

Java

When JavaBean classes cannot be defined ahead of time (for example, the structure of records is encoded in a string, or a text dataset will be parsed and fields will be projected differently for different users), a Dataset<Row> can be created programmatically with three steps.

Create an RDD of Rows from the original RDD; Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1. Apply the schema to the RDD of Rows via createDataFrame method provided by SparkSession. For example:

```
import java.util.ArrayList;
import java.util.List;

import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.function.Function;

import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

import org.apache.spark.sql.types.DataTypes;
import org.apache.spark.sql.types.StructField;
import org.apache.spark.sql.types.StructType;

// Create an RDD
JavaRDD<String> peopleRDD = spark.sparkContext()
  .textFile("examples/src/main/resources/people.txt", 1)
  .toJavaRDD();

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

// Generate the schema based on the string of schema
List<StructField> fields = new ArrayList<>();
for (String fieldName : schemaString.split(" ")) {
  StructField field = DataTypes.createStructField(fieldName, DataTypes.StringType,␣
↪true);
  fields.add(field);
}
```

```
StructType schema = DataTypes.createStructType(fields);

// Convert records of the RDD (people) to Rows
JavaRDD<Row> rowRDD = peopleRDD.map((Function<String, Row>) record -> {
  String[] attributes = record.split(",");
  return RowFactory.create(attributes[0], attributes[1].trim());
});

// Apply the schema to the RDD
Dataset<Row> peopleDataFrame = spark.createDataFrame(rowRDD, schema);

// Creates a temporary view using the DataFrame
peopleDataFrame.createOrReplaceTempView("people");

// SQL can be run over a temporary view created using DataFrames
Dataset<Row> results = spark.sql("SELECT name FROM people");

// The results of SQL queries are DataFrames and support all the normal RDD operations
// The columns of a row in the result can be accessed by field index or by field name
Dataset<String> namesDS = results.map(
    (MapFunction<Row, String>) row -> "Name: " + row.getString(0),
    Encoders.STRING());
namesDS.show();
// +-------------+
// |        value|
// +-------------+
// |Name: Michael|
// |   Name: Andy|
// | Name: Justin|
// +-------------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java” 文件。

Python

When a dictionary of kwargs cannot be defined ahead of time (for example, the structure of records is encoded in a string, or a text dataset will be parsed and fields will be projected differently for different users), a DataFrame can be created programmatically with three steps.

Create an RDD of tuples or lists from the original RDD; Create the schema represented by a StructType matching the structure of tuples or lists in the RDD created in the step 1. Apply the schema to the RDD via createDataFrame method provided by SparkSession. For example:

```
# Import data types
from pyspark.sql.types import *

sc = spark.sparkContext

# Load a text file and convert each line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
# Each line is converted to a tuple.
people = parts.map(lambda p: (p[0], p[1].strip()))

# The schema is encoded in a string.
schemaString = "name age"

fields = [StructField(field_name, StringType(), True) for field_name in schemaString.
↪split()]
```

```
schema = StructType(fields)

# Apply the schema to the RDD.
schemaPeople = spark.createDataFrame(people, schema)

# Creates a temporary view using the DataFrame
schemaPeople.createOrReplaceTempView("people")

# SQL can be run over DataFrames that have been registered as a table.
results = spark.sql("SELECT name FROM people")

results.show()
# +-------+
# |   name|
# +-------+
# |Michael|
# |   Andy|
# | Justin|
# +-------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/basic.py” 文件。

聚合

The built-in DataFrames functions provide common aggregations such as count(), countDistinct(), avg(), max(), min(), etc. While those functions are designed for DataFrames, Spark SQL also has type-safe versions for some of them in Scala and Java to work with strongly typed Datasets. Moreover, users are not limited to the predefined aggregate functions and can create their own.

Untyped User-Defined Aggregate Functions

Users have to extend the UserDefinedAggregateFunction abstract class to implement a custom untyped aggregate function. For example, a user-defined average can look like:

Scala

```
import org.apache.spark.sql.expressions.MutableAggregationBuffer
import org.apache.spark.sql.expressions.UserDefinedAggregateFunction
import org.apache.spark.sql.types._
import org.apache.spark.sql.Row
import org.apache.spark.sql.SparkSession

object MyAverage extends UserDefinedAggregateFunction {
  // Data types of input arguments of this aggregate function
  def inputSchema: StructType = StructType(StructField("inputColumn", LongType) ::␣
↪Nil)
  // Data types of values in the aggregation buffer
  def bufferSchema: StructType = {
    StructType(StructField("sum", LongType) :: StructField("count", LongType) :: Nil)
  }
  // The data type of the returned value
  def dataType: DataType = DoubleType
  // Whether this function always returns the same output on the identical input
  def deterministic: Boolean = true
  // Initializes the given aggregation buffer. The buffer itself is a `Row` that in␣
↪addition to
```

```
  // standard methods like retrieving a value at an index (e.g., get(), getBoolean()),
↪ provides
  // the opportunity to update its values. Note that arrays and maps inside the␣
↪buffer are still
  // immutable.
  def initialize(buffer: MutableAggregationBuffer): Unit = {
    buffer(0) = 0L
    buffer(1) = 0L
  }
  // Updates the given aggregation buffer `buffer` with new input data from `input`
  def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
      buffer(0) = buffer.getLong(0) + input.getLong(0)
      buffer(1) = buffer.getLong(1) + 1
    }
  }
  // Merges two aggregation buffers and stores the updated buffer values back to␣
↪`buffer1`
  def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
    buffer1(0) = buffer1.getLong(0) + buffer2.getLong(0)
    buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1)
  }
  // Calculates the final result
  def evaluate(buffer: Row): Double = buffer.getLong(0).toDouble / buffer.getLong(1)
}

// Register the function to access it
spark.udf.register("myAverage", MyAverage)

val df = spark.read.json("examples/src/main/resources/employees.json")
df.createOrReplaceTempView("employees")
df.show()
// +-------+------+
// |   name|salary|
// +-------+------+
// |Michael|  3000|
// |   Andy|  4500|
// | Justin|  3500|
// |  Berta|  4000|
// +-------+------+

val result = spark.sql("SELECT myAverage(salary) as average_salary FROM employees")
result.show()
// +--------------+
// |average_salary|
// +--------------+
// |        3750.0|
// +--------------+
```

完整的示例代码参见 Spark 源码仓库中的“examples/src/main/scala/org/apache/spark/examples/sql/UserDefinedUntypedAggregation
文件。

Java

```
import java.util.ArrayList;
import java.util.List;

import org.apache.spark.sql.Dataset;
```

```
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.expressions.MutableAggregationBuffer;
import org.apache.spark.sql.expressions.UserDefinedAggregateFunction;
import org.apache.spark.sql.types.DataType;
import org.apache.spark.sql.types.DataTypes;
import org.apache.spark.sql.types.StructField;
import org.apache.spark.sql.types.StructType;

public static class MyAverage extends UserDefinedAggregateFunction {

  private StructType inputSchema;
  private StructType bufferSchema;

  public MyAverage() {
    List<StructField> inputFields = new ArrayList<>();
    inputFields.add(DataTypes.createStructField("inputColumn", DataTypes.LongType,␣
↪true));
    inputSchema = DataTypes.createStructType(inputFields);

    List<StructField> bufferFields = new ArrayList<>();
    bufferFields.add(DataTypes.createStructField("sum", DataTypes.LongType, true));
    bufferFields.add(DataTypes.createStructField("count", DataTypes.LongType, true));
    bufferSchema = DataTypes.createStructType(bufferFields);
  }
  // Data types of input arguments of this aggregate function
  public StructType inputSchema() {
    return inputSchema;
  }
  // Data types of values in the aggregation buffer
  public StructType bufferSchema() {
    return bufferSchema;
  }
  // The data type of the returned value
  public DataType dataType() {
    return DataTypes.DoubleType;
  }
  // Whether this function always returns the same output on the identical input
  public boolean deterministic() {
    return true;
  }
  // Initializes the given aggregation buffer. The buffer itself is a `Row` that in␣
↪addition to
  // standard methods like retrieving a value at an index (e.g., get(), getBoolean()),
↪ provides
  // the opportunity to update its values. Note that arrays and maps inside the␣
↪buffer are still
  // immutable.
  public void initialize(MutableAggregationBuffer buffer) {
    buffer.update(0, 0L);
    buffer.update(1, 0L);
  }
  // Updates the given aggregation buffer `buffer` with new input data from `input`
  public void update(MutableAggregationBuffer buffer, Row input) {
    if (!input.isNullAt(0)) {
      long updatedSum = buffer.getLong(0) + input.getLong(0);
      long updatedCount = buffer.getLong(1) + 1;
      buffer.update(0, updatedSum);
```

```
      buffer.update(1, updatedCount);
    }
  }
  // Merges two aggregation buffers and stores the updated buffer values back to␣
↪`buffer1`
  public void merge(MutableAggregationBuffer buffer1, Row buffer2) {
    long mergedSum = buffer1.getLong(0) + buffer2.getLong(0);
    long mergedCount = buffer1.getLong(1) + buffer2.getLong(1);
    buffer1.update(0, mergedSum);
    buffer1.update(1, mergedCount);
  }
  // Calculates the final result
  public Double evaluate(Row buffer) {
    return ((double) buffer.getLong(0)) / buffer.getLong(1);
  }
}

// Register the function to access it
spark.udf().register("myAverage", new MyAverage());

Dataset<Row> df = spark.read().json("examples/src/main/resources/employees.json");
df.createOrReplaceTempView("employees");
df.show();
// +-------+------+
// |   name|salary|
// +-------+------+
// |Michael|  3000|
// |   Andy|  4500|
// | Justin|  3500|
// |  Berta|  4000|
// +-------+------+

Dataset<Row> result = spark.sql("SELECT myAverage(salary) as average_salary FROM␣
↪employees");
result.show();
// +--------------+
// |average_salary|
// +--------------+
// |        3750.0|
// +--------------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaUserDefinedUntypedAggrega
文件。

Type-Safe User-Defined Aggregate Functions

User-defined aggregations for strongly typed Datasets revolve around the Aggregator abstract class. For example, a type-safe user-defined average can look like:

Scala

```
import org.apache.spark.sql.expressions.Aggregator
import org.apache.spark.sql.Encoder
import org.apache.spark.sql.Encoders
import org.apache.spark.sql.SparkSession

case class Employee(name: String, salary: Long)
```

```scala
case class Average(var sum: Long, var count: Long)

object MyAverage extends Aggregator[Employee, Average, Double] {
  // A zero value for this aggregation. Should satisfy the property that any b + zero
  // = b
  def zero: Average = Average(0L, 0L)
  // Combine two values to produce a new value. For performance, the function may
  // modify `buffer`
  // and return it instead of constructing a new object
  def reduce(buffer: Average, employee: Employee): Average = {
    buffer.sum += employee.salary
    buffer.count += 1
    buffer
  }
  // Merge two intermediate values
  def merge(b1: Average, b2: Average): Average = {
    b1.sum += b2.sum
    b1.count += b2.count
    b1
  }
  // Transform the output of the reduction
  def finish(reduction: Average): Double = reduction.sum.toDouble / reduction.count
  // Specifies the Encoder for the intermediate value type
  def bufferEncoder: Encoder[Average] = Encoders.product
  // Specifies the Encoder for the final output value type
  def outputEncoder: Encoder[Double] = Encoders.scalaDouble
}

val ds = spark.read.json("examples/src/main/resources/employees.json").as[Employee]
ds.show()
// +-------+------+
// |   name|salary|
// +-------+------+
// |Michael|  3000|
// |   Andy|  4500|
// | Justin|  3500|
// |  Berta|  4000|
// +-------+------+

// Convert the function to a `TypedColumn` and give it a name
val averageSalary = MyAverage.toColumn.name("average_salary")
val result = ds.select(averageSalary)
result.show()
// +--------------+
// |average_salary|
// +--------------+
// |        3750.0|
// +--------------+
```

完整的示例代码参见 Spark 源码仓库中的“examples/src/main/scala/org/apache/spark/examples/sql/UserDefinedTypedAggregation.s
文件。

Java

```java
import java.io.Serializable;

import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Encoder;
```

```java
import org.apache.spark.sql.Encoders;
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.TypedColumn;
import org.apache.spark.sql.expressions.Aggregator;

public static class Employee implements Serializable {
  private String name;
  private long salary;

  // Constructors, getters, setters...

}

public static class Average implements Serializable  {
  private long sum;
  private long count;

  // Constructors, getters, setters...

}

public static class MyAverage extends Aggregator<Employee, Average, Double> {
  // A zero value for this aggregation. Should satisfy the property that any b + zero
↪= b
  public Average zero() {
    return new Average(0L, 0L);
  }
  // Combine two values to produce a new value. For performance, the function may
↪modify `buffer`
  // and return it instead of constructing a new object
  public Average reduce(Average buffer, Employee employee) {
    long newSum = buffer.getSum() + employee.getSalary();
    long newCount = buffer.getCount() + 1;
    buffer.setSum(newSum);
    buffer.setCount(newCount);
    return buffer;
  }
  // Merge two intermediate values
  public Average merge(Average b1, Average b2) {
    long mergedSum = b1.getSum() + b2.getSum();
    long mergedCount = b1.getCount() + b2.getCount();
    b1.setSum(mergedSum);
    b1.setCount(mergedCount);
    return b1;
  }
  // Transform the output of the reduction
  public Double finish(Average reduction) {
    return ((double) reduction.getSum()) / reduction.getCount();
  }
  // Specifies the Encoder for the intermediate value type
  public Encoder<Average> bufferEncoder() {
    return Encoders.bean(Average.class);
  }
  // Specifies the Encoder for the final output value type
  public Encoder<Double> outputEncoder() {
    return Encoders.DOUBLE();
  }
}
```

```
Encoder<Employee> employeeEncoder = Encoders.bean(Employee.class);
String path = "examples/src/main/resources/employees.json";
Dataset<Employee> ds = spark.read().json(path).as(employeeEncoder);
ds.show();
// +-------+------+
// |   name|salary|
// +-------+------+
// |Michael|  3000|
// |   Andy|  4500|
// | Justin|  3500|
// |  Berta|  4000|
// +-------+------+

MyAverage myAverage = new MyAverage();
// Convert the function to a `TypedColumn` and give it a name
TypedColumn<Employee, Double> averageSalary = myAverage.toColumn().name("average_
↪salary");
Dataset<Double> result = ds.select(averageSalary);
result.show();
// +--------------+
// |average_salary|
// +--------------+
// |        3750.0|
// +--------------+
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaUserDefinedTypedAggregatio
文件。

1.3.3 数据源

Spark SQL supports operating on a variety of data sources through the DataFrame interface. A DataFrame can be operated on using relational transformations and can also be used to create a temporary view. Registering a DataFrame as a temporary view allows you to run SQL queries over its data. This section describes the general methods for loading and saving data using the Spark Data Sources and then goes into specific options that are available for the built-in data sources.

Generic Load/Save Functions

In the simplest form, the default data source (parquet unless otherwise configured by spark.sql.sources.default) will be used for all operations.

Scala

```
val usersDF = spark.read.load("examples/src/main/resources/users.parquet")
usersDF.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala”
文件。

Java

```
Dataset<Row> usersDF = spark.read().load("examples/src/main/resources/users.parquet");
usersDF.select("name", "favorite_color").write().save("namesAndFavColors.parquet");
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSQLDataSourceExample.java 文件。

Python

```
df = spark.read.load("examples/src/main/resources/users.parquet")
df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/datasource.py” 文件。

R

```
df <- read.df("examples/src/main/resources/users.parquet")
write.df(select(df, "name", "favorite_color"), "namesAndFavColors.parquet")
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/r/RSparkSQLExample.R” 文件。

手动指定选项

You can also manually specify the data source that will be used along with any extra options that you would like to pass to the data source. Data sources are specified by their fully qualified name (i.e., org.apache.spark.sql.parquet), but for built-in sources you can also use their short names (json, parquet, jdbc, orc, libsvm, csv, text). DataFrames loaded from any data source type can be converted into other types using this syntax.

Scala

```
val peopleDF = spark.read.format("json").load("examples/src/main/resources/people.json
↪")
peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala” 文件。

Java

```
Dataset<Row> peopleDF =
  spark.read().format("json").load("examples/src/main/resources/people.json");
peopleDF.select("name", "age").write().format("parquet").save("namesAndAges.parquet");
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSQLDataSourceExample.java 文件。

Python

```
df = spark.read.load("examples/src/main/resources/people.json", format="json")
df.select("name", "age").write.save("namesAndAges.parquet", format="parquet")
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/datasource.py” 文件。

R

```
df <- read.df("examples/src/main/resources/people.json", "json")
namesAndAges <- select(df, "name", "age")
write.df(namesAndAges, "namesAndAges.parquet", "parquet")
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/r/RSparkSQLExample.R” 文件。

直接在文件上运行 SQL

Instead of using read API to load a file into DataFrame and query it, you can also query that file directly with SQL.

Scala

```
val sqlDF = spark.sql("SELECT * FROM parquet.`examples/src/main/resources/users.
↪parquet`")
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala” 文件。

Java

```
Dataset<Row> sqlDF =
  spark.sql("SELECT * FROM parquet.`examples/src/main/resources/users.parquet`");
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSQLDataSourceExample.java” 文件。

Python

```
df = spark.sql("SELECT * FROM parquet.`examples/src/main/resources/users.parquet`")
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/datasource.py” 文件。

R

```
df <- sql("SELECT * FROM parquet.`examples/src/main/resources/users.parquet`")
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/r/RSparkSQLExample.R” 文件。

Save Modes

Save operations can optionally take a SaveMode, that specifies how to handle existing data if present. It is important to realize that these save modes do not utilize any locking and are not atomic. Additionally, when performing an Overwrite, the data will be deleted before writing out the new data.

Scala/Java	Any Language	Meaning
SaveMode.ErrorIfExists (default)	“error” (default)	When saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown.
SaveMode.Append	“append”	When saving a DataFrame to a data source, if data/table already exists, contents of the DataFrame are expected to be appended to existing data.
SaveMode.Overwrite	“overwrite”	Overwrite mode means that when saving a DataFrame to a data source, if data/table already exists, existing data is expected to be overwritten by the contents of the DataFrame.
SaveMode.Ignore	“ignore”	Ignore mode means that when saving a DataFrame to a data source, if data already exists, the save operation is expected to not save the contents of the DataFrame and to not change the existing data. This is similar to a CREATE TABLE IF NOT EXISTS in SQL.

Saving to Persistent Tables

DataFrames can also be saved as persistent tables into Hive metastore using the saveAsTable command. Notice that an existing Hive deployment is not necessary to use this feature. Spark will create a default local Hive metastore (using Derby) for you. Unlike the createOrReplaceTempView command, saveAsTable will materialize the contents of the DataFrame and create a pointer to the data in the Hive metastore. Persistent tables will still exist even after your Spark program has restarted, as long as you maintain your connection to the same metastore. A DataFrame for a persistent table can be created by calling the table method on a SparkSession with the name of the table.

For file-based data source, e.g. text, parquet, json, etc. you can specify a custom table path via the path option, e.g. df.write.option("path", "/some/path").saveAsTable("t"). When the table is dropped, the custom table path will not be removed and the table data is still there. If no custom table path is specified, Spark will write data to a default table path under the warehouse directory. When the table is dropped, the default table path will be removed too.

Starting from Spark 2.1, persistent datasource tables have per-partition metadata stored in the Hive metastore. This brings several benefits:

Since the metastore can return only necessary partitions for a query, discovering all the partitions on the first query to the table is no longer needed. Hive DDLs such as ALTER TABLE PARTITION ... SET LOCATION are now available for tables created with the Datasource API. Note that partition information is not gathered by default when creating external datasource tables (those with a path option). To sync the partition information in the metastore, you can invoke MSCK REPAIR TABLE.

Bucketing, Sorting and Partitioning

For file-based data source, it is also possible to bucket and sort or partition the output. Bucketing and sorting are applicable only to persistent tables:

Scala

```
peopleDF.write.bucketBy(42, "name").sortBy("age").saveAsTable("people_bucketed")
```

完整的示例代码参见 Spark 源码仓库中的 "examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala" 文件。

while partitioning can be used with both save and saveAsTable when using the Dataset APIs.

```
usersDF.write.partitionBy("favorite_color").format("parquet").save("namesPartByColor.
↪parquet")
```

完整的示例代码参见 Spark 源码仓库中的 "examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala" 文件。 It is possible to use both partitioning and bucketing for a single table:

```
peopleDF
  .write
  .partitionBy("favorite_color")
  .bucketBy(42, "name")
  .saveAsTable("people_partitioned_bucketed")
```

完整的示例代码参见 Spark 源码仓库中的 "examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala" 文件。 partitionBy creates a directory structure as described in the Partition Discovery section. Thus, it has limited applicability to columns with high cardinality. In contrast bucketBy distributes data across a fixed number of buckets and can be used when a number of unique values is unbounded.

Java

```
peopleDF.write().bucketBy(42, "name").sortBy("age").saveAsTable("people_bucketed");
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSQLDataSourceExample.java” 文件。 while partitioning can be used with both save and saveAsTable when using the Dataset APIs.

```
usersDF
  .write()
  .partitionBy("favorite_color")
  .format("parquet")
  .save("namesPartByColor.parquet");
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSQLDataSourceExample.java” 文件。 It is possible to use both partitioning and bucketing for a single table:

```
peopleDF
  .write()
  .partitionBy("favorite_color")
  .bucketBy(42, "name")
  .saveAsTable("people_partitioned_bucketed");
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSQLDataSourceExample.java” 文件。 partitionBy creates a directory structure as described in the Partition Discovery section. Thus, it has limited applicability to columns with high cardinality. In contrast bucketBy distributes data across a fixed number of buckets and can be used when a number of unique values is unbounded.

Python

```
df.write.bucketBy(42, "name").sortBy("age").saveAsTable("people_bucketed")
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/datasource.py” 文件。

while partitioning can be used with both save and saveAsTable when using the Dataset APIs.

```
df.write.partitionBy("favorite_color").format("parquet").save("namesPartByColor.
↪parquet")
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/datasource.py” 文件。

It is possible to use both partitioning and bucketing for a single table:

```
df = spark.read.parquet("examples/src/main/resources/users.parquet")
(df
    .write
    .partitionBy("favorite_color")
    .bucketBy(42, "name")
    .saveAsTable("people_partitioned_bucketed"))
```

完整的示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/datasource.py” 文件。 partitionBy creates a directory structure as described in the Partition Discovery section. Thus, it has limited applicability to columns with high cardinality. In contrast bucketBy distributes data across a fixed number of buckets and can be used when a number of unique values is unbounded.

Sql

```
CREATE TABLE users_bucketed_by_name(
  name STRING,
  favorite_color STRING,
  favorite_numbers array<integer>
```

```
) USING parquet
CLUSTERED BY(name) INTO 42 BUCKETS;
```

while partitioning can be used with both save and saveAsTable when using the Dataset APIs.

```
CREATE TABLE users_by_favorite_color(
  name STRING,
  favorite_color STRING,
  favorite_numbers array<integer>
) USING csv PARTITIONED BY(favorite_color);
```

It is possible to use both partitioning and bucketing for a single table:

```
CREATE TABLE users_bucketed_and_partitioned(
  name STRING,
  favorite_color STRING,
  favorite_numbers array<integer>
) USING parquet
PARTITIONED BY (favorite_color)
CLUSTERED BY(name) SORTED BY (favorite_numbers) INTO 42 BUCKETS;
```

partitionBy creates a directory structure as described in the Partition Discovery section. Thus, it has limited applicability to columns with high cardinality. In contrast bucketBy distributes data across a fixed number of buckets and can be used when a number of unique values is unbounded.

Parquet 文件

Parquet 是一种列式存储格式，很多其它的数据处理系统都支持它。Spark SQL 提供了对 Parquet 文件的读写支持，而且 Parquet 文件能够自动保存原始数据的 schema。写 Parquet 文件的时候，所有列都自动地转化成 nullable，以便向后兼容。

编程方式加载数据

仍然使用上面例子中的数据：

Scala

```
// Encoders for most common types are automatically provided by importing spark.
↪implicits._
import spark.implicits._

val peopleDF = spark.read.json("examples/src/main/resources/people.json")

// DataFrames can be saved as Parquet files, maintaining the schema information
peopleDF.write.parquet("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 DataFrame
val parquetFileDF = spark.read.parquet("people.parquet")

// Parquet files can also be used to create a temporary view and then used in SQL␣
↪statements
parquetFileDF.createOrReplaceTempView("parquetFile")
val namesDF = spark.sql("SELECT name FROM parquetFile WHERE age BETWEEN 13 AND 19")
```

```
namesDF.map(attributes => "Name: " + attributes(0)).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala” 文件。

Java

```
import org.apache.spark.api.java.function.MapFunction;
import org.apache.spark.sql.Encoders;
// import org.apache.spark.sql.Encoders;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

Dataset<Row> peopleDF = spark.read().json("examples/src/main/resources/people.json");

// DataFrames can be saved as Parquet files, maintaining the schema information
peopleDF.write().parquet("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 DataFrame
Dataset<Row> parquetFileDF = spark.read().parquet("people.parquet");

// Parquet files can also be used to create a temporary view and then used in SQL␣
↪statements
parquetFileDF.createOrReplaceTempView("parquetFile");
Dataset<Row> namesDF = spark.sql("SELECT name FROM parquetFile WHERE age BETWEEN 13␣
↪AND 19");
Dataset<String> namesDS = namesDF.map(new MapFunction<Row, String>() {
  public String call(Row row) {
    return "Name: " + row.getString(0);
  }
}, Encoders.STRING());
namesDS.show();
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSQLDataSourceExample.java” 文件。

Python

```
peopleDF = spark.read.json("examples/src/main/resources/people.json")

# DataFrames can be saved as Parquet files, maintaining the schema information.
peopleDF.write.parquet("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 DataFrame.
```

```
parquetFile = spark.read.parquet("people.parquet")

# Parquet files can also be used to create a temporary view and then used in SQL␣
↪statements.
parquetFile.createOrReplaceTempView("parquetFile")
teenagers = spark.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenagers.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/datasource.py” 文件。

R

```
df <- read.df("examples/src/main/resources/people.json", "json")

# SparkDataFrame can be saved as Parquet files, maintaining the schema information.
write.parquet(df, "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 DataFrame.
parquetFile <- read.parquet("people.parquet")

# Parquet files can also be used to create a temporary view and then used in SQL␣
↪statements.
createOrReplaceTempView(parquetFile, "parquetFile")
teenagers <- sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
head(teenagers)
##     name
## 1 Justin

# We can also run custom R-UDFs on Spark DataFrames. Here we prefix all the names␣
↪with "Name:"
schema <- structType(structField("name", "string"))
teenNames <- dapply(df, function(p) { cbind(paste("Name:", p$name)) }, schema)
for (teenName in collect(teenNames)$name) {
  cat(teenName, "\n")
}
## Name: Michael
## Name: Andy
## Name: Justin
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/r/RSparkSQLExample.R” 文件。

Sql

```
CREATE TEMPORARY VIEW parquetTable
USING org.apache.spark.sql.parquet
OPTIONS (
  path "examples/src/main/resources/people.parquet"
)

SELECT * FROM parquetTable
```

分区发现

像Hive这样的系统中，一个常用的优化方式就是表分区。在一个分区表中，数据通常存储在不同的目录中，分区列值被编码到各个分区目录的路径。Parquet数据源现在可以自动发现和推导分区信息。例如，我们可以使用下面的目录结构把之前使用的人口数据存储到一个分区表中，其中2个额外的字段，gender和country，作为分区列：

```
path
└── to
    └── table
        ├── gender=male
        │   ├── ...
        │   │
        │   ├── country=US
        │   │   └── data.parquet
        │   ├── country=CN
        │   │   └── data.parquet
        │   └── ...
        └── gender=female
            ├── ...
            │
            ├── country=US
            │   └── data.parquet
            ├── country=CN
            │   └── data.parquet
            └── ...
```

通过传递 path/to/table 给 SparkSession.read.parquet 或 SparkSession.read.load, Spark SQL将会自动从路径中提取分区信息。现在返回的DataFrame的schema如下：

```
root
|-- name: string (nullable = true)
|-- age: long (nullable = true)
|-- gender: string (nullable = true)
|-- country: string (nullable = true)
```

注意，分区列的数据类型是自动推导出来的。目前，分区列只支持数值类型和字符串类型。有时候用户可能不想要自动推导分区列的数据类型，对于这种情况，自动类型推导可以通过spark.sql.sources.partitionColumnTypeInference.enabled来配置，其默认值是true。当禁用类型推导后，字符串类型将用于分区列类型。

从Spark 1.6.0 版本开始，分区发现默认只查找给定路径下的分区。拿上面的例子来说，如果用户传递 path/to/table/gender=male 给 SparkSession.read.parquet 或者 SparkSession.read.load，那么gender将不会被当作分区列。如果用户想要指定分区发现开始的基础目录，可以在数据源选项中设置basePath。例如，如果把path/to/table/gender=male作为数据目录，并且将basePath设为 path/to/table，那么gender仍然会最为分区键。

Schema 合并

和 ProtocolBuffer、Avro 以及 Thrift 一样，Parquet也支持 schema 演变。用户可以从一个简单的 schema 开始，逐渐增加所需要的列。这样的话，用户最终会得到多个Parquet文件, 这些文件的schema不同但互相兼容。Parquet数据源目前已经支持自动检测这种情况并合并所有这些文件的schema。

因为schema合并相对来说是一个代价高昂的操作，并且在大多数情况下不需要，所以从Spark 1.5.0 版本开始，默认禁用Schema合并。你可以这样启用这一功能：

1. 当读取Parquet文件时，将数据源选项 mergeSchema设置为true（见下面的示例代码）

2. 或者，将全局SQL选项 spark.sql.parquet.mergeSchema设置为true。

Scala

```
// This is used to implicitly convert an RDD to a DataFrame.
import spark.implicits._

// Create a simple DataFrame, store into a partition directory
val squaresDF = spark.sparkContext.makeRDD(1 to 5).map(i => (i, i * i)).toDF("value",
↪"square")
squaresDF.write.parquet("data/test_table/key=1")

// Create another DataFrame in a new partition directory,
// adding a new column and dropping an existing column
val cubesDF = spark.sparkContext.makeRDD(6 to 10).map(i => (i, i * i * i)).toDF("value
↪", "cube")
cubesDF.write.parquet("data/test_table/key=2")

// Read the partitioned table
val mergedDF = spark.read.option("mergeSchema", "true").parquet("data/test_table")
mergedDF.printSchema()

// The final schema consists of all 3 columns in the Parquet files together
// with the partitioning column appeared in the partition directory paths
// root
// |-- value: int (nullable = true)
// |-- square: int (nullable = true)
// |-- cube: int (nullable = true)
// |-- key : int (nullable = true)
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala” 文件。

Java

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

public static class Square implements Serializable {
  private int value;
  private int square;

  // Getters and setters...

}

public static class Cube implements Serializable {
  private int value;
  private int cube;

  // Getters and setters...

}
```

```
List<Square> squares = new ArrayList<>();
for (int value = 1; value <= 5; value++) {
  Square square = new Square();
  square.setValue(value);
  square.setSquare(value * value);
  squares.add(square);
}

// Create a simple DataFrame, store into a partition directory
Dataset<Row> squaresDF = spark.createDataFrame(squares, Square.class);
squaresDF.write().parquet("data/test_table/key=1");

List<Cube> cubes = new ArrayList<>();
for (int value = 6; value <= 10; value++) {
  Cube cube = new Cube();
  cube.setValue(value);
  cube.setCube(value * value * value);
  cubes.add(cube);
}

// Create another DataFrame in a new partition directory,
// adding a new column and dropping an existing column
Dataset<Row> cubesDF = spark.createDataFrame(cubes, Cube.class);
cubesDF.write().parquet("data/test_table/key=2");

// Read the partitioned table
Dataset<Row> mergedDF = spark.read().option("mergeSchema", true).parquet("data/test_
↪table");
mergedDF.printSchema();

// The final schema consists of all 3 columns in the Parquet files together
// with the partitioning column appeared in the partition directory paths
// root
//  |-- value: int (nullable = true)
//  |-- square: int (nullable = true)
//  |-- cube: int (nullable = true)
//  |-- key: int (nullable = true)
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSQLDataSourceExample.java” 文件。

Python

```
from pyspark.sql import Row

# spark is from the previous example.
# Create a simple DataFrame, stored into a partition directory
sc = spark.sparkContext

squaresDF = spark.createDataFrame(sc.parallelize(range(1, 6))
                                  .map(lambda i: Row(single=i, double=i ** 2)))
squaresDF.write.parquet("data/test_table/key=1")

# Create another DataFrame in a new partition directory,
# adding a new column and dropping an existing column
cubesDF = spark.createDataFrame(sc.parallelize(range(6, 11))
                                .map(lambda i: Row(single=i, triple=i ** 3)))
cubesDF.write.parquet("data/test_table/key=2")
```

```
# Read the partitioned table
mergedDF = spark.read.option("mergeSchema", "true").parquet("data/test_table")
mergedDF.printSchema()

# The final schema consists of all 3 columns in the Parquet files together
# with the partitioning column appeared in the partition directory paths.
# root
#  |-- double: long (nullable = true)
#  |-- single: long (nullable = true)
#  |-- triple: long (nullable = true)
#  |-- key: integer (nullable = true)
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/datasource.py” 文件。

R

```
df1 <- createDataFrame(data.frame(single=c(12, 29), double=c(19, 23)))
df2 <- createDataFrame(data.frame(double=c(19, 23), triple=c(23, 18)))

# Create a simple DataFrame, stored into a partition directory
write.df(df1, "data/test_table/key=1", "parquet", "overwrite")

# Create another DataFrame in a new partition directory,
# adding a new column and dropping an existing column
write.df(df2, "data/test_table/key=2", "parquet", "overwrite")

# Read the partitioned table
df3 <- read.df("data/test_table", "parquet", mergeSchema = "true")
printSchema(df3)
# The final schema consists of all 3 columns in the Parquet files together
# with the partitioning column appeared in the partition directory paths
## root
##  |-- single: double (nullable = true)
##  |-- double: double (nullable = true)
##  |-- triple: double (nullable = true)
##  |-- key: integer (nullable = true)
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/r/RSparkSQLExample.R” 文件。

Hive metastore Parquet表转换

当读写Hive metastore Parquet表时，为了达到更好的性能, Spark SQL使用它自己的Parquet支持库，而不是Hive SerDe。这一行为是由 spark.sql.hive.convertMetastoreParquet 这个配置项来控制的，它默认是开启的。

Hive/Parquet Schema调整

从表 schema 处理的角度来看, Hive和Parquet有2个关键的不同点：

1. Hive是非大小写敏感的，而Parquet是大小写敏感的。
2. Hive认为所有列都是nullable，而Parquet中为空性是很重要的。

基于以上原因，在将一个Hive metastore Parquet表转换成一个Spark SQL Parquet表的时候，必须要对Hive metastore schema做调整，调整规则如下：

1. 两个schema中字段名称一致的话那么字段类型也必须一致（不考虑为空性）。调整后的字段应该有Parquet端的数据类型，所以为空性也是需要考虑的。
2. **调整后的schema必须完全包含Hive metastore schema中定义的字段。**
    - 只出现在Parquet schema中的字段将在调整后的schema中丢弃。
    - 只出现在Hive metastore schema中的字段将作为nullable字段添加到调整后的schema。

元数据刷新

Spark SQL 会缓存 Parquet 元数据以提高性能。如果启用了Hive metastore Parquet table转换，那么转换后的表的schema也会被缓存起来。如果这些表被Hive或其它外部工具更新, 那么你需要手动地刷新它们以确保元数据一致性。

Scala

```
// spark is an existing SparkSession
spark.catalog.refreshTable("my_table")
```

Java

```
// spark is an existing SparkSession
spark.catalog().refreshTable("my_table");
```

Python

```
# spark is an existing HiveContext
spark.refreshTable("my_table")
```

Sql

```
REFRESH TABLE my_table;
```

配置

Parquet配置可以使用 SparkSession 上的 setConf 方法或者使用 SQL 语句中的 SET key=value 命令来完成。

属性名	默认值	含义
spark.sql.parquet.binaryAsString	false	一些其它的Parquet生产系统, 特别是Impala，Hive以及老版本的Spark SQL，当写Parquet schema时都不区分二进制数据和字符串。这个标识告诉Spark SQL把二进制数据当字符串处理，以兼容老系统。
spark.sql.parquet.int96AsTimestamp	true	一些Parquet生产系统, 特别是Impala和Hive，把时间戳存成INT96。这个标识告诉Spark SQL将INT96数据解析成timestamp，以兼容老系统。
spark.sql.parquet.cacheMetadata	true	开启Parquet schema元数据缓存。可以提升查询静态数据的速度。
spark.sql.parquet.compression.codec	gzip	当写Parquet文件时，设置压缩编码格式。可接受的值有：uncompressed, snappy, gzip, lzo
spark.sql.parquet.filterPushdown	true	当设置为true时启用Parquet过滤器下推优化
spark.sql.hive.convertMetastoreParquet	true	当设置为false时，Spark SQL将使用Hive SerDe，而不是内建的Parquet tables支持
spark.sql.parquet.mergeSchema	false	如果设为true，那么Parquet数据源将会合并所有数据文件的schema，否则，从汇总文件中选取schema，如果没有汇总文件，则随机选取一个数据文件）

JSON Datasets

Scala

Spark SQL可以自动推导JSON数据集的schema并且将其加载为一个 Dataset[Row]。这种转换可以在一个包含String的RDD或一个JSON文件上使用SparkSession.read.json() 来完成。

注意，作为json文件提供的文件并不是一个典型的JSON文件。JSON文件的每一行必须包含一个独立的、完整有效的JSON对象。因此，一个常规的多行json文件经常会加载失败。

```
// Primitive types (Int, String, etc) and Product types (case classes) encoders are
// supported by importing this when creating a Dataset.
import spark.implicits._

// 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"
val peopleDF = spark.read.json(path)

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

// Creates a temporary view using the DataFrame
peopleDF.createOrReplaceTempView("people")

// SQL statements can be run by using the sql methods provided by spark
val teenagerNamesDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19")
teenagerNamesDF.show()
// +------+
// |  name|
// +------+
// |Justin|
// +------+

// Alternatively, a DataFrame can be created for a JSON dataset represented by
// a Dataset[String] storing one JSON object per string
val otherPeopleDataset = spark.createDataset(
  """{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: Nil)
val otherPeople = spark.read.json(otherPeopleDataset)
otherPeople.show()
// +---------------+----+
// |        address|name|
// +---------------+----+
// |[Columbus,Ohio]| Yin|
// +---------------+----+
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala” 文件。

Java

Spark SQL 可以自动推导 JSON 数据集的 schema 并且将其加载为一个 Dataset<Row>. 这种转换可以在一个包含String的RDD或一个JSON文件上使用SparkSession.read.json() 来完成。

注意，作为json文件提供的文件并不是一个典型的JSON文件。JSON文件的每一行必须包含一个独立的、完整有效的JSON对象。因此，一个常规的多行json文件经常会加载失败。

```
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

// A JSON dataset is pointed to by path.
// The path can be either a single text file or a directory storing text files
Dataset<Row> people = spark.read().json("examples/src/main/resources/people.json");

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

// Creates a temporary view using the DataFrame
people.createOrReplaceTempView("people");

// SQL statements can be run by using the sql methods provided by spark
Dataset<Row> namesDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19
↪");
namesDF.show();
// +------+
// |  name|
// +------+
// |Justin|
// +------+

// Alternatively, a DataFrame can be created for a JSON dataset represented by
// a Dataset<String> storing one JSON object per string.
List<String> jsonData = Arrays.asList(
        "{\"name\":\"Yin\",\"address\":{\"city\":\"Columbus\",\"state\":\"Ohio\"}}");
Dataset<String> anotherPeopleDataset = spark.createDataset(jsonData, Encoders.
↪STRING());
Dataset<Row> anotherPeople = spark.read().json(anotherPeopleDataset);
anotherPeople.show();
// +---------------+----+
// |        address|name|
// +---------------+----+
// |[Columbus,Ohio]| Yin|
// +---------------+----+
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSQLDataSourceExample.java” 文件。

Python

Spark SQL可以自动推导JSON数据集的schema并且将其加载为一个 DataFrame。这种转换可以在一个JSON文件上使用SparkSession.read.json 来完成。

注意，作为 json 文件提供的文件并不是一个典型的 JSON 文件。JSON 文件的每一行必须包含一个独立的、完整有效的JSON对象。因此，一个常规的多行json文件经常会加载失败。

```
# spark is from the previous example.
sc = spark.sparkContext

# A JSON dataset is pointed to by path.
# The path can be either a single text file or a directory storing text files
path = "examples/src/main/resources/people.json"
peopleDF = spark.read.json(path)
```

```
# The inferred schema can be visualized using the printSchema() method
peopleDF.printSchema()
# root
#  |-- age: long (nullable = true)
#  |-- name: string (nullable = true)

# Creates a temporary view using the DataFrame
peopleDF.createOrReplaceTempView("people")

# SQL statements can be run by using the sql methods provided by spark
teenagerNamesDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19")
teenagerNamesDF.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+

# Alternatively, a DataFrame can be created for a JSON dataset represented by
# an RDD[String] storing one JSON object per string
jsonStrings = ['{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}']
otherPeopleRDD = sc.parallelize(jsonStrings)
otherPeople = spark.read.json(otherPeopleRDD)
otherPeople.show()
# +---------------+----+
# |        address|name|
# +---------------+----+
# |[Columbus,Ohio]| Yin|
# +---------------+----+
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/datasource.py” 文件。

R

Spark SQL can automatically infer the schema of a JSON dataset and load it as a DataFrame. using the read.json() function, which loads data from a directory of JSON files where each line of the files is a JSON object.

Note that the file that is offered as a json file is not a typical JSON file. Each line must contain a separate, self-contained valid JSON object. For more information, please see JSON Lines text format, also called newline-delimited JSON.

For a regular multi-line JSON file, set a named parameter multiLine to TRUE.

```
# A JSON dataset is pointed to by path.
# The path can be either a single text file or a directory storing text files.
path <- "examples/src/main/resources/people.json"
# Create a DataFrame from the file(s) pointed to by path
people <- read.json(path)

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

# Register this DataFrame as a table.
createOrReplaceTempView(people, "people")

# SQL statements can be run by using the sql methods.
teenagers <- sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")
```

```
head(teenagers)
##     name
## 1 Justin
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/r/RSparkSQLExample.R” 文件。

Sql

```
CREATE TEMPORARY VIEW jsonTable
USING org.apache.spark.sql.json
OPTIONS (
  path "examples/src/main/resources/people.json"
)

SELECT * FROM jsonTable
```

Hive Tables

Spark SQL 还支持从 Apache Hive 读写数据。然而，由于 Hive 依赖项太多，这些依赖没有包含在默认的 Spark 发行版本中。如果在classpath上配置了Hive依赖，那么 Spark 会自动加载它们。注意，Hive 依赖也必须放到所有的 worker 节点上，因为如果要访问 Hive 中的数据它们需要访问 Hive 序列化和反序列化库(SerDes)。

Hive配置是通过将 hive-site.xml，core-site.xml(用于安全配置)以及 hdfs-site.xml(用于 HDFS 配置)文件放置在 conf/ 目录下来完成的。

如果要使用 Hive, 你必须要实例化一个支持 Hive 的 SparkSession, 包括连接到一个持久化的 Hive metastore, 支持 Hive serdes 以及 Hive 用户自定义函数。即使用户没有安装部署 Hive 也仍然可以启用Hive支持。如果没有在 hive-site.xml 文件中配置, Spark 应用程序启动之后，上下文会自动在当前目录下创建一个 metastore_db 目录并创建一个由 spark.sql.warehouse.dir 配置的、默认值是当前目录下的 spark-warehouse 目录的目录。请注意: 从 Spark 2.0.0 版本开始, hive-site.xml 中的 hive.metastore.warehouse.dir 属性就已经过时了, 你可以使用 spark.sql.warehouse.dir 来指定仓库中数据库的默认存储位置。你可能还需要给启动Spark应用程序的用户赋予写权限。

Scala

```
import java.io.File

import org.apache.spark.sql.Row
import org.apache.spark.sql.SparkSession

case class Record(key: Int, value: String)

// warehouseLocation points to the default location for managed databases and tables
val warehouseLocation = new File("spark-warehouse").getAbsolutePath

val spark = SparkSession
  .builder()
  .appName("Spark Hive Example")
  .config("spark.sql.warehouse.dir", warehouseLocation)
  .enableHiveSupport()
  .getOrCreate()

import spark.implicits._
import spark.sql

sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")
```

```
sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

// Queries are expressed in HiveQL
sql("SELECT * FROM src").show()
// +---+-------+
// |key|  value|
// +---+-------+
// |238|val_238|
// | 86| val_86|
// |311|val_311|
// ...

// Aggregation queries are also supported.
sql("SELECT COUNT(*) FROM src").show()
// +--------+
// |count(1)|
// +--------+
// |    500 |
// +--------+

// The results of SQL queries are themselves DataFrames and support all normal␣
↪functions.
val sqlDF = sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")

// The items in DataFrames are of type Row, which allows you to access each column by␣
↪ordinal.
val stringsDS = sqlDF.map {
  case Row(key: Int, value: String) => s"Key: $key, Value: $value"
}
stringsDS.show()
// +--------------------+
// |               value|
// +--------------------+
// |Key: 0, Value: val_0|
// |Key: 0, Value: val_0|
// |Key: 0, Value: val_0|
// ...

// You can also use DataFrames to create temporary views within a SparkSession.
val recordsDF = spark.createDataFrame((1 to 100).map(i => Record(i, s"val_$i")))
recordsDF.createOrReplaceTempView("records")

// Queries can then join DataFrame data with data stored in Hive.
sql("SELECT * FROM records r JOIN src s ON r.key = s.key").show()
// +---+------+---+------+
// |key| value|key| value|
// +---+------+---+------+
// |  2| val_2|  2| val_2|
// |  4| val_4|  4| val_4|
// |  5| val_5|  5| val_5|
// ...
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/hive/SparkHiveExample.scala” 文件。

Java

```
import java.io.File;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import org.apache.spark.api.java.function.MapFunction;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Encoders;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;

public static class Record implements Serializable {
  private int key;
  private String value;

  public int getKey() {
    return key;
  }

  public void setKey(int key) {
    this.key = key;
  }

  public String getValue() {
    return value;
  }

  public void setValue(String value) {
    this.value = value;
  }
}

// warehouseLocation points to the default location for managed databases and tables
String warehouseLocation = new File("spark-warehouse").getAbsolutePath();
SparkSession spark = SparkSession
  .builder()
  .appName("Java Spark Hive Example")
  .config("spark.sql.warehouse.dir", warehouseLocation)
  .enableHiveSupport()
  .getOrCreate();

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

// Queries are expressed in HiveQL
spark.sql("SELECT * FROM src").show();
// +---+-------+
// |key|  value|
// +---+-------+
// |238|val_238|
// | 86| val_86|
// |311|val_311|
// ...

// Aggregation queries are also supported.
spark.sql("SELECT COUNT(*) FROM src").show();
// +--------+
```

```
// |count(1)|
// +--------+
// |    500 |
// +--------+

// The results of SQL queries are themselves DataFrames and support all normal␣
↪functions.
Dataset<Row> sqlDF = spark.sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key
↪");

// The items in DataFrames are of type Row, which lets you to access each column by␣
↪ordinal.
Dataset<String> stringsDS = sqlDF.map(
    (MapFunction<Row, String>) row -> "Key: " + row.get(0) + ", Value: " + row.get(1),
    Encoders.STRING());
stringsDS.show();
// +--------------------+
// |               value|
// +--------------------+
// |Key: 0, Value: val_0|
// |Key: 0, Value: val_0|
// |Key: 0, Value: val_0|
// ...

// You can also use DataFrames to create temporary views within a SparkSession.
List<Record> records = new ArrayList<>();
for (int key = 1; key < 100; key++) {
  Record record = new Record();
  record.setKey(key);
  record.setValue("val_" + key);
  records.add(record);
}
Dataset<Row> recordsDF = spark.createDataFrame(records, Record.class);
recordsDF.createOrReplaceTempView("records");

// Queries can then join DataFrames data with data stored in Hive.
spark.sql("SELECT * FROM records r JOIN src s ON r.key = s.key").show();
// +---+------+---+------+
// |key| value|key| value|
// +---+------+---+------+
// |  2| val_2|  2| val_2|
// |  2| val_2|  2| val_2|
// |  4| val_4|  4| val_4|
// ...
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/hive/JavaSparkHiveExample.java” 文件。

Python

```
from os.path import expanduser, join, abspath

from pyspark.sql import SparkSession
from pyspark.sql import Row

# warehouse_location points to the default location for managed databases and tables
warehouse_location = abspath('spark-warehouse')
```

```
spark = SparkSession \
    .builder \
    .appName("Python Spark SQL Hive integration example") \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .enableHiveSupport() \
    .getOrCreate()

# spark is an existing SparkSession
spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")
spark.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src
↪")

# Queries are expressed in HiveQL
spark.sql("SELECT * FROM src").show()
# +---+-------+
# |key|  value|
# +---+-------+
# |238|val_238|
# | 86| val_86|
# |311|val_311|
# ...

# Aggregation queries are also supported.
spark.sql("SELECT COUNT(*) FROM src").show()
# +--------+
# |count(1)|
# +--------+
# |    500 |
# +--------+

# The results of SQL queries are themselves DataFrames and support all normal␣
↪functions.
sqlDF = spark.sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")

# The items in DataFrames are of type Row, which allows you to access each column by␣
↪ordinal.
stringsDS = sqlDF.rdd.map(lambda row: "Key: %d, Value: %s" % (row.key, row.value))
for record in stringsDS.collect():
    print(record)
# Key: 0, Value: val_0
# Key: 0, Value: val_0
# Key: 0, Value: val_0
# ...

# You can also use DataFrames to create temporary views within a SparkSession.
Record = Row("key", "value")
recordsDF = spark.createDataFrame([Record(i, "val_" + str(i)) for i in range(1, 101)])
recordsDF.createOrReplaceTempView("records")

# Queries can then join DataFrame data with data stored in Hive.
spark.sql("SELECT * FROM records r JOIN src s ON r.key = s.key").show()
# +---+------+---+------+
# |key| value|key| value|
# +---+------+---+------+
# |  2| val_2|  2| val_2|
# |  4| val_4|  4| val_4|
# |  5| val_5|  5| val_5|
# ...
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/hive.py” 文件。

R

When working with Hive one must instantiate SparkSession with Hive support. This adds support for finding tables in the MetaStore and writing queries using HiveQL.

```
# enableHiveSupport defaults to TRUE
sparkR.session(enableHiveSupport = TRUE)
sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")
sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

# Queries can be expressed in HiveQL.
results <- collect(sql("FROM src SELECT key, value"))
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/r/RSparkSQLExample.R” 文件

Specifying storage format for Hive tables

When you create a Hive table, you need to define how this table should read/write data from/to file system, i.e. the “input format” and “output format”. You also need to define how this table should deserialize the data to rows, or serialize rows to data, i.e. the “serde”. The following options can be used to specify the storage format(“serde”, “input format”, “output format”), e.g. CREATE TABLE src(id int) USING hive OPTIONS(fileFormat ‘parquet’). By default, we will read the table files as plain text. Note that, Hive storage handler is not supported yet when creating table, you can create a table using storage handler at Hive side, and use Spark SQL to read it.

属性名	含义
fileFormat	A fileFormat is kind of a package of storage format specifications, including “serde”, “input format” and “output format”. Currently we support 6 fileFormats: ‘sequencefile’, ‘rcfile’, ‘orc’, ‘parquet’, ‘textfile’ and ‘avro’.
inputFormat, outputFormat	These 2 options specify the name of a corresponding *InputFormat* and *OutputFormat* class as a string literal, e.g. *org.apache.hadoop.hive.ql.io.orc.OrcInputFormat*. These 2 options must be appeared in pair, and you can not specify them if you already specified the *fileFormat* option.
serde	This option specifies the name of a serde class. When the *fileFormat* option is specified, do not specify this option if the given *fileFormat* already include the information of serde. Currently “sequencefile”, “textfile” and “rcfile” don’t include the serde information and you can use this option with these 3 fileFormats.
fieldDelim, escapeDelim,	These options can only be used with “textfile” fileFormat. They define how to read delimited files into rows.
collectionDelim, mapkeyDelim, lineDelim	

All other properties defined with OPTIONS will be regarded as Hive serde properties.

与不同版本的Hive Metastore交互

Spark SQL对Hive最重要的一个支持就是可以和Hive metastore进行交互，这使得Spark SQL可以访问Hive表的元数据。从Spark 1.4.0版本开始，通过使用下面描述的配置, Spark SQL一个简单的二进制编译版本可以用来查询不同版本的Hive metastore。注意，不管用于访问 metastore的Hive是什么版本，Spark SQL内部都使用 Hive 1.2.1 版本进行编译, 并且使用这个版本的一些类用于内部执行（serdes，UDFs，UDAFs等）。

下面的选项可用来配置用于检索元数据的Hive版本:

属性名	默认值	含义
spark.sql.hive.metastore.version	1.2.1	Version of the Hive metastore. Available options are 0.12.0 through 1.2.1.
spark.sql.hive.metastore.jars	builtin	Location of the jars that should be used to instantiate the HiveMetastoreClient. This property can be one of three options: builtin Use Hive 1.2.1, which is bundled with the Spark assembly when -Phive is enabled. When this option is chosen, spark.sql.hive.metastore.version must be either 1.2.1 or not defined. maven Use Hive jars of specified version downloaded from Maven repositories. This configuration is not generally recommended for production deployments. A classpath in the standard format for the JVM. This classpath must include all of Hive and its dependencies, including the correct version of Hadoop. These jars only need to be present on the driver, but if you are running in yarn cluster mode then you must ensure they are packaged with your application.
spark.sql.hive.metastore.sharedPrefixes	com.mysql.jdbc, org.postgresql, com.microsoft.sqlserver, oracle.jdbc	A comma separated list of class prefixes that should be loaded using the classloader that is shared between Spark SQL and a specific version of Hive. An example of classes that should be shared is JDBC drivers that are needed to talk to the metastore. Other classes that need to be shared are those that interact with classes that are already shared. For example, custom appenders that are used by log4j.
spark.sql.hive.metastore.barrierPrefixes	(empty)	A comma separated list of class prefixes that should explicitly be reloaded for each version of Hive that Spark SQL is communicating with. For example, Hive UDFs that are declared in a prefix that typically would be shared (i.e. org.apache.spark.*).

JDBC To Other Databases

Spark SQL也包括一个可以使用JDBC从其它数据库读取数据的数据源。该功能应该优于使用JdbcRDD，因为它的返回结果是一个DataFrame，而在Spark SQL中DataFrame处理简单，且和其它数据源进行关联操作。JDBC数据源在Java和Python中用起来很简单，因为不需要用户提供一个ClassTag。（注意，这和 Spark SQL JDBC server不同，Spark SQL JDBC server 允许其它应用程序使用Spark SQL执行查询）

首先，你需要在 Spark classpath 中包含对应数据库的 JDBC driver。例如，为了从 Spark Shell 连接到 postgres 数据库，你需要运行下面的命令：

```
bin/spark-shell --driver-class-path postgresql-9.4.1207.jar --jars postgresql-9.4.
↪1207.jar
```

通过使用 Data Sources API, 远程数据库的表可以加载为一个 DataFrame 或 Spark SQL 临时表。支持的选项如下：

属性名	含义
url	The JDBC URL to connect to. The source-specific connection properties may be specified in the URL. e.g., jdbc:postgresql://localhost/test?user=fred&password=secret
dbtable	The JDBC table that should be read. Note that anything that is valid in a FROM clause of a SQL query can be used. For example, instead of a full table you could also use a subquery in parentheses.
driver	The class name of the JDBC driver to use to connect to this URL.
partitionColumn, lowerBound, upperBound	These options must all be specified if any of them is specified. In addition, numPartitions must be specified. They describe how to partition the table when reading in parallel from multiple workers. partitionColumn must be a numeric column from the table in question. Notice that lowerBound and upperBound are just used to decide the partition stride, not for filtering the rows in table. So all rows in the table will be partitioned and returned. This option applies only to reading.
numPartitions	The maximum number of partitions that can be used for parallelism in table reading and writing. This also determines the maximum number of concurrent JDBC connections. If the number of partitions to write exceeds this limit, we decrease it to this limit by calling coalesce(numPartitions) before writing.
fetchsize	The JDBC fetch size, which determines how many rows to fetch per round trip. This can help performance on JDBC drivers which default to low fetch size (eg. Oracle with 10 rows). This option applies only to reading.
batchsize	The JDBC batch size, which determines how many rows to insert per round trip. This can help performance on JDBC drivers. This option applies only to writing. It defaults to 1000.
isolationLevel	The transaction isolation level, which applies to current connection. It can be one of NONE, READ_COMMITTED, READ_UNCOMMITTED, REPEATABLE_READ, or SERIALIZABLE, corresponding to standard transaction isolation levels defined by JDBC's Connection object, with default of READ_UNCOMMITTED. This option applies only to writing. Please refer the documentation in java.sql.Connection.
truncate	This is a JDBC writer related option. When SaveMode.Overwrite is enabled, this option causes Spark to truncate an existing table instead of dropping and recreating it. This can be more efficient, and prevents the table metadata (e.g., indices) from being removed. However, it will not work in some cases, such as when the new data has a different schema. It defaults to false. This option applies only to writing.
createTableOptions	This is a JDBC writer related option. If specified, this option allows setting of database-specific table and partition options when creating a table (e.g., CREATE TABLE t (name string) ENGINE=InnoDB.). This option applies only to writing.
createTableColumnTypes	The database column data types to use instead of the defaults, when creating the table. Data type information should be specified in the same format as CREATE TABLE columns syntax (e.g: "name CHAR(64), comments VARCHAR(1024)"). The specified types should be valid spark sql data types. This option applies only to writing.

Scala

```
// Note: JDBC loading and saving can be achieved via either the load/save or jdbc␣
↪methods
// Loading data from a JDBC source
val jdbcDF = spark.read
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "schema.tablename")
  .option("user", "username")
  .option("password", "password")
  .load()

val connectionProperties = new Properties()
connectionProperties.put("user", "username")
connectionProperties.put("password", "password")
val jdbcDF2 = spark.read
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties)

// Saving data to a JDBC source
jdbcDF.write
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "schema.tablename")
  .option("user", "username")
  .option("password", "password")
  .save()

jdbcDF2.write
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties)

// Specifying create table column data types on write
jdbcDF.write
  .option("createTableColumnTypes", "name CHAR(64), comments VARCHAR(1024)")
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties)
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala” 文件。

Java

```
// Note: JDBC loading and saving can be achieved via either the load/save or jdbc␣
↪methods
// Loading data from a JDBC source
Dataset<Row> jdbcDF = spark.read()
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "schema.tablename")
  .option("user", "username")
  .option("password", "password")
  .load();

Properties connectionProperties = new Properties();
connectionProperties.put("user", "username");
connectionProperties.put("password", "password");
Dataset<Row> jdbcDF2 = spark.read()
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties);

// Saving data to a JDBC source
```

```
jdbcDF.write()
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "schema.tablename")
  .option("user", "username")
  .option("password", "password")
  .save();

jdbcDF2.write()
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties);

// Specifying create table column data types on write
jdbcDF.write()
  .option("createTableColumnTypes", "name CHAR(64), comments VARCHAR(1024)")
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties);
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/java/org/apache/spark/examples/sql/JavaSQLDataSourceExample.java” 文件。

Python

```
# Note: JDBC loading and saving can be achieved via either the load/save or jdbc␣
↪methods
# Loading data from a JDBC source
jdbcDF = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:postgresql:dbserver") \
    .option("dbtable", "schema.tablename") \
    .option("user", "username") \
    .option("password", "password") \
    .load()

jdbcDF2 = spark.read \
    .jdbc("jdbc:postgresql:dbserver", "schema.tablename",
          properties={"user": "username", "password": "password"})

# Saving data to a JDBC source
jdbcDF.write \
    .format("jdbc") \
    .option("url", "jdbc:postgresql:dbserver") \
    .option("dbtable", "schema.tablename") \
    .option("user", "username") \
    .option("password", "password") \
    .save()

jdbcDF2.write \
    .jdbc("jdbc:postgresql:dbserver", "schema.tablename",
          properties={"user": "username", "password": "password"})

# Specifying create table column data types on write
jdbcDF.write \
    .option("createTableColumnTypes", "name CHAR(64), comments VARCHAR(1024)") \
    .jdbc("jdbc:postgresql:dbserver", "schema.tablename",
          properties={"user": "username", "password": "password"})
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/python/sql/datasource.py” 文件。

R

```
# Loading data from a JDBC source
df <- read.jdbc("jdbc:postgresql:dbserver", "schema.tablename", user = "username",␣
↪password = "password")

# Saving data to a JDBC source
write.jdbc(df, "jdbc:postgresql:dbserver", "schema.tablename", user = "username",␣
↪password = "password")
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/r/RSparkSQLExample.R” 文件。

Sql

```
CREATE TEMPORARY VIEW jdbcTable
USING org.apache.spark.sql.jdbc
OPTIONS (
  url "jdbc:postgresql:dbserver",
  dbtable "schema.tablename",
  user 'username',
  password 'password'
)

INSERT INTO TABLE jdbcTable
SELECT * FROM resultTable
```

Troubleshooting

- 在client session以及所有的executor上，JDBC驱动器类必须对启动类加载器可见。这是因为Java的DriverManager类在打开一个连接之前会做一个安全检查，这样就导致它忽略了对于启动类加载器不可见的所有驱动器。一种简单的方法就是修改所有worker节点上的compute_classpath.sh以包含你驱动器的jar包。
- 有些数据库，比如H2，会把所有的名称转换成大写。在Spark SQL中你也需要使用大写来引用这些名称。

1.3.4 性能调优

对于有一定计算量的Spark任务，可以将数据放入内存缓存或开启一些试验选项来提升性能。

缓存数据到内存中

通过调用 spark.cacheTable(“tableName”) 或者 dataFrame.cache() 方法, Spark SQL可以使用一种内存列存储格式缓存表。接着Spark SQL只扫描必要的列，并且自动调整压缩比例，以最小化内存占用和GC压力。你可以调用 spark.uncacheTable(“tableName”) 方法删除内存中的表。

内存缓存配置可以使用 SparkSession 类中的 setConf 方法或在SQL语句中运行 SET key=value命令来完成。

属性名	默认值	含义
spark.sql.inMemoryColumnarStorage.compressed	true	如果设置为true，Spark SQL将会基于统计数据自动地为每一列选择一种压缩编码方式。
spark.sql.inMemoryColumnarStorage.batchSize	10000	控制列式缓存批处理大小。缓存数据时, 较大的批处理大小可以提高内存利用率和压缩率，但同时也会带来OOM（Out Of Memory）的风险。

其它配置选项

下面的选项也可以用来提升执行的查询性能。随着Spark自动地执行越来越多的优化操作, 这些选项在未来的发布版本中可能会过时。

属性名	默认值	含义
spark.sql.files.maxPartitionBytes	134217728 (128 MB)	读取文件时单个分区可容纳的最大字节数
spark.sql.files.openCostInBytes	4194304 (4 MB)	打开文件的估算成本, 按照同一时间能够扫描的字节数来测量。当往一个分区写入多个文件的时候会使用。高估更好, 这样的话小文件分区将比大文件分区更快(先被调度).
spark.sql.autoBroadcastJoinThreshold	10485760 (10 MB)	用于配置一个表在执行 join 操作时能够广播给所有worker节点的最大字节大小。通过将这个值设置为 -1 可以禁用广播。注意，当前数据统计仅支持已经运行了ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan 命令的Hive Metastore表。
spark.sql.shuffle.partitions	200	用于配置join或聚合操作混洗（shuffle）数据时使用的分区数。

1.3.5 分布式 SQL 引擎

通过使用 JDBC/ODBC或者命令行接口，Spark SQL还可以作为一个分布式查询引擎。在这种模式下，终端用户或应用程序可以运行SQL查询来直接与Spark SQL交互，而不需要编写任何代码。

运行 Thrift JDBC/ODBC 服务器

这里实现的Thrift JDBC/ODBC server对应于Hive 1.2.1 版本中的HiveServer2。你可以使用Spark或者Hive 1.2.1自带的beeline脚本来测试这个JDBC server。

要启动JDBC/ODBC server， 需要在Spark安装目录下运行下面这个命令:

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

这个脚本能接受所有 bin/spark-submit 命令行选项，外加一个用于指定Hive属性的 –hiveconf 选项。你可以运行 ./sbin/start-thriftserver.sh —help 来查看所有可用选项的完整列表。默认情况下，启动的 server 将会在 localhost:10000 上进行监听。你可以覆盖该行为, 比如使用以下环境变量:

```
export HIVE_SERVER2_THRIFT_PORT=<listening-port>
export HIVE_SERVER2_THRIFT_BIND_HOST=<listening-host>
./sbin/start-thriftserver.sh \
  --master <master-uri> \
  ...
```

或者系统属性:

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

现在你可以使用 beeline来测试这个Thrift JDBC/ODBC server:

```
./bin/beeline
```

在beeline中使用以下命令连接到JDBC/ODBC server：

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

Beeline会要求你输入用户名和密码。在非安全模式下，只需要输入你本机的用户名和一个空密码即可。对于安全模式，请参考beeline文档中的指示.

将 hive-site.xml，core-site.xml以及hdfs-site.xml文件放置在conf目录下可以完成Hive配置。

你也可以使用Hive 自带的 beeline 的脚本。

Thrift JDBC server还支持通过HTTP传输来发送Thrift RPC消息。使用下面的设置来启用HTTP模式：

hive.server2.transport.mode - Set this to value: http hive.server2.thrift.http.port - HTTP port number fo listen on; default is 10001 hive.server2.http.endpoint - HTTP endpoint; default is cliservice

为了测试，下面在HTTP模式中使用beeline连接到JDBC/ODBC server:

```
beeline> !connect jdbc:hive2://<host>:<port>/<database>?hive.server2.transport.
↪mode=http;hive.server2.thrift.http.path=<http_endpoint>
```

运行 Spark SQL CLI

Spark SQL CLI是一个很方便的工具，它可以在本地模式下运行Hive metastore服务，并且执行从命令行中输入的查询语句。注意：Spark SQL CLI无法与Thrift JDBC server通信。

要启动用Spark SQL CLI, 可以在Spark安装目录运行下面的命令:

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

将 hive-site.xml，core-site.xml以及hdfs-site.xml文件放置在conf目录下可以完成Hive配置。你可以运行 ./bin/spark-sql –help 来获取所有可用选项的完整列表。

1.3.6 迁移指南

Spark SQL 从 2.1 版本升级到2.2 版本

- Spark 2.1.1 introduced a new configuration key: spark.sql.hive.caseSensitiveInferenceMode. It had a default setting of NEVER_INFER, which kept behavior identical to 2.1.0. However, Spark 2.2.0 changes this setting's default value to INFER_AND_SAVE to restore compatibility with reading Hive metastore tables whose underlying file schema have mixed-case column names. With the INFER_AND_SAVE configuration value, on first access Spark will perform schema inference on any Hive metastore table for which it has not already saved an inferred schema. Note that schema inference can be a very time consuming operation for tables with thousands of partitions. If compatibility with mixed-case column names is not a concern, you can safely set spark.sql.hive.caseSensitiveInferenceMode to NEVER_INFER to avoid the initial overhead of schema inference. Note that with the new default INFER_AND_SAVE setting, the results of the schema inference are saved as a metastore key for future use. Therefore, the initial schema inference occurs only at a table's first access.

Spark SQL 从 2.0 版本升级到 2.1 版本

- Datasource tables now store partition metadata in the Hive metastore. This means that Hive DDLs such as ALTER TABLE PARTITION ... SET LOCATION are now available for tables created with the Datasource API.

Legacy datasource tables can be migrated to this format via the MSCK REPAIR TABLE command. Migrating legacy tables is recommended to take advantage of Hive DDL support and improved planning performance. To determine if a table has been migrated, look for the PartitionProvider: Catalog attribute when issuing DESCRIBE FORMATTED on the table.

- Changes to INSERT OVERWRITE TABLE ... PARTITION ... behavior for Datasource tables.

In prior Spark versions INSERT OVERWRITE overwrote the entire Datasource table, even when given a partition specification. Now only partitions matching the specification are overwritten. Note that this still differs from the behavior of Hive tables, which is to overwrite only partitions overlapping with newly inserted data.

Spark SQL 从 1.6 版本升级到 2.0 版本

- SparkSession 现在是 Spark 新的切入点, 它替代了老的 SQLContext 和 HiveContext。注意：为了向下兼容, 老的 SQLContext 和 HiveContext 仍然保留。可以从 SparkSession 获取一个新的 catalog 接口- 现有的访问数据库和表的API, 如 listTables, createExternalTable, dropTempView, cacheTable 都被移到该接口。
- Dataset API 和 DataFrame API 进行了统一。在 Scala 中, DataFrame 变成了 Dataset[Row]的一个类型别名, 而Java API使用者必须将 DataFrame 替换成 Dataset<Row>。Dataset 类既提供了强类型转换操作 (如 map, filter 以及 groupByKey) 也提供了非强类型转换操作 (如 select 和 groupBy) 。由于编译期的类型安全不是 Python 和 R 语言的一个特性, Dataset 的概念并不适用于这些语言的 API。相反, DataFrame 仍然是最基本的编程抽象, 就类似于这些语言中单节点数据帧的概念。
- Dataset 和 DataFrame API 中 unionAll 已经过时并且由 union 替代。
- Dataset 和 DataFrame API 中 explode 已经过时。或者 functions.explode() 可以结合 select 或 flatMap 一起使用。
- Dataset 和 DataFrame API 中 registerTempTable 已经过时并且由 createOrReplaceTempView 替代。

Spark SQL 从 1.5 版本升级到 1.6 版本

- 从 Spark 1.6 版本开始，Thrift server 默认运行于多会话模式下, 这意味着每个JDBC/ODBC连接都有独有一份SQL配置和临时函数注册表的拷贝。尽管如此, 缓存的表仍然可以共享。如果你更喜欢在老的单会话模式中运行Thrift server，只需要将spark.sql.hive.thriftServer.singleSession选项设置为true即可。当然，你也可在spark-defaults.conf文件中添加这个选项，或者通过–conf将其传递给start-thriftserver.sh：

```
./sbin/start-thriftserver.sh \
    --conf spark.sql.hive.thriftServer.singleSession=true \
    ...
```

- 从 Spark 1.6.1 版本开始, sparkR 中的 withColumn 方法支持向 DataFrame 新增一列 或 替换已有的名称相同的列。
- 从 Spark 1.6 版本开始, LongType 转换成 TimestampType 将源值以秒而不是毫秒作为单位处理。做出这个变更是为了的匹配Hive 1.2 版本中从数值类型转换成TimestampType的这个行为以获得更一致的类型。更多细节请参见 SPARK-11724 。

Spark SQL 从 1.4 版本升级到 1.5 版本

- 使用手动管理内存(Tungsten引擎)的执行优化以及用于表达式求值的代码自动生成现在默认是启用的。这些特性可以通过将spark.sql.tungsten.enabled的值设置为false来同时禁用。
- 默认不启用 Parquet schema 合并。可以将 spark.sql.parquet.mergeSchema 的值设置为true来重新启用。

- Python 中对于列的字符串分解现在支持使用点号(.)来限定列或访问内嵌值，例如df['table.column.nestedField']。然而这也意味着如果你的列名包含任何点号(.)的话，你就必须要使用反引号来转义它们(例如：table.`column.with.dots`.nested)。
- 默认启用内存中列式存储分区修剪。可以通过设置 spark.sql.inMemoryColumarStorage.partitionPruning值为false来禁用它。
- 不再支持无精度限制的decimal，相反, Spark SQL现在强制限制最大精度为38位。从BigDecimal对象推导schema时会使用（38，18）这个精度。如果在DDL中没有指定精度，则默认使用精度Decimal(10，0)。
- 存储的时间戳(Timestamp)现在精确到1us（微秒），而不是1ns（纳秒）
- 在 sql 方言中，浮点数现在被解析成decimal。HiveQL 的解析保持不变。
- SQL/DataFrame函数的规范名称均为小写(例如：sum vs SUM)。
- JSON数据源不会再自动地加载其他应用程序创建的新文件（例如，不是由Spark SQL插入到dataset中的文件）。对于一个JSON持久化表（例如：存储在Hive metastore中的表的元数据），用户可以使用 REFRESH TABLE 这个SQL命令或者 HiveContext 的 refreshTable 方法来把新文件添加进表。对于一个表示JSON数据集的DataFrame, 用户需要重建这个 DataFrame, 这样新的 DataFrame 就会包含新的文件。
- pySpark 中的 DataFrame.withColumn 方法支持新增一列或是替换名称相同列。

Spark SQL 从 1.3 版本升级到 1.4 版本

DataFrame数据读写接口

根据用户的反馈，我们提供了一个用于数据读入（SQLContext.read）和数据写出（DataFrame.write）的新的、更加流畅的API，同时老的API（如：SQLCOntext.parquetFile, SQLContext.jsonFile）将被废弃。

有关 SQLContext.read (Scala, Java, Python) 和 DataFrame.write (Scala, Java, Python) 的更多信息，请参考API文档。

DataFrame.groupBy保留分组的列

根据用户的反馈，我们改变了DataFrame.groupBy().agg()的默认行为，就是在返回的DataFrame结果中保留分组的列。如果你想保持1.3版本中的行为，可以将spark.sql.retainGroupColumns设置为false。

Scala

```
// In 1.3.x, in order for the grouping column "department" to show up,
// it must be included explicitly as part of the agg function call.
df.groupBy("department").agg($"department", max("age"), sum("expense"))

// In 1.4+, grouping column "department" is included automatically.
df.groupBy("department").agg(max("age"), sum("expense"))

// Revert to 1.3 behavior (not retaining grouping column) by:
sqlContext.setConf("spark.sql.retainGroupColumns", "false")
```

Java

```
// In 1.3.x, in order for the grouping column "department" to show up,
// it must be included explicitly as part of the agg function call.
df.groupBy("department").agg(col("department"), max("age"), sum("expense"));
```

```
// In 1.4+, grouping column "department" is included automatically.
df.groupBy("department").agg(max("age"), sum("expense"));

// Revert to 1.3 behavior (not retaining grouping column) by:
sqlContext.setConf("spark.sql.retainGroupColumns", "false");
```

Python

```
import pyspark.sql.functions as func

# In 1.3.x, in order for the grouping column "department" to show up,
# it must be included explicitly as part of the agg function call.
df.groupBy("department").agg(df["department"], func.max("age"), func.sum("expense"))

# In 1.4+, grouping column "department" is included automatically.
df.groupBy("department").agg(func.max("age"), func.sum("expense"))

# Revert to 1.3.x behavior (not retaining grouping column) by:
sqlContext.setConf("spark.sql.retainGroupColumns", "false")
```

Behavior change on DataFrame.withColumn

1.4版本之前, DataFrame.withColumn() 只支持新增一列。在DataFrame结果中指定名称的列总是作为一个新列添加进来，即使已经存在了相同名称的列。从1.4版本开始, DataFrame.withColumn() 支持新增一个和现有列名不重复的新列和替换有相同名称的列。

注意：这个变更只针对 Scala API, 不针对 PySpark 和 SparkR。

Spark SQL 从 1.0-1.2 版本升级到 1.3 版本

Spark 1.3版本我们去掉了Spark SQL的 "Alpha" 标签并且作为其中的一部分我们在现有的API上做了清理。从Spark 1.3版本开始，Spark SQL将提供1.x系列中其它发行版本的二进制兼容。这个兼容性保证不包括显式地标注为不稳定（例如：DeveloperAPI 或 Experimental）的API。

SchemaRDD重命名为DataFrame

升级到Spark SQL 1.3后，用户将会注意到最大的改动就是 SchemaRDD 改名为 DataFrame。主要原因是DataFrame不再直接继承于RDD，而是通过自己的实现来提供RDD中提供的绝大多数功能。通过调用 .rdd 方法 DataFrame 仍然可以转换成RDD。

在Scala中有一个从SchemaRDD到DataFrame的类型别名来提供某些使用场景下的代码兼容性。但仍然建议用户在代码中改用DataFrame。Java和Python用户必须要修改代码。

统一Java和Scala API

Spark 1.3 之前的版本中有两个单独的Java兼容类（JavaSQLContext 和 JavaSchemaRDD）可以映射到 Scala API。Spark 1.3版本将Java API和Scala API进行了统一。两种语言的用户都应该使用SQLContext和DataFrame。通常情况下这些类都会使用两种语言中都支持的类型（例如：使用Array来取代语言特有的集合）。有些情况下没有通用的类型（例如：闭包或maps中用于传值），则会使用函数重载。

另外，移除了Java特有的类型API。Scala 和 Java 用户都应该使用 org.apache.spark.sql.types 包中的类来编程式地描述 schema。

隔离隐式转换并删除dsl包(仅针对Scala)

Spark 1.3版本之前的很多示例代码都以 import sqlContext._ 语句作为开头，这样会引入sqlContext的所有函数。在Spark 1.3版本中我们隔离了RDD到DataFrame的隐式转换，将其单独放到SQLContext内部的一个对象中。用户现在应该这样写：import sqlContext.implicits._。

另外，隐式转换现在也只能使用toDF方法来增加由Product（例如：case classes 或 元祖）组成的RDD，而不是自动转换。

使用 DSL（现在被DataFrame API取代）的内部方法时，用户需要引入 import org.apache.spark.sql.catalyst.dsl。而现在应该要使用公用的 DataFrame函数API：import org.apache.spark.sql.functions._

移除org.apache.spark.sql中DataType的类型别名(仅针对Scala)

Spark 1.3版本删除了基础sql包中DataType的类型别名。开发人员应该引入 org.apache.spark.sql.types 中的类。

UDF注册迁移到sqlContext.udf中(Java&Scala)

用于注册UDF的函数，不管是DataFrame DSL还是SQL中用到的，都被迁移到SQLContext中的udf对象中。

Scala

```
sqlContext.udf.register("strLen", (s: String) => s.length())
```

Java

```
sqlContext.udf().register("strLen", (String s) -> s.length(), DataTypes.IntegerType);
```

Python UDF注册保持不变。

Python的DataType不再是单例的

在 Python 中使用DataTypes时，你需要先构造它们（如：StringType()），而不是引用一个单例对象。

兼容Apache Hive

Spark SQL 在设计时就考虑到了和 Hive metastore，SerDes 以及 UDF 之间的兼容性。目前 Hive SerDes 和 UDF 都是基于Hive 1.2.1版本，并且Spark SQL可以连接到不同版本的Hive metastore（从0.12.0到1.2.1，可以参考[与不同版本的Hive Metastore交互]）

在已有的Hive仓库中部署

Spark SQL Thrift JDBC server采用了开箱即用的设计以兼容已有的 Hive 安装版本。你不需要修改现有的Hive Metastore , 或者改变数据的位置和表的分区。

支持的 Hive 功能

Spark SQL 支持绝大部分的Hive功能，如：

- **Hive查询语句, 包括：**
  - SELECT
  - GROUP BY
  - ORDER BY
  - CLUSTER BY
  - SORT BY
- **所有的Hive运算符， 包括：**
  - 关系运算符 (=, , ==, <>, <, >, >=, <=, etc)
  - 算术运算符 (+, -, *, /, %, etc)
  - 逻辑运算符 (AND, &&, OR, ||, etc)
  - 复杂类型构造器
  - 数学函数 (sign, ln, cos等)
  - String 函数 (instr, length, printf等)
- 用户自定义函数（UDF）
- 用户自定义聚合函数（UDAF）
- 用户自定义序列化格式（SerDes）
- 窗口函数
- **Joins**
  - JOIN
  - {LEFT|RIGHT|FULL} OUTER JOIN
  - LEFT SEMI JOIN
  - CROSS JOIN
- Unions
- **子查询**
  - SELECT col FROM (SELECT a + b AS col from t1) t2
- 采样
- Explain
- 分区表，包括动态分区插入
- 视图
- **所有Hive DDL功能, 包括：**
  - CREATE TABLE
  - CREATE TABLE AS SELECT
  - ALTER TABLE

- 绝大多数 Hive 数据类型，包括：
  - TINYINT
  - SMALLINT
  - INT
  - BIGINT
  - BOOLEAN
  - FLOAT
  - DOUBLE
  - STRING
  - BINARY
  - TIMESTAMP
  - DATE
  - ARRAY<>
  - MAP<>
  - STRUCT<>

不支持的 Hive 功能

以下是目前还不支持的 Hive 功能列表。在 Hive 部署中这些功能大部分都用不到。

Hive 核心功能

- bucket：bucket是 Hive 表分区内的一个哈希分区，Spark SQL 目前还不支持 bucket。

Hive 高级功能

- UNION 类型
- Unique join
- 列统计数据收集：Spark SQL 目前不依赖扫描来收集列统计数据并且仅支持填充Hive metastore 的 sizeInBytes 字段。

Hive输入输出格式

- CLI文件格式：对于回显到CLI中的结果，Spark SQL 仅支持 TextOutputFormat。
- Hadoop archive

Hive优化

有少数Hive优化还没有包含在Spark中。其中一些（比如索引）由于Spark SQL的这种内存计算模型而显得不那么重要。另外一些在Spark SQL未来的版本中会持续跟踪。

- 块级别位图索引和虚拟列（用来建索引）
- 自动为 join 和 groupBy 计算 reducer 个数：目前在 Spark SQL 中，你需要使用 "SET spark.sql.shuffle.partitions=[num_tasks];" 来控制后置混洗的并行程度。
- 仅查询元数据：对于只需要使用元数据的查询请求，Spark SQL 仍需要启动任务来计算结果
- 数据倾斜标志：Spark SQL 不遵循 Hive 中的数据倾斜标志
- STREAMTABLE join操作提示：Spark SQL 不遵循 STREAMTABLE 提示。
- 对于查询结果合并多个小文件：如果返回的结果有很多小文件，Hive有个选项设置，来合并小文件，以避免超过HDFS的文件数额度限制。Spark SQL不支持这个。

1.3.7 参考

数据类型

Spark SQL 和 DataFrame 支持以下数据类型：

- **数值类型**
  - ByteType: 表示1字节长的有符号整型，数值范围：-128 到 127。
  - ShortType: 表示2字节长的有符号整型，数值范围：-32768 到 32767。
  - IntegerType: 表示4字节长的有符号整型，数值范围：-2147483648 到 2147483647。
  - LongType: 表示8字节长的有符号整型，数值范围：-9223372036854775808 to 9223372036854775807。
  - FloatType: 表示4字节长的单精度浮点数。
  - DoubleType: 表示8字节长的双精度浮点数。
  - DecimalType: 表示任意精度的有符号的十进制数。内部使用 java.math.BigDecimal 实现。一个 BigDecimal 由一个任意精度的整数非标度值和一个32位的整数标度组成。
- **字符串类型**
  - StringType: 表示字符串值。
- **二进制类型**
  - BinaryType: 表示字节序列值。
- **布尔类型**
  - BooleanType: 表示布尔值。
- **时间类型**
  - TimestampType: 表示由年、月、日、时、分以及秒等字段值组成的时间值。
  - DateType: 表示由年、月、日字段值组成的日期值。
- **复杂类型**
  - ArrayType(elementType, containsNull): 表示由元素类型为 elementType 的序列组成的值，containsNull 用来标识 ArrayType 中的元素值能否为 null。

  - MapType(keyType, valueType, valueContainsNull)：表示由一组键值对组成的值。键的数据类型由 keyType 表示，值的数据类型由 valueType 表示。对于 MapType 值，键值不允许为 null。valueContainsNull 用来表示一个 MapType 的值是否能为 null。

- **StructType(fields)：表示由 StructField 序列描述的结构。**
  - StructField(name, datatype, nullable)：表示 StructType 中的一个字段，name 表示字段名。dataType 表示字段的数据类型，nullable 用来表示该字段的值是否可以为 null。

Scala

Spark SQL 所有的数据类型都位于 org.apache.spark.sql.types 包中。你可以使用下面的语句访问他们:

```
import org.apache.spark.sql.types._
```

完整示例代码参见 Spark 源码仓库中的 “examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala” 文件。

Java

Spark SQL 所有的数据类型都位于 org.apache.spark.sql.types 包中。如果想要访问或创建一个数据类型, 请使用 org.apache.spark.sql.types.DataTypes 中提供的工厂方法。

Python

Spark SQL 所有的数据类型都位于 pyspark.sql.types 包中。你可以使用下面的语句访问他们:

```
from pyspark.sql.types import *
```

R

NaN 语义

当处理一些不符合标准浮点语义的 float 或 double 类型时，会对 Not-a-Number(NaN) 做一些特殊处理。具体如下:

- NaN = NaN 返回true。
- 在聚合操作中，所有 NaN 值都被分到同一组。
- 在连接键中 NaN 被当做普通值。
- NaN 值按升序排序时排最后，比其他任何数值都大。

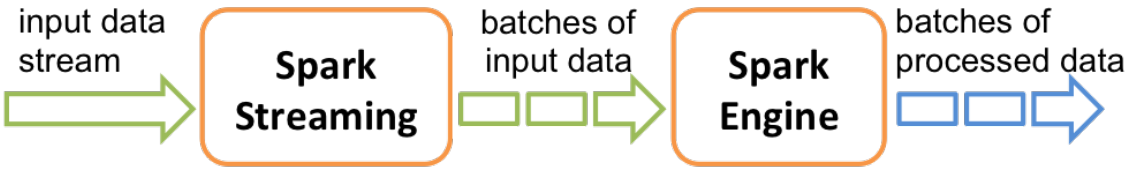
1.4 Structured Streaming编程指南

1.5 Spark Streaming 编程指南

1.5.1 概述

Spark Streaming 是对核心 Spark API 的一个扩展，它能够实现对实时数据流的流式处理，并具有很好的可扩展性、高吞吐量和容错性。Spark Streaming 支持从多种数据源提取数据，如：Kafka、Flume、Twitter、ZeroMQ、Kinesis 以及 TCP 套接字，并且可以提供一些高级 API 来表达复杂的处理算法，如：map、reduce、join 和 window 等。最后，Spark Streaming 支持将处理完的数据推送到文件系统、数据库或者实时仪表盘中展示。实际上，你完全可以将 Spark 的机器学习（machine learning） 和图计算（graph processing）的算法应用于 Spark Streaming 的数据流当中。

下图展示了 Spark Streaming 的内部工作原理。Spark Streaming 接收实时输入数据流并将数据划分为一个个小的批次供 Spark Engine 处理，最终生成多个批次的结果流。



Spark Streaming 为这种持续的数据流提供了一个高级抽象，即：discretized stream(离散数据流) 或 DStream。DStream 既可以从输入数据源创建而来，如：Kafka、Flume或者Kinesis，也可以从其他 DStream 上应用一些高级操作得到。在 Spark 内部，一个 DStream 代表一个 RDD 序列。

本文档将向你展示如何用 DStream 进行 Spark Streaming 编程。Spark Streaming 支持 Scala、Java 和 Python（始于Spark 1.2），本文档的示例包括这三种语言。

注意：对 Python 来说，有一部分 API 尚不支持，或者是和 Scala、Java 不同。本文档中会用高亮形式来注明这部分 Python API。

1.5.2 一个小例子

在深入 Spark Streaming 编程细节之前，我们先来看看一个简单的小例子以便有个感性认识。假设我们在一个 TCP 端口上监听一个数据服务器的数据，并对收到的文本数据中的单词计数。以下你所需的全部工作：

Scala

首先，我们需要导入Spark Streaming 的相关class的一些包，以及一些支持 StreamingContext 隐式转换的包（这些隐式转换能给DStream之类的class增加一些有用的方法）。StreamingContext 是 Spark Streaming 的入口。我们将会创建一个本地 StreamingContext 对象，包含两个执行线程，并将批次间隔设为1秒。

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._ // 从Spark 1.3之后这行就可以不需要了

// 创建一个local StreamingContext，包含2个工作线程，并将批次间隔设为1秒
// master至少需要2个CPU核，以避免出现任务饿死的情况
```

```
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

利用这个上下文对象（StreamingContext），我们可以创建一个DStream，该DStream代表从前面的TCP数据源流入的数据流，同时TCP数据源是由主机名（如：hostnam）和端口（如：9999）来描述的。

```
// 创建一个连接到hostname:port的DStream，如：localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)
```

这里的 lines 就是从数据server接收到的数据流。其中每一条记录都是一行文本。接下来，我们就需要把这些文本行按空格分割成单词。

```
// 将每一行分割成多个单词
val words = lines.flatMap(_.split(" "))
```

flatMap 是一种 “一到多”（one-to-many）的映射算子，它可以将源DStream中每一条记录映射成多条记录，从而产生一个新的DStream对象。在本例中，lines中的每一行都会被flatMap映射为多个单词，从而生成新的words DStream对象。然后，我们就能对这些单词进行计数了。

```
import org.apache.spark.streaming.StreamingContext._ // Spark 1.3之后不再需要这行
// 对每一批次中的单词进行计数
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// 将该DStream产生的RDD的头十个元素打印到控制台上
wordCounts.print()
```

words这个DStream对象经过map算子（一到一的映射）转换为一个包含（word, 1）键值对的DStream对象pairs，再对pairs使用reduce算子，得到每个批次中各个单词的出现频率。最后，wordCounts.print() 将会每秒（前面设定的批次间隔）打印一些单词计数到控制台上。

注意，执行以上代码后，Spark Streaming 只是将计算逻辑设置好，此时并未真正的开始处理数据。要启动之前的处理逻辑，我们还需要如下调用：

```
ssc.start()             // 启动流式计算
ssc.awaitTermination()  // 等待直到计算终止
```

完整的代码可以在 Spark Streaming 的例子 NetworkWordCount 中找到。

如果你已经有一个 Spark 包（下载在这里downloaded，自定义构建在这里built），就可以执行按如下步骤运行这个例子。

首先，你需要运行 netcat（Unix-like系统都会有这个小工具），将其作为data server

```
$ nc -lk 9999
```

然后，在另一个终端，按如下指令执行这个例子

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

好了，现在你尝试可以在运行 netcat 的终端里敲几个单词，你会发现这些单词以及相应的计数会出现在启动 Spark Streaming 例子的终端屏幕上。看上去应该和下面这个示意图类似：

TERMINAL 1: # Running Netcat

$ nc -lk 9999

hello world

...

TERMINAL 2: RUNNING NetworkWordCount$ ./bin/run-example streaming.NetworkWordCount localhost 9999 ... ——————————————————————- Time: 1357008430000 ms ——————————————————————- (hello,1) (world,1) ...

1.5.3 基本概念

下面，我们在之前的小栗子基础上，继续深入了解一下 Spark Streaming 的一些基本概念。

链接依赖项

和 Spark 类似，Spark Streaming 也能在 Maven 库中找到。如果你需要编写 Spark Streaming 程序，你就需要将以下依赖加入到你的 SBT 或 Maven 工程依赖中。

Maven

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.10</artifactId>
    <version>1.6.1</version>
</dependency>
```

SBT

```
libraryDependencies += "org.apache.spark" % "spark-streaming_2.11" % "2.2.1"
```

还有，对于从 Kafka、Flume 以及 Kinesis 这类数据源提取数据的流式应用来说，还需要额外增加相应的依赖项，下表列出了各种数据源对应的额外依赖项：

数据源	Maven构件
Kafka	spark-streaming-kafka_2.11
Flume	spark-streaming-flume_2.11
Kinesis	spark-streaming-kinesis-asl_2.11 [Amazon Software License]

最新的依赖项信息（包括源代码和 Maven 构件）请参考 Maven repository。

初始化 StreamingContext

要初始化任何一个 Spark Streaming 程序，都需要在入口代码中创建一个 StreamingContext 对象。

Scala

而 StreamingContext 对象需要一个 SparkConf 对象作为其构造参数。

```
import org.apache.spark._
import org.apache.spark.streaming._

val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

上面代码中的 appName 是你给该应用起的名字，这个名字会展示在 Spark 集群的 web UI上。而 master 是 Spark, Mesos or YARN cluster URL，如果支持本地测试，你也可以用"local[*]"为其赋值。通常在实际工作中，你不应该将master参数硬编码到代码里，而是应用通过spark-submit的参数来传递master的值（launch the application with spark-submit ）。不过对本地测试来说，"local[*]"足够了（该值传给master后，Spark Streaming将在本地进程中，启动n个线程运行，n与本地系统CPU core数相同）。注意，StreamingContext在

内部会创建一个 SparkContext 对象（SparkContext是所有Spark应用的入口，在StreamingContext对象中可以这样访问：ssc.sparkContext）。

StreamingContext 还有另一个构造参数，即：批次间隔，这个值的大小需要根据应用的具体需求和可用的集群资源来确定。详见Spark性能调优（Performance Tuning）。

StreamingContext 对象也可以通过已有的 SparkContext 对象来创建，示例如下：

```
import org.apache.spark.streaming._

val sc = ...                // 已有的SparkContext
val ssc = new StreamingContext(sc, Seconds(1))
```

Java

A JavaStreamingContext object can be created from a SparkConf object.

```
import org.apache.spark.*;
import org.apache.spark.streaming.api.java.*;

SparkConf conf = new SparkConf().setAppName(appName).setMaster(master);
JavaStreamingContext ssc = new JavaStreamingContext(conf, new Duration(1000));
```

The appName parameter is a name for your application to show on the cluster UI. master is a Spark, Mesos or YARN cluster URL, or a special "local[*]" string to run in local mode. In practice, when running on a cluster, you will not want to hardcode master in the program, but rather launch the application with spark-submit and receive it there. However, for local testing and unit tests, you can pass "local[*]" to run Spark Streaming in-process. Note that this internally creates a JavaSparkContext (starting point of all Spark functionality) which can be accessed as ssc.sparkContext.

The batch interval must be set based on the latency requirements of your application and available cluster resources. See the Performance Tuning section for more details.

A JavaStreamingContext object can also be created from an existing JavaSparkContext.

```
import org.apache.spark.streaming.api.java.*;

JavaSparkContext sc = ...   //existing JavaSparkContext
JavaStreamingContext ssc = new JavaStreamingContext(sc, Durations.seconds(1));
```

Python

A StreamingContext object can be created from a SparkContext object.

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext(master, appName)
ssc = StreamingContext(sc, 1)
```

The appName parameter is a name for your application to show on the cluster UI. master is a Spark, Mesos or YARN cluster URL, or a special "local[*]" string to run in local mode. In practice, when running on a cluster, you will not want to hardcode master in the program, but rather launch the application with spark-submit and receive it there. However, for local testing and unit tests, you can pass "local[*]" to run Spark Streaming in-process (detects the number of cores in the local system).

The batch interval must be set based on the latency requirements of your application and available cluster resources. See the Performance Tuning section for more details.

StreamingContext 对象创建后，你还需要如下步骤：

1. 创建 DStream 对象，并定义好输入数据源。

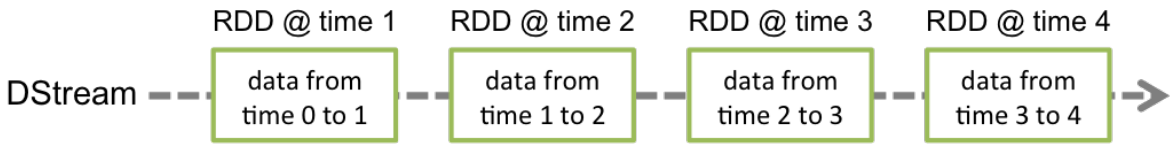
2. 基于数据源 DStream 定义好计算逻辑和输出。
3. 调用 streamingContext.start() 启动接收并处理数据。
4. 调用 streamingContext.awaitTermination() 等待流式处理结束（不管是手动结束，还是发生异常错误）
5. 你可以主动调用 streamingContext.stop() 来手动停止处理流程。

需要关注的重点:

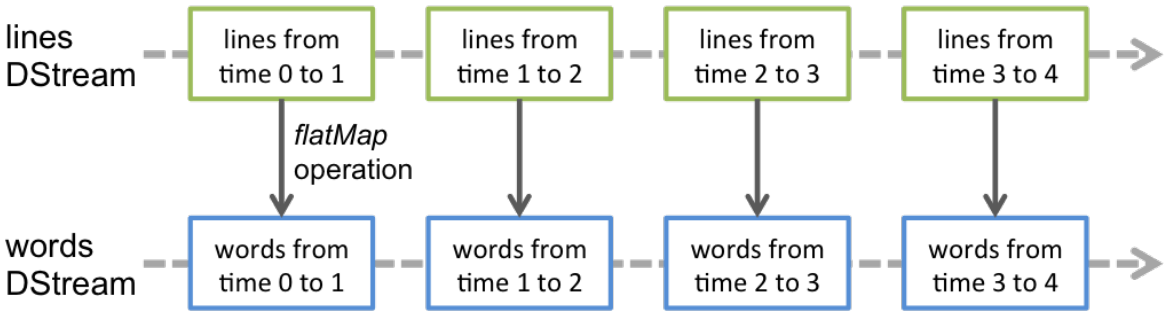
- 一旦 streamingContext 启动，就不能再对其计算逻辑进行添加或修改。
- 一旦 streamingContext 被 stop 掉，就不能 restart。
- 单个 JVM 虚机 同一时间只能包含一个 active 的 StreamingContext。
- StreamingContext.stop() 也会把关联的 SparkContext 对象 stop 掉，如果不想把 SparkContext 对象也 stop 掉，可以将StreamingContext.stop 的可选参数 stopSparkContext 设为false。
- 一个 SparkContext 对象可以和多个 StreamingContext 对象关联，只要先对前一个StreamingContext.stop(sparkContext=false)，然后再创建新的StreamingContext对象即可。

离散数据流(DStreams)

离散数据流（DStream）是 Spark Streaming 最基本的抽象。它代表了一种连续的数据流，要么从某种数据源提取数据，要么从其他数据流映射转换而来。DStream 内部是由一系列连续的RDD组成的，每个RDD都是不可变、分布式的数据集（详见Spark编程指南 – Spark Programming Guide）。每个 RDD 都包含了特定时间间隔内的一批数据，如下图所示：



任何作用于 DStream 的算子，其实都会被转化为对其内部 RDD 的操作。例如，在前面的例子中，我们将 lines 这个 DStream 转成 words DStream 对象，其实作用于 lines 上的 flatMap 算子，会施加于 lines 中的每个 RDD 上，并生成新的对应的 RDD，而这些新生成的 RDD 对象就组成了 words 这个 DStream 对象。其过程如下图所示：



底层的 RDD 转换仍然是由 Spark 引擎来计算。DStream 的算子将这些细节隐藏了起来，并为开发者提供了更为方便的高级API。后续会详细讨论这些高级算子。

输入DStream和接收器

输入 DStream 代表从某种流式数据源流入的数据流。在之前的例子里，lines 对象就是输入 DStream，它代表从 netcat server收到的数据流。每个输入DStream（除文件数据流外）都和一个接收器（Receiver – Scala doc, Java doc）相关联，而接收器则是专门从数据源拉取数据到内存中的对象。

Spark Streaming 主要提供两种内建的流式数据源:

- 基础数据源（Basic sources）：在 StreamingContext API 中可直接使用的源，如：文件系统，套接字连接或者Akka actor。
- 高级数据源（Advanced sources）：需要依赖额外工具类的源，如：Kafka、Flume、Kinesis、Twitter等数据源。这些数据源都需要增加额外的依赖，详见依赖链接（linking）这一节。

本节中，我们将会从每种数据源中挑几个继续深入讨论。

注意： 如果你需要同时从多个数据源拉取数据，那么你就需要创建多个 DStream 对象（详见后续的性能调优这一小节）。多个 DStream 对象其实也就同时创建了多个数据流接收器。但是请注意，Spark的 worker/executor 都是长期运行的，因此它们都会各自占用一个分配给 Spark Streaming 应用的 CPU。所以，在运行nSpark Streaming 应用的时候，需要注意分配足够的CPU core（本地运行时，需要足够的线程）来处理接收到的数据，同时还要足够的CPU core来运行这些接收器。

要点

- 如果本地运行 Spark Streaming 应用，记得不能将 master 设为 ”local” 或 “local[1]”。这两个值都只会在本地启动一个线程。而如果此时你使用一个包含接收器（如：套接字、Kafka、Flume等）的输入DStream，那么这一个线程只能用于运行这个接收器，而处理数据的逻辑就没有线程来执行了。因此，本地运行时，一定要将 master 设为 ”local[n]”，其中 n > 接收器的个数（有关master的详情请参考Spark Properties）。
- 将 Spark Streaming 应用置于集群中运行时，同样，分配给该应用的 CPU core 数必须大于接收器的总数。否则，该应用就只会接收数据，而不会处理数据。

基础数据源

前面的小栗子中，我们已经看到，使用ssc.socketTextStream(...) 可以从一个TCP连接中接收文本数据。而除了TCP套接字外，StreamingContext API 还支持从文件或者Akka actor中拉取数据。 * 文件数据流（File Streams）：可以从任何兼容HDFS API（包括：HDFS、S3、NFS等）的文件系统，创建方式如下：

Scala

```
streamingContext.fileStream[KeyClass, ValueClass,␣
↪InputFormatClass](dataDirectory)

**Java**
```

```
streamingContext.fileStream<KeyClass, ValueClass, InputFormatClass>
↪(dataDirectory);

**Python**
```

```
streamingContext.textFileStream(dataDirectory)
```

- **Spark Streaming将监视该dataDirectory目录，并处理该目录下任何新建的文件（目前还不支持嵌套目录）。注意：**
  - 各个文件数据格式必须一致。
  - dataDirectory中的文件必须通过moving或者renaming来创建。
  - 一旦文件move进dataDirectory之后，就不能再改动。所以如果这个文件后续还有写入，这些新写入的数据不会被读取。
- Python API fileStream目前暂时不可用，Python目前只支持textFileStream。另外，文件数据流不是基于接收器的，所以不需要为其单独分配一个CPU core。对于简单的文本文件，更简单的方式是调用streamingContext.textFileStream(dataDirectory)。
- 基于自定义Actor的数据流（Streams based on Custom Actors）：DStream可以由Akka actor创建得到，只需调用 streamingContext.actorStream(actorProps, actor-name)。详见自定义接收器（Custom Receiver Guide）。actorStream暂时不支持Python API。
- RDD队列数据流（Queue of RDDs as a Stream）：如果需要测试Spark Streaming应用，你可以创建一个基于一批RDD的DStream对象，只需调用 streamingContext.queueStream(queueOfRDDs)。RDD会被一个个依次推入队列，而DStream则会依次以数据流形式处理这些RDD的数据。

关于套接字、文件以及Akka actor数据流更详细信息，请参考相关文档：StreamingContext for Scala,JavaStreamingContext for Java, and StreamingContext for Python。

高级数据源

Python API 自 Spark 1.6.1 起，Kafka、Kinesis、Flume 和 MQTT 这些数据源将支持 Python。

使用这类数据源需要依赖一些额外的代码库，有些依赖还挺复杂的（如：Kafka、Flume）。因此为了减少依赖项版本冲突问题，各个数据源DStream的相关功能被分割到不同的代码包中，只有用到的时候才需要链接打包进来。例如，如果你需要使用Twitter的tweets作为数据源，你需要以下步骤： 1. Linking: 将spark-streaming-twitter_2.10工件加入到SBT/Maven项目依赖中。 2. Programming: 导入TwitterUtils class，然后调用TwitterUtils.createStream 创建一个DStream，具体代码见下放。 3. Deploying: 生成一个uber Jar包，并包含其所有依赖项（包括 spark-streaming-twitter_2.10及其自身的依赖树），再部署这个Jar包。部署详情请参考部署这一节（Deploying section）。 * Scala * Java import org.apache.spark.streaming.twitter._

TwitterUtils.createStream(ssc, None)

注意，高级数据源在spark-shell中不可用，因此不能用spark-shell来测试基于高级数据源的应用。如果真有需要的话，你需要自行下载相应数据源的Maven工件及其依赖项，并将这些Jar包部署到spark-shell的classpath中。

下面列举了一些高级数据源:

- Kafka: Spark Streaming 2.2.1 可兼容 Kafka 0.8.2.1。详见 Kafka Integration Guide。
- Flume: Spark Streaming 2.2.1 可兼容 Flume 1.6.0 。详见Flume Integration Guide。
- Kinesis: Spark Streaming 2.2.1 可兼容 Kinesis Client Library 1.2.1。详见Kinesis Integration Guide。

自定义数据源

Python API 自定义数据源目前还不支持Python。

输入DStream也可以用自定义的方式创建。你需要做的只是实现一个自定义的接收器（receiver），以便从自定义的数据源接收数据，然后将数据推入Spark中。详情请参考自定义接收器指南（Custom Receiver Guide）。

接收器可靠性

从可靠性角度来划分，大致有两种数据源。其中，像Kafka、Flume这样的数据源，它们支持对所传输的数据进行确认。系统收到这类可靠数据源过来的数据，然后发出确认信息，这样就能够确保任何失败情况下，都不会丢数据。因此我们可以将接收器也相应地分为两类：

1. 可靠接收器（Reliable Receiver） – 可靠接收器会在成功接收并保存好Spark数据副本后，向可靠数据源发送确认信息。
2. 不可靠接收器（Unreliable Receiver） – 不可靠接收器不会发送任何确认信息。不过这种接收器常用语于不支持确认的数据源，或者不想引入数据确认的复杂性的数据源。

自定义接收器指南（Custom Receiver Guide）中详细讨论了如何写一个可靠接收器。

DStream支持的transformation算子

和 RDD 类似，DStream 也支持从输入 DStream 经过各种 transformation 算子映射成新的 DStream。DStream 支持很多 RDD 上常见的 transformation 算子，一些常用的见下表：

Transformation算子	用途
map(func)	返回会一个新的DStream，并将源DStream中每个元素通过func映射为新的元素
flatMap(func)	和map类似，不过每个输入元素不再是映射为一个输出，而是映射为0到多个输出
filter(func)	返回一个新的DStream，并包含源DStream中被func选中（func返回true）的元素
repartition(numPartitions)	更改DStream的并行度（增加或减少分区数）
union(otherStream)	返回新的DStream，包含源DStream和otherDStream元素的并集
count()	返回一个包含单元素RDDs的DStream，其中每个元素是源DStream中各个RDD中的元素个数
reduce(func)	返回一个包含单元素RDDs的DStream，其中每个元素是通过源RDD中各个RDD的元素经func（func输入两个参数并返回一个同类型结果数据）聚合得到的结果。func必须满足结合律，以便支持并行计算。
countByValue()	如果源DStream包含的元素类型为K，那么该算子返回新的DStream包含元素为(K, Long)键值对，其中K为源DStream各个元素，而Long为该元素出现的次数。
reduceByKey(func, [numTasks])	如果源DStream 包含的元素为 (K, V) 键值对，则该算子返回一个新的也包含(K, V)键值对的DStream，其中V是由func聚合得到的。注意：默认情况下，该算子使用Spark的默认并发任务数（本地模式为2，集群模式下由spark.default.parallelism 决定）。你可以通过可选参数numTasks来指定并发任务个数。
join(otherStream, [numTasks])	如果源DStream包含元素为(K, V)，同时otherDStream包含元素为(K, W)键值对，则该算子返回一个新的DStream，其中源DStream和otherDStream中每个K都对应一个 (K, (V, W))键值对元素。
cogroup(otherStream, [numTasks])	如果源DStream包含元素为(K, V)，同时otherDStream包含元素为(K, W)键值对，则该算子返回一个新的DStream，其中每个元素类型为包含(K, Seq[V], Seq[W])的tuple。
transform(func)	返回一个新的DStream，其包含的RDD为源RDD经过func操作后得到的结果。利用该算子可以对DStream施加任意的操作。
updateStateByKey(func)	返回一个包含新"状态"的DStream。源DStream中每个key及其对应的values会作为func的输入，而func可以用于对每个key的"状态"数据作任意的更新操作。

下面我们会挑几个transformation算子深入讨论一下。

updateStateByKey算子

updateStateByKey 算子支持维护一个任意的状态。要实现这一点，只需要两步：

1. 定义状态 – 状态数据可以是任意类型。
2. 定义状态更新函数 – 定义好一个函数，其输入为数据流之前的状态和新的数据流数据，且可其更新步骤1中定义的输入数据流的状态。

在每一个批次数据到达后，Spark都会调用状态更新函数，来更新所有已有key（不管key是否存在于本批次中）的状态。如果状态更新函数返回None，则对应的键值对会被删除。

举例如下。假设你需要维护一个流式应用，统计数据流中每个单词的出现次数。这里将各个单词的出现次数这个整型数定义为状态。我们接下来定义状态更新函数如下：

Scala

```
def updateFunction(newValues: Seq[Int], runningCount: Option[Int]): Option[Int] = {
    val newCount = ...  // 将新的计数值和之前的状态值相加，得到新的计数值
```

```
    Some(newCount)
}
```

该状态更新函数可以作用于一个包括(word, 1) 键值对的DStream上（见本文开头的小栗子）。

```
val runningCounts = pairs.updateStateByKey[Int](updateFunction _)
```

该状态更新函数会为每个单词调用一次，且相应的newValues是一个包含很多个"1"的数组（这些1来自于(word,1)键值对），而runningCount包含之前该单词的计数。本例的完整代码请参考 StatefulNetworkWordCount.scala。

Java

```
Function2<List<Integer>, Optional<Integer>, Optional<Integer>> updateFunction =
  (values, state) -> {
    Integer newSum = ...  // add the new values with the previous running count to␣
↪get the new count
    return Optional.of(newSum);
  };
```

This is applied on a DStream containing words (say, the pairs DStream containing (word, 1) pairs in the quick example).

```
JavaPairDStream<String, Integer> runningCounts = pairs.
↪updateStateByKey(updateFunction);
```

The update function will be called for each word, with newValues having a sequence of 1's (from the (word, 1) pairs) and the runningCount having the previous count. For the complete Java code, take a look at the example JavaStatefulNetworkWordCount.java.

Python

```
def updateFunction(newValues, runningCount):
    if runningCount is None:
        runningCount = 0
    return sum(newValues, runningCount)  # add the new values with the previous␣
↪running count to get the new count
```

This is applied on a DStream containing words (say, the pairs DStream containing (word, 1) pairs in the earlier example).

```
runningCounts = pairs.updateStateByKey(updateFunction)
```

The update function will be called for each word, with newValues having a sequence of 1's (from the (word, 1) pairs) and the runningCount having the previous count. For the complete Python code, take a look at the example stateful_network_wordcount.py.

注意，调用 updateStateByKey 前需要配置检查点目录，后续对此有详细的讨论，见检查点（checkpointing）这节。

transform 算子

transform算子（及其变体transformWith）可以支持任意的RDD到RDD的映射操作。也就是说，你可以用tranform算子来包装任何DStream API所不支持的RDD算子。例如，将DStream每个批次中的RDD和另一个Dataset进行关联（join）操作，这个功能DStream API并没有直接支持。不过你可以用transform来实现这个

功能，可见transform其实为DStream提供了非常强大的功能支持。比如说，你可以用事先算好的垃圾信息，对DStream进行实时过滤。

Scala

```
val spamInfoRDD = ssc.sparkContext.newAPIHadoopRDD(...) // 包含垃圾信息的RDD

val cleanedDStream = wordCounts.transform(rdd => {
  rdd.join(spamInfoRDD).filter(...) // 将DStream中的RDD和spamInfoRDD关联，并实时过滤垃圾数据
  ...
})
```

Java

```
import org.apache.spark.streaming.api.java.*;
// RDD containing spam information
JavaPairRDD<String, Double> spamInfoRDD = jssc.sparkContext().newAPIHadoopRDD(...);

JavaPairDStream<String, Integer> cleanedDStream = wordCounts.transform(rdd -> {
  rdd.join(spamInfoRDD).filter(...); // join data stream with spam information to do␣
↪data cleaning
  ...
});
```

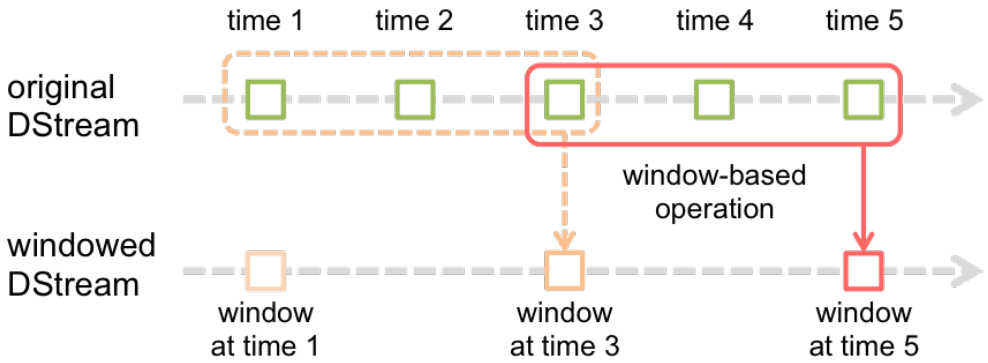
Python

```
spamInfoRDD = sc.pickleFile(...)  # RDD containing spam information

# join data stream with spam information to do data cleaning
cleanedDStream = wordCounts.transform(lambda rdd: rdd.join(spamInfoRDD).filter(...))
```

注意： 这里transform包含的算子，其调用时间间隔和批次间隔是相同的。所以你可以基于时间改变对RDD的操作，如：在不同批次，调用不同的RDD算子，设置不同的RDD分区或者广播变量等。

基于窗口（window）的算子 Spark Streaming 同样也提供基于时间窗口的计算，也就是说，你可以对某一个滑动时间窗内的数据施加特定tranformation算子。如下图所示：



如上图所示，每次窗口滑动时，源 DStream 中落入窗口的 RDDs 就会被合并成新的 windowed DStream。在上图的例子中，这个操作会施加于3个RDD单元，而滑动距离是2个RDD单元。由此可以得出任何窗口相关操作都需要指定一下两个参数:

- （窗口长度）window length – 窗口覆盖的时间长度（上图中为3）
- （滑动距离）sliding interval – 窗口启动的时间间隔（上图中为2）

注意，这两个参数都必须是 DStream 批次间隔（上图中为1）的整数倍.

下面咱们举个栗子。假设，你需要扩展前面的那个小栗子，你需要每隔10秒统计一下前30秒内的单词计数。为此，我们需要在包含(word, 1)键值对的DStream上，对最近30秒的数据调用reduceByKey算子。不过这些都可以简单地用一个 reduceByKeyAndWindow搞定。

Scala

```
// 每隔10秒归约一次最近30秒的数据
val windowedWordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b),␣
↪Seconds(30), Seconds(10))
```

Java

```
// Reduce last 30 seconds of data, every 10 seconds
JavaPairDStream<String, Integer> windowedWordCounts = pairs.reduceByKeyAndWindow((i1,␣
↪i2) -> i1 + i2, Durations.seconds(30), Durations.seconds(10));
```

Python

```
# Reduce last 30 seconds of data, every 10 seconds
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x -␣
↪y, 30, 10)
```

以下列出了常用的窗口算子。所有这些算子都有前面提到的那两个参数 – 窗口长度 和 滑动距离。

Transformation窗口算子	用途
window(windowLength, slideInterval)	将源DStream窗口化，并返回转化后的DStream
countByWindow(windowLength,slideInterval)	返回数据流在一个滑动窗口内的元素个数
reduceByWindow(func, windowLength,slideInterval)	基于数据流在一个滑动窗口内的元素，用func做聚合，返回一个单元素数据流。func必须满足结合律，以便支持并行计算。
reduceByKeyAndWindow(func,windowLength,slideInterval, [numTasks])	基于(K, V)键值对DStream，将一个滑动窗口内的数据进行聚合，返回一个新的包含(K,V)键值对的DStream，其中每个value都是各个key经过func聚合后的结果。注意：如果不指定numTasks，其值将使用Spark的默认并行任务数（本地模式下为2，集群模式下由spark.default.parallelism决定）。当然，你也可以通过numTasks来指定任务个数。
reduceByKeyAndWindow(func, invFunc,windowLength,slideInterval, [numTasks])	和前面的reduceByKeyAndWindow() 类似，只是这个版本会用之前滑动窗口计算结果，递增地计算每个窗口的归约结果。当新的数据进入窗口时，这些values会被输入func做归约计算，而这些数据离开窗口时，对应的这些values又会被输入 invFunc 做”反归约”计算。举个简单的例子，就是把新进入窗口数据中各个单词个数“增加”到各个单词统计结果上，同时把离开窗口数据中各个单词的统计个数从相应的统计结果中“减掉”。不过，你的自己定义好”反归约”函数，即：该算子不仅有归约函数（见参数func），还得有一个对应的”反归约”函数（见参数中的 invFunc）。和前面的reduceByKeyAndWindow() 类似，该算子也有一个可选参数numTasks来指定并行任务数。注意，这个算子需要配置好检查点（checkpointing）才能用。
countByValueAndWindow(windowLength,slideInterval, [numTasks])	基于包含(K, V)键值对的DStream，返回新的包含(K, Long)键值对的DStream。其中的Long value都是滑动窗口内key出现次数的计数。和前面的reduceByKeyAndWindow() 类似，该算子也有一个可选参数numTasks来指定并行任务数。

Join 算子

最后，值得一提的是，你在Spark Streaming中做各种关联（join）操作非常简单。

流-流（Stream-stream）关联

一个数据流可以和另一个数据流直接关联。

Scala

```
val stream1: DStream[String, String] = ...
val stream2: DStream[String, String] = ...
```

```scala
val joinedStream = stream1.join(stream2)
```

Java

```java
JavaPairDStream<String, String> stream1 = ...
JavaPairDStream<String, String> stream2 = ...
JavaPairDStream<String, Tuple2<String, String>> joinedStream = stream1.join(stream2);
```

Python

```python
stream1 = ...
stream2 = ...
joinedStream = stream1.join(stream2)
```

上面代码中，stream1的每个批次中的RDD会和stream2相应批次中的RDD进行join。同样，你可以类似地使用 leftOuterJoin, rightOuterJoin, fullOuterJoin 等。此外，你还可以基于窗口来join不同的数据流，其实现也很简单，如下；）

Scala

```scala
val windowedStream1 = stream1.window(Seconds(20))
val windowedStream2 = stream2.window(Minutes(1))
val joinedStream = windowedStream1.join(windowedStream2)
```

Java

```java
JavaPairDStream<String, String> windowedStream1 = stream1.window(Durations.
↪seconds(20));
JavaPairDStream<String, String> windowedStream2 = stream2.window(Durations.
↪minutes(1));
JavaPairDStream<String, Tuple2<String, String>> joinedStream = windowedStream1.
↪join(windowedStream2);
```

Python

```python
windowedStream1 = stream1.window(20)
windowedStream2 = stream2.window(60)
joinedStream = windowedStream1.join(windowedStream2)
```

流-数据集(stream-dataset)关联

其实这种情况已经在前面的DStream.transform算子中介绍过了，这里再举个基于滑动窗口的例子。

Scala

```scala
val dataset: RDD[String, String] = ...
val windowedStream = stream.window(Seconds(20))...
val joinedStream = windowedStream.transform { rdd => rdd.join(dataset) }
```

Java

```java
JavaPairRDD<String, String> dataset = ...
JavaPairDStream<String, String> windowedStream = stream.window(Durations.seconds(20));
JavaPairDStream<String, String> joinedStream = windowedStream.transform(rdd -> rdd.
↪join(dataset));
```

Python

```
dataset = ... # some RDD
windowedStream = stream.window(20)
joinedStream = windowedStream.transform(lambda rdd: rdd.join(dataset))
```

实际上，在上面代码里，你可以动态地该表join的数据集（dataset）。传给tranform算子的操作函数会在每个批次重新求值，所以每次该函数都会用最新的dataset值，所以不同批次间你可以改变dataset的值。

完整的DStream transformation算子列表见API文档。Scala请参考 DStream 和 PairDStreamFunctions. Java请参考 JavaDStream 和 JavaPairDStream. Python见 DStream。

DStream输出算子

输出算子可以将 DStream 的数据推送到外部系统，如：数据库或者文件系统。因为输出算子会将最终完成转换的数据输出到外部系统，因此只有输出算子调用时，才会真正触发 DStream transformation 算子的真正执行（这一点类似于RDD 的action算子）。目前所支持的输出算子如下表：

输出算子	用途
print()	在驱动器（driver）节点上打印DStream每个批次中的头十个元素。Python API 对应的Python API为 pprint()
saveAsTextFiles(prefix, [suffix])	将DStream的内容保存到文本文件。每个批次一个文件，各文件命名规则为 “prefix-TIME_IN_MS[.suffix]”
saveAsObjectFiles(prefix, [suffix])	将DStream内容以序列化Java对象的形式保存到顺序文件中。每个批次一个文件，各文件命名规则为 “prefix-TIME_IN_MS[.suffix]”Python API 暂不支持Python
saveAsHadoopFiles(prefix, [suffix])	将DStream内容保存到Hadoop文件中。每个批次一个文件，各文件命名规则为 “prefix-TIME_IN_MS[.suffix]”Python API 暂不支持Python
foreachRDD(func)	这是最通用的输出算子了，该算子接收一个函数func，func将作用于DStream的每个RDD上。func应该实现将每个RDD的数据推到外部系统中，比如：保存到文件或者写到数据库中。注意，func函数是在streaming应用的驱动器进程中执行的，所以如果其中包含RDD的action算子，就会触发对DStream中RDDs的实际计算过程。

使用foreachRDD的设计模式

DStream.foreachRDD是一个非常强大的原生工具函数，用户可以基于此算子将DStream数据推送到外部系统中。不过用户需要了解如何正确而高效地使用这个工具。以下列举了一些常见的错误。

通常，对外部系统写入数据需要一些连接对象（如：远程server的TCP连接），以便发送数据给远程系统。因此，开发人员可能会不经意地在Spark驱动器（driver）进程中创建一个连接对象，然后又试图在Spark worker节点上使用这个连接。如下例所示：

Scala

```
dstream.foreachRDD { rdd =>
  val connection = createNewConnection()  // 这行在驱动器（driver）进程执行
  rdd.foreach { record =>
    connection.send(record) // 而这行将在worker节点上执行
  }
}
```

Java

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

Python

```
def sendRecord(rdd):
    connection = createNewConnection()  # executed at the driver
    rdd.foreach(lambda record: connection.send(record))
    connection.close()

dstream.foreachRDD(sendRecord)
```

这段代码是错误的，因为它需要把连接对象序列化，再从驱动器节点发送到worker节点。而这些连接对象通常都是不能跨节点（机器）传递的。比如，连接对象通常都不能序列化，或者在另一个进程中反序列化后再次初始化（连接对象通常都需要初始化，因此从驱动节点发到worker节点后可能需要重新初始化）等。解决此类错误的办法就是在worker节点上创建连接对象。

然而，有些开发人员可能会走到另一个极端 – 为每条记录都创建一个连接对象，例如：

Scala

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

Java

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

Python

```
def sendRecord(record):
    connection = createNewConnection()
    connection.send(record)
    connection.close()

dstream.foreachRDD(lambda rdd: rdd.foreach(sendRecord))
```

一般来说，连接对象是有时间和资源开销限制的。因此，对每条记录都进行一次连接对象的创建和销毁会增加很多不必要的开销，同时也大大减小了系统的吞吐量。一个比较好的解决方案是使用rdd.foreachPartition – 为RDD的每个分区创建一个单独的连接对象，示例如下：

Scala

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

Java

```
dstream.foreachRDD(rdd -> {
  rdd.foreachPartition(partitionOfRecords -> {
    Connection connection = createNewConnection();
    while (partitionOfRecords.hasNext()) {
      connection.send(partitionOfRecords.next());
    }
    connection.close();
  });
});
```

Python

```
def sendPartition(iter):
    connection = createNewConnection()
    for record in iter:
        connection.send(record)
    connection.close()

dstream.foreachRDD(lambda rdd: rdd.foreachPartition(sendPartition))
```

这样一来，连接对象的创建开销就摊到很多条记录上了。

最后，还有一个更优化的办法，就是在多个RDD批次之间复用连接对象。开发者可以维护一个静态连接池来保存连接对象，以便在不同批次的多个RDD之间共享同一组连接对象，示例如下：

Scala

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    // ConnectionPool 是一个静态的、懒惰初始化的连接池
    val connection = ConnectionPool.getConnection()
    partitionOfRecords.foreach(record => connection.send(record))
    ConnectionPool.returnConnection(connection)  // 将连接返还给连接池，以便后续复用之
  }
}
```

Java

```
dstream.foreachRDD(rdd -> {
  rdd.foreachPartition(partitionOfRecords -> {
    // ConnectionPool is a static, lazily initialized pool of connections
    Connection connection = ConnectionPool.getConnection();
    while (partitionOfRecords.hasNext()) {
      connection.send(partitionOfRecords.next());
    }
    ConnectionPool.returnConnection(connection); // return to the pool for future␣
↪reuse
  });
});
```

Python

```
def sendPartition(iter):
    # ConnectionPool is a static, lazily initialized pool of connections
    connection = ConnectionPool.getConnection()
    for record in iter:
        connection.send(record)
    # return to the pool for future reuse
    ConnectionPool.returnConnection(connection)

dstream.foreachRDD(lambda rdd: rdd.foreachPartition(sendPartition))
```

注意：连接池中的连接应该是懒惰创建的，并且有确定的超时时间，超时后自动销毁。这个实现应该是目前发送数据最高效的实现方式。

其他要点：

- DStream的转化执行也是懒惰的，需要输出算子来触发，这一点和RDD的懒惰执行由action算子触发很类似。特别地，DStream输出算子中包含的RDD action算子会强制触发对所接收数据的处理。因此，如果你的Streaming应用中没有输出算子，或者你用了dstream.foreachRDD(func)却没有在func中调用RDD action算子，那么这个应用只会接收数据，而不会处理数据，接收到的数据最后只是被简单地丢弃掉了。
- 默认地，输出算子只能一次执行一个，且按照它们在应用程序代码中定义的顺序执行。

DataFrame 和 SQL 算子

在Streaming应用中可以调用DataFrames and SQL来处理流式数据。开发者可以用通过StreamingContext中的SparkContext对象来创建一个SQLContext，并且，开发者需要确保一旦驱动器（driver）故障恢复后，该SQLContext对象能重新创建出来。同样，你还是可以使用懒惰创建的单例模式来实例化SQLContext，如下面的代码所示，这里我们将最开始的那个小栗子做了一些修改，使用DataFrame和SQL来统计单词计数。其实就是，将每个RDD都转化成一个DataFrame，然后注册成临时表，再用SQL查询这些临时表。

Scala

```
/** streaming应用中调用DataFrame算子 */

val words: DStream[String] = ...

words.foreachRDD { rdd =>

  // 获得SQLContext单例
  val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
  import sqlContext.implicits._

  // 将RDD[String] 转为 DataFrame
  val wordsDataFrame = rdd.toDF("word")

  // DataFrame注册为临时表
  wordsDataFrame.registerTempTable("words")

  // 再用SQL语句查询，并打印出来
  val wordCountsDataFrame =
    sqlContext.sql("select word, count(*) as total from words group by word")
```

```
  wordCountsDataFrame.show()
}
```

这里有完整代码：source code。

Java

```
/** Java Bean class for converting RDD to DataFrame */
public class JavaRow implements java.io.Serializable {
  private String word;

  public String getWord() {
    return word;
  }

  public void setWord(String word) {
    this.word = word;
  }
}

...

/** DataFrame operations inside your streaming program */

JavaDStream<String> words = ...

words.foreachRDD((rdd, time) -> {
  // Get the singleton instance of SparkSession
  SparkSession spark = SparkSession.builder().config(rdd.sparkContext().getConf()).
↪getOrCreate();

  // Convert RDD[String] to RDD[case class] to DataFrame
  JavaRDD<JavaRow> rowRDD = rdd.map(word -> {
    JavaRow record = new JavaRow();
    record.setWord(word);
    return record;
  });
  DataFrame wordsDataFrame = spark.createDataFrame(rowRDD, JavaRow.class);

  // Creates a temporary view using the DataFrame
  wordsDataFrame.createOrReplaceTempView("words");

  // Do word count on table using SQL and print it
  DataFrame wordCountsDataFrame =
    spark.sql("select word, count(*) as total from words group by word");
  wordCountsDataFrame.show();
});
```

这里有完整代码：source code。

Python

```
# Lazily instantiated global instance of SparkSession
def getSparkSessionInstance(sparkConf):
    if ("sparkSessionSingletonInstance" not in globals()):
        globals()["sparkSessionSingletonInstance"] = SparkSession \
            .builder \
            .config(conf=sparkConf) \
            .getOrCreate()
```

```
    return globals()["sparkSessionSingletonInstance"]

...

# DataFrame operations inside your streaming program

words = ... # DStream of strings

def process(time, rdd):
    print("========= %s =========" % str(time))
    try:
        # Get the singleton instance of SparkSession
        spark = getSparkSessionInstance(rdd.context.getConf())

        # Convert RDD[String] to RDD[Row] to DataFrame
        rowRdd = rdd.map(lambda w: Row(word=w))
        wordsDataFrame = spark.createDataFrame(rowRdd)

        # Creates a temporary view using the DataFrame
        wordsDataFrame.createOrReplaceTempView("words")

        # Do word count on table using SQL and print it
        wordCountsDataFrame = spark.sql("select word, count(*) as total from words␣
↪group by word")
        wordCountsDataFrame.show()
    except:
        pass

words.foreachRDD(process)
```

这里有完整代码：source code。

你也可以在其他线程里执行SQL查询（异步查询，即：执行SQL查询的线程和运行StreamingContext的线程不同）。不过这种情况下，你需要确保查询的时候 StreamingContext 没有把所需的数据丢弃掉，否则StreamingContext有可能已将老的RDD数据丢弃掉了，那么异步查询的SQL语句也可能无法得到查询结果。举个栗子，如果你需要查询上一个批次的数据，但是你的SQL查询可能要执行5分钟，那么你就需要StreamingContext至少保留最近5分钟的数据：streamingContext.remember(Minutes(5))（这是Scala为例，其他语言差不多）

更多DataFrame和SQL的文档见这里： DataFrames and SQL

MLlib 算子

MLlib 提供了很多机器学习算法。首先，你需要关注的是流式计算相关的机器学习算法（如：Streaming Linear Regression, Streaming KMeans），这些流式算法可以在流式数据上一边学习训练模型，一边用最新的模型处理数据。除此以外，对更多的机器学习算法而言，你需要离线训练这些模型，然后将训练好的模型用于在线的流式数据。详见MLlib。

缓存/持久化

和RDD类似，DStream也支持将数据持久化到内存中。只需要调用 DStream的persist() 方法，该方法内部会自动调用DStream中每个RDD的persist方法进而将数据持久化到内存中。这对于可能需要计算很多次的DStream非常有用（例如：对于同一个批数据调用多个算子）。对于基于滑动窗口的算子，如：reduceByWindow和reduceByKeyAndWindow，或者有状态的算子，如：updateStateByKey，数据持久化

就更重要了。因此，滑动窗口算子产生的DStream对象默认会自动持久化到内存中（不需要开发者调用persist）。

对于从网络接收数据的输入数据流（如：Kafka、Flume、socket等），默认的持久化级别会将数据持久化到两个不同的节点上互为备份副本，以便支持容错。

注意，与RDD不同的是，DStream的默认持久化级别是将数据序列化到内存中。进一步的讨论见性能调优这一小节。关于持久化级别（或者存储级别）的更详细说明见Spark编程指南（Spark Programming Guide）。

检查点

一般来说Streaming 应用都需要7*24小时长期运行，所以必须对一些与业务逻辑无关的故障有很好的容错（如：系统故障、JVM崩溃等）。对于这些可能性，Spark Streaming 必须在检查点保存足够的信息到一些可容错的外部存储系统中，以便能够随时从故障中恢复回来。所以，检查点需要保存以下两种数据：

- **元数据检查点（Metadata checkpointing） – 保存流式计算逻辑的定义信息到外部可容错存储系统（如：HDFS）。主**
  - Configuration – 创建Streaming应用程序的配置信息。
  - DStream operations – 定义流式处理逻辑的DStream操作信息。
  - Incomplete batches – 已经排队但未处理完的批次信息。
- 数据检查点（Data checkpointing） – 将生成的RDD保存到可靠的存储中。这对一些需要跨批次组合数据或者有状态的算子来说很有必要。在这种转换算子中，往往新生成的RDD是依赖于前几个批次的RDD，因此随着时间的推移，有可能产生很长的依赖链条。为了避免在恢复数据的时候需要恢复整个依赖链条上所有的数据，检查点需要周期性地保存一些中间RDD状态信息，以斩断无限制增长的依赖链条和恢复时间。

总之，元数据检查点主要是为了恢复驱动器节点上的故障，而数据或RDD检查点是为了支持对有状态转换操作的恢复。

何时启用检查点

如果有以下情况出现，你就必须启用检查点了：

- 使用了有状态的转换算子（Usage of stateful transformations） – 不管是用了 updateStateByKey 还是用了 reduceByKeyAndWindow（有”反归约”函数的那个版本），你都必须配置检查点目录来周期性地保存RDD检查点。
- 支持驱动器故障中恢复（Recovering from failures of the driver running the application） – 这时候需要元数据检查点以便恢复流式处理的进度信息。

注意，一些简单的流式应用，如果没有用到前面所说的有状态转换算子，则完全可以不开启检查点。不过这样的话，驱动器（driver）故障恢复后，有可能会丢失部分数据（有些已经接收但还未处理的数据可能会丢失）。不过通常这点丢失时可接受的，很多Spark Streaming应用也是这样运行的。对非Hadoop环境的支持未来还会继续改进。

如何配置检查点

检查点的启用，只需要设置好保存检查点信息的检查点目录即可，一般会会将这个目录设为一些可容错的、可靠性较高的文件系统（如：HDFS、S3等）。开发者只需要调用 streamingContext.checkpoint(checkpointDirectory)。设置好检查点，你就可以使用前面提到的有状态转换算子了。另外，如果你需要你的应用能够支持从驱动器故障中恢复，你可能需要重写部分代码，实现以下行为：

- 如果程序是首次启动，就需要new一个新的StreamingContext，并定义好所有的数据流处理，然后调用StreamingContext.start()。
- 如果程序是故障后重启，就需要从检查点目录中的数据中重新构建StreamingContext对象。

不过这个行为可以用StreamingContext.getOrCreate来实现，示例如下：

Scala

```
// 首次创建StreamingContext并定义好数据流处理逻辑
def functionToCreateContext(): StreamingContext = {
    val ssc = new StreamingContext(...)   // 新建一个StreamingContext对象
    val lines = ssc.socketTextStream(...) // 创建DStreams
    ...
    ssc.checkpoint(checkpointDirectory)   // 设置好检查点目录
    ssc
}

// 创建新的StreamingContext对象，或者从检查点构造一个
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. ...

// 启动StreamingContext对象
context.start()
context.awaitTermination()
```

如果 checkpointDirectory 目录存在，则context对象会从检查点数据重新构建出来。如果该目录不存在（如：首次运行），则 functionToCreateContext 函数会被调用，创建一个新的StreamingContext对象并定义好DStream数据流。完整的示例请参见RecoverableNetworkWordCount，这个例子会将网络数据中的单词计数统计结果添加到一个文件中。

除了使用getOrCreate之外，开发者还需要确保驱动器进程能在故障后重启。这一点只能由应用的部署环境基础设施来保证。进一步的讨论见部署（Deployment）这一节。

另外需要注意的是，RDD检查点会增加额外的保存数据的开销。这可能会导致数据流的处理时间变长。因此，你必须仔细的调整检查点间隔时间。如果批次间隔太小（比如：1秒），那么对每个批次保存检查点数据将大大减小吞吐量。另一方面，检查点保存过于频繁又会导致血统信息和任务个数的增加，这同样会影响系统性能。对于需要RDD检查点的有状态转换算子，默认的间隔是批次间隔的整数倍，且最小10秒。开发人员可以这样来自定义这个间隔：dstream.checkpoint(checkpointInterval)。一般推荐设为批次间隔时间的5~10倍。

累加器, 广播变量以及检查点

Accumulators and Broadcast variables cannot be recovered from checkpoint in Spark Streaming. If you enable checkpointing and use Accumulators or Broadcast variables as well, you'll have to create lazily instantiated singleton instances for Accumulators and Broadcast variables so that they can be re-instantiated after the driver restarts on failure. This is shown in the following example.

Scala

```
object WordBlacklist {

  @volatile private var instance: Broadcast[Seq[String]] = null
```

```
  def getInstance(sc: SparkContext): Broadcast[Seq[String]] = {
    if (instance == null) {
      synchronized {
        if (instance == null) {
          val wordBlacklist = Seq("a", "b", "c")
          instance = sc.broadcast(wordBlacklist)
        }
      }
    }
    instance
  }
}

object DroppedWordsCounter {

  @volatile private var instance: LongAccumulator = null

  def getInstance(sc: SparkContext): LongAccumulator = {
    if (instance == null) {
      synchronized {
        if (instance == null) {
          instance = sc.longAccumulator("WordsInBlacklistCounter")
        }
      }
    }
    instance
  }
}

wordCounts.foreachRDD { (rdd: RDD[(String, Int)], time: Time) =>
  // Get or register the blacklist Broadcast
  val blacklist = WordBlacklist.getInstance(rdd.sparkContext)
  // Get or register the droppedWordsCounter Accumulator
  val droppedWordsCounter = DroppedWordsCounter.getInstance(rdd.sparkContext)
  // Use blacklist to drop words and use droppedWordsCounter to count them
  val counts = rdd.filter { case (word, count) =>
    if (blacklist.value.contains(word)) {
      droppedWordsCounter.add(count)
      false
    } else {
      true
    }
  }.collect().mkString("[", ", ", "]")
  val output = "Counts at time " + time + " " + counts
})
```

See the full source code.

Java

```
class JavaWordBlacklist {

  private static volatile Broadcast<List<String>> instance = null;

  public static Broadcast<List<String>> getInstance(JavaSparkContext jsc) {
    if (instance == null) {
      synchronized (JavaWordBlacklist.class) {
        if (instance == null) {
```

```
          List<String> wordBlacklist = Arrays.asList("a", "b", "c");
          instance = jsc.broadcast(wordBlacklist);
        }
      }
    }
    return instance;
  }
}

class JavaDroppedWordsCounter {

  private static volatile LongAccumulator instance = null;

  public static LongAccumulator getInstance(JavaSparkContext jsc) {
    if (instance == null) {
      synchronized (JavaDroppedWordsCounter.class) {
        if (instance == null) {
          instance = jsc.sc().longAccumulator("WordsInBlacklistCounter");
        }
      }
    }
    return instance;
  }
}

wordCounts.foreachRDD((rdd, time) -> {
  // Get or register the blacklist Broadcast
  Broadcast<List<String>> blacklist = JavaWordBlacklist.getInstance(new␣
↪JavaSparkContext(rdd.context()));
  // Get or register the droppedWordsCounter Accumulator
  LongAccumulator droppedWordsCounter = JavaDroppedWordsCounter.getInstance(new␣
↪JavaSparkContext(rdd.context()));
  // Use blacklist to drop words and use droppedWordsCounter to count them
  String counts = rdd.filter(wordCount -> {
    if (blacklist.value().contains(wordCount._1())) {
      droppedWordsCounter.add(wordCount._2());
      return false;
    } else {
      return true;
    }
  }).collect().toString();
  String output = "Counts at time " + time + " " + counts;
}
```

See the full source code.

Python

```
def getWordBlacklist(sparkContext):
    if ("wordBlacklist" not in globals()):
        globals()["wordBlacklist"] = sparkContext.broadcast(["a", "b", "c"])
    return globals()["wordBlacklist"]

def getDroppedWordsCounter(sparkContext):
    if ("droppedWordsCounter" not in globals()):
        globals()["droppedWordsCounter"] = sparkContext.accumulator(0)
    return globals()["droppedWordsCounter"]
```

```
def echo(time, rdd):
    # Get or register the blacklist Broadcast
    blacklist = getWordBlacklist(rdd.context)
    # Get or register the droppedWordsCounter Accumulator
    droppedWordsCounter = getDroppedWordsCounter(rdd.context)

    # Use blacklist to drop words and use droppedWordsCounter to count them
    def filterFunc(wordCount):
        if wordCount[0] in blacklist.value:
            droppedWordsCounter.add(wordCount[1])
            False
        else:
            True

    counts = "Counts at time %s %s" % (time, rdd.filter(filterFunc).collect())

wordCounts.foreachRDD(echo)
```

See the full source code.

部署应用程序

本节中将主要讨论一下如何部署Spark Streaming应用。

前提条件

要运行一个Spark Streaming 应用，你首先需要具备以下条件：

- 集群以及集群管理器 – 这是一般Spark应用的基本要求，详见 deployment guide。
- 给Spark应用打个JAR包 – 你需要将你的应用打成一个JAR包。如果使用spark-submit 提交应用，那么你不需要提供Spark和Spark Streaming的相关JAR包。但是，如果你使用了高级数据源（advanced sources – 如：Kafka、Flume、Twitter等），那么你需要将这些高级数据源相关的JAR包及其依赖一起打包并部署。例如，如果你使用了TwitterUtils，那么就必须将spark-streaming-twitter_2.10及其相关依赖都打到应用的JAR包中。
- 为执行器（executor）预留足够的内存 – 执行器必须配置预留好足够的内存，因为接受到的数据都得存在内存里。注意，如果某些窗口长度达到10分钟，那也就是说你的系统必须知道保留10分钟的数据在内存里。可见，到底预留多少内存是取决于你的应用处理逻辑的。
- 配置检查点 – 如果你的流式应用需要检查点，那么你需要配置一个Hadoop API兼容的可容错存储目录作为检查点目录，流式应用的信息会写入这个目录，故障恢复时会用到这个目录下的数据。详见前面的检查点小节。
- **配置驱动程序自动重启 – 流式应用自动恢复的前提就是，部署基础设施能够监控驱动器进程，并且能够在其故障时，**
  - Spark独立部署 – Spark独立部署集群可以支持将Spark应用的驱动器提交到集群的某个worker节点上运行。同时，Spark的集群管理器可以对该驱动器进程进行监控，一旦驱动器退出且返回非0值，或者因worker节点原始失败，Spark集群管理器将自动重启这个驱动器。详见Spark独立部署指南（Spark Standalone guide）。
- **YARN – YARN支持和独立部署类似的重启机制。详细请参考YARN的文档。**
  - Mesos – Mesos上需要用Marathon来实现这一功能。

- 配置WAL（write ahead log）- 从Spark 1.2起，我们引入了write ahead log来提高容错性。如果启用这个功能，则所有接收到的数据都会以write ahead log形式写入配置好的检查点目录中。这样就能确保数据零丢失（容错语义有详细的讨论）。用户只需将 spark.streaming.receiver.writeAheadLog 设为true。不过，这同样可能会导致接收器的吞吐量下降。不过你可以启动多个接收器并行接收数据，从而提升整体的吞吐量（more receivers in parallel）。另外，建议在启用WAL后禁用掉接收数据多副本功能，因为WAL其实已经是存储在一个多副本存储系统中了。你只需要把存储级别设为 StorageLevel.MEMORY_AND_DISK_SER。如果是使用S3（或者其他不支持flushing的文件系统）存储WAL，一定要记得启用这两个标识：spark.streaming.driver.writeAheadLog.closeFileAfterWrite 和 spark.streaming.receiver.writeAheadLog.closeFileAfterWrite。更详细请参考： Spark Streaming Configuration。
- 设置好最大接收速率 – 如果集群可用资源不足以跟上接收数据的速度，那么可以在接收器设置一下最大接收速率，即：每秒接收记录的条数。相关的主要配置有：spark.streaming.receiver.maxRate，如果使用Kafka Direct API 还需要设置 spark.streaming.kafka.maxRatePerPartition。从Spark 1.5起，我们引入了backpressure的概念来动态地根据集群处理速度，评估并调整该接收速率。用户只需将spark.streaming.backpressure.enabled设为true即可启用该功能。

升级应用程序代码

升级Spark Streaming应用程序代码，可以使用以下两种方式:

- 新的Streaming程序和老的并行跑一段时间，新程序完成初始化以后，再关闭老的。注意，这种方式适用于能同时发送数据到多个目标的数据源（即：数据源同时将数据发给新老两个Streaming应用程序）。
- 老程序能够优雅地退出（参考 StreamingContext.stop(...) or JavaStreamingContext.stop(...)），即：确保所收到的数据都已经处理完毕后再退出。然后再启动新的Streaming程序，而新程序将接着在老程序退出点上继续拉取数据。注意，这种方式需要数据源支持数据缓存（或者叫数据堆积，如：Kafka、Flume），因为在新旧程序交接的这个空档时间，数据需要在数据源处缓存。目前还不能支持从检查点重启，因为检查点存储的信息包含老程序中的序列化对象信息，在新程序中将其反序列化可能会出错。这种情况下，只能要么指定一个新的检查点目录，要么删除老的检查点目录。

应用程序监控

除了Spark自身的监控能力（monitoring capabilities）之外，对Spark Streaming还有一些额外的监控功能可用。如果实例化了StreamingContext，那么你可以在Spark web UI上看到多出了一个Streaming tab页，上面显示了正在运行的接收器（是否活跃，接收记录的条数，失败信息等）和处理完的批次信息（批次处理时间，查询延时等）。这些信息都可以用来监控streaming应用。

web UI上有两个度量特别重要:

- 批次处理耗时（Processing Time） – 处理单个批次耗时
- 批次调度延时（Scheduling Delay） -各批次在队列中等待时间（等待上一个批次处理完）

如果批次处理耗时一直比批次间隔时间大，或者批次调度延时持续上升，就意味着系统处理速度跟不上数据接收速度。这时候你就得考虑一下怎么把批次处理时间降下来（reducing）。

Spark Streaming 程序的处理进度可以用StreamingListener接口来监听，这个接口可以监听到接收器的状态和处理时间。不过需要注意的是，这是一个developer API接口，换句话说这个接口未来很可能会变动（可能会增加更多度量信息）。

1.5.4 性能调优

要获得Spark Streaming应用的最佳性能需要一点点调优工作。本节将深入解释一些能够改进Streaming应用性

能的配置和参数。总体上来说，你需要考虑这两方面的事情：

1. 提高集群资源利用率，减少单批次处理耗时。
2. 设置合适的批次大小，以便使数据处理速度能跟上数据接收速度。

减少批次处理时间

有不少优化手段都可以减少Spark对每个批次的处理时间。细节将在优化指南（Tuning Guide）中详谈。这里仅列举一些最重要的。

数据接收并发度

跨网络接收数据（如：从Kafka、Flume、socket等接收数据）需要在Spark中序列化并存储数据。

如果接收数据的过程是系统瓶颈，那么可以考虑增加数据接收的并行度。注意，每个输入DStream只包含一个单独的接收器（receiver，运行约worker节点），每个接收器单独接收一路数据流。所以，配置多个输入DStream就能从数据源的不同分区分别接收多个数据流。例如，可以将从Kafka拉取两个topic的数据流分成两个Kafka输入数据流，每个数据流拉取其中一个topic的数据，这样一来会同时有两个接收器并行地接收数据，因而增加了总体的吞吐量。同时，另一方面我们又可以把这些DStream数据流合并成一个，然后可以在合并后的DStream上使用任何可用的transformation算子。示例代码如下：

Scala

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

Java

```
int numStreams = 5;
List<JavaPairDStream<String, String>> kafkaStreams = new ArrayList<>(numStreams);
for (int i = 0; i < numStreams; i++) {
  kafkaStreams.add(KafkaUtils.createStream(...));
}
JavaPairDStream<String, String> unifiedStream = streamingContext.union(kafkaStreams.
↪get(0), kafkaStreams.subList(1, kafkaStreams.size()));
unifiedStream.print();
```

Python

```
numStreams = 5
kafkaStreams = [KafkaUtils.createStream(...) for _ in range (numStreams)]
unifiedStream = streamingContext.union(*kafkaStreams)
unifiedStream.pprint()
```

另一个可以考虑优化的参数就是接收器的阻塞间隔，该参数由配置参数（configuration parameter）spark.streaming.blockInterval决定。大多数接收器都会将数据合并成一个个数据块，然后再保存到spark内存中。对于map类算子来说，每个批次中数据块的个数将会决定处理这批数据并行任务的个数，每个接收器每批次数据处理任务数约等于（批次间隔 / 数据块间隔）。例如，对于2秒的批次间隔，如果数据块间隔为200ms，则创建的并发任务数为10。如果任务数太少（少于单机cpu core个数），则资源利用不够充分。如需增加这个任务数，对于给定的批次间隔来说，只需要减少数据块间隔即可。不过，我们还是建议数据块间隔至少要50ms，否则任务的启动开销占比就太高了。

另一个切分接收数据流的方法是，显示地将输入数据流划分为多个分区（使用 inputStream.repartition(<number of partitions>)）。该操作会在处理前，将数据散开重新分发到集群中多个节点上。

数据处理并发度

在计算各个阶段（stage）中，任何一个阶段的并发任务数不足都有可能造成集群资源利用率低。例如，对于reduce类的算子，如：reduceByKey 和 reduceByKeyAndWindow，其默认的并发任务数是由 spark.default.parallelism 决定的。你既可以修改这个默认值（spark.default.parallelism），也可以通过参数指定这个并发数量（见PairDStreamFunctions）。

数据序列化

调整数据的序列化格式可以大大减少数据序列化的开销。在 Spark Streaming 中主要有两种类型的数据需要序列化：

- 输入数据: 默认地，接收器收到的数据是以 StorageLevel.MEMORY_AND_DISK_SER_2 的存储级别存储到执行器（executor）内存中的。也就是说，收到的数据会被序列化以减少GC开销，同时保存两个副本以容错。同时，数据会优先保存在内存里，当内存不足时才吐出到磁盘上。很明显，这个过程中会有数据序列化的开销 – 接收器首先将收到的数据反序列化，然后再以spark所配置指定的格式来序列化数据。
- Streaming 算子所生产的持久化的RDDs：Streaming计算所生成的RDD可能会持久化到内存中。例如，基于窗口的算子会将数据持久化到内存，因为窗口数据可能会多次处理。所不同的是，spark core默认用 StorageLevel.MEMORY_ONLY 级别持久化RDD数据，而spark streaming默认使用StorageLevel.MEMORY_ONLY_SER 级别持久化接收到的数据，以便尽量减少GC开销。

不管是上面哪一种数据，都可以使用Kryo序列化来减少CPU和内存开销，详见Spark Tuning Guide。另，对于Kryo，你可以考虑这些优化：注册自定义类型，禁用对象引用跟踪（详见Configuration Guide）。

在一些特定的场景下，如果数据量不是很大，那么你可以考虑不用序列化格式，不过你需要注意的是取消序列化是否会导致大量的GC开销。例如，如果你的批次间隔比较短（几秒）并且没有使用基于窗口的算子，这种情况下你可以考虑禁用序列化格式。这样可以减少序列化的CPU开销以优化性能，同时GC的增长也不多。

任务启动开销

如果每秒启动的任务数过多（比如每秒50个以上），那么将任务发送给slave节点的开销会明显增加，那么你也就很难达到亚秒级（sub-second）的延迟。不过以下两个方法可以减少任务的启动开销：

- 任务序列化（Task Serialization）：使用Kryo来序列化任务，以减少任务本身的大小，从而提高发送任务的速度。任务的序列化格式是由 spark.closure.serializer 属性决定的。不过，目前还不支持闭包序列化，未来的版本可能会增加对此的支持。
- 执行模式（Execution mode）：Spark独立部署或者Mesos粗粒度模式下任务的启动时间比Mesos细粒度模式下的任务启动时间要短。详见Running on Mesos guide。

这些调整有可能能够减少100ms的批次处理时间，这也使得亚秒级的批次间隔成为可能。

设置合适的批次间隔

要想streaming应用在集群上稳定运行，那么系统处理数据的速度必须能跟上其接收数据的速度。换句话说，批次数据的处理速度应该和其生成速度一样快。对于特定的应用来说，可以从其对应的监控（monitoring）页面上观察验证，页面上显示的处理耗时应该要小于批次间隔时间。

根据 Spark Streaming 计算的性质，在一定的集群资源限制下，批次间隔的值会极大地影响系统的数据处理能力。例如，在 WordCountNetwork 示例中，对于特定的数据速率，一个系统可能能够在批次间隔为2秒时跟上数据接收速度，但如果把批次间隔改为500毫秒系统可能就处理不过来了。所以，批次间隔需要谨慎设置，以确保生产系统能够处理得过来。

要找出适合的批次间隔，你可以从一个比较保守的批次间隔值（如5~10秒）开始测试。要验证系统是否能跟上当前的数据接收速率，你可能需要检查一下端到端的批次处理延迟（可以看看Spark驱动器log4j日志中的Total delay，也可以用StreamingListener接口来检测）。如果这个延迟能保持和批次间隔差不多，那么系统基本就是稳定的。否则，如果这个延迟持久在增长，也就是说系统跟不上数据接收速度，那也就意味着系统不稳定。一旦系统文档下来后，你就可以尝试提高数据接收速度，或者减少批次间隔值。不过需要注意，瞬间的延迟增长可以只是暂时的，只要这个延迟后续会自动降下来就没有问题（如：降到小于批次间隔值）

内存调优

Spark应用内存占用和GC调优已经在调优指南（Tuning Guide）中有详细的讨论。墙裂建议你读一读那篇文档。本节中，我们只是讨论一下几个专门用于Spark Streaming的调优参数。

Spark Streaming 应用在集群中占用的内存量严重依赖于具体所使用的tranformation算子。例如，如果想要用一个窗口算子操纵最近10分钟的数据，那么你的集群至少需要在内存里保留10分钟的数据；另一个例子是updateStateByKey，如果key很多的话，相对应的保存的key的state也会很多，而这些都需要占用内存。而如果你的应用只是做一个简单的 “映射-过滤-存储”（map-filter-store）操作的话，那需要的内存就很少了。

一般情况下，streaming 接收器接收到的数据会以 StorageLevel.MEMORY_AND_DISK_SER_2 这个存储级别存到spark中，也就是说，如果内存装不下，数据将被吐到磁盘上。数据吐到磁盘上会大大降低streaming应用的性能，因此还是建议根据你的应用处理的数据量，提供充足的内存。最好就是，一边小规模地放大内存，再观察评估，然后再放大，再评估。

另一个内存调优的方向就是垃圾回收。因为streaming应用往往都需要低延迟，所以肯定不希望出现大量的或耗时较长的JVM垃圾回收暂停。

以下是一些能够帮助你减少内存占用和GC开销的参数或手段：

- DStream持久化级别（Persistence Level of DStreams）：前面数据序列化（Data Serialization）这小节已经提到过，默认streaming的输入RDD会被持久化成序列化的字节流。相对于非序列化数据，这样可以减少内存占用和GC开销。如果启用Kryo序列化，还能进一步减少序列化数据大小和内存占用量。如果你还需要进一步减少内存占用的话，可以开启数据压缩（通过spark.rdd.compress这个配置设定），只不过数据压缩会增加CPU消耗。
- 清除老数据（Clearing old data）：默认情况下，所有的输入数据以及DStream的transformation算子产生的持久化RDD都是自动清理的。Spark Streaming会根据所使用的transformation算子来清理老数据。例如，你用了一个窗口操作处理最近10分钟的数据，那么Spark Streaming会保留至少10分钟的数据，并且会主动把更早的数据都删掉。当然，你可以设置 streamingContext.remember 以保留更长时间段的数据（比如：你可能会需要交互式地查询更老的数据）。
- CMS垃圾回收器（CMS Garbage Collector）：为了尽量减少GC暂停的时间，我们强烈建议使用CMS垃圾回收器（concurrent mark-and-sweep GC）。虽然CMS GC会稍微降低系统的总体吞吐量，但我们仍建议使用它，因为CMS GC能使批次处理的时间保持在一个比较恒定的水平上。最后，你需要确保在驱动器（通过spark-submit中的–driver-java-options设置）和执行器（使用spark.executor.extraJavaOptions配置参数）上都设置了CMS GC。
- **其他提示: 如果还想进一步减少GC开销，以下是更进一步的可以尝试的手段：**
  - 配合Tachyon使用堆外内存来持久化RDD。详见Spark编程指南（Spark Programming Guide）
  - 使用更多但是更小的执行器进程。这样GC压力就会分散到更多的JVM堆中。

Important points to remember:

A DStream is associated with a single receiver. For attaining read parallelism multiple receivers i.e. multiple DStreams need to be created. A receiver is run within an executor. It occupies one core. Ensure that there are enough cores for processing after receiver slots are booked i.e. spark.cores.max should take the receiver slots into account. The receivers are allocated to executors in a round robin fashion.

When data is received from a stream source, receiver creates blocks of data. A new block of data is generated every blockInterval milliseconds. N blocks of data are created during the batchInterval where N = batchInterval/blockInterval. These blocks are distributed by the BlockManager of the current executor to the block managers of other executors. After that, the Network Input Tracker running on the driver is informed about the block locations for further processing.

An RDD is created on the driver for the blocks created during the batchInterval. The blocks generated during the batchInterval are partitions of the RDD. Each partition is a task in spark. blockInterval== batchinterval would mean that a single partition is created and probably it is processed locally.

The map tasks on the blocks are processed in the executors (one that received the block, and another where the block was replicated) that has the blocks irrespective of block interval, unless non-local scheduling kicks in. Having bigger blockinterval means bigger blocks. A high value of spark.locality.wait increases the chance of processing a block on the local node. A balance needs to be found out between these two parameters to ensure that the bigger blocks are processed locally.

Instead of relying on batchInterval and blockInterval, you can define the number of partitions by calling inputDstream.repartition(n). This reshuffles the data in RDD randomly to create n number of partitions. Yes, for greater parallelism. Though comes at the cost of a shuffle. An RDD's processing is scheduled by driver's jobscheduler as a job. At a given point of time only one job is active. So, if one job is executing the other jobs are queued.

If you have two dstreams there will be two RDDs formed and there will be two jobs created which will be scheduled one after the another. To avoid this, you can union two dstreams. This will ensure that a single unionRDD is formed for the two RDDs of the dstreams. This unionRDD is then considered as a single job. However the partitioning of the RDDs is not impacted.

If the batch processing time is more than batchinterval then obviously the receiver's memory will start filling up and will end up in throwing exceptions (most probably BlockNotFoundException). Currently there is no way to pause the receiver. Using SparkConf configuration spark.streaming.receiver.maxRate, rate of receiver can be limited.

1.5.5 容错语义

本节中，我们将讨论Spark Streaming应用在出现失败时的具体行为。

背景

要理解Spark Streaming所提供的容错语义，我们首先需要回忆一下Spark RDD所提供的基本容错语义。

1. RDD是不可变的，可重算的，分布式数据集。每个RDD都记录了其创建算子的血统信息，其中每个算子都以可容错的数据集作为输入数据。
2. 如果RDD的某个分区因为节点失效而丢失，则该分区可以根据RDD的血统信息以及相应的原始输入数据集重新计算出来。
3. 假定所有RDD transformation算子计算过程都是确定性的，那么通过这些算子得到的最终RDD总是包含相同的数据，而与Spark集群的是否故障无关。

Spark主要操作一些可容错文件系统的数据，如：HDFS或S3。因此，所有从这些可容错数据源产生的RDD也是可容错的。然而，对于Spark Streaming并非如此，因为多数情况下Streaming需要从网络远端接收数据，这回导致Streaming的数据源并不可靠（尤其是对于使用了fileStream的应用）。要实现RDD相同的容错属性，数据接收就必须用多个不同worker节点上的Spark执行器来实现（默认副本因子是2）。因此一旦出现故障，系统需要恢复两种数据：

1. 接收并保存了副本的数据 – 数据不会因为单个worker节点故障而丢失，因为有副本！
2. 接收但尚未保存副本数据 – 因为数据并没有副本，所以一旦故障，只能从数据源重新获取。

此外，还有两种可能的故障类型需要考虑：

1. Worker节点故障 – 任何运行执行器的worker节点一旦故障，节点上内存中的数据都会丢失。如果这些节点上有接收器在运行，那么其包含的缓存数据也会丢失。
2. Driver节点故障 – 如果Spark Streaming的驱动节点故障，那么很显然SparkContext对象就没了，所有执行器及其内存数据也会丢失。

有了以上这些基本知识，下面我们就进一步了解一下Spark Streaming的容错语义。

定义

流式系统的可靠度语义可以据此来分类：单条记录在系统中被处理的次数保证。一个流式系统可能提供保证必定是以下三种之一（不管系统是否出现故障）：

1. 至多一次（At most once）：每条记录要么被处理一次，要么就没有处理。
2. 至少一次（At least once）：每条记录至少被处理过一次（一次或多次）。这种保证能确保没有数据丢失，比“至多一次”要强。但有可能出现数据重复。
3. 精确一次（Exactly once）：每条记录都精确地只被处理一次 – 也就是说，既没有数据丢失，也不会出现数据重复。这是三种保证中最强的一种。

基础语义

任何流式处理系统一般都会包含以下三个数据处理步骤：

1. 数据接收（Receiving the data）：从数据源拉取数据。
2. 数据转换（Transforming the data）：将接收到的数据进行转换（使用DStream和RDD transformation算子）。
3. 数据推送（Pushing out the data）：将转换后最终数据推送到外部文件系统，数据库或其他展示系统。

如果Streaming应用需要做到端到端的“精确一次”的保证，那么就必须在以上三个步骤中各自都保证精确一次：即，每条记录必须，只接收一次、处理一次、推送一次。下面让我们在Spark Streaming的上下文环境中来理解一下这三个步骤的语义：

1. 数据接收: 不同数据源提供的保证不同，下一节再详细讨论。
2. 数据转换: 所有的数据都会被“精确一次”处理，这要归功于RDD提供的保障。即使出现故障，只要数据源还能访问，最终所转换得到的RDD总是包含相同的内容。
3. 数据推送: 输出操作默认保证“至少一次”的语义，是否能“精确一次”还要看所使用的输出算子（是否幂等）以及下游系统（是否支持事务）。不过用户也可以开发自己的事务机制来实现“精确一次”语义。这个后续会有详细讨论。

接收数据语义

不同的输入源提供不同的数据可靠性级别，从“至少一次”到“精确一次”。

从文件接收数据

如果所有的输入数据都来源于可容错的文件系统，如HDFS，那么Spark Streaming就能在任何故障中恢复并处理所有的数据。这种情况下就能保证精确一次语义，也就是说不管出现什么故障，所有的数据总是精确地只处理一次，不多也不少。

基于接收器接收数据

对于基于接收器的输入源，容错语义将同时依赖于故障场景和接收器类型。前面也已经提到过，spark Streaming主要有两种类型的接收器：

1. 可靠接收器 – 这类接收器会在数据接收并保存好副本后，向可靠数据源发送确认信息。这类接收器故障时，是不会给缓存的（已接收但尚未保存副本）数据发送确认信息。因此，一旦接收器重启，没有收到确认的数据，会重新从数据源再获取一遍，所以即使有故障也不会丢数据。
2. 不可靠接收器 – 这类接收器不会发送确认信息，因此一旦worker和driver出现故障，就有可能会丢失数据。

对于不同的接收器，我们可以获得如下不同的语义。如果一个worker节点故障了，对于可靠接收器来书，不会有数据丢失。而对于不可靠接收器，缓存的（接收但尚未保存副本）数据可能会丢失。如果driver节点故障了，除了接收到的数据之外，其他的已经接收且已经保存了内存副本的数据都会丢失，这将会影响有状态算子的计算结果。

为了避免丢失已经收到且保存副本的数，从 spark 1.2 开始引入了WAL（write ahead logs），以便将这些数据写入到可容错的存储中。只要你使用可靠接收器，同时启用WAL（write ahead logs enabled），那么久再也不用为数据丢失而担心了。并且这时候，还能提供“至少一次”的语义保证。

下表总结了故障情况下的各种语义： ========== ========== 部署场景 Worker故障 Driver故障 ========== ========== Spark 1.1及以前版本或者Spark 1.2及以后版本，且未开启WAL 若使用不可靠接收器，则可能丢失缓存（已接收但尚未保存副本）数据； 若使用不可靠接收器，则缓存数据和已保存数据都可能丢失；

> 若使用可靠接收器，则没有数据丢失，且提供至少一次处理语义 若使用可靠接收器，则没有缓存数据丢失，但已保存数据可能丢失，且不提供语义保证

Spark 1.2及以后版本，并启用WAL 若使用可靠接收器，则没有数据丢失，且提供至少一次语义保证 若使用可靠接收器和文件，则无数据丢失，且提供至少一次语义保证 ========== ========== ==========

从Kafka Direct API接收数据

从Spark 1.3开始，我们引入 Kafka Direct API，该API能为Kafka数据源提供“精确一次”语义保证。有了这个输入API，再加上输出算子的“精确一次”保证，你就能真正实现端到端的“精确一次”语义保证。（改功能截止Spark 1.6.1还是实验性的）更详细的说明见：Kafka Integration Guide。

输出算子的语义

输出算子（如 foreachRDD）提供“至少一次”语义保证，也就是说，如果worker故障，单条输出数据可能会被多次写入外部实体中。不过这对于文件系统来说是可以接受的（使用saveAs***Files 多次保存文件会覆盖之前的），所以我们需要一些额外的工作来实现“精确一次”语义。主要有两种实现方式： * 幂等更新（Idempotent updates）：就是说多次操作，产生的结果相同。例如，多次调用saveAs***Files保存的文件总是

包含相同的数据。 * 事务更新（Transactional updates）：所有的更新都是事务性的，这样一来就能保证更新的原子性。以下是一种实现方式：

- 用批次时间（在foreachRDD中可用）和分区索引创建一个唯一标识，该标识代表流式应用中唯一的一个数据块。
- 基于这个标识建立更新事务，并使用数据块数据更新外部系统。也就是说，如果该标识未被提交，则原子地将标识代表的数据更新到外部系统。否则，就认为该标识已经被提交，直接忽略之。

```
dstream.foreachRDD { (rdd, time) =>
  rdd.foreachPartition { partitionIterator =>
    val partitionId = TaskContext.get.partitionId()
    val uniqueId = generateUniqueId(time.milliseconds, partitionId)
    // 使用uniqueId作为事务的唯一标识，基于uniqueId实现partitionIterator所指向数据的原子事务提交
  }
}
```

1.5.6 下一步

- 其他相关参考文档
  - Kafka Integration Guide
  - Flume Integration Guide
  - Kinesis Integration Guide
  - Custom Receiver Guide
- Third-party DStream data sources can be found in Third Party Projects
- **API** 文档
  - **Scala** 文档
    - StreamingContext 和 DStream
    - KafkaUtils, FlumeUtils, KinesisUtils, TwitterUtils, ZeroMQUtils, 以及 MQTTUtils
  - **Java** 文档
    - JavaStreamingContext, JavaDStream 以及 JavaPairDStream
    - KafkaUtils, FlumeUtils, KinesisUtils TwitterUtils, ZeroMQUtils, 以及 MQTTUtils
  - **Python** 文档
    - StreamingContext 和 DStream
    - KafkaUtils
- 其他示例：Scala ， Java 以及 Python
- Spark Streaming 相关的 Paper 和 video。

1.6 机器学习库(MLib)编程指南

CHAPTER 2

部署

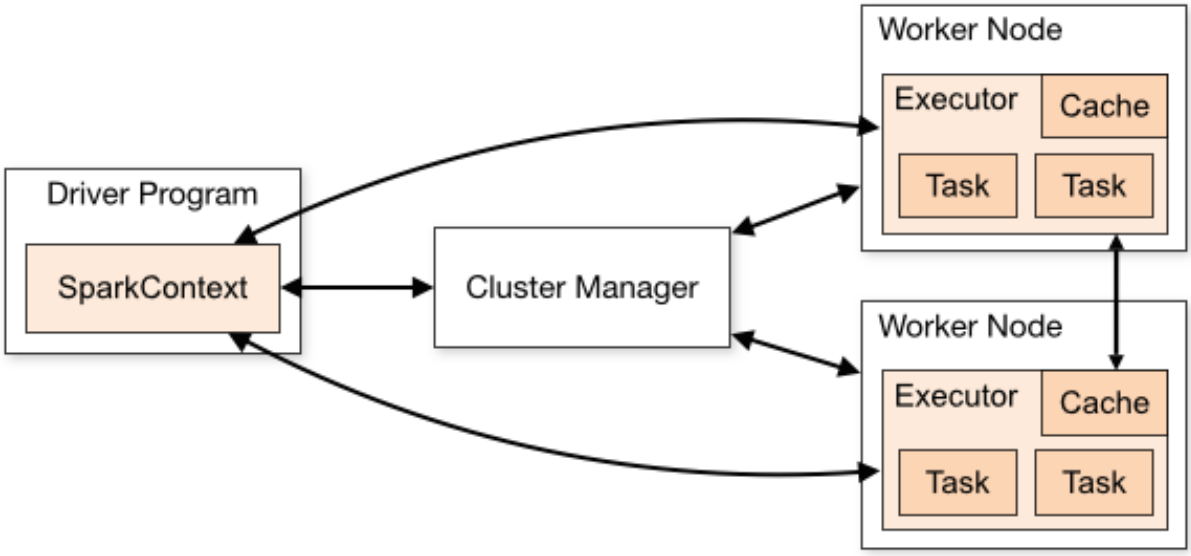
2.1 集群模式概述

本文简要描述了如何在集群环境中运行 Spark, 以便更容易理解其中涉及到的各个组件。如果想了解如何在集群中启动 Spark 应用程序，请参考 Spark应用程序提交指南 这篇文章。

2.1.1 组件

Spark 应用程序在集群上作为独立的进程集合运行, 这些进程之间通过主程序（也被称为 Driver 程序）中的 SparkContext 对象来进行协调。

具体来说，为了能够在集群上运行应用程序, SparkContext 要能连接到多种类型的集群管理器（Spark 自带的 Standalone 集群管理器, Mesos 或 YARN），这些集群管理器用于在应用程序之间分配资源。一旦连接上集群管理器，Spark 会在集群中的各个节点上为应用程序申请 Executor 用于执行计算任务和存储数据。接着，Spark 将应用程序代码（传递给 SparkContext 的 JAR 包或者 Python 文件）发送给 Executor。最后 SparkContext 将 Task 分发给各个 Executor 执行。



这个架构中有几个值得注意的地方：

1. 每个 Spark 应用程序都有自己的 Executor 进程，整个应用程序生命周期内 Executor 进程都保持着运行状态，并且以多线程方式运行所接收到的任务。这样的好处是，可以隔离各个 Spark 应用程序，从调度角度来看，每个 Driver 可以独立调度本应用程序内部的任务，从 Executor 角度来看，来自不同 Spark 应用程序的任务将会在不同的 JVM 中运行。然而这也意味着无法在多个 Spark 应用程序（SparkContext 实例）之间共享数据，除非把数据写到外部存储系统中。
2. Spark 对底层的集群管理器一无所知。只要 Spark 能获取到执行器进程，并且能与之通信即可。这样即使在一个支持多种应用程序的集群管理器（如：Mesos 或 YARN）上运行 Spark 程序也相对比较容易。
3. Driver 程序在整个生命周期内必须监听并接受其对应的各个 Executor 的连接请求（参考：spark.driver.port and spark.fileserver.port in the network config section）。因此，Driver 程序必须能够被所有 Worker 节点访问到。
4. 因为集群上的任务是由 Driver 来调度的，所以 Driver 应该在 Worker 节点附近运行，最好在同一个本地局域网内。如果你需要远程对集群发起请求，最好是开启到 Driver 的 RPC 服务并且让其就近提交操作，而不是在离集群 Worker 节点很远的地方运行 Driver。

2.1.2 集群管理器类型

Spark 目前支持以下3种集群管理器：

- Standalone – Spark 自带的一个简单的集群管理器，它使得启动一个 Spark 集群变得非常简单。
- Apache Mesos – 一个可以运行 Hadoop MapReduce 和服务型应用程序的通用集群管理器。
- Hadoop YARN – Hadoop 2 的资源管理器。
- Kubernetes (experimental) – 除上述之外，还有Kubernetes的实验支持。 Kubernetes是一个提供以容器为中心的基础设施的开源平台。 Kubernetes的支持正在积极开发在apache-spark-on-k8s Github组织中。 有关文档，请参阅该项目的自述文件.

2.1.3 提交 Spark 应用

利用 spark-submit 脚本，可以向 Spark 所支持的任意一种集群管理器提交应用程序。更多详细信息请参见

Spark应用程序提交指南 这篇文章。

2.1.4 监控

每个 Driver 程序都有一个 web UI，通常会绑定到 4040 端口，它会展示集群上正在执行的 Tasks、Executors 以及存储空间使用等信息。你只需要在浏览器中输入 http://<driver-node>:4040 这个地址即可访问这个 web UI。监控指南 这篇文章中描述了其它的一些监控选项。

2.1.5 作业调度

Spark 可以在应用程序之间（集群管理器这一层面）和之内（如：同一个SparkContext对象运行了多个计算作业）控制资源分配。更多详细信息参见 作业调度概览 这篇文章。

2.1.6 术语表

下表总结了一些之前看到的集群概念相关的术语:

术语	含义
Application(应用程序)	构建于 Spark 上的用户程序。它由集群上的一个驱动器（driver）和多个执行器（executor）组成。
Application jar(应用程序jar)	包含 Spark 应用程序的 jar 包。有时候，用户会想要把应用程序代码及其依赖打到一起，形成一个 “uber jar”（包含自身以及所有依赖库的jar包），注意这时候不要把 Spark 或 Hadoop 的库打进来，这些库会在运行时加载。
Driver Program(驱动器程序)	运行 main 函数并创建 SparkContext 的进程。
Cluster Manager(集群管理器)	用于在集群上分配资源的外部服务（如：Standalone 集群管理器、Mesos或者YARN。
Deploy Mode(部署模式)	用于区分 Driver 进程在哪儿运行。在 “cluster” 模式下，框架在集群内部启动 Driver 程序；在 “client” 模式下，提交者在集群外部启动 Driver 程序。
Worker Node(工作节点)	集群中可以运行应用程序代码的任意一个节点。
Executor(执行器)	在集群 Worker 节点上为某个应用程序启动的工作进程, 它专门用于运行计算任务，并在内存或磁盘上保存数据。每个应用程序都有自己的 Executor。
Task(任务)	下发给 Executor 的工作单元。
Job(作业)	一个并行计算 Job 由一组 Task 组成，并由 Spark action（如：save、collect）触发启动；你将会在 Driver 的日志中看到这个术语。
Stage(步骤)	每个 Job 可以划分为多个更小的 Task 集合，这些 Task 集合称为 Stage，这些 Stage 彼此依赖形成一个有向无环图（类似于 MapReduce 中的 map 和 reduce）；你将会在 Driver 的日志中看到这个术语。

2.2 提交 Spark 应用程序

Spark bin 目录下的 spark-submit 脚本用于在集群中启动 Spark 应用程序。通过一个统一的接口它可以使用 Spark 支持的所有类型的集群管理器, 因此不需要为每个集群管理器专门配置你的应用程序。

2.2.1 应用程序依赖打包

如果你的代码依赖于其它工程，为了将代码发布到 Spark 集群, 你需要将应用程序的依赖项一起打包进来。这需要创建一个包含你自己代码和其依赖程序集 jar 包（或者 “uber” jar）。sbt 和 Maven 都有 assembly 插件。创建程序集 jar 包时，需要把 Spark 和 Hadoop 的 jar 包的依赖范围声明为 provided；因为这些 jar 包会由集群管理器在运行时提供，所以不需要再打包进来。一旦打完 jar 包之后，你就可以调用 bin/spark-submit 脚本来提交你的 jar 包了。

对于 Python，你可以使用 spark-submit 的 –py-files 参数，将你的程序以 .py、.zip 或 .egg 文件格式提交给集群。如果你需要依赖很多 Python 文件，我们推荐你将它们打成一个 .zip 或者 .egg 包。

2.2.2 使用 spark-submit 启动应用程序

打包好一个应用程序之后，就可以使用 bin/spark-submit 脚本来提交它。这个脚本会负责设置 Spark 及其依赖的 classpath，同时它可以支持 Spark 所支持的所有不同类型的集群管理器和部署模式:

```
./bin/spark-submit \
  --class <main-class>
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ... # 其他选项
  <application-jar> \
  [application-arguments]
```

一些常用选项如下：

- –class: 应用程序的入口（例如：org.apache.spark.examples.SparkPi）
- –master: 集群的 master URL（如：spark://23.195.26.187:7077）
- –deploy-mode: Driver 进程是在集群的 Worker 节点上运行（cluster模式），还是在本地作为一个外部客户端运行（client模式）（默认值是：client）
- –conf: 可以设置任意的Spark配置属性，键值对（key=value）格式。如果值中包含空白字符，可以用双引号括起来（”key=value”）。
- application-jar: 应用程序jar包路径，该jar包必须包括你自己的代码及其所有的依赖项。如果是URL，那么该路径URL必须是对整个集群可见且一致的，如：hdfs://path 或者 file://path （要求对所有节点都一致）
- application-arguments: 传给应用程序入口类 main 函数的启动参数，可选。

一种常见的部署策略是，从一台距离 Worker 节点的物理距离比较近的网关机器上提交你的应用程序(例如 Standalone EC2 集群中的 Master 节点)。这种情况下比较适合使用 client 模式。client 模式下，Driver 直接运行在 spark-submit 的进程中，对于集群来说它就像是一个客户端。应用程序的输入输出也被绑定到控制台上。因此，这种模式尤其适合于交互式执行（REPL）的应用程序，(例如 spark-shell)。

当然, 如果从距离 Worker 节点很远的机器(例如你的笔记本)上提交应用程序，通常使用 cluster 模式来尽量减少 Driver 和 Executor 之间的网络延迟。目前，只有 YARN 支持 Python 应用程序的 cluster 模式部署。

对于 Python 应用，只要把 <application-jar> 替换成一个 .py 文件，再把 Python 的 .zip、.egg 或者 .py 文件传给 –py-files 参数即可。

只有很少几个参数是专门用于所使用的集群管理器。例如，对于一个使用 cluster 模式部署的 Spark Standalone集群，你可以指定 –supervise 参数来确保 Driver 在异常退出码非0的情况下能够自动重启。运行 spark-submit –help 命令可查看所有这样的选项列表。下面是常用选项的几个示例：

```
# 本地运行应用程序，使用8个core
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  /path/to/examples.jar \
  100

# 在 client 部署模式中的一个 Spark 独立集群上运行应用程序，
./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

# 在 cluster 部署模式中的一个 Spark Standalone集群上运行应用程序，异常退出时自动重启
./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

# 在 YARN 集群上运行应用程序
export HADOOP_CONF_DIR=XXX
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \  # 对于 client 模式其值为 client
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000

# 在一个 Spark Standalone集群上运行 python 应用程序
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000

# 在 cluster 部署模式中的一个 Mesos 集群上运行应用程序，异常时自动重启
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master mesos://207.184.161.138:7077 \
  --deploy-mode cluster
  --supervise
  --executor-memory 20G \
  --total-executor-cores 100 \
```

```
  http://path/to/examples.jar \
  1000
```

2.2.3 Master URLs

传给 Spark 的 master URL 可以是以下几种格式:

Master URL	含义
local	本地运行 Spark，只使用 1 个 Worker 线程（即没有并行计算）
local[K]	本地运行 Spark，使用 K 个 Worker 线程（理想情况是将这个值设为你机器上 CPU core的个数）
local[K,F]	本地运行 Spark，使用 K 个 Worker 线程并且允许 Task 最大失败次数为 F (see spark.task.maxFailures for an explanation of this variable)
local[*]	本地运行 Spark，使用 Worker 线程数和机器上逻辑 CPU core个数一样。
local[*,F]	本地运行 Spark，使用 Worker 线程数和机器上逻辑 CPU core个数一样并且允许 Task 最大失败次数为 F。
spark://HOST:PORT	连接到指定的 Spark Standalone 集群的 master。端口是可以配置的，默认是7077。
mesos://HOST:PORT	连接到指定的 Mesos 集群。端口号是可以配置的，默认是 5050。如果 Mesos 集群依赖于 ZooKeeper，可以使用 mesos://zk://... 来提交，注意 –deploy-mode需要设置为 cluster，同时，HOST:PORT 应指向 MesosClusterDispatcher.
yarn	连接到指定的 YARN 集群，使用–deploy-mode 来指定 client 模式或是 cluster 模式。YARN 集群位置需要通过 $HADOOP_CONF_DIR 或者 $YARN_CONF_DIR 变量来查找。

2.2.4 从文件中加载配置

spark-submit 脚本可以从一个属性文件加载默认的 Spark配置值，并将这些属性值传给你的应用程序。Spark 默认会从 Spark 安装目录中的 conf/spark-defaults.conf 文件读取这些属性配置。更详细信息，请参考 加载默认配置 这篇文章。

用这种方式加载默认Spark属性配置，可以在调用 spark-submit 脚本时省略一些参数标志。例如：如果属性文件中设置了 spark.master 属性，那么你就可以忽略 spark-submit 的 –master参数。通常，在代码里显示地在 SparkConf 对象上设置的参数具有最高的优先级，其次是 spark-submit 中传的参数，再次才是spark-defaults.conf文件中的配置值。

如果你总是搞不清楚最终生效的配置值是从哪里来的，你可以通过 spark-submit 的 —verbose 选项来打印细粒度的调试信息。

2.2.5 高级依赖管理

使用 spark-submit 时，application jar 和 –jars 选项指定的 jar 包都会自动传到集群上。—jars后面的 URL 必须以逗号分隔。这个列表已经包含在驱动器和执行器的类路径上扩展目录不支持 —jars。

Spark 使用下面的URL协议以允许不同的 jar 包分发策略:

- file: – 文件绝对路径，并且file:/URI是通过驱动器的HTTP文件服务器来下载的，每个执行器都从驱动器的HTTP server拉取这些文件。
- hdfs:, http:, https:, ftp: – 设置这些参数后，Spark将会从指定的URI位置下载所需的文件和jar包。

- local: – local:/ 打头的URI用于指定在每个工作节点上都能访问到的本地或共享文件。这意味着，不会占用网络IO，特别是对一些大文件或jar包，最好使用这种方式，当然，你需要把文件推送到每个工作节点上，或者通过NFS和GlusterFS共享文件。

注意，每个 SparkContext 对应的 jar 包和文件都需要拷贝到所对应 Executor 的工作目录下。一段时间之后，这些文件可能会占用相当多的磁盘。在 YARN 上，这些清理工作是自动完成的；而在Spark独立部署时，这种自动清理需要配置 spark.worker.cleanup.appDataTtl 属性。

用户还可以用 –packages 参数，通过给定一个逗号分隔的 maven 坐标，来指定其它依赖项。这个命令会自动处理依赖树。额外的 maven库（或者SBT resolver）可以通过 –repositories 参数来指定。Spark 命令（pyspark，spark-shell，spark-submit）都支持这些参数。

对于 Python，也可以使用等价的 –py-files 选项来分发 .egg、.zip 以及 .py 库到执行器上。

2.2.6 更多信息

部署完了你的应用程序后，集群模式概览 一文中描述了分布式执行中所涉及到的各个组件，以及如何监控和调试应用程序。

2.3 Spark 独立模式

除了可以在 Mesos 和 YARN 集群管理器上运行之外，Spark 还提供一种简单的独立部署模式。你既可以通过手工启动一个master和多个worker来手动地启动一个独立集群，也可以使用我们提供的启动脚本来启动一个独立集群。为了方便测试，你也可以在单机上运行这些后台程序。

2.3.1 Spark集群独立安装

要独立安装Spark，你只需要将编译好的 Spark 版本复制到集群中每一个节点上即可。你可以下载Spark 每个 release 的预编译版本，也可以自己构建Spark。

2.3.2 手动启动集群

执行以下命令，就可以启动一个独立的 master 服务器：

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

启动完成之后，master 自己会打印出一个 spark://HOST:PORT URL，你可以使用这个URL将worker 连接到 master，或者作为master参数传递给。你还可以在 master 的 web UI（默认地址是：http://localhost:8080）上查看 master URL。

类似地，你可以通过以下命令，启动一个或多个worker节点，并将其连接到 master：

```
./sbin/start-slave.sh <master-spark-URL>
```

启动一个 worker 以后，查看 master 的 web UI（默认地址是：http://localhost:8080），你应该可以看到一个新的节点，以及节点的CPU个数和内存(为操作系统预留1GB)。

最后，下面的配置选项将可以传给 master 和 worker：

参数	含义
-h HOST, –host HOST	监听的主机名
-i HOST, –ip HOST	监听的主机名（已经废弃，请使用-h 或者–host）
-p PORT, –port PORT	服务监听的端口（master节点默认7077，worker节点随机）
–webui-port PORT	web UI端口（master节点默认8080，worker节点默认8081）
-c CORES, –cores CORES	Spark应用程序能够使用的CPU core数上限（默认值是：所有可用的CPU core个数）；仅worker节点有效
-m MEM, –memory MEM	Spark应用程序能够使用的内存上限，格式为 1000M 或者 2G（默认值是：机器总内存减去1G）；仅worker节点有效
-d DIR, –work-dir DIR	工作目录，同时 job 的日志也输出到该目录（默认值是：${SPAKR_HOME}/work）；仅worker节点有效
–properties-file FILE	自定义 Spark 属性文件的加载路径（默认值是：conf/spark-defaults.conf）

2.3.3 集群启动脚本

要使用启动脚本来启动一个Spark独立集群，你应该在 Spark 安装目录下创建一个文件conf/slaves，它包括你打算作为worker节点启动的每一台机器的主机名（或IP），每行一台机器。如果 conf/slaves 文件不存在，启动脚本会默认会用单机方式启动，这种方式非常方便测试。注意，master 节点访问各个 worker 时使用 ssh。默认情况下，你需要配置 ssh免密码登陆（使用秘钥文件）。如果你没有设置免密码登陆，那么你也可以通过环境变量SPARK_SSH_FOREGROUND来一个一个地设置每个 worker 的密码。

设置好 conf/slaves 文件以后，你就可以使用以下基于 Hadoop 部署脚本的 shell 脚本来启动或停止集群，这些脚本都在 ${SPARK_HOME}/sbin 目录下：

- sbin/start-master.sh – 在执行该脚本的机器上启动一个 master 实例
- sbin/start-slaves.sh – conf/slaves 文件中指定的每一台机器上都启动一个 slave 实例
- sbin/start-slave.sh – 在执行该脚本的机器上启动一个 slave 实例
- sbin/start-all.sh – 启动一个 master 和多个 slave 实例，详细见上面的描述。
- sbin/stop-master.sh – 停止 start-master.sh 所启动的 master 实例
- sbin/stop-slaves.sh – 停止所有在 conf/slaves 中指定的 slave 实例
- sbin/stop-all.sh – 停止 master 节点和所有的 slave 节点，详细见上面的描述

注意，这些脚本都需要在你启动Spark master的机器上运行，而不是你的本地机器。

通过在 conf/spark-env.sh 文件中设置环境变量, 你可以进一步的配置集群。可以通过复制 conf/spark-env.sh.template 来创建这个文件，并且为了使这些环境变量设置生效你还需要将其拷贝到所有worker节点上。以下是可用的设置：

环境变量	含义
SPARK_MASTER_IP	master实例绑定的IP地址，例如，绑定到一个公网IP
SPARK_MASTER_PORT	master实例绑定的端口（默认7077）
SPARK_MASTER_WEBUI_PORT	master的web UI的端口（默认8080）
SPARK_MASTER_OPTS	master的专用配置属性，格式如”-Dx=y”（默认空），可能的选项请参考下面的列表。
SPARK_LOCAL_DIRS	Spark的本地工作目录，包括：映射输出的临时文件和RDD保存到磁盘上的临时数据。这个目录需要快速访问，最好设成本地磁盘上的目录。也可以通过使用逗号分隔列表，将其设成多个磁盘上的不同路径。
SPARK_WORKER_CORES	本机上Spark应用程序可以使用的CPU core上限（默认所有CPU core）
SPARK_WORKER_MEMORY	本机上Spark应用程序可以使用的内存上限，如：1000m，2g（默认为本机总内存减去1GB）；注意每个应用单独使用的内存大小要用 spark.executor.memory 属性配置的。
SPARK_WORKER_PORT	Spark worker绑定的端口（默认随机）
SPARK_WORKER_WEBUI_PORT	worker web UI端口（默认8081）
SPARK_WORKER_INSTANCES	每个slave机器上启动的worker实例个数（默认：1）。如果你的slave机器非常强劲，可以把这个值设为大于1；相应的，你需要设置SPARK_WORKER_CORES参数来显式地限制每个worker实例使用的CPU个数，否则每个worker实例都会使用所有的CPU。
SPARK_WORKER_DIR	Spark worker的工作目录，包括worker的日志以及临时存储空间（默认：${SPARK_HOME}/work）
SPARK_WORKER_OPTS	worker的专用配置属性，格式为：”-Dx=y”，可能的选项请参考下面的列表。
SPARK_DAEMON_MEMORY	Spark master和worker后台进程所使用的内存（默认：1g）
SPARK_DAEMON_JAVA_OPTS	Spark master和workers后台进程所使用的JVM选项，格式为：”-Dx=y”（默认空）
SPARK_PUBLIC_DNS	Spark master和workers使用的公共DNS（默认空）

注意: 启动脚本目前不支持Windows。如需在Windows上运行，请手工启动 master 和 workers。

SPARK_MASTER_OPTS支持以下系统属性:

属性名	默认值	含义
spark.deploy.retainedApplications	200	web UI上最多展示几个已结束应用。更早的应用的数将被删除。
spark.deploy.retainedDrivers	200	web UI上最多展示几个已结束的驱动器。更早的驱动器进程数据将被删除。
spark.deploy.spreadOut	true	独立部署集群的master是否应该尽可能将应用分布到更多的节点上；设为true，对数据本地性支持较好；设为false，计算会收缩到少数几台机器上，这对计算密集型任务比较有利。
spark.deploy.defaultCores	无限制)	Spark独立模式下应用程序默认使用的CPU个数（没有设置spark.cores.max的情况下）。如果不设置，则为所有可用CPU个数（除非设置了spark.cores.max）。如果集群是共享的，最好将此值设小一些，以避免用户占满整个集群。
spark.worker.timeout	60	如果master没有收到worker的心跳，那么将在这么多秒之后，master将丢弃该worker。

SPARK_WORKER_OPTS支持以下系统属性:

属性名	默认值	含义
spark.worker.cleanup.enabled	false	是否定期清理 worker 和应用的工作目录。注意，该设置仅在独立模式下有效，YARN有自己的清理方式；同时，只会清理已经结束的应用对应的目录。
spark.worker.cleanup.interval	1800 (30 minutes)	worker清理本地应用工作目录的时间间隔（秒）
spark.worker.cleanup.appDataTtl	7 * 24 * 3600 (7 days)	清理多久以前的应用的工作目录。这个选项值将取决于你的磁盘总量。spark应用会将日志和jar包都放在其对应的工作目录下。随着时间流逝，应用的工作目录很快会占满磁盘，尤其是在你的应用提交比较频繁的情况下。

2.3.4 连接应用程序到集群

要在 Spark 集群上运行一个应用程序，只需把 spark://IP:PORT 这个master URL 传给SparkContext构造器。

如需要运行交互式的spark shell，运行如下命令：

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

你也可以通过设置选项 –total-executor-cores <numCores> 来控制 spark-shell 在集群上使用的CPU 核心总数。

2.3.5 启动Spark应用

spark-submit脚本提供了最直接的方式来提交一个编译好的Spark应用程序到集群上。对于独立集群来说，Spark目前支持两种部署模式。在客户端（client）模式下，驱动器进程（driver）将在提交应用程序的机器上启动。而在集群（cluster）模式下，驱动器（driver）将会在集群中的某一台worker上启动，同时提交应用程序的客户端进程在完成提交应用程序之后立即退出，而不会等到Spark应用运行结束。

如果你的应用程序是通过Spark submit 来启动的，那么应用程序 jar 包会自动分发到所有的 Worker节点上。应用程序所依赖的任何额外的jar包，都必须在 –jars 参数中指明，并以逗号分隔（如：–jars jar1,jar2）。

另外，独立集群模式还支持异常退出（返回值非0）时自动重启。想要使用这个特性，你需要在启动应用程序时将 –supervise 标识传递给 spark-submit。随后如果你需要杀掉一个不断失败的应用程序，你可能需要运行如下指令：

```
./bin/spark-class org.apache.spark.deploy.Client kill <master url> <driver ID>
```

你可以在 master web UI（http://<master url>:8080）上查看驱动器ID。

2.3.6 资源调度

独立集群模式目前只支持简单的先进先出（FIFO）调度器。这个调度器可以支持多用户，你可以控制每个应用程序所使用的最大资源。默认情况下，Spark应用会申请集群中所有的CPU，这不太合理，除非你的集群同一时刻只运行一个应用程序。你可以通过在 SparkConf 中设置 spark.cores.max 来限制其使用的CPU核心总数。例如：

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

另外，你也可以通过 conf/spark-env.sh 中的 spark.deploy.defaultCores 设置应用默认使用的CPU个数（特别针对没有设置 spark.cores.max 的应用）。

```
export SPARK_MASTER_OPTS="-Dspark.deploy.defaultCores=<value>"
```

在一些共享的集群上，用户很可能忘记单独设置一个最大CPU限制，那么这个参数将很有用。

2.3.7 监控和日志

Spark 独立安装模式提供了一个基于web的集群监控用户界面。master 和每个 worker 都有其对应的web UI，展示集群和 Spark 作业的统计数据。默认情况下，你可以在 master 机器的8080端口上访问到这个 web UI。这个端口可以通过配置文件或者命令行来设置。

另外，每个作业的详细日志，将被输出到每个 slave 节点上的工作目录下（默认为：${SPARK_HOME}/work）。每个 Spark 作业下都至少有两个日志文件，stdout 和 stderr，这里将包含所有的输出到控制台的信息。

2.3.8 和Hadoop同时运行

你可以让 Spark 和现有的Hadoop集群同时运行，只需要将Spark作为独立的服务在同样的机器上启动即可。这样Spark可以通过hdfs:// URL来访问Hadoop上的数据（通常情况下是，hdfs://<namenode>:9000/path，你可以在Hadoop Namenode的web UI上找到正确的链接）。当然，你也可以为 Spark 部署一个独立的集群，这时候 Spark 仍然可以通过网络访问 HDFS 上的数据；这会比访问本地磁盘慢一些，但如果Spark和Hadoop集群都在同一个本地局域网内的话，问题不大（例如，你可以在Hadoop集群的每个机架上新增一些部署Spark的机器）。

2.3.9 网络安全端口配置

Spark会大量使用网络资源，而有些环境会设置严密的防火墙设置，以严格限制网络访问。完整的端口列表，请参考这里：security page.

2.3.10 高可用性

默认情况下，独立调度的集群能够容忍worker节点的失败（在Spark本身来说，它能够将失败的工作移到其他worker节点上）。然而，调度器需要master做出调度决策，而这（默认行为）会造成单点失败：如果master挂了，任何应用都不能提交和调度。为了绕过这个单点问题，我们有两种高可用方案，具体如下：

基于Zookeeper的热备master

概要

利用Zookeeper来提供领导节点选举以及一些状态数据的存储，你可以在集群中启动多个master并连接到同一个Zookeeper。其中一个将被选举为“领导”，而另一个将处于备用（standby）状态。如果“领导”挂了，则另一个master会立即被选举，并从Zookeeper恢复已挂“领导”的状态，并继续调度。整个恢复流程（从“领导”挂开始计时）可能需要1到2分钟的时间。注意，整个延时只会影响新增应用 – 已经运行的应用不会受到影响。

更多关于Zookeeper信息请参考这里：here

配置

要启用这种恢复模式，你可以在spark-env中设置 SPARK_DAEMON_JAVA_OPTS，有关这些配置的更多信息，请参阅配置文档

可能的问题：如果你有多个master，但没有正确设置好master使用Zookeeper的配置，那么这些master彼此都不可见，并且每个master都认为自己是“领导”。这件会导致整个集群处于不稳定状态（多个master都会独立地进行调度）

详细

如果你已经有一个Zookeeper集群，那么启动高可用特性是很简单的。只需要在不同节点上启动多个master，并且配置相同的Zookeeper（包括Zookeeper URL和目录）即可。masters可以随时添加和删除。

在调度新提交的Spark应用或者新增worker节点时，需要知道当前”领导“的IP地址。你只需要将以前单个的master地址替换成多个master地址即可。例如，你可以在SparkContext中设置master URL为spark://host1:port1.host2:port2。这会导致SparkContext在两个master中都进行登记 – 那么这时候，如果host1挂了，这个应用的配置同样可以在新”领导“（host2）中找到。

”在master注册“和普通操作有一个显著的区别。在Spark应用或worker启动时，它们需要找当前的”领导“master，并在该master上注册。一旦注册成功，它们的状态将被存储到Zookeeper上。如果”老领导“挂了，”新领导“将会联系所有之前注册过的Spark应用和worker并通知它们领导权的变更，所以Spark应用和worker在启动时甚至没有必要知道”新领导“的存在。

由于这一特性，新的master可以在任何时间添加进来，你唯一需要关注的就是，新的应用或worker能够访问到这个master。总之，只要应用和worker注册成功，其他的你都不用管了。

基于本地文件系统的单点恢复

概要

利用Zookeeper当然是生成环境下高可用的最佳选择，但有时候你仍然希望在master挂了的时候能够重启之，FILESYSTEM模式能帮你实现这一需求。当应用和worker注册到master的时候，他们的状态都将被写入文件系统目录中，一旦master挂了，你只需要重启master，这些状态都能够恢复。

配置

要使用这种恢复模式，你需要在spark-env中设置SPARK_DAEMON_JAVA_OPTS，可用的属性如下：

系统属性	含义
spark.deploy.recoveryMode	设为FILESYSTEM以启用单点恢复模式（默认空）
spark.deploy.recoveryDirectory	用于存储可恢复状态数据的目录，master进程必须有访问权限

Details（详细）

- 这个解决方案可以用于和一些监控、管理系统进行串联（如：monit），或者与手动重启相结合。
- 至少，基于文件系统工单恢复总比不能恢复强；同时，这种恢复模式也会是开发或者实验场景下的不错的选择。在某些情况下，通过stop-master.sh杀死master可能不会清理其状态恢复目录下的数据，那么这时候你启动或重启master，将会进入恢复模式。这可能导致master的启动时间长达一分钟（master可能要等待之前注册的worker和客户端超时）。
- 你也可以使用NFS目录来作为数据恢复目录（虽然这不是官方声明支持的）。如果老的master挂了，你可以在另一个节点上启动master，这个master只要能访问同一个NFS目录，它就能够正确地恢复状态数据，包括之前注册的worker和应用（等价于Zookeeper模式）。后续的应用必须使用新的master来进行注册。

2.4 在Mesos上运行Spark

2.5 在 YARN 上运行 Spark

Spark 对 YARN (Hadoop NextGen) 的支持是从 0.6.0 版本开始的，后续的版本也在持续的改进。

2.5.1 在YARN上启动

首先要确保 HADOOP_CONF_DIR 或者 YARN_CONF_DIR 指向一个包含 Hadoop 集群客户端配置文件的目录。这些配置用于向 HDFS 写数据以及连接到 YARN 的 ResourceManager（资源管理器）。这个目录下的配置将会分发到 YARN 集群上的各个节点，这样应用程序使用的所有YARN 容器都将使用同样的配置。如果这些配置引用了 Java 系统属性或其它不属于 YARN 管理的环境变量，那么这些系统属性和环境变量也应该在 Spark 应用程序的配置中设置（包括 Driver、Executors，以及运行于 client 模式时的 YARN Application Master, 简称 AM）。

有两种部署模式可用于在 YARN 上启动 Spark 应用程序。在 cluster 模式下，Spark driver 在YARN Application Master中运行（运行于集群中），因此客户端可以在Spark应用启动之后关闭退出。而client模式下，Spark驱动器在客户端进程中，这时的YARN Application Master只用于向YARN申请资源。

不同于 Spark standalone 和 Mesos 模式，YARN 的master地址不是在–master参数中指定的，而是在Hadoop配置文件中设置。因此，这种情况下，–master只需设置为yarn。

以下用cluster模式启动一个Spark应用：

```
$ ./bin/spark-submit --class path.to.your.Class --master yarn --deploy-mode cluster␣
↪[options] <app jar> [app options]
```

例如：

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

以上例子中，启动了一个YARN客户端程序，使用默认的Application Master。而后SparkPi在Application Master中的子线程中运行。客户端会周期性的把Application Master的状态信息拉取下来，并更新到控制台。客户端会在你的应用程序结束后退出。参考“调试你的应用”，这一节说明了如何查看驱动器和执行器的日志。

要以client模式启动一个spark应用，只需在上面的例子中把cluster换成client。下面这个例子就是以client模式启动spark-shell：

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

2.5.2 增加其他JAR包

在cluster模式下，驱动器不在客户端机器上运行，所以SparkContext.addJar添加客户端本地文件就不好使了。要使客户端上本地文件能够用SparkContext.addJar来添加，可以用–jars选项：

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

2.5.3 准备

在YARN上运行Spark需要其二进制发布包构建的时候增加YARN支持。二进制发布包可以在这里下载：downloads page 。

想要自己编译，参考这里： Building Spark

2.5.4 配置

大多数配置，对于YARN或其他集群模式下，都是一样的。详细请参考这里： configuration page。

以下是YARN上专有的配置项。

2.5.5 调试应用程序

在YARN术语集中，执行器和Application Master在容器（container）中运行。YARN在一个应用程序结束后，有两种处理容器日志的模式。如果开启了日志聚合（yarn.log-aggregation-enable），那么容器日志将被复制到HDFS，并删除本地日志。而后这些日志可以在集群任何节点上用yarn logs命令查看：

```
yarn logs -applicationId <app ID>
```

以上命令，将会打印出指定应用的所有日志文件的内容。你也可以直接在HDFS上查看这些日志（HDFS shell或者HDFS API）。这些目录可以在你的YARN配置中指定（yarn.nodemanager.remote-app-log-dir和yarn.nodemanager-remote-app-log-dir-suffix）。这些日志同样还可以在Spark Web UI上Executors tab页查看。当然，你需要启动Spark history server和 MapReduce history server，再在 yarn-site.xml 中配置好yarn.log.server.url。Spark history server UI 将把你重定向到MapReduce history server 以查看这些聚合日志。

如果日志聚合没有开启，那么日志文件将在每台机器上的 YARN_APP_LOGS_DIR 目录保留，通常这个目录指向 /tmp/logs 或者 $HADOOP_HOME/log/userlogs（这取决于Hadoop版本和安全方式）。查看日志的话，需要到每台机器上查看这些目录。子目录是按 application ID 和 container ID来组织的。这些日志同样可以在 Spark Web UI 上 Executors tab 页查看，而且这时你不需要运行MapReduce history server。

如果需要检查各个容器的启动环境，可以先把 yarn.nodemanager.delete.debug-delay-sec 增大（如：36000），然后访问应用缓存目录yarn.nodemanager.local-dirs，这时容器的启动目录。这里包含了启动脚本、jar包以及容器启动所用的所有环境变量。这对调试 classpath 相关问题尤其有用。（注意，启用这个需要管理员权限，并重启所有的node managers，因此，对托管集群不适用）

要自定义Application Master或执行器的 log4j 配置，有如下方法： * 通过spark-submit –files 上传一个自定义的 log4j.properties 文件。 * 在 spark.driver.extraJavaOptions（对Spark驱动器）或者 spark.executor.extraJavaOptions（对Spark执行器）增加 -Dlog4j.configuration=<location of configuration file>。注意，如果使用文件，那么 file: 协议头必须显式写上，且文件必须在所节点上都存在。 * 更新 ${SPARK_CONF_DIR}/log4j.properties 文件以及其他配置。注意，如果在多个地方都配置了log4j，那么上面其他两种方法的配置优先级比本方法要高。注意，第一种方法中，执行器和Application Master共享同一个log4j配置，在有些环境下（AM和执行器在同一个节点上运行）可能会有问题（例如，AM和执行器日志都写入到同一个日志文件）

如果你需要引用YARN放置日志文件的路径，以便YARN可以正确地展示和聚合日志，请在log4j.properties文件中使用spark.yarn.app.container.log.dir。例如，log4j.appender.file_appender.File=${spark.yarn.app.container.log.dir}/spark.log 。对于流式应用，可以配置RollingFileAppender，并将文件路径设置为YARN日志目录，以避免磁盘打满，而且这些日志还可以利用YARN的日志工具访问和查看。

2.5.6 重要提示

- 对CPU资源的请求是否满足，取决于调度器如何配置和使用。
- cluster模式下，Spark执行器（executor）和驱动器（driver）的local目录都由YARN配置决定（yarn.nodemanager.local-dirs）；如果用户指定了spark.local.dir，这时候将被忽略。在client模式下，Spark执行器（executor）的local目录由YARN决定，而驱动器（driver）的local目录由spark.local.dir决定，因为这时候，驱动器不在YARN上运行。
- 选项参数–files和–archives中井号（#）用法类似于Hadoop。例如，你可以指定–files localtest.txt#appSees.txt，这将会把localtest.txt文件上传到HDFS上，并重命名为 appSees.txt，而你的程序应用用 appSees.txt来引用这个文件。
- 当你在cluster模式下使用本地文件时，使用选项–jar 才能让SparkContext.addJar正常工作，而不必使用HDFS，HTTP，HTTPS或者FTP上的文件。

2.5.7 在安全的集群中运行

在安全性方面，Kerberos用于安全的Hadoop集群，以对与服务和客户端关联的主体进行身份验证。这允许客户提出这些认证服务的请求;向经认证的校长授予权利的服务。

Hadoop服务发布hadoop令牌来授予对服务和数据的访问权限。客户必须首先获取他们将访问的服务的标记，并将其与在YARN集群中启动的应用程序一起传递。

对于Spark应用程序来与任何Hadoop文件系统（例如hdfs，webhdfs等），HBase和Hive进行交互，它必须使用启动应用程序的用户的Kerberos凭据来获取相关的令牌，也就是说，将成为Spark应用程序的启动。

这通常在启动时完成：在一个安全的集群中，Spark将自动获取集群的默认Hadoop文件系统的标记，并可能为HBase和Hive获取标记。

如果HBase位于类路径中，HBase配置声明应用程序是安全的（即，hbase-site.xml将hbase.security.authentication设置为kerberos），则将获得HBase令牌，并且spark.yarn.security.credentials.hbase.enabled没有设置为false。

同样，如果Hive位于类路径上，则会获得Hive标记，其配置包含“hive.metastore.uris”中的元数据存储的URI，并且spark.yarn.security.credentials.hive.enabled未设置为false。

如果应用程序需要与其他安全的Hadoop文件系统进行交互，则在启动时必须明确请求访问这些群集所需的令牌。这是通过将它们列在spark.yarn.access.hadoopFileSystems属性中完成的。

spark.yarn.access.hadoopFileSystems hdfs://ireland.example.org:8020/,webhdfs://frankfurt.example.org:50070/

Spark通过Java服务机制支持与其他安全感知服务的集成（请参阅java.util.ServiceLoader）。要做到这一点，org.apache.spark.deploy.yarn.security.ServiceCredentialProvider的实现应该可用于Spark，方法是将它们的名称列在jar的META-INF / services目录下的相应文件中。可以通过将spark.yarn.security.credentials {service} .enabled设置为false来禁用这些插件，其中{service}是凭证提供程序的名称。

2.5.8 配置外部的 Shuffle 服务

要在您的YARN集群中的每个NodeManager上启动Spark Shuffle服务，请按照以下说明进行操作：

1、使用YARN配置文件构建Spark。 如果您使用的是预打包发行版，请跳过此步骤。 2、找到spark- <version> -yarn-shuffle.jar。 这应该在$ SPARK_HOME / common / network-yarn / target / scala- <version>之下，如果你自己创建Spark，并且在使用分发的情况下使用yarn。 3、将此jar添加到群集中所有NodeManagers的类路径中。 4、在每个节点的yarn-site.xml中，将spark_shuffle添加到yarn.nodemanager.aux-services，然后将yarn.nodemanager.aux-services.spark_shuffle.class设置为org.apache.spark.network.yarn.YarnShuffleService。 5、通过在etc / hadoop / yarn-env.sh中设置YARN_HEAPSIZE（默认为1000）来增加NodeManager的堆大小，以避免在shuffle期间垃圾收集问题。 6、重新启动群集中的所有节点管理器。

在YARN上运行shuffle服务时，可以使用以下额外的配置选项：

2.5.9 使用 Apache Oozie启动应用程序

Apache Oozie can launch Spark applications as part of a workflow. In a secure cluster, the launched application will need the relevant tokens to access the cluster's services. If Spark is launched with a keytab, this is automatic. However, if Spark is to be launched without a keytab, the responsibility for setting up security must be handed over to Oozie. The details of configuring Oozie for secure clusters and obtaining credentials for a job can be found on the Oozie web site in the "Authentication" section of the specific release's documentation. For Spark applications, the Oozie workflow must be set up for Oozie to request all tokens which the application needs, including: * The YARN resource manager. * The local HDFS filesystem. * Any remote HDFS filesystems used as a source or destination of I/O. * Hive —if used. * HBase —if used. * The YARN timeline server, if the application interacts with this. To avoid Spark attempting —and then failing— to obtain Hive, HBase and remote HDFS tokens, the Spark configuration must be set to disable token collection for the services.

The Spark configuration must include the lines: spark.yarn.security.tokens.hive.enabled false spark.yarn.security.tokens.hbase.enabled false

The configuration option spark.yarn.access.namenodes must be unset.

2.5.10 Troubleshooting Kerberos

Debugging Hadoop/Kerberos problems can be "difficult". One useful technique is to enable extra logging of Kerberos operations in Hadoop by setting the HADOOP_JAAS_DEBUG environment variable. bash export HADOOP_JAAS_DEBUG=true The JDK classes can be configured to enable extra logging of their Kerberos and SPNEGO/REST authentication via the system propertiessun.security.krb5.debug and sun.security.spnego.debug=true -Dsun.security.krb5.debug=true -Dsun.security.spnego.debug=true All these options can be enabled in the Application Master: spark.yarn.appMasterEnv.HADOOP_JAAS_DEBUG true spark.yarn.am.extraJavaOptions -Dsun.security.krb5.debug=true -Dsun.security.spnego.debug=true Finally, if the log level for org.apache.spark.deploy.yarn.Client is set to DEBUG, the log will include a list of all tokens obtained, and their expiry details

2.5.11 使用 Spark History Server 替代 Spark Web UI

It is possible to use the Spark History Server application page as the tracking URL for running applications when the application UI is disabled. This may be desirable on secure clusters, or to reduce the memory usage of the Spark driver. To set up tracking through the Spark History Server, do the following:

- On the application side, set spark.yarn.historyServer.allowTracking=true in Spark's configuration. This will tell Spark to use the history server's URL as the tracking URL if the application's UI is disabled.
- On the Spark History Server, add org.apache.spark.deploy.yarn.YarnProxyRedirectFilter to the list of filters in the spark.ui.filters configuration.

Be aware that the history server information may not be up-to-date with the application's state.

CHAPTER 3

更多

3.1 Spark 配置

Spark 提供以下三种方式修改配置： * Spark properties （Spark属性）可以控制绝大多数应用程序参数，而且既可以通过 SparkConf 对象来设置，也可以通过Java系统属性来设置。 * Environment variables （环境变量）可以指定一些各个机器相关的设置，如IP地址，其设置方法是写在每台机器上的conf/spark-env.sh中。 * Logging （日志）可以通过log4j.properties配置日志。

3.1.1 Spark属性

Spark属性可以控制大多数的应用程序设置，并且每个应用的设定都是分开的。这些属性可以用SparkConf对象直接设定。SparkConf为一些常用的属性定制了专用方法（如，master URL和application name），其他属性都可以用键值对做参数，调用set()方法来设置。例如，我们可以初始化一个包含2个本地线程的Spark应用，代码如下：

注意，local[2]代表2个本地线程 – 这是最小的并发方式，可以帮助我们发现一些只有在分布式上下文才能复现的bug。

```
val conf = new SparkConf()
             .setMaster("local[2]")
             .setAppName("CountingSheep")
val sc = new SparkContext(conf)
```

注意，本地模式下，我们可以使用n个线程（n >= 1）。而且在像Spark Streaming这样的场景下，我们可能需要多个线程来防止类似线程饿死这样的问题。

配置时间段的属性应该写明时间单位，如下格式都是可接受的：

25ms (milliseconds) 5s (seconds) 10m or 10min (minutes) 3h (hours) 5d (days) 1y (years)

配置大小的属性也应该写明单位，如下格式都是可接受的：

1b (bytes) 1k or 1kb (kibibytes = 1024 bytes) 1m or 1mb (mebibytes = 1024 kibibytes) 1g or 1gb (gibibytes = 1024 mebibytes) 1t or 1tb (tebibytes = 1024 gibibytes) 1p or 1pb (pebibytes = 1024 tebibytes)

动态加载 Spark 属性

在某些场景下，你可能需要避免将属性值写死在 SparkConf 中。例如，你可能希望在同一个应用上使用不同的master或不同的内存总量。Spark允许你简单地创建一个空的SparkConf对象：

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

然后在运行时设置这些属性：

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

Spark shell和spark-submit工具支持两种动态加载配置的方法。第一种，通过命令行选项，如：上面提到的–master（设置master URL）。spark-submit可以在启动Spark应用时，通过–conf标志接受任何属性配置，同时有一些特殊配置参数同样可用（如，–master）。运行./bin/spark-submit –help可以展示这些选项的完整列表。

同时，bin/spark-submit 也支持从conf/spark-defaults.conf 中读取配置选项，在该文件中每行是一个键值对，并用空格分隔，如下：

```
spark.master            spark://5.6.7.8:7077
spark.executor.memory   4g
spark.eventLog.enabled  true
spark.serializer        org.apache.spark.serializer.KryoSerializer
```

这些通过参数或者属性配置文件传递的属性，最终都会在SparkConf 中合并。其优先级是：首先是SparkConf代码中写的属性值，其次是spark-submit或spark-shell的标志参数，最后是spark-defaults.conf文件中的属性。

有一些配置项被重命名过，这种情形下，老的名字仍然是可以接受的，只是优先级比新名字优先级低。

查看Spark属性

每个SparkContext都有其对应的Spark UI，所以Spark应用程序都能通过Spark UI查看其属性。默认你可以在这里看到：http://<driver>:4040，页面上的”Environment“ tab页可以查看Spark属性。如果你真的想确认一下属性设置是否正确的话，这个功能就非常有用了。注意，只有显式地通过SparkConf对象、在命令行参数、或者spark-defaults.conf设置的参数才会出现在页面上。其他属性，你可以认为都是默认值。

可用的属性

绝大多数属性都有合理的默认值。这里是部分常用的选项：

应用属性

属性名称	默认值	含义
spark.app.name	(none)	Spark应用的名字。会在SparkUI和日志中出现。
spark.driver.cores	1	在cluster模式下，用几个core运行驱动器（driver）进程。
spark.driver.maxResultSize	1g	Spark action算子返回的结果最大多大。至少要1M，可以设为0表示无限制。如果结果超过这一大小，Spark作业（job）会直接中断退出。但是，设得过高有可能导致驱动器OOM（out-of-memory）（取决于spark.driver.memory设置，以及驱动器JVM的内存限制）。设一个合理的值，以避免驱动器OOM。
spark.driver.memory	1g	驱动器进程可以用的内存总量（如：1g，2g）。注意，在客户端模式下，这个配置不能在SparkConf中直接设置（因为驱动器JVM都启动完了呀！）。驱动器客户端模式下，必须要在命令行里用 –driver-memory 或者在默认属性配置文件里设置。
spark.executor.memory	1g	单个执行器（executor）使用的内存总量（如，2g，8g）
spark.extraListeners	(none)	逗号分隔的SparkListener子类的类名列表；初始化SparkContext时，这些类的实例会被创建出来，并且注册到Spark的监听总线上。如果这些类有一个接受SparkConf作为唯一参数的构造函数，那么这个构造函数会被优先调用；否则，就调用无参数的默认构造函数。如果没有构造函数，SparkContext创建的时候会抛异常。
spark.local.dir	/tmp	Spark的”草稿“目录，包括map输出的临时文件，或者RDD存在磁盘上的数据。这个目录最好在本地文件系统中，这样读写速度快。这个配置可以接受一个逗号分隔的列表，通常用这种方式将文件IO分散不同的磁盘上去。注意：Spark-1.0及以后版本中，这个属性会被集群管理器所提供的环境变量覆盖：SPARK_LOCAL_DIRS（独立部署或Mesos）或者LOCAL_DIRS（YARN）。
spark.logConf	false	SparkContext启动时是否把生效的 SparkConf 属性以INFO日志打印到日志里
spark.master	(none)	集群管理器URL。参考allowed master URL’s.
spark.submit.deployMode	(none)	The deploy mode of Spark driver program, either “client” or “cluster”, Which means to launch driver program locally (“client”) or remotely (“cluster”) on one of the nodes inside the cluster.
spark.log.callerContext	(none)	Application information that will be written into Yarn RM log/HDFS audit log when running on Yarn/HDFS. Its length depends on the Hadoop configuration hadoop.caller.context.max.size. It should be concise, and typically can have up to 50 characters.
spark.driver.supervise	false	If true, restarts the driver automatically if it fails with a non-zero exit status. Only has effect in Spark standalone mode or Mesos cluster deploy mode.

除了这些以外，以下还有很多可用的参数配置，在某些特定情形下，可能会用到：

运行时环境

属性名	默认值	含义
spark.driver.extraClassPath	(none)	额外的classpath，将插入到到驱动器的classpath开头。注意：驱动器如果运行客户端模式下，这个配置不能通过SparkConf 在程序里配置，因为这时候程序已经启动呀！而是应该用命令行参数（–driver-class-path）或者在 conf/spark-defaults.conf 配置。
spark.driver.extraJavaOptions	(none)	驱动器额外的JVM选项。如：GC设置或其他日志参数。注意：驱动器如果运行客户端模式下，这个配置不能通过SparkConf在程序里配置，因为这时候程序已经启动呀！而是应该用命令行参数（–driver-java-options）或者conf/spark-defaults.conf 配置。
spark.driver.extraLibraryPath	(none)	启动驱动器JVM时候指定的依赖库路径。注意：驱动器如果运行客户端模式下，这个配置不能通过SparkConf在程序里配置，因为这时候程序已经启动呀！而是应该用命令行参数（–driver-library-path）或者conf/spark-defaults.conf 配置。
spark.driver.userClassPathFirst	false	试验性的，即未来不一定会支持该配置)驱动器是否首选使用用户指定的jars，而不是spark自身的。这个特性可以用来处理用户依赖和spark本身依赖项之间的冲突。目前还是试验性的，并且只能用在集群模式下。
spark.executor.extraClassPath	(none)	添加到执行器（executor）classpath开头的classpath。主要为了向后兼容老的spark版本，不推荐使用。
spark.executor.extraJavaOptions	(none)	传给执行器的额外JVM参数。如：GC设置或其他日志设置等。注意，不能用这个来设置Spark属性或者堆内存大小。Spark属性应该用SparkConf对象，或者spark-defaults.conf文件（会在spark-submit脚本中使用）来配置。执行器堆内存大小应该用spark.executor.memory配置。
spark.executor.extraLibraryPath	(none)	启动执行器JVM时使用的额外依赖库路径。
spark.executor.logs.rolling.maxRetainedFiles	(none)	Sets the number of latest rolling log files that are going to be retained by the system. Older log files will be deleted. Disabled by default.设置日志文件最大保留个数。老日志文件将被干掉。默认禁用的。
spark.executor.logs.rolling.enableCompression	false	Enable executor log compression. If it is enabled, the rolled executor logs will be compressed. Disabled by default.
spark.executor.logs.rolling.maxSize	(none)	设置执行器日志文件大小上限。默认禁用的。需要自动删日志请参考spark.executor.logs.rolling.maxRetainedFiles.
spark.executor.logs.rolling.strategy	(none)	执行器日志滚动策略。默认可接受的值有”time”（基于时间滚动） 或者 “size”（基于文件大小滚动）。time：将使用 spark.executor.logs.rolling.time.interval设置滚动时间间隔size：将使用 spark.executor.logs.rolling.size.maxBytes设置文件大小上限
spark.executor.logs.rolling.time.interval	daily	设置执行器日志滚动时间间隔。日志滚动默认是禁用的。可用的值有 “daily”, “hourly”, “minutely”，也可设为数字（则单位为秒）。关于日志自动清理，请参考spark.executor.logs.rolling.maxRetainedFiles
spark.executor.userClassPathFirst	false	试验性的，与 spark.driver.userClassPathFirst类似，只不过这个参数将应用于执行器
spark.executorEnv.[EnvironmentVariableName]	(none)	向执行器进程增加名为EnvironmentVariableName的环境变量。用户可以指定多个来设置不同的环境变量。
spark.redaction.regex	(?i)secret|password	Regex to decide which Spark configuration properties and environment variables in driver and executor environments contain sensitive information. When this regex matches a property key or value, the value is redacted from the environment UI and various logs like YARN and event logs.
spark.python.profile	false	对Python worker启用性能分析，性能分析结果会在sc.show_profile()中，或者在驱动器退出前展示。也可以用sc.dump_profiles(path)输出到磁盘上。如果部分分析结果被手动展示过，那么驱动器退出前就不再自动展示了。默认会使用pyspark.profiler.BasicProfiler，也可以自己传一个profiler 类参数给SparkContext构造函数。
spark.python.profile.dump	(none)	这个目录是用来在驱动器退出前，dump性能分析结果。性能分析结果会按RDD分别dump。同时可以使用ptats.Stats()来装载。如果制定了这个，那么分析结果就不再自动展示。
spark.python.worker.memory	512m	聚合时每个python worker使用的内存总量，和JVM的内存字符串格式相同（如，512m，2g）。如果聚合时使用的内存超过这个量，就将数据溢出到磁盘上。
spark.python.worker.reuse	true	是否复用Python worker。如果是，则每个任务会启动固定数量的Python worker，并且不需要fork() python进程。如果需要广播的数据量很大，设为true能大大减少广播数据量，因为需要广播的进程数减少了。
spark.files		Comma-separated list of files to be placed in the working directory of each executor.
spark.submit.pyFiles		Comma-separated list of .zip, .egg, or .py files to place on the PYTHONPATH for Python apps.

混洗行为

属性名	默认值	含义
spark.reducer.maxSizeInFlight	48m	从每个输出同时reduce任务获取的最大内存占用量。每个输出需要创建buffer来接收，对于每个reduce任务来说，有一个固定的内存开销上限，所以最好别设太大，除非你内存非常大。
spark.reducer.maxReqsInFlight	Int.MaxValue	This configuration limits the number of remote requests to fetch blocks at any given point. When the number of hosts in the cluster increase, it might lead to very large number of inbound connections to one or more nodes, causing the workers to fail under load. By allowing it to limit the number of fetch requests, this scenario can be mitigated.
spark.reducer.maxBlocksInFlightPerAddress	Int.MaxValue	This configuration limits the number of remote blocks being fetched per reduce task from a given host port. When a large number of blocks are being requested from a given address in a single fetch or simultaneously, this could crash the serving executor or Node Manager. This is especially useful to reduce the load on the Node Manager when external shuffle is enabled. You can mitigate this issue by setting it to a lower value.
spark.reducer.maxReqSizeShuffleToMem	Long.MaxValue	The blocks of a shuffle request will be fetched to disk when size of the request is above this threshold. This is to avoid a giant request takes too much memory. We can enable this config by setting a specific value(e.g. 200m). Note that this config can be enabled only when the shuffle shuffle service is newer than Spark-2.2 or the shuffle service is disabled.
spark.shuffle.compress	true	是否压缩map任务的输出文件。通常来说，压缩是个好主意。使用的压缩算法取决于spark.io.compression.codec
spark.shuffle.file.buffer	32k	每个混洗输出流的内存buffer大小。这个buffer能减少混洗文件的创建和磁盘寻址。
spark.shuffle.io.maxRetries	3	（仅netty）如果IO相关异常发生，重试次数（如果设为非0的话）。重试能是大量数据的混洗操作更加稳定，因为重试可以有效应对长GC暂停或者网络闪断。
spark.shuffle.io.numConnectionsPerPeer	1	（仅netty）主机之间的连接是复用的，这样可以减少大集群中重复建立连接的次数。然而，有些集群是机器少，磁盘多，这种集群可以考虑增加这个参数值，以便充分利用所有磁盘并发性能。
spark.shuffle.io.preferDirectBufs	true	（仅netty）堆外缓存可以有效减少垃圾回收和缓存复制。对于堆外内存紧张的用户来说，可以考虑禁用这个选项，以迫使netty所有内存都分配在堆上。
spark.shuffle.io.retryWait	5s	（仅netty）混洗重试获取数据的间隔时间。默认最大重试延迟是15秒，设置这个参数后，将变成maxRetries* retryWait。
spark.shuffle.service.enabled	false	启用外部混洗服务。启用外部混洗服务后，执行器生成的混洗中间文件就由该服务保留，这样执行器就可以安全的退出了。如果 spark.dynamicAllocation.enabled启用了，那么这个参数也必须启用，这样动态分配才能有外部混洗服务可用。更多请参考：dynamic allocation configuration and setup documentation
spark.shuffle.service.port	7337	外部混洗服务对应端口
spark.shuffle.service.index.cache.entries	1024	Max number of entries to keep in the index cache of the shuffle service.
spark.shuffle.sort.bypassMergeThreshold	200	（高级）在基于排序（sort）的混洗管理器中，如果没有map端聚合的话，就会最多存在这么多个reduce分区。
spark.shuffle.spill.compress	true	是否在混洗阶段压缩溢出到磁盘的数据。压缩算法取决于spark.io.compression.codec
spark.shuffle.accurateBlockThreshold	100 * 1024 * 1024	When we compress the size of shuffle blocks in HighlyCompressedMapStatus, we will record the size accurately if it's above this config. This helps to prevent OOM by avoiding underestimating shuffle block size when fetch shuffle blocks.
spark.io.encryption.enabled	false	Enable IO encryption. Currently supported by all modes except Mesos. It's recommended that RPC encryption be enabled when using this feature.
spark.io.encryption.keySizeBits	128	IO encryption key size in bits. Supported values are 128, 192 and 256.
spark.io.encryption.keygen.algorithm	HmacSHA1	The algorithm to use when generating the IO encryption key. The supported algorithms are described in the KeyGenerator section of the Java Cryptography Architecture Standard Algorithm Name Documentation.

Spark UI

属性名	默认值	含义
spark.eventLog.compress	false	是否压缩事件日志（当然spark.eventLog.enabled必须开启）
spark.eventLog.dir	file:///tmp/spark-events	Spark events日志的基础目录（当然spark.eventLog.enabled必须开启）。在这个目录中，spark会给每个应用创建一个单独的子目录，然后把应用的events log打到子目录里。用户可以设置一个统一的位置（比如一个HDFS目录），这样history server就可以从这里读取历史文件。
spark.eventLog.enabled	false	是否启用Spark事件日志。如果Spark应用结束后，仍需要在SparkUI上查看其状态，必须启用这个。
spark.ui.enabled	true	Whether to run the web UI for the Spark application.
spark.ui.killEnabled	true	允许从SparkUI上杀掉stage以及对应的作业（job）
spark.ui.port	4040	SparkUI端口，展示应用程序运行状态。
spark.ui.retainedJobs	1000	SparkUI和status API最多保留多少个spark作业的数据（当然是在垃圾回收之前）
spark.ui.retainedStages	1000	SparkUI和status API最多保留多少个spark步骤（stage）的数据（当然是在垃圾回收之前）
spark.ui.retainedTasks	100000	How many tasks the Spark UI and status APIs remember before garbage collecting. This is a target maximum, and fewer elements may be retained in some circumstances.
spark.ui.reverseProxy	false	Enable running Spark Master as reverse proxy for worker and application UIs. In this mode, Spark master will reverse proxy the worker and application UIs to enable access without requiring direct access to their hosts. Use it with caution, as worker and application UI will not be accessible directly, you will only be able to access them through spark master/proxy public URL. This setting affects all the workers and application UIs running in the cluster and must be set on all the workers, drivers and masters.
spark.ui.reverseProxyUrl		This is the URL where your proxy is running. This URL is for proxy which is running in front of Spark Master. This is useful when running proxy for authentication e.g. OAuth proxy. Make sure this is a complete URL including scheme (http/https) and port to reach your proxy.
spark.ui.showConsoleProgress	true	Show the progress bar in the console. The progress bar shows the progress of stages that run for longer than 500ms. If multiple stages run at the same time, multiple progress bars will be displayed on the same line.
spark.worker.ui.retainedExecutors	1000	SparkUI和status API最多保留多少个已结束的执行器（executor）的数据（当然是在垃圾回收之前）
spark.worker.ui.retainedDrivers	1000	SparkUI和status API最多保留多少个已结束的驱动器（driver）的数据（当然是在垃圾回收之前）
spark.sql.ui.retainedExecutions	1000	SparkUI和status API最多保留多少个已结束的执行计划（execution）的数据（当然是在垃圾回收之前）
spark.streaming.ui.retainedBatches	1000	SparkUI和status API最多保留多少个已结束的批量（batch）的数据（当然是在垃圾回收之前）
spark.ui.retainedDeadExecutors	100	How many dead executors the Spark UI and status APIs remember before garbage collecting.

压缩和序列化

属性名	默认值	含义
spark.broadcast.compress	true	是否在广播变量前使用压缩。通常是个好主意。
spark.io.compression.codec	lz4	内部数据使用的压缩算法，如：RDD分区、广播变量、混洗输出。Spark提供了3中算法：lz4，lzf，snappy。你也可以使用全名来指定压缩算法：org.apache.spark.io.LZ4CompressionCodec,org.apache.spark.io.LZFCompressionCodec,org.apache.spark
spark.io.compression.lz4.blockSize	32k	LZ4算法使用的块大小。当然你需要先使用LZ4压缩。减少块大小可以减少混洗时LZ4算法占用的内存量。
spark.io.compression.snappy.blockSize	32k	Snappy算法使用的块大小（先得使用Snappy算法）。减少块大小可以减少混洗时Snappy算法占用的内存量。
spark.kryo.classesToRegister	(none)	如果你使用Kryo序列化，最好指定这个以提高性能（tuning guide）。本参数接受一个逗号分隔的类名列表，这些类都会注册为Kryo可序列化类型。
spark.kryo.referenceTracking	false (false when using Spark SQL Thrift Server)	是否跟踪同一对象在Kryo序列化的引用。如果你的对象图中有循环护着包含统一对象的多份拷贝，那么最好启用这个。如果没有这种情况，那就禁用以提高性能。
spark.kryo.registrationRequired	false	Kryo序列化时，是否必须事先注册。如果设为true，那么Kryo遇到没有注册过的类型，就会抛异常。如果设为false（默认）Kryo会序列化未注册类型的对象，但会有比较明显的性能影响，所以启用这个选项，可以强制必须在序列化前，注册可序列化类型。
spark.kryo.registrator	(none)	如果你使用Kryo序列化，用这个class来注册你的自定义类型。如果你需要自定义注册方式，这个参数很有用。否则，使用 spark.kryo.classesRegister更简单。要设置这个参数，需要用KryoRegistrator的子类。详见：tuning guide 。
spark.kryo.unsafe	false	Whether to use unsafe based Kryo serializer. Can be substantially faster by using Unsafe Based IO.
spark.kryoserializer.buffer.max	64m	最大允许的Kryo序列化buffer。必须必你所需要序列化的对象要大。如果你在Kryo中看到”buffer limit exceeded”这个异常，你就得增加这个值了。
spark.kryoserializer.buffer	64k	Kryo序列化的初始buffer大小。注意，每台worker上对应每个core会有一个buffer。buffer最大增长到 spark.kryoserializer.buffer.max
spark.rdd.compress	false	是否压缩序列化后RDD的分区（如：StorageLevel.MEMORY_ONLY_SER）。能节省大量空间，但多消耗一些CPU。
spark.serializer	org.apache.spark.serializer.JavaSerializer (org.apache.spark.serializer.KryoSerializer when using Spark SQL Thrift Server)	用于序列化对象的类，序列化后的数据将通过网络传输，或从缓存中反序列化回来。默认的Java序列化使用java的Serializable接口，但速度较慢，所以我们建议使用usingorg.apache.spark.serializer.KryoSerializer and configuring Kryo serialization如果速度需要保证的话。当然你可以自定义一个序列化器，通过继承org.apache.spark.Serializer.
spark.serializer.objectStreamReset	100	如果使用org.apache.spark.serializer.JavaSerializer做序列化器，序列化器缓存这些对象，以避免输出多余数据，然而，这个会打断垃圾回收。通过调用reset来flush序列化器，从而使老对象被回收。要禁用这一周期性reset，需要把这个参数设为-1，。默认情况下，序列化器会每过100个对象，被reset一次。

内存管理

属性名	默认值	含义
spark.memory.fraction	0.75	堆内存中用于执行、混洗和存储（缓存）的比例。这个值越低，则执行中溢出到磁盘越频繁，同时缓存被逐出内存也更频繁。这个配置的目的，是为了留出用户自定义数据结构、内部元数据使用的内存。推荐使用默认值。请参考this description.
spark.memory.storageFraction	0.5	不会被逐出内存的总量，表示一个相对于 spark.memory.fraction的比例。这个越高，那么执行混洗等操作用的内存就越少，从而溢出磁盘就越频繁。推荐使用默认值。更详细请参考 this description.
spark.memory.offHeap.enabled	true	如果true，Spark会尝试使用堆外内存。启用 后，spark.memory.offHeap.size必须为正数。
spark.memory.offHeap.size	0	堆外内存分配的大小（绝对值）。这个设置不会影响堆内存的使用，所以你的执行器总内存必须适应JVM的堆内存大小。必须要设为正数。并且前提是spark.memory.offHeap.enabled=true.
spark.memory.useLegacyMode	false	是不使用老式的内存管理模式（1.5以及之前）。老模式在堆内存管理上更死板，使用固定划分的区域做不同功能，潜在的会导致过多的数据溢出到磁盘（如果不小心调整性能）。必须启用本参数，以下选项才可用:
spark.shuffle.memoryFraction	0.2	（废弃）必须先启用spark.memory.useLegacyMode这个才有用。混洗阶段用于聚合和协同分组的JVM堆内存比例。在任何指定的时间，所有用于混洗的内存总和不会超过这个上限，超出的部分会溢出到磁盘上。如果溢出台频繁，考虑增加spark.storage.memoryFraction的大小。
spark.storage.memoryFraction	0.6	（废弃）必须先启用spark.memory.useLegacyMode这个才有用。Spark用于缓存数据的对内存比例。这个值不应该比JVM 老生代（old generation）对象所占用的内存大，默认是60%的堆内存，当然你可以增加这个值，同时配置你所用的老生代对象占用内存大小。
spark.storage.unrollFraction	0.2	（废弃）必须先启用spark.memory.useLegacyMode这个才有用。Spark块展开的内存占用比例。如果没有足够的内存来完整展开新的块，那么老的块将被抛弃。
spark.storage.replication.proactive	false	Enables proactive block replication for RDD blocks. Cached RDD block replicas lost due to executor failures are replenished if there are any existing available replicas. This tries to get the replication level of the block to the initial number.

Executor 行为

网络

属性名	默认值	含义
spark.rpc.message.maxSize	128	Maximum message size (in MB) to allow in "control plane" communication; generally only applies to map output size information sent between executors and the driver. Increase this if you are running jobs with many thousands of map and reduce tasks and see messages about the RPC message size.
spark.blockManager.port	(随机)	块管理器（block manager）监听端口。在驱动器和执行器上都有。
spark.driver.blockManager.port	(spark.blockManager.port的值)	Port for the block manager to listen on, for cases where it cannot use the same configuration as executors.
spark.driver.bindAddress	(spark.driver.host的值)	Hostname or IP address where to bind listening sockets. This config overrides the SPARK_LOCAL_IP environment variable (see below).It also allows a different address from the local one to be advertised to executors or external systems. This is useful, for example, when running containers with bridged networking. For this to properly work, the different ports used by the driver (RPC, block manager and UI) need to be forwarded from the container's host.
spark.driver.host	(本地主机名)	驱动器主机名。用于和执行器以及独立部署时集群master通讯。
spark.driver.port	(随机)	驱动器端口。用于和执行器以及独立部署时集群master通讯。
spark.network.timeout	120s	所有网络交互的默认超时。这个配置是以下属性的默认值：spark.core.connection.ack.wait.timeout,spark.akka.timeout,spark.storage.blockManagerSlaveTimeoutMs,spark.shuffle.io.connectionTimeout,spark.rpc.askTimeout or spark.rpc.lookupTimeout
spark.port.maxRetries	16	绑定一个端口的最大重试次数。如果指定了一个端口（非0），每个后续重试会在之前尝试的端口基础上加1，然后再重试绑定。本质上，这确定了一个绑定端口的范围，就是 [start port, start port + maxRetries]
spark.rpc.numRetries	3	RPC任务最大重试次数。RPC任务最多重试这么多次。
spark.rpc.retry.wait	3s	RPC请求操作重试前等待时间。
spark.rpc.askTimeout	spark.network.timeout	RPC请求操作超时等待时间。
spark.rpc.lookupTimeout	120s	RPC远程端点查询超时。

调度

属性名	默认值	含义
spark.cores.max	(not set)	如果运行在独立部署模式下的集群或粗粒度共享模式下的Mesos集群，这个值决定了Spark应用可以使用的最大CPU总数（应用在整个集群中可用CPU总数，而不是单个机器）。如果不设置，那么独立部署时默认为spark.deploy.defaultCores，Mesos集群则默认无限制（即所有可用的CPU）。
spark.locality.wait	3s	为了数据本地性最长等待时间（spark会根据数据所在位置，尽量让任务也启动于相同的节点，然而可能因为该节点上资源不足等原因，无法满足这个任务分配，spark最多等待这么多时间，然后放弃数据本地性）。数据本地性有多个级别，每一级别都是等待这么多时间（同一进程、同一节点、同一机架、任意）。你也可以为每个级别定义不同的等待时间，需要设置spark.locality.wait.node等。如果你发现任务数据本地性不佳，可以增加这个值，但通常默认值是ok的。
spark.locality.wait.node	spark.locality.wait	单独定义同一节点数据本地性任务等待时间。你可以设为0，表示忽略节点本地性，直接跳到下一级别，即机架本地性（如果你的集群有机架信息）。
spark.locality.wait.process	spark.locality.wait	单独定义同一进程数据本地性任务等待时间。这个参数影响试图访问特定执行器上的缓存数据的任务。
spark.locality.wait.rack	spark.locality.wait	单独定义同一机架数据本地性等待时间。
spark.scheduler.maxRegisteredResourcesWaitingTime	30s	调度Job开始前，以集群管理器注册使用资源的最大等待时间。
spark.scheduler.minRegisteredResourcesRatio	YARN模式是0.8,standalone和Mesos粗粒度模式是0.0	调度启动前，需要注册得到资源的最小比例(注册到的资源数/需要资源总数)(YARN模式下，资源是执行器；独立部署和Mesos粗粒度模式下时资源是CPU core[‘spark.cores.max’是期望得到的资源总数])。可以设为0.0~1.0的一个浮点数。不管job是否得到了最小资源比例，最大等待时间都是由spark.scheduler.maxRegisteredResourcesWaitingTime控制的。
spark.scheduler.mode	FIFO	提交到同一个SparkContext上job的调度模式（scheduling mode）。另一个可接受的值是FAIR，而FIFO只是简单的把job按先来后到排队。对于多用户服务很有用。
spark.scheduler.revive.interval	1s	调度器复活worker的间隔时间。
spark.blacklist.enabled	false	If set to “true”, prevent Spark from scheduling tasks on executors that have been blacklisted due to too many task failures. The blacklisting algorithm can be further controlled by the other “spark.blacklist” configuration options.
spark.blacklist.timeout	1h	(Experimental) How long a node or executor is blacklisted for the entire application, before it is unconditionally removed from the blacklist to attempt running new tasks.
spark.blacklist.task.maxTaskAttemptsPerExecutor	1	(Experimental) For a given task, how many times it can be retried on one executor before the executor is blacklisted for that task.
spark.blacklist.task.maxTaskAttemptsPerNode	2	(Experimental) For a given task, how many times it can be retried on one node, before the entire node is blacklisted for that task.
spark.blacklist.stage.maxFailedTasksPerExecutor	2	(Experimental) How many different tasks must fail on one executor, within one stage, before the executor is blacklisted for that stage.
spark.blacklist.stage.maxFailedExecutorsPerNode	2	(Experimental) How many different executors are marked as blacklisted for a given stage, before the entire node is marked as failed for the stage.
spark.blacklist.application.maxFailedTasksPerExecutor	2	(Experimental) How many different tasks must fail on one executor, in successful task sets, before the executor is blacklisted for the entire application. Blacklisted executors will be automatically added back to the pool of available resources after the timeout specified by spark.blacklist.timeout. Note that with dynamic allocation, though, the executors may get marked as idle and be reclaimed by the cluster manager.
spark.blacklist.application.maxFailedExecutorsPerNode	2	(Experimental) How many different executors must be blacklisted for the entire application, before the node is blacklisted for the entire application. Blacklisted nodes will be automatically added back to the pool of available resources after the timeout specified by spark.blacklist.timeout. Note that with dynamic allocation, though, the executors on the node may get marked as idle and be reclaimed by the cluster manager.
spark.blacklist.killBlacklistedExecutors	false	(Experimental) If set to “true”, allow Spark to automatically kill, and attempt to re-create, executors when they are blacklisted. Note that, when an entire node is added to the blacklist, all of the executors on that node will be killed.
spark.speculation	false	如果设为true，将会启动推测执行任务。这意味着，如果stage中有任务执行缓慢，他们会被重新调度到别的节点上执行。
spark.speculation.interval	100ms	Spark检查慢任务的时间间隔。
spark.speculation.multiplier	1.5	比任务平均执行时间慢多少倍的任务会被认为是慢任务。

动态分配

属性名	默认值	含义
spark.dynamicAllocation.enabled	false	是否启用动态资源分配特性，启用后，执行器的个数会根据工作负载动态的调整（增加或减少）。注意，目前在YARN模式下不用。更详细信息，请参考：here该特性依赖于 spark.shuffle.service.enabled 的启用。同时还和以下配置相关：spark.dynamicAllocation.minExecutors, spark.dynamicAllocation.maxExecutors以及 spark.dynamicAllocation.initialExecutors
spark.dynamicAllocation.executorIdleTimeout	60s	动态分配特性启用后，空闲时间超过该配置时间的执行器都会被移除。更详细请参考这里：description
spark.dynamicAllocation.cachedExecutorIdleTimeout	Infinity	动态分配特性启用后，包含缓存数据的执行器如果空闲时间超过该配置设置的时间，则被移除。更详细请参考：description
spark.dynamicAllocation.initialExecutors	spark.dynamicAllocation.minExecutors	动态分配开启后，执行器的初始个数
spark.dynamicAllocation.maxExecutors	Infinity	动态分配开启后，执行器个数的上限
spark.dynamicAllocation.minExecutors	0	动态分配开启后，执行器个数的下限
spark.dynamicAllocation.schedulerBacklogTimeout	1s	动态分配启用后，如果有任务积压的持续时间长于该配置设置的时间，则申请新的执行器。更详细请参考：description
spark.dynamicAllocation.sustainedSchedulerBacklogTimeout	和spark.dynamicAllocation.schedulerBacklogTimeout类似，只不过该配置对应于随后持续的执行器申请。更详细请参考： description	

安全

属性名	默认值	含义
spark.acls.enable	false	是否启用Spark acls（访问控制列表）。如果启用，那么将会检查用户是否有权限查看或修改某个作业（job）。注意，检查的前提是需要知道用户是谁，所以如果用户是null，则不会做任何检查。你可以在Spark UI上设置过滤器（Filters）来做用户认证，并设置用户名。
spark.admin.acls	Empty	逗号分隔的用户列表，在该列表中的用户/管理员将能够访问和修改所有的Spark作业（job）。如果你的集群是共享的，并且有集群管理员，还有需要调试的开发人员，那么这个配置会很有用。如果想让所有人都有管理员权限，只需把该配置设置为”*”
spark.admin.acls.groups	Empty	Comma separated list of groups that have view and modify access to all Spark jobs. This can be used if you have a set of administrators or developers who help maintain and debug the underlying infrastructure. Putting a “*” in the list means any user in any group can have the privilege of admin. The user groups are obtained from the instance of the groups mapping provider specified by spark.user.groups.mapping. Check the entry spark.user.groups.mapping for more details.
spark.user.groups.mapping	org.apache.spark.security.ShellBasedGroupsMappingProvider	This list of ShellBasedGroupsMappingProvider defined by a group mapping service defined by the trait org.apache.spark.security.GroupMappingServiceProvider which can configured by this property. A default unix shell based implementation is provided org.apache.spark.security.ShellBasedGroupsMappingProvider which can be specified to resolve a list of groups for a user. Note: This implementation supports only a Unix/Linux based environment. Windows environment is currently not supported. However, a new platform/protocol can be supported by implementing the trait org.apache.spark.security.GroupMappingServiceProvider.
spark.authenticate	false	设置Spark是否认证集群内部连接。如果不是在YARN上运行，请参考spark.authenticate.secret
spark.authenticate.secret	None	设置Spark用于内部组件认证的秘钥。如果不是在YARN上运行，且启用了spark.authenticate，那么该配置必须设置
spark.network.crypto.enabled	false	Enable encryption using the commons-crypto library for RPC and block transfer service. Requires spark.authenticate to be enabled.
spark.network.crypto.keyLength	128	The length in bits of the encryption key to generate. Valid values are 128, 192 and 256.
spark.network.crypto.keyFactoryAlgorithm	PBKDF2WithHmacSHA1	The key factory algorithm to use when generating encryption keys. Should be one of the algorithms supported by the javax.crypto.SecretKeyFactory class in the JRE being used.
spark.network.crypto.saslFallback	true	Whether to fall back to SASL authentication if authentication fails using Spark’s internal mechanism. This is useful when the application is connecting to old shuffle services that do not support the internal Spark authentication protocol. On the server side, this can be used to block older clients from authenticating against a new shuffle service.
spark.network.crypto.config.*	None	Configuration values for the commons-crypto library, such as which cipher implementations to use. The config name should be the name of commons-crypto configuration without the “commons.crypto” prefix.
spark.authenticate.enableSaslEncryption	false	是否对Spark内部组件认证使用加密通信。该配置目前只有 block transfer service 使用。
spark.network.sasl.serverAlwaysEncrypt	false	是否对支持SASL认证的service禁用非加密通信。该配置目前只有 external shuffle service 支持。
spark.core.connection.ack.wait.timeout	60s	网络连接等待应答信号的超时时间。为了避免由于GC等导致的意外超时，你可以设置一个较大的值。
spark.core.connection.auth.wait.timeout	30s	网络连接等待认证的超时时间。
spark.modify.acls	Empty	逗号分隔的用户列表，在改列表中的用户可以修改Spark作业。默认情况下，只有启动该Spark作业的用户可以修改之（比如杀死该作业）。如果想要任何用户都可以修改作业，请将该配置设置为”*”
spark.modify.acls.groups	Empty	Comma separated list of groups that have modify access to the Spark job. This can be used if you have a set of administrators or developers from the same team to have access to control the job. Putting a “*” in the list means any user in any group has the access to modify the Spark job. The user groups are obtained from the instance of the groups mapping provider specified by spark.user.groups.mapping. Check the entry spark.user.groups.mapping for more details.
spark.ui.filters	None	逗号分隔的过滤器class列表，这些过滤器将用于Spark web UI。这

TLS / SSL

属性名	默认值	含义
spark.ssl.enabled	false	是否启用SSL连接（在所有所支持的协议上）。所有SSL相关配置（spark.ssl.xxx，其中xxx是一个特定的配置属性），都是全局的。如果需要在某些协议上覆盖全局设置，那么需要在该协议命名空间上进行单独配置。使用 spark.ssl.YYY.XXX 来为协议YYY覆盖全局配置XXX。目前YYY的可选值有 akka（用于基于AKKA框架的网络连接） 和 fs（用于应广播和文件服务器）
spark.ssl.[namespace].port	None	The port where the SSL service will listen on.The port must be defined within a namespace configuration; see SSL Configuration for the available namespaces.When not set, the SSL port will be derived from the non-SSL port for the same service. A value of "0" will make the service bind to an ephemeral port.
spark.ssl.enabledAlgorithms	Empty	逗号分隔的加密算法列表。这些加密算法必须是JVM所支持的。这里有个可用加密算法参考列表： this
spark.ssl.keyPassword	None	在key-store中私匙对应的密码。
spark.ssl.keyStore	None	key-store文件路径。可以是绝对路径，或者以本组件启动的工作目录为基础的相对路径。
spark.ssl.keyStorePassword	None	key-store的密码。
spark.ssl.keyStoreType	JKS	The type of the key-store.
spark.ssl.protocol	None	协议名称。该协议必须是JVM所支持的。这里有JVM支持的协议参考列表： this
spark.ssl.needClientAuth	false	Set true if SSL needs client authentication.
spark.ssl.trustStore	None	trust-store文件路径。可以是绝对路径，或者以本组件启动的工作目录为基础的相对路径。
spark.ssl.trustStorePassword	None	trust-store的密码
spark.ssl.trustStoreType	JKS	The type of the trust-store.

Spark SQL

Running the SET -v command will show the entire list of the SQL configuration.

Scala

```
// spark is an existing SparkSession
spark.sql("SET -v").show(numRows = 200, truncate = false)
```

Java

```
// spark is an existing SparkSesson
spark.sql("SET -v").show(200, false);
```

Python

```
# spark is an existing SparkSession
spark.sql("SET -v").show(n=200, truncate=False)
```

R

```
sparkR.session()
properties <- sql("SET -v")
showDF(properties, numRows = 200, truncate = FALSE)
```

Spark Streaming

属性名	默认值	含义
spark.streaming.backpressure.enabled	false	是否启用Spark Streaming的内部反压机制（spark 1.5以上支持）。启用后，Spark Streaming会根据当前批次的调度延迟和处理时长来控制接收速率，这样一来，系统的接收速度会和处理速度相匹配。该特性会在内部动态地设置接收速率。该速率的上限将由 spark.streaming.receiver.maxRate 和 spark.streaming.kafka.maxRatePerPartition 决定（如果它们设置了的话）。
spark.streaming.backpressure.initialRate	not set	This is the initial maximum receiving rate at which each receiver will receive data for the first batch when the backpressure mechanism is enabled.
spark.streaming.blockInterval	200ms	在将数据保存到Spark之前，Spark Streaming接收器组装数据块的时间间隔。建议不少于50ms。关于Spark Streaming编程指南细节，请参考 performance tuning 这一节。
spark.streaming.receiver.maxRate	not set	接收速度的最大速率（每秒记录条数）。实际上，每个流每秒将消费这么多条记录。设置为0或者负数表示不限制速率。更多细节请参考： deployment guide
spark.streaming.receiver.writeAheadLog.enable	false	是否启用接收器预写日志。所有的输入数据都会保存到预写日志中，这样在驱动器失败后，可以基于预写日志来恢复数据。更详细请参考：deployment guide
spark.streaming.unpersist	true	是否强制Spark Streaming自动从内存中清理掉所生成并持久化的RDD。同时，Spark Streaming收到的原始数据也将会被自动清理掉。如果设置为false，那么原始数据以及持久化的RDD将不会被自动清理，以便外部程序可以访问这些数据。当然，这将导致Spark消耗更多的内存。
spark.streaming.stopGracefullyOnShutdown	false	如果设为true，Spark将会在JVM关闭时，优雅地关停StreamingContext，而不是立即关闭之。
spark.streaming.kafka.maxRatePerPartition	not set	在使用Kafka direct stream API时，从每个Kafka数据分区读取数据的最大速率（每秒记录条数）。更详细请参考：Kafka Integration guide
spark.streaming.kafka.maxRetries	1	驱动器连续重试的最大次数，这个配置是为了让驱动器找出每个Kafka分区上的最大offset（默认值为1，意味着驱动器将最多尝试2次）。只对新的Kafka direct stream API有效。
spark.streaming.ui.retainedBatches	1000	Spark Streaming UI 以及 status API 中保留的最大批次个数。
spark.streaming.driver.writeAheadLog.closeFileAfterWrite	false	Whether to close the file after writing a write ahead log record on the driver. Set this to 'true' when you want to use S3 (or any file system that does not support flushing) for the metadata WAL on the driver.
spark.streaming.receiver.writeAheadLog.closeFileAfterWrite	false	Whether to close the file after writing a write ahead log record on the receivers. Set this to 'true' when you want to use S3 (or any file system that does not support flushing) for the data WAL on the receivers.

SparkR

属性名	默认值	含义
spark.r.numRBackendThreads	2	SparkR RBackEnd处理RPC调用的后台线程数
spark.r.command	Rscript	集群模式下，驱动器和worker上执行的R脚本可执行文件
spark.r.driver.command	spark.r.command	客户端模式的驱动器执行的R脚本。集群模式下会忽略
spark.r.shell.command	R	Executable for executing sparkR shell in client modes for driver. Ignored in cluster modes. It is the same as environment variable SPARKR_DRIVER_R, but take precedence over it. spark.r.shell.command is used for sparkR shell while spark.r.driver.command is used for running R script.
spark.r.backendConnectionTimeout	6000	Connection timeout set by R process on its connection to RBackend in seconds.
spark.r.heartBeatInterval	100	Interval for heartbeats sent from SparkR backend to R process to prevent connection timeout.

GraphX

属性名	默认值	含义
spark.graphx.pregel.checkpointInterval	-1	Checkpoint interval for graph and message in Pregel. It used to avoid stackOverflowError due to long lineage chains after lots of iterations. The checkpoint is disabled by default.

部署

属性名	默认值	含义
spark.deploy.recoveryMode	NONE	The recovery mode setting to recover submitted Spark jobs with cluster mode when it failed and relaunches. This is only applicable for cluster mode when running with Standalone or Mesos.
spark.deploy.zookeeper.url	None	When *spark.deploy.recoveryMode* is set to ZOOKEEPER, this configuration is used to set the zookeeper URL to connect to.
spark.deploy.zookeeper.dir	None	When *spark.deploy.recoveryMode* is set to ZOOKEEPER, this configuration is used to set the zookeeper directory to store recovery state.

集群管理器

每个集群管理器都有一些额外的配置选项。详细请参考这里：

YARN Mesos Standalone Mode

3.1.2 环境变量

有些Spark设置需要通过环境变量来设定，这些环境变量可以在${SPARK_HOME}/conf/spark-env.sh脚本（Windows下是conf/spark-env.cmd）中设置。如果是独立部署或者Mesos模式，这个文件可以指定机器相关

信息（如hostname）。运行本地Spark应用或者submit脚本时，也会引用这个文件。

注意，conf/spark-env.sh默认是不存在的。你需要复制conf/spark-env.sh.template这个模板来创建，还有注意给这个文件附上可执行权限。

以下变量可以在spark-env.sh中设置：

环境变量	含义
JAVA_HOME	Java安装目录（如果没有在PATH变量中指定）
PYSPARK_PYTHON	驱动器和worker上使用的Python二进制可执行文件（默认是python）
PYSPARK_DRIVER_PYTHON	仅在驱动上使用的Python二进制可执行文件（默认同PYSPARK_PYTHON）
SPARKR_DRIVER_R	SparkR shell使用的R二进制可执行文件（默认是R）
SPARK_LOCAL_IP	本地绑定的IP
SPARK_PUBLIC_DNS	Spark程序公布给其他机器的hostname

除了上面的环境变量之外，还有一些选项需要在Spark standalone cluster scripts里设置，如：每台机器上使用的core数量，和最大内存占用量。

spark-env.sh是一个shell脚本，因此一些参数可以通过编程方式来设定 – 例如，你可以获取本机IP来设置SPARK_LOCAL_IP。

3.1.3 日志配置

Spark使用log4j打日志。你可以在conf目录下用log4j.properties来配置。复制该目录下已有的log4j.properties.template并改名为log4j.properties即可。

3.1.4 覆盖配置目录

默认Spark配置目录是"${SPARK_HOME}/conf"，你也可以通过${SPARK_CONF_DIR}指定其他目录。Spark会从这个目录下读取配置文件（spark-defaults.conf，spark-env.sh，log4j.properties等）

3.1.5 继承Hadoop集群配置

如果你打算用Spark从HDFS读取数据，那么有2个Hadoop配置文件必须放到Spark的classpath下： * hdfs-site.xml，配置HDFS客户端的默认行为 * core-site.xml，默认文件系统名 这些配置文件的路径在不同发布版本中不太一样（如CDH和HDP版本），但通常都能在 ${HADOOP_HOME}/etc/hadoop/conf目录下找到。一些工具，如Cloudera Manager，可以动态修改配置，而且提供了下载一份拷贝的机制。

要想让这些配置对Spark可见，请在${SPARK_HOME}/spark-env.sh中设置HADOOP_CONF_DIR变量。

3.2 监控和工具

监控Spark应用有很多种方式：web UI，metrics 以及外部工具。

3.2.1 Web界面

每个SparkContext都会启动一个web UI，其默认端口为4040，并且这个web UI能展示很多有用的Spark应用相关信息。包括： * 一个stage和task的调度列表 * 一个关于RDD大小以及内存占用的概览 * 运行环境相关信息 * 运行中的执行器相关信息

你只需打开浏览器，输入 http://<driver-node>:4040 即可访问该web界面。如果有多个SparkContext在同时运行中，那么它们会从4040开始，按顺序依次绑定端口（4041,4042，等）。

注意，默认情况下，这些信息只有在Spark应用运行期内才可用。如果需要在Spark应用退出后仍然能在web UI上查看这些信息，则需要在应用启动前，将 spark.eventLog.enabled 设为 true。这项配置将会把Spark事件日志都记录到持久化存储中。

3.2.2 事后查看

Spark独立部署时，其对应的集群管理器也有其对应的 web UI。如果Spark应用将其运行期事件日志保留下来了，那么独立部署集群管理器对应的web UI将会根据这些日志自动展示已经结束的Spark应用。

如果Spark是运行于Mesos或者YARN上的话，那么你需要开启Spark的history server，开启event log。开启history server需要如下指令：

./sbin/start-history-server.sh

如果使用file-system provider class（参考下面的 spark.history.provider），那么日志目录将会基于 spark.history.fs.logDirectory 配置项，并且在表达Spark应用的事件日志路径时，应该带上子目录。history server对应的web界面默认在这里 http://<server-url>:18080。同时，history server有一些可用的配置如下：

3.2.3 环境变量

环境变量	含义
SPARK_DAEMON_MEMORY	history server分配多少内存（默认: 1g）
SPARK_DAEMON_JAVA_OPTS	history server的 JVM参数（默认：none）
SPARK_DAEMON_CLASSPATH	Classpath for the history server (default: none).
SPARK_PUBLIC_DNS	history server的外部访问地址，如果不配置，那么history server有可能会绑定server的内部地址，这可能会导致外部不能访问（默认：none）
SPARK_HISTORY_OPTS	history server配置项（默认：none）：spark.history.*

3.2.4 Spark 配置选项

属性名称	默认值	含义
spark.history.provider	org.apache.spark.deploy.history.FsHistoryProvider	Spark应用历史后台实现的类名。目前可用的只有spark自带的一个实现，支持在本地文件系统中查询应用日志。
spark.history.fs.logDirectory	file:/tmp/spark-events	history server加载应用日志的目录
spark.history.fs.update.interval	10s	history server更新信息的时间间隔。每次更新将会检查磁盘上的日志是否有更新。
spark.history.retainedApplications	50	UI上保留的spark应用历史个数。超出的将按时间排序，删除最老的。
spark.history.ui.maxApplications	Int.MaxValue	The number of applications to display on the history summary page. Application UIs are still available by accessing their URLs directly even if they are not displayed on the history summary page.
spark.history.ui.port	18080	history server绑定的端口
spark.history.kerberos.enabled	false	history server是否启用kerberos验证登陆。如果history server需要访问一个需要安全保证的hadoop集群，则需要开启这个功能。该配置设为true以后，需要同时配置 spark.history.kerberos.principal 和 spark.history.kerberos.keytab
spark.history.kerberos.principal	(none)	登陆history server的kerberos 主体名称
spark.history.kerberos.keytab	(none)	history server对应的kerberos keytab文件路径
spark.history.ui.acls.enable	false	指定是否启用ACL以控制用户访问验证。如果启用，那么不管某个应用是否设置了 spark.ui.acls.enabled，访问控制都将检查用户是否有权限。Spark应用的owner始终有查看的权限，而其他用户则需要通过 spark.ui.view.acls 配置其访问权限。如果禁用，则不会检查访问权限。
spark.history.ui.admin.acls	empty	Comma separated list of users/administrators that have view access to all the Spark applications in history server. By default only the users permitted to view the application at run-time could access the related application history, with this, configured users/administrators could also have the permission to access it. Putting a "*" in the list means any user can have the privilege of admin.
spark.history.ui.admin.acls.groups	empty	Comma separated list of groups that have view access to all the Spark applications in history server. By default only the groups permitted to view the application at run-time could access the related application history, with this, configured groups could also have the permission to access it. Putting a "*" in the list means any group can have the privilege of admin.
spark.history.fs.cleaner.enabled	false	指定history server是否周期性清理磁盘上的event log
spark.history.fs.cleaner.interval	1d	history server清理磁盘文件的时间间隔。只会清理比spark.history.fs.cleaner.maxAge 时间长的磁盘文件。
spark.history.fs.cleaner.maxAge	7d	如果启用了history server周期性清理，比这个时间长的Spark作业历史文件将会被清理掉
spark.history.fs.numReplayThreads	25% of available cores	Number of threads that will be used by history server to process event logs.

注意，所有web界面上的 table 都可以点击其表头来排序，这样可以帮助用户做一些简单分析，如：发现跑的最慢的任务、数据倾斜等。

注意history server只展示已经结束的Spark作业。一种通知Spark作业结束的方法是，显式地关闭SparkContext（通过调用 sc.stop()，或者在使用 SparkContext() 处理其 setup 和 tear down 事件（适用于python），然后作业历史就会出现在web UI上了。

3.2.5 REST API

度量信息除了可以在UI上查看之外，还可以以JSON格式访问。这能使开发人员很容易构建新的Spark可视化和监控工具。JSON格式的度量信息对运行中的Spark应用和history server中的历史作业均有效。其访问端点挂载在 /api/v1 路径下。例如，对于history server，一般你可以通过 http://<server-url>:18080/api/v1 来访问，而对于运行中的应用，可以通过 http://localhost:4040/api/v1 来访问。

端点	含义
/applications	所有应用的列表
/applications/[app-id]/jobs	给定应用的全部作业列表
/applications/[app-id]/jobs/[job-id]	给定作业的细节
/applications/[app-id]/stages	给定应用的stage列表
/applications/[app-id]/stages/[stage-id]	给定stage的所有attempt列表
/applications/[app-id]/stages/[stage-id]/[stage-attempt-id]	给定attempt的详细信息
/applications/[app-id]/stages/[stage-id]/[stage-attempt-id]/taskSummary	指定attempt对应的所有task的概要度量信息
/applications/[app-id]/stages/[stage-id]/[stage-attempt-id]/taskList	指定的attempt的所有task的列表
/applications/[app-id]/executors	给定应用的所有执行器
/applications/[app-id]/storage/rdd	给定应用的已保存的RDD列表
/applications/[app-id]/storage/rdd/[rdd-id]	给定的RDD的存储详细信息
/applications/[app-id]/logs	将给定应用的所有attempt对应的event log以zip格式打包下载
/applications/[app-id]/[attempt-id]/logs	将给定attempt的所有attempt对应的event log以zip格式打包下载

如果在YARN上运行，每个应用都由多个attempts，所以 [app-id] 实际上是 [app-id]/[attempt-id]。

3.2.6 API Versioning Policy

这些API端点都有版本号，所以基于这些API开发程序就比较容易。Spark将保证：

- 端点一旦添加进来，就不会删除
- 某个端点支持的字段永不删除
- 未来可能会增加新的端点
- 已有端点可能会增加新的字段
- 未来可能会增加新的API版本，但会使用不同的端点（如：api/v2 ）。但新版本不保证向后兼容。
- API版本可能会整个丢弃掉，但在丢弃前，一定会和新版本API共存至少一个小版本。

注意，在UI上检查运行中的应用时，虽然每次只能查看一个应用， 但applicatoins/[app-id] 这部分路径仍然是必须的。例如，你需要查看运行中应用的作业列表时，你需要输入 http://localhost:4040/api/v1/applications/{[}app-id{]}/jobs。虽然麻烦点，但这能保证两种模式下访问路径的一致性。

3.2.7 度量

Spark的度量子系统是可配置的，其功能是基于Coda Hale Metrics Library开发的。这套度量子系统允许用户以多种形式的汇报槽（sink）汇报Spark度量信息，包括：HTTP、JMX和CSV文件等。其对应的配置文件路径为：${SPARK_HOME}/conf/metrics.properties。当然，你可以通过spark.metrics.conf 这个Spark属性来自定义配置文件路径（详见configuration property）。Spark的各个组件都有其对应的度量实例，且这些度量实例

之间是解耦的。这些度量实例中，你都可以配置一系列不同的汇报槽来汇报度量信息。以下是目前支持的度量实例：

- master: 对应Spark独立部署时的master进程。
- applications: master进程中的一个组件，专门汇报各个Spark应用的度量信息。
- worker: 对应Spark独立部署时的worker进程。
- executor: 对应Spark执行器。
- driver: 对应Spark驱动器进程（即创建SparkContext对象的进程）。

每个度量实例可以汇报给0~n个槽。以下是目前 org.apache.spark.metrics.sink 包中包含的几种汇报槽（sink）： * ConsoleSink:将度量信息打印到控制台。 * CSVSink: 以特定的间隔，将度量信息输出到CSV文件。 * JmxSink: 将度量信息注册到JMX控制台。 * MetricsServlet: 在已有的Spark UI中增加一个servlet，对外提供JSON格式的度量数据。 * GraphiteSink: 将度量数据发到Graphite 节点。 * Slf4jSink: 将度量数据发送给slf4j 打成日志。 Spark同样也支持Ganglia，但因为license限制的原因没有包含在默认的发布包中： * GangliaSink: 将度量信息发送给一个Ganglia节点或者多播组。

如果需要支持GangliaSink的话，你需要自定义Spark构建包。注意，如果你包含了GangliaSink代码包的话，就必须同时将 LGPL-licensed 协议包含进你的Spark包中。对于sbt用户，只需要在编译打包前设置好环境变量：SPARK_GANGLIA_LGPL即可。对于maven用户，启用 -Pspark-ganglia-lgpl 即可。另外，除了修改集群的Spark之外，用户程序还需要链接 spark-ganglia-lgpl 工件。

度量系统配置文件语法可以参考这个配置文件示例：${SPARK_HOME}/conf/metrics.properties.template

3.2.8 高级工具

以下是几个可以用以分析Spark性能的外部工具：

- 集群整体监控工具，如：Ganglia，可以提供集群整体的使用率和资源瓶颈视图。比如，Ganglia的仪表盘可以迅速揭示出整个集群的工作负载是否达到磁盘、网络或CPU限制。
- 操作系统分析工具，如：dstat, iostat, 以及 iotop ，可以提供单个节点上细粒度的分析剖面。
- JVM工具可以帮助你分析JVM虚拟机，如：jstack可以提供调用栈信息，jmap可以转储堆内存数据，jstat可以汇报时序统计信息，jconsole可以直观的探索各种JVM属性，这对于熟悉JVM内部机制非常有用。

3.3 Spark 性能调优

由于大部分的 Spark 计算都是在内存中完成的，集群中的任何资源（CPU，网络带宽，或者内存）都可能成为 Spark 应用程序的瓶颈。最常见的情况是，数据能装进内存，而瓶颈是网络带宽；当然，有时候我们也需要做一些优化调整来减少内存占用，例如将RDD以序列化格式保存（storing RDDs in serialized form）。本文将主要涵盖两个主题：1.数据序列化（这对于优化网络性能极为重要）；2.减少内存占用以及内存调优。同时，我们也会提及其他几个比较小的主题。

3.3.1 数据序列化

序列化在任何一种分布式应用性能优化时都扮演几位重要的角色。如果序列化格式序列化过程缓慢，或者需要占用字节很多，都会大大拖慢整体的计算效率。通常，序列化都是Spark应用优化时首先需要关注的地方。Spark着眼于要达到便利性（允许你在计算过程中使用任何Java类型）和性能的一个平衡。Spark主要提供了两个序列化库： * Java serialization: 默认情况，Spark使用Java自带的ObjectOutputStream 框架来序列化对象，这样任何实现了 java.io.Serializable 接口的对象，都能被序列化。同时，你还可以通过扩展 java.io.Externalizable 来控制序列化性能。Java序列化很灵活但性能较差，同时序列化后占用的字节数也较

多。 * Kryo serialization: Spark还可以使用Kryo 库（版本2）提供更高效的序列化格式。Kryo的序列化速度和字节占用都比Java序列化好很多（通常是10倍左右），但Kryo不支持所有实现了Serializable 接口的类型，它需要你在程序中 register 需要序列化的类型，以得到最佳性能。 要切换到使用 Kryo，你可以在 SparkConf 初始化的时候调用 conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")。这个设置不仅控制各个worker节点之间的混洗数据序列化格式，同时还控制RDD存到磁盘上的序列化格式。目前，Kryo不是默认的序列化格式，因为它需要你在使用前注册需要序列化的类型，不过我们还是建议在对网络敏感的应用场景下使用Kryo。

Spark对一些常用的Scala核心类型（包括在Twitter chill 库的AllScalaRegistrar中）自动使用Kryo序列化格式。

如果你的自定义类型需要使用Kryo序列化，可以用 registerKryoClasses 方法先注册：

val conf = new SparkConf().setMaster(...).setAppName(...) conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2])) val sc = new SparkContext(conf)

Kryo的文档（Kryo documentation ）中有详细描述了更多的高级选项，如：自定义序列化代码等。

如果你的对象很大，你可能需要增大 spark.kryoserializer.buffer 配置项（config）。其值至少需要大于最大对象的序列化长度。

最后，如果你不注册需要序列化的自定义类型，Kryo也能工作，不过每一个对象实例的序列化结果都会包含一份完整的类名，这有点浪费空间。

3.3.2 内存调优

调整内存使用有三个注意事项：对象使用的内存量（您可能希望整个数据集适合内存），访问这些对象的成本，以及垃圾收集的开销（如果您有更高的营业额对象的条款）。

默认情况下，Java对象的访问速度很快，但是与其"字段"中的"原始"数据相比，可以轻松地占用多达2-5倍的空间。这是由于几个原因：

每个不同的Java对象都有一个"对象头"，大约有16个字节，并包含诸如指向其类的指针等信息。对于数据非常少的对象（比如一个Int字段），这可能比数据大。 Java字符串在原始字符串数据上有大约40字节的开销（因为它们将字符串数据存储在一个字符数组中，并保留额外的数据，如长度），并将每个字符存储为两个字节，这是由于字符串内部使用了UTF-16编码。因此一个10个字符的字符串可以很容易地消耗60个字节 常见的集合类（如HashMap和LinkedList）使用链接的数据结构，每个条目都有一个"包装器"对象（例如Map.Entry）。这个对象不仅有一个头，而且还有指向列表中下一个对象的指针（通常是8个字节）。 基本类型的集合通常将它们存储为"装箱"对象，如java.lang.Integer。 本节将首先概述Spark的内存管理，然后讨论用户可以在他/她的应用程序中更有效地使用内存的具体策略。具体来说，我们将介绍如何确定对象的内存使用情况，以及如何改进它，或者通过更改数据结构，或者以序列化格式存储数据。接下来我们将介绍Spark的缓存大小和Java垃圾收集器。

内存管理概述

Spark中的内存使用大部分属于两类：执行和存储。执行内存是指在混洗，连接，排序和聚合中用于计算的内存，而存储内存指的是用于跨群集缓存和传播内部数据的内存。在Spark中，执行和存储共享一个统一的区域（M）。当不使用执行内存时，存储器可以获取所有可用内存，反之亦然。如有必要，执行可以驱逐存储器，但是只有在总存储器内存使用量低于特定阈值（R）时才执行。换句话说，R描述了M内的一个分区，缓存块不会被驱逐。由于执行的复杂性，存储可能不会执行。

这种设计确保了几个理想的性能首先，不使用缓存的应用程序可以使用整个空间执行，避免不必要的磁盘溢出。其次，使用高速缓存的应用程序可以保留最小的存储空间（R），使数据块不受驱逐。最后，这种方法为各种工作负载提供了合理的开箱即用性能，而不需要用户如何在内部划分内存的专业知识。

虽然有两种相关配置，但典型用户不需要调整它们，因为默认值适用于大多数工作负载：

spark.memory.fraction将M的大小表示为（JVM堆空间 - 300MB）的一部分（默认值为0.6）。其余空间（40%）保留给用户数据结构，Spark中的内部元数据，并在稀疏和异常大的记录的情况下防止OOM错

误。 spark.memory.storageFraction将R的大小表示为M的一部分（默认为0.5）。 R是M中的存储空间，缓存的块不会被执行驱逐。 应该设置spark.memory.fraction的值，以便在JVM的旧时代或“终身”时代中舒适地适应这种堆空间。有关详细信息，请参阅下面对高级GC调整的讨论。

确定内存消耗

调整数据集所需的内存消耗量的最佳方法是创建RDD，将其放入缓存，然后查看Web UI中的“存储”页面。页面会告诉你RDD占用了多少内存。

要估计特定对象的内存消耗，请使用SizeEstimator的估计方法。这对于尝试使用不同数据布局来调整内存使用情况以及确定每个执行程序堆中广播变量占用的空间量非常有用。

调整数据结构

减少内存消耗的第一种方法是避免增加开销的Java功能，例如基于指针的数据结构和包装对象。 做这件事有很多种方法：

1、设计你的数据结构来优先选择对象数组和基本类型，而不是标准的Java或Scala集合类（例如HashMap）。 fastutil库为与Java标准库兼容的基本类型提供了方便的集合类。 2、尽可能避免使用大量小对象和指针的嵌套结构。 3、考虑使用数字ID或枚举对象而不是键的字符串。 4、如果RAM少于32 GB，请设置JVM标志-XX：+ UseCompressedOops使指针为4个字节而不是8个。 您可以在spark-env.sh中添加这些选项。

序列化的 RDD 存储

如果对象仍然太大，无法进行高效存储，但是使用RDD持久性API中的序列化存储级别（例如MEMORY_ONLY_SER）来减少内存使用的一种更简单的方法是以序列化的形式存储它们。 Spark然后将每个RDD分区存储为一个大字节数组。 以序列化形式存储数据的唯一缺点是访问速度较慢，这是由于必须快速反序列化每个对象。 如果你想以序列化的形式缓存数据，我们强烈推荐使用Kryo，因为它比Java序列化（当然还有原始的Java对象）要小得多。

垃圾回收调优

当您的程序存储RDD时，JVM垃圾回收会成为问题。 （在只读RDD的程序中，通常不会出现问题，然后在其上运行很多操作。）当Java需要驱逐旧对象以腾出空间给新对象时，它需要跟踪所有的Java对象并找到未使用的。这里要记住的要点是垃圾收集的成本与Java对象的数量成正比，所以使用较少对象的数据结构（例如Ints数组而不是LinkedList）大大降低了成本。更好的方法是以序列化的形式保存对象，如上所述：现在每个RDD分区只有一个对象（一个字节数组）。在尝试其他技术之前，如果GC是一个问题，首先要尝试使用序列化缓存。

由于任务的工作内存（运行任务所需的空间量）和缓存在节点上的RDD之间的干扰，GC也可能成为问题。我们将讨论如何控制分配给RDD缓存的空间来缓解这个问题。

测量GC的影响

GC调优的第一步是收集垃圾收集发生的频率和GC花费的时间。这可以通过在Java选项中添加-verbose：gc -XX：+ PrintGCDetails -XX：+ PrintGCTimeStam来完成。 （有关将Java选项传递给Spark作业的信息，请参阅配置指南。）下次运行Spark作业时，每次发生垃圾回收时都会在工作人员的日志中看到消息。请注意，这些日志将位于群集的工作节点上（在其工作目录中的stdout文件中），而不是在驱动程序上。

高级GC调整

为了进一步调整垃圾收集，我们首先需要了解JVM中有关内存管理的一些基本信息：

爪哇堆空间分为两个地区的年轻人和老人。年轻一代是为了保存短寿命的物体，而老一代则是为了寿命更长的物体。

年轻一代进一步分为三个地区[伊甸园，幸存者1，幸存者2]。

垃圾收集过程的简单描述：当Eden已满时，在Eden上运行一个小型GC，并将从Eden和Survivor1中存活的对象复制到Survivor2。幸存者地区交换。如果一个对象足够旧或者Survivor2已满，则将其移至Old。最后，当Old接近满时，调用完整的GC。

在Spark中进行GC调优的目标是确保只有长寿命的RDD才被存储在旧一代中，并且Young生成的大小足以存储短期对象。这将有助于避免完整的GC收集任务执行期间创建的临时对象。一些可能有用的步骤是：

通过收集GC统计信息来检查是否有太多的垃圾回收。如果在任务完成之前多次调用完整的GC，则意味着没有足够的内存可用于执行任务。

如果有太多次要收集，但没有太多主要地理信息，那么为伊甸园分配更多的内存将会有所帮助。您可以将Eden的大小设置为高估每个任务需要多少内存。如果Eden的大小确定为E，则可以使用选项-Xmn = 4/3 * E来设置Young代的大小。（增加4/3也是为了解释幸存者地区所使用的空间）。

在打印的GC统计信息中，如果OldGen接近满，则通过降低spark.memory.fraction来减少用于缓存的内存量;缓存更少的对象比减慢任务执行更好。或者，考虑减少年轻一代的规模。这意味着如果你按照上面的方式设置，则降低-Xmn。如果不是，请尝试更改JVM的NewRatio参数的值。许多JVM默认这个为2，这意味着老一代占2/3的堆。它应该足够大，使得这个分数超过spark.memory.fraction。

使用-XX：+ UseG1GC试用G1GC垃圾回收器。在某些垃圾收集是瓶颈的情况下，它可以提高性能。请注意，对于较大的执行程序堆大小，使用-XX：G1HeapRegionSize增加G1区大小可能很重要

例如，如果您的任务正在从HDFS中读取数据，则可以使用从HDFS读取的数据块的大小来估计该任务使用的内存量。请注意，解压缩块的大小通常是块大小的2到3倍。所以如果我们希望有3或4个任务的工作空间，HDFS块大小为128 MB，我们可以估计Eden的大小为4 * 3 * 128MB。

监视垃圾收集所花费的时间和频率如何随新设置发生变化。

我们的经验表明，GC调整的效果取决于您的应用程序和可用的内存量。在线描述的调谐选项还有很多，但在较高的层次上，管理全面GC发生的频率有助于减少开销。

GC调整标志

3.3.3 其它考虑事项

并行度

除非您将每个操作的并行度设置得足够高，否则群集不会被充分利用。 Spark会根据自己的大小（尽管可以通过可选参数控制SparkContext.textFile等）自动设置每个文件上运行的“map”任务的数量，而对于分布式的“reduce”操作，比如groupByKey和reduceByKey，它使用最大的父RDD的分区数量。 您可以将并行级别作为第二个参数（请参阅spark.PairRDDFunctions文档），或者将config属性设置为spark.default.parallelism以更改默认值。 一般来说，我们建议您的群集中每个CPU核心有2-3个任务。

Reduce 任务的内存使用

有时，你会得到一个OutOfMemoryError，不是因为你的RDD不适合内存，而是因为你的一个任务的工作集，比如groupByKey中的一个reduce任务，太大了。 Spark的shuffle操作（sortByKey，groupByKey，reduceByKey，join等）在每个任务中构建一个哈希表来执行分组，这通常会很大。 这里最简单的解决方法是增加并行度，使每个任务的输入集合更小。 Spark能够有效地支持短至200毫秒的任务，因为它可以在一个任务中重复使用一个执行器JVM，并且任务启动成本较低，因此可以安全地将并行级别提高到超过集群内核的数量。

广播超大变量

使用SparkContext中可用的广播功能可以大大减少每个序列化任务的大小，以及通过集群启动作业的成本。 如果您的任务使用其中的驱动程序的任何大对象（例如静态查找表），请考虑将其转换为广播变量。 Spark打印每个任务的序列化大小，所以你可以看看，以确定你的任务是否太大; 一般来说大于20KB的任务可能是值得优化的。

数据本地化

数据局部性可能会对Spark作业的性能产生重大影响。 如果数据和在其上运行的代码在一起，那么计算就会很快。 但是，如果代码和数据是分开的，就必须转移到另一个。 通常情况下，由于代码大小比数据小得多，所以将数据从一个地方传输到另一个地方比传输数据更快。 Spark围绕这个数据局部性的一般原则构建调度。

数据局部性是数据与代码的接近程度。 根据数据的当前位置，有几个级别的地点。 从最近到最远的顺序：

PROCESS_LOCAL 数据与运行代码位于同一个JVM中。 这是最好的地方可能 NODE_LOCAL 数据在同一个节点上。 例子可能在同一个节点上的HDFS中，或者在同一个节点上的另一个执行器上。 这比PROCESS_LOCAL稍慢，因为数据必须在进程之间传输 NO_PREF 数据可以从任何地方以相同的速度访问，并且没有本地偏好 RACK_LOCAL 数据位于同一台服务器上。 数据位于同一机架上的不同服务器上，因此需要通过网络进行发送，通常通过一台交换机进行发送 ANY 数据都在网络上的其他地方，而不在同一个机架上

Spark更喜欢在最好的地点级别安排所有任务，但这并不总是可能的。 在任何空闲的执行器上没有未处理的数据的情况下，Spark会切换到较低的地点级别。 有两种选择：a）等待一个繁忙的CPU释放，以便在同一台服务器上的数据上启动一个任务;或者b）立即在较远的地方开始一个需要移动数据的新任务。

Spark通常所做的就是等待繁忙的CPU释放的希望。 一旦超时，它就开始将数据从远处移动到空闲的CPU。 每个级别之间回退的等待超时可以单独配置，也可以全部配置在一个参数中; 有关详细信息，请参阅配置页面上的spark.locality参数。 如果你的任务很长，看到地方不好，你应该增加这些设置，但是默认情况下通常效果不错。

3.3.4 小结

这是一个简短的指南，指出调整Spark应用程序时应该了解的主要问题 - 最重要的是数据序列化和内存调整。 对于大多数程序来说，切换到Kryo序列化和以序列化形式保存数据将解决最常见的性能问题。 请随时在Spark邮件列表上询问其他调整最佳实践。

3.4 Spark 任务调度

3.4.1 概览

Spark有好几种计算资源调度的方式。首先，回忆一下集群模式概览（cluster mode overview）中每个Spark应用（包含一个SparkContext实例）中运行了一些其独占的执行器（executor）进程。集群管理器提供了Spark应用之间的资源调度（scheduling across applications）。其次，在各个Spark应用内部，各个线程可能并发地通过action算子提交多个Spark作业（job）。如果你的应用服务于网络请求，那这种情况是很常见的。在Spark应用内部（对应同一个SparkContext）各个作业之间，Spark默认FIFO调度，同时也可以支持公平调度（fair scheduler）。

3.4.2 Spark应用之间的资源调度

如果在集群上运行，每个Spark应用都会获得一批独占的执行器JVM，来运行其任务并存储数据。如果有多个用户共享集群，那么会有很多资源分配相关的选项，如何设置还取决于具体的集群管理器。

对Spark所支持的各个集群管理器而言，最简单的资源分配，就是对资源静态划分。这种方式就意味着，每个Spark应用都是设定一个最大可用资源总量，并且该应用在整个生命周期内都会占住这些资源。这种方式在Spark独立部署（standalone）和YARN调度，以及Mesos粗粒度模式（coarse-grained Mesos mode）下都可用。

- Standalone mode: 默认情况下，Spark应用在独立部署的集群中都会以FIFO（first-in-first-out）模式顺序提交运行，并且每个Spark应用都会占用集群中所有可用节点。不过你可以通过设置spark.cores.max或者spark.deploy.defaultCores 来限制单个应用所占用的节点个数。最后，除了可以控制对CPU的使用数量之外，还可以通过 spark.executor.memory 来控制各个应用的内存占用量。
- Mesos: 在Mesos中要使用静态划分的话，需要将 spark.mesos.coarse 设为true，同样，你也需要设置 spark.cores.max 来控制各个应用的CPU总数，以及 spark.executor.memory 来控制各个应用的内存占用。
- YARN: 在YARN中需要使用 –num-executors 选项来控制Spark应用在集群中分配的执行器的个数，对于单个执行器（executor）所占用的资源，可以使用 –executor-memory 和 –executor-cores 来控制。

Mesos上另一种可用的方式是动态共享CPU。在这种模式下，每个Spark应用的内存占用仍然是固定且独占的（仍由spark.executor.memory决定），但是如果该Spark应用没有在某个机器上执行任务的话，那么其他应用可以占用该机器上的CPU。这种模式对集群中有大量不是很活跃应用的场景非常有效，例如：集群中有很多不同用户的Spark shell session。但这种模式不适用于低延迟的场景，因为当Spark应用需要使用CPU的时候，可能需要等待一段时间才能取得CPU的使用权。要使用这种模式，只需要在 mesos:// URL 上设置 spark.mesos.coarse 属性为flase即可。

注意，目前还没有任何一种资源分配模式能支持跨Spark应用的内存共享。如果你需要跨Spark应用共享内存，我们建议你用单独用一个server来计算和保留同一个RDD查询的结果，这样就能在多个请求（request）之间共享同一个RDD的数据。在未来的发布版本中，一些内存存储系统（如：Tachyon）或许能够提供这种跨Spark应用共享RDD的能力。

动态资源分配

Spark还提供了一种基于负载来动态调节Spark应用资源占用的机制。这意味着，你的应用会在资源空闲的时候将其释放给集群，而后续用到的时候再重新申请。这一特性在多个应用共享Spark集群资源的情况下特别有用。

注意，这个特性默认是禁用的，但是在所有的粗粒度集群管理器上都是可用的，如：独立部署模式（standalone mode），YARN模式（YARN mode）以及Mesos粗粒度模式（Mesos coarse-grained mode）。

配置和部署

要使用这一特性有两个前提条件。首先，你的应用必须设置 spark.dynamicAllocation.enabled 为 true。其次，你必须在每个节点上启动一个外部混洗服务（external shuffle service），并在你的应用中将 spark.shuffle.service.enabled 设为true。外部混洗服务的目的就是为了在删除执行器的时候，能够保留其输出的混洗文件（本文后续有更详细的描述）。启用外部混洗的方式在各个集群管理器上各不相同：

在Spark独立部署的集群中，你只需要在worker启动前设置 spark.shuffle.server.enabled 为true即可。

在Mesos粗粒度模式下，你需要在各个节点上运行 ${SPARK_HOME}/sbin/start-mesos-shuffle-service.sh 并设置 spark.shuffle.service.enabled 为true 即可。例如，你可以用Marathon来启用这一功能。

在YARN模式下，参照这里的说明。

所有相关的配置都是可选的，并且都在 spark.dynamicAllocation.* 和 spark.shuffle.service.* 命名空间下。更详细请参考：configurations page。

资源分配策略

总体上来说，Spark应该在执行器空闲时将其关闭，而在后续要用时再次申请。因为没有一个固定的方法，可以预测一个执行器在后续是否马上回被分配去执行任务，或者一个新分配的执行器实际上是空闲的，所以我们需要一些试探性的方法，来决定是否申请或移除一个执行器。

请求策略

一个启用了动态分配的Spark应用会在有等待任务需要调度的时候，申请额外的执行器。这种情况下，必定意味着已有的执行器已经不足以同时执行所有未完成的任务。

Spark会分轮次来申请执行器。实际的资源申请，会在任务挂起 spark.dynamicAllocation.schedulerBacklogTimeout 秒后首次触发，其后如果等待队列中仍有挂起的任务，则每过 spark.dynamicAlloction.sustainedSchedulerBacklogTimeout 秒触发一次资源申请。另外，每一轮所申请的执行器个数以指数形式增长。例如，一个Spark应用可能在首轮申请1个执行器，后续的轮次申请个数可能是2个、4个、8个... ... 。

采用指数级增长策略的原因有两个：第一，对于任何一个Spark应用如果只是需要多申请少数几个执行器的话，那么必须非常谨慎地启动资源申请，这和TCP慢启动有些类似；第二，如果一旦Spark应用确实需要申请很多个执行器的话，那么可以确保其所需的计算资源及时地增长。

移除策略

移除执行器的策略就简单多了。Spark应用会在某个执行器空闲超过 spark.dynamicAllocation.executorIdleTimeout 秒后将其删除。在绝大多数情况下，执行器的移除条件和申请条件都是互斥的，也就是说，执行器在有待执行任务挂起时，不应该空闲。

优雅地关闭执行器

非动态分配模式下，执行器可能的退出原因有执行失败或者相关Spark应用已经退出。不管是那种原因，执行器的所有状态都已经不再需要，可以丢弃掉。但在动态分配的情形下，执行器有可能在Spark应用运行期间被移除。这时候，如果Spark应用尝试去访问该执行器存储的状态，就必须重算这一部分数据。因此，Spark需要一种机制，能够优雅地关闭执行器，同时还保留其状态数据。

这种需求对于混洗操作尤其重要。混洗过程中，Spark执行器首先将map输出写到本地磁盘，同时执行器本身又是一个文件服务器，这样其他执行器就能够通过该执行器获得对应的map结果数据。一旦有某些任务执行时间过长，动态分配有可能在混洗结束前移除任务异常的执行器，而这些被移除的执行器对应的数据将会被重新计算，但这些重算其实是不必要的。

要解决这一问题，就需要用到一个外部混洗服务（external shuffle service），该服务在Spark 1.2引入。该服务在每个节点上都会启动一个不依赖于任何Spark应用或执行器的独立进程。一旦该服务启用，Spark执行器不再从各个执行器上获取shuffle文件，转而从这个service获取。这意味着，任何执行器输出的混洗状态数据都可能存留时间比对应的执行器进程还长。

除了混洗文件之外，执行器也会在磁盘或者内存中缓存数。一旦执行器被移除，其缓存数据将无法访问。这个问题目前还没有解决。或许在未来的版本中，可能会采用外部混洗服务类似的方法，将缓存数据保存在堆外存储中以解决这一问题。

3.4.3 Spark应用内部的资源调度

在指定的Spark应用内部（对应同一SparkContext实例），多个线程可能并发地提交Spark作业（job）。在本节中，作业（job）是指，由Spark action算子（如：collect）触发的一系列计算任务的集合。Spark调度器是完全线程安全的，并且能够支持Spark应用同时处理多个请求（比如：来自不同用户的查询）。

默认，Spark应用内部使用FIFO调度策略。每个作业被划分为多个阶段（stage）（例如：map阶段和reduce阶段），第一个作业在其启动后会优先获取所有的可用资源，然后是第二个作业再申请，再第三个.....。如果前面的作业没有把集群资源占满，则后续的作业可以立即启动运行，否则，后提交的作业会有明显的延迟等待。

不过从Spark 0.8开始，Spark也能支持各个作业间的公平（Fair）调度。公平调度时，Spark以轮询的方式给每个作业分配资源，因此所有的作业获得的资源大体上是平均分配。这意味着，即使有大作业在运行，小

的作业再提交也能立即获得计算资源而不是等待前面的作业结束，大大减少了延迟时间。这种模式特别适合于多用户配置。

要启用公平调度器，只需设置一下 SparkContext中spark.scheduler.mode 属性为 FAIR即可：

```
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.set("spark.scheduler.mode", "FAIR")
val sc = new SparkContext(conf)
```

公平调度资源池

公平调度器还可以支持将作业分组放入资源池（pool），然后给每个资源池配置不同的选项（如：权重）。这样你就可以给一些比较重要的作业创建一个“高优先级”资源池，或者你也可以把每个用户的作业分到一组，这样一来就是各个用户平均分享集群资源，而不是各个作业平分集群资源。Spark公平调度的实现方式基本都是模仿 Hadoop Fair Scheduler 来实现的。

默认情况下，新提交的作业都会进入到默认资源池中，不过作业对应于哪个资源池，可以在提交作业的线程中用SparkContext.setLocalProperty 设定 spark.scheduler.pool 属性。示例代码如下：

```
// Assuming sc is your SparkContext variable
sc.setLocalProperty("spark.scheduler.pool", "pool1")
```

一旦设好了局部属性，所有该线程所提交的作业（即：在该线程中调用action算子，如：RDD.save/count/collect 等）都会使用这个资源池。这个设置是以线程为单位保存的，你很容易实现用同一线程来提交同一用户的所有作业到同一个资源池中。同样，如果需要清除资源池设置，只需在对应线程中调用如下代码：

```
sc.setLocalProperty("spark.scheduler.pool", null)
```

资源池默认行为

默认地，各个资源池之间平分整个集群的资源（包括default资源池），但在资源池内部，默认情况下，作业是FIFO顺序执行的。举例来说，如果你为每个用户创建了一个资源池，那么久意味着各个用户之间共享整个集群的资源，但每个用户自己提交的作业是按顺序执行的，而不会出现后提交的作业抢占前面作业的资源。

配置资源池属性

资源池的属性需要通过配置文件来指定。每个资源池都支持以下3个属性：

- schedulingMode：可以是FIFO或FAIR，控制资源池内部的作业是如何调度的。
- weight：控制资源池相对其他资源池，可以分配到资源的比例。默认所有资源池的weight都是1。如果你将某个资源池的weight设为2，那么该资源池中的资源将是其他池子的2倍。如果将weight设得很高，如1000，可以实现资源池之间的调度优先级 – 也就是说，weight=1000的资源池总能立即启动其对应的作业。
- minShare：除了整体weight之外，每个资源池还能指定一个最小资源分配值（CPU个数），管理员可能会需要这个设置。公平调度器总是会尝试优先满足所有活跃（active）资源池的最小资源分配值，然后再根据各个池子的weight来分配剩下的资源。因此，minShare属性能够确保每个资源池都能至少获得一定量的集群资源。minShare的默认值是0。

资源池属性是一个XML文件，可以基于 conf/fairscheduler.xml.template 修改，然后在 SparkConf 的 spark.scheduler.allocation.file 属性指定文件路径：

```
conf.set("spark.scheduler.allocation.file", "/path/to/file")
```

资源池XML配置文件格式如下，其中每个池子对应一个<pool>元素，每个资源池可以有其独立的配置:

```
<?xml version="1.0"?>
<allocations>
  <pool name="production">
    <schedulingMode>FAIR</schedulingMode>
    <weight>1</weight>
    <minShare>2</minShare>
  </pool>
  <pool name="test">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>3</minShare>
  </pool>
</allocations>
```

完整的例子可以参考 conf/fairscheduler.xml.template。注意，没有在配置文件中配置的资源池都会使用默认配置（schedulingMode：FIFO，weight：1，minShare：0）。

3.5 Spark安全

Spark 目前已经支持以共享秘钥的方式进行身份认证。开启身份认证配置参数为 spark.authenticate。这个配置参数决定了Spark通讯协议是否使用共享秘钥做身份验证。验证过程就是一个基本的握手过程，确保通讯双方都有相同的秘钥并且可以互相通信。如果共享秘钥不同，双方是不允许通信的。共享秘钥可用以下方式创建:

- 对于以 YARN 方式部署的 Spark，将 spark.authenticate 设为true可以自动生成并分发共享秘钥。每个 Spark 应用会使用唯一的共享秘钥。
- 而对于其他部署类型，需要在每个节点上设置 spark.authenticate.secret 参数。这个秘钥将会在由所有 Master/Workers以及各个Spark应用共享。

3.5.1 Web UI

Spark UI 也可以通过配置 spark.ui.filters 来使用 javax servlet filters 确保安全性，

认证

因为某些用户可能希望web UI上的某些数据应该保密，并对其他用户不可见。用户可以自定义 javax servlet filter 来对登陆用户进行认证，Spark会根据用户的ACL（访问控制列表）来确保该登陆用户有权限访问某个Spark应用的web UI。Spark ACL的行为可由 spark.acls.enable 和 spark.ui.view.acls 共同控制。注意，启动Spark应用的用户总是会有权限访问该应用对应的UI。而在YARN模式下，Spark web UI会使用YARN的web应用代理机制，通过Hadoop过滤器进行认证。

Spark还支持修改运行中的Spark应用对应的ACL以便更改其权限控制，不过这可能会导致杀死应用或者杀死任务的动作。这一特性由 spark.acls.enable 和 spark.modify.acls 共同控制。注意，如果你需要web UI的认证，比如为了能够在web UI上使用杀死应用的按钮，那么就需要将用户同时添加到modify ACL和view ACL中。在YARN上，控制用户修改权限的modify ACL是通过YARN接口传进来的。

Spark可以允许多个管理员共同管理ACL，而且这些管理员总是能够访问和修改所有的Spark应用。而管理员本身是由 spark.admin.acls 配置的。在一个共享的多用户集群中，你可能需要配置多个管理员，另外，该配置也可以用于支持开发人员调试Spark应用。

3.5.2 事件日志

如果你启用了事件日志（event logging），那么这些日志对应的目录（spark.eventLog.dir）需要事先手动创建并设置好权限。如果你要限制这些日志文件的权限，首先还是要将日志目录的权限设为 drwxrwxrwxt，目录owner应该是启动history server的超级用户，对应的组限制为该超级用户所在组。这样其他所有用户就只能往该目录下写日志，但同时又不能删除或改名。事件日志文件只有在用户或者用户组具有读写权限时才能写入。

3.5.3 加密

Spark对Akka和HTTP协议（广播和文件服务器中使用）支持SSL。同时，对数据块传输服务（block transfer service）支持SASL加密。Spark web UI暂时还不支持任何加密。

Spark的临时数据存储，如：混洗中间文件、缓存数据或其他应用相关的临时文件，目前也没有加密。如果需要加密这些数据，只能通过配置集群管理器将数据存储到加密磁盘上。

SSL配置

Spark SSL相关配置是层级式的。用户可以配置SSL的默认配置，以便支持所有的相关通讯协议，当然，如果设置了针对某个具体协议配置值，其值将会覆盖默认配置对应的值。这种方式主要是为了方便用户，用户可以轻松地为所有协议都配置好默认值，同时又不会影响针对某一个具体协议的特殊配置需求。通用的SSL默认配置在 spark.ssl 这一配置命名空间下，对于Akka的SSL配置，在spark.ssl.akka下，而对用于广播和文件服务器中的HTTP协议，其配置在 spark.ssl.fs 下。详细的清单见配置指南（configuration page）。

SSL必须在每个节点上都配置好，并且包括各个使用特定通讯协议的相关模块。

YARN模式

key-store 文件可以由客户端准备好，然后作为Spark应用的一部分分发到各个执行器（executor）上使用。用户也可以在Spark应用启动前，通过配置 spark.yarn.dist.files 或者 spark.yarn.dist.archives 来部署key-store文件。这些文件传输过程的加密是由YARN本身负责的，和Spark就没什么关系了。

对于一些长期运行并且可以写HDFS的Spark应用，如：Spark Streaming 上的应用，可以用 spark-submit 的 –principal 和 –keytab 参数分别设置principal 和 keytab 信息。keytab文件将会通过 Hadoop Distributed Cache（如果YARN配置了 SSL并且HDFS启用了加密，那么分布式缓存的传输也会被加密） 复制到Application Master所在机器上。Kerberos的登陆信息将会被principal和keytab周期性地刷新，同时HDFS所需的代理token也会被周期性的刷新，这样Spark应用就能持续地写入HDFS了。

独立模式

独立模式下，用户需要为master和worker分别提供key-store和相关配置选项。这些配置可以通过在SPARK_MASTER_OPTS 和 SPARK_WORKER_OPTS，或者 SPARK_DEAMON_JAVA_OPTS环境变量中添加相应的java系统属性来设置。独立模式下，用户可以通过worker的SSL设置来改变执行器（executor）的配置，因为这些执行器进程都是worker的子进程。不过需要注意的是，执行器如果需要启用本地SSL配置值（如：从worker进程继承而来的环境变量），而不是用户在客户端设置的值，就需要将 spark.ssl.userNodeLocalConf 设为 true。

准备key-stores

key-stores 文件可以由 keytool 程序生成。keytool 相关参考文档见这里：here。独立模式下，配置key-store和trust-store至少有这么几个基本步骤:

- 为每个节点生成一个秘钥对
- 导出各节点秘钥对中的公匙（public key）到一个文件
- 将所有这些公匙导入到一个 trust-store 文件中
- 将该trust-store文件发布到所有节点

配置SASL加密

启用认证后（spark.authenticate），数据块传输服务（block transfer service）可以支持SASL加密。如需启用SASL加密的话，还需要在 Spark 应用中设置 spark.authenticate.enableSaslEncryption 为 true。

如果是开启了外部混洗服务（external shuffle service），那么只需要将 spark.network.sasl.serverAlwaysEncrypt 设为true即可禁止非加密的网络连接。因为这个配置一旦启用，所有未使用 SASL加密的Spark应用都无法连接到外部混洗服务上。

3.5.4 配置网络安全端口

Spark计算过程中大量使用网络通信，而有些环境中对网络防火墙的设置要求很严格。下表列出来Spark用于通讯的一些主要端口，以及如何配置这些端口。

仅独立部署适用

访问源	访问目标	默认端口	用途	配置	注意
Browser	Standalone Master	8080	Web UI	spark.master.ui.port/SPARK_MASTER_WEBUI_PORT	基于Web UI（仅独立模式有效。
Browser	Standalone Worker	8081	Web UI	spark.worker.ui.port/SPARK_WORKER_WEBUI_PORT	基于Web UI（仅独立模式有效。
Driver/Standalone Worker	Standalone Master	7077	提交作业/合并集群	SPARK_MASTER_PORT	设为0表示随机。仅独立模式有效。
Standalone Master	Standalone Worker	(random)	调度执行器	SPARK_WORKER_PORT	设为0表示随机。仅独立模式有效。

所有集群管理器适用

更多关于安全方面的配置参数，请参考配置指南（configuration page），安全方面的一些实现细节可以参考org.apache.spark.SecurityManager。

3.6 硬件配置

Spark 开发者们常常被问到的一个问题就是：如何为 Spark 配置硬件。但具体的硬件配置还依赖于实际的使用情况, 我们通常会给出以下的建议。

3.6.1 存储系统

因为绝大多数Spark作业都很可能是从外部存储系统加载输入数据（如：HDFS 或者 HBase），所以最好把 Spark 部署在离这些存储比较近的地方。建议如下：

- 只要有可能，就尽量在 HDFS 相同的节点上部署 Spark。最简单的方式就是，在 HDFS 相同的节点上独立部署 Spark（standalone mode cluster），并配置好 Spark 和 Hadoop 的内存和 CPU 占用，以避免互相干扰（对 Hadoop 来说，相关的选项有 mapred.child.java.opts – 配置单个任务的内存，mapred.tasktracker.map.tasks.maximun和mapred.tasktracker.reduce.tasks.maximum – 配置任务个数）。当然，你也可以在一些通用的集群管理器上同时运行 Hadoop 和 Spark，如：Mesos 或 Hadoop YARN。
- 如果不能将 Spark 和 HDFS 放在一起，那么至少要将它们部署到同一局域网的节点中。
- 对于像 HBase 这类低延迟数据存储来说，比起一味地避免存储系统的互相干扰，更需要关注的是将计算分布到不同节点上去。

3.6.2 本地磁盘

虽然大部分情况下，Spark 都是在内存里做计算，但它仍会使用本地磁盘存储数据，如：存储无法装载进内存的 RDD 数据，存储各个阶段（stage）输出的临时文件。因此，我们建议每个节点上用4~8块磁盘，非磁盘阵列方式挂载（只需分开使用单独挂载点即可）。在 Linux 中，挂载磁盘时使用 noatime option 可以减少不必要的写操作。在Spark中，配置（configure）spark.local.dir 属性可指定Spark使用的本地磁盘目录，其值可以是逗号分隔的列表以指定多个磁盘目录。如果该节点上也有 HDFS 目录，可以和 HDFS 共用同一个块磁盘。

3.6.3 内存

一般来说，Spark 可以在8GB~几百GB内存的机器上运行得很好。不过，我们还是建议最多给 Spark 分配75%的内存，剩下的内存留给操作系统和系统缓存。

每次计算具体需要多少内存，取决于你的应用程序。如需评估你的应用程序在使用某个数据集时会占用多少内存，可以尝试先加载一部分数据集，然后在 Spark 的监控UI（http://<driver-node>:4040）上查看其占用内存大小。需要注意的是，内存占用很大程度受存储级别和序列化格式影响 – 更多内存优化建议，请参考调优指南（tuning guide）。

最后，还需要注意的是，Java 虚拟机在 200GB 以上内存的机器上并非总是表现良好。如果你的单机内存大于200GB，建议在单个节点上启动多个 worker JVM。在Spark独立部署模式下（standalone mode），你可在conf/spark-env.sh 中设置 SPARK_WORKER_INSTANCES 来配置单节点上worker个数，而且在该文件中你还可以通过 SPARK_WORKER_CORES 设置单个worker占用的CPU core个数。

3.6.4 网络

以我们的经验来说，如果数据能加载进内存，那么多数Spark应用的瓶颈都是网络带宽。对这类应用，使用万兆网（10 Gigabit）或者更强的网络是最好的优化方式。对于一些包含有分布式归约相关算子（distributed reduce相关算子，如：group-by系列，reduce-by系列以及SQL join系列）的应用尤其是如此。对于任何一个应用，你可以在监控UI (http://<driver-node>:4040) 上查看Spark混洗跨网络传输了多少数据量。

3.6.5 CPU Cores

Spark 在单机几十个 CPU 的机器上也能表现良好，因为 Spark 尽量减少了线程间共享的数据。但一般你至少需要单机8~16个CPU cores。当然，根据具体的计算量你可能需要更多的CPU，但是：一旦数据加载进内存，绝大多数应用的瓶颈要么是 CPU，要么是网络。

CHAPTER 4

Indices and tables

- genindex
- modindex
- search