Spark 笔记

Spark

1.概述

Spark 是一种快速、通用、可扩展的大数据分析引擎。目前，Spark 生态系统已经发展成为一个包含多个子项目的集合，其中包含 SparkSQL、SparkStreaming、GraphX、MLlib 等子项目，Spark 是基于内存计算的大数据并行计算框架。Spark 基于内存计算，提高了在大数据环境下数据处理的实时性，同时保证了高容错性和高可伸缩性，允许用户将 Spark 部署在大量廉价硬件之上，形成集群。基于 MapReduce 的计算引擎通常会将中间结果输出到磁盘上，进行存储和容错。出于任务管道承接的考虑，当一些查询翻译到 MapReduce 任务时，往往会产生多个 Stage，而这些串联的 Stage 又依赖于底层文件系统（如 HDFS）来存储每一个 Stage 的输出结果。
Spark 是 MapReduce 的替代方案，而且兼容 HDFS、Hive，可融入 Hadoop 的生态系统，以弥补 MapReduce 的不足。

2.特点

2.1.易用性

支持 Scala(原生语言)、Java、Python 和 Spark SQL。Spark SQL 非常类似于 SQL 92，所以几乎不需要经历一番学习，马上可以上手。

Spark 还有一种交互模式，那样开发人员和用户都可以获得查询和其他操作的即时反馈。MapReduce 没有交互模式，不过有了 Hive 和 Pig 等附加模块，采用者使用 MapReduce 来得容易一点。

2.2.成本低

“Spark 已证明在数据多达 PB 的情况下也轻松自如。它被用于在数量只有十分之一的机器上，对 100TB 数据进行排序的速度比 Hadoop MapReduce 快 3 倍。”这一成绩让 Spark 成为 2014 年 Daytona GraySort 基准。

2.3.兼容性

MapReduce 和 Spark 相互兼容;MapReduce 通过 JDBC 和 ODC 兼容诸多数据源、文件格式和商业智能工具，Spark 具有与 MapReduce 同样的兼容性。

2.4.数据处理

MapReduce 是一种批量处理引擎。MapReduce 以顺序步骤来操作，先从集群读取数据，然后对数据执行操作，将结果写回到集群，从集群读 取更新后的数据，执行下一个数据操作，将那些结果写回到结果，依次类推。Spark 执行类似的操作，不过是在内存中一步执行。它从集群读取数据后，对数据 执行操作，然后写回到集群。

Spark 还包括自己的图形计算库 GraphX。GraphX 让用户可以查看与图形和集合同样的数据。用户还可以使用弹性分布式数据集(RDD)，改变和联合图形，容错部分作了讨论。

2.5.容错

至于容错，MapReduce 和 Spark 从两个不同的方向来解决问题。MapReduce 使用 TaskTracker 节点，它为 JobTracker 节点提供了心跳(heartbeat)。如果没有心跳，那么 JobTracker 节点重新调度所有将执行的操作和正在进行的操作，交给另一个 TaskTracker 节点。这种方法在提供容错性方面很有效，可是会大大延长某些操作(即便只有一个故障)的完成时间。

Spark 使用弹性分布式数据集(RDD)，它们是容错集合，里面的数据元素可执行并行操作。RDD 可以引用外部存储系统中的数据集，比如共享式文件系统、HDFS、HBase，或者提供 Hadoop InputFormat 的任何数据源。Spark 可以用 Hadoop 支持的任何存储源创建 RDD，包括本地文件系统，或前面所列的其中一种文件系统。

RDD 拥有五个主要属性：

- 分区列表
- 计算每个分片的函数
- 依赖其他 RDD 的项目列表
- 面向键值 RDD 的分区程序(比如说 RDD 是散列分区)，这是可选属性
- 计算每个分片的首选位置的列表(比如 HDFS 文件的数据块位置)，这是可选属性

RDD 可能具有持久性，以便将数据集缓存在内存中。这样一来，以后的操作大大加快，最多达 10 倍。Spark 的缓存具有容错性，原因在于如果 RDD 的任何分区丢失，就会使用原始转换，自动重新计算。

2.6.可扩展性

按照定义，MapReduce 和 Spark 都可以使用 HDFS 来扩展。那么，Hadoop 集群能变得多大呢?

据称雅虎有一套 42000 个节点组成的 Hadoop 集群，可以说扩展无极限。最大的已知 Spark 集群是 8000 个节点，不过随着大数据增多，预计集群规模也会随之变大，以便继续满足吞吐量方面的预期。

2.7.安全

Hadoop 支持 Kerberos 身份验证，这管理起来有麻烦。然而，第三方厂商让企业组织能够充分利用活动目录 Kerberos 和 LDAP 用于身份验证。同样那些第三方厂商还为传输中数据和静态数据提供数据加密。
Hadoop 分布式文件系统支持访问控制列表(ACL)和传统的文件权限模式。Hadoop 为任务提交中的用户控制提供了服务级授权(Service Level Authorization)，这确保客户拥有正确的权限。
Spark 的安全性弱一点，目前只支持通过共享密钥(密码验证)的身份验证。Spark 在安全方面带来的好处是，如果你在 HDFS 上运行 Spark，它可以使用 HDFS ACL 和文件级权限。此外，Spark 可以在 YARN 上运行，因而能够使用 Kerberos 身份验证。

3.用于 Spark 的资源调度系统

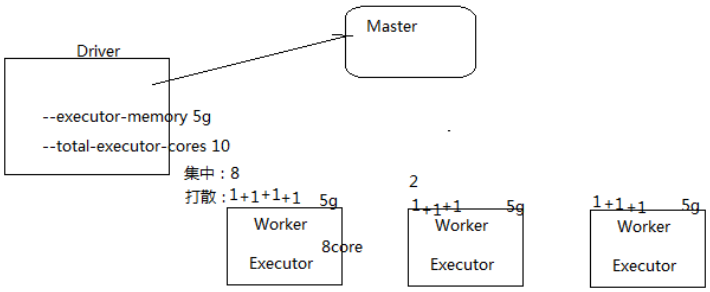
Local（本地模式）
Standalone-是 Spark 自己独有的资源调度系统
On yarn-可以运行多种任务（MapReduce，Spark，Storm，Flink…）
Messos apache 专门用于资源调度
Docker 云计算（创造容器，甚至可以安装 Windows 系统）

现在还是有很多公司会把 Spark 应用程序提交到 yarn 上运行（spark-on-yarn），为什么要用 spark-on-yarn?
1. 因为历史原因，方便运维部门去维护。
2. 用 yarn 来运行各种任务相比较其他的资源调度系统更稳定，便于升级优化。

4.资源调度的两种方式

尽量集中 尽量打散

资源调度：默认是尽量打散，还有一种是尽量集中



5.集群搭建

01.Spark 基础.docx

6.执行 Spark 程序

6.1.执行第一个 spark 程序

```
/usr/local/spark-1.6.1-bin-hadoop2.6/bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master spark://node01:7077 \
--executor-memory 1G \
--total-executor-cores 2 \
/usr/local/spark-1.6.1-bin-hadoop2.6/lib/spark-examples-1.6.1-hadoop2.6.0.jar \
100
```

6.2.启动 Spark Shell

spark-shell 是 Spark 自带的交互式 Shell 程序，方便用户进行交互式编程，用户可以在该命令行下用 scala 编写 spark 程序。

```
/usr/local/spark-1.6.1-bin-hadoop2.6/bin/spark-shell \
--master spark://node01:7077 \
--executor-memory 2g \
```

--total-executor-cores 2

参数说明：
--master spark://node01:7077 指定 Master 的地址
--executor-memory 2g 指定每个 worker 可用内存为 2G
--total-executor-cores 2 指定整个集群使用的 cup 核数为 2 个

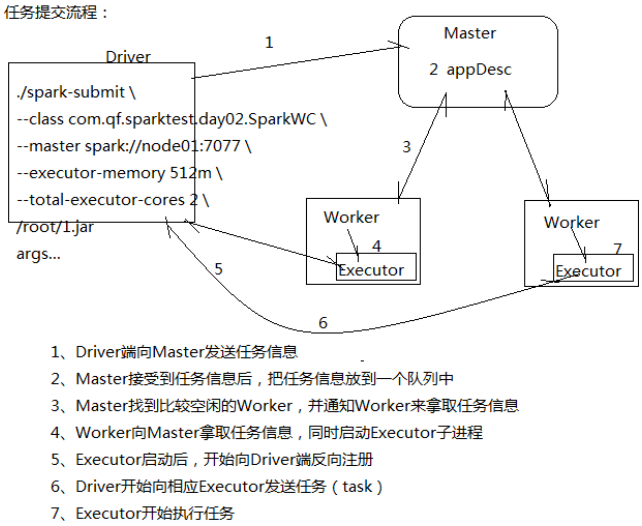
注意：
如果启动 spark shell 时没有指定 master 地址，但是也可以正常启动 spark shell 和执行 spark shell 中的程序，其实是启动了 spark 的 local 模式，该模式仅在本机启动一个进程，没有与集群建立联系。

Spark Shell 中已经默认将 SparkContext 类初始化为对象 sc。用户代码如果需要用到，则直接应用 sc 即可

7.Spark 集群启动流程

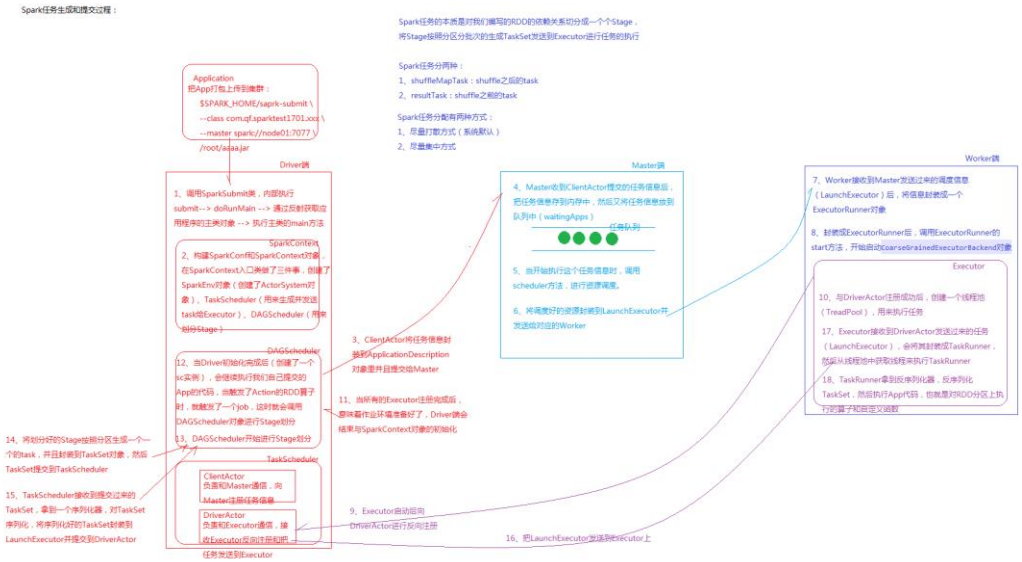
1. 通过调用 start-all.sh 来启动 Master 和 Worker，首先启动的是 Master
2. Master 服务启动后，在 prestart 方法中会启动一个定时器，定时检查超时的 Worker
3. 执行 receive 方法，不断地接受其他 actor 发送过来的请求
4. 再调用 start-all.sh 脚本的时候，会解析 slaves 配置文件，获取到用于启动 Worker 的节点
5. 开始在相应节点启动 Worker 服务
6. Worker 服务启动的过程中，也会先执行 prestart 方法，该方法主要是向 Master 进行注册
7. Worker 向 Master 进行注册
8. Master 收到注册信息后，把注册信息保存到缓存和磁盘
9. Master 保存完注册信息后，开始向 Worker 响应注册成功的信息（masterURL）
10. Worker 收到注册成功的信息，把 masterURL 保存一次，并开始心跳，到这里，整个集群启动成功。

8.Spark 任务提交流程



1. Driver 端向 Master 发送任务消息
2. Master 接受到任务信息后，把任务信息放到一个队列中
3. Master 找到比较空闲的 Worker，并通知 Worker 来拿取任务信息
4. Worker 向 Master 拿取任务信息，同时启动 Executor 子进程
5. Executor 启动后，开始向 Driver 端反向注册
6. Driver 开始向相应 Executor 发送任务（task）
7. Executor 开始执行任务

9.Spark 任务生成和提交流程（源码级分析）



把 App 打包上传到集群：$SPARK_HOME/spark-submit \
--class com.qf.sparktest.XXX \
--master spark://mini1:7077 \
--executor-memory 1g \
--total-executor-cores 2 \
/jar/XXX.jar
100

1. 调用 SparkSubmit 类，内部执行 submit-->doRunMain-->通过反射获取应用程序的主类对象（远程代理对象）-->执行主类的 main 方法
2. 构建 SparkConf 和 SparkContext 对象，在 SparkContext 入口类做了三件事，创建了 SparkEnv 对象（创建了 ActorSystem 对象）、TaskScheduler（用来生成并发送 task 给 Executor）、DAGScheduler（用来划分 Stage）
3. ClientActor 将任务信息封装到 ApplicationDescription 对象里并且提交给 Master
4. Master 收到 ClientActor 提交的任务信息后，把任务信息存到内存中，然后又将任务信息放到队列中（waitingApps）
5. 当开始执行这个任务信息时，调用 scheduler 方法，进行资源调度。
6. 将调度好的资源封装到 LaunchExecutor 并发送给对应的 Worker
7. Worker 接收到 Master 发送过来的调度信息（LaunchExecutor）后，将信息封装成一个 ExecutorRunner 对象
8. 封装成 ExecutorRunner 后，调用 ExecutorRunner 的 Start 方法，开始启动 CoarseGrainedExecutorBackend 对象（启动 Executor）
9. Executor 启动后向 DriverActor 进行反向注册
10. 与 DriverActor 注册成功后，创建一个线程池（ThreadPool），用来执行任务
11. 当所有 Executor 注册完成后，意味着作业环境准备好了，Driver 端会结束与 SparkContext 对象的初始化

12. 当 Driver 初始化完成后（创建了一个 sc 示例），会持续执行我们自己提交的 App 的代码，当触发了 Action 的 RDD 算子时，就触发了一个 job，这时会调用 DAGScheduler 对象进行 Stage 划分
13. DAGScheduler 开始进行 Stage 划分
14. 将划分好的 Stage 按照分区生成一个一个的 task, 并且封装到 TaskSet 对象, 然后 TaskSet 提交到 TaskScheduler
15. TaskScheduler 接收到提交过来的 TaskSet，拿到一个序列化器对 TaskSet 序列化，将序列化好的 TaskSet 封装到 LaunchExecutor 并提交到 DriverActor
16. DriverActor 把 LaunchExecutor 发送到 Executor 上
17. Executor 接收到 DriverActor 发送过来的任务（LaunchExecutor），会将其封装成 TaskRunner，然后从线程池中获取线程来执行 TaskRunner
18. TaskRunner 拿到反序列化器，反序列化 TaskSet，然后执行 App 代码，也就是对 RDD 分区上执行的算子和自定义函数

10.SparkContext 初始化

SparkContext 是 Spark 提交任务到集群的入口
我们看一下 SparkContext 的主构造器：(SparkContext 初始化时做的三件事)

1. 调用 CreateSparkEnv 方法创建 SparkEnv，里面有一个非常重要的对象 ActorSystem
2. 创建 TaskScheduler（生成 task 并且提交任务）->根据任务提交的 URL 进行匹配->TaskSchedulerImpl ->SparkDeployScheduler
3. 创建 DAGScheduler（划分 stage）
4. taskScheduler.start（）//上述初始化操作完成后调用 start 方法开始执行

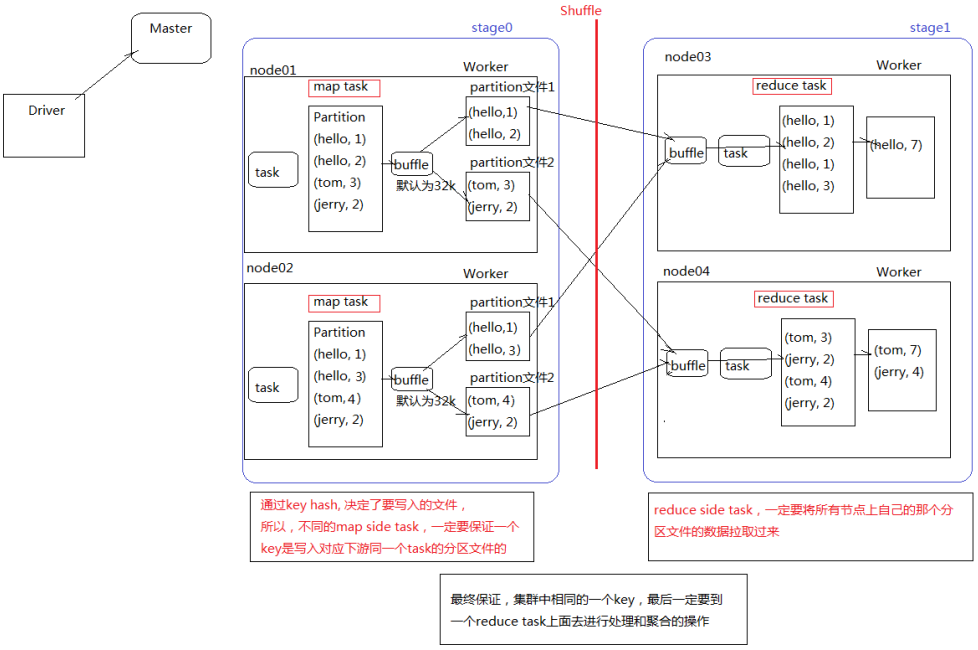
11.后端调度器创建了那两个 actor，分别是和谁进行交互的

1.DriverActor：driver 端与 Executor 通信，向 Executor 发送任务
2.ClientActor：driver 端与 Master 通信，向 Master 发送任务信息

12.clientActor 是怎么向 master 注册的

与 Worker 向 Master 注册类似
1.调用 tryRegisterAllMaster 方法向 Master 注册（实际就是把封装后任务信息发送给 Master）
2.循环所有 Master 地址，跟 Master 建立连接
3.拿到了 Master 的一个引用，然后向 Master 发送注册应用的请求，所有的参数都封装到 appDescription
4.Master 首先把应用的消息放到内存中存储
5.利用持久化引擎保存到磁盘
6.Master 向 ClientActor 发送注册成功的消息
7.Master 开始调度资源，其实就是把任务启动到哪些 Worker 上

13.Spark 的 shuffle 过程



shuffle write：

map side 端将相同 key 的数据先写入内存缓冲区 buffer（默认 32k）,缓冲区满，溢出到磁盘文件,多个 key 会被写到多个分区文件中。

shuffle read：

从上游 stage 的所有 task 节点上拉取属于自己分区的磁盘文件，保证相同的 key 写入到同一个 reduce task 中，每个 read task 会有自己的 buffer 缓冲，每次只能拉取与 buffer 缓冲相同大小的数据，然后聚合，聚合完一批后拉取下一批，该拉取过程，边拉取边聚合，这个过程就称之为 shuffle。

Spark shuffle 过程中不会对数据进行排序，而且是跨 stage 的。

Spark Core

1. RDD

1.1.RDD 的概念

RDD 的中文解释为： 弹性分布式数据集，全称 Resilient Distributed Datasets。RDD 只读、可分区，这个数据集的全部或部分可以缓存在内存中，在多次计算间重用。 所谓弹性，是指内存不够时可以与磁盘进行交换。这涉及到了 RDD 的另一特性：内存计算，就是将数据保存到内存中。同时，为解决内存容量限制问题，Spark 为我们提供了最大的自由度，所有数据均可由我们来进行 cache 的设置，包括是否 cache 和如何 cache。

1.2.RDD 的属性

1.一组分片（Partition），即数据集的基本组成单位。对于 RDD 来说，每个分片都会被一个计算任务处理，并决定并行计算的粒度。用户可以在创建 RDD 时指定 RDD 的分片个数，如果没有指定，那么就会采用默认值。默认值就是程序所分配到的 CPU Core 的数目。

2.一个计算每个分区的函数。Spark 中 RDD 的计算是以分片为单位的，每个 RDD 都会实现 compute 函数以达到这个目的。compute 函数会对迭代器进行复合，不需要保存每次计算的结果。

3.RDD 之间的依赖关系。RDD 的每次转换都会生成一个新的 RDD，所以 RDD 之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark 可以通过这个依赖关系重新计算丢失的分区数据，而不是对 RDD 的所有分区进行重新计算。

4.一个 Partitioner，即 RDD 的分片函数。当前 Spark 中实现了两种类型的分片函数，一个是基于哈希的 HashPartitioner，另外一个是基于范围的 RangePartitioner。只有对于 key-value 的 RDD，才会有 Partitioner，非 key-value 的 RDD 的 Parititioner 的值是 None。Partitioner 函数不但决定了 RDD 本身的分片数量，也决定了 parent RDD Shuffle 输出时的分片数量。

5.一个列表，存储存取每个 Partition 的优先位置（preferred location）。对于一个 HDFS 文件来说，这个列表保存的就是每个 Partition 所在的块的位置。按照“移动数据不如移动计算”的理念，Spark 在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置。

1.3.创建 RDD

1.由一个已经存在的 Scala 集合创建。

```
val rdd1 = sc.parallelize(Array(1,2,3,4,5,6,7,8))
```

2.由外部存储系统的数据集创建，包括本地的文件系统，还有所有 Hadoop 支持的数据集，比如 HDFS、Cassandra、HBase 等

```
val rdd2 = sc.textFile("hdfs://mini1:8020/jar/words.txt")
```

1.4.RDD 常用 API

1.4.1.Transformation

RDD 中的所有转换都是延迟加载的，也就是说，它们并不会直接计算结果。相反的，它们只是记住这些应用到基础数据集（例如一个文件）上的转换动作。只有当发生一个要求返回结果给 Driver 的动作(Action)时，这些转换才会真正运行。这种设计让 Spark 更加有效率地

运行。

转换	含义
map(func)	返回一个新的 RDD，该 RDD 由每一个输入元素经过 func 函数转换后组成
filter(func)	返回一个新的 RDD，该 RDD 由经过 func 函数计算后返回值为 true 的输入元素组成
flatMap(func)	类似于 map，但是每一个输入元素可以被映射为 0 或多个输出元素（所以 func 应该返回一个序列，而不是单一元素）
mapPartitions(func)	类似于 map，但独立地在 RDD 的每一个分片上运行，因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是 Iterator[T] => Iterator[U]
mapPartitionsWithIndex(func)	类似于 mapPartitions，但 func 带有一个整数参数表示分片的索引值，因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是 (Int, Iterator[T]) => Iterator[U]
sample(withReplacement, fraction, seed)	根据 fraction 指定的比例对数据进行采样，可以选择是否使用随机数进行替换，seed 用于指定随机数生成器种子
union(otherDataset)	对源 RDD 和参数 RDD 求并集后返回一个新的 RDD
intersection(otherDataset)	对源 RDD 和参数 RDD 求交集后返回一个新的 RDD
distinct([numTasks]))	对源 RDD 进行去重后返回一个新的 RDD
groupByKey([numTasks])	在一个(K,V)的 RDD 上调用，返回一个(K, Iterator[V])的 RDD
reduceByKey(func, [numTasks])	在一个(K,V)的 RDD 上调用，返回一个(K,V)的 RDD，使用指定的 reduce 函数，将相同 key 的值聚合到一起，与 groupByKey 类似，reduce 任务的个数可以通过第二个可选的参数来设置
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	
sortByKey([ascending], [numTasks])	在一个(K,V)的 RDD 上调用，K 必须实现 Ordered 接口，返回一个按照 key 进行排序的(K,V)的 RDD
sortBy(func,[ascending], [numTasks])	与 sortByKey 类似，但是更灵活
join(otherDataset, [numTasks])	在类型为(K,V)和(K,W)的 RDD 上调用，返回一个相同 key 对应的所有元素对在一起的(K,(V,W))的 RDD
cogroup(otherDataset, [numTasks])	在类型为(K,V)和(K,W)的 RDD 上调用，返回一个(K,(Iterable<V>,Iterable<W>))类型的 RDD
cartesian(otherDataset)	笛卡尔积
pipe(command, [envVars])	
coalesce(numPartitions)	重新分区
repartition(numPartitions)	重新分区
repartitionAndSortWithinPartitions(p	重新分区

artitioner)	

1.4.2.Action

动作	含义
reduce(*func*)	通过 func 函数聚集 RDD 中的所有元素，这个功能必须是可交换且可并联的
collect()	在驱动程序中，以数组的形式返回数据集的所有元素
count()	返回 RDD 的元素个数
first()	返回 RDD 的第一个元素（类似于 take(1)）
take(*n*)	返回一个由数据集的前 n 个元素组成的数组
takeSample(*withReplacement*,*num*, [*seed*])	返回一个数组，该数组由从数据集中随机采样的 num 个元素组成，可以选择是否用随机数替换不足的部分，seed 用于指定随机数生成器种子
takeOrdered(*n*, *[ordering]*)	takeOrdered 和 top 类似，只不过以和 top 相反的顺序返回元素
saveAsTextFile(*path*)	将数据集的元素以 textfile 的形式保存到 HDFS 文件系统或者其他支持的文件系统，对于每个元素，Spark 将会调用 toString 方法，将它装换为文件中的文本
saveAsSequenceFile(*path*)	将数据集中的元素以 Hadoop sequencefile 的格式保存到指定的目录下,可以使 HDFS 或者其他 Hadoop 支持的文件系统。
saveAsObjectFile(*path*)	
countByKey()	针对(K,V)类型的 RDD，返回一个(K,Int)的 map，表示每一个 key 对应的元素个数。
foreach(*func*)	在数据集的每一个元素上，运行函数 func 进行更新。

1.5.相似 api

2.RDD 的依赖关系

RDD 和它依赖的父 RDD（s）的关系有两种不同的类型，即窄依赖（narrow dependency）和宽依赖（wide dependency）

窄依赖指的是每一个父 RDD 的 Partition 最多被子 RDD 的一个 Partition 使用
总结：窄依赖我们形象的比喻为**独生子女**
宽依赖指的是多个子 RDD 的 Partition 会依赖同一个父 RDD 的 Partition
总结：宽依赖我们形象的比喻为**超生**
Lineage：RDD 只支持粗粒度转换，即在大量记录上执行的单个操作。将创建 RDD 的一系列 Lineage（即血统）记录下来，以便恢复丢失的分区。RDD 的 Lineage 会记录 RDD 的元数据信息和转换行为，当该 RDD 的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

3. RDD 的缓存

Spark 速度非常快的原因之一，就是在不同操作中可以在内存中持久化或缓存多个数据集。当持久化某个 RDD 后，每一个节点都将把计算的分片结果保存在内存中，并在对此 RDD 或衍生出的 RDD 进行的其他动作中重用。这使得后续的动作变得更加迅速。RDD 相关的持久化和缓存，是 Spark 最重要的特征之一。可以说，缓存是 Spark 构建迭代式算法和快速交互式查询的关键。

RDD 缓存方式（缓存级别）

RDD 通过 persist 方法或 cache 方法可以将前面的计算结果缓存，但是并不是这两个方法被调用时立即缓存，而是触发后面的 action 时，该 RDD 将会被缓存在计算节点的内存中，并供后面重用。
通过查看源码发现 **cache 最终也是调用了 persist 方法**，默认的存储级别都是仅在内存存储一份，Spark 的存储级别还有好多种，存储级别在 object StorageLevel 中定义的。

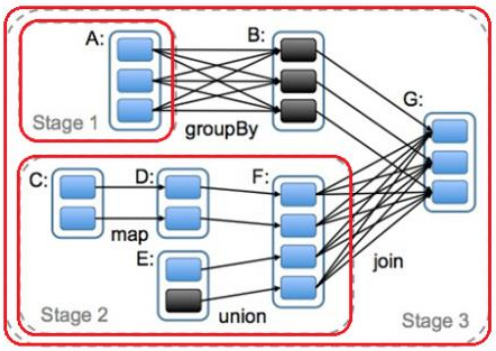
```
object StorageLevel {
  val NONE = new StorageLevel(false, false, false, false)
  val DISK_ONLY = new StorageLevel(true, false, false, false)
  val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)
  val MEMORY_ONLY = new StorageLevel(false, true, false, true)
  val MEMORY_ONLY_2 = new StorageLevel(false, true, false, true, 2)
  val MEMORY_ONLY_SER = new StorageLevel(false, true, false, false)
  val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, false, 2)
  val MEMORY_AND_DISK = new StorageLevel(true, true, false, true)
  val MEMORY_AND_DISK_2 = new StorageLevel(true, true, false, true, 2)
  val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false, false)
  val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, false, 2)
  val OFF_HEAP = new StorageLevel(false, false, true, false)
```

缓存有可能丢失，或者存储于内存的数据由于内存不足而被删除，RDD 的缓存容错机制保证了即使缓存丢失也能保证计算的正确执行。通过基于 RDD 的一系列转换，丢失的数据会被重算，由于 RDD 的各个 Partition 是相对独立的，因此只需要计算丢失的部分即可，并不需要重算全部 Partition。

4.DAG 的生成

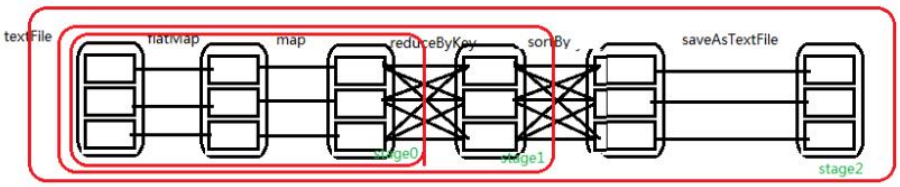
DAG(Directed Acyclic Graph)叫做有向无环图，原始的 RDD 通过一系列的转换就形成了 DAG，根据 RDD 之间的依赖关系的不同将 DAG 划分成不同的 Stage，对于窄依赖，partition 的转换处理在 Stage 中完成计算。对于宽依赖，由于有 Shuffle 的存在，只能在 parent RDD 处理完成后，才能开始接下来的计算，因此宽依赖是划分 Stage 的依据，而划分 stage 的目的是为了生成 task。

5.Stage 的划分依据和划分过程



划分stage的目的是为了生成task，划分stage的依据是查看RDD之间是否发生宽依赖（shuffle）
task的生成一定是在stage范围之内的

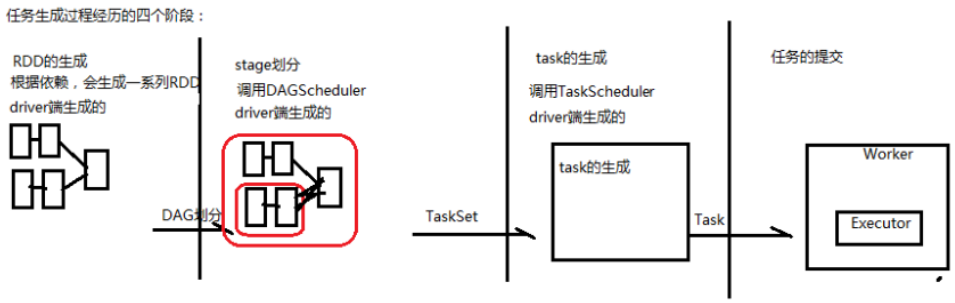
stage的划分过程：从最后一个RDD开始，调用递归，从后往前推，找该RDD和父RDD之间的依赖关系，如果是窄依赖，会继续找父RDD的父RDD，如果是宽依赖，就会从该RDD开始到前面所有的RDD划分为一个stage，递归的出口是直到找不到父RDD，最后把所有的RDD划分为一个stage



划分 Stage 的目的是为了生成 task，划分 stage 的依据是查看 RDD 之间是否发生宽依赖（shuffle），task 生成一定是在 stage 范围之内的。

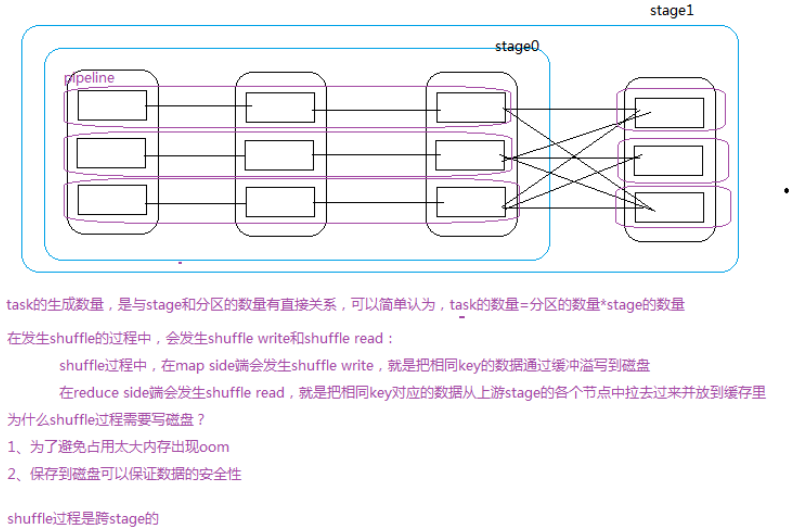
Stage 的划分过程：从最后一个 RDD 开始，调用递归，从后往前推，找该 RDD 和 父 RDD 的依赖关系，如果是窄依赖，会继续找父 RDD 的 RDD，如果是宽依赖，就会从该 RDD 开始到前面所有的 RDD 划分为一个 stage，递归的出口是知道找不到父 RDD，最后把所有的 RDD 或分为一个 stage

6.任务生成并提交的四个阶段



1. RDD 的生成：在 driver 端，根据依赖，会生成一系列 RDD。
2. Stage 划分：在 driver 端，调用 DAGScheduler（SparkContext 初始化），划分 stage。
3. Task 的生成：在 driver 端，调用 TaskScheduler（SparkContext 初始化），生成任务
4. 任务的提交：TaskScheduler 将任务发送给 Executor

7.观察 task 的生成



task 的生成数量，是与 stage 和分区的数量有直接关系，可以简单的认为：
task 的数量=分区的数量*stage 的数量
在 shuffle 过程中，会发生 shuffle write 和 shuffle read：
在 map side 端发生 shufflewrite，就是把相同 key 的数据通过缓冲溢写到磁盘
在 reduce side 端会发生 shuffle read，就是把相同 key 对应的数据从上游 stage 的各个节点中拉取过来并放到缓存中。

为什么 shuffle 过程需要写磁盘?

1. 为了避免占用太大内存出现 oom

2. 保存到磁盘可以保证数据的安全性
注意：shuffle 过程是跨 stage

8.checkpoint 的应用背景和使用步骤

什么时候需要做检查点?

有时候中间结果数据或者 shuffle 后的数据或者血统很长需要在以后的 job 中经常调用，此时需要做 checkpoint（应用背景）

推荐最好把数据 checkpoint 到 HDFS，保证数据安全性的前提下也便于集群所有节点能够获取到

checkpoint 的目的?

提高运算效率、保证数据的安全性

步骤:

1、设置 checkpoint 的目录

sc.setCheckpointDir("hdfs://node01:9000/cp-20180830-1")

2、把中间结果进行缓存

val rdd1 = rdd1.cache()// 找 RDD 数据，1.先找有没有 cache 过 2.找有没有 checkpoint 3.都没有自己重新计算

3、进行 checkpoint

rdd1.checkpoint

rdd1.getCheckpointFile // 获取 checkpoint 的地址

rdd1.isCheckpointed // 查看是否 checkpoint

Spark Sql

1.概述

我们已经学习了 Hive，它是将 Hive SQL 转换成 MapReduce 然后提交到集群上执行，大大简化了编写 MapReduce 的程序的复杂性，由于 MapReduce 这种计算模型执行效率比较慢。所有 Spark SQL 的应运而生，它是将 Spark SQL 转换成 RDD，然后提交到集群执行，执行效率非常快！它有以下几个特点:

1. 易整合
2. 统一的数据访问方式
3. 兼容 Hive
4. 标准的数据连接

2.DataFrame

与 RDD 类似，DataFrame 也是一个数据分布容器，然而 DataFrame 更像传统数据库的二维表格。除了数据以外，还记录数据的结构信息。操作的是里面的 rowRDD 和 schema

3.编写

04.Spark SQL.docx

1. import spark.impilicits._
   rdd1.toDF(colName)
2. Row+StructType
3. case class

4.广播变量

某一变量在 Executor 中计算的时候，需要一次或多次拉取 Driver 端数据，可以从 Driver 端广播到 Executor，计算时直接从本地拉取了。广播变量的好处，不是每个 task 有一份变量副本，变成了每个节点的 executor 才一份副本。这样的话，就可以让变量产生的副本大大减少。提高运算效率。但要注意的是，

1、能不能将一个 RDD 使用广播变量广播出去？不能，因为 RDD 是不存储数据的。可以将 RDD 的结果广播出去。

2、 广播变量只能在 Driver 端定义，不能在 Executor 端定义。

3、 在 Driver 端可以修改广播变量的值，在 Executor 端无法修改广播变量的值。

4、如果 executor 端用到了 Driver 的变量，如果不使用广播变量在 Executor 有多少 task 就有多少 Driver 端的变量副本。

5、如果 Executor 端用到了 Driver 的变量，如果使用广播变量在每个 Executor 中只有一份 Driver 端的变量副本。

6、广播的变量值不能很大。

5.Hive on Spark

Hive On Spark: 为了方便对 Hive 操作习惯了的用户，查询层还是用的 Hive，底层用 Spark 来进行计算的

部署环境:

1、将 Hadoop/etc/hadoop/下的 core-site.xml 和 Hive/conf/hive-site.xml 放到 Spark/conf/目录下

2、注意:如果 mysql 安装在 windows 系统，需要设置字符集为 latin1.

查看 hive-site.xml 配置信息里，如果没有 hive 数据库就创建一个

3、注意：Hive-On-Spark 的环境一旦配置好了以后，在这个集群上启动 Spark Shell 后进行操作会出错误

启动:

Spark/bin/spark-sql \

--master spark://node01:7077 \
--executor-memory 512m \
--total-executor-cores 2 \
--driver-class-path /export/software/mysql-connector-java-5.1.35-bin.jar

操作:

创建表 create table person(id int, name string, age int, fv int) row format delimited fields terminated by ',';

加载数据 load data local inpath "/root/person.txt" into table person;

查看数据是否加载 dfs -ls /user/hive/warehouse;

查询 select name,age,fv from person where age > 20;

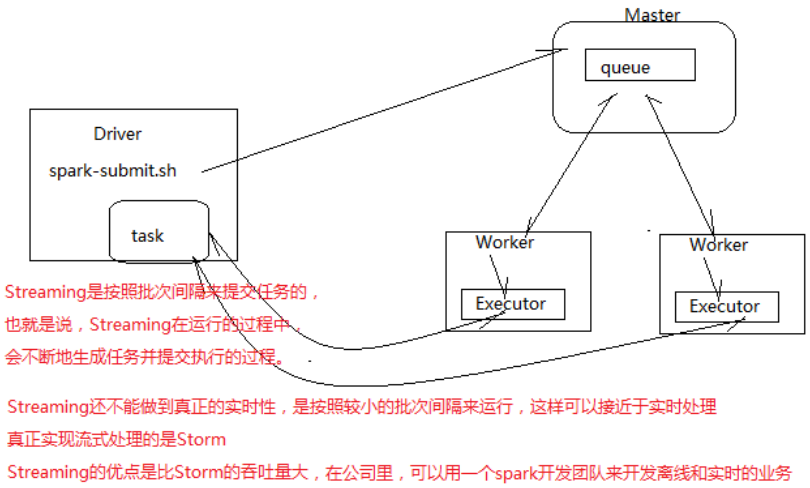
删除表 drop table person;

Spark Streaming

1.概念

Spark Streaming 类似于 Apache Storm，用于流式数据的处理。根据其官方文档介绍，Spark Streaming 有高吞吐量和容错能力强等特点。Spark Streaming 支持的数据输入源很多，例如：Kafka、Flume、Twitter、ZeroMQ 和简单的 TCP 套接字等等。数据输入后可以用 Spark 的高度抽象原语如：map、reduce、join、window 等进行运算。而结果也能保存在很多地方，如 HDFS，数据库等。另外 Spark Streaming 也能和 MLlib（机器学习）以及 Graphx 完美融合。

Streaming 是按照批次间隔来提交任务的，也就是说，Streaming 在运行过程中，会不断生成任务并提交执行的过程。
Streaming 还不能做到真正的实时性，是按照较小的批次间隔来运行，这样可以接近于实时处理，真正实现流失处理的是 Storm，Streaming 的优点是比 Storm 的吞吐量大，在公司里，可以用一个 Spark 开发团队来开发离线和实时业务。



2.Dstream

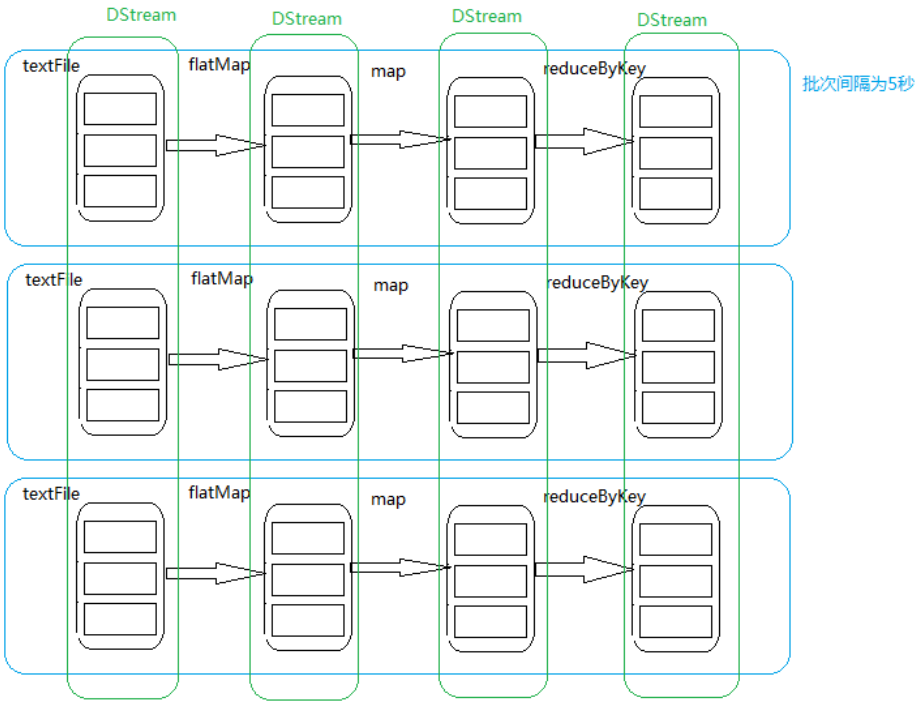
概念

DStream(Discretized Stream)是一个离散流，是 spark 基本的数据抽象，SparkStreaming 的操作对象，提供了很多原语，与 RDD 的操作一样，但内部的实现不一样，方便了我们开发。它的原语分为 transformations 和 output operations。DStream 有以下几个特性：

1.一个放了多个 DStream 的列表之内的 DStream 间是有依赖关系的

2.每隔一段时间（时间间隔）就会生成一个 RDD

3.每隔一段时间生成的 RDD 会有一个函数作用在这个 RDD 上



DStream是一个离散流，是spark基本的数据抽象，它里面包含了RDD，我们在操作DStream的过程中起始就是操作的里面的RDD，它有以下一个特性？

一个放了多个DStream的列表，并且DStream之间是有依赖关系的

- A list of other DStreams that the DStream depends on

每隔一段时间（时间间隔）就会生成一个RDD

- A time interval at which the DStream generates an RDD

每隔一段时间生成的RDD会有一个函数作用在这个RDD上

- A function that is used to generate an RDD after each time interval

相关操作

特殊的 Transformations

1.UpdateStateByKey Operation

拉取历史批次记录（checkpoint 存储），应用到当前批次当中。

1.把历史批次结果应用到当前批次来进行聚合计算的时候，相同 key 对应的 value 聚合。

（以 DStream 中的数据进行按 key 做 reduce 操作，然后对各个批次的数据进行累加）

2.1.调用时传的三个参数

* String:当前批次对应 key
* Seq[Int]:当前批次相同 key 对应 value
* Option[Int]:历史批次相同 key 对应 value

2.2.updateStateByKey 自身不能存储，所以要事先做 checkpoint，从 checkpoint 目录中取数据

2.Transform Operation

Transform 原语允许 DStream 上执行任意的 RDD-to-RDD 函数。通过该函数可以方便的扩展 Spark API。此外，MLlib（机器学习）以及 Graphx 也是通过本函数来进行结合的。

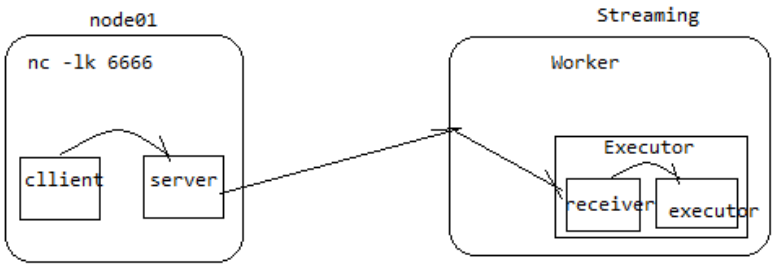
transform 可以让我们不用操作 DStream 而是操作 DStream 里的 RDD，极大丰富了我们平时调用的 api
(可扩展算子)

3.Window Operations

窗口操作是指一段时间内数据发生变化，也就是说用窗口的概念来进行结果的展示。操作窗口函数，必须设置两个重要的参数：窗口长度和滑动间隔。
窗口长度：窗口持续的时间，就是指每次展示结果的范围。
滑动间隔：执行窗口操作的窗口之间的间隔，就是指上一个窗口到下一个窗口的时间间隔。
注意：这两个参数，必须是批次间隔的倍数

窗口操作： 一段时间内数据发生变化。也就是说用窗口的概念来进行结果的展示。
操作窗口函数，必须设置两个重要的参数：窗口长度、滑动间隔
窗口长度：窗口持续的时间。就是指每次展示结果的范围。
滑动间隔：执行窗口操作的窗口之间的间隔。就是指上一个窗口到下一个窗口的时间间隔。
注意：这两个参数，必须是批次间隔的倍数

NetCat 服务流程



3.updateStateByKey(wordcount)

```
object StreamingWCACC {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("").setMaster("")
    val sc = new SparkContext(conf)
    val ssc = new StreamingContext(sc,Seconds(5))

    //获取数据
    val dStream = ssc.socketTextStream("mini1",6666)

    //进行计算
    val tups = dStream.flatMap(_.split(" ")).map((_,1))
    val res: DStream[(String, Int)] = tups.updateStateByKey(func,new HashPartitioner(3),true)

    res.print()
    ssc.start()
    ssc.awaitTermination()
  }
/**
  * String:当前批次对应 key
  * Seq[Int]:当前批次相同 key 对应 value
  * Option[Int]:历史批次相同 key 对应 value
  */
  val func = (it:Iterator[(String,Seq[Int],Option[Int])]) => {
    it.map{
      x => {
        (x._1,x._2.sum+x._3.getOrElse(0))
      }
    }
  }
}
```

4. Streaming 提供的两种获取 kafka 的方式的优缺点

receive 方式：调用了 kafka 的高阶 api,它把数据先获取到缓存中以供调用，但这样会造成数据不安全，如果想实现数据安全，可以使用 WAL，预写日志的方式来实现数据安全，但这样会影响性能以及计算的效率，而且 offset 值自动维护在 zooKeeper 中，想要更改不太方便，所以很多公司并不使用这种方式。

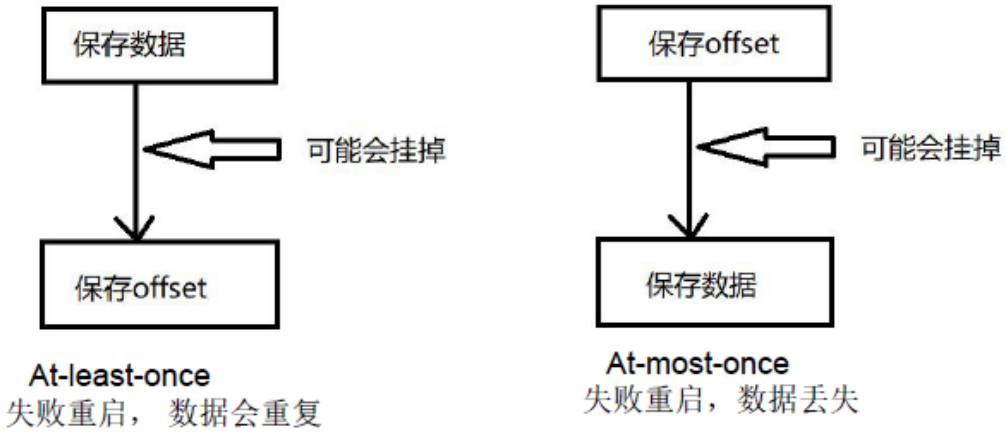
direct 方式：调用了 kafka 底层的 api,它拉取到数据就进行处理，相较 receive 效率要高，而且可以手动管理 offset 值，所以常用这种方式。

5.通过哪种方式可以手动管理 offset

1.checkpoint，简单(常用)
2.手动维护到 zooKeeper 中
3.hbase，hdfs(小文件的问题)(不常用)
4.kafka(这种方式比较新)(不常用)


streaing-kafka.md

6.为什么会出现数据丢失与数据重复问题



至少一次语义：先保存数据再保存 offset，但保存数据后发生宕机，再重启后拉取数据还是从之前的 offset 处开始，那么会再拉取到同样一份数据，造成数据的重复。
最多一次语义：先保存 offset 再保存数据，但保存 offset 后发生宕机，再重启后拉取数据会从改变后的 offset 处开始，那么变化的 offset 那一段数据找不到了，造成数据的丢失。

7.如何保证数据零丢失（怎么解决数据丢失与数据重复问题）

一次仅一次语义：
1.幂等写入：在软件开发领域，幂等写入即为同样的请求被执行一次与连续执行多次的效果是一样的，服务器的状态也是一样的，实际上就是接口的可重复调用(包括时间和空间两个维度)

mysql，一个分区对应一个数据库的连接(优化方式)dup

licate 实现了幂等写入，后期同样的 orderid 来了，不会新插入，而是更新

2.事务控制：保存数据和 offset 在同一个事务里面，要成功都成功，只有一个成功会回滚。比如用 mysql，这样的方式需要事务存储的支持

mysql，scalikejdbc(pom.xml,加入第三方依赖)

DB.localTx--开启 scalike 提供的事务机制

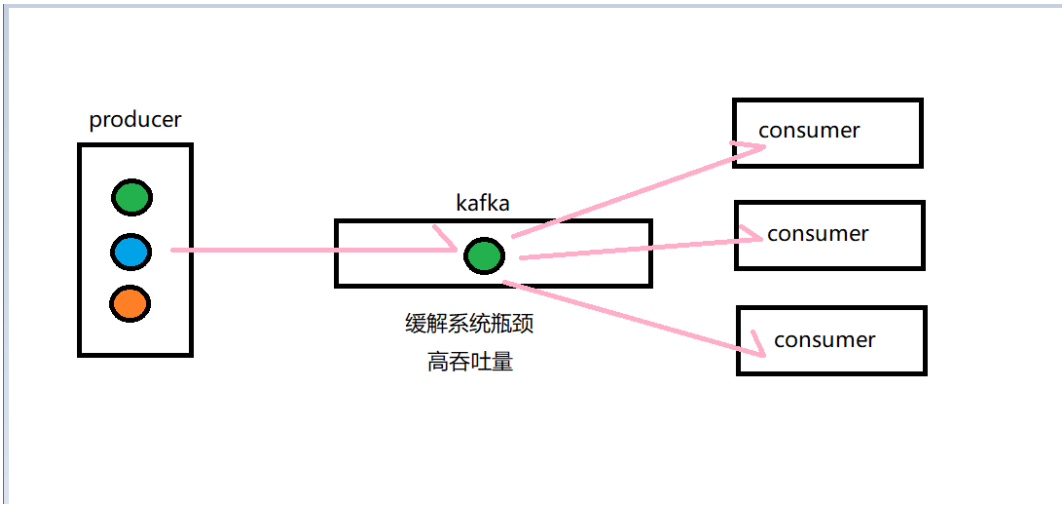
3.自己实现 Excactly-once：offset 和数据库绑定保存等

Kafka

1.概念

Kafka 是一个分布式消息队列：生产者、消费者的功能。它提供了类似于 JMS（JMS 是 Java 提供的一套技术规范）的特性，但是在设计实现上完全不同，此外它并不是 JMS 规范的实现。Kafka 只有数据存储的功能，通过多副本机制保证数据的安全性，副本的数量可以通过配置文件等更改。

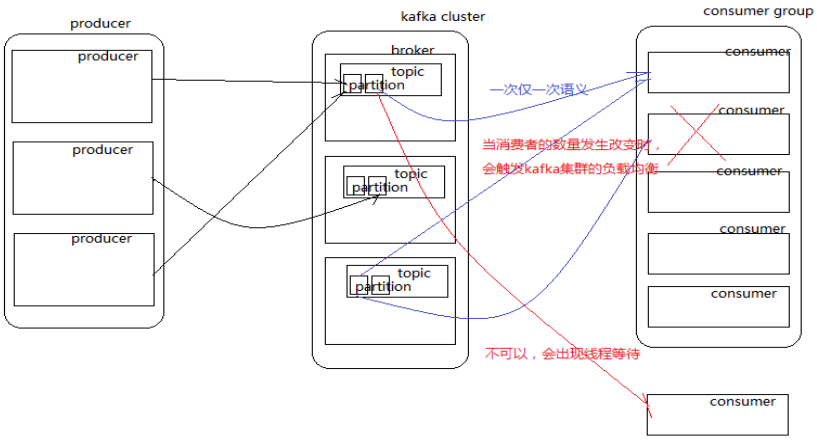
2.为什么使用 kafka



消息系统的核心作用就是三点：解耦（随时取），异步（增加 kafka）和并行（高吞吐量，多个放多个取）。
如果多个 consumer 同时从一个 producer 处拉取数据，producer 很可能承受不住，发生宕机。

1. 此时，producer 将数据传送给 kafka，kafka 有高吞吐量，，可以缓解系统瓶颈。
2. Kafka 中存放了 producer 传来的数据，consumer 何时需要可以何时拉取，实现（解耦）异构系统。
3. 数据的安全性（flume，hbase，kafka…）分布式，高可用，多副本机制。

3.Kafka 的主要组件



核心

Producer：

--生产者负责采集数据并把数据传输到 Kafka 的某个 topic 中。比如：flume、Java 后台服务、Shell 脚本、logstash

--生产者是由多个进程组成的（可以有多个生产者组成）。一个生产者可以作为一个独立的进程，可以独立的分发数据。

--多个生产者发送的数据时可以存储到同一个 topic 的同一个 partition 的

--一个生产者的数据也可以放到多个 topic 中

Kafka 集群：

--Kafka 集群可以保存多种数据类型的数据，一种数据类型可以存放到一个 topic 中，一个 Kafka 集群可以创建多个 topic

--每个 topic 可以创建一个或多个分区，分区的数量和副本的数量是在创建 topic 时指定的。在后期，某个 topic 的分区数可以重新指定（只能由少增多）。

--每个分区是由多个 segment 组成，segment 的大小是可以配置，默认是 1G。segment 里有两种类型的文件（index 和 log 文件），index 文件是存放 log 数据对应的索引，log 文件是存放数据的。

--Kafka 是具有多副本机制的，原始数据和副本数据是不可以在同一个节点中的。

Consumer Group：

--Consumer（消费者）负责消费数据：flume、SparkStreaming、Storm...

--一个 ConsumerGroup 也被称为 Consumer 集群

--新增或减少 Consumer 的数量会触发 Kafka 的负载均衡

--ConsumerGroup 可以消费一个或多个分区的数据，相反，一个分区的数据同时只能被一个 Consumer 消费

--ConsumerGroup 成员之间消费的数据各不相同，在同一个 group 中数据是不可以重

复消费（一次仅一次的语义）。

其他

- Broker ：一台 kafka 服务器就是一个 broker。一个集群由多个 broker 组成。一个 broker 可以容纳多个 topic。
- Topic ： 我们可以理解为一个队列。对数据进行归类。
- Partition：为了实现扩展性，一个非常大的 topic 可以分布到多个 broker（即服务器）上，一个 topic 可以分为多个 partition，每个 partition 是一个有序的队列。partition 中的每条消息都会被分配一个有序的 id（offset）。kafka 只保证按一个 partition 中的顺序将消息发给 consumer，不保证一个 topic 的整体（多个 partition 间）的顺序。
- Offset：kafka 的存储文件都是按照 offset.kafka 来命名，用 offset 做名字的好处是方便查找。例如你想找位于 2049 的位置，只要找到 2048.kafka 的文件即可。当然 the first offset 就是 00000000000.kafka。

一个 topic 有多个副本，一个 group 下 consumer 读取一个 topic 中数据，另一个 consumer 读取这个 topic 的副本数据，是可以的（多线程，提高吞吐量）。而同一个组里的 consumer 获取同一个 topic 中的数据，是不可以的，会发生线程等待。一次仅一次语义。
一次仅一次语义：一个 group 的一个 consumer 拉取过 topic 中的数据就不能再拉取这份数据了。
解决：改组名或者创建镜像 kafka（相当于两个 kafka 集群）

4.ISR 机制

kafka 不是完全同步，也不是完全异步，是一种 ISR 机制：
1. leader 会维护一个与其基本保持同步的 Replica 列表，该列表称为 ISR(in-sync Replica)，每个 Partition 都会有一个 ISR，而且是由 leader 动态维护
2. 如果一个 flower 比一个 leader 落后太多，或者超过一定时间未发起数据复制请求，则 leader 将其重 ISR 中移除
3. 当 ISR 中所有 Replica 都向 Leader 发送 ACK 时，leader 才 commit

既然所有 Replica 都向 Leader 发送 ACK 时，leader 才 commit，那么 flower 怎么会 leader 落后太多?
producer 往 kafka 中发送数据，不仅可以一次发送一条数据，还可以发送 message 的数组；批量发送，同步的时候批量发送，异步的时候本身就是就是批量；底层会有队列缓存起来，批量发送，对应 broker 而言，就会收到很多数据(假设 1000)，这时候 leader 发现自己有 1000 条数据，flower 只有 500 条数据，落后了 500 条数据，就把它从 ISR 中移除出去，这时候发现其他的 flower 与他的差距都很小，就等待；如果因为内存等原因，差距很大，就把它从 ISR 中移除出去。

5.Kafka 的几个问题

1、每个 topic 的分区中有多个 segment，一个分区会被分成相同大小数据数量不等的 segment，数据的生命周期就是指的 segment 的生命周期。
2、数据的存储机制：
首先是 Broker 接收到数据，将数据放到操作系统的缓存里（pagecache），
pagecache 会尽可能多的使用空闲内存，
使用 sendfile 技术尽可能多的减少操作系统和应用程序之间地重复缓存，
写数据的时候使用的是顺序写入，顺序写入的速度可达 600m/s
3、kafka 是怎么解决负载均衡的呢？
首先获取 Consumer 消费的起始分区号
然后计算出 Consumer 要消费的分区数量
用起始分区号的 hash 值%分区数
4、数据是怎么分发的？
kafka 默认调用自己的分区器（DefaultPartitioner），
当然也可以自定义分区器，需要实现 Partitioner 特质，实现 partition 方法
5、怎么保证存储数据不丢失？
kafka 的多副本机制就保证了数据的不丢失，副本数是在创建 topic 时指定的。

6.集群安装常用命令


Kafka.docx

其他

1.Yarn 工作流程

1. client 端请求 ResourceManager 进行提交任务
2. ResourceManager 和 NodeManager 进行通信
3. NodeManager 会在其中一个节点启动 ApplicationMaster，ApplicationMaster 相当于当前任务调度的管家
4. ApplicationMaster 开始向 ResourceManager 申请资源
5. ApplicationsManager 负责资源的调度，通知相应的 NodeManager 来启动 YarnChild
6. YarnChild 和 ApplicationMaster 进行通信，ApplicationMaster 对 YarnChild 进行监控

2.Spark-On-YARN（cluster 模式）

概念

cluster 模式：Driver 程序在 YARN 中运行，应用的运行结果不能在客户端显示，所以最好运行那些将结果最终保存在外部存储介质（如 HDFS、Redis、Mysql）而非 stdout 输出的应用程序，客户端的终端显示的仅是作为 YARN 的 job 的简单运行状况。

提交语句.

```
/bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master yarn \
--deploy-mode cluster \
--driver-memory 1g \
--executor-memory 1g \
--executor-cores 2 \
--queue default \
lib/spark-examples*.jar \
100
```

工作机制

Spark Driver 首先作为一个 ApplicationMaster 在 YARN 集群中启动，客户端提交给 ResourceManager 的每一个 job 都会在集群的 NodeManager 节点上分配一个唯一的 ApplicationMaster，由该 ApplicationMaster 管理全生命周期的应用。具体过程：

1. 由 client 向 ResourceManager 提交请求，并上传 jar 到 HDFS 上
这期间包括四个步骤：
a).连接到 RM
b).从 RM 的 ASM（ApplicationsManager ）中获得 metric、queue 和 resource 等信息。
c). upload app jar and spark-assembly jar
d).设置运行环境和 container 上下文（launch-container.sh 等脚本)

2. NodeManager 向 ResouceManager 申请资源，创建 Spark ApplicationMaster（每个 SparkContext 都有一个 ApplicationMaster）
3. NodeManager 启动 ApplicationMaster，并向 ResourceManager AsM 注册
4. ApplicationMaster 从 HDFS 中找到 jar 文件，启动 SparkContext、DAGscheduler 和 YARN Cluster Scheduler

5. ResourceManager 向 ResourceManager AsM 注册申请 container 资源
6. ResourceManager 通知 NodeManager 分配 Container，这时可以收到来自 ASM 关于 container 的报告。（每个 container 对应一个 executor）
7. Spark ApplicationMaster 直接和 container（executor）进行交互，完成这个分布式任务。