# 一 Hadoop之MapReduce

## 海量数据分流处理技术思想（负载均衡）

### 1传统hash

问题：如何将大数据、流量均匀分布到N台服务器

方案：找到合理的key，hash（key）尽量分布均匀

Hash（key）mod N==0 分到第0台机器

Hash（key）mod N==1 分到第1台机器

Hash（key）mod N==N-1 分到第N-1台机器

传统hash算法存在致命问题，如果某一台机器宕机，那么应该落在该机器的请求就无法得到正确的处理，这时需要将宕掉的服务器使用算法去除，此时候会有(N-1)的服务器的缓存数据需要重新进行计算；如果新增一台机器，会有(N+1)的服务器的缓存数据需要进行重新计算。对于系统而言，这通常是不可接受的颠簸（因为这意味着大量缓存的失效或者数据需要转移）

### 2随机划分

假设有N个节点，随机划分即根据key随机生成一个数，这个数代表节点标号N

### 3一致性hash

一致性hash算法可以有效地解决分布式存储结构下动态增加和删除节点所带来的问题



首先将hash算法的值域映射成一个环形，如0~2^32-1的整数（每个整数代表一个桶空间），每个缓存key取hash转换成整数，映射到该环上，对服务器节点采用相同的hash算法（如根据ip或主机名或网卡等）转为整数映射到环上，key对应的顺时针方向最近的节点就是所属节点

增加节点

（如node4）时，有一小部分key的归属会受到影响，如key2原本属于node2，增加node4后，现在属于node4，原来node2上key2的缓存失效，缓存会在node4上重新生成

删除节点

（如node3），同样有一小部分的key归属会受到影响，如key4原本属于node3，node3被删除后，key4落到node1节点，在node3上的所有缓存key失效（这里只有key4），这些key缓存重新在node1上生成

### 4一致性hash+虚拟节点

一致性hash算法可能存在key分布不均匀的情况，如下图，所有的key都归属以一个节点node2



为了优化这种节点太少导致key分布不均匀的情况，一致性hash引入了虚拟节点的概念。

虚拟节点就是基于原来的物理节点映射成多个子节点，最后把所有子节点映射到环形空间



### 5一致性hash算法实现

## （二）MapReduce

[link\MapReduce过程详解(基于hadoop2.x架构) - py小杰 - 博客园.mhtml](link/MapReduce过程详解(基于hadoop2.x架构) - py小杰 - 博客园.mhtml)





# 二 hive

## hive简介

Hive是一个sql解析引擎，将sql转译成MR job，然后在Hadoop平台运行

### 特点：

#### 1 Hive的元数据依赖于关系型数据库，hive本身不存储数据，数据存储在hdfs中（即文件），完全依赖Hadoop和MapReduce

#### 2 hive不支持对数据的修改和删除

#### Hive元数据之各表简介：

[link\Hive 元数据表结构详解\_hive,元数据,表结构\_阳光Xiao伙的博客-CSDN博客.mhtml](link/Hive 元数据表结构详解_hive,元数据,表结构_阳光Xiao伙的博客-CSDN博客.mhtml)

##### VERSION：存储hive版本，该表中数据只有一条，如果存在多条，会造成hive启动不起来

##### Hive数据库相关的元数据表(DBS、DATABASE\_PARAMS)

DBS ：该表存储Hive中所有数据库的基本信息

DATABASE\_PARAMS ：该表存储数据库的相关参数，在CREATE DATABASE时候用WITH DBPROPERTIES(property\_name=property\_value, …)指定的参数

##### 3）Hive表和视图相关的元数据表

主要有TBLS（该表中存储Hive表，视图，索引表的基本信息）

TABLE\_PARAMS（该表存储表/视图的属性信息）

TBL\_PRIVS（该表存储表/视图的授权信息）

这三张表通过TBL\_ID关联

##### 4）Hive文件存储信息相关的元数据表

主要涉及SDS、SD\_PARAMS、SERDES、SERDE\_PARAMS，由于HDFS支持的文件格式很多，而建Hive表时候也可以指定各种文件格式，Hive在将HQL解析成MapReduce时候，需要知道去哪里，使用哪种格式去读写HDFS文件，而这些信息就保存在这几张表中

##### 5）Hive表字段相关的元数据表

主要涉及COLUMNS\_V2，该表存储表对应的字段信息

##### 6）Hive表分分区相关的元数据表

主要涉及PARTITIONS、PARTITION\_KEYS、PARTITION\_KEY\_VALS、PARTITION\_PARAMS

## Hive的sql与传统sql区别





## Hive体系架构



## Hive数据管理

### 内部表和外部表

未被external修饰的是内部表（managed table），被external修饰的为外部表（external table）；

区别：

内部表数据由Hive自身管理，外部表数据由HDFS管理；

内部表数据存储的位置是hive.metastore.warehouse.dir（默认：/user/hive/warehouse），外部表数据的存储位置由自己制定（如果没有LOCATION，Hive将在HDFS上的/user/hive/warehouse文件夹下以外部表的表名创建一个文件夹，并将属于这个表的数据存放在这里）；

删除内部表会直接删除元数据（metadata）及存储数据；删除外部表仅仅会删除元数据，HDFS上的文件并不会被删除；

对内部表的修改会将修改直接同步给元数据，而对外部表的表结构和分区进行修改，则需要修复（MSCK REPAIR TABLE table\_name;）

### Partition（分区）

### Bucket（分桶）

作用：方便抽样，提高join查询效率

### 数据类型

### Hive的窗口函数

[link\Hive 之 分析窗口函数 - happy19870612's blog - CSDN博客.mhtml](link/Hive 之 分析窗口函数 - happy19870612's blog - CSDN博客.mhtml)

#### 5.1分析函数定义

分析函数又被称为窗口函数。在联机分析(OLAP)的时候，传统是SQL可能需要进行多个自连接，从而导致查询新能很差。又或者有时候需要多维分析需要在不同层级进行聚合，并且聚合后数据行和密钥聚合的数据行都必须返回。传统是SQL都有一定的局限性。分析函数很好的可以解决这些问题，可以不使用自联结就能在一行中取出聚合和非聚合的行

#### 5.2分析函数组成

function(argument1, argument2…)over([partition-by-cluase][order-by-clasue][windowing-clause])

分析函数有三个部分组成：分区子句，排序子句，以及窗口子句。

分区子句： 其实可以理解为根据什么分组

排序子句：对每一个分区的结果集排序

窗口子句：

[ROWS|RANGE] BETWEEN <开始表达式> AND <结束表达式>

<开始表达式>:[UNBOUNDED PRECEDING| CURRENT ROW|

nRECEDING | n FOLLOWING]

<结束表达式>:[UNBOUNDED FOLLOWING|CURRENT ROW|

nRECEDING | n FOLLOWING]

UNBOUNDEDPRECEDING：针对当前分区里面的当前行而言的前面所有行

UNBOUNDEDFOLLOWING：针对当前分区里面的当前行而言的后面所有行

CURRENTROW： 当前行

nRECEDING： 针对当前行而言，前n行

nFOLLOWING：针对当前行而言，后n行

## hive的优化

[link\Hive优化（整理版） - 牧梦者 - 博客园.mhtml](link/Hive优化（整理版） - 牧梦者 - 博客园.mhtml)

1 当任务有大量小文件时，考虑合并小文件，减少map和reduce数量

2 当input的文件都很大，任务逻辑复杂，map执行非常慢的时候，可以考虑增加Map和reduce数

# 三 Hadoop1和2的区别

## slot的含义

1）slot就是槽的意思，是一个资源单位，只有给task分配了一个slot之后，这个task才可以运行。slot分两种，map slot沪蓉reduce slot。另外，slot是一个逻辑概念，一个数据节点的slots数量既不是CPU的核数，也不是memory chip，一个节点的slot数量用来表示此节点的资源容量或是计算计算能力的大小，也就是说slot是hadoop的资源单位。

2）系统中map slots总数与reduce slots总数的计算方式如下:

map slots 总数=集群节点数\*mapred.tasktracker.map.tasks.maximum（默认是2）;

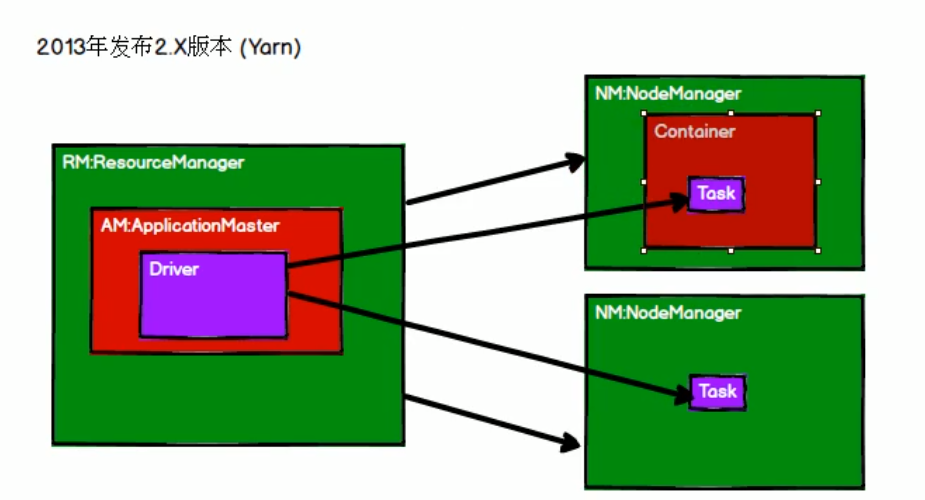
reduce slots 总数=集群节点数\*mapred.tasktracker.reduce.tasks.maximum（默认是2）;

## Yarn解决的问题

1 降低运维成本和数据共享成本

2 减小jobTracker（2.0的RM）的负担，1.x中jobTracker既是资源调度器，又是任务调度器。

2.x中，资源调度器分为RM和NM，任务执行为Driver和task，AM和Container 负责串连资源调度器RM NM与Dirver task之间的关系，如task资源不够，向AM申请，AM再向RM协商资源（container）



3 使得多种计算框架可以运行在一个集群中

4 资源表示成内存量，解决了1.x中的，map slot和reduce slot分开造成的资源闲置

## hdfs2.0解决的问题

1 NameNode HA解决了Hadoop1.X的单点故障问题，2.X中，NN可主备切换

2 NameNode Federation（联盟） 解决了NameNode的横向扩展问题

3 snapshot快照防止用户误操作和容灾恢复

4 hdfs cache缓存可以防止频繁使用的数据从内存中清除

5 hdfs ACL使得权限控制可以对单个用户操作

6 异构层级存储结构 支持一个集群中有多种不同的存储介质

# 四 yarn

[link\hadoop之yarn详解（基础架构篇） - 一寸HUI - 博客园.mhtml](link/hadoop之yarn详解（基础架构篇） - 一寸HUI - 博客园.mhtml)

## 简介

Yarn（Yet Another Resource Negotiator的缩写）是hadoop集群资源管理器系统

MapReduce1中，具有如下局限性：

1、扩展性差：jobtracker兼顾资源管理和作业控制跟踪功能跟踪任务，启动失败或迟缓的任务，记录任务的执行状态，维护计数器），压力大，成为系统的瓶颈

2、可靠性差：采用了master/slave结构，master容易单点故障

3、资源利用率低：基于槽位的资源分配模型，槽位是一种粗粒度的资源划分单位，通常一个任务不会用完一个槽位的资源，hadoop1分为map slot和reduce slot，而它们之间资源不共享，造成一些资源空闲。

4、不支持多框架：不支持多种计算框架并行

yarn很好解决了MapReduce1中的局限性：yarn基本思想；一个全局的资源管理器resourcemanager和与每个应用对用的ApplicationMaster，Resourcemanager和NodeManager组成全新的通用系统，以分布式的方式管理应用程序

## （二）特点：

### 1、支持非mapreduce应用的需求

### 2、可扩展性

### 3、提高资源使用率

### 4、用户敏捷性

### 5、可以通过搭建为高可用

## （三）各组件概念



### 1 Resourcemanager

ResourceManager 拥有系统所有资源分配的决定权，负责集群中所有应用程序的资源分配，拥有集群资源主要、全局视图。因此为用户提供公平的，基于容量的，本地化资源调度。根据程序的需求，调度优先级以及可用资源情况，动态分配特定节点运行应用程序。它与每个节点上的NodeManager和每一个应用程序的ApplicationMaster协调工作

### 2 NodeManager

NodeManager是yarn节点的一个“工作进程”代理，管理hadoop集群中独立的计算节点，主要负责与ResourceManager通信，负责启动和管理应用程序的container的生命周期，监控它们的资源使用情况（cpu和内存），跟踪节点的监控状态，管理日志等。并报告给RM

主要职责：

1、接收ResourceManager的请求，分配Container给应用的某个任务

2、和ResourceManager交换信息以确保整个集群平稳运行。ResourceManager就是通过收集每个NodeManager的报告信息来追踪整个集群健康状态的，而NodeManager负责监控自身的健康状态。

3、管理每个Container的生命周期

4、管理每个节点上的日志

5、执行Yarn上面应用的一些额外的服务，比如MapReduce的shuffle过程

### 3 ApplicationMaster

ApplicationMaster负责与scheduler协商合适的container，跟踪应用程序的状态，以及监控它们的进度，ApplicationMaster是协调集群中应用程序执行的进程。每个应用程序都有自己的ApplicationMaster，负责与ResourceManager协商资源（container）和NodeManager协同工作来执行和监控任务

### 4 container

理论概念，一个Container就是一组分配的系统资源，现阶段只包含两种系统资源（之后可能会增加磁盘、网络、CPU等资源），由NodeManager监控，Resourcemanager调度。

每一个应用程序从ApplicationMaster开始，它本身就是一个container（第0个），一旦启动，ApplicationMaster就会根据任务需求与Resourcemanager协商更多的container，在运行过程中，可以动态释放和申请container

## yarn作业调度流程

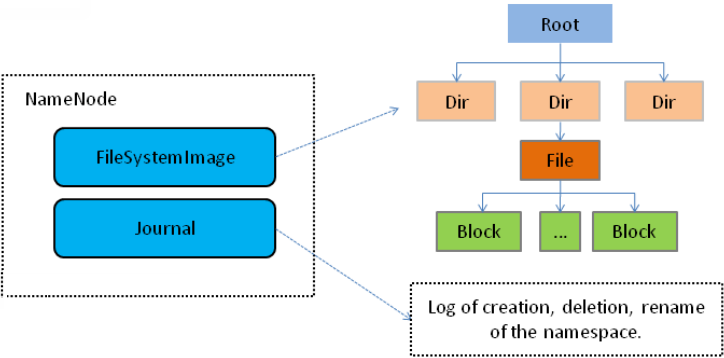


## hdfs基本概念

### （1）Namenode

管理者文件系统的Namespace。它维护着文件系统树(filesystem tree)以及文件树中所有的文件和文件夹的元数据(metadata)。管理这些信息的文件有两个，分别是Namespace 镜像文件(Namespace image)和操作日志文件(edit log)，这些信息被Cache在RAM中，当然，这两个文件也会被持久化存储在本地硬盘。Namenode记录着每个文件中各个块所在的数据节点的位置信息

Namenode结构图课抽象为如图



### （2）Datanode

是文件系统的工作节点，存储具体的数据文件，他们根据客户端或者是namenode的调度存储和检索数据，并且定期向namenode发送他们所存储的块(block)的列表

### （3）Namenode容错机制

　　没有Namenode，HDFS就不能工作。事实上，如果运行namenode的机器坏掉的话，系统中的文件将会完全丢失，因为没有其他方法能够将位于不同datanode上的文件块(blocks)重建文件。因此，namenode的容错机制非常重要，Hadoop提供了两种机制。

　　第一种方式是将持久化存储在本地硬盘的文件系统元数据备份。Hadoop可以通过配置来让Namenode将他的持久化状态文件写到不同的文件系统中。这种写操作是同步并且是原子化的。比较常见的配置是在将持久化状态写到本地硬盘的同时，也写入到一个远程挂载的网络文件系统。

　　第二种方式是运行一个辅助的Namenode(Secondary Namenode)。 事实上Secondary Namenode并不能被用作Namenode它的主要作用是定期的将Namespace镜像与操作日志文件(edit log)合并，以防止操作日志文件(edit log)变得过大。通常，Secondary Namenode 运行在一个单独的物理机上，因为合并操作需要占用大量的CPU时间以及和Namenode相当的内存。辅助Namenode保存着合并后的Namespace镜像的一个备份，万一哪天Namenode宕机了，这个备份就可以用上了。

　　但是辅助Namenode总是落后于主Namenode，所以在Namenode宕机时，数据丢失是不可避免的。在这种情况下，一般的，要结合第一种方式中提到的远程挂载的网络文件系统(NFS)中的Namenode的元数据文件来使用，把NFS中的Namenode元数据文件，拷贝到辅助Namenode，并把辅助Namenode作为主Namenode来运行。

# 五spark

## 简介

Spark是加州大学伯克利分校AMP实验室（Algorithms, Machines, and People Lab）开发的通用内存并行计算框架。Spark使用Scala语言进行实现，它是一种面向对象、函数式编程语言，能够像操作本地集合对象一样轻松地操作分布式数据集，具有以下特点:

1.运行速度快：Spark拥有DAG执行引擎，支持在内存中对数据进行迭代计算。官方提供的数据表明，如果数据由磁盘读取，速度是Hadoop MapReduce的10倍以上，如果数据从内存中读取，速度可以高达100多倍。

2.易用性好：Spark不仅支持Scala编写应用程序，而且支持Java和Python等语言进行编写，特别是Scala是一种高效、可拓展的语言，能够用简洁的代码处理较为复杂的处理工作。

3.通用性强：Spark生态圈即BDAS（伯克利数据分析栈）包含了Spark Core、Spark SQL、Spark Streaming、MLLib和GraphX等组件，这些组件分别处理Spark Core提供内存计算框架、SparkStreaming的实时处理应用、Spark SQL的即席查询、MLlib或MLbase的机器学习和GraphX的图处理。

4.随处运行：Spark具有很强的适应性，能够读取HDFS、Cassandra、HBase、S3和Techyon为持久层读写原生数据，能够以Mesos、YARN和自身携带的Standalone作为资源管理器调度job，来完成Spark应用程序的计算

## （一）批处理和流处理

### 1批处理系统

***批处理***在大数据世界有着悠久的历史。批处理主要操作大容量静态数据集，并在计算过程完成后返回结果。

#### 批处理模式中使用的数据集通常符合下列特征：

有界：批处理数据集代表数据的有限集合

持久：数据通常始终存储在某种类型的持久存储位置中

大量：批处理操作通常是处理极为海量数据集的唯一方法

Apache Hadoop是一种专用于批处理的处理框架，Hadoop的处理功能来自MapReduce引擎。MapReduce的处理技术符合使用键值对的map、shuffle、reduce算法要求。

#### 使用场景：

需要处理大量数据的任务通常最适合用批处理操作进行处理（准确性要求较高），大量数据的处理需要付出大量时间，因此批处理不适合实时性要求较高的场合

### 2流处理系统

流处理中的数据集是“无边界”的，流处理方式无需针对整个数据集执行操作，而是对通过系统传输的每个数据项执行操作，即处理瞬时数据，实时性要求较高

## spark之批处理和流处理模式

### 1批处理模式

与MapReduce不同，Spark的数据处理工作全部在内存中进行，只在一开始将数据读入内存，以及将最终结果持久存储时需要与存储层交互。所有中间态的处理结果均存储在内存中。

为了实现内存中批计算，Spark会使用一种名为Resilient Distributed Dataset（弹性分布式数据集），即RDD的模型来处理数据。这是一种代表数据集，只位于内存中，永恒不变的结构。针对RDD执行的操作可生成新的RDD。每个RDD可通过世系（Lineage）回溯至父级RDD，并最终回溯至磁盘上的数据。Spark可通过RDD在无需将每个操作的结果写回磁盘的前提下实现容错

### 2流处理模式

流处理能力是由Spark Streaming实现的。Spark本身在设计上主要面向批处理工作负载，为了弥补引擎设计和流处理工作负载特征方面的差异，Spark实现了一种叫做微批（Micro-batch）\*的概念

Spark Streaming会以亚秒级增量对流进行缓冲，随后这些缓冲会作为小规模的固定数据集进行批处理。这种方式的实际效果非常好，但相比真正的流处理框架在性能方面依然存在不足

### 3优势和使用场景

#### 3.1优势

1）使用Spark而非Hadoop MapReduce的主要原因是速度。在内存计算策略和先进的DAG调度等机制的帮助下，Spark可以用更快速度处理相同的数据集。

2）Spark的另一个重要优势在于多样性。该产品可作为独立集群部署，或与现有Hadoop集群集成。该产品可运行批处理和流处理，运行一个集群即可处理不同类型的任务。

3）除了引擎自身的能力外，围绕Spark还建立了包含各种库的生态系统，可为机器学习、交互式查询等任务提供更好的支持。相比MapReduce，Spark任务更是“众所周知”地易于编写，因此可大幅提高生产力。

#### 3.2使用场景

1. 为流处理系统采用批处理的方法，需要对进入系统的数据进行缓冲。缓冲机制使得该技术可以处理非常大量的传入数据，提高整体吞吐率，但等待缓冲区清空也会导致延迟增高。这意味着Spark Streaming可能不适合处理对延迟有较高要求的工作负载。

（对于重视吞吐率而非延迟的工作负载，则比较适合使用Spark Streaming作为流处理解决方案）

2）由于内存通常比磁盘空间更贵，因此相比基于磁盘的系统，Spark成本更高。然而处理速度的提升意味着可以更快速完成任务，在需要按照小时数为资源付费的环境中，这一特性通常可以抵消增加的成本。

3）Spark内存计算这一设计的另一个后果是，如果部署在共享的集群中可能会遇到资源不足的问题。相比Hadoop MapReduce，Spark的资源消耗更大，可能会对需要在同一时间使用集群的其他任务产生影响。从本质来看，Spark更不适合与Hadoop堆栈的其他组件共存一处。

## （三）spark vs MapReduce（spark计算框架为什么比MR块）

1 数据存储结构

MR中，每个map处理一个hdfs的数据切片；spark中，使用内存构建弹性分布式数据集RDD，对数据进行运算和cache

2 进行迭代计算时，MR会将结果存储到hdfs（磁盘IO），而spark中间结果存储在内存，存取速度是磁盘的多个数量级



3 MR中，一个task对应一个进程，spark中一个task对应一个线程，而计算机启动进程比启动线程慢，因为启动进程还要为其分配CPU和内存等资源

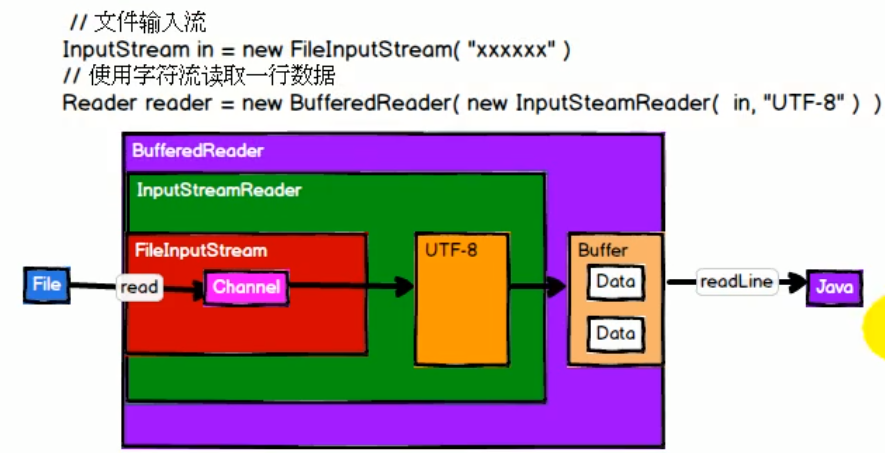
## 基本概念

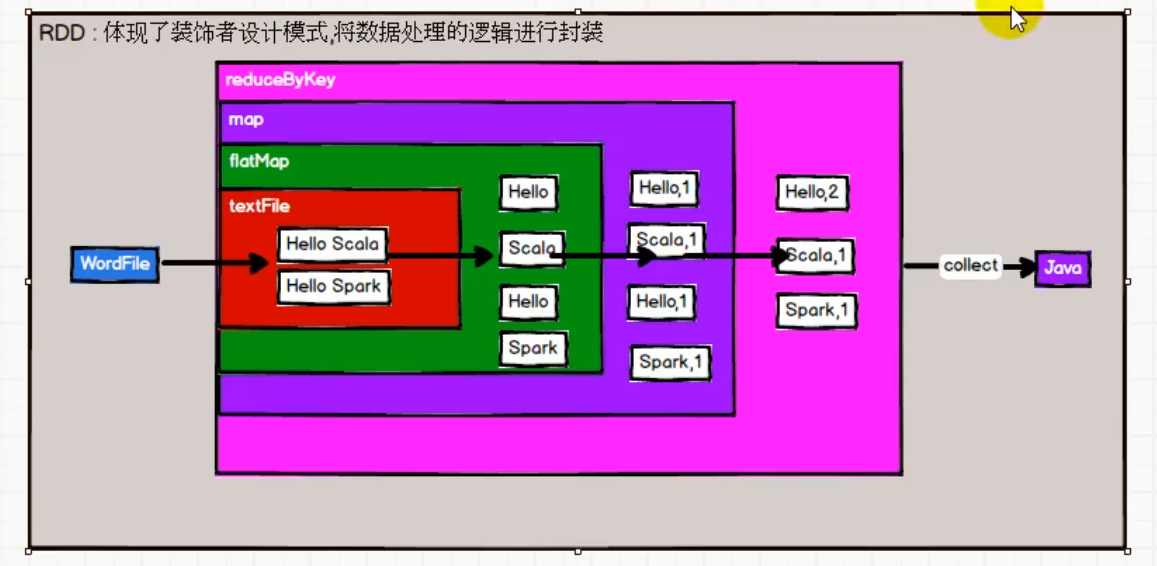
### Driver和executor





## spark 和JavaIO对比





Spark参照Java的装饰者设计模式，将数据的处理逻辑进行封装，同Java的reader.reaLine()类似，spark只有调用collect或print方法时才会触发数据的读取和处理

## sparkRDD的自身优化

### 1 本地化：移动数据不如移动计算，数据和计算都在本地为最优

final def preferredLocations(split: Partition): Seq[String] = {

checkpointRDD.map(\_.getPreferredLocations(split)).getOrElse {

getPreferredLocations(split)

}

}

当客户端提交任务到Driver时，假设有slave1（源头数据在slave1的Executor1进程中）:Executor1,Executor2

#### 1）进程(executor)本地化：

理想情况是将计算放到和数据同一个节点的同一个executor1进程中，

#### **2）节点本地化**：

次之（executor1繁忙）是放到同一个节点的另外一个executor2，

#### **3）机架本地化**：

再次之放到和slave1同机架的另外一个节点如slave2

# 六 flume

Flume事务

Source到channel的doPut事务

Channel到sink的take事务

Source类型：

Exec 使用linux命令（如 tail -f）监控文件变化，但存在单点故障问题（挂掉重启后，无法接着读取）

Taildir 监控某个目录的所有文件，支持断点续传

# 七 kafka

kafka是一个分布式的基于发布订阅模式的消息队列

## 常用命令：

1.kafka启动: ./kafka-server-start.sh ../config/server.properties &

2.创建topic: ./kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic tes

3.查看kafka的topic：./kafka-topics.sh --zookeeper master:2181 --list

4.查看kafka某个topic下partition信息: ./kafka-topics.sh --describe --zookeeper master:2181 --topic test-topic

5.查看kafka的指定topic: ./kafka-topics.sh --zookeeper master:2181 --describe --topic yq20171220

6.控制台向kafka生产数据: ./kafka-console-producer.sh --broker-list master:9092 --topic jason\_20180519

7.控制台消费kafka的数据: ./kafka-console-consumer.sh --zookeeper storm1:2181 --topic jason\_20180519 --from-beginning

8.查看topic下某分区偏移量的最小值: ./kafka-run-class.sh kafka.tools.GetOffsetShell --topic test-topic --time -1 --broker-list master:9092 --partitions 0

9.增加topic的partition:/kafka-topics.sh --alter --topic jason\_20180519 --zookeeper 10.200.10.24:2181,10.200.10.26:2181,10.200.10.29:2181 --partitions 5

10.删除topic，慎用，只会删除zookeeper中的元数据，消息文件须手动删除: ./kafka-run-class.sh kafka.admin.DeleteTopicCommand --zookeeper master:2181 --topic yq20171220

11.彻底删除topic: rmr /brokers/topics/【topic name】即可

## （一）消息类型

### 1点对点模式

消息生产者生产消息发送到queue中，然后消息消费者从queue中取出并且消费消息。这里要注意：

消息被消费以后，queue中不再有存储，所以消息消费者不可能消费到已经被消费的消息。

Queue支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费

### 2发布订阅模式

消息生产者（发布）将消息发布到topic中，同时有多个消息消费者（订阅）消费该消息。和点对点方式不同，发布到topic的消息会被所有订阅者消费

## （二）基本概念



### producer：生产者

### consumer：消费者，如果是消费者组，那么一个分区的数据只会被组中的某个消费者消费，而不是一个分区的数据被多个消费者消费

### broker：每个节点叫一个broker

### topic：不同的生产者写数据到不同的topic，不同的消费者订阅不同的topic，消费数据。

### partition：一个topic可分为多个partition。

每个partition有对应的leader和follower，每个partition的副本个数=该partition的leader数+follower数，且只有leader对外提供服务，leader挂掉后，follower转变为leader提供服务

### offset：消费者每次消费消息时，会记录消费的位置，即offset

### 注意：

一个partition的一条消息，只能被一个消费者组中的一个消费者消费，但是其他消费者消费

如broker1，topicA-partition0的消息msg1，订阅topicA的消费组（A，B），A如果消费了msg1，则B不能再次消费，但B可以消费broker2，topicA-partition1的消息msg2。并且msg1可以被C消费。这样设计，可以使得消费组提高消费速度，即如果topicA 分区p1 p2，有100条消息，如果只有一个消费者A，那么A会消费100次，而如果两个消费者A，B，则分别只需消费50次。

### zookeeper在kafka的作用：

1管理kafka集群

2 kafka0.9之前，offset存储在zookeeper，0.9之后存储在kafka集群的系统topic中（实质在磁盘上）

## kafka查找原理

### 数据文件的分段

Kafka解决查询效率的手段之一是将数据文件分段，比如有100条Message，它们的offset是从0到99。假设将数据文件分成5段，第一段为0-19，第二段为20-39，以此类推，每段放在一个单独的数据文件里面，数据文件以该段中最小的offset命名。这样在查找指定offset的Message的时候，用二分查找就可以定位到该Message在哪个段中。

### 为数据文件建索引

数据文件分段使得可以在一个较小的数据文件中查找对应offset的Message了，但是这依然需要顺序扫描才能找到对应offset的Message。为了进一步提高查找的效率，Kafka为每个分段后的数据文件建立了索引文件，文件名与数据文件的名字是一样的，只是文件扩展名为.index。

索引文件中包含若干个索引条目，每个条目表示数据文件中一条Message的索引。索引包含两个部分（均为4个字节的数字），分别为相对offset和position。

相对offset：因为数据文件分段以后，每个数据文件的起始offset不为0，相对offset表示这条Message相对于其所属数据文件中最小的offset的大小。举例，分段后的一个数据文件的offset是从20开始，那么offset为25的Message在index文件中的相对offset就是25-20 = 5。存储相对offset可以减小索引文件占用的空间。

position，表示该条Message在数据文件中的绝对位置。只要打开文件并移动文件指针到这个position就可以读取对应的Message了。

index文件中并没有为数据文件中的每条Message建立索引，而是采用了稀疏存储的方式，每隔一定字节的数据建立一条索引。这样避免了索引文件占用过多的空间，从而可以将索引文件保留在内存中。但缺点是没有建立索引的Message也不能一次定位到其在数据文件的位置，从而需要做一次顺序扫描，但是这次顺序扫描的范围就很小了



### 在需要查找一个offset为368801的message是什么样的过程

#### 1、 先找到offset的368801message所在的segment文件（利用二分法查找），这里找到的就是在第二个segment文件。

#### 2、 打开找到的segment中的.index文件（也就是368796.index文件，该文件起始偏移量为368796+1，我们要查找的offset为368801的message在该index内的偏移量为368796+5=368801，所以这里要查找的相对offset为5）。由于该文件采用的是稀疏索引的方式存储着相对offset及对应message物理偏移量的关系，所以直接找相对offset为5的索引找不到，这里同样利用二分法查找相对offset小于或者等于指定的相对offset的索引条目中最大的那个相对offset，所以找到的是相对offset为4的这个索引。

#### 3、 根据找到的相对offset为4的索引确定message存储的物理偏移位置为256。打开数据文件，从位置为256的那个地方开始顺序扫描直到找到offset为368801的那条Message。

这套机制是建立在offset为有序的基础上，利用segment+有序offset+稀疏索引+二分查找+顺序查找等多种手段来高效的查找数据！至此，消费者就能拿到需要处理的数据进行处理了。那每个消费者又是怎么记录自己消费的位置呢？在早期的版本中，消费者将消费到的offset维护zookeeper中，consumer每间隔一段时间上报一次，这里容易导致重复消费，且性能不好！在新的版本中消费者消费到的offset已经直接维护在kafk集群的\_\_consumer\_offsets这个topic中