PARTⅠ Hadoop Command

一. 概述

所有的 hadoop 命令 由 bin / hadoop 脚本调用。不带任何参数运行 hadoop 脚本会打印所有命令的描述。用法：

```
hadoop [--config confdir] [COMMAND] [GENERIC_OPTIONS] [COMMAND_OPTIONS]
```

二. 用户命令

1. archive

Hadoop Archive 是一个高效地将小文件放入 HDFS 块中的文件存档文件格式，它能够将多个小文件打包成一个后缀为.har 文件，这样减少 namenode 内存使用的同时，仍然允许对文件进行透明的访问。用法：

```
hadoop archive -archiveName NAME -p <parent path> <src>* <dest>
```

COMMAND_OPTION	Description
-archiveName NAME	创建的归档文件名字
src	源文件系统的路径名
dest	保存档案文件的目标目录

例子：将/user/tom/cs 下的 11 文件夹归档到/user/tom/33 文件夹下

bin/hadoop archive -archiveName 11.har -p /user/tom cs/11/ 33

2. distcp

用于集群内部或者集群之间递归地拷贝文件或目录。用法:

```
hadoop distcp [选项] <srcurl> <desturl>
```

选项	Description
-m	表示启用多少 map
-delete	删除已经存在的目标文件，不会删除源文件
-i	忽略失败
-overwrite	覆盖目标
-update	更新目标

例子：

```
bin/hadoop distcp hdfs://A:9000/user/foo/bar hdfs://B:9000/user/foo/baz
```

3. fs

用法: `bin/hadoop fs [GENERIC_OPTIONS] [COMMAND_OPTIONS]`

选项名称	使用格式	含义
-ls	-ls <路径>	查看指定路径的当前目录结构
-lsr	-lsr <路径>	查看指定路径的当前目录结构
-du	-du <路径>	统计目录下各文件大小
-dus	-dus <路径>	统计目录下所有文件大小
-count	-count [-q] <路径>	统计文件(夹)数量
-mv	-mv <源路径> <目的路径>	移动
-cp	-cp <源路径> <目的路径>	复制
-rm	-rm [-skiptrash] <路径>	删除文件/空白文件夹
-rmr	-rmr [-skiptrash] <路径>	递归删除
-put	-put <linux 上的文件> <hdfs 路径>	从本地上传文件到 hdfs
-copyFromLocal	-copyFromLocal <linux 上的文件> <hdfs 路径>	从本地复制到 hdfs
-moveFromLocal	-moveFromLocal <linux 上的文件> <hdfs 路径>	从本地移动到 hdfs
-getmerge	-getmerge <源路径> <linux 路径>	合并到本地
-cat	-cat <hdfs 路径>	查看文件内容
-text	-text <hdfs 路径>	查看文件内容
-copyToLocal	-copyToLocal [-ignorecrc] [-crc] [hdfs 源路径] [linux 目的路径]	从 hdfs 复制到本地
-moveToLocal	-moveToLocal [-crc] <hdfs 源路径> <linux 目的路径>	从 hdfs 移动到本地
-mkdir	-mkdir <hdfs 路径>	创建空白文件夹
-setrep	-setrep [-r] [-w] <副本数> <路径>	修改副本数量
-touchz	-touchz <文件路径>	创建空白文件
-stat	-stat [format] <路径>	显示文件统计信息
-tail	-tail [-f] <文件>	查看文件尾部信息
-chmod	-chmod [-r] <权限模式> [路径]	修改权限
-chown	-chown [-r] [属主][:[属组]] 路径	修改属主
-chgrp	-chgrp [-r] 属组名称 路径	修改属组
-help	-help [命令选项]	帮助
-get	-put <hdfs 路径> <linux 上的文件>	从 hdfs 上下载文件到本地
-expunge	-expunge	清空回收站
-test	-test -[ezd] <路径>	-e 检查文件是否存在。如果存在则返回 0。 -z 检查文件是否是 0 字节。如果是则返回 0。 -d 如果路径是个目录，则返回 1，否则返回 0。

注意：以上表格中路径包括 hdfs 中的路径和 linux 中的路径。对于容易产生歧义的地方，会特别指出“linux 路径”或者“hdfs 路径”。如果没有明确指出，意味着是 hdfs 路径。

4. fsck

用来检查整个文件系统的健康状况，但是要注意它不会主动恢复备份缺失的 block，

这个是由 NameNode 单独的线程异步处理的。用法：

```
bin/hadoop fsck [GENERIC_OPTIONS] [-move|-delete|-openforwrite] [-files [-blocks [-locations | -racks]]]
```

例子：`bin/hadoop fsck /`

5. jar

运行 jar 文件。用户可以把他们的 Map Reduce 代码捆绑到 jar 文件中，使用这个命令执行。用法：

```
hadoop jar <jar> [mainClass] args...
```

例子：在集群上运行 Map Reduce 程序，以 WordCount 程序为例

```
bin/hadoop jar /home/hadoop/hadoop-1.1.1/hadoop-examples.jar wordcount input output
```

wordcount 是程序主类名，input 和 output 是输入输出文件夹。

6. job

用于和 Map Reduce 作业交互和命令。用法：

```
hadoop job [GENERIC_OPTIONS] [-submit ] | [-status ] | [-counter ] | [-kill ] | [-events <#-of-events>] | [-history [all] ] | [-list [all]] | [-kill-task ] | [-fail-task ]
```

参数选项	Description
-submit <job-file>	提交作业
-status <job-id>	打印 map 和 reduce 完成百分比和所有计数器
-counter <job-id> <group-name> <counter-name>	打印计数器的值
-kill <job-id>	杀死指定作业
-events <job-id> <from-event-#> <#-of-events>	打印给定范围内 jobtracker 接收到的事件细节
-history [all]	打印作业的细节、失败及被杀死原因的细节
-list [all]	-list all 显示所有作业。-list 只显示将要完成的作业
-kill-task <task-id>	杀死任务。被杀死的任务不会不利于失败尝试
-fail-task <task-id>	使任务失败。被失败的任务会对失败尝试不利

三. 管理命令

1. datanode

运行一个 HDFS 的 datanode。用法：

```
hadoop datanode [-rollback]
```

-rollback：将 datanode 回滚到前一个版本。这需要在停止 datanode，分发老的 hadoop 版本之

后使用。

2. namenode

运行 namenode。用法：

`hadoop namenode [-format] | [-upgrade] | [-rollback] | [-finalize] | [-importCheckpoint]`

参数选项	Description
-format	格式化 namenode。它启动 namenode，格式化 namenode，之后关闭 namenode
-upgrade	分发新版本的 hadoop 后，namenode 应以 upgrade 选项启动
-rollback	将 namenode 回滚到前一版本。这个选项要在停止集群，分发老的 hadoop 版本后使用
-finalize	finalize 会删除文件系统的前一状态。最近的升级会被持久化，rollback 选项将再不可用，升级终结操作之后，它会停掉 namenode
-importCheckpoint	从检查点目录装载镜像并保存到当前检查点目录，检查点目录由 fs.checkpoint.dir 指定

例子：`bin/hadoop namenode -format`

8.secondarynamenode

运行 HDFS 的 secondary namenode。 用法：

`hadoop secondarynamenode [-checkpoint [force]] | [-geteditsize]`

参数选项	Description
-checkpoint [force]	如果 EditLog 的大小 >= fs.checkpoint.size，启动 Secondary namenode 的检查点过程。 如果使用了-force，将不考虑 EditLog 的大小。
-geteditsize	打印 EditLog 大小

9. tasktracker

运行 MapReduce 的 task Tracker 节点。用法：

`hadoop tasktracker`

PART Ⅱ WebHDFS

1. Create and Write to a File

（1）提交一个 HTTP PUT 请求，不传送文件数据

```
curl -i -X PUT "http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=CREATE
              [&overwrite=<true|false>][&blocksize=<LONG>][&replication=<SHORT>]
              [&permission=<OCTAL>][&buffersize=<INT>]"
```

这个请求将重定向到文件数据将被写入的 datanode 上，可是设置 overwrite,blocksize,replication,permission 和 buffersize 参数值。

```
HTTP/1.1 307 TEMPORARY_REDIRECT
Location: http://<DATANODE>:<PORT>/webhdfs/v1/<PATH>?op=CREATE...
Content-Length: 0
```

（2）提交另一个 HTTP PUT 请求，用上面得到的 Location URL，加上要写入的文件数据

```
curl -i -X PUT -T <LOCAL_FILE> "http://<DATANODE>:<PORT>/webhdfs/v1/<PATH>?
op=CREATE..."
```

客户端接收到一个 201 Created 回复，内容长度为 0,和文件的 WebHDFS URI

```
HTTP/1.1 201 Created
Location: webhdfs://<HOST>:<PORT>/<PATH>
Content-Length: 0
```

2. Append to a File

（1）提交一个 HTTP POST 请求，不传送文件数据

```
curl -i -X POST "http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=APPEND
               [&buffersize=<INT>]"
```

这个请求将重定向到文件数据将被追加的 datanode 上

```
HTTP/1.1 307 TEMPORARY_REDIRECT
Location: http://<DATANODE>:<PORT>/webhdfs/v1/<PATH>?op=APPEND...
Content-Length: 0
```

（2）提交另一个 HTTP POST 请求，用上面得到的 Location URL，加上要追加的文件数据

```
curl -i -X POST -T <LOCAL_FILE> "http://<DATANODE>:<PORT>/webhdfs/v1/<PATH>?
op=APPEND..."
```

客户端接收到内容长度为 0 的回复

```
HTTP/1.1 200 OK
Content-Length: 0
```

3. Open and Read a File

提交一个 HTTP GET 请求，自动重定向。

```
curl -i -L "http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=OPEN
```

```
                  [&offset=<LONG>][&length=<LONG>][&buffersize=<INT>]"
```

请求定位到能读取文件数据的 datanode 上

```
HTTP/1.1 307 TEMPORARY_REDIRECT
Location: http://<DATANODE>:<PORT>/webhdfs/v1/<PATH>?op=OPEN...
Content-Length: 0
```

客户端重定向到 datanode，获取到文件数据

```
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Length: 22

Hello, webhdfs user!
```

4. Make a Directory

提交一个 HTTP PUT 请求

```
curl -i -X PUT "http://<HOST>:<PORT>/<PATH>?op=MKDIRS[&permission=<OCTAL>]"
```

客户端接收到一个 boolean JSON object 回复

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

{"boolean": true}
```

5. Rename a File/Directory

提交一个 HTTP PUT 请求

```
curl -i -X PUT "<HOST>:<PORT>/webhdfs/v1/<PATH>?op=RENAME&destination=<PATH>"
```

客户端接收到一个 boolean JSON object 回复

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

{"boolean": true}
```

6. Delete a File/Directory

提交一个 HTTP DELETE 请求

```
curl -i -X DELETE "http://<host>:<port>/webhdfs/v1/<path>?op=DELETE
                            [&recursive=<true|false>]"
```

客户端接收到一个 boolean JSON object 回复

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

{"boolean": true}
```

7. Status of a File/Directory

提交一个 HTTP GET 请求

```
curl -i  "http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=GETFILESTATUS"
```

客户端接收到一个 FileStatus JSON object 回复

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

{
  "FileStatus":
  {
    "accessTime"      : 0,
    "blockSize"       : 0,
    "group"           : "supergroup",
    "length"          : 0,             //in bytes, zero for directories
    "modificationTime": 1320173277227,
    "owner"           : "webuser",
    "pathSuffix"      : "",
    "permission"      : "777",
    "replication"     : 0,
    "type"            : "DIRECTORY"    //enum {FILE, DIRECTORY}
  }
}
```

8. List a Directory

提交一个 HTTP GET 请求

```
curl -i  "http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=LISTSTATUS"
```

客户端接收到一个 FileStatuses JSON object 回复

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 427

{
  "FileStatuses":
  {
    "FileStatus":
    [
      {
        "accessTime"      : 1320171722771,
        "blockSize"       : 33554432,
        "group"           : "supergroup",
        "length"          : 24930,
        "modificationTime": 1320171722771,
        "owner"           : "webuser",
        "pathSuffix"      : "a.patch",
        "permission"      : "644",
        "replication"     : 1,
        "type"            : "FILE"
      },
      {
        "accessTime"      : 0,
        "blockSize"       : 0,
        "group"           : "supergroup",
        "length"          : 0,
```

```
        "modificationTime": 1320895981256,
        "owner"           : "szetszwo",
        "pathSuffix"      : "bar",
        "permission"      : "711",
        "replication"     : 0,
        "type"            : "DIRECTORY"
      },
      ...
    ]
  }
}
```

9. Get Content Summary of a Directory

提交一个 HTTP GET 请求

```
curl -i "http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=GETCONTENTSUMMARY"
```

客户端接收到一个 ContentSummary JSON object 回复

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

{
  "ContentSummary":
  {
    "directoryCount": 2,
    "fileCount"     : 1,
    "length"        : 24930,
    "quota"         : -1,
    "spaceConsumed" : 24930,
    "spaceQuota"    : -1
  }
}
```

10. Get Home Directory

提交一个 HTTP GET 请求

```
curl -i "http://<HOST>:<PORT>/webhdfs/v1/?op=GETHOMEDIRECTORY"
```

客户端接收到一个 Path JSON object 回复

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

{"Path": "/user/szetszwo"}
```

11. 查看创建的目录下所有文件

```
bin/hadoop fs -lsr <PATH>
```

PART Ⅲ Spark-Shell

一. SparkContext

1. 使用 textFile()方法可以将本地文件或 HDFS 文件转换成 RDD

```
scala> val textFile = sc.textFile("README.md")
```

2. 转换与操作

对于 RDD 可以有两种计算方式：转换（返回值还是一个 RDD）与操作。

（1）转换(Transformations) (如：map, filter, groupBy, join 等)，Transformations 操作是 Lazy 的，也就是说从一个 RDD 转换生成另一个 RDD 的操作不是马上执行，Spark 在遇到 Transformations 操作时只会记录需要这样的操作，并不会去执行，需要等到有 Actions 操作的时候才会真正启动计算过程进行计算。

Transformation	Meaning
map(func)	返回一个新的分布式数据集，由每个原元素经过 func 函数转换后组成
filter(func)	返回一个新的数据集，由经过 func 函数后返回值为 true 的原元素组成
flatMap(func)	类似于 map，但是每一个输入元素，会被映射为 0 到多个输出元素（因此，func 函数的返回值是一个 Seq，而不是单一元素）
sample(withReplacement, frac, seed)	根据给定的随机种子 seed，随机抽样出数量为 frac 的数据
union(otherDataset)	返回一个新的数据集，由原数据集和参数联合而成
intersection(otherDataset)	返回一个新的 RDD，由原数据集和 otherDataset 的交集构成
distinct([numTasks]))	返回一个新的数据集，包含原数据集的 distinct 元素
groupByKey([numTasks])	在一个由（K,V）对组成的数据集上调用，返回一个（K，Seq[V])对的数据集。注意：默认情况下，使用 8 个并行任务进行分组，你可以传入 numTask 可选参数，根据数据量设置不同数目的 Task
reduceByKey(func, [numTasks])	在一个（K，V)对的数据集上使用，返回一个（K，V）对的数据集，key 相同的值，都被使用指定的 reduce 函数聚合到一起。和 groupbykey 类似，任务的个数是可以通过第二个可选参数来配置的
sortByKey([ascending], [numTasks])	在一个（K，V)对的数据集上使用，K 实现排序，返回一个（K，V）对的数据集，该数据集按照 keys 递增或递减排序。
join(otherDataset, [numTasks])	在类型为（K,V)和（K,W)类型的数据集上调用，返回一个（K,(V,W))对，每个 key 中的所有元素都在一起的数据集
groupWith(otherDataset, [numTasks])	在类型为（K,V)和(K,W)类型的数据集上调用，返回一个数据集，组成元素为（K, Seq[V], Seq[W]) Tuples。这个操作在其它框架，称为 CoGroup
cartesian(otherDataset)	笛卡尔积。但在数据集 T 和 U 上调用时，返回一个(T，U）对的数据集，所有元素交互进行笛卡尔积

（2）操作(Actions) (如：count, collect, save 等)，Actions 操作会返回结果或把 RDD 数据写到

存储系统中。Actions 是触发 Spark 启动计算的动因。

Transformation	Meaning
reduce(func)	通过函数 func 聚集数据集中的所有元素。Func 函数接受 2 个参数，返回一个值。这个函数必须是关联性的，确保可以被正确的并发执行
collect()	在 Driver 的程序中，以数组的形式，返回数据集的所有元素。这通常会在使用 filter 或者其它操作后，返回一个足够小的数据子集。
count()	返回数据集的元素个数
take(n)	返回一个数组，由数据集的前 n 个元素组成
takeOrdered(n, [ordering])	返回 RDD 前 n 个元素，按他们原本的顺序或一个定制的比较器
first()	返回数据集的第一个元素（类似于 take(1)）
saveAsTextFile(path)	将数据集的元素，以 textfile 的形式，保存到本地文件系统，hdfs 或者任何其它 hadoop 支持的文件系统。Spark 将会调用每个元素的 toString 方法，并将它转换为文件中的一行文本
saveAsSequenceFile(path)	将数据集的元素，以 sequencefile 的格式，保存到指定的目录下，本地系统，hdfs 或者任何其它 hadoop 支持的文件系统。RDD 的元素必须 由 key-value 对组成，并都实现了 Hadoop 的 Writable 接口，或隐式可以转换为 Writable（Spark 包括了基本类型的转换，例如 Int，Double，String 等等）
saveAsObjectFile(path)	用 java 序列化将数据集一种简单的格式写入，可用 SparkContext.objectFile()加载该数据
foreach(func)	在数据集的每一个元素上，运行函数 func。这通常用于更新一个累加器变量，或者和外部存储系统做交互
countByKey()	返回 hashmap，由(K, Int)和其 count 组成，只有 RDD 类型(K, V)可用

3. 例子

```
val textFile = sc.textFile("README.md")
```

(1) RDD textFile 的行数

```
textFile.count()
```

(2) RDD textFile 第一行的内容

```
textFile.first()
```

(3) textFile 前 5 行内容

```
textFile.take(5)
```

(4) 按行打印出 textFile

```
textFile.collect().foreach(println)
```

(5) 返回 textFile 中包含”Spark”的行，得到一个新 RDD linesWithSpark

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

(6) 返回 textFile 中包含”Spark”的总行数

```
textFile.filter(line => line.contains("Spark")).count()
```

(7) textFile 的字节数

```
textFile.map(s => s.length).reduce((a, b) => a + b)
```

(8) 每行以空格拆分，返回最大拆分数

```
textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
```

(9) word count

```
val wordCounts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a + b)

wordCounts.collect()

输出:Array[(String, Int)] = Array((means,1), (under,2), (this,3), (Because,1), (Python,2), (agree,1), (cluster.,1), …)
```

(10) 并行化集合例子:通过 parallelize 方法定义了一个从 1~10 的数据集，然后通过 map(_*2)对数据集每个数乘以 2，接着通过 filter(_%3==0)过滤被 3 整除的数字，最后使用 toDebugString 显示 RDD 的 LineAge，并通过 collect 计算出最终的结果。

```
val num=sc.parallelize(1 to 10)
val doublenum = num.map(_*2)
val threenum = doublenum.filter(_ % 3 == 0)
threenum.toDebugString
threenum.collect
```

(11) Shuffle 操作例子：通过 parallelize 方法定义了 K-V 键值对数据集合，通过 sortByKey()进行按照 Key 值进行排序，然后通过 collect 方法触发作业运行得到结果。groupByKey()为按照 Key 进行归组，reduceByKey(_+_)为按照 Key 进行累和，这三个方法的计算和前面的例子不同，因为这些 RDD 类型为宽依赖，在计算过程中发生了 Shuffle 动作。

```
val kv1=sc.parallelize(List(("A",1),("B",2),("C",3),("A",4),("B",5)))
kv1.sortByKey().collect
kv1.groupByKey().collect
kv1.reduceByKey(_+_).collect
```

同样，distinct、union、join 和 cogroup 等操作中也涉及到 Shuffle 过程

```
val kv2=sc.parallelize(List(("A",4),("A",4),("C",3),("A",4),("B",5)))
kv2.distinct.collect
kv1.union(kv2).collect

val kv3=sc.parallelize(List(("A",10),("B",20),("D",30)))
kv1.join(kv3).collect
kv1.cogroup(kv3).collect
```

(12)文件例子读取:通过不同方式读取 HDFS 中的文件，然后进行单词计数，最终通过运行作业计算出结果。
第一步 按照文件夹读取计算每个单词出现个数

```
val text = sc.textFile("hdfs://hadoop1:9000/class3/directory/")
```

```
text.toDebugString
val words=text.flatMap(_.split(" "))
val wordscount=words.map(x=>(x,1)).reduceByKey(_+_)
wordscount.toDebugString
wordscount.collect
```

第二步 按照匹配模式读取计算单词个数

```
val rdd2 = sc.textFile("hdfs://hadoop1:9000/class3/directory/*.txt")
rdd2.flatMap(_.split(" ")).map(x=>(x,1)).reduceByKey(_+_).collect
```

第三步 读取 gz 压缩文件计算单词个数

```
val rdd3 = sc.textFile("hdfs://hadoop1:8000/class2/test.txt.gz")
rdd3.flatMap(_.split(" ")).map(x=>(x,1)).reduceByKey(_+_).collect
```

二. Spark SQLContext

1. DataFrame

可以通过如下数据源创建 DataFrame：

- 已有的 RDD
- 结构化数据文件
- JSON 数据集
- Hive 表
- 外部数据库

以下示例中，我们将从文本文件中加载用户数据并从数据集中创建一个 DataFrame 对象。然后运行 DataFrame 函数，执行特定的数据选择查询。文本文件 customers.txt 中的内容如下：

```
100, John Smith, Austin, TX, 78727
200, Joe Johnson, Dallas, TX, 75201
300, Bob Jones, Houston, TX, 77028
400, Andy Davis, San Antonio, TX, 78227
500, James Williams, Austin, TX, 78727
```

（1）通过反射得到模式
首先用已有的 Spark Context 对象创建 SQLContext 对象

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

导入语句，可以隐式地将 RDD 转化成 DataFrame

```
import sqlContext.implicits._
```

创建一个表示客户的自定义类

```
case class Customer(customer_id: Int, name: String, city: String, state: String,
zip_code: String)
```

用数据集文本文件创建一个 Customer 对象的 DataFrame

```
val dfCustomers = sc.textFile("data/customers.txt").map(_.split(",")).map(p =>
Customer(p(0).trim.toInt, p(1), p(2), p(3), p(4))).toDF()
```

将 DataFrame 注册为一个表

```
dfCustomers.registerTempTable("customers")
```

显示 DataFrame 的内容

```
dfCustomers.show()
```

打印 DF 模式

```
dfCustomers.printSchema()
```

选择客户名称列

```
dfCustomers.select("name").show()
```

选择客户名称和城市列

```
dfCustomers.select("name", "city").show()
```

根据 id 选择客户

```
dfCustomers.filter(dfCustomers("customer_id").equalTo(500)).show()
```

根据邮政编码统计客户数量

```
dfCustomers.groupBy("zip_code").count().show()
```

用 sqlContext 对象提供的 sql 方法执行 SQL 语句

```
val custNames = sqlContext.sql("SELECT name FROM customers")
```

SQL 查询的返回结果为 DataFrame 对象，支持所有通用的 RDD 操作。可以按照顺序访问结果行的各个列。

```
custNames.map(t => "Name: " + t(0)).collect().foreach(println)
```

（2）通过编程的方式指定数据集得到模式。这种方法在由于数据的结构以字符串的形式编码而无法提前定义定制类的情况下非常实用。
用已有的 Spark Context 对象 sc 创建 SQLContext 对象

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

创建 RDD 对象

```
val rddCustomers = sc.textFile("data/customers.txt")
```

用字符串编码模式

```
val schemaString = "customer_id name city state zip_code"
```

导入 Spark SQL 数据类型和 Row

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

用模式字符串生成模式对象

```
val schema = StructType(schemaString.split(" ").map(fieldName =>
StructField(fieldName, StringType, true)))
```

将 RDD（rddCustomers）记录转化成 Row

```
val rowRDD = rddCustomers.map(_.split(",")).map(p =>
Row(p(0).trim,p(1),p(2),p(3),p(4)))
```

将模式应用于 RDD 对象

```
val dfCustomers = sqlContext.createDataFrame(rowRDD, schema)
```

将 DataFrame 注册为表

```
dfCustomers.registerTempTable("customers")
```

用 sqlContext 对象提供的 sql 方法执行 SQL 语句

```
val custNames = sqlContext.sql("SELECT name FROM customers")
```

SQL 查询的返回结果为 DataFrame 对象，支持所有通用的 RDD 操作。可以按照顺序访问结果行的各个列。

```
custNames.map(t => "Name: " + t(0)).collect().foreach(println)
```

用 sqlContext 对象提供的 sql 方法执行 SQL 语句。

```
val customersByCity = sqlContext.sql("SELECT name,zip_code FROM customers ORDER
BY zip_code")
```

SQL 查询的返回结果为 DataFrame 对象，支持所有通用的 RDD 操作。可以按照顺序访问结果行的各个列。

```
customersByCity.map(t => t(0) + "," + t(1)).collect().foreach(println)
```

2. Data Sources

（1）一般 Load/Save 方法

Spark SQL 的默认数据源为 Parquet 格式。数据源为 Parquet 文件时，Spark SQL 可以方便的执行所有的操作。修改配置项 spark.sql.sources.default，可修改默认数据源格式。读取 Parquet 文件示例如下：

```
val df = sqlContext.load("people.parquet")
df.select(“name”.”age”).save(“nameAndAges.parquet”)
```

A. 手动指定选项

当数据源格式不是 parquet 格式文件时，需要手动指定数据源的格式。数据源格式需要指定全名（例 如：org.apache.spark.sql.parquet），如果数据源格式为内置格式，则只需要指定简称（json,parquet,jdbc）。通过指定的数据源格式名，可以对

DataFrames 进行类型转换操作。示例如下：

```
val df = sqlContext.read.format("json").load("people.json")
df.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```

B. 存储模式

可以采用 SaveMode 执行存储操作，SaveMode 定义了对数据的处理模式。需要注意的是，这些保存模式不使用任何锁定，不是原子操作，因此，当多个写操作同时写入到同一位置是不安全的。此外，当使用 Overwrite 方式执行时，在输出新数据之前原数据就已经被删除。SaveMode 详细介绍如下表：

Scala/Java	Meaning
SaveMode.ErrorIfExists (default)	当将 DataFrame 保存成 data source，如果数据已经存在，将会抛出异常
SaveMode.Append	当将 DataFrame 保存成 data source，如果数据/表已经存在，DataFrame 的内容将追加到已有数据中
SaveMode.Overwrite	重写模式意味着当将 DataFrame 保存成 data source，如果数据/表已经存在，已有数据会被 DataFrame 的内容重写
SaveMode.Ignore	Ignore 模式意味着当将 DataFrame 保存成 data source，如果数据存在，将不会存储 DataFrame 的内容，也不会改变已有数据

C. 持久化到表

当使用 HiveContext 时，可以通过 saveAsTable 方法将 DataFrames 存储到表中。与 registerTempTable 方法不同的是，saveAsTable 将 DataFrame 中的内容持久化到表中，并在 HiveMetastore 中存储元数据。存储一个 DataFrame，可以使用 SQLContext 的 table 方法。table 先创建一个表，方法参数为要创建的表的表名，然后将 DataFrame 持 久化到这个表中。

默认的 saveAsTable 方法将创建一个“managed table”，表示数据的位置可以通过 metastore 获得。当存储数据的表被删除时，managed table 也将自动删除。

（2）Parquet 文件

Parquet 是一种支持多种数据处理系统的柱状的数据格式，Parquet 文件中保留了原始数据的模式。Spark SQL 提供了 Parquet 文件的读写功能。

隐式地将 RDD 转化成 DataFrame

```
import sqlContext.implicits._
val people: RDD[Person] = ...
```

RDD 隐式的转化成 DataFrame，并允许其用 Parquet 存储

```
people.saveAsParquetFile("people.parquet")
```

读取 Parquet 文件，加载结果也是 DataFrame

```
val parquetFile = sqlContext.parquetFile("people.parquet")
```

Parquet 文件也能注册为 tables，然后用 SQL 语句进行查询

```
parquetFile.registerTempTable("parquetFile")
```

```
val teenagers = sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13 AND
age <= 19")
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

（3）JSON 数据集

Spark SQL 能自动解析 JSON 数据集的 Schema，读取 JSON 数据集为 DataFrame 格式。这种转换可以用以下两种方法的其中一种实现：

- jsonFile - 从 JSON 文件目录加载数据，文件中每一行是个 JSON 对象
- jsonRDD 从一个已有的 RDD 加载数据，RDD 的每个元素是个包含一个 JSON 对象的字符串

sc 是已存在的 SparkContext

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

指定一个 JSON 数据集的路径，路径可以是一个文本，也可以是一个存有文本的目录

```
val path = "examples/src/main/resources/people.json"
```

根据路径指定的文件创建 DataFrame

```
val people = sqlContext.jsonFile(path)
```

模式可以用 printSchema()方法查看

```
people.printSchema()
```

将 DataFrame 注册为一个表

```
people.registerTempTable("people")
```

运行 SQL 语句

```
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age
<= 19")
```

或者， 从一个已有的 RDD 加载数据创建 DataFrame，RDD 的每个元素是个包含一个 JSON 对象的字符串

```
val anotherPeopleRDD = sc.parallelize(
  """{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: Nil)
val anotherPeople = sqlContext.jsonRDD(anotherPeopleRDD)
```

（4）Hive Tables

Spark SQL 支持对 Hive 的读写操作。需要注意的是，Hive 所依赖的包，没有包含在 Spark assembly 包中。增加 Hive 时，需要在 Spark 的 build 中添加 -Phive 和 -Phivethriftserver 配置。这两个配置将 build 一个新的 assembly 包，这个 assembly 包含了 Hive 的依赖包。注意，必须将这个新的 assembly 包加到所有的 worker 节点上。因为 worker 节点在访问 Hive 中数据时，会调用 Hive 的 serialization and deserialization libraries（SerDes）。

Hive 的配置文件为 conf/目录下的 hive-site.xml 文件。

操作 Hive 时，必须创建一个 HiveContext 对象，HiveContext 继承了 SQLContext，并增加了对 MetaStore 和 HiveQL 的支持。示例如下：

sc 是已存在的 SparkContext.

```
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt'
INTO TABLE src")
```

查询用 HiveQL 表示

```
sqlContext.sql("FROM src SELECT key, value").collect().foreach(println)
```

（5）JDBC To Other Databases

Spark SQL 支持使用 JDBC 访问其他数据库。当时用 JDBC 访问其它数据库时，最好使用 JdbcRDD。使用 JdbcRDD 时，Spark SQL 操作返回的 DataFrame 会很方便，也会很方便的添加其他数据源数据。JDBC 数据源因为不需要用户提供 ClassTag，所以很适合使用 Java 或 Python 进行操作。

使用 JDBC 访问数据源，需要在 spark classpath 添加 JDBC driver 配置。例如，从 Spark Shell 连接 postgres 的配置为：

```
SPARK_CLASSPATH=postgresql-9.3-1102-jdbc41.jar bin/spark-shell
```

远程数据库的表，可用 DataFrame 或 Spark SQL 临时表的方式调用数据源 API。支持的参数有

属性名	Meaning
url	连接所需的 JDBC URL
dbtable	可以读取的 JDBC 表。SQL 查询 FROM 子句中任何有效的表都能读取。比如，你可以读括号中的子查询。
driver	JDBC 驱动程序的类名必须连接到这个 URL。运行 JDBC 命令允许 driver 注册到 JDBC 子系统之前加载这个类到 master 和 workers。
partitionColumn, lowerBound, upperBound, numPartitions	这些选项都必须指定如果指定其中任何一个。他们描述了如何分表当多个 workers 并行读取。partitionColumn 必须是一个 numeric 列。

```
val jdbcDF = sqlContext.load("jdbc", Map(
  "url" -> "jdbc:postgresql:dbserver",
  "dbtable" -> "schema.tablename"))
```