第四章　批处理系统Spark

徐 辰
cxu@dase.ecnu.edu.cn
華東師範大學

Spark简介

2

- Spark由美国加州伯克利大学（UC Berkeley）的AMP实验室于2009年开发
- 最初是基于内存计算的批处理系统，用于构建大型的、低延迟的数据分析应用程序
- 逐步发展成为内外存同时使用的批处理系统，并增加了Spark Streaming支持实时流计算，以及Structured Streaming支持批流融合

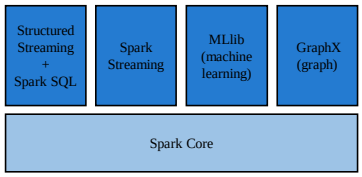
Spark简介

3

- 2013年Spark加入Apache孵化器项目后发展迅猛
- Spark在2014年打破了Hadoop保持的基准排序纪录
  - Spark/206个节点/23分钟/100TB数据
  - Hadoop/2000个节点/72分钟/100TB数据
  - Spark用十分之一的计算资源，获得了比Hadoop快3倍的速度

Spark软件栈

4



大纲

5

- 设计思想
  - MapReduce的局限性
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

MapReduce编程范型

6

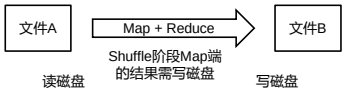
- 编程容易，不需要掌握分布式并行编程细节，很容易把程序运行在分布式系统上
- 但是基础算子太少：例如，如何实现join?

```
Class X {
    map() {              //map函数的实现
      …
    }

    reduce() {           //reduce函数的实现

    }

    main(){
       Job job = …       //定义分布式作业
       job.config =      //作业参数设置
    }
}
```

单个MapReduce作业

7

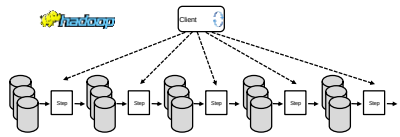
- Map端的结果要先写入本地磁盘，再由Reduce端来拉取



多个MapReduce作业

8

- 例子：迭代计算过程中每一迭代步结束时将结果写入HDFS，下一步将该结果再次从HDFS读出



MapReduce局限性

9

- 编程框架的表达能力有限，用户编程复杂
  - 仅map和reduce函数，无法直接用join等操作
- 单个作业中需要Shuffle的数据以阻塞方式传输，磁盘IO开销大、延迟高
  - Shuffle数据需要先写磁盘
- 多个作业之间衔接涉及IO开销，应用程序的延迟高
  - 特别是迭代计算，迭代中间结果的反复读写，使得整个应用程序的延迟非常高

大纲

10

- 设计思想
  - MapReduce的局限性
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

RDD概念

11

- Resilient Distributed Dataset
  - RDD是一个数据集(Dataset)：与MapReduce不同，Spark操作对象是抽象的数据集，而不是文件
  - RDD是分布式的(Distributed)：每个RDD可分成多个分区，每个分区就是一个数据集片段，一个RDD的不同分区可以存到集群中不同的节点上
  - RDD具有弹性(Resilient)：即具备可恢复的容错特性

大纲

12

- 设计思想
  - MapReduce的局限性
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

RDD操作算子

13

- 创建：从本地内存或外部数据源创建RDD，提供了数据输入的功能

操作算子	含义
parallelize(seq, [numSlices])	从内存集合创建RDD
textFile(path, [minPartitions])	读取HDFS兼容的文件系统下的文件来创建RDD
wholeTextFiles(path, [minPartitions])	读取HDFS兼容的文件系统下的文件夹中的所有文件来创建RDD
hadoopFile(path, inputFormatClass, keyClass, valueClass, [minPartitions])	读取HDFS兼容的文件系统下拥有任意inputFormat的文件来创建RDD

RDD操作算子

14

- 转换(Transformation)：描述RDD的转换逻辑，提供对RDD进行变换的功能

转换操作	含义
map(func)	对RDD中的每个记录都使用func进行转换，返回一个新的RDD
filter(func)	过滤出对RDD中的每个记录都使用func后返回值为true的记录
flatMap(func)	与map类似，但是对RDD中的每个记录可以映射成0个或多个新的记录
mapPartitions(func)	与map类似，但是mapPartitions中的func是对每个分区进行操作
union(otherRDD)	两个RDD取并集得到一个新的RDD
intersect(otherRDD)	两个RDD取交集得到一个新的RDD
groupByKey([numPartitions])	将[K, V]键值对按键分组，返回一个[K, Iterable<V>]对组成的新的RDD
reduceByKey(func, [numPartitions])	将键值对按键聚合，在每一个键的所有值上使用func，返回一个[K, V]对组成的新的RDD
sortByKey([ascending], [numPartitions])	将键值对按键进行排序，返回一个新的RDD
join(otherRDD, [numPartitions])	[K, V1]和[K, V2]分别属于两个RDD，返回一个[K, (V1, V2)]组成的RDD
cogroup(otherRDD, [numPartitions])	[K, V1]和[K, V2]分别属于两个RDD，返回一个[K, (Iterable<V1>, Iterable<V2>)]组成的RDD

RDD操作算子

15

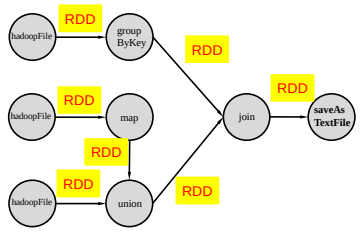
- 行动(Action)：标志转换结束，触发DAG生成

转换	含义
reduce(func)	对RDD中的记录按func聚合，这个func必须满足交换律和结合律
collect()	收集RDD中的所有记录到driver中，返回一个Array
count()	返回RDD中记录的个数
first()	返回RDD中的第一个记录
take(n)	返回RDD中的前n个记录
saveAsTextFile(path)	将RDD中的记录以文本文件的形式写入本地文件系统、HDFS或任何其他Hadoop支持的文件系统中的给定目录中
countByKey()	按key统计计数，返回一个由[K, Long]组成的Map
foreach(func)	对RDD中的每个记录都使用func

逻辑计算模型：Operator DAG

16

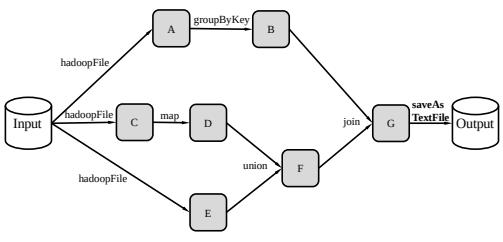
- 从算子操作的角度来描述计算的过程



逻辑计算模型：RDD Lineage

17

- 从RDD变换的角度来描述计算过程



RDD Lineage

18

- RDD Lineage（DAG拓扑结构）

非严格情况下两个概念可以混用

  - RDD读入外部数据源进行创建
  - RDD经过一系列的转换（Transformation）操作，每一次都会产生不同的RDD，供给下一个转换操作使用
  - 最后一个RDD经过"动作"操作进行转换，并输出到外部数据源
- Spark系统保留RDD Lineage的信息
  - 为什么?

RDD只读、不可变

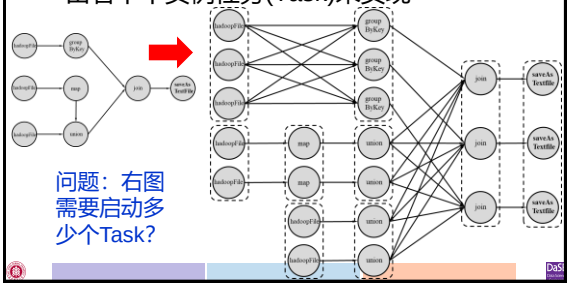
19

- RDD是只读的记录分区的集合
  - 本质上一个只读的对象集合
  - RDD经创建后，不能进行修改
- RDD不可变（Immutable）
  - 通过在其他RDD上执行确定的转换操作（如map、join和group by）而得到新的RDD，而不是改变原有的RDD
- 遵循了函数式编程的特性
  - 变量的值是不可变的

物理计算模型：Operator DAG

20

- 分布式架构中，DAG中的操作算子实际上由若干个实例任务(Task)来实现

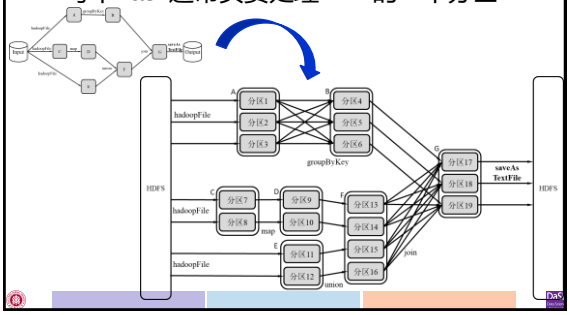


问题：右图需要启动多少个Task?

物理计算模型：RDD Lineage

21

- 每个Task通常负责处理RDD的一个分区



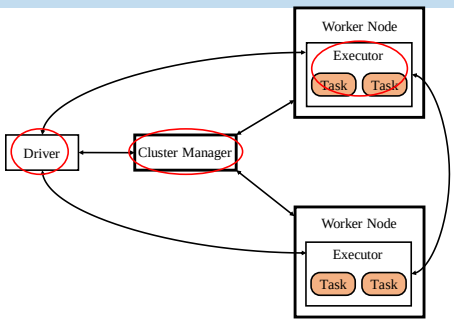
大纲

22

- 设计思想
- 体系架构
  - 架构图
  - 应用程序执行流程
- 工作原理
- 容错机制
- 编程示例

抽象架构图

23



Cluster Manager

24

- 集群管理器：负责管理整个系统的资源、监控工作节点
- 根据Spark部署方式的不同，
  - 在Standalone方式（即不使用Yarn或Mesos等其它资源管理系统）中，集群管理器包含Master和Worker
  - 在Yarn方式中集群管理器包括Resource Manager和Node Manager

Executor

25

- 执行器：负责任务执行
  - Executor是运行在工作节点上的一个进程，它启动若干个线程Task或线程组TaskSet来进行执行任务
  - 在Standalone部署方式下，Executor进程的名称为CoarseGrainedExecutorBackend
  - **问题：MapReduce中Task是线程还是进程?**

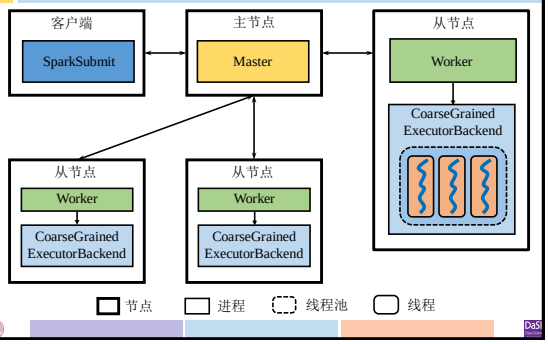
Driver

26

- 驱动器：负责启动应用程序的主方法并管理作业运行
- Spark的架构实现了资源管理和作业管理两大功能的分离
  - Cluster Manager负责集群资源管理
  - Driver负责作业管理

Standalone架构图

27



架构图比较

28

Standalone架构图	抽象架构图
SparkSubmit (Client)	/
Master	Cluster Manager
Worker	
CoarseGrained ExecutorBackend	Executor
物理节点（从节点）	Worker Node
?	Driver

Standalone中的Driver

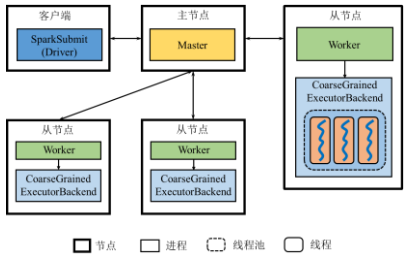
29

- 逻辑上，Driver独立于主节点、从节点以及客户端
- 但根据应用程序的Client或Cluster运行方式，Driver会以不同的形式存在
  - Client方式: Driver和客户端以同一个进程存在
  - Cluster方式: 系统将由某一Worker启动一个进程作为Driver
- 客户端提交应用程序时可以选择Client或Cluster方式

Standalone Client模式

30

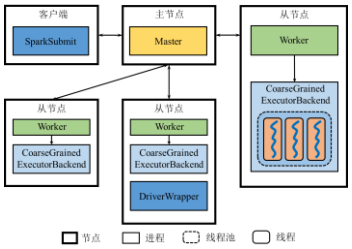
- Driver和客户端以同一个进程存在



Standalone Cluster模式

31

- 某一Worker启动一个名为DriverWrapper的进程作为Driver



Spark vs. MapReduce

32

- 二者架构比较

	MapReduce	Spark
系统进程	JobTracker	Master
	TaskTracker	Worker
	Child	CoarseGrained ExecutorBackend
工作线程	/	Task
任务代码	Task	
基础接口	map/reduce	RDD API

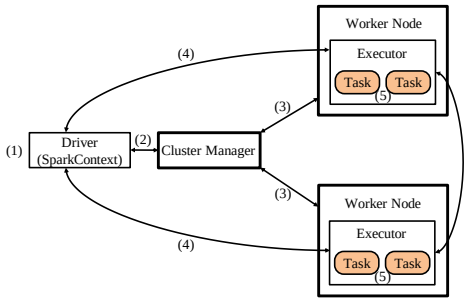
大纲

33

- 设计思想
- 体系架构
  - 架构图
  - 应用程序执行流程
- 工作原理
- 容错机制
- 编程示例

抽象执行流程

34



应用程序执行流程

35

1. 启动Driver，以Standalone模式为例
   - 如果使用Client部署方式，客户端端直接启动Driver，并向Master注册
   - 如果使用Cluster部署方式，客户端将应用程序提交给Master，由Master选择一个Worker启动Driver进程(DriverWrapper)
2. 构建基本运行环境，即由Driver创建SparkContext，向Cluster Manager进行资源申请，并由Driver进行任务分配和监控

应用程序执行流程（续）

36

3. Cluster Manager通知工作节点启动Executor进程，该进程内部以多线程方式运行任务
4. Executor进程向Driver注册
5. SparkContext构建DAG并进行任务划分，从而交给Executor进程中的线程来执行任务

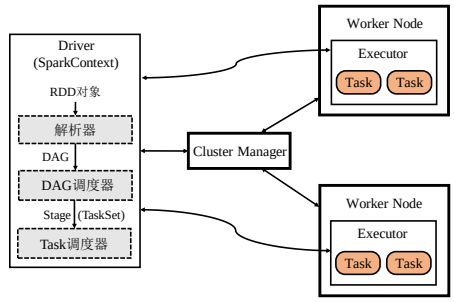
大纲

37

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

Driver内部工作过程

38



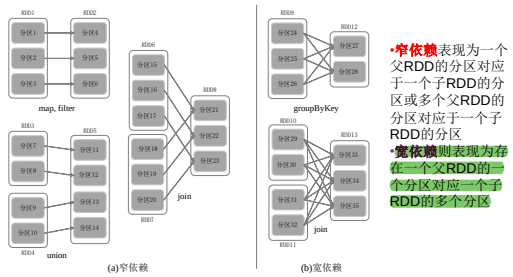
大纲

39

- 设计思想
- 体系架构
- 工作原理
  - DAG Stage划分
  - Stage内部数据传输
  - Stage之间数据传输
  - 应用与作业
- 容错机制
- 编程示例

RDD依赖关系

40



•窄依赖表现为一个父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

•宽依赖则表现为存在一个父RDD的一个分区对应一个子RDD的多个分区

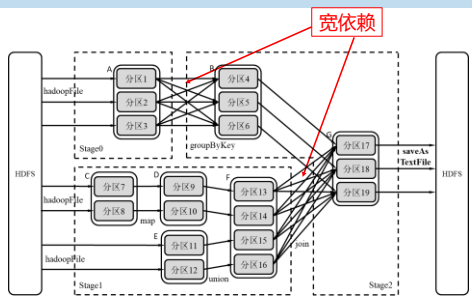
通过依赖关系进行Stage划分

41

- 分析各个RDD的偏序关系生成DAG，再通过分析各个RDD中的分区之间的依赖关系来决定如何划分Stage
- 具体划分方法：
  - 在DAG中进行反向解析，遇到宽依赖就断开
  - 遇到窄依赖就把当前的RDD加入到Stage中
  - 为什么将窄依赖尽可能划分在同一个Stage?

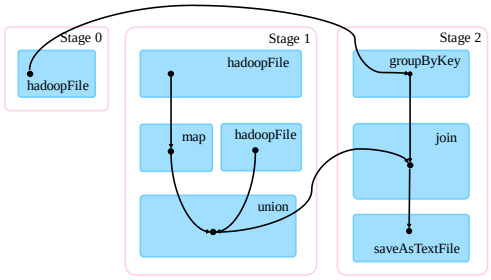
基于RDD Lineage的Stage划分

42



基于Operator DAG的Stage划分

43



Stage类型

44

- ShuffleMapStage
  - 输入/输出
    - 输入可以是从外部获取数据，也可以是另一个ShuffleMapStage的输出
    - 以Shuffle为输出，作为另一个Stage开始
  - 特点
    - 不是最终的Stage，在它之后还有其他Stage
    - 它的输出一定需要经过Shuffle过程，并作为后续Stage的输入
    - 在一个DAG里可能有该类型的Stage，也可能没有该类型Stage

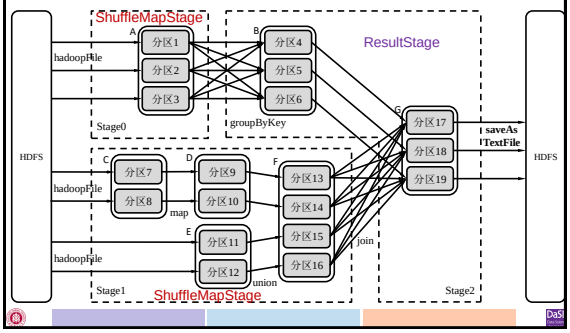
Stage类型

45

- ResultStage
  - 输入/输出
    - 其输入可以是从外部获取数据，也可以是另一个ShuffleMapStage的输出
    - 输出直接产生结果或存储
  - 特点
    - 最终的Stage
    - 在一个DAG里必定有该类型Stage
- 因此，一个DAG含有一个或多个Stage，其中至少含有一个ResultStage

ShuffleMapStage vs. ResultStage

46



大纲

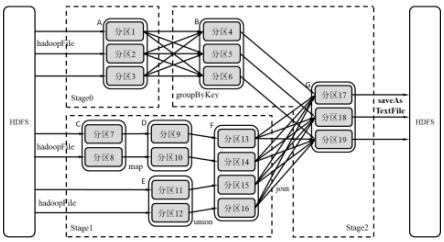
47

- 设计思想
- 体系架构
- 工作原理
  - DAG Stage划分
  - Stage内部数据传输
  - Stage之间数据传输
  - 应用与作业
- 容错机制

Stage内部的特点

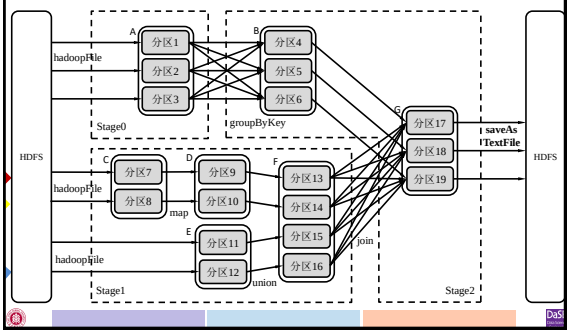
48

- 所有依赖关系都是窄依赖，可以实现pipeline方式进行数据传输



流水线（Pipeline）方式

49



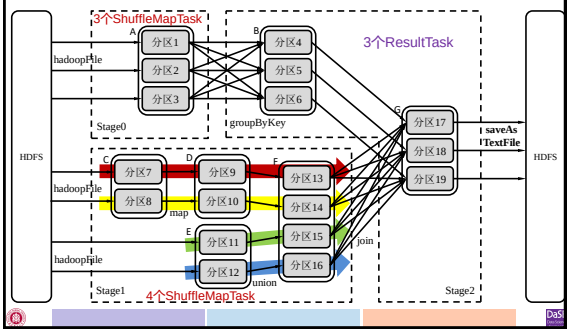
Spark Pipeline vs. MapReduce Shuffle

50

- 与MapReduce中Shuffle方式不同，流水线方式不要求物化前序算子的所有计算结果
  - 分区7通过map操作生成的分区9，并不需要物化分区9，而且可以不用等待分区8到分区10这个map操作的计算结束，继续进行union操作，得到分区13
  - 如果采用MapReduce中的Shuffle方式，那么意味着分区7、8经map计算得到分区9、10并将这两个分区进行物化之后，才可以进行union

ShuffleMapTask vs. ResultTask

51



大纲

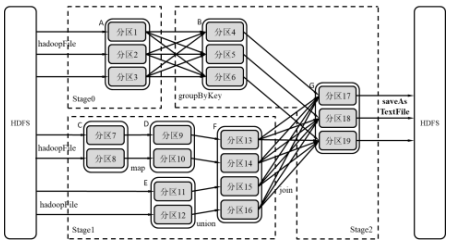
52

- 设计思想
- 体系架构
- 工作原理
  - DAG Stage划分
  - Stage内部数据传输
  - Stage之间数据传输
  - 应用、作业与任务
- 容错机制
- 编程示例

Stage之间的特点

53

- 所有依赖关系都是宽依赖，不可以实现pipeline方式进行数据传输，只能Shuffle



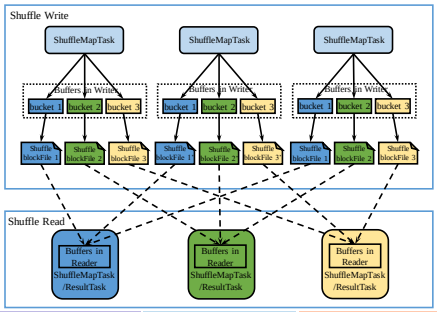
Stage之间的Shuffle

54

- Stage之间的数据传输需要进行Shuffle，该过程与MapReduce中的Shuffle类似
- Shuffle过程可能发生在两个ShuffleMapStage之间，或者ShuffleMapStage与ResultStage之间
- 从Task的层面来看，该过程表现为两组ShuffleMapTask之间，或一组ShuffleMapTask与一组ResultTask之间的数据传输

Spark Shuffle

55



Shuffle Write vs. Shuffle Read

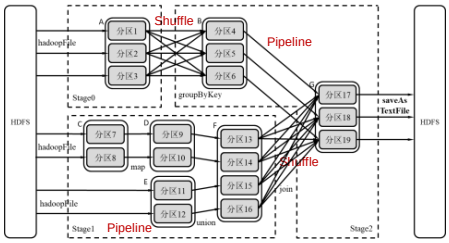
56

- 在Shuffle Write阶段，ShuffleMapTask需要将输出RDD的记录按照partition函数划分到相应的bucket当中并物化到本地磁盘形成ShuffleblockFile，之后才可以在Shuffle Read阶段被拉取
- 在Shuffle Read阶段，ShuffleMapTask或ResultTask根据partiton函数读取相应的ShuffleblockFile，存入buffer并进行继续后续的计算

Shuffle vs. Pipeline

57

- Stage之间：Shuffle
- Stage内部：Pipeline



大纲

58

- 设计思想
- 体系架构
- 工作原理
  - DAG Stage划分
  - Stage内部数据传输
  - Stage之间数据传输
  - 应用与作业
- 容错机制
- 编程示例

应用与作业

59

- Application：用户编写的Spark应用程序
- Job：一个Job包含多个RDD及作用于相应RDD转换操作，其中最后一个为action
- MapReduce vs. Spark
  - MapReduce中一个应用就是一个作业，
  - Spark中的一个应用可以由多个作业来构成

名称	MapReduce	Spark
应用	Job	Application
作业		Job/DAG

作业与任务

60

- Stage：一个Job会分为多组Task，每组Task被称为Stage，或者也被称为TaskSet
  - Job的基本调度单位
  - 代表了一组关联的、相互之间没有Shuffle依赖关系的任务组成的任务集
- Task：运行在Executor上的工作单元

应用、作业与任务

61

- 逻辑执行角度
  - 一个Application=一个或多个DAG
  - 一个DAG=一个或多个Stage
  - 一个Stage=若干窄依赖的**RDD操作**
- 物理执行角度
  - 一个Application=一个或多个Job
  - 一个Job=一个或多个TaskSet
  - 一个TaskSet=多个没有Shuffle关系的**Task**

Job=DAG

Stage=TaskSet

逻辑概念与物理概念

62

	逻辑	物理
Application	DAG	Job
	Stage	TaskSet
	Operator	Task

大纲

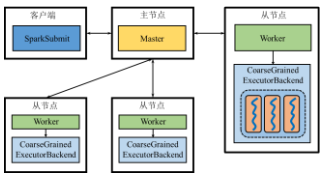
63

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

故障类型

64

- Master故障：ZooKeeper配置多个Master
- Worker故障
- Executor故障
- Driver故障：重启



大纲

65

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - RDD持久化
  - 故障恢复
  - 检查点
- 编程示例

RDD存储机制

66

- 由于计算过程中会不断地产生新的RDD，所以系统不能将所有的RDD都存在内存
- 一旦达到相应存储空间的阈值，Spark会使用置换算法（例如，LRU）将部分RDD的内存空间腾出
- 如果不做任何声明，这些RDD会被直接丢弃。但是，某些RDD在后续很可能会被再次使用

RDD存储机制

67

- RDD提供的持久化（缓存）接口
  - persist(StorageLevel)
    - 接受StorageLevel类型参数，可配置各种级别
    - 持久化后的RDD将会被保留在工作节点的中被后面的行动操作重复使用
  - cache()
    - 相当于persist(MEMORY_ONLY)

StorageLevel

68

- MEMORY_ONLY
- MEMORY_AND_DISK
- MEMORY_ONLY_SER
- MEMORY_AND_DISK_SER
- DISK_ONLY
- MEMORY_ONLY_2, MEMORY_AND_DISK_2

StorageLevel

69

- MEMORY_ONLY：
  - 在JVM中缓存Java的对象。如果内存不足，直接丢弃某些partition
- MEMORY_AND_DISK：
  - 在JVM中缓存Java的对象。如果内存不足，则将某些partitions写入到磁盘中
- MEMORY_ONLY_SER：
  - 在内存为每个partition存储一个byte数组，数组内容为当前partition中Java对象的序列化结果

StorageLevel

70

- MEMORY_AND_DISK_SER：
  - 与MEMORY_AND_DISK类似，但每个分区存储的是Java对象序列化后组成的byte数组
- DISK_ONLY:
  - 将每个分区的数据序列化到磁盘中

StorageLevel

71

- MEMORY_ONLY_2：
  - 与MEMORY_ONLY相同，但是每个分区备份到两台机器上
- MEMORY_AND_DISK_2:
  - 与MEMORY_AND_DISK相同，但是每个分区备份到两台机器上

大纲

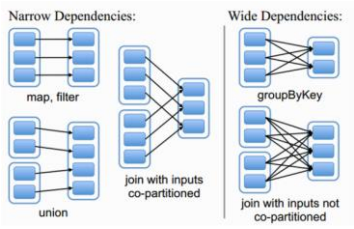
72

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - RDD持久化
  - 故障恢复
  - 检查点
- 编程示例

Lineage机制

73

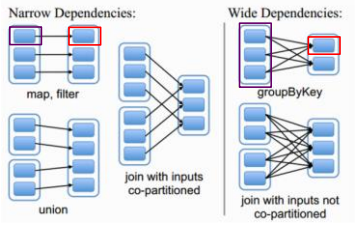
- RDD dependencies



Lineage机制

74

- 红色部分丢失，需重算紫色部分



基于RDD Lineage恢复

75

- 利用RDD Lineage的故障恢复
  - 重新计算丢失分区
  - 重算过程在不同节点之间可以并行
- 与数据库恢复的比较
  - RDD Lineage：记录粗粒度的操作
  - 数据复制或者日志：记录细粒度的操作

大纲

76

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - RDD持久化
  - 故障恢复
  - 检查点
- 编程示例

检查点机制

77

- 前述机制的不足之处
  - Lineage可能非常长
  - RDD持久化机制保存到集群内机器的磁盘，并不完全可靠
- 检查点机制将RDD写入外部可靠的（本身具有容错机制）分布式文件系统，例如HDFS
  - 在实现层面，写检查点的过程是一个独立的作业，在用户作业结束后运行

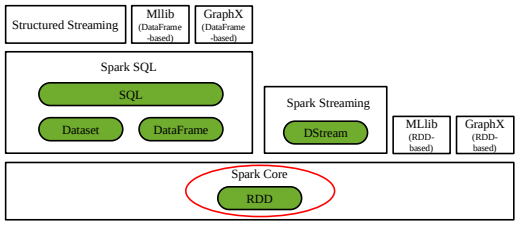
大纲

78

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

Spark API

79



RDD API编程框架

80

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

object CustomApp {
  def run(args: Array[String]): Unit = {
    /* 步骤1：通过SparkConf设置配置信息，并创建SparkContext */
    val conf = new SparkConf
      .setAppName("appName")
      .setMaster("local") // 仅用于本地进行调试，如在集群中运行则删除该行
      ......
    val sc = new SparkContext(conf)

    /* 步骤2：按应用逻辑使用操作算子编写DAG，其中包括RDD的创建、转换和行动等 */
    ......

    /* 步骤3：关闭SparkContext */
    sc.stop()
  }

  def main(args: Array[String]): Unit = {
    run(args)
  }
}
```

大纲

81

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 关系表自然连接及其优化
  - 网页链接排名
  - K均值聚类
  - 检查点

词频统计

82

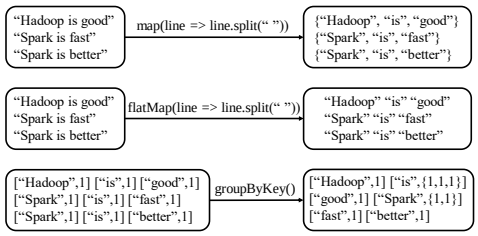
- 输入：一个包含大量单词的文本文件
- 输出：
  - 文件中每个单词及其出现次数（频数）
  - 每个单词和其频数占一行，单词和频数之间有间隔

输入	输出
An An My Me An An My He My My An My	An 5 He 1 Me 1 My 5

转换操作算子回顾

83

- map、flatMap、groupByKey



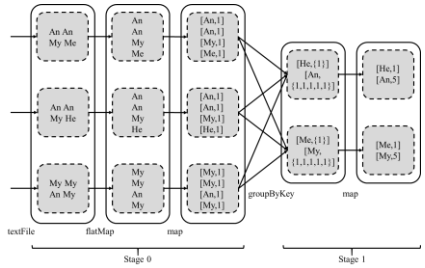
解决方案

84

- flatMap
  - 将每行文本按分隔符拆分成单词
- map
  - 将每个单词映射为[单词, 1]键值对
- groupByKey
  - 将[单词, 1]键值对按单词进行分组
- map
  - 对组内每个单词的频数进行求和

RDD Lineage

85



Spark提供比MapReduce更丰富的算子，更易于编程。

代码

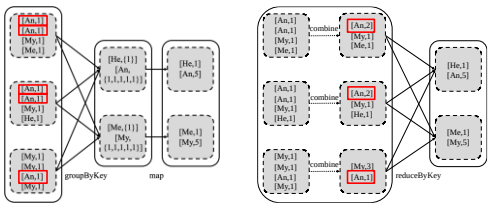
86

```
// 读入文本数据，创建名为lines的RDD
val lines = sc.textFile(filePath)

// 将lines中的每一个文本行按空格分割成单个单词
val words = lines.flatMap { line => line.split(" ") }

// 将每个单词的频数设置为1，即将每个单词映射为[单词, 1]
val pairs = words.map { word => (word, 1) }

// 按单词聚合，并对相同单词的频数使用sum进行累计
val wordCounts = pairs.groupByKey().map{ t => (t._1, t._2.sum) }
```

优化方案

87

- 减少Shuffle数据量



(a) 优化前　　(b) 优化后

优化方案的代码

88

```
// 读入文本数据，创建名为lines的RDD
val lines = sc.textFile(filePath)

// 将lines中的每一个文本行按空格分割成单个单词
val words = lines.flatMap { line => line.split(" ") }

// 将每个单词的频数设置为1，即将每个单词映射为[单词, 1]
val pairs = words.map { word => (word, 1) }

// 按单词聚合，并对相同单词的频数进行累计
val wordCounts = pairs.reduceByKey(_+_)
```

MapReduce要求用户自定义combine方法，
而Spark内置combine机制，简化了用户编程。

大纲

89

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 关系表自然连接及其优化
  - 网页链接排名
  - K均值聚类
  - 检查点

关系表的自然连接

90

- 输入：两个CSV文件，分别保存雇员表和部门表
- 输出：雇员表和部门表的自然连接结果

雇员表

Name	EmpId	DeptName
Harry	3415	会计
Sally	2241	销售
George	3401	会计
Harriet	2202	销售
Bart	6077	会计
Elise	9263	销售
Gemma	3870	会计
Tyler	6236	销售
Camden	4527	销售
Sofia	2035	会计

部门表

DeptName	Manager
会计	George
销售	Harriet

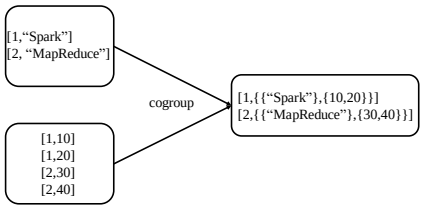
雇员表⋈部门表

Name	EmpId	DeptName	Manager
Sofia	2035	会计	George
Gemma	3870	会计	George
Bart	6077	会计	George
George	3401	会计	George
Harry	3415	会计	George
Camden	4527	销售	Harriet
Tyler	6236	销售	Harriet
Elise	9263	销售	Harriet
Harriet	2202	销售	Harriet
Sally	2241	销售	Harriet

转换操作算子回顾

91

- cogroup



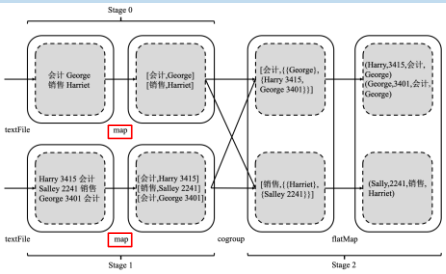
解决方案

92

- map
  - 将来自部门表的每个元组映射为[DeptName, Manager]键值对
  - 将来自雇员表的每个元组映射为[DeptName, Name EmpId]键值对
- cogroup
  - 将雇员表和部门表按DeptName进行协同分组
- flatMap
  - 对组内具有相同DeptName的元组两两组合，得到自然连接结果

RDD Lineage

93



MapReduce要求用户显示标记每个元组来自哪张表，
而Spark可由2个算子分别处理2张关系表得到不同的RDD。

代码

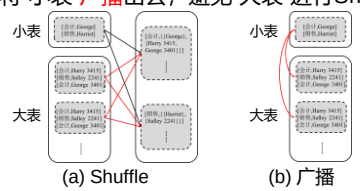
94

```
// 读入部门表
val departmentsRDD = sc.textFile("departmentFilePath")
  .map(line => {
    // 按制表符分割文本行，将每行文本映射为[DeptName, Manager]键值对
    val tokens = line.split("\t")
    (tokens(0), tokens(1))
  })

// 读入雇员表
val employeesRDD = sc.textFile("employeeFilePath")
  .map(line => {
    // 按制表符分割文本行，将每行文本映射为[DeptName, Name EmpId]键值对
    val tokens = line.split("\t")
    (tokens(2), (tokens(0), tokens(1)))
  })

// 用cogroup算子对雇员表和部门表按DeptName聚合
employeesRDD.cogroup(departmentsRDD)
  // 对[DeptName, {{Name EmpId}, {Manager}}]进行连接操作
  .flatMap( tuple =>
    // 返回连接结果(Name, EmpId, DeptName, Manager)
    for (v <- tuple._2._1.iterator; w <- tuple._2._2.iterator) yield (v._1, v._2,
        tuple._1, w)
  )
```

大表⋈小表

95

- 假如部门表比较小，雇员表非常大
  - 雇员表的Shuffle开销非常大
- 优化方案
  - 将“小表”广播出去，避免“大表”进行Shuffle



(a) Shuffle (b) 广播

优化方案的代码

96

```
// 读入部门表
val departmentsMap = sc.textFile("departmentFilePath")
  // 按制表符解析为[DeptName, Manager]键值对后collectAsMap到Driver中
  .map(line => {
    val tokens = line.split("\t")
    (tokens(0), tokens(1))
  }).collectAsMap()

// 广播部门表
val departmentsBroadCast = sc.broadcast(departmentsMap)

// 读入雇员表
val employeesRDD = sc.textFile("employeeFilePath")
  .map(line => {
    // 按制表符解析为(DeptName, Name, EmpId)元组
    val tokens = line.split("\t")
    (tokens(2), tokens(0), tokens(1))
  })

// 在map转换操作中对雇员表与广播的部门表做自然连接
employeesRDD.map(r => {
  // 获取广播变量部门表的值
  val departmentsBroadCastValue = departmentsBroadCast.value
  if (departmentsBroadCastValue.contains(r._1)) {
    // 获取departmentsBroadCastValue中对应key的value
    val left = departmentsBroadCastValue.get(r._1).get
    // 返回连接结果(Name, EmpId, DeptName, Manager)
    (r._2, r._3, r._1, left)
  } else {
    null
  }
})
```

大纲

97

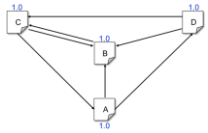
- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 关系表自然连接及其优化
  - 网页链接排名
  - K均值聚类
  - 检查点

网页链接排名

98

- 输入：保存在文本文件中，一行为一项网页信息
  - 网页信息：(网页名 网页排名值 (出站链接 出站链接的权重...))
- 输出：网页名及其排名值

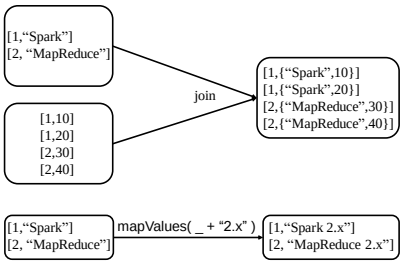
输入	输出
A 1.0 B 1.0 D 1.0 B 1.0 C 1.0 C 1.0 A 1.0 B 1.0 D 1.0 B 1.0 C 1.0	A 0.21436 B 0.36332 C 0.40833 D 0.13027



转换操作算子回顾

99

- join、mapValues



解决方案

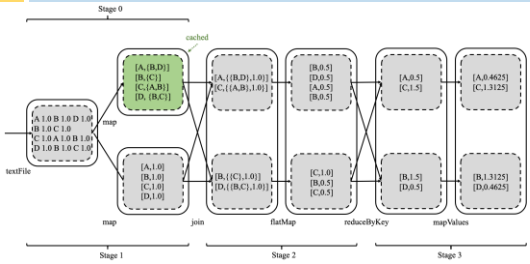
100

- 算法执行过程
  - 初始时，每个网页的排名值为1
  - 每一步迭代，网页p_j对其链向的网页的排名值贡献$PR(p_j)/L(p_j)$　　join、flatMap
  - 每个网页p_i累加所有链向p_i的网页$M(p_i)$的贡献值，然后更新排名值为$0.15/N + 0.85 * \sum_{p_{j \in M(p_i)}} \frac{PR(p_j)}{L(p_j)}$　　reduceByKey、mapValues
  - 最终收敛

整个迭代是一个Spark应用

RDD Lineage

101



MapReduce迭代间数据的传递需要借助读写HDFS来完成，而Spark迭代间的RDD数据可以直接驻留在内存中。

代码

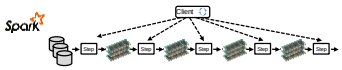
102

```
// 读取输入文本数据
val text = sc.textFile("inputFilePath")
// 将文本数据转换成[网页，{链接列表}]键值对，并持久化到内存
val links = text.map(...).cache()
// 初始化每个页面的排名值，得到[网页，排名值]键值对
var ranks = text.map(...)

// 执行iterateNum次迭代计算
for (iter <- 1 to iterateNum) {
  val contributions = links
    // 将links和ranks做join，得到[网页，({链接列表}，排名值)]
    .join(ranks)
    // 计算出每个网页对其每个链接网页的贡献值
    .flatMap {
      case (page, (links, rank)) =>
        // 网页排名值除以链接总数
        links.map(dest => (dest, rank / links.size))
    }

  ranks = contributions
    // 聚合对相同网页的贡献值，求和得到对每个网页的总贡献值
    .reduceByKey(_+_)
    // 根据公式计算得到每个网页的新排名值
    .mapValues(v => (1 - factor) * 1.0 / N + factor * v)
}
```

迭代比较

103

MapReduce每一迭代步骤(Step)结束时将结果写入HDFS，下一迭代步骤将该结果再次从HDFS读出。
迭代间反复读写HDFS开销大。



Spark每一迭代步骤(Step)结束时得到的RDD可以驻留在内存中直接作为下一迭代步骤的输入。
避免了迭代间反复读写磁盘。

大纲

104

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 关系表自然连接及其优化
  - 网页链接排名
  - K均值聚类
  - 检查点

K均值聚类

105

- 输入：两个文本文件，分别保存数据集和聚类中心集
  - 数据集：每行为一个二维数据点及其类别标签
  - 聚类中心集：每行为一个二维数据点
- 输出：数据点及其类别标签

数据集	聚类中心集	聚类结果
0,0 -1 1,2 -1 3,1 -1 8,8 -1 9,10 -1 10,7 -1	1,2 3,1	0,0 1.0 1,2 1.0 10,7 2.0 3,1 1.0 8,8 2.0 9,10 2.0

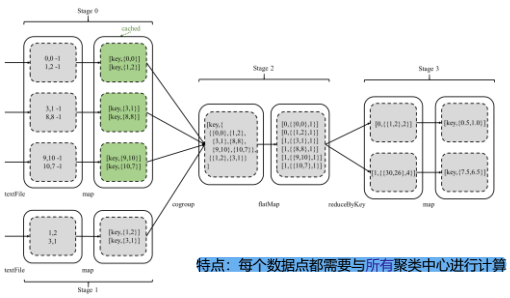
Naïve解决方案

106

- 算法执行过程
  1. 设定聚类中心数k。例如，k=2
  2. 选取聚类中心
  3. 寻找每个数据点距离最近的聚类中心 — cogroup、flatMap
  4. 计算同属一个聚类中心的数据点的新的聚类中心 — reduceByKey、map
  5. 重复3-4直到满足迭代终止条件 — 整个迭代是一个Spark应用

Naïve解决方案的RDD Lineage

107



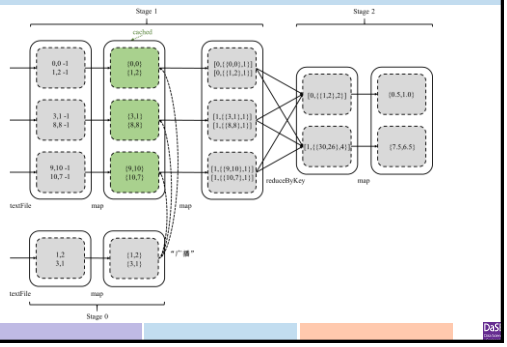
优化方案

108

- 算法执行过程
  1. 设定聚类中心数k。例如，k=2
  2. 选取聚类中心 — collect
  3. 寻找每个数据点距离最近的聚类中心 — map
  4. 计算同属一个聚类中心的数据点的新的聚类中心 — reduceByKey、map collect
  5. 重复3-4直到满足迭代终止条件 — 整个迭代是一个Spark应用

优化方案的RDD Lineage

109



优化方案的代码

110

```
// 从数据源读入数据点
val points = sc.textFile("dataFilePath")
  // 按逗号解析每行数据，转换为Int类型，并持久化到内存
  .map(_.split("\t")(0).split(",").map(_.toInt)).cache()

// 获取设置的初始中心点
val kPoints = sc.textFile("centersFilePath")
  // 按逗号解析每行数据，转换为Int类型，并收集到Driver中
  .map(_.split(",").map(_.toDouble)).collect()

// 执行iterateNum次迭代计算
for (iter <- 1 to iterateNum - 1) {
  val closest = points.map{p => {
    // 计算距离最近的聚类中心
    (closestPoint(p, kPoints), (p, 1))
  }}

  // 按类别号标识聚合，并计算新的聚类中心
  val newPoints = closest.reduceByKey{
    (t1, t2) => {
      // 计算两个点的和，并累加数据点个数
      (addPoints(t1._1, t2._1), t1._2 + t2._2)
    }
  }.map{
    case (index,(point,n)) => {
      val newPoint = ArrayBuffer[Double]()
      for (i <- point.indices){
        // 每个维度的和值除以数据点个数得到每个维度的均值
        newPoint += point(i).toDouble / n
      }
      newPoint.toArray
    }
  }.collect()

  // 将旧的聚类中心替换为新的聚类中心
  for (i <- kPoints.indices) {
    kPoints(i) = newPoints(i)
  }

  // 如果是最后一次迭代，则输出聚类结果
  if (iter == iterateNum - 1) {
    closest.foreach(...) // 输出最终的聚类结果
  }
}
```

大纲

111

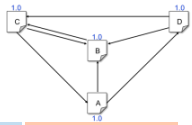
- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 关系表自然连接及其优化
  - 网页链接排名
  - K均值聚类
  - 检查点

检查点

112

- 输入：保存在文本文件中，一行为一项网页信息
  - 网页信息：(网页名 网页排名值 (出站链接 出站链接的权重...))
- 输出：网页名及其排名值
- 要求：每5轮迭代存一个网页排名的检查点

输入	输出
A 1.0 B 1.0 D 1.0 B 1.0 C 1.0 C 1.0 A 1.0 B 1.0 D 1.0 B 1.0 C 1.0	A 0.21436 B 0.36332 C 0.40833 D 0.13027



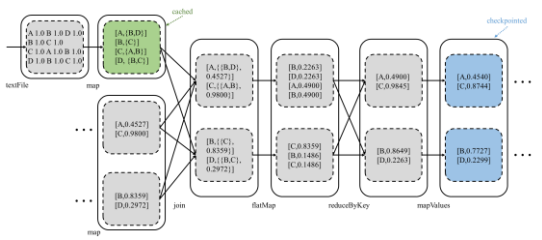
解决方案

113

- 在每轮迭代中增加一个行动操作
  - 写检查点操作发生在一个作业结束之后
- 迭代次数达到5的倍数时写检查点
  - 每隔5轮迭代对网页排名写一次检查点
- 写检查点前缓存网页排名RDD
  - 写检查点是一个独立的作业，写检查点前先缓存待写入检查点的RDD以避免重复计算

RDD Lineage

114



代码

115

```
sc.setCheckpointDir("checkpointPath") // 设置检查点路径
...
// 执行iterateNum次迭代计算
for (iter <- 1 to iterateNum) {
  val contributions = links
    // 将links和ranks做join，得到(网页，({链接列表}，排名值))
    .join(ranks)
    // 计算出每个网页对其每个链接网页的贡献值
    .flatMap {
      case (page, (links, rank)) =>
        // 网页指名假设出链接总数
        links.map(dest => (dest, rank / links.size))
    }

  ranks = contributions
    // 聚合对相同网页的贡献值，求和得到对每个网页的总贡献值
    .reduceByKey(_+_)
    // 根据公式计算得到每个网页的新排名值
    .mapValues(v => (1 - factor) * 1.0 / N + factor * v)

  if (iter % 5 == 0) { // 每5次迭代保存一次检查点
    // 缓存将写入检查点的RDD
    ranks.cache()
    // 调用checkpoint方法设置检查点
    ranks.checkpoint()
  }

  // 对排名值保留前5位，并打印每轮迭代的网页排名值
  ranks.foreach(t => println(t._1 + " " + t._2.formatted("%.5f")))
}
```

课后阅读

116

- 论文
  - Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark : Cluster Computing with Working Sets. In HotCloud (pp. 1–7).
  - Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., Mccauley, M., … Stoica, I. (2012). Resilient Distributed Datasets : A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In NSDI (pp. 15–28).

本章小结

117

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

谢谢！Q&A