Spark 学习笔记

harli

2014/12/9

shanghai-pd

[Scala语言 4](#_Toc405898066)

[Spark平台的API 4](#_Toc405898067)

[Spark内核 4](#_Toc405898068)

[Spark上的核心框架 4](#_Toc405898069)

[商业级别的Spark项目 4](#_Toc405898070)

[Spark解决方案 4](#_Toc405898071)

[Spark内核 4](#_Toc405898072)

[Spark核心框架 5](#_Toc405898073)

[Spark Streaming 5](#_Toc405898074)

[Spark SQL 5](#_Toc405898075)

[MLlib(Machine Learning library) 5](#_Toc405898076)

[GraphX(Graph Processing) 5](#_Toc405898077)

[Bagel (Pregel on Spark) 5](#_Toc405898078)

[SparkR 5](#_Toc405898079)

[Spark开发与应用 5](#_Toc405898080)

[部署 5](#_Toc405898081)

[提交应用程序 5](#_Toc405898082)

[Amazon EC2 5](#_Toc405898083)

[Standalone模式 6](#_Toc405898084)

[Mesos 6](#_Toc405898085)

[YARN 6](#_Toc405898086)

[监控 6](#_Toc405898087)

[配置 6](#_Toc405898088)

[调优 6](#_Toc405898089)

[作业调度 6](#_Toc405898090)

[安全 6](#_Toc405898091)

[硬件配置 6](#_Toc405898092)

[技巧、FQA 6](#_Toc405898093)

# Spark学习阶段篇

<http://book.51cto.com/art/201409/451443.htm>

Spark 的学习阶段篇包含六大部分，从Scala语言到spark内核、框架，以及spark在商业上的企业级实践等。

# Scala语言

1，Spark框架是采用Scala语言编写的，精致而优雅。要想成为Spark高手，你就必须阅读Spark的源代码，就必须掌握Scala,；

2，虽然说现在的Spark可以采用多语言Java、Python等进行应用程序开发，但是最快速的和支持最好的开发API依然并将永远是Scala方式的API，所以你必须掌握Scala来编写复杂的和高性能的Spark分布式程序；

3，尤其要熟练掌握Scala的trait、apply、函数式编程、泛型、逆变与协变等；

推荐课程：”精通Spark的开发语言：Scala最佳实践”

# Spark平台的API

1，掌握Spark中面向RDD的开发模式，掌握各种transformation和action函数的使用；

2，掌握Spark中的宽依赖和窄依赖以及lineage机制；

3，掌握RDD的计算流程，例如Stage的划分、Spark应用程序提交给集群的基本过程和Worker节点基础的工作原理等

推荐课程：“18小时内掌握Spark：把云计算大数据速度提高100倍以上!”

# Spark内核

Spark的内核部分的学习笔记，在***学习路线篇***中对应章节记录。

此阶段主要是通过Spark框架的源码研读来深入Spark内核部分：

1，通过源码掌握Spark的任务提交过程；

2，通过源码掌握Spark集群的任务调度；

3，尤其要精通DAGScheduler、TaskScheduler和Worker节点内部的工作的每一步的细节；

推荐课程：“Spark 1.0.0企业级开发动手：实战世界上第一个Spark 1.0.0课程，涵盖Spark 1.0.0所有的企业级开发技术”

# Spark上的核心框架

Spark上的核心框架部分的学习笔记，在***学习路线篇***中对应章节记录。

Spark作为云计算大数据时代的集大成者，在实时流处理、图技术、机器学习、NoSQL查询等方面具有显著的优势，我们使用Spark的时候大部分时间都是在使用其上的框架例如Shark、Spark Streaming等：

1，Spark Streaming是非常出色的实时流处理框架，要掌握其DStream、transformation和checkpoint等；

2，Spark的离线统计分析功能，Spark 1.0.0版本在Shark的基础上推出了Spark SQL，离线统计分析的功能的效率有显著的提升，需要重点掌握；

3，对于Spark的机器学习和GraphX等要掌握其原理和用法；

推荐课程：“Spark企业级开发最佳实践”

# 商业级别的Spark项目

通过一个完整的具有代表性的Spark项目来贯穿Spark的方方面面，包括项目的架构设计、用到的技术的剖析、开发实现、运维等，完整掌握其中的每一个阶段和细节，这样就可以让您以后可以从容面对绝大多数Spark项目。

推荐课程：“Spark架构案例鉴赏：Conviva、Yahoo！、优酷土豆、网易、腾讯、淘宝等公司的实际Spark案例”

# Spark解决方案

1，彻底掌握Spark框架源码的每一个细节；

2，根据不同的业务场景的需要提供Spark在不同场景的下的解决方案；

3，根据实际需要，在Spark框架基础上进行二次开发，打造自己的Spark框架；

推荐课程：“精通Spark：Spark内核剖析、源码解读、性能优化和商业案例实战”

前面所述的成为Spark高手的六个阶段中的第一和第二个阶段可以通过自学逐步完成，随后的三个阶段最好是由高手或者专家的指引下一步步完成，最后一个阶段，基本上就是到”无招胜有招”的时期，很多东西要用心领悟才能完成。

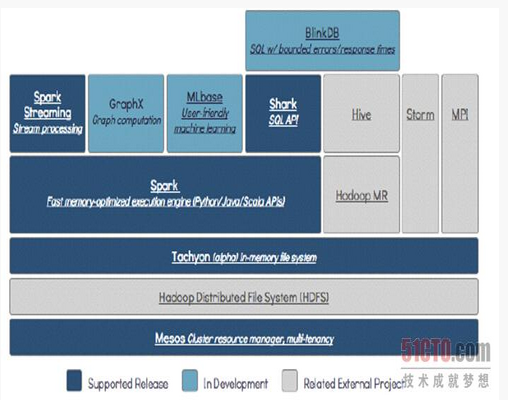
# Spark的学习路线篇

Spark学习路线篇，基于spark的官网文档组织结构，分别对各个部分进行学习，并记录各个部分学习笔记。

# 生态圈

## BDAS

BDAS(Berkeley Data Analytics Stack) ，伯克利大学提出的关于数据分析的软件栈：



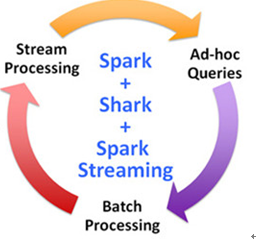
目前的大数据的处理类型：

1. 复杂的批量数据处理（batch data processing），通常的时间跨度在数十分钟到数小时之间。
2. 基于历史数据的交互式查询（interactive query），通常的时间跨度在数十秒到数分钟之间。
3. 基于实时数据流的数据处理（streaming data processing），通常的时间跨度在数百毫秒到数秒之间。

目前已有很多相对成熟的开源软件来处理以上三种情景，我们可以利用MapReduce来进行批量数据处理，可以用Impala来进行交互式查询，对于流式数据处理，我们可以采用Storm。对于大多数互联网公司来说，一般都会同时遇到以上三种情景，那么在使用的过程中这些公司可能会遇到如下的不便。

1. 三种情景的输入输出数据无法无缝共享，需要进行格式相互转换。
2. 每一个开源软件都需要一个开发和维护团队，提高了成本。
3. 在同一个集群中对各个系统协调资源分配比较困难。

BDAS就是以Spark为基础的一套软件栈，利用基于内存的通用计算模型将以上三种情景一网打尽，同时支持Batch、Interactive、Streaming的处理，且兼容支持HDFS和S3等分布式文件系统，可以部署在YARN和Mesos等流行的集群资源管理器之上。



图：Batch + Interactive + Streaming

## Spark与Hadoop的比较

Spark是UC Berkeley AMP lab所开源的类Hadoop MapReduce的通用的并行计算框架，Spark基于map reduce算法实现的分布式计算，拥有Hadoop MapReduce所具有的优点；但不同于MapReduce的是Job中间输出和结果可以保存在内存中，从而不再需要读写HDFS，因此Spark能更好地适用于数据挖掘与机器学习等需要迭代的map reduce的算法。

### Spark与Hadoop的对比

1. Spark的中间数据放到内存中，对于迭代运算效率更高

* Spark更适合于迭代运算比较多的ML和DM运算。因为在Spark里面，有RDD的抽象概念。

1. Spark比Hadoop更通用

* Spark提供的数据集操作类型有很多种，不像Hadoop只提供了Map和Reduce两种操作。比如map, filter, flatMap, sample, groupByKey, reduceByKey, union, join, cogroup, mapValues,sort,partionBy等多种操作类型，Spark把这些操作称为Transformations。同时还提供Count, collect, reduce, lookup, save等多种actions操作。
* 这些多种多样的数据集操作类型，给给开发上层应用的用户提供了方便。各个处理节点之间的通信模型不再像Hadoop那样就是唯一的Data Shuffle一种模式。用户可以命名，物化，控制中间结果的存储、分区等。可以说编程模型比Hadoop更灵活。
* 不过由于RDD的特性，Spark不适用那种异步细粒度更新状态的应用，例如web服务的存储或者是增量的web爬虫和索引。就是对于那种增量修改的应用模型不适合。

1. 容错性

* 在RDD计算，通过checkpint进行容错，而checkpoint有两种方式，一个是checkpoint data，一个是logging the updates。用户可以控制采用哪种方式来实现容错，默认是logging the updates方式，通过记录跟踪所有生成RDD的转换（transformations）也就是记录每个RDD的lineage（血统）来重新计算生成丢失的分区数据。

1. 可用性

* Spark通过提供丰富的Scala, Java，Python API及交互式Shell来提高可用性。

各公司对spark与Hadoop的选择：

1，原先支持Hadoop的四大商业机构纷纷宣布支持Spark；

2，Mahout前一阶段表示从现在起他们将不再接受任何形式的以MapReduce形式实现的算法，另外一方面，Mahout宣布新的算法基于Spark；

3，Cloudera的机器学习框架Oryx的执行引擎也将由Hadoop的MapReduce替换成Spark；

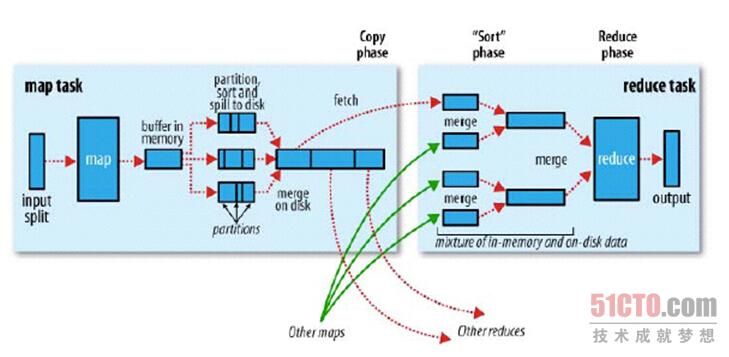
4，Google已经开始将负载从MapReduce转移到Pregel和Dremel上；

5，FaceBook则将负载转移到Presto上；

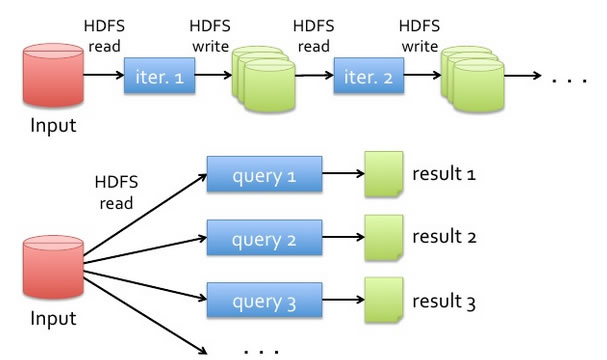
### Hadoop的MapReduce计算模型

Hadoop本身的计算模型决定了Hadoop上的所有工作都要转化成Map、Shuffle和Reduce等核心阶段，由于每次计算都要从磁盘读或者写数据，同时真个计算模型需要网络传输，这就导致了越来越不能忍受的延迟性，同时在前一个任务运行完之前，任何一个任务都不可以运行，这直接导致了其无力支持交互式应用；

首先我们看一下Hadoop经典的处理过程：

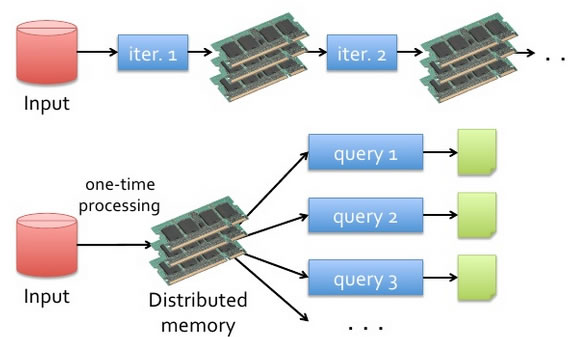
[](http://s8.51cto.com/wyfs02/M01/49/AF/wKiom1QZA-Kx2E5iAACCOw5Z3jo227.jpg)

MapReduce在每次执行的时候都要从磁盘读数据，计算完毕后都要把数据存放到磁盘上：

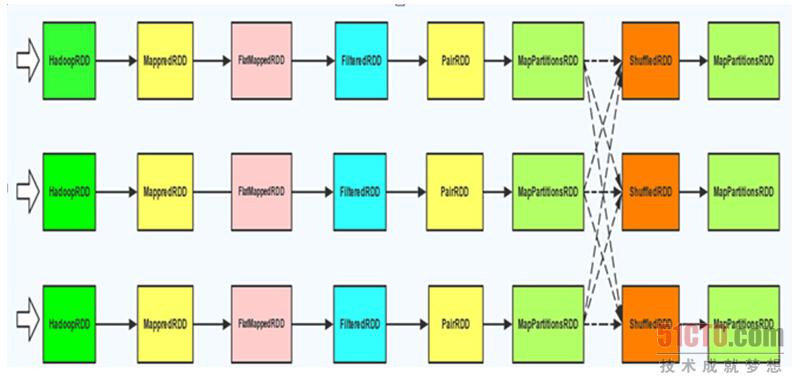


### Spark的计算模型

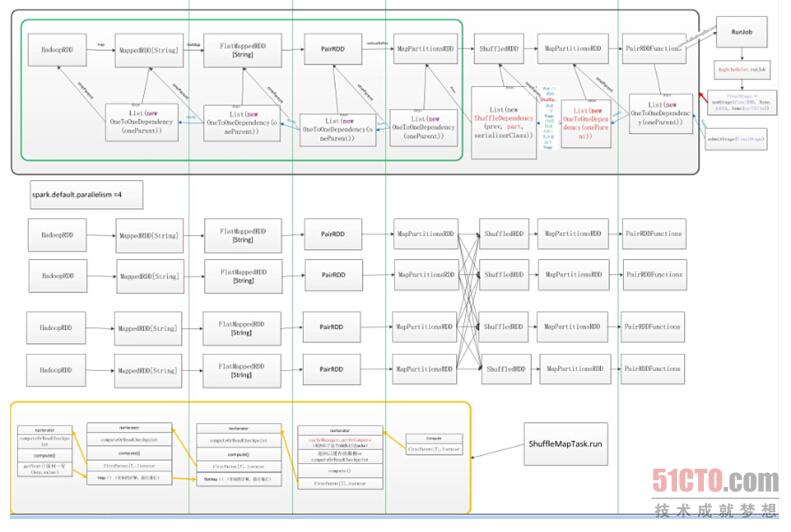
而在Spark中，使用内存替代了使用HDFS存储中间结果：



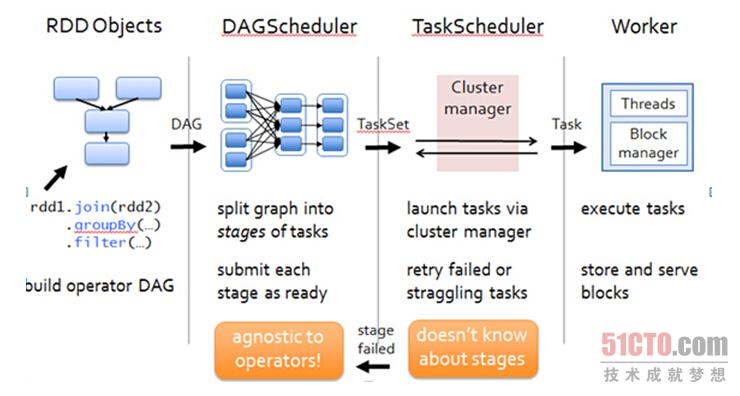
另外一方面，DAG也是Spark快的极为重要的原因，下面是一张DAG图的示例：

[](http://s5.51cto.com/wyfs02/M00/49/B1/wKioL1QZBAvzkg6dAACVeKealQc484.jpg)

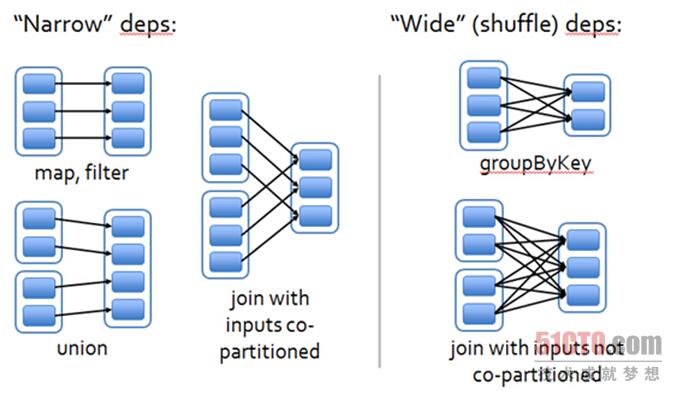
大家也可以看一下网络上一张描述DAG更多细节的图片：

[](http://s3.51cto.com/wyfs02/M00/49/AF/wKiom1QZBAaDh4bFAADd6A7hits354.jpg)

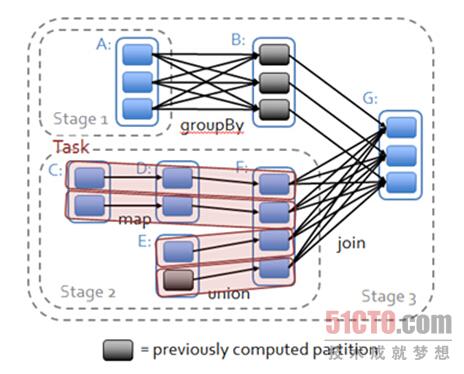
基于DAG，Spark具备了非常精致的作业调度系统：

[](http://s4.51cto.com/wyfs02/M01/49/B1/wKioL1QZBCuyVPxaAACdnxSZEyM489.jpg)

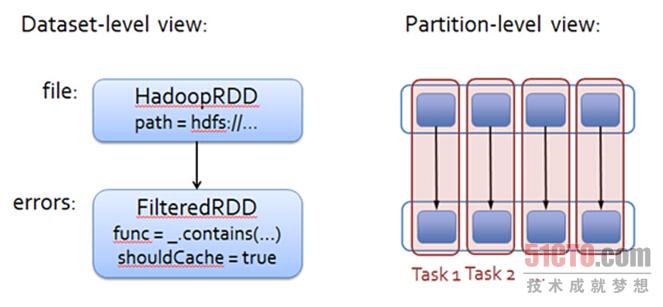
DAG中的依赖有宽依赖和窄依赖之分：

[](http://s6.51cto.com/wyfs02/M02/49/AF/wKiom1QZBCGjW-JaAACUgui8TKY273.jpg)

在DAG图中可以根据依赖对pipeline等优化操作：

[](http://s6.51cto.com/wyfs02/M00/49/B1/wKioL1QZBESB00coAAB8EujG9YY249.jpg)

基于RDD和DAG，并行计算整个Job：

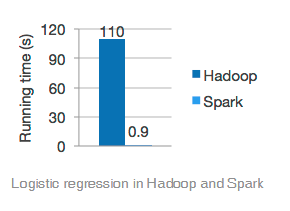
[](http://s5.51cto.com/wyfs02/M00/49/AF/wKiom1QZBDqBolUVAABsotUtKaM866.jpg)

Spark之所以快，还有一个原因就是其容错机制，这个我们会在本讲的后面和大家分享。

### 对比实例

<http://spark.apache.org/>

**Apache Spark™** is a fast and general engine for large-scale data processing.



Speed

Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

Spark has an advanced DAG execution engine that supports cyclic data flow and in-memory computing.

## Spark与Hadoop的结合

Spark与Hadoop的结合主要包含两个方面：

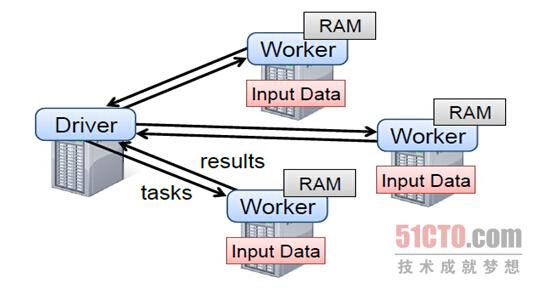
1. HDFS：Spark可以直接对HDFS进行数据的读写；
2. YARN：Spark支持Spark on YARN部署模式，由YARN负责集群的资源管理与调度；Spark的应用可以与Hadoop MapReduce运行在同一个集群中，共享存储资源与计算。

## Shark

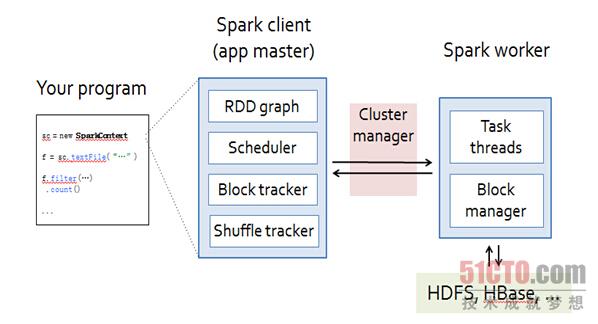
Shark ( Hive on Spark): Shark基本上就是在Spark的框架基础上提供和Hive一样的H iveQL命令接口，为了最大程度的保持和Hive的兼容性，Shark使用了Hive的API来实现query Parsing和 Logic Plan generation，最后的PhysicalPlan execution阶段用Spark代替Hadoop MapReduce。通过配置Shark参数，Shark可以自动在内存中缓存特定的RDD，实现数据重用，进而加快特定数据集的检索。同时，Shark通过UDF用户自定义函数实现特定的数据分析学习算法，使得SQL数据查询和运算分析能结合在一起，最大化RDD的重复使用。

# Spark架构设计

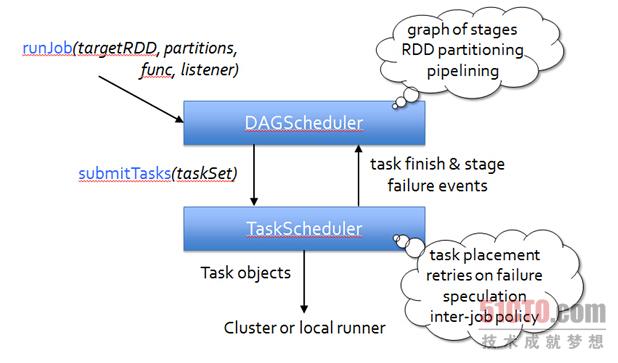
正如所有的分布式结构一样，Spark分布式集群也是主从结构的：

[](http://s8.51cto.com/wyfs02/M00/49/A5/wKiom1QX1vjDc1LgAABZYY1wZtU595.jpg)

Spark运行时组件如下所示：

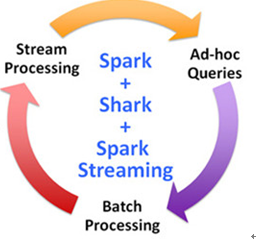
[](http://s3.51cto.com/wyfs02/M01/49/A7/wKioL1QX1x2TlbcZAABm0_1xrPo716.jpg)

Spark运行时候的事件流如下所示：

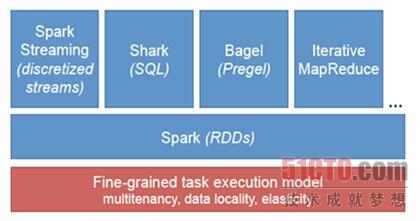
[](http://s1.51cto.com/wyfs02/M01/49/A5/wKiom1QX1xayHKacAAB6bg-lKGA055.jpg)

# Spark编程模型

Spark是大数据时代通用而高效的计算平台，基于RDD成功实现了“One stack to rule them all”理念。



在Spark中一切都是以RDD为基础和核心。以RDD为基石的Spark编程模型。



## RDD

### RDD简介

Resilient Distributed Datasets(RDD) 弹性分布数据集，是一个容错的、并行的数据结构，可以让用户显式地将数据存储到磁盘和内存中，并能控制数据的分区。同时，RDD还提供了一组丰富的操作来处理数据。在这些操作中，诸如map、flatMap、filter等转换操作实现了monad模式，很好地契合了Scala的集合操作。除此之外，RDD还提供了诸如join、groupBy、reduceByKey等更为方便的操作（注意，reduceByKey是action，而非transformation），以支持常见的数据运算。

RDD具有以下特点：

1. 它是在集群节点上的不可变的、已分区的集合对象。
2. 通过并行转换的方式来创建如（map, filter, join, etc）。
3. 失败自动重建。
4. 可以控制存储级别（内存、磁盘等）来进行重用。
5. 必须是可序列化的。
6. 是静态类型的。

RDD的好处：

1. RDD只能从持久存储或通过Transformations操作产生，相比于分布式共享内存（DSM）可以更高效实现容错，对于丢失部分数据分区只需根据它的lineage就可重新计算出来，而不需要做特定的Checkpoint。
2. RDD的不变性，可以实现类Hadoop MapReduce的推测式执行。
3. RDD的数据分区特性，可以通过数据的本地性来提高性能，这与Hadoop MapReduce是一样的。
4. RDD都是可序列化的，在内存不足时可自动降级为磁盘存储，把RDD存储于磁盘上，这时性能会有大的下降但不会差于现在的MapReduce。

### RDD的实现

RDD本身是一个抽象类，其内部实现包括以下5个部分，其中前三个必须实现：

1. 分区列表（数据块列表）
2. 计算每个分片的函数（根据父RDD计算出此RDD）
3. 对父RDD的依赖列表
4. 对key-value RDD的Partitioner【可选】
5. 每个数据分片的预定义地址列表(如HDFS上的数据块的地址)【可选】

另外有两个特殊的RDD:

[](http://s9.51cto.com/wyfs02/M02/49/AF/wKiom1QZBP6Ba_qxAAAW12cp5t4430.jpg)

他们都是controlling operations：

[](http://s5.51cto.com/wyfs02/M02/49/B1/wKioL1QZBUrxSvD2AABRqQUGmfo748.jpg)

### RDD的存储

RDD的存储级别：RDD根据useDisk、useMemory、deserialized、replication四个参数的组合提供了11种存储级别：

|  |
| --- |
| val NONE = new StorageLevel(false, false, false)  val DISK\_ONLY = new StorageLevel(true, false, false)  val DISK\_ONLY\_2 = new StorageLevel(true, false, false, 2)  val MEMORY\_ONLY = new StorageLevel(false, true, true)  val MEMORY\_ONLY\_2 = new StorageLevel(false, true, true, 2)  val MEMORY\_ONLY\_SER = new StorageLevel(false, true, false)  val MEMORY\_ONLY\_SER\_2 = new StorageLevel(false, true, false, 2)  val MEMORY\_AND\_DISK = new StorageLevel(true, true, true)  val MEMORY\_AND\_DISK\_2 = new StorageLevel(true, true, true, 2)  val MEMORY\_AND\_DISK\_SER = new StorageLevel(true, true, false)  val MEMORY\_AND\_DISK\_SER\_2 = new StorageLevel(true, true, false, 2) |

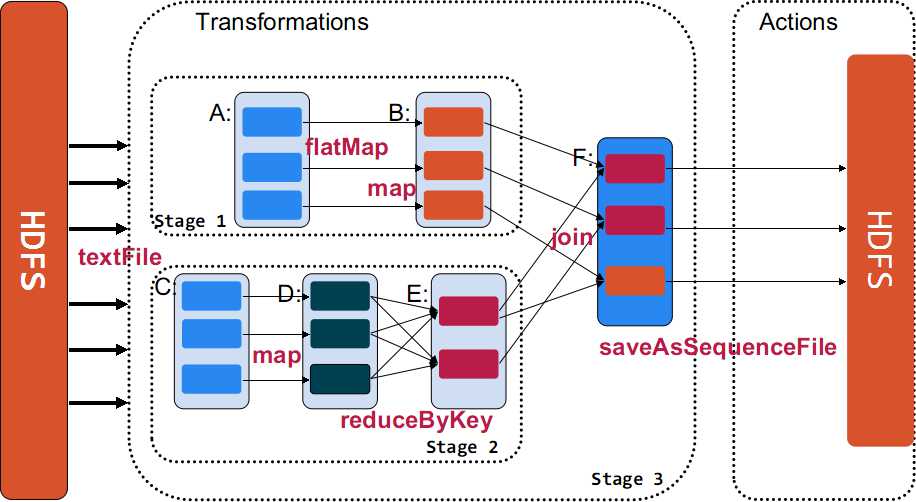
用户可以选择不同的存储级别存储RDD以便重用。

### RDD的分区

### RDD的操作

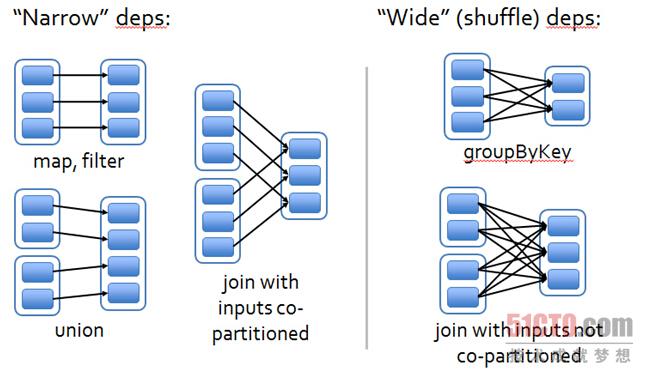
RDD将操作分为两类：transformation与action。其中transformations是lazy execution的，需要具体的action去触发，每个action操作都是一个单独的job；

在RDD的内部实现机制中，底层接口则是基于迭代器的，从而使得数据访问变得更高效，也避免了大量中间结果对内存的消耗。



### RDD的依赖

在Spark中RDD是具备依赖关系的，而依赖分为两种：

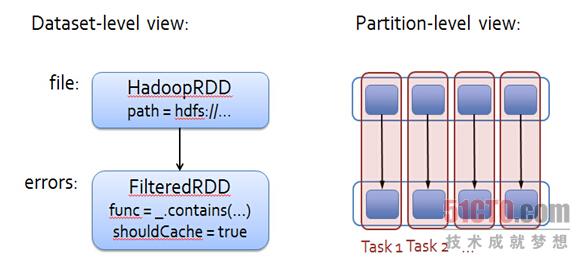
[](http://s8.51cto.com/wyfs02/M00/49/A5/wKiom1QX1seBZgvlAACcdmWVUf4081.jpg)

“Narrow”依赖的一个好处就是可以进行内部的pipeline操作：

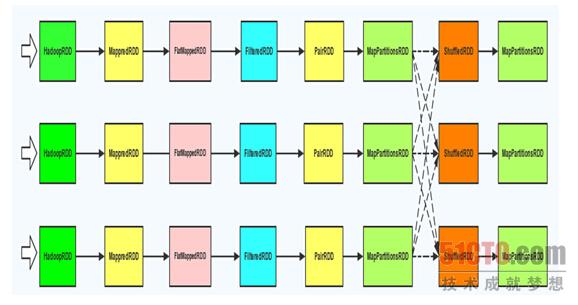
[](http://s3.51cto.com/wyfs02/M02/49/A7/wKioL1QX1uTjo6DMAACfyI_BJzU555.jpg)

### RDD执行框架

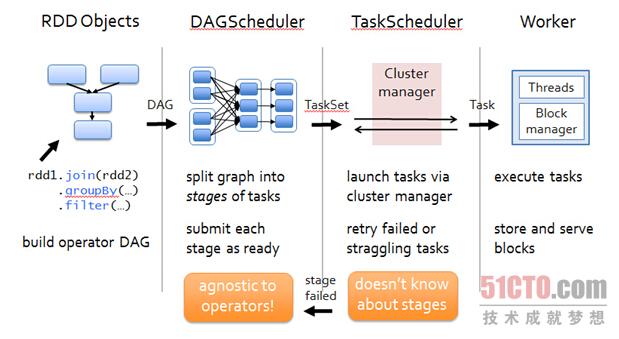
运行的时候是以RDD为统一抽象并行化运行：

[](http://s7.51cto.com/wyfs02/M02/49/A5/wKiom1QX1uHQiSh_AABdq_Oy4tE861.jpg)

更进一步的详细RDD并行化计算过程如下所示：

[](http://s1.51cto.com/wyfs02/M00/49/A7/wKioL1QX1v3RVxuwAABsO4PxdvM244.jpg)

RDD在产生作业调用的时候，经典的过程如下所示：

[](http://s8.51cto.com/wyfs02/M01/49/A6/wKiom1QX2l6SgaISAACN5rHw90c511.jpg)

### RDD的持久化

RDD在持久化的需要考虑内存策略

### RDD对容错的支持

支持容错通常采用两种方式：数据复制或日志记录。对于以数据为中心的系统而言，这两种方式都非常昂贵，因为它需要跨集群网络拷贝大量数据，毕竟带宽的数据远远低于内存。

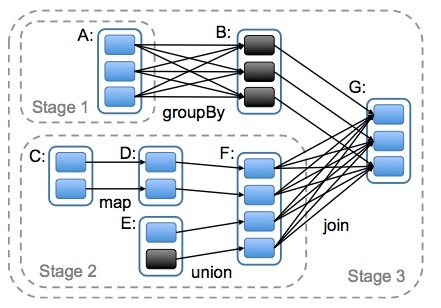
RDD天生是支持容错的。首先，它自身是一个不变的(immutable)数据集，其次，它能够记住构建它的操作图（Graph of Operation），因此当执行任务的Worker失败时，完全可以通过操作图获得之前执行的操作，进行重新计算。由于无需采用replication方式支持容错，很好地降低了跨网络的数据传输成本。

不过，在某些场景下，Spark也需要利用记录日志的方式来支持容错。例如，在Spark Streaming中，针对数据进行update操作，或者调用Streaming提供的window操作时，就需要恢复执行过程的中间状态。此时，需要通过Spark提供的checkpoint机制，以支持操作能够从checkpoint得到恢复。

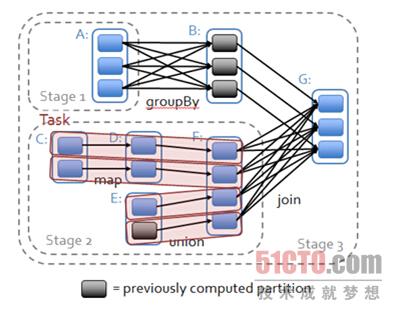
针对RDD的wide dependency，最有效的容错方式同样还是采用checkpoint机制。不过，似乎Spark的最新版本仍然没有引入auto checkpointing机制。

## Spark的高容错机制lineage

基于DAG图，lineage是轻量级而高效的，操作之间相互具备lineage的关系，每个操作只关心其父操作，各个分片的数据之间互不影响，出现错误的时候只要恢复单个Split的特定部分即可：



本图来自Matei Zaharia撰写的论文An Architecture for Fast and General Data Processing on Large Clusters。图中，一个box代表一个RDD，一个带阴影的矩形框代表一个partition。



1. 利用内存加快数据加载,在众多的其它的In-Memory类数据库或Cache类系统中也有实现，Spark的主要区别在于它处理分布式运算环境下的数据容错性（节点实效/数据丢失）问题时采用的方案。为了保证RDD中数据的鲁棒性，RDD数据集通过所谓的血统关系(Lineage)记住了它是如何从其它RDD中演变过来的。相比其它系统的细颗粒度的内存数据更新级别的备份或者LOG机制，RDD的Lineage记录的是粗颗粒度的特定数据转换（Transformation）操作（filter, map, join etc.)行为。当这个RDD的部分分区数据丢失时，它可以通过Lineage获取足够的信息来重新运算和恢复丢失的数据分区。这种粗颗粒的数据模型，限制了Spark的运用场合，但同时相比细颗粒度的数据模型，也带来了性能的提升。
2. RDD在Lineage依赖方面分为两种Narrow Dependencies与Wide Dependencies用来解决数据容错的高效性。Narrow Dependencies是指父RDD的每一个分区最多被一个子RDD的分区所用，表现为一个父RDD的分区对应于一个子RDD的分区或多个父RDD的分区对应于一个子RDD的分区，也就是说一个父RDD的一个分区不可能对应一个子RDD的多个分区。Wide Dependencies是指子RDD的分区依赖于父RDD的多个分区或所有分区，也就是说存在一个父RDD的一个分区对应一个子RDD的多个分区。对与Wide Dependencies，这种计算的输入和输出在不同的节点上，lineage方法对与输入节点完好，而输出节点宕机时，通过重新计算，这种情况下，这种方法容错是有效的，否则无效，因为无法重试，需要向上其祖先追溯看是否可以重试（这就是lineage，血统的意思），Narrow Dependencies对于数据的重算开销要远小于Wide Dependencies的数据重算开销。

# Spark内核

Spark内核揭秘共分四个部分：

第一部分：Spark内核初探

第二部分：Spark内核核心源码解析

第三部分：Job全生命周期源码解读

第四部分：Spark性能优化 —— 参加性能篇或FAQ

## Spark内核初探

本讲是Spark内核揭秘的第一部分：Spark内核初探，具体内容如下所示：

1，Spark内核核心术语解析；

2，Spark集群概览；

3，Spark核心组件；

4，Spark任务调度系统初见；

### Spark内核核心术语解析

#### Application

Application是创建了SparkContext实例对象的Spark用户，包含了Driver程序。

Spark-shell是一个应用程序，因为spark-shell在启动的时候创建了SparkContext对象，其名称为sc。

#### Job

和Spark的action相对应，每一个action例如count、savaAsTextFile等都会对应一个Job实例，该Job实例包含多任务的并行计算。

#### Driver Program

运行main函数并且新建SparkContext实例的程序。

#### Cluster Manager

集群资源管理的外部服务，在Spark上现在主要有Standalone、Yarn、Mesos等三种集群资源管理器，Spark自带的Standalone模式能够满足绝大部分纯粹的Spark计算环境中对集群资源管理的需求，基本上只有在集群中运行多套计算框架的时候才建议考虑Yarn和Mesos。

#### Worker Node

集群中可以运行应用程序代码的工作节点，相当于Hadoop的slave节点。

#### Executor

在一个Worker Node上为应用启动的工作进程，在进程中负责任务的运行，并且负责将数据存放在内存或磁盘上，必须注意的是，每个应用在一个Worker Node上只会有一个Executor，在Executor内部通过多线程的方式并发处理应用的任务。

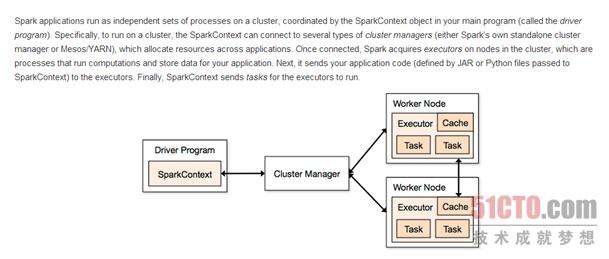
#### Task

被Driver送到executor上的工作单元，通常情况下一个task会处理一个split的数据，每个split一般就是一个Block块的大小：

#### Stage

一个Job会被拆分成很多任务，每一组任务被成为Stage，这个MapReduce的map和reduce任务很像，划分Stage的依据在于：Stage开始一般是由于读取外部数据或者Shuffle数据、一个Stage的结束一般是由于发生Shuffle（例如reduceByKey操作）或者整个Job结束时例如要把数据放到hdfs等存储系统上。

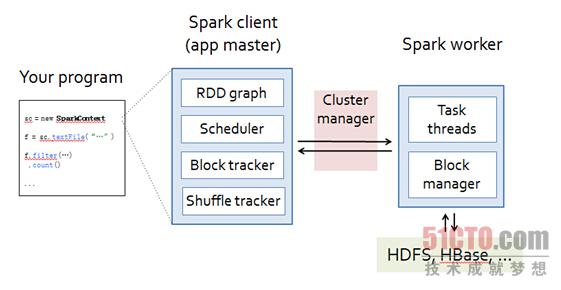
## Spark集群概览



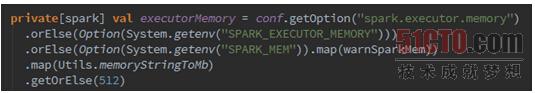
需要注意的是Spark Driver所在的机器需要和Spark集群最好位于同一个网络环境中，因为Driver中的SparkContext实例要要发送任务给不同Worker Node的Executor并接受Executor的一些执行结果信息，一般而言，在企业实际的生产环境中Driver所在机器的配置往往都是比较不错的，尤其是其CPU的处理能力往往都很强悍。

## Spark核心组件

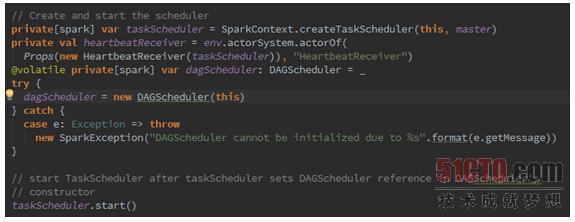
Spark核心组件如下所示：

[](http://s9.51cto.com/wyfs02/M02/4B/4D/wKioL1QpPcGytxZoAABc0lnRzzA334.jpg)

在初始话SparkContext的时候会初始化一系列的内容，例如会查看内存的设置情况：

[](http://s4.51cto.com/wyfs02/M02/4B/4D/wKioL1QpPcqiWS2KAAAxbfwyBas120.jpg)

SparkContext在初始化的时候也会创建和启动scheduler：

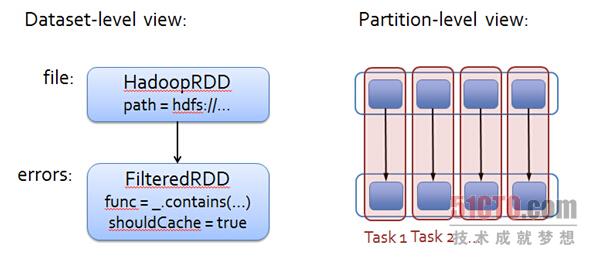
[](http://s5.51cto.com/wyfs02/M02/4B/4B/wKiom1QpPanhXG1AAABWRCj1G3U061.jpg)

集群核心组件中的Block tracker是用于Block和partition对应关系的管理器。

集群核心组件中的Shuffle tracker是用于记录Shuffle操作过程细节的。

从集群中也可以清晰的看出，Executor在执行任务的时候是采用多线程的方式执行的并能够在HDFS或者HBase等系统上存取数据。

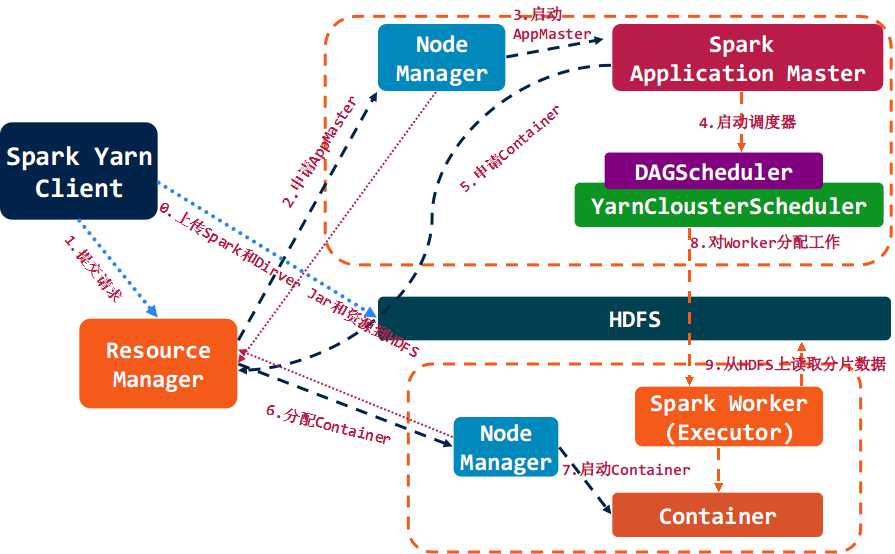
在实际的Driver Program运行的时候每个Partition都会由一个Task负责运行：

[](http://s4.51cto.com/wyfs02/M00/4B/4D/wKioL1QpPevzSmJoAABf694Jo1U497.jpg)

也就是说有多少partition就会有多少task在运行，而这些task是并发的运行在executor中的。

## 资源管理与作业调度

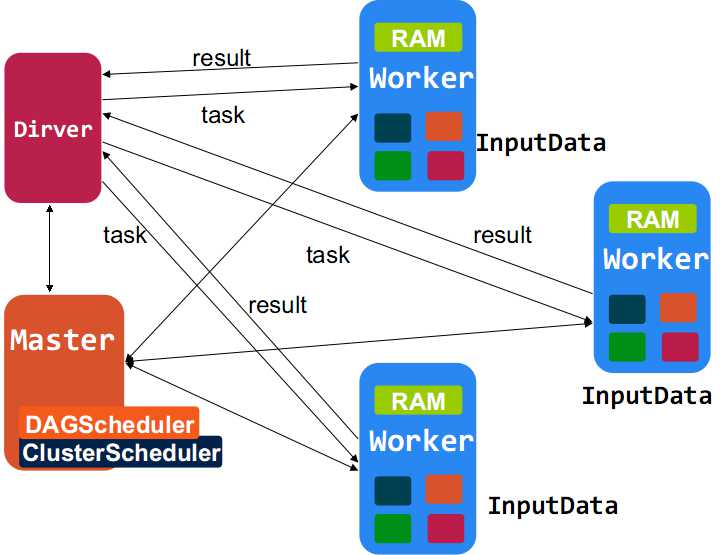
Spark对于资源管理与作业调度可以使用Standalone(独立模式)，Apache Mesos及Hadoop YARN来实现。 Spark on Yarn在Spark0.6时引用，但真正可用是在现在的branch-0.8版本。Spark on Yarn遵循YARN的官方规范实现，得益于Spark天生支持多种Scheduler和Executor的良好设计，对YARN的支持也就非常容易，Spark on Yarn的大致框架图。



让Spark运行于YARN上与Hadoop共用集群资源可以提高资源利用率。

## Spark作业的执行

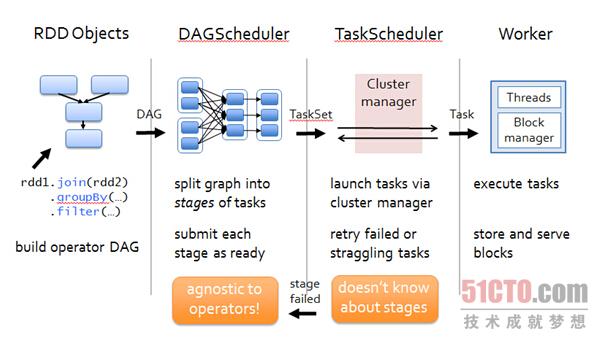
用户编写的Spark程序被称为Driver程序，Dirver程序会连接master并定义了对各RDD的转换与操作，而对RDD的转换与操作通过Scala闭包(字面量函数)来表示，Scala使用Java对象来表示闭包且都是可序列化的，以此把对RDD的闭包操作发送到各Workers节点。 Workers存储着数据分块和享有集群内存，是运行在工作节点上的守护进程，当它收到对RDD的操作时，根据数据分片信息进行本地化数据操作，生成新的数据分片、返回结果或把RDD写入存储系统。



## Spark任务调度系统初见

<http://book.51cto.com/art/201409/453029.htm>

Spark的任务调度系统如下所示：

[](http://s7.51cto.com/wyfs02/M00/4B/4E/wKioL1QpQFnQFyX4AACDAUTzfkE773.jpg)

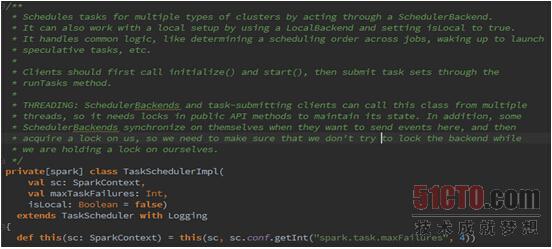
从上图中可以看出由RDD Objects产生DAG，然后进入了DAGScheduler阶段，DAGScheduler是面向Stage的高层次的调度器，DAGScheduler把DAG拆分成很多的Tasks，每组的tasks都是一个Stage，每当遇到Shuffle就会产生新的Stage，可以看出上图一共有三个stage；DAGScheduler需要记录那些RDD被存入磁盘等物化动作，同时要寻求Task的最优化调度，例如数据本地性等；DAGScheduler还要监视因为Shuffle输出导致的失败，如果发现这种失败，可能就要重新提交 该Stage：

[](http://s6.51cto.com/wyfs02/M00/4B/4C/wKiom1QpQEGS95HBAAB4sOQpc_I551.jpg)

DAGScheduler划分Stage后以TaskSet为单位把任务交给低层次的可插拔的调度器TaskScheduler来处理：

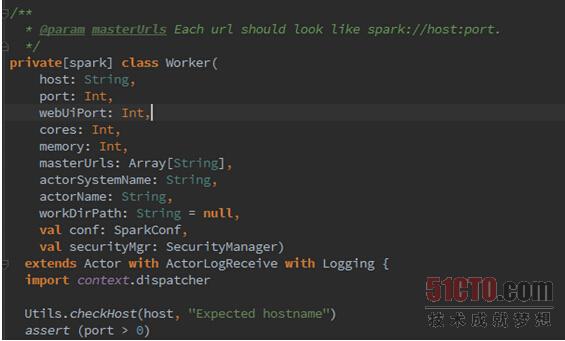
[](http://s6.51cto.com/wyfs02/M01/4B/4C/wKiom1QpQFST50_HAABHzUnehOk889.jpg)

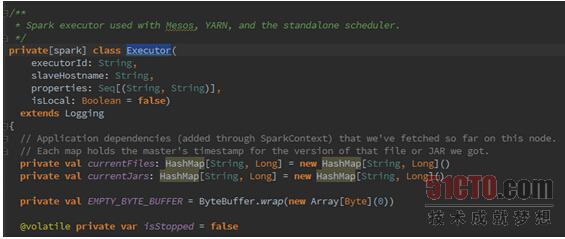
可以看出TaskScheduler是一个trait，在目前的Spark系统中TaskScheduler的实现类只有一个TaskSchedulerImpl：

[](http://s8.51cto.com/wyfs02/M02/4B/4E/wKioL1QpQJTBfThfAABnK4erJVE230.jpg)

一个TaskScheduler只为一个SparkContext实例服务，TaskScheduler接受来自DAGScheduler发送过来的分组的任务，DAGScheduler给TaskScheduler发送任务的时候是以Stage为单位来提交的，TaskScheduler收到任务后负责把任务分发到集群中Worker的Executor中去运行，如果某个task运行失败，TaskScheduler要负责重试；另外如果TaskScheduler发现某个Task一直未运行完，就可能启动同样的任务运行同一个Task，那个任务先运行完就用哪个任务的结果。

TaskScheduler发送的任务交给了Worker上的Executor以多线程的方式运行，每一个线程负责一个任务：

[](http://s4.51cto.com/wyfs02/M02/4B/4C/wKiom1QpQIGBvyLVAABabNOMtz4915.jpg)

[](http://s2.51cto.com/wyfs02/M00/4B/4E/wKioL1QpQMeQ6QCYAABUvrH6_dQ350.jpg)

其中的存储系统的管理是BlockManager来负责的：

[](http://s1.51cto.com/wyfs02/M00/4B/4C/wKiom1QpQLmxH9XhAABmU5NCJt4768.jpg)

下面看一下TaskSet的源码：

[](http://s7.51cto.com/wyfs02/M01/4B/4E/wKioL1QpQPGwQyXlAABR2RWTNoo047.jpg)

从TaskSet源码的第一个参数tasks就可以看出其是一个Task的数组，包含一组Task。

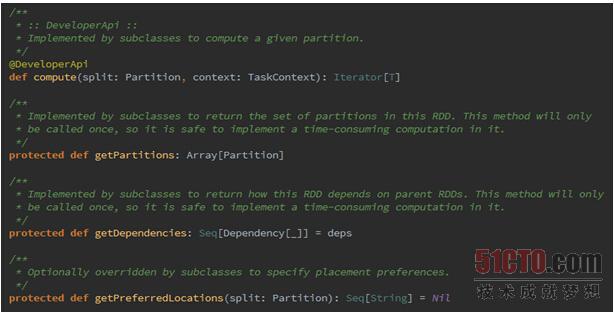
## Spark内核核心源码解析

### RDD源码解析

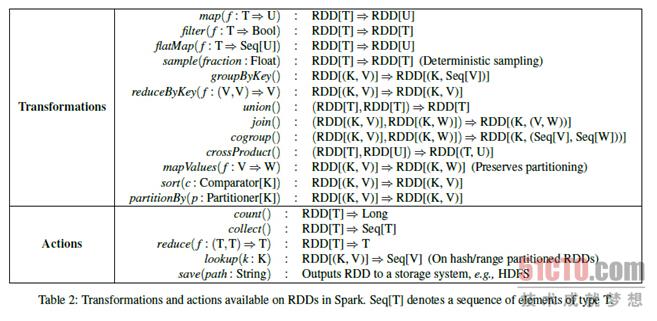
RDD的官方解释是：

[](http://s7.51cto.com/wyfs02/M00/4B/4E/wKioL1QpS3PBzCVgAAEADbHHKk0215.jpg)

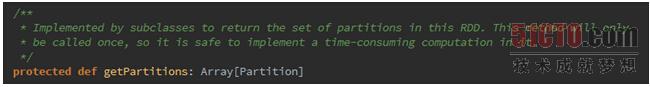
每个RDD的核心方法如下所示:

[](http://s4.51cto.com/wyfs02/M00/4B/4C/wKiom1QpS1LQ1m-2AABlBvr0Zgk926.jpg)

RDD主要分为两种：

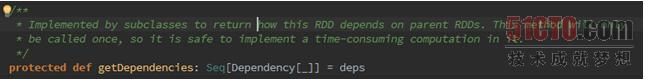
[](http://s2.51cto.com/wyfs02/M02/4B/4C/wKiom1QpSySAy0oqAAC8e1WWIEY817.jpg)

下面我们初步分析一下RDD的四个核心方法，首先看一下getPartitions方法的源码：

[](http://s1.51cto.com/wyfs02/M01/4B/4E/wKioL1QpS7Dxt5l0AAAqxiHfoSo380.jpg)

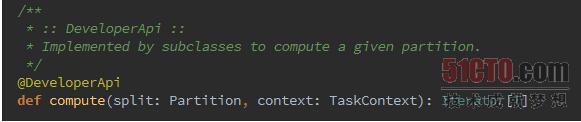
getPartitions返回的是一系列partitions的集合，即一个Partition类型的数组；

而getDependencies表达式RDD之间的依赖关系，如下所示：

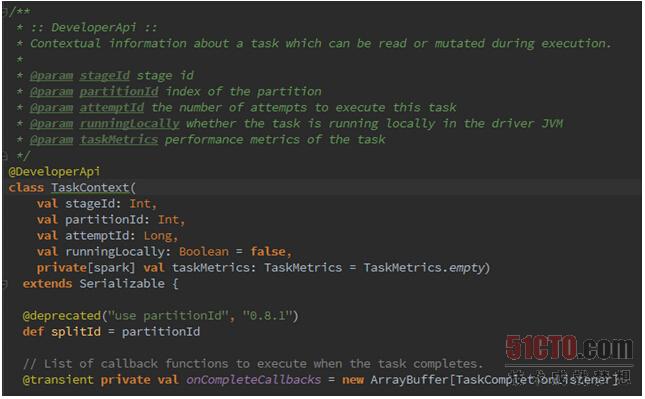
[](http://s7.51cto.com/wyfs02/M01/4B/4C/wKiom1QpS46A7YNEAAAsxMNwLok061.jpg)

getDependencies返回的是依赖关系的一个Seq集合，里面的Dependency数组中的下划线是类型的PlaceHolder；

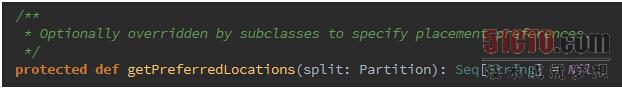
每个RDD都会具有计算的函数，如下所示：

[](http://s1.51cto.com/wyfs02/M02/4B/4E/wKioL1QpS8DiqejhAAAtkYMFwgQ106.jpg)

Compute方法是针对RDD的每个Partition进行计算的，其TaskContext参数的源码如下：

[](http://s3.51cto.com/wyfs02/M00/4B/4E/wKioL1QpTA7hfNMfAAB8dA40O8c622.jpg)

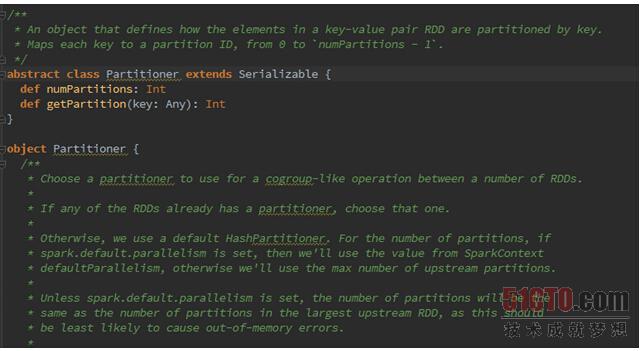
getPreferredLocations是寻找Partition的首选位置：

[](http://s5.51cto.com/wyfs02/M00/4B/4C/wKiom1QpS_PyK2mUAAAtdA56K88713.jpg)

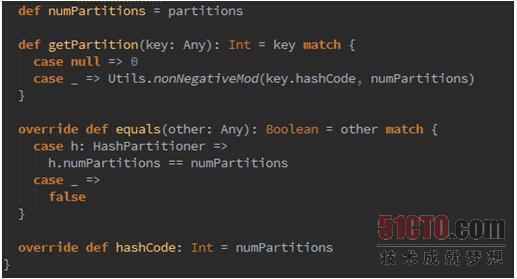
其实RDD还有一个可选的分区策略：

[http://s8.51cto.com/wyfs02/M01/4B/4E/wKioL1QpTC_T6qS3AAAmSdUmTq4532.jpg](http://s8.51cto.com/wyfs02/M01/4B/4E/wKioL1QpTC_T6qS3AAAmSdUmTq4532.jpg)

Partitioner的源码如下：

[](http://s6.51cto.com/wyfs02/M01/4B/4C/wKiom1QpTCqwDFhnAAB9nZWsLbw431.jpg)

[](http://s5.51cto.com/wyfs02/M02/4B/4E/wKioL1QpTGDC1VusAACD4t5pNRI058.jpg)

[](http://s4.51cto.com/wyfs02/M02/4B/4C/wKiom1QpTF-QGd0FAABPW_hd1eo286.jpg)

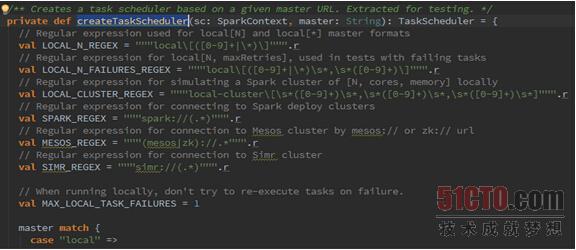
可以看出默认使用的是HashPartitioner，要注意key为Array的情况；

### SparkContext核心源码解析

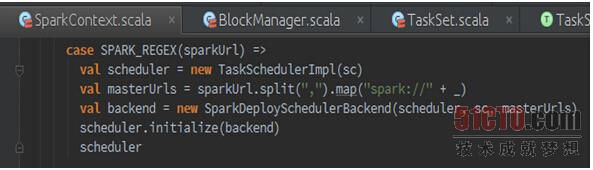
SparkContext在获得了一系列的初始化信息后开始创建并启动TaskScheduler实例：

[](http://s1.51cto.com/wyfs02/M01/4B/4E/wKioL1QpQd2QvuEDAABSPX4elNI148.jpg)

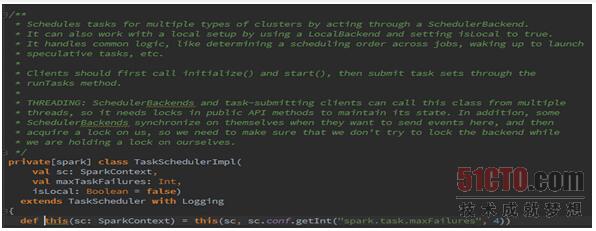
进入createTaskScheduler方法：

[](http://s9.51cto.com/wyfs02/M02/4B/4C/wKiom1QpQb3yWMUnAABsVSH-Ux4133.jpg)

我们看一下其Standalone的方式：

[](http://s9.51cto.com/wyfs02/M00/4B/4E/wKioL1QpQfGA0FCMAABJy58UudA735.jpg)

在上述代码中首先实例化一个TaskSchedulerImpl：

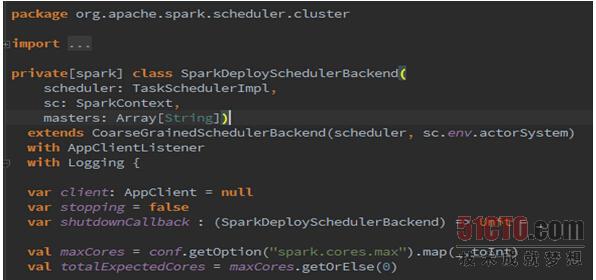
[](http://s7.51cto.com/wyfs02/M00/4B/4C/wKiom1QpQeLB6gniAABfJXuxG_0220.jpg)

然后构建出了masterUrls；

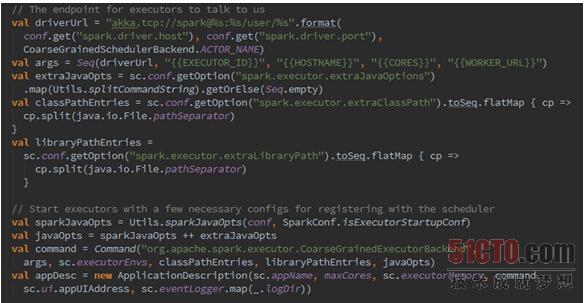
接着创建出了非常关键的backend：

[http://s5.51cto.com/wyfs02/M01/4B/4C/wKiom1QpQfyz-qI_AAAauddyIwA951.jpg](http://s5.51cto.com/wyfs02/M01/4B/4C/wKiom1QpQfyz-qI_AAAauddyIwA951.jpg)

我们进入其实现看一下：

[](http://s9.51cto.com/wyfs02/M02/4B/4C/wKiom1QpQiGB_2GHAABm-dChrUE164.jpg)

SparkDeploySchedulerBackend核心是为了启动CoarseGrainedExecutorBackend：

[](http://s2.51cto.com/wyfs02/M00/4B/4E/wKioL1QpQmyApJt2AACJs9PeO7k781.jpg)

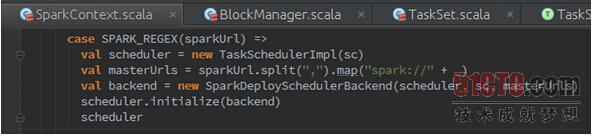
此处使用了Akka技术进行不同机器之间的通信，CoarseGrainedExecutorBackend是具体在Worker上执行具体的任务的进程的代表，所以我们的backend实例就是用来提交任务给Executor的：

[](http://s6.51cto.com/wyfs02/M02/4B/4E/wKioL1QpQnbDX2chAABd5MxhB78435.jpg)

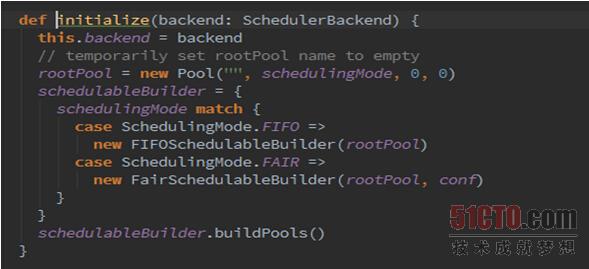
其实CoarseGrainedExecutorBackend是Executor的代理人，能够完成很多任务，例如启动一个任务：

[](http://s4.51cto.com/wyfs02/M00/4B/4C/wKiom1QpQnCzeSNIAABgo1D6jHs852.jpg)

回到Standalone的方式的代码处：

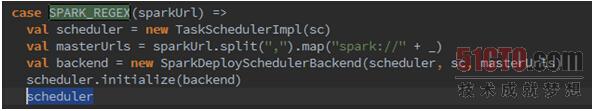
[](http://s8.51cto.com/wyfs02/M01/4B/4E/wKioL1QpQqeh_OeOAABETeeFiKI419.jpg)

接着代码是把backend传给了initialize方法中：

[](http://s1.51cto.com/wyfs02/M02/4B/4C/wKiom1QpQoPARgGXAABbfJU3IPU086.jpg)

在上述代码中显示处理调度模式例如FIFO和Fair的模式。

在代码块的最后返回实例化后的scheduer：

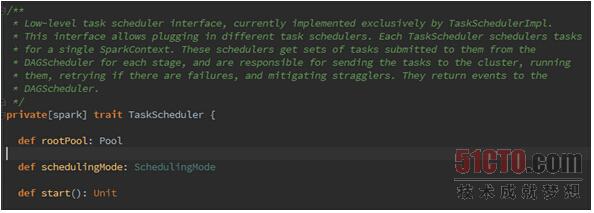
[](http://s9.51cto.com/wyfs02/M00/4B/4E/wKioL1QpQr2C17KTAAA7MYB-skc049.jpg)

### TaskSceduler启动源码解析

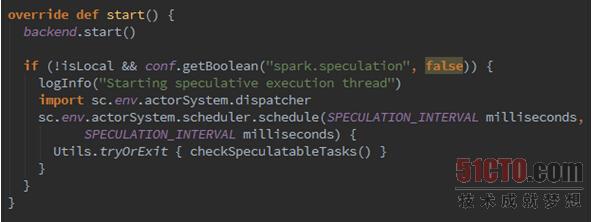
TaskScheduler实例对象启动源代码如下所示：

[](http://s7.51cto.com/wyfs02/M01/4B/4E/wKioL1QpQxCAmizWAAAiRONLkuY670.jpg)

进入start方法：

[](http://s9.51cto.com/wyfs02/M02/4B/4E/wKioL1QpQxyxjBduAABPXYiDA5w865.jpg)

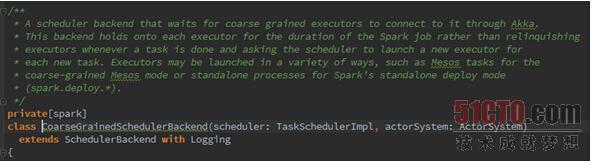
找到TaskSchedulerImpl实现类中的start方法实现：

[](http://s2.51cto.com/wyfs02/M02/4B/4C/wKiom1QpQv7T_u6RAABHWvgEYEc723.jpg)

其中TaskSchedulerImpl实例对象的主要目的在于启动backend，backend的类型如下所示：

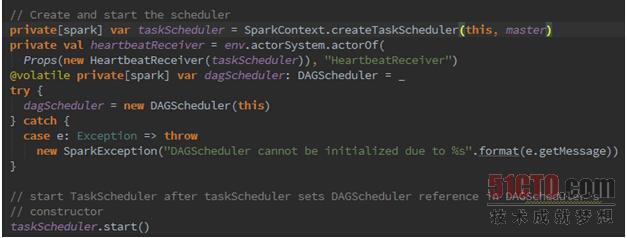
[](http://s5.51cto.com/wyfs02/M01/4B/4C/wKiom1QpQwiBu7eQAAAkksesoTw502.jpg)

SchedulerBackend的具体实现如下所示：

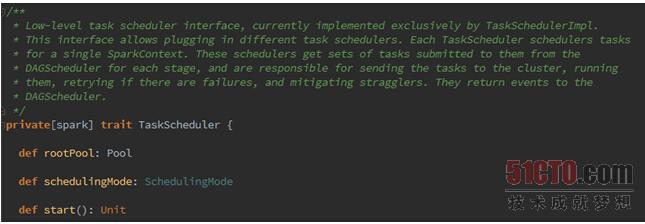
[](http://s2.51cto.com/wyfs02/M00/4B/4E/wKioL1QpQ0PznWKeAABMvDzDDSE700.jpg)

### Driver中AppClient源码解析

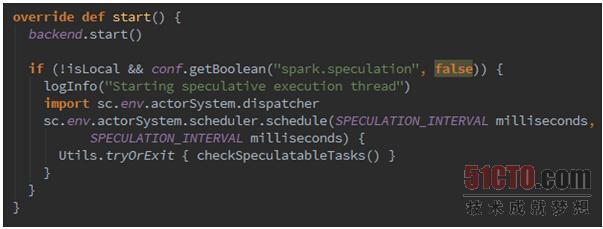
首先从SparkContext中TaskScheduler实例的创建开始：

[](http://s9.51cto.com/wyfs02/M00/4B/4E/wKioL1QpTSvwpb8vAABbHtYkHas033.jpg)

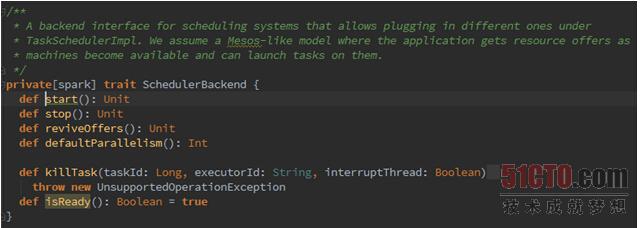
进入start代码内部：

[](http://s1.51cto.com/wyfs02/M00/4B/4C/wKiom1QpTQuCuONCAABWqKjidaw528.jpg)

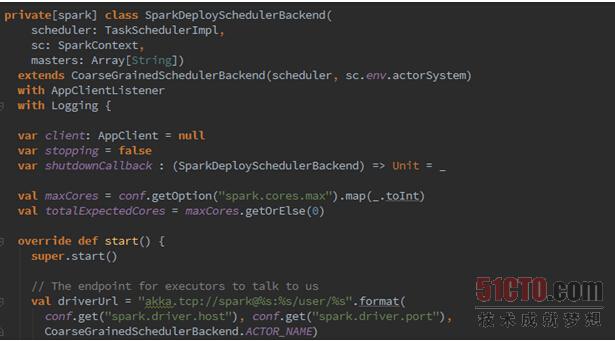
进入其实现者TaskSchedulerImpl内部：

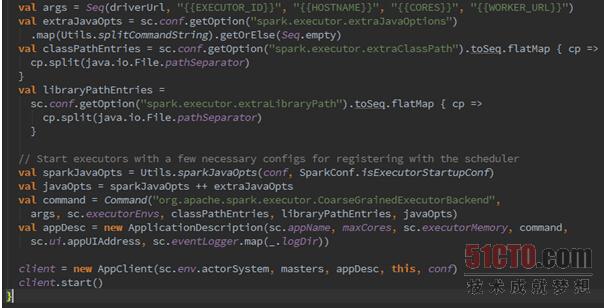
[](http://s9.51cto.com/wyfs02/M01/4B/4E/wKioL1QpTTzSYvJnAABSrf2WsEM511.jpg)

可以发现在start具体实现的内部首先是有个backend.start方法：

[](http://s4.51cto.com/wyfs02/M01/4B/4C/wKiom1QpTSnQsO5UAABUp0kB2jU756.jpg)

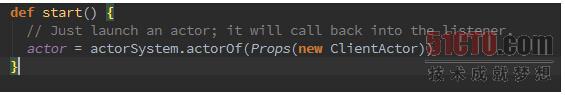
其最终具体的实现类为：

[](http://s7.51cto.com/wyfs02/M02/4B/4C/wKiom1QpTTyjHW7OAABsi15t8EA708.jpg)

[](http://s1.51cto.com/wyfs02/M02/4B/4E/wKioL1QpTXCjpMVqAACCpUbYi-g629.jpg)

从代码中可以看出，我们把CoarseGrainedExecutorBackend封装成command，然后交给appDesc，接着交给了Appclient，此时的AppClient就是客户端程序！

Appclient会调用start方法：

[](http://s6.51cto.com/wyfs02/M00/4B/4E/wKioL1QpTYGhQmOTAAAsGOuoRJw538.jpg)

此时启动了ClientActor：

[](http://s4.51cto.com/wyfs02/M00/4B/4C/wKiom1QpTW3Rk66_AABzzC4AZQM037.jpg)

#### AppClient注册Master

注册Master有两种，一种是registerWithMaster方法，一种是tryRegisterAllMasters方法，前者是单Master的情况，后者是多Master，一般情况下是满足HA机制，我们看一下registerWithMaster方法：

[](http://s8.51cto.com/wyfs02/M00/4B/4E/wKioL1QpTeaS7XI6AABqIJ5mm18920.jpg)

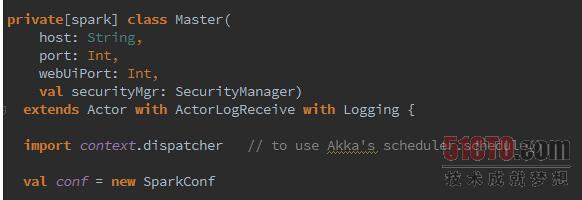
此时会发生tryRegisterAllMasters方法：

[](http://s3.51cto.com/wyfs02/M01/4B/4E/wKioL1QpTfGBcDl5AAA-ARZNTbc756.jpg)

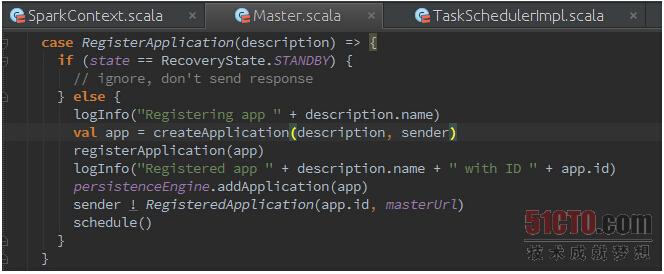
此时通过Akka通过消息机制发送消息给Master来注册程序，RegisterApplication是一个case class，来封装消息：

[](http://s7.51cto.com/wyfs02/M00/4B/4C/wKiom1QpTePhzJrSAAAsPIXUPJ8308.jpg)

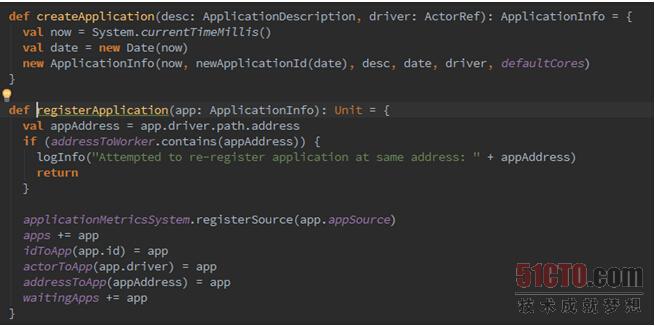
我们进入Master的源代码：

[](http://s4.51cto.com/wyfs02/M01/4B/4E/wKioL1QpTh3xICLfAAA4bRdT6FE082.jpg)

看一下接受客户端发送过来消息RegisterApplication的代码如下所示：

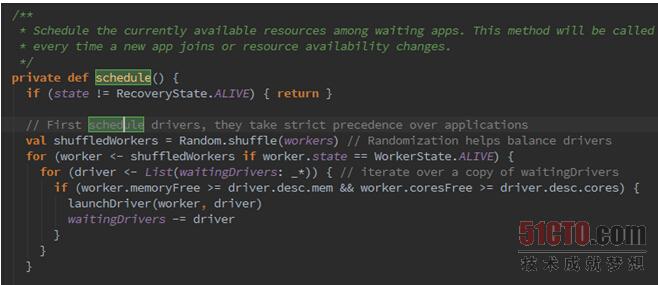
[](http://s8.51cto.com/wyfs02/M01/4B/4C/wKiom1QpTiHRbFmrAAB6hHwXiqw253.jpg)

此时首先使用ApplicationInfo构建一些准备信息，然后会导致registerApplication代码的调用：

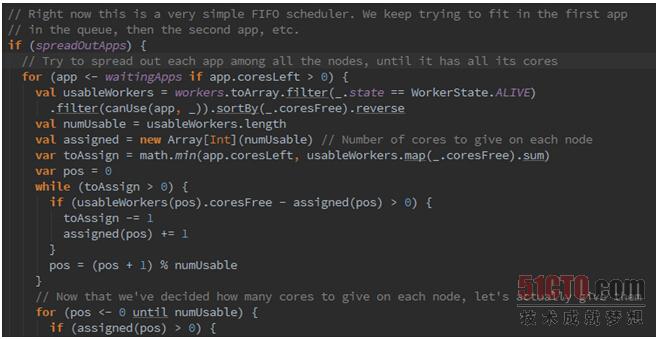
[](http://s9.51cto.com/wyfs02/M02/4B/4E/wKioL1QpTnWyhfsxAAB4Z7_SaQA820.jpg)

代码中就是一个注册应用的过程。

接着在Master的消息响应中会调用schedule方法：

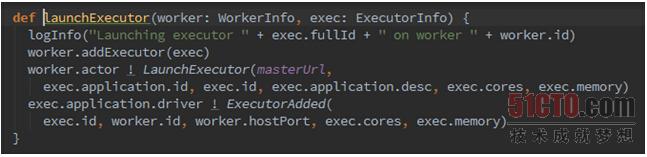
[](http://s9.51cto.com/wyfs02/M01/4B/4C/wKiom1QpTlnRmXhnAABpRUKSd4c647.jpg)

可以看到schedule方法中首先要启动Driver程序，也就是有main函数的程序，然后在schedule中会调度Worker的过程：

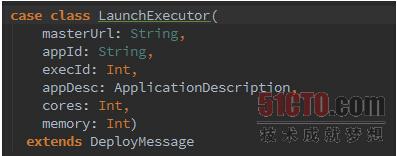
[](http://s2.51cto.com/wyfs02/M02/4B/4C/wKiom1QpTnTBxqZWAACHhojUF-8750.jpg)

[](http://s4.51cto.com/wyfs02/M02/4B/4E/wKioL1QpTqLj0eerAAB3mZauwAI068.jpg)

改代码会导致launchExecutor代码的执行：

[](http://s6.51cto.com/wyfs02/M00/4B/4E/wKioL1QpTr7CyuvHAABT28a4Z3g100.jpg)

在launchExecutor内部Master发送消息给Worker节点，消息为LaunchExecutor：

[](http://s8.51cto.com/wyfs02/M00/4B/4C/wKiom1QpTp7x7mjeAAAt25cOMr4123.jpg)

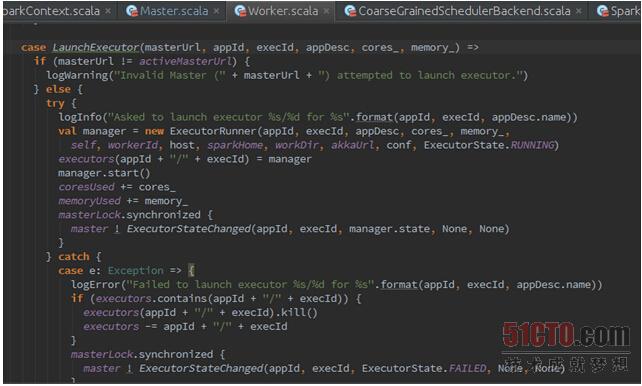
### Worker中Executor启动过程源代码解析

进入Worker源代码：

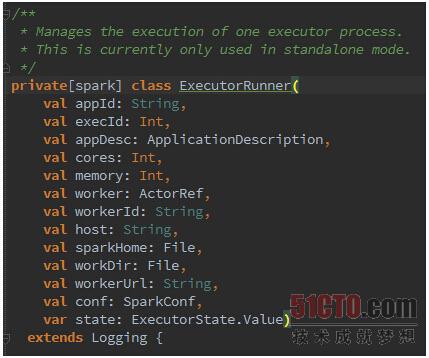
[](http://s8.51cto.com/wyfs02/M01/4B/4C/wKiom1QpTvbTkwbXAABdzs0I7dM279.jpg)

可以看出Worker本身是Akka中的一个Actor。

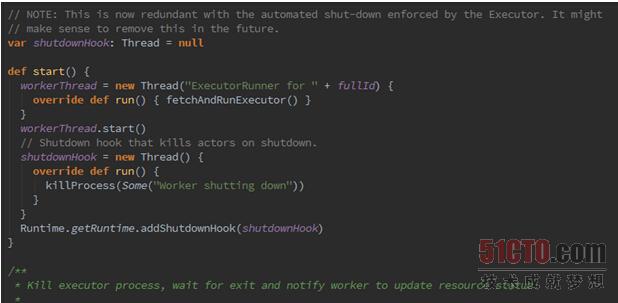
我们看一下Worker对LaunchExecutor消息的处理：

[](http://s3.51cto.com/wyfs02/M02/4B/4C/wKiom1QpTwjzUPUyAACVRoApllM822.jpg)

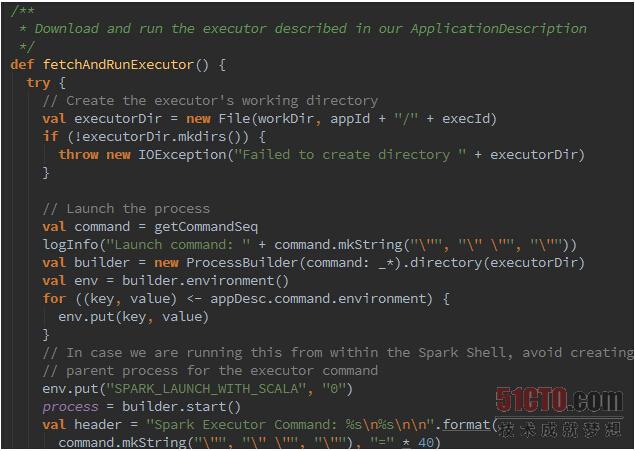
从源代码可以看出Worker节点上要分配CPU和Memory给新的Executor，首先需要创建一个ExecutorRunner：

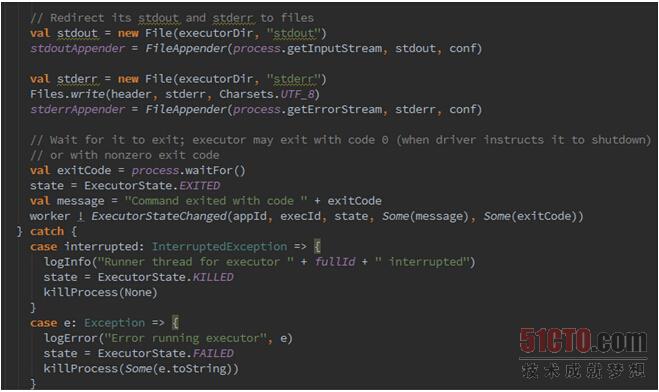
[](http://s6.51cto.com/wyfs02/M02/4B/4C/wKiom1QpTyXx0NORAABsrwf2gGQ974.jpg)

ExecutorRunner是用于维护executor进程的：

[](http://s3.51cto.com/wyfs02/M00/4B/4E/wKioL1QpT2OBAhmaAABTZjYBLts687.jpg)

其中最重要的方法是fetchAndRunExecutor：

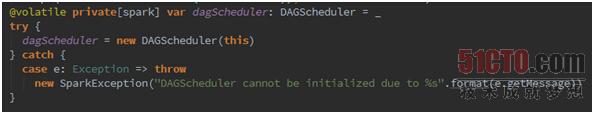
[](http://s1.51cto.com/wyfs02/M00/4B/4C/wKiom1QpT1aCxgdlAAC5GRM6v_I394.jpg)

[](http://s6.51cto.com/wyfs02/M01/4B/4E/wKioL1QpT5XiXwXPAACVBvABqmY094.jpg)

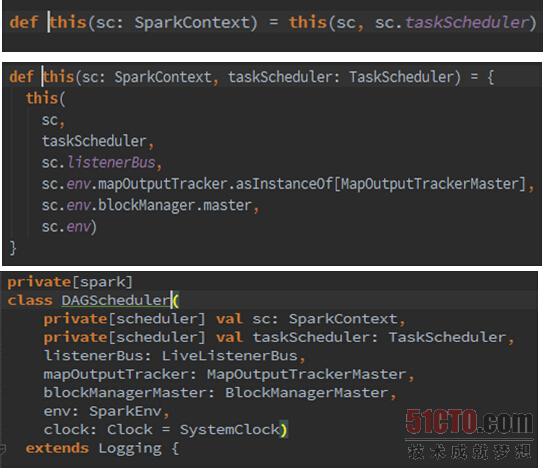
至此，Worker节点上的Executor启动运行。

### DAGScheduler源码解析

当构建完TaskScheduler之后，我们需要构建DAGScheduler这个核心对象：

[](http://s8.51cto.com/wyfs02/M00/4B/4C/wKiom1QpQ1XBYcTjAAAoKhY-U2g778.jpg)

进入其构造器代码：

[](http://s2.51cto.com/wyfs02/M01/4B/4C/wKiom1QpQ3DTASlXAACcCJ0EXWs487.jpg)

可以看出构建DAGScheduler实例的时候需要把TaskScheduler实例对象作为参数传入。

在DAGScheduler实例化最终调用其primary constructor的时候会导致以下函数的执行：

[](http://s9.51cto.com/wyfs02/M02/4B/4E/wKioL1QpQ6yiRF5bAAAjKOs8ADs925.jpg)

看一下该函数内部所做的事情：

[](http://s1.51cto.com/wyfs02/M01/4B/4E/wKioL1QpQ7yCKWZIAABDAW9EVlM820.jpg)

可以看出核心在于实例化eventProcessActor对象，eventProcessActor会负责接收和发送DAGScheduler的消息，是DAGScheduler的通信载体。

### Spark的Web监控源码解析

在SparkContext中可以看到如下代码：

[](http://s1.51cto.com/wyfs02/M01/4B/4C/wKiom1QpQ83xaClUAAApkJpuVow666.jpg)

首先是创建一个Spark Application的Web监控实例对象：

[](http://s2.51cto.com/wyfs02/M02/4B/4C/wKiom1QpQ-ajgEl2AAA8F6kvqKk190.jpg)

然后bind方法会绑定一个web服务器：

[](http://s1.51cto.com/wyfs02/M00/4B/4E/wKioL1QpRBeAg-e-AABE2emHZrQ817.jpg)

可以看出我们使用Jetty服务器来监控程序的运行和显示Spark集群的信息的。

### 提交代码源码初步解读

从官网等渠道了解spark提交应用程序的方法。

在通读一遍源码后，对源码解读进行整理、总结，以提高对源码的理解。

（个人经验：王老师的视频或书籍中，会讲解涉及到的重要部分，通读一遍源码后，再跟着视频或书籍看一遍，可以抓住重点，加快源码理解）

内容包含以下几个部分。

#### 入口点

入口脚本：.bin/spark-submit

Object：org.apache.spark.deploy.SparkSubmit

入口方法：def main(args: Array[String]) ——主构造函数做了哪些事？（忽略）

参数类：SparkSubmitArguments

#### 入口点开始源码解读

找到Object SparkSubmit入口方法：def main(args: Array[String])

**def** main(args: Array[String]) {  
 **val** appArgs = **new** SparkSubmitArguments(args)  
 **if** (appArgs.*verbose*) {  
 *printStream*.println(appArgs)  
 }  
 **val** (childArgs, classpath, sysProps, mainClass) = *createLaunchEnv*(appArgs)  
 *launch*(childArgs, classpath, sysProps, mainClass, appArgs.*verbose*)  
}

通过main源码解读可得最终调用：

1. mainClass = Class.*forName*(childMainClass, **true**, loader)
2. **val** mainMethod = mainClass.getMethod("main", **new** Array[String](0).getClass)
3. mainMethod.invoke(**null**, childArgs.toArray)

即：SparkSubmit最终调用的是childMainClass类的main静态方法。

问题：为什么最终调用的不是提交的mainClass，而是childMainClass类？

由该问题引入下一步：

#### 由SparkSubmit的主要功能指导源码解读

回到SparkSubmit的描述，对不同集群管理及部署模式的封装体现在childMainClass的设置（注意：我们提交的是mainClass类，调用childMainClass的main方法时，最终一定会调用我们提交的mainClass类，即以childMainClass的调用，封装了mainClass类的调用）

*/\*\*  
 \* Main gateway of launching a Spark application.  
 \*  
 \* This program handles setting up the classpath with relevant Spark dependencies and provides  
 \* a layer over the different cluster managers and deploy modes that Spark supports.  
 \*/*

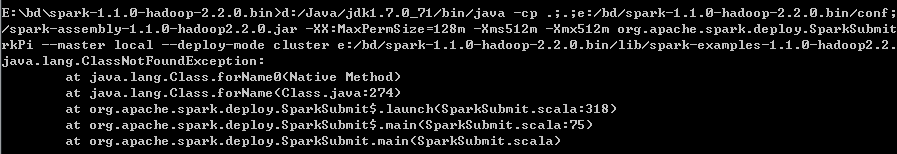
#### (集群管理,部署模式)源码解读

重点关注：集群管理+部署模式 ——> childMainClass的设置

集群管理+部署模式组合情况的childMainClass设置（提交mainClass类的封装）：

|  |  |  |  |
| --- | --- | --- | --- |
| **(clusterManager, deployMode)** |  | **childMainClass** | **说明** |
| (YARN,CLIENT) | (\_,CLIENT) | args.mainClass | Client模式：mainClass就在提交点执行main方法 |
| (STANDALONE ,CLIENT) |
| (MESOS ,CLIENT) |
| (LOCAL ,CLIENT) |
| (YARN, CLUSTER) | *isPython*  *isShell*  *isSqlShell* | "org.apache.spark.deploy.yarn.Client" |  |
| (STANDALONE , CLUSTER) | "org.apache.spark.deploy.Client" |
| (LOCAL , CLUSTER) | “”/args.mainClass | 代码可参考SparkContext对*isLocal的*处理 |
| (MESOS , CLUSTER) |  |  |  |

测试(LOCAL , CLUSTER):



测试(\_ , CLUSTER) + isShell :



红色部分：当前不支持的组合。

提交mainClass类的封装本质上就是如何将mainClass类设置到childMainClass中：

1. childArgs += (args.*master*, args.primaryResource, args.*mainClass*)
2. childArgs += ("--class", args.*mainClass*)

### 提交代码源码解读之(STANDALONE , CLUSTER)方式

(STANDALONE , CLIENT)方式，由上一节可知，是在提交点直接调用提交类mainClass的main方法（可以用提交WordCount来理解下）。

分析(STANDALONE , CLUSTER)方式的提交源码解读，两者可对比理解。

|  |  |
| --- | --- |
| **(clusterManager, deployMode)** | **childMainClass** |
| (STANDALONE , CLIENT) | args.mainClass |
| (STANDALONE , CLUSTER) | "org.apache.spark.deploy.Client" |

(STANDALONE , CLIENT) 与(STANDALONE , CLUSTER)两种提交的差异在于执行的主类不同，接下来对以CLUSTER部署模式提交时调用的Client类进行源码解读。

#### Client源码解读

1. 关键点

入口方法：def main(args: Array[String]) ——主构造函数做了哪些事？（忽略）

参数类：ClientArguments： SparkSubmit类在(STANDALONE , CLUSTER)提交模式下，SparkSubmitArguments 重新封装为ClientArguments。

1. 入口点的源码解读

Client的入口方法：

*/\*\*  
 \* Executable utility for starting and terminating drivers inside of a standalone cluster.  
 \* 在一个 Standalone 集群中，启动、停止 divers 的一个可执行工具类  
 \*/***object** Client {  
 **def** main(args: Array[String]) {

**……**

**val** (actorSystem, \_) = AkkaUtils.*createActorSystem*(  
 "driverClient", Utils.*localHostName*(), 0, conf, **new** SecurityManager(conf))  
  
 actorSystem.actorOf(Props(*classOf*[ClientActor], driverArgs, conf))  
  
 actorSystem.awaitTermination()  
 }  
}

即调用Client的main方法后，构建ActorSystem，然后通过ClientActor与Master进行通讯。

#### ClientActor源码解读

ClientActor是一个Actor，Actor的关键点：preStart()、receiveWithLogging，…

1. preStart() ：发送启动、停止Driver的消息

*RequestSubmitDriver* + *RequestKillDriver*

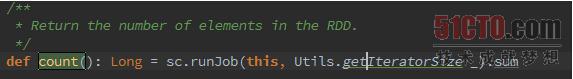
**val** command = **new** Command(mainClass, *Seq*("{{WORKER\_URL}}", driverArgs.*mainClass*) ++  
 driverArgs.driverOptions, sys.env, classPathEntries, libraryPathEntries, javaOpts)  
  
**val** driverDescription = **new** DriverDescription(  
 driverArgs.*jarUrl*,  
 driverArgs.*memory*,  
 driverArgs.*cores*,  
 driverArgs.*supervise*,  
 command)  
  
*masterActor* ! *RequestSubmitDriver*(driverDescription)

可以看到，SparkSubmit提交的mainClass被封装到了Command中，最终封装成DriverDescription，用RequestSubmitDriver消息发送到Master，由Master负责分配调度。

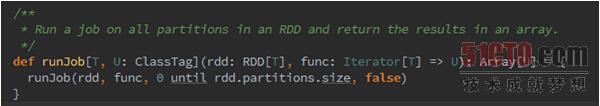
## Job全生命周期源码解读

### RDD的count操作为例触发Job全生命周期源码研究

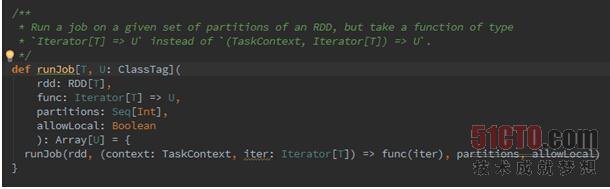
要想从源码级别研究Job全生命周期，往往需要action触发Job的运行，我们以RDD的count操作为例，如下所示：

[](http://s5.51cto.com/wyfs02/M00/4B/4E/wKioL1QpRPGzikyNAAAkY3oEryU806.jpg)

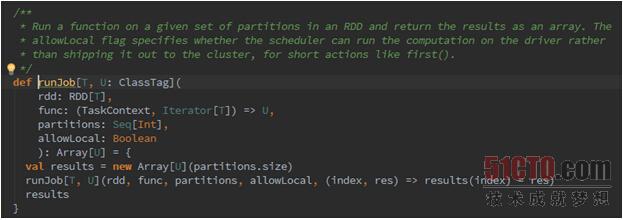
可以看到count方法触发SparkContext的runJob方法的调用：

[](http://s6.51cto.com/wyfs02/M01/4B/4C/wKiom1QpRM_BkVFRAAAvjrb-XIU239.jpg)

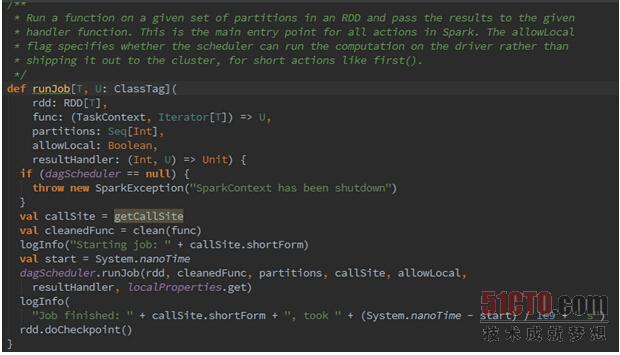
继续跟踪进去：

[](http://s6.51cto.com/wyfs02/M02/4B/4E/wKioL1QpRQKDqziCAAA6m8N5BuU647.jpg)

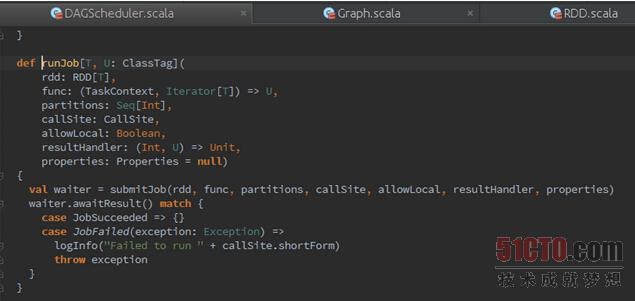
进一步跟踪下去：

[](http://s1.51cto.com/wyfs02/M02/4B/4C/wKiom1QpROTgGGIfAABTPP-2MKA748.jpg)

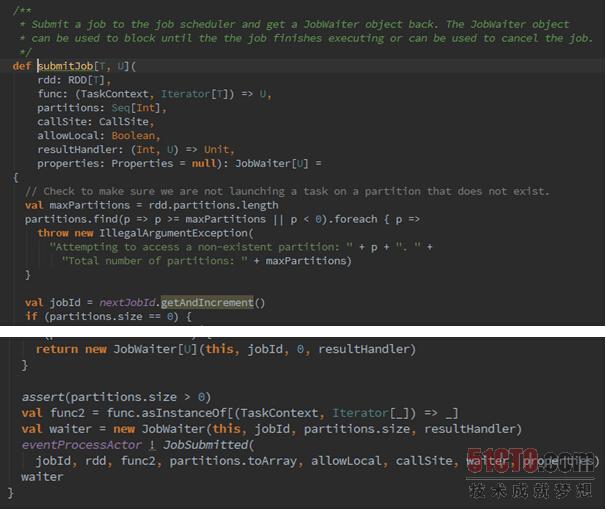
继续跟踪下去：

[](http://s1.51cto.com/wyfs02/M00/4B/4E/wKioL1QpRSejrs5bAAB6bHZLCnQ844.jpg)

此时发现进入DagScheduler的runJob方法：

[](http://s1.51cto.com/wyfs02/M00/4B/4C/wKiom1QpRQyizYn6AABWdRzev3s974.jpg)

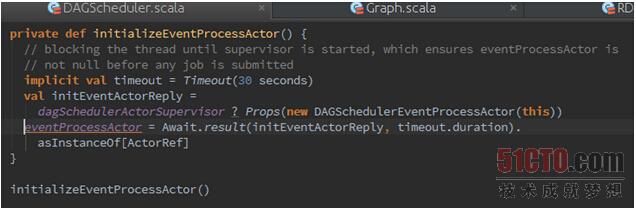
此时执行流程进入submitJob中：

[](http://s7.51cto.com/wyfs02/M01/4B/4E/wKioL1QpRUzj_dIpAACsPV_PqDA715.jpg)

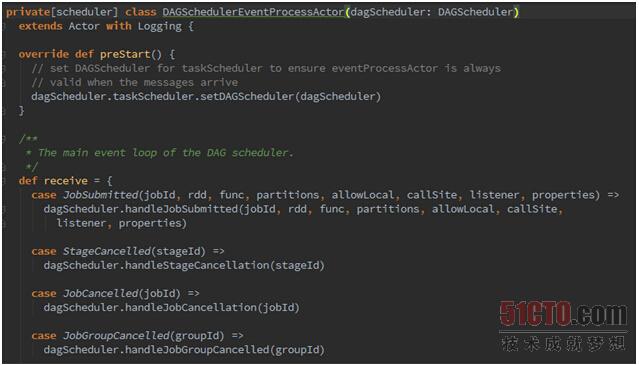
此时我们发现会向eventProcessActor发送JobSubmitted这个消息，JobSubmitted是一个case class：

[](http://s3.51cto.com/wyfs02/M01/4B/4C/wKiom1QpRWChd-tUAABZipicWWY062.jpg)

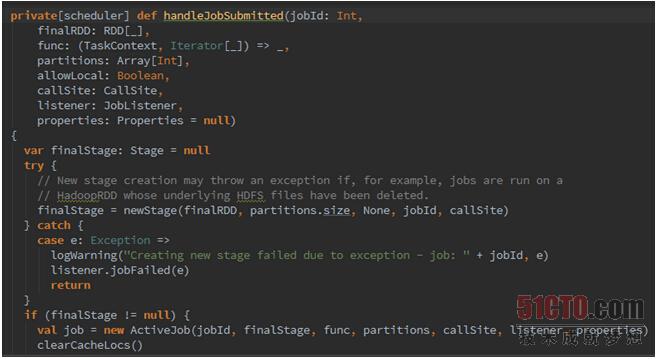
进入eventProcessActor定义的地方：

[](http://s2.51cto.com/wyfs02/M02/4B/4C/wKiom1QpRfzgvCd2AABccUQqal8547.jpg)

此时需要跟踪进DAGSchedulerEventProcessActor看具体的eventProcessActor的实现：

[](http://s9.51cto.com/wyfs02/M00/4B/4E/wKioL1QpRkLiv7vWAAB7YCS4RDo225.jpg)

可以看到我们的JobSubmitted消息是交给DAGScheduler的handleJobSubmitted方法来处理的：

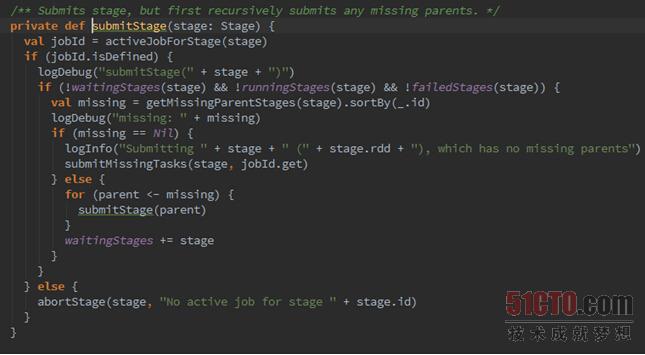
[](http://s4.51cto.com/wyfs02/M00/4B/4C/wKiom1QpRjeykcx6AAB8tole2yg196.jpg)

[](http://s6.51cto.com/wyfs02/M02/4B/4C/wKiom1QpRkWDzvttAACacDzleGw993.jpg)

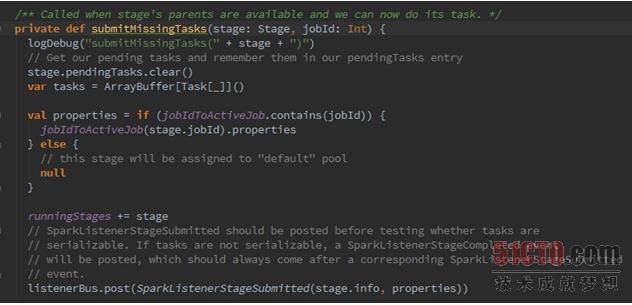
源代码中首先构建finalStage，然后又一个getMissingParentsStages方法，可以发现运行有本地运行和集群运行两种模式，本地运行主要用于本地实验和调试：

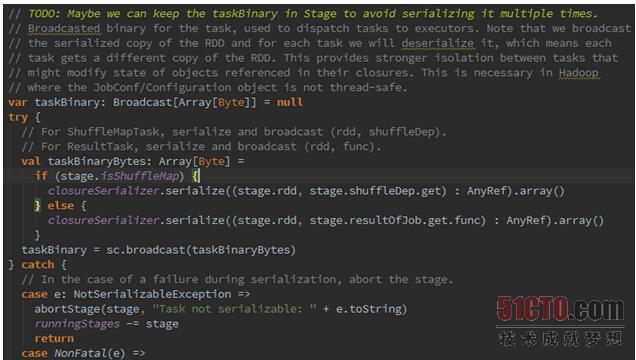
[](http://s7.51cto.com/wyfs02/M01/4B/4E/wKioL1QpRyHh81YIAABMNforeZo491.jpg)

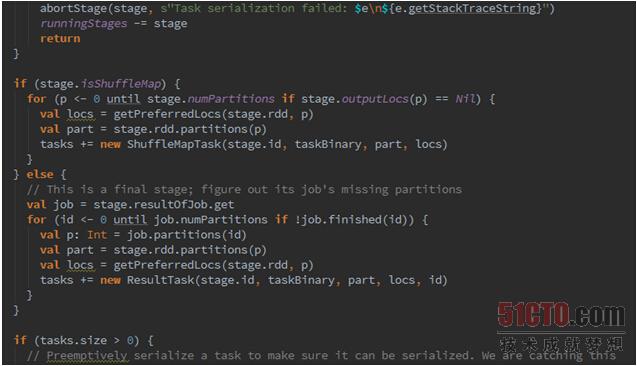
集群运行模式下回调用submitStage：

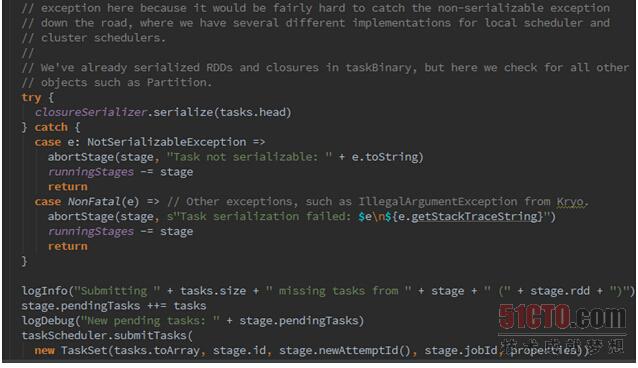
[](http://s6.51cto.com/wyfs02/M01/4B/4E/wKioL1QpRzXBfYpfAABub25udVA084.jpg)

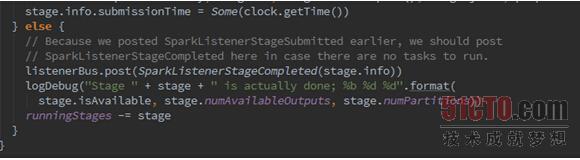
submitStage第一次传入的参数是Job的最后一个Stage，然后判断一下是否缺失父Stage，如果没有依赖的parent Stage的话就可以submitMissingTasks运行，如果有parent Stage的话就要再一次submitStage做递归操作，最终会导致submitMissingTasks的调用：

[](http://s8.51cto.com/wyfs02/M01/4B/4C/wKiom1QpRx6hNA2cAABzdTgxX9M855.jpg)

[](http://s1.51cto.com/wyfs02/M02/4B/4E/wKioL1QpR1vTmuEyAACaoSPDvgo492.jpg)

[](http://s1.51cto.com/wyfs02/M02/4B/4C/wKiom1QpR0jiB-2KAAB-e8PDVj8417.jpg)

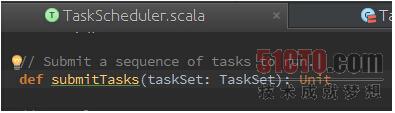
[](http://s4.51cto.com/wyfs02/M01/4B/4C/wKiom1QpR06S4UevAACNYB6aRY8076.jpg)

[](http://s1.51cto.com/wyfs02/M00/4B/4E/wKioL1QpR5GB68yNAABFNGVUKXU529.jpg)

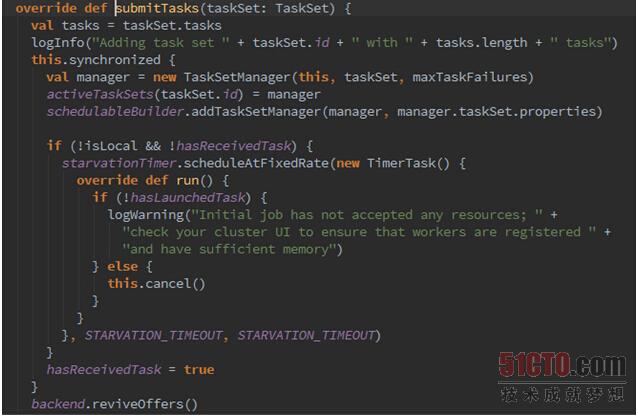
从源代码中可以看出DAGScheduler中向TaskScheduler以Stage为单位提交任务，Stage是以TaskSet为单位的：

[](http://s6.51cto.com/wyfs02/M00/4B/4C/wKiom1QpR6Li5RQ9AABV7UzuQTc031.jpg)

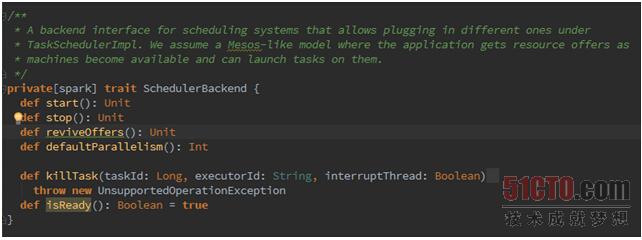
进入TaskScheduler的submitTasks：

[](http://s4.51cto.com/wyfs02/M01/4B/4E/wKioL1QpR-KDDF-9AAAlqkO07A8493.jpg)

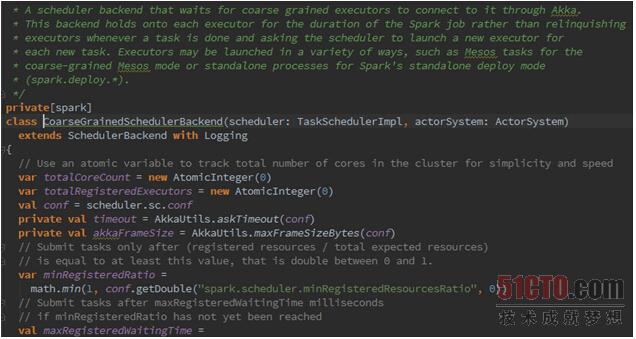
进入TaskSchedulerImpl中的具体实现：

[](http://s3.51cto.com/wyfs02/M01/4B/4C/wKiom1QpR8_hXoHqAACPKFMiZ3I552.jpg)

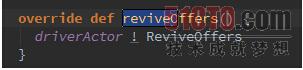
获取资源需要使用reviveOffers方法：

[](http://s1.51cto.com/wyfs02/M02/4B/4E/wKioL1QpSAuzWTVrAABcBhMg3ro418.jpg)

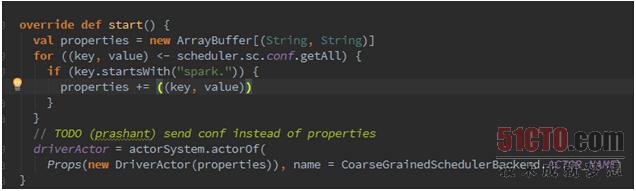
此时的SchedulerBackend是CoarseGrainedSchedulerBackend:

[](http://s4.51cto.com/wyfs02/M00/4B/4C/wKiom1QpSDPQE5_WAACf60dnS6I981.jpg)

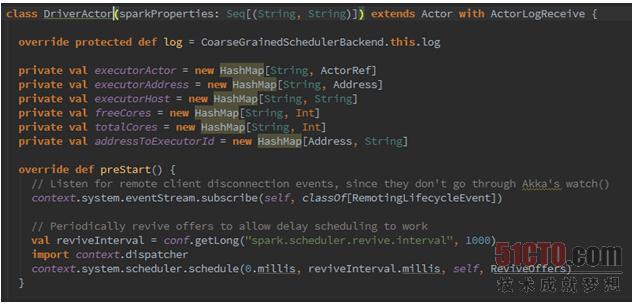
此时看reviveOffers方法的实现：

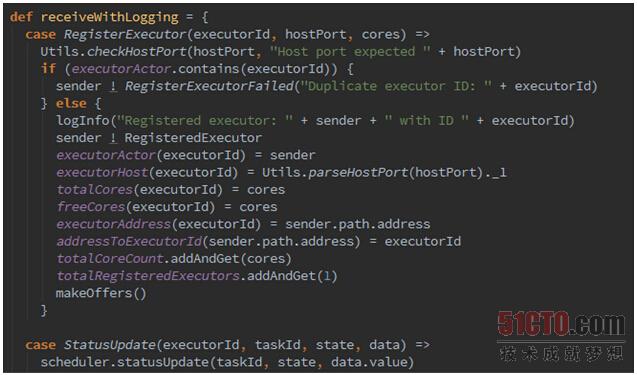
[](http://s8.51cto.com/wyfs02/M02/4B/4C/wKiom1QpSDjyCMV5AAAWNRKs7u4414.jpg)

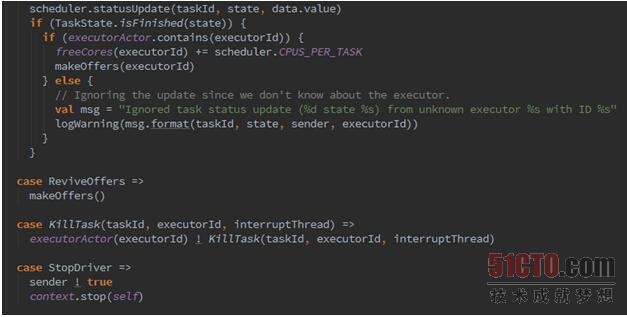
此时会向driverActor发送ReviveOffers消息，driverActor的实现代码如下：

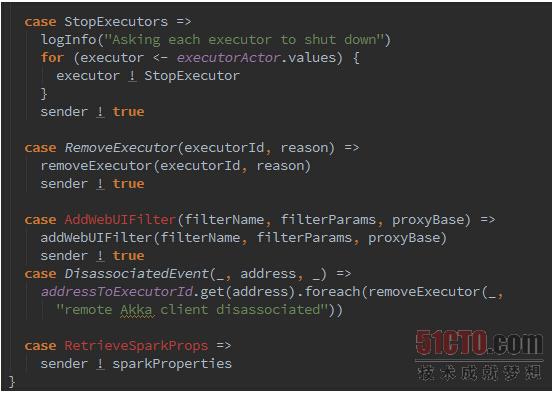
[](http://s8.51cto.com/wyfs02/M01/4B/4E/wKioL1QpSHHg-HtUAABNu36TuiE358.jpg)

此时跟踪进DriverActor的实现中：

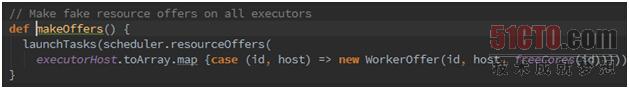
[](http://s7.51cto.com/wyfs02/M02/4B/4E/wKioL1QpSK2g7KyAAACEZhzSKpk467.jpg)

[](http://s4.51cto.com/wyfs02/M02/4B/4C/wKiom1QpSIvhvYROAACfSBBehw4293.jpg)

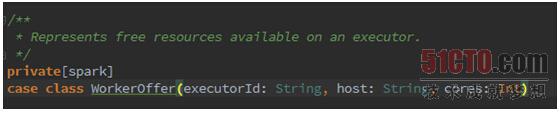
[](http://s5.51cto.com/wyfs02/M00/4B/4E/wKioL1QpSL6gLMCnAABlUNmkTbo911.jpg)

[](http://s4.51cto.com/wyfs02/M00/4B/4E/wKioL1QpSMrBu0j6AAB6nAv2NMc324.jpg)

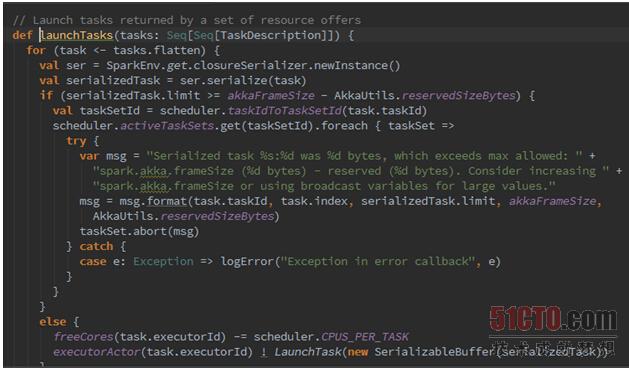
可以看到ReviveOffers消息的具体实现是makeOffers方法：

[](http://s1.51cto.com/wyfs02/M00/4B/4C/wKiom1QpSPKwd0X0AAAsjNIoJyE793.jpg)

WorkerOffer对象代表是某个Executor上可用的资源，freeCores(id)是该executor上空余的CPU数目：

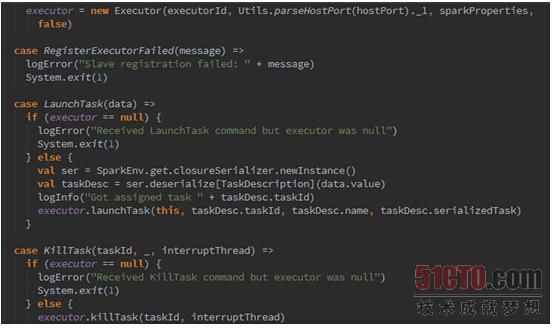
[](http://s8.51cto.com/wyfs02/M01/4B/4E/wKioL1QpSSOSgXqzAAArgp9VwhE761.jpg)

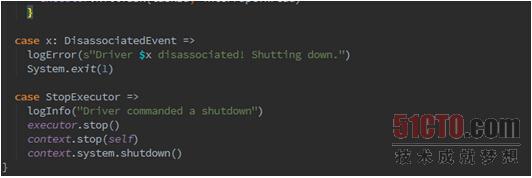
进入launchTasks：

[](http://s7.51cto.com/wyfs02/M01/4B/4C/wKiom1QpSQSwI5LVAACUpJzfeU4516.jpg)

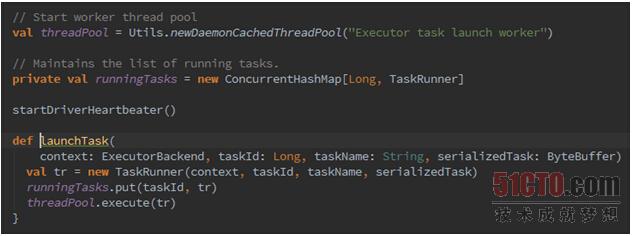
此时想executorActor发送启动Task的请求，其实是向CoarseGrainedExecutorBackend发送LaunchTask消息：

[](http://s7.51cto.com/wyfs02/M02/4B/4E/wKioL1QpSVawp9NaAAB7UsuUyx0667.jpg)

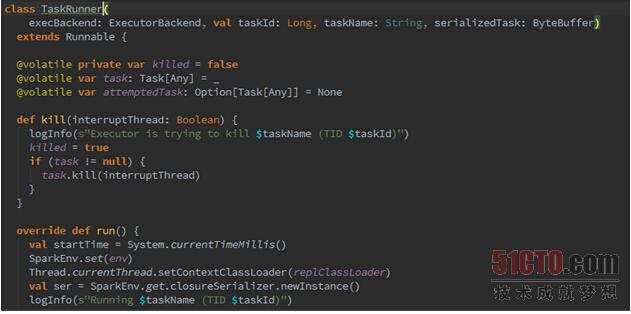
[](http://s4.51cto.com/wyfs02/M02/4B/4C/wKiom1QpSTPhA_LDAABqqfMYlBg963.jpg)

[](http://s7.51cto.com/wyfs02/M01/4B/4E/wKioL1QpSWTTyEY4AAAvS9nInEw241.jpg)

在LaunchTask消息中会导致executor.lauchTask的调用：

[](http://s8.51cto.com/wyfs02/M00/4B/4E/wKioL1QpSYLQeEEuAABXnZqUFuc278.jpg)

其中的TaskRunner封装了任务本身：

[](http://s9.51cto.com/wyfs02/M00/4B/4C/wKiom1QpSW_j7vv8AABoh0vdngE147.jpg)

任务执行的是交给了线程池去执行的。

# Spark性能优化

# Spark核心框架篇

# Spark Streaming

Spark streaming: 构建在Spark上处理Stream数据的框架，基本的原理是将Stream数据分成小的时间片断（几秒），以类似batch批量处理的方式来处理这小部分数据。Spark Streaming构建在Spark上，一方面是因为Spark的低延迟执行引擎（100ms+）可以用于实时计算，另一方面相比基于Record的其它处理框架（如Storm），RDD数据集更容易做高效的容错处理。此外小批量处理的方式使得它可以同时兼容批量和实时数据处理的逻辑和算法。方便了一些需要历史数据和实时数据联合分析的特定应用场合。

## 外部交互的数据流图



除了以上这些输入源，Spark Streaming还可以从普通TCP Socket 获取实时输入流数据。

输入数据流 (Input DStreams) 分类：

1. Basic sources: Sources directly available in the StreamingContext API. Example: file systems, socket connections, and Akka actors.
2. Advanced sources: Sources like Kafka, Flume, Kinesis, Twitter, etc. are available through extra utility classes. These require linking against extra dependencies as discussed in the [linking](http://spark.apache.org/docs/latest/streaming-programming-guide.html#linking) section.

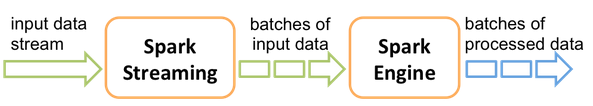
一个  [Receiver](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.receiver.Receiver) 关联一个Input DStreams（ file stream除外）。

一个  [Receiver](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.receiver.Receiver)  对应一个数据源，

所以 Input DStreams对应一个流数据。

一个streaming 应用程序可以同时创建多个Input DStreams并行接收多个流数据。

## Spark内部数据处理图



## SparkStreaming框架

**计算流程**：Spark Streaming是将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是Spark，也就是把Spark Streaming的输入数据按照batch size（如1秒）分成一段一段的数据（Discretized Stream），每一段数据都转换成Spark中的RDD（Resilient Distributed Dataset），然后将Spark Streaming中对DStream的Transformation操作变为针对Spark中对RDD的Transformation操作，将RDD经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加，或者存储到外部设备。图2显示了Spark Streaming的整个流程。

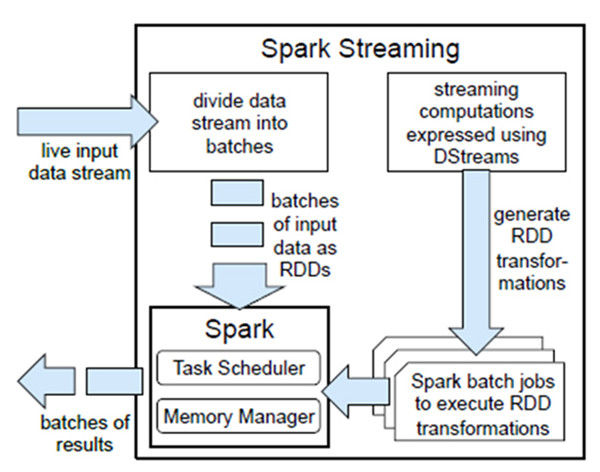
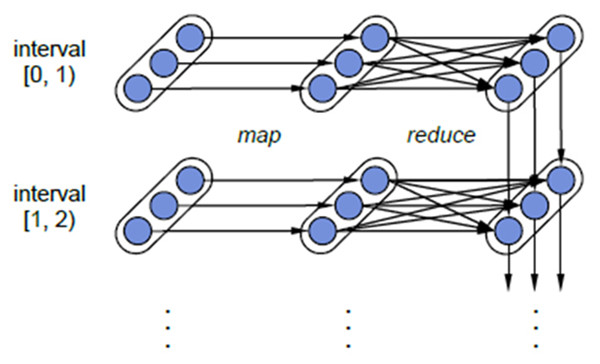
[[](http://cms.csdnimg.cn/article/201401/27/52e5f207e3d5b.jpg)](http://cms.csdnimg.cn/article/201401/27/52e5f207e3d5b.jpg)

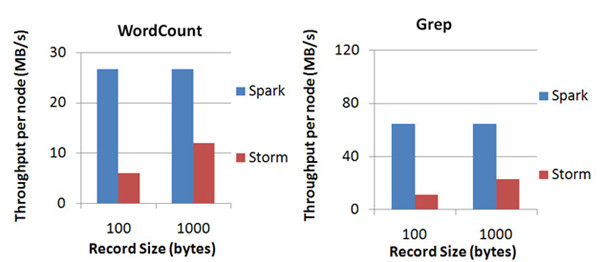
图2 Spark Streaming构架图

**容错性**：对于流式计算来说，容错性至关重要。首先我们要明确一下Spark中RDD的容错机制。每一个RDD都是一个不可变的分布式可重算的数据集，其记录着确定性的操作继承关系（lineage），所以只要输入数据是可容错的，那么任意一个RDD的分区（Partition）出错或不可用，都是可以利用原始输入数据通过转换操作而重新算出的。[](http://cms.csdnimg.cn/article/201401/27/52e5f1aaef7db.jpg)

**图3 Spark Streaming中RDD的lineage关系图**

对于Spark Streaming来说，其RDD的传承关系如图3所示，图中的每一个椭圆形表示一个RDD，椭圆形中的每个圆形代表一个RDD中的一个Partition，图中的每一列的多个RDD表示一个DStream（图中有三个DStream），而每一行最后一个RDD则表示每一个Batch Size所产生的中间结果RDD。我们可以看到图中的每一个RDD都是通过lineage相连接的，由于Spark Streaming输入数据可以来自于磁盘，例如HDFS（多份拷贝）或是来自于网络的数据流（Spark Streaming会将网络输入数据的每一个数据流拷贝两份到其他的机器）都能保证容错性。所以RDD中任意的Partition出错，都可以并行地在其他机器上将缺失的Partition计算出来。这个容错恢复方式比连续计算模型（如Storm）的效率更高。

**实时性**：对于实时性的讨论，会牵涉到流式处理框架的应用场景。Spark Streaming将流式计算分解成多个Spark Job，对于每一段数据的处理都会经过Spark DAG图分解，以及Spark的任务集的调度过程。对于目前版本的Spark Streaming而言，其最小的Batch Size的选取在0.5~2秒钟之间（Storm目前最小的延迟是100ms左右），所以Spark Streaming能够满足除对实时性要求非常高（如高频实时交易）之外的所有流式准实时计算场景。

扩展性与吞吐量：Spark目前在EC2上已能够线性扩展到100个节点（每个节点4Core），可以以数秒的延迟处理6GB/s的数据量（60M records/s），其吞吐量也比流行的Storm高2～5倍，图4是Berkeley利用WordCount和Grep两个用例所做的测试，在Grep这个测试中，Spark Streaming中的每个节点的吞吐量是670k records/s，而Storm是115k records/s。[](http://cms.csdnimg.cn/article/201401/27/52e5fc3d07287.jpg)

**图4 Spark Streaming与Storm吞吐量比较图**

## Spark Streaming的编程模型

Spark Streaming的编程和Spark的编程如出一辙，对于编程的理解也非常类似。对于Spark来说，编程就是对于RDD的操作；而对于Spark Streaming来说，就是对DStream的操作。下面将通过一个大家熟悉的WordCount的例子来说明Spark Streaming中的输入操作、转换操作和输出操作。

Spark Streaming初始化：在开始进行DStream操作之前，需要对Spark Streaming进行初始化生成StreamingContext。参数中比较重要的是第一个和第三个，第一个参数是指定Spark Streaming运行的集群地址，而第三个参数是指定Spark Streaming运行时的batch窗口大小。在这个例子中就是将1秒钟的输入数据进行一次Spark Job处理。

*val ssc = new StreamingContext(“Spark://…”, “WordCount”, Seconds(1), [Homes], [Jars])*

 Spark Streaming的输入操作：目前Spark Streaming已支持了丰富的输入接口，大致分为两类：一类是磁盘输入，如以batch size作为时间间隔监控HDFS文件系统的某个目录，将目录中内容的变化作为Spark Streaming的输入；另一类就是网络流的方式，目前支持Kafka、Flume、Twitter和TCP socket。在WordCount例子中，假定通过网络socket作为输入流，监听某个特定的端口，最后得出输入DStream（lines）。

*val lines = ssc.socketTextStream(“localhost”,8888)*

Spark Streaming的转换操作：与Spark RDD的操作极为类似，Spark Streaming也就是通过转换操作将一个或多个DStream转换成新的DStream。常用的操作包括map、filter、flatmap和join，以及需要进行shuffle操作的groupByKey/reduceByKey等。在WordCount例子中，我们首先需要将DStream(lines)切分成单词，然后将相同单词的数量进行叠加, 最终得到的wordCounts就是每一个batch size的（单词，数量）中间结果。

*val words = lines.flatMap(\_.split(“ ”))*  
*val wordCounts = words.map(x => (x, 1)).reduceByKey(\_ + \_)*

另外，Spark Streaming有特定的窗口操作，窗口操作涉及两个参数：一个是滑动窗口的宽度（Window Duration）；另一个是窗口滑动的频率（Slide Duration），这两个参数必须是batch size的倍数。例如以过去5秒钟为一个输入窗口，每1秒统计一下WordCount，那么我们会将过去5秒钟的每一秒钟的WordCount都进行统计，然后进行叠加，得出这个窗口中的单词统计。

*val wordCounts = words.map(x => (x, 1)).reduceByKeyAndWindow(\_ + \_, Seconds(5s)，seconds(1))*

但上面这种方式还不够高效。如果我们以增量的方式来计算就更加高效，例如，计算t+4秒这个时刻过去5秒窗口的WordCount，那么我们可以将t+3时刻过去5秒的统计量加上[t+3，t+4]的统计量，在减去[t-2，t-1]的统计量（如图5所示），这种方法可以复用中间三秒的统计量，提高统计的效率。

*val wordCounts = words.map(x => (x, 1)).reduceByKeyAndWindow(\_ + \_, \_ - \_, Seconds(5s)，seconds(1))*

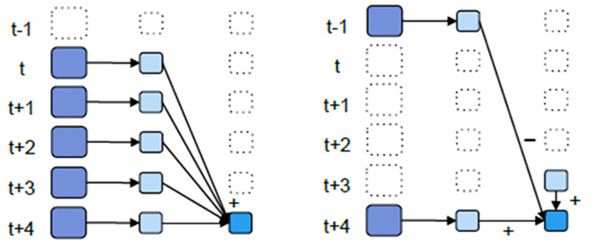
[](http://cms.csdnimg.cn/article/201401/27/52e5f3dc6fd38.jpg)

图5 Spark Streaming中滑动窗口的叠加处理和增量处理

Spark Streaming的输入操作：对于输出操作，Spark提供了将数据打印到屏幕及输入到文件中。在WordCount中我们将DStream wordCounts输入到HDFS文件中。

*wordCounts = saveAsHadoopFiles(“WordCount”)*

Spark Streaming启动：经过上述的操作，Spark Streaming还没有进行工作，我们还需要调用Start操作，Spark Streaming才开始监听相应的端口，然后收取数据，并进行统计。

*ssc.start()*

## Spark Streaming案例分析

在互联网应用中，网站流量统计作为一种常用的应用模式，需要在不同粒度上对不同数据进行统计，既有实时性的需求，又需要涉及到聚合、去重、连接等较为复杂的统计需求。传统上，若是使用Hadoop MapReduce框架，虽然可以容易地实现较为复杂的统计需求，但实时性却无法得到保证；反之若是采用Storm这样的流式框架，实时性虽可以得到保证，但需求的实现复杂度也大大提高了。Spark Streaming在两者之间找到了一个平衡点，能够以准实时的方式容易地实现较为复杂的统计需求。 下面介绍一下使用Kafka和Spark Streaming搭建实时流量统计框架。

数据暂存：Kafka作为分布式消息队列，既有非常优秀的吞吐量，又有较高的可靠性和扩展性，在这里采用Kafka作为日志传递中间件来接收日志，抓取客户端发送的流量日志，同时接受Spark Streaming的请求，将流量日志按序发送给Spark Streaming集群。

数据处理：将Spark Streaming集群与Kafka集群对接，Spark Streaming从Kafka集群中获取流量日志并进行处理。Spark Streaming会实时地从Kafka集群中获取数据并将其存储在内部的可用内存空间中。当每一个batch窗口到来时，便对这些数据进行处理。

结果存储：为了便于前端展示和页面请求，处理得到的结果将写入到数据库中。

相比于传统的处理框架，Kafka+Spark Streaming的架构有以下几个优点。

Spark框架的高效和低延迟保证了Spark Streaming操作的准实时性。

利用Spark框架提供的丰富API和高灵活性，可以精简地写出较为复杂的算法。

编程模型的高度一致使得上手Spark Streaming相当容易，同时也可以保证业务逻辑在实时处理和批处理上的复用。

在基于Kafka+Spark Streaming的流量统计应用运行过程中，有时会遇到内存不足、GC阻塞等各种问题。下面介绍一下如何对Spark Streaming应用程序进行调优来减少甚至避免这些问题的影响。

## 性能调优

### 优化运行时间

1. 增加并行度。确保使用整个集群的资源，而不是把任务集中在几个特定的节点上。对于包含shuffle的操作，增加其并行度以确保更为充分地使用集群资源。
2. 减少数据序列化、反序列化的负担。Spark Streaming默认将接收到的数据序列化后存储以减少内存的使用。但序列化和反序列化需要更多的CPU时间，因此更加高效的序列化方式（Kryo）和自定义的序列化接口可以更高效地使用CPU。
3. 设置合理的batch窗口。在Spark Streaming中，Job之间有可能存在着依赖关系，后面的Job必须确保前面的Job执行结束后才能提交。若前面的Job执行时间超出了设置的batch窗口，那么后面的Job就无法按时提交，这样就会进一步拖延接下来的Job，造成后续Job的阻塞。因此，设置一个合理的batch窗口确保Job能够在这个batch窗口中结束是必须的。
4. 减少任务提交和分发所带来的负担。通常情况下Akka框架能够高效地确保任务及时分发，但当batch窗口非常小（500ms）时，提交和分发任务的延迟就变得不可接受了。使用Standalone模式和Coarse-grained Mesos模式通常会比使用Fine-Grained Mesos模式有更小的延迟。

### 优化内存使用

1. 控制batch size。Spark Streaming会把batch窗口内接收到的所有数据存放在Spark内部的可用内存区域中，因此必须确保当前节点Spark的可用内存至少能够容纳这个batch窗口内所有的数据，否则必须增加新的资源以提高集群的处理能力。
2. 及时清理不再使用的数据。上面说到Spark Streaming会将接收到的数据全部存储于内部的可用内存区域中，因此对于处理过的不再需要的数据应及时清理以确保Spark Streaming有富余的可用内存空间。通过设置合理的spark.cleaner.ttl时长来及时清理超时的无用数据。
3. 观察及适当调整GC策略。GC会影响Job的正常运行，延长Job的执行时间，引起一系列不可预料的问题。观察GC的运行情况，采取不同的GC策略以进一步减小内存回收对Job运行的影响。

## 核心术语

1. discretized stream or DStream

高级抽象概念，表示持续流数据。

在内部DStream 表现为一个 RDDs序列。

## 源码解析

### 数据源源码

1. 普通TCP套接字对应的数据源

方法：ssc.socketTextStream(…)

记录数据：每一条记录对应一个文本行( a line of text)。

# Spark SQL

# MLlib(Machine Learning library)

# GraphX(Graph Processing)

# Bagel (Pregel on Spark)

Bagel: Pregel on Spark，可以用Spark进行图计算，这是个非常有用的小项目。Bagel自带了一个例子，实现了Google的PageRank算法。

# SparkR

SparkR是AMPLab发布的一个R开发包，为Apache Spark提供了轻量的前端。SparkR提供了Spark中弹性分布式数据集（RDD）的API，用户可以在集群上通过R shell交互性的运行job。

开源地址：https://github.com/amplab-extras/SparkR-pkg

简介：https://github.com/amplab-extras/SparkR-pkg#readme

## R on Spark

SparkR是一个 R package，向从R转到Spark者提供了轻量级的前端。

### 安装

#### 安装要求

1. 版本要求

Scala 2.10 + Spark version >= 0.9.0.

当前默认使用apache仓库上的1.1.0版本进行构建。

可以通过修改 pkg/src/build.sbt 文件来指定对应的spark版本号。

1. rJava

SparkR也要求安装名为rJava 的 R package， 在R中，可以通过以下命令来安装：

install.packages("rJava")

#### package安装

可以通过以下命令构建scala package 和 R package，用于SparkR开发：

./install-dev.sh

To develop SparkR, you can build the scala package and the R package using

./install-dev.sh

If you wish to try out the package directly from github, you can use [install\_github](http://www.inside-r.org/packages/cran/devtools/docs/install_github) from [devtools](http://www.inside-r.org/packages/cran/devtools). Note that you can specify which branch, tag etc to install from.

library(devtools)

install\_github("amplab-extras/SparkR-pkg", subdir="pkg")

SparkR by default links to Hadoop 1.0.4. To use SparkR with other Hadoop versions, you will need to rebuild SparkR with the same version that [Spark is linked to](http://spark.apache.org/docs/latest/index.html#a-note-about-hadoop-versions). For example to use SparkR with a CDH 4.2.0 MR1 cluster, you can run

SPARK\_HADOOP\_VERSION=2.0.0-mr1-cdh4.2.0 ./install-dev.sh

By default, SparkR uses [sbt](http://www.scala-sbt.org/) to build an assembly jar. If you wish to use [maven](http://maven.apache.org/) instead, you can set the environment variable USE\_MAVEN=1. For example

USE\_MAVEN=1 ./install-dev.sh

If you are building SparkR from behind a proxy, you can [setup maven](https://maven.apache.org/guides/mini/guide-proxies.html) to use the right proxy server.

## Running sparkR

If you have cloned and built SparkR, you can start using it by launching the SparkR shell with

./sparkR

The sparkR script automatically creates a SparkContext with Spark by default in local mode. To specify the Spark master of a cluster for the automatically created SparkContext, you can run

MASTER=<Spark master URL> ./sparkR

If you have installed it directly from github, you can include the SparkR package and then initialize a SparkContext. For example to run with a local Spark master you can launch R and then run

library(SparkR)

sc <- sparkR.init(master="local")

To increase the memory used by the driver you can export the SPARK\_MEM environment variable. For example to use 1g, you can run

SPARK\_MEM=1g ./sparkR

In a cluster setting to set the amount of memory used by the executors you can pass the variablespark.executor.memory to the SparkContext constructor.

library(SparkR)

sc <- sparkR.init(master="spark://<master>:7077",

sparkEnvir=list(spark.executor.memory="1g"))

## Examples, Unit tests

SparkR comes with several sample programs in the examples directory. To run one of them, use./sparkR <filename> <args>. For example:

./sparkR examples/pi.R local[2]

You can also run the unit-tests for SparkR by running

./run-tests.sh

## Running on EC2

Instructions for running SparkR on EC2 can be found in the [SparkR wiki](https://github.com/amplab-extras/SparkR-pkg/wiki/SparkR-on-EC2).

## Running on YARN

Currently, SparkR supports running on YARN with the yarn-client mode. These steps show how to build SparkR with YARN support and run SparkR programs on a YARN cluster:

# assumes Java, R, rJava, yarn, spark etc. are installed on the whole cluster.

cd SparkR-pkg/

USE\_YARN=1 SPARK\_YARN\_VERSION=2.4.0 SPARK\_HADOOP\_VERSION=2.4.0 ./install-dev.sh

Before launching an application, make sure each worker node has a local copy of lib/SparkR/sparkr-assembly-0.1.jar. With a cluster launched with the spark-ec2 script, do:

~/spark-ec2/copy-dir ~/SparkR-pkg

Finally, when launching an application, the environment variable YARN\_CONF\_DIR needs to be set to the directory which contains the client-side configuration files for the Hadoop cluster (with a cluster launched with spark-ec2, this defaults to /root/ephemeral-hdfs/conf/):

YARN\_CONF\_DIR=/root/ephemeral-hdfs/conf/ MASTER=yarn-client ./sparkR

YARN\_CONF\_DIR=/root/ephemeral-hdfs/conf/ ./sparkR examples/pi.R yarn-client

## Report Issues/Feedback

For better tracking and collaboration, issues and TODO items are reported to a dedicated [SparkR JIRA](https://sparkr.atlassian.net/browse/SPARKR/).

In your pull request, please cross reference the ticket item created. Likewise, if you already have a pull request ready, please reference it in your ticket item.

# Spark开发与应用

## Spark-shell

# 部署

## 提交应用程序

## Amazon EC2

## Standalone模式

## Mesos

## YARN

# 监控

## Spark的Web监控页面

# 配置

# 调优

# 作业调度

# 安全

# 硬件配置

# 技巧、FAQ

## Spark性能优化的10大问题及其解决方案

问题1：reduce task数目不合适

解决方式：

需根据实际情况调节默认配置，调整方式是修改参数spark.default.parallelism。通常，reduce数目设置为core数目的2到3倍。数量太大，造成很多小任务，增加启动任务的开销；数目太少，任务运行缓慢。

问题2：shuffle磁盘IO时间长

解决方式：

设置spark.local.dir为多个磁盘，并设置磁盘为IO速度快的磁盘，通过增加IO来优化shuffle性能；

问题3：map|reduce数量大，造成shuffle小文件数目多

解决方式：

默认情况下shuffle文件数目为map tasks \* reduce tasks

通过设置spark.shuffle.consolidateFiles为true，来合并shuffle中间文件，此时文件数为reduce tasks数目；

问题4：序列化时间长、结果大

解决方式：

Spark默认使.用JDK.自带的ObjectOutputStream，这种方式产生的结果大、CPU处理时间长，可以通过设置spark.serializer为org.apache.spark.serializer.KryoSerializer。

另外如果结果已经很大，可以使用广播变量；

问题5：单条记录消耗大

解决方式：

使用mapPartition替换map，mapPartition是对每个Partition进行计算，而map是对partition中的每条记录进行计算；

问题6 : collect输出大量结果时速度慢

解决方式：

collect源码中是把所有的结果以一个Array的方式放在内存中，可以直接输出到分布式?文件系统，然后查看文件系统中的内容；

问题7: 任务执行速度倾斜

解决方式：

如果是数据倾斜，一般是partition key取的不好，可以考虑其它的并行处理方式 ，并在中间加上aggregation操作；

如果是Worker倾斜，例如在某些worker上的executor执行缓慢，可以通过设置spark.speculation=true 把那些持续慢的节点去掉；

问题9: 通过多步骤的RDD操作后有很多空任务或者小任务产生

解决方式：

使用coalesce或repartition去减少RDD中partition数量；

问题10：Spark Streaming吞吐量不高

解决方式：

可以设置spark.streaming.concurrentJobs

## 提交失败

1. TaskSchedulerImpl 调度失败

报错信息:

TaskSchedulerImpl: Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient memory

原因分析：1. worker注册失败，导致资源不够用; 2. 现有资源（core和memory）不满足提交时的资源申请要求；

根本解决方案：

* 查看worker节点日志：找出注册失败原因。
* 熟悉Master对 Application的调度机制，一方面core，另一方面memory（必须满足可用的条件）。

1. Master URL的设置

在conf里设置master url时，必须和界面上显示的一样，都使用主机名或都使用ip。对应的值在spark-default.conf中的。。。。

## 页面相关

1. 在界面跳转时，找不到页面的话，查看下本地的hosts中是否有该主机名的映射

## 应用程序管理

### 如何终止应用程序

根据提交方式：

1. Client 部署模式提交时
   * 通用方法

在提交点直接使用kill命令杀掉名为SparkSubmit进程

ps -aux |grep SparkSubmit

kill -9 $pid // pid 为ps后的进程ID值

1. Cluster部署模式提交时：

* Standalone 集群

在UI的Stage界面？ 点击kill（需设置相应配置属性），杀掉应用程序；

在UI界面获取Driver所在Worker节点，通过kill命令，杀掉名为DriverWrapper的进程，命令参考前面；

使用…deploy.Client类支持的--kill参数，利用spark的脚本 ./spark-class，通过akka消息发送，杀掉driver。

* Yarn集群

使用Yarn自带的job管理命令，杀掉进程：

  Hadoop job –list ： 查询job列表

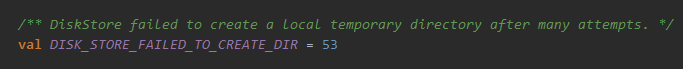
hadoop job -kill  jobid：kill指定id的job

## Exited Code

### 53

提问：北京-spark-小唐spark-shell进入后一直出现这个日志，这是咋回事，哪位大牛解释下。

解答：南京-东 2014/12/18 15:25:50

 这是源码上的提示，创建一个本地的临时目录失败，你在本地节点上查看下，一个是权限问题，一个是目录结构问题。跟其他节点比较下

原因：北京-spark-小唐 —— 磁盘空间不足

## 群问题记录

### Spark亚太研究院3群 - 安留军

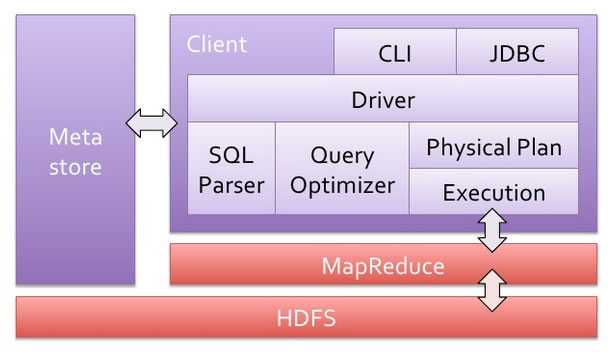
1. 问题

mkdir: Permission denied: user=spark, access=WRITE, inode="/":hdfs:supergroup:drwxr-xr-x  
14/12/28 17:46:13 WARN fs.FileSystem: "192.168.192.114" is a deprecated filesystem name. Use "hdfs://192.168.192.114/" instead.  
14/12/28 17:46:14 WARN fs.FileSystem: "192.168.192.114" is a deprecated filesystem name. Use "hdfs://192.168.192.114/" instead.  
put: `/user/spark/share/lib/spark-assembly.jar': No such file or directory  
  
增加权限（不能修复） 看到回复下我。 这一步解决不了。还是报这个错误  
sudo -u hdfs hadoop fs -mkdir -p /user  
sudo -u hdfs hadoop fs -chown spark:spark /user

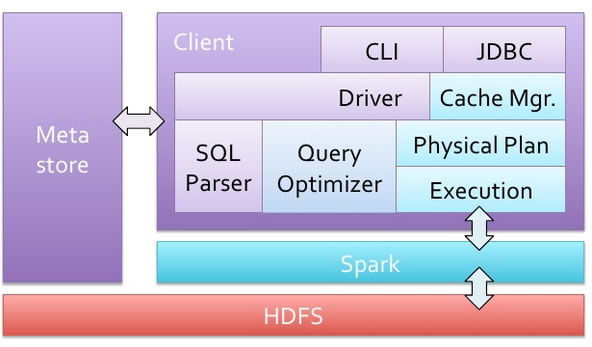
# 参考资料

## Shark

Shark是基于Spark上的“Hive”，看看基于hadoop的Hive：



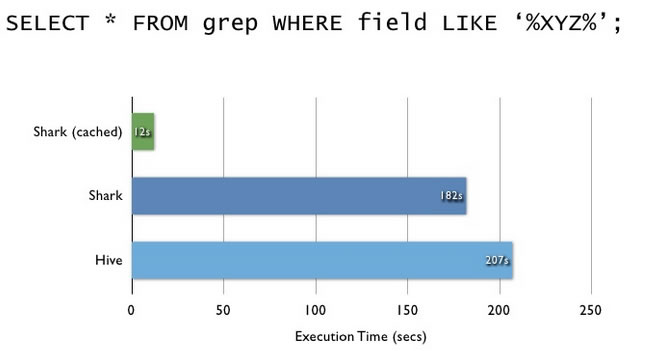
而Shark的结构图：



Hive是记录每行记录一个对象，而shark是每列记录：



执行SQL时间对比：



## Spark的适用场景

* Spark是基于内存的迭代计算框架，适用于需要多次操作特定数据集的应用场合。需要反复操作的次数越多，所需读取的数据量越大，受益越大，数据量小但是计算密集度较大的场合，受益就相对较小
* 由于RDD的特性，Spark不适用那种异步细粒度更新状态的应用，例如web服务的存储或者是增量的web爬虫和索引。就是对于那种增量修改的应用模型不适合。

## 图

# 参考文献

<http://www.infoq.com/cn/articles/spark-core-rdd>

<http://www.csdn.net/article/2014-01-28/2818282-Spark-Streaming-big-data>