

目录

Kafka	4
第 1 章 Kafka 概述	5
1.1 定义	5
1.2 消息队列（Message Queue）	5
1.3 Kafka 基础架构	错误！未定义书签。
第 2 章 Kafka 快速入门	5
2.1 安装部署	6
2.2 Kafka 命令行操作	8
第 3 章 Kafka 架构深入	9
3.1 Kafka 工作流程及文件存储机制	9
3.2 Kafka 生产者	10
3.3 Kafka 消费者	13
3.4 Kafka 高效读写数据	15
3.5 Zookeeper 在 Kafka 中的作用	20
第 4 章 Kafka API	21
4.1 Producer API	21
4.2 Consumer API	23
4.3 自定义 Interceptor	25
第 5 章 Flume 对接 Kafka	28
第 6 章 Kafka 监控	29
6.1 Kafka Monitor	29
6.2 Kafka Manager	30
第 7 章 Kafka 面试题	30
7.1 面试问题	30
7.2 参考答案	33
SparkCore	32
第 1 章 RDD 概述	33
1.1 什么是 RDD	33
1.2 RDD 的属性	33
1.3 RDD 特点	33
第 2 章 RDD 编程	33
2.1 编程模型	35
2.2 RDD 的创建	36
2.3 Transformation（面试开发重点）	36
2.4 Action	47
2.5 RDD 中的函数传递	49
2.6 RDD 依赖关系	51
2.7 RDD 缓存	53
2.8 RDD CheckPoint	54
第 3 章 键值对 RDD 数据分区器	55
3.1 获取 RDD 分区	55
3.2 Hash 分区	56
3.3 Ranger 分区	56
3.4 自定义分区	56
第 4 章 数据读取与保存	57
4.1 文件类数据读取与保存	57

4.2 文件系统类数据读取与保存	59
第 5 章 RDD 编程进阶	62
5.1 累加器	62
5.2 广播变量（调优策略）	64
第 6 章 扩展	65
6.1 RDD 相关概念关系	65
SparkSQL	67
第 1 章 Spark SQL 概述	67
1.1 什么是 Spark SQL	67
1.2 Spark SQL 的特点	67
1.3 什么是 DataFrame	67
1.4 什么是 DataSet	69
第 2 章 SparkSQL 编程	69
2.1 SparkSession 新的起始点	69
2.2 DataFrame	69
2.3 DataSet	72
2.4 DataFrame 与 DataSet 的互操作	73
2.5 RDD、DataFrame、DataSet	74
2.6 IDEA 创建 SparkSQL 程序	76
2.7 用户自定义函数	76
第 3 章 SparkSQL 数据源	79
3.1 通用加载/保存方法	79
3.2 JSON 文件	80
3.3 Parquet 文件	81
3.4 JDBC	81
3.5 Hive 数据库	82
第 4 章 Spark SQL 实战	89
4.1 数据说明	90
4.2 加载数据	90
4.3 计算所有订单中每年的销售单数、销售总额	92
4.4 计算所有订单每年最大金额订单的销售额	93
4.5 计算所有订单中每年最畅销货品	94
SparkStreaming	97
第 1 章 Spark Streaming 概述	97
1.1 Spark Streaming 是什么	97
1.2 Spark Streaming 特点	97
1.3 SparkStreaming 架构	98
第 2 章 Dstream 入门	98
2.1 WordCount 案例实操	98
2.2 WordCount 解析	99
第 3 章 Dstream 创建	100
3.1 文件数据源	100
3.2 RDD 队列（了解）	101
3.3 Socket 数据源	103
3.4 Kafka 数据源（重点）	104
3.5 Kafka 数据源高级开发（重点）	106
3.6 offset 管理	109

3.7 从 checkpoint 恢复数据	114
3.8 多个 receiver 源 union	115
第 4 章 DStream 转换	115
4.1 无状态转化操作	115
4.2 有状态转化操作（重点）	116
4.3 其他重要操作	120
第 5 章 DStream 输出	120
5.1 输出到 HDFS	121
Spark 内核	123
第 1 章 Spark 内核概述	123
1.1 Spark 核心组件回顾	123
1.2 Spark 通用运行流程概述	123
第 2 章 Spark 部署模式	124
2.1 Standalone 模式运行机制	124
2.2 YARN 模式运行机制	126
第 3 章 Spark 通讯架构	127
3.1 Spark 通信架构概述	127
3.2 Spark 通讯架构解析	128
第 4 章 Spark 任务调度机制	129
4.1 Spark 任务提交流程	129
4.2 Spark 任务调度概述	130
4.3 Spark Stage 级调度	132
4.4 Spark Task 级调度	133
第 5 章 Spark Shuffle 解析	137
5.1 Shuffle 的核心要点	137
5.2 HashShuffle 解析	138
5.3 SortShuffle 解析	140
第 6 章 Spark 内存管理	142
6.1 堆内和堆外内存规划	142
6.2 内存空间分配	143
6.3 存储内存管理	146
6.4 执行内存管理	149
第 7 章 Spark 核心组件解析	150
7.1 BlockManager 数据存储与管理机制	150
7.2 Spark 共享变量底层实现	151
Spark 调优	157
第 1 章 Spark 性能调优	157
1.1 常规性能调优	157
1.2 算子调优	160
1.3 Shuffle 调优	164
1.4 JVM 调优	165
第 2 章 Spark 数据倾斜	166
2.1 解决方案一：聚合原数据	167
2.2 解决方案二：过滤导致倾斜的 key	167
2.3 解决方案三：提高 shuffle 操作中的 reduce 并行度	167
2.4 解决方案四：使用随机 key 实现双重聚合	167
2.5 解决方案五：将 reduce join 转换为 map join	168

2.6 解决方案六：sample 采样对倾斜 key 单独进行 join	169
2.7 解决方案七：使用随机数以及扩容进行 join	170
第 3 章 Spark Troubleshooting	171
3.1 故障排除一：控制 reduce 端缓冲大小以避免 OOM	171
3.2 故障排除二：JVM GC 导致的 shuffle 文件拉取失败	171
3.3 故障排除三：解决各种序列化导致的报错	171
3.4 故障排除四：解决算子函数返回 NULL 导致的问题	171
3.5 故障排除五：解决 YARN-CLIENT 模式导致的网卡流量激增问题	171
3.6 故障排除六：解决 YARN-CLUSTER 模式的 JVM 栈内存溢出无法执行问题	172
3.7 故障排除七：解决 SparkSQL 导致的 JVM 栈内存溢出	173
3.8 故障排除八：持久化与 checkpoint 的使用	173

Kafka

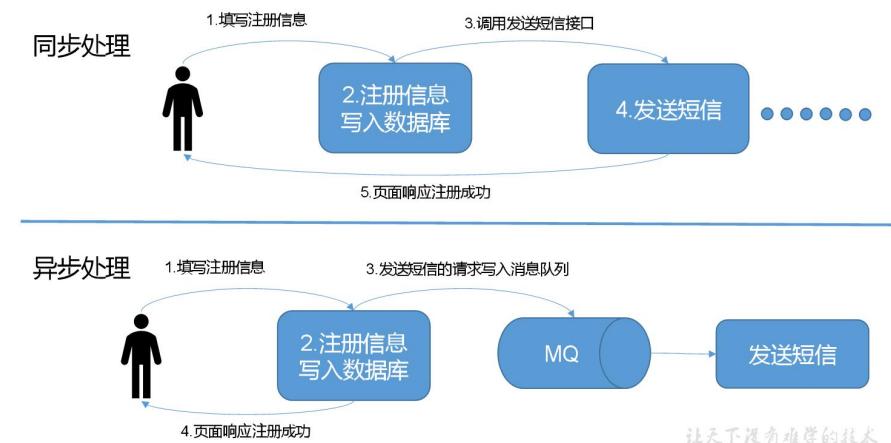
第 1 章 Kafka 概述

1.1 定义

Kafka 是一个分布式的基于发布/订阅模式的消息队列，主要应用于大数据实时处理领域。

1.2 消息队列（Message Queue）

1.2.1 传统消息队列的应用场景

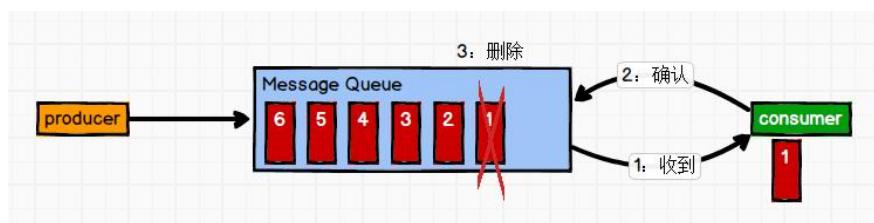


1.2.2 消息队列的两种模式

(1) 点对点模式（一对多，消费者主动拉取数据，消息收到后消息清除）

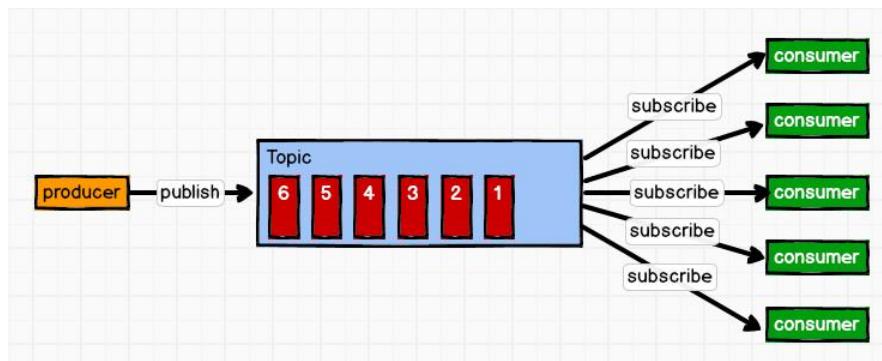
消息生产者生产消息发送到 Queue 中，然后消息消费者从 Queue 中取出并且消费消息。

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。



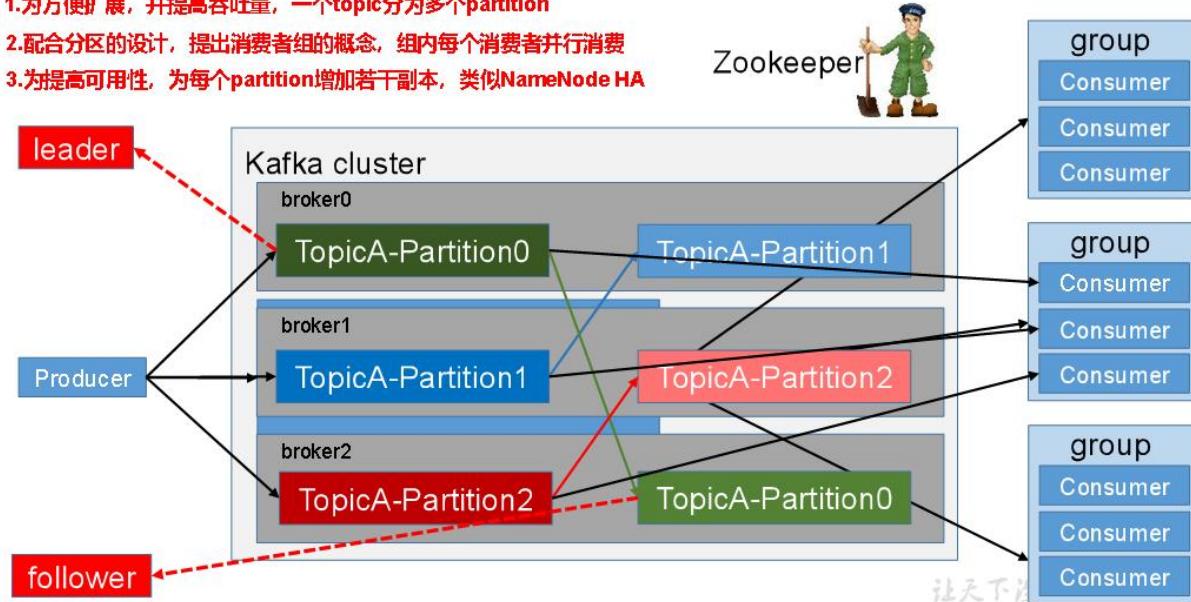
(2) 发布/订阅模式（一对多，消费者消费数据之后不会清除消息）

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。



1.3 Kafka 基础架构

1. 为方便扩展，并提高吞吐量，一个topic分为多个partition
2. 配合分区的设计，提出消费者组的概念，组内每个消费者并行消费
3. 为提高可用性，为每个partition增加若干副本，类似NameNode HA



- 1) Producer : 消息生产者，就是向 kafka broker 发消息的客户端；
- 2) Consumer : 消息消费者，向 kafka broker 取消息的客户端；
- 3) Consumer Group (CG) : 消费者组，由多个 consumer 组成。消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个消费者消费；消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。
- 4) 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。
- 5) Topic : 可以理解为一个队列，生产者和消费者面向的都是一个 topic；topic 由多个 partition 组成。
- 6) 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；
- 7) Replica: 副本，为保证集群中的某个节点发生故障时，该节点上的 partition 数据不丢失，且 kafka 仍然能够继续工作，kafka 提供了副本机制，一个 topic 的每个分区都有若干个副本，一个 leader 和若干个 follower。
- 8) leader: 每个分区多个副本的“主”，生产者发送数据的对象，以及消费者消费数据的对象都是 leader。
- 9) follower: 每个分区多个副本中的“从”，实时从 leader 中同步数据，保持和 leader 数据的同步。leader 发生故障时，某个 follower 会成为新的 follower。

第 2 章 Kafka 快速入门

2.1 安装部署

2.1.1 集群规划

hadoop102	hadoop103	hadoop104
zk	zk	zk
kafka	kafka	kafka

2.1.2 jar 包下载

<http://kafka.apache.org/downloads.html>

HOME
INTRODUCTION
QUICKSTART
USE CASES
DOCUMENTATION
PERFORMANCE
POWERED BY
PROJECT INFO
ECOSYSTEM

Download

0.11.0.0 is the latest release. The current stable version is 0.11.0.0.

You can verify your download by following these [procedures](#) and using these [KEYS](#).

0.11.0.0

- Released June 28, 2017
- [Release Notes](#)
- Source download: [kafka-0.11.0.0-src.tgz](#) (asc, md5)
- Binary downloads:
 - Scala 2.11 - [kafka_2.11-0.11.0.0.tgz](#) (asc, md5)
 - Scala 2.12 - [kafka_2.12-0.11.0.0.tgz](#) (asc, md5)

2.1.3 集群部署

1) 解压安装包

```
[atguigu@hadoop102 software]$ tar -zvxf kafka_2.11-0.11.0.0.tgz -C /opt/module/
```

2) 修改解压后的文件名称

```
[atguigu@hadoop102 module]$ mv kafka_2.11-0.11.0.0/ kafka
```

3) 在 /opt/module/kafka 目录下创建 logs 文件夹

```
[atguigu@hadoop102 kafka]$ mkdir logs
```

4) 修改配置文件

```
[atguigu@hadoop102 kafka]$ cd config/
```

```
[atguigu@hadoop102 config]$ vi server.properties
```

输入以下内容：

#broker 的全局唯一编号，不能重复

broker.id=0

#删除 topic 功能使能

delete.topic.enable=true

#处理网络请求的线程数量

num.network.threads=3

#用来处理磁盘 IO 的现成数量

num.io.threads=8

#发送套接字的缓冲区大小

socket.send.buffer.bytes=102400

#接收套接字的缓冲区大小

socket.receive.buffer.bytes=102400

#请求套接字的缓冲区大小

socket.request.max.bytes=104857600

#kafka 运行日志存放的路径

log.dirs=/opt/module/kafka/logs

#topic 在当前 broker 上的分区个数

num.partitions=1

#用来恢复和清理 data 下数据的线程数量

num.recovery.threads.per.data.dir=1

#segment 文件保留的最长时间，超时将被删除

log.retention.hours=168

#配置连接 Zookeeper 集群地址

```
zookeeper.connect=hadoop102:2181,hadoop103:2181,hadoop104:2181
```

5) 配置环境变量

```
[atguigu@hadoop102 module]$ sudo vi /etc/profile
```

```
#KAFKA_HOME  
export KAFKA_HOME=/opt/module/kafka  
export PATH=$PATH:$KAFKA_HOME/bin
```

```
[atguigu@hadoop102 module]$ source /etc/profile
```

6) 分发安装包

```
[atguigu@hadoop102 module]$ xsync kafka/
```

注意：分发之后记得配置其他机器的环境变量

7) 分别在 hadoop103 和 hadoop104 上修改配置文件 /opt/module/kafka/config/server.properties 中的 broker.id=1、broker.id=2

注意： broker.id 不得重复

8) 启动集群

依次在 hadoop102、hadoop103、hadoop104 节点上启动 kafka

```
[atguigu@hadoop102 kafka]$ bin/kafka-server-start.sh -daemon config/server.properties  
[atguigu@hadoop103 kafka]$ bin/kafka-server-start.sh -daemon config/server.properties  
[atguigu@hadoop104 kafka]$ bin/kafka-server-start.sh -daemon config/server.properties
```

9) 关闭集群

```
[atguigu@hadoop102 kafka]$ bin/kafka-server-stop.sh stop  
[atguigu@hadoop103 kafka]$ bin/kafka-server-stop.sh stop  
[atguigu@hadoop104 kafka]$ bin/kafka-server-stop.sh stop
```

10) kafka 群起脚本

```
for i in `cat /opt/module/hadoop-2.7.2/etc/hadoop/slaves`  
do  
echo "===== $i ====="  
ssh $i 'source /etc/profile&&/opt/module/kafka_2.11-0.11.0.2/bin/kafka-server-start.sh -daemon /opt/module/kafka_2.11-0.11.0.2/config/server.properties'  
echo $?  
done
```

2.2 Kafka 命令行操作

1) 查看当前服务器中的所有 topic

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper hadoop102:2181 --list
```

2) 创建 topic

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper hadoop102:2181 \  
--create --replication-factor 3 --partitions 1 --topic first
```

选项说明：

- topic 定义 topic 名
- replication-factor 定义副本数
- partitions 定义分区数

3) 删除 topic

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper hadoop102:2181 \  
--delete --topic first
```

需要 server.properties 中设置 delete.topic.enable=true，否则只是标记删除。

4) 发送消息

```
[atguigu@hadoop102 kafka]$ bin/kafka-console-producer.sh \  
--broker-list hadoop102:9092 --topic first  
>hello world  
>atguigu atguigu
```

5) 消费消息

```
[atguigu@hadoop103 kafka]$ bin/kafka-console-consumer.sh \  
--topic first --from-beginning
```

```
--bootstrap-server hadoop102:9092 --from-beginning --topic first
```

```
[atguigu@hadoop103 kafka]$ bin/kafka-console-consumer.sh \  
--bootstrap-server hadoop102:9092 --from-beginning --topic first  
--from-beginning: 会把主题中以往所有的数据都读取出来。
```

6) 查看某个 Topic 的详情

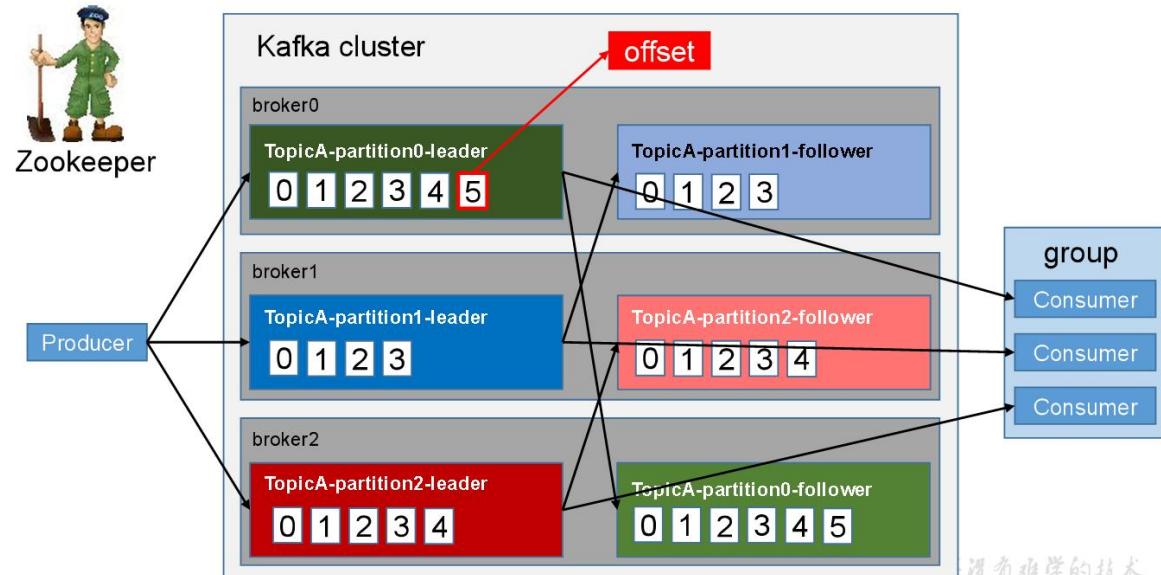
```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper hadoop102:2181 \  
--describe --topic first
```

7) 修改分区数

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper hadoop102:2181 --alter --topic first --partitions 6
```

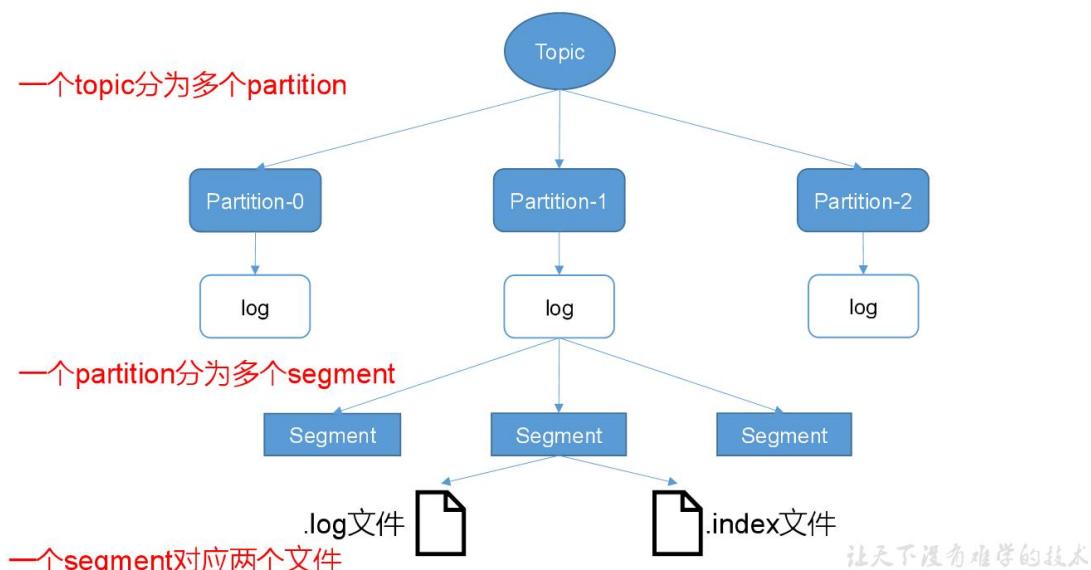
第 3 章 Kafka 架构深入

3.1 Kafka 工作流程及文件存储机制



Kafka 中消息是以 **topic** 进行分类的，生产者生产消息，消费者消费消息，都是面向 **topic** 的。

topic 是逻辑上的概念，而 **partition** 是物理上的概念，每个 **partition** 各自对应一个 **log** 文件，该 **log** 文件中存储的就是 **producer** 生产的数据。**Producer** 生产的数据会被不断追加到该 **log** 文件末端，且每条数据都有自己的 **offset**。消费者组中的每个消费者，都会实时记录自己消费到了哪个 **offset**，以便出错恢复时，从上次的位置继续消费。

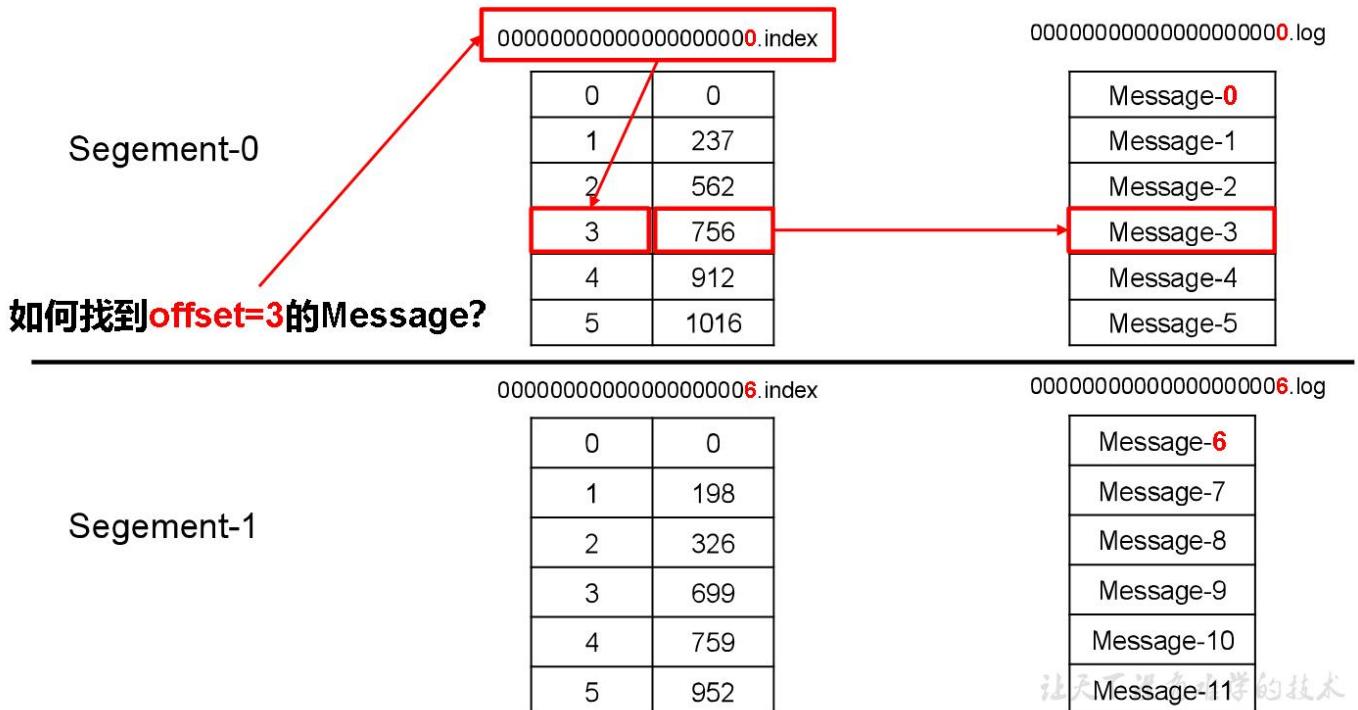


由于生产者生产的消息会不断追加到 **log** 文件末尾，为防止 **log** 文件过大导致数据定位效率低下，Kafka 采取了

分片和索引机制，将每个 partition 分为多个 segment。每个 segment 对应两个文件——“.index”文件和“.log”文件。这些文件位于一个文件夹下，该文件夹的命名规则为：topic 名称+分区序号。例如，first 这个 topic 有三个分区，则其对应的文件夹为 first-0,first-1,first-2。

```
00000000000000000000000000000000.index
00000000000000000000000000000000.log
00000000000000000000000000000000170410.index
00000000000000000000000000000000170410.log
00000000000000000000000000000000239430.index
00000000000000000000000000000000239430.log
```

index 和 log 文件以当前 segment 的第一条消息的 offset 命名。下图为 index 文件和 log 文件的结构示意图。



“.index”文件存储大量的索引信息，“.log”文件存储大量的数据，索引文件中的元数据指向对应数据文件中 message 的物理偏移地址。

3.2 Kafka 生产者

3.2.1 分区策略

1) 分区的原因

(1) 方便在集群中扩展，每个 Partition 可以通过调整以适应它所在的机器，而一个 topic 又可以有多个 Partition 组成，因此整个集群就可以适应任意大小的数据了；

(2) 可以提高并发，因为可以以 Partition 为单位读写了。

2) 分区的原则

我们需要将 producer 发送的数据封装成一个 **ProducerRecord** 对象。

```
ProducerRecord(@NotNull String topic, Integer partition, Long timestamp, String key, String value, @Nullable Iterable<Header> headers)
ProducerRecord(@NotNull String topic, Integer partition, Long timestamp, String key, String value)
ProducerRecord(@NotNull String topic, Integer partition, String key, String value, @Nullable Iterable<Header> headers)
ProducerRecord(@NotNull String topic, Integer partition, String key, String value)
ProducerRecord(@NotNull String topic, String key, String value)
ProducerRecord(@NotNull String topic, String value)
```

(1) 指明 partition 的情况下，直接将指明的值直接作为 partition 值；

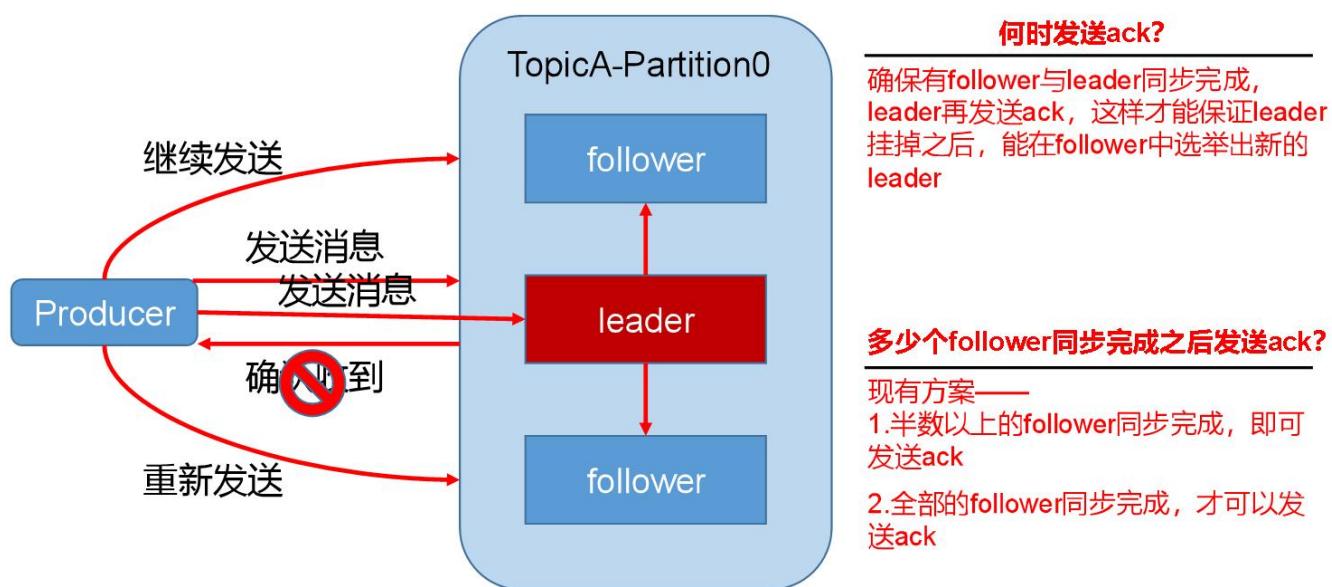
(2) 没有指明 partition 值但有 key 的情况下，将 key 的 hash 值与 topic 的 partition 数进行取余得到 partition 值；

(3) 既没有 partition 值又没有 key 值的情况下，第一次调用时随机生成一个整数（后面每次调用在这个整数

上自增），将这个值与 topic 可用的 partition 总数取余得到 partition 值，也就是常说的 round-robin 算法。

3.2.2 数据可靠性保证

为保证 producer 发送的数据，能可靠的发送到指定的 topic，topic 的每个 partition 收到 producer 发送的数据后，都需要向 producer 发送 ack（acknowledgement 确认收到），如果 producer 收到 ack，就会进行下一轮的发送，否则重新发送数据。



1) 副本数据同步策略

	方案	优点	缺点
第一种	半数以上完成同步，就发送 ack	延迟低	选举新的 leader 时，容忍 n 台节点的故障，需要 $2n+1$ 个副本
第二种	全部完成同步，才发送 ack	选举新的 leader 时，容忍 n 台节点的故障，需要 $n+1$ 个副本	延迟高

Kafka 选择了第二种方案，原因如下：

- 同样为了容忍 n 台节点的故障，第一种方案需要 $2n+1$ 个副本，而第二种方案只需要 $n+1$ 个副本，而 Kafka 的每个分区都有大量的数据，第一种方案会造成大量数据的冗余。
- 虽然第二种方案的网络延迟会比较高，但网络延迟对 Kafka 的影响较小。

个人理解：

一个 topic 的一个分区在一个节点中只有一个副本（只有一组数据），第一种方案可以表述为容忍 n 台节点故障需要 $2n+1$ 个节点，假设方案 1 也为 $n+1$ 个节点，此时只剩下 1 个可用的节点，不符合半数以上的规则；第二种方案只需要 $n+1$ 个节点；

2) ISR

采用第二种方案之后，设想以下情景：leader 收到数据，所有 follower 都开始同步数据，但有一个 follower，因为某种故障，迟迟不能与 leader 进行同步，那 leader 就要一直等下去，直到它完成同步，才能发送 ack。这个问题怎么解决呢？

Leader 维护了一个动态的 in-sync replica set (ISR)，意为和 leader 保持同步的 follower 集合。当 ISR 中的 follower 完成数据的同步之后，leader 就会给 follower 发送 ack。如果 follower 长时间未向 leader 同步数据，则该 follower 将被踢出 ISR，该时间阈值由 `replica.lag.time.max.ms` 参数设定。Leader 发生故障之后，就会从 ISR 中选举新的 leader。当被踢出的 follower 同步数据到最新的时候，就可以重新加入 ISR 了。

3) ack 应答机制

对于某些不太重要的数据，对数据的可靠性要求不是很高，能够容忍数据的少量丢失，所以没必要等 ISR 中的 follower 全部接收成功。

所以 Kafka 为用户提供了三种可靠性级别，用户根据对可靠性和延迟的要求进行权衡，选择以下的配置。

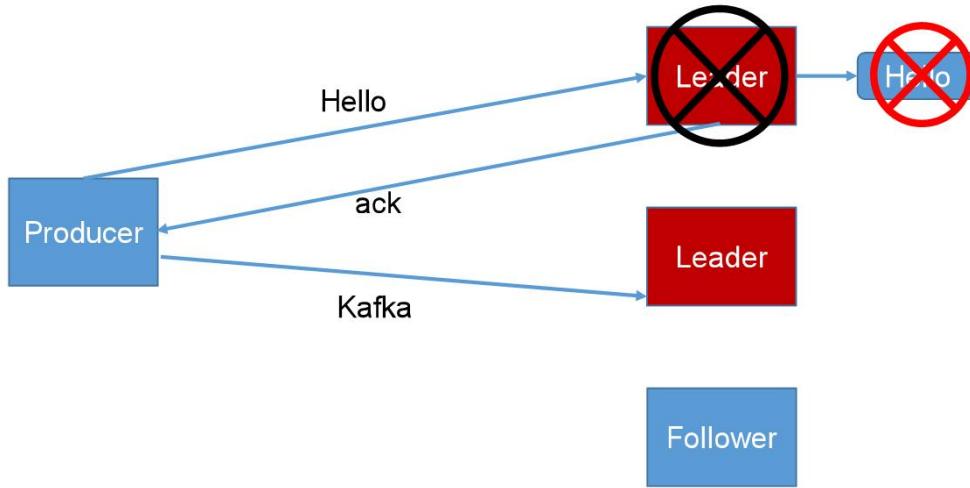
acks 参数配置:

acks:

0: producer 不等待 broker 的 ack, 这一操作提供了一个最低的延迟, broker 一接收到还没有写入磁盘就已经返回, 当 broker 故障时有可能丢失数据;

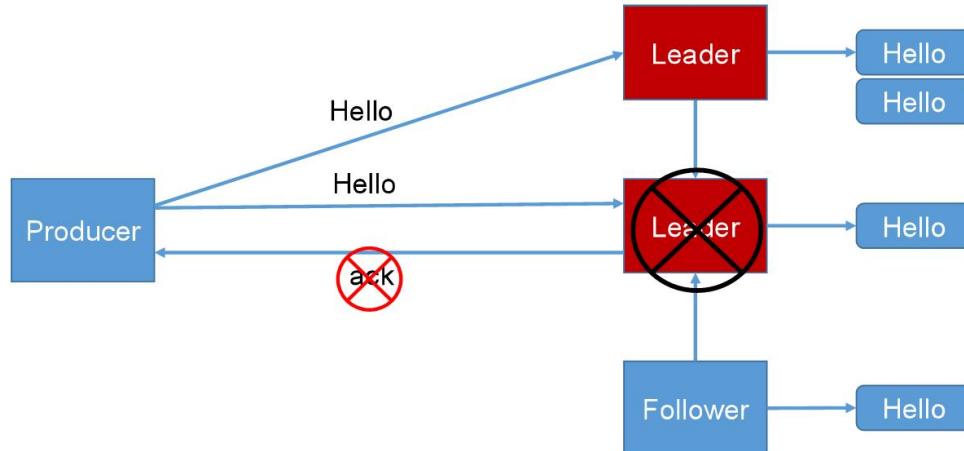
1: producer 等待 broker 的 ack, partition 的 leader 落盘成功后返回 ack, 如果在 follower 同步成功之前 leader 故障, 那么将会丢失数据;

Acks=1 丢失案例:



-1 (all) : producer 等待 broker 的 ack, partition 的 leader 和 follower 全部落盘成功后才返回 ack。但是如果在 follower 同步完成后, broker 发送 ack 之前, leader 发生故障, 那么会造成数据重复。

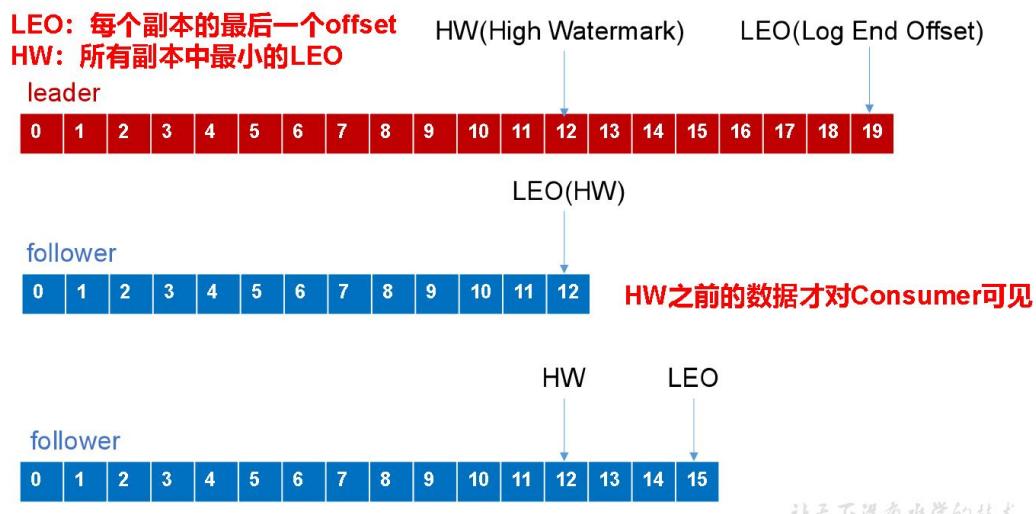
ack=-1 数据重复案例:



4) 故障处理细节

HW: 所有副本中最小的 LEO

LEO: 副本的最后一个 offset



(1) follower 故障

follower发生故障后会被临时踢出 ISR，待该 follower 恢复后，follower 会读取本地磁盘记录的上次的 HW，并将 log 文件高于 HW 的部分截取掉，从 HW 开始向 leader 进行同步。等该 follower 的 LEO 大于等于该 Partition 的 HW，即 follower 追上 leader 之后，就可以重新加入 ISR 了。

(2) leader 故障

leader发生故障之后，会从 ISR 中选出一个新的 leader，之后，为保证多个副本之间的数据一致性，其余的 follower 会先将各自的 log 文件高于 HW 的部分截掉，然后从新的 leader 同步数据。

注意：这只能保证副本之间的数据一致性，并不能保证数据不丢失或者不重复。

3.2.3 Exactly Once 语义

对于某些比较重要的消息，我们需要保证 exactly once 语义，即保证每条消息被发送且仅被发送一次。

在 0.11 版本之后，Kafka 引入了幂等性机制(idempotent)，配合 acks = -1 时的 at least once 语义，实现了 producer 到 broker 的 exactly once 语义。

idempotent + at least once = exactly once

使用时，只需将 enable.idempotence 属性设置为 true，kafka 自动将 acks 属性设为-1。

3.2.4 单 partition 有序

当先后两条消息发送时，前一条消息失败，后一条消息成功，然后失败的消息重试后成功，这样会造成乱序。为了解决重试机制引起的消息乱序为实现 Producer 的幂等性，Kafka 引入了 Producer ID(即 PID)和 Sequence Number。对于每个 PID，该 Producer 发送消息的每个<Topic, Partition>都对应一个单调递增的 Sequence Number。

同样，Broker 端也会为每个<PID, Topic, Partition>维护一个序号，并且每 Commit 一条消息时将其对应序号递增。

- 对于接收的每条消息，如果其序号比 Broker 维护的序号大一，则 Broker 会接受它，否则将其丢弃
- 如果消息序号比 Broker 维护的序号差值比一大，说明中间有数据尚未写入，即乱序，此时 Broker 拒绝该消息
- 如果消息序号小于等于 Broker 维护的序号，说明该消息已被保存，即为重复消息，Broker 直接丢弃该消息
- 发送失败后会重试，这样可以保证每个消息都被发送到 broker

消费者从 partition 中取出来数据的时候，也一定是有顺序的。到这里顺序还是 ok 的，没有错乱。但是消费者这里还是可能会有多个线程来并发来处理消息，因为如果消费者是单线程消费数据，那么这个吞吐量太低了。而多个线程并发的话，顺序可能就乱掉了。

解决方案：

消费者端创建多个内存队列，具有相同 key 的数据都进入到同一个内存队列；然后每个线程分别消费一个内存队列即可，这样就能保证顺序性。

3.3 Kafka 消费者

3.3.1 消费方式

consumer 采用 pull (拉) 模式从 broker 中读取数据。

push (推) 模式很难适应消费速率不同的消费者，因为消息发送速率是由 broker 决定的。它的目标是尽可能以

最快速度传递消息，但是这样很容易造成 consumer 来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而 pull 模式则可以根据 consumer 的消费能力以适当的速率消费消息。

pull 模式不足之处是，如果 kafka 没有数据，消费者可能会陷入循环中，一直返回空数据。针对这一点，Kafka 的消费者在消费数据时会传入一个时长参数 **timeout**，如果当前没有数据可供消费，consumer 会等待一段时间之后再返回，这段时长即为 timeout。

3.3.2 分区分配策略

一个 consumer group 中有多个 consumer，一个 topic 有多个 partition，所以必然会涉及到 partition 的分配问题，即确定那个 partition 由哪个 consumer 来消费。

Kafka 有三种分配策略，分别是 roundrobin, range 和 StickyAssignor

1) Roundrobin: 轮询策略

把主题和分区组成 **topicAndPartition** 列表，再把列表按照 **hashcode** 排序，轮询分配给消费者。

加入按照 **hashCode** 排序完的 **topic-partitions** 组依次为 T1-5, T1-3, T1-0, T1-8, T1-2, T1-1, T1-4, T1-7, T1-6, T1-9 我们的消费者线程排序为 C1-0, C1-1, C2-0, C2-1，最后分区分配的结果为：

C1-0 将消费 T1-5, T1-2, T1-6 分区；

C1-1 将消费 T1-3, T1-1, T1-9 分区；

C2-0 将消费 T1-0, T1-4 分区；

C2-1 将消费 T1-8, T1-7 分区；

弊端：3 个消费者：C0、C1 和 C2，集群中有 3 个主题：t0、t1 和 t2，这 3 个主题分别有 1、2、3 个分区，也就是说集群中有 t0p0、t1p0、t1p1、t2p0、t2p1、t2p2 这 6 个分区

消费者 C0 订阅了主题 t0，消费者 C1 订阅了主题 t0 和 t1，消费者 C2 订阅了主题 t0、t1 和 t2

消费者 C0: t0p0

消费者 C1: t1p0

消费者 C2: t1p1、t2p0、t2p1、t2p2

不是最优解（不均衡），t1p1 给 C1 才是最优的

2) range: 范围策略

假设 $n = \text{分区数}/\text{消费者数量}$, $m = \text{分区数} \% \text{消费者数量}$, 那么前 m 个消费者每个分配 $n+1$ 个分区，后面的（消费者数量- m ）个消费者每个分配 n 个分区。

排完序的分区将会是 0, 1, 2, 3, 4, 5, 6, 7, 8, 9；消费者线程排完序将会是 C1-0, C2-0, C2-1。

C1-0 将消费 0, 1, 2, 3 分区

C2-0 将消费 4, 5, 6 分区

C2-1 将消费 7, 8, 9 分区

弊端：假如我们有 2 个主题(T1 和 T2)，分别有 10 个分区，那么最后分区分配的结果看起来是这样的：

C1-0 将消费 T1 主题的 0, 1, 2, 3 分区以及 T2 主题的 0, 1, 2, 3 分区

C2-0 将消费 T1 主题的 4, 5, 6 分区以及 T2 主题的 4, 5, 6 分区

C2-1 将消费 T1 主题的 7, 8, 9 分区以及 T2 主题的 7, 8, 9 分区

可以看出，C1-0 消费者线程比其他消费者线程多消费了 2 个分区，这就是 Range strategy 的一个很明显的弊端。

3) StickyAssignor (0.11.x 版本开始引入)

特点 1：轮训(组内消费者消费同样主题，和 RoundRobinAssignor 类似

假设消费组内有 3 个消费者：C0、C1 和 C2，它们都订阅了 4 个主题：t0、t1、t2、t3，并且每个主题有 2 个分区，也就是说整个消费组订阅了 t0p0、t0p1、t1p0、t1p1、t2p0、t2p1、t3p0、t3p1 这 8 个分区

消费者 C0: t0p0、t1p1、t3p0

消费者 C1: t0p1、t2p0、t3p1

消费者 C2: t1p0、t2p1

特点 2：最优配置(消费者和分区数对应不均衡时体现)

3 个消费者：C0、C1 和 C2，集群中有 3 个主题：t0、t1 和 t2，这 3 个主题分别有 1、2、3 个分区，也就是说集群中有 t0p0、t1p0、t1p1、t2p0、t2p1、t2p2 这 6 个分区，消费者 C0 订阅了主题 t0，消费者 C1 订阅了主题 t0 和 t1，消费者 C2 订阅了主题 t0、t1 和 t2，消除了上面 RoundRobinAssignor 的弊端

消费者 C0: t0p0

消费者 C1: t1p0、t1p1

消费者 C2: t2p0、t2p1、t2p2

特点 3: 分配尽可能的与上次分配的保持相同(一个消费者挂了重新分区体现)

针对上面“特点 1”中情况, C1 挂了, 重新分配, RoundRobinAssignor 和 StickyAssignor 对比:

消费者 C0: t0p0、t1p0、t2p0、t3p0

消费者 C2: t0p1、t1p1、t2p1、t3p12

原有的保持不变, 挂的开始轮训分给 C0、C2

消费者 C0: t0p0、t1p1、t3p0、t2p0

消费者 C2: t1p0、t2p1、t0p1、t3p1

针对上面“特点 2”中情况, C1 挂了, 重新分配, RoundRobinAssignor 和 StickyAssignor 对比:

消费者 C1: t0p0、t1p1

消费者 C2: t1p0、t2p0、t2p1、t2p2

原有的保持不变, 采用最佳分配(比另外两者分配策略而言显得更加的优异)

消费者 C1: t1p0、t1p1、t0p0

消费者 C2: t2p0、t2p1、t2p2

3.3.3 offset 的维护

由于 consumer 在消费过程中可能会出现断电宕机等故障, consumer 恢复后, 需要从故障前的位置的继续消费, 所以 consumer 需要实时记录自己消费到了哪个 offset, 以便故障恢复后继续消费。

Kafka 0.9 版本之前, consumer 默认将 offset 保存在 Zookeeper 中, 从 0.9 版本开始, consumer 默认将 offset 保存在 Kafka 一个内置的 topic 中, 该 topic 为 `_consumer_offsets`。

3.3.4 实现精确一次消费

从 kafka 的消费机制, 我们可以得到是否能够精确的消费关键在 **消费进度信息的准确性**, 如果能够保证消费进度的准确性, 也就保证了消费数据的准确性

数据有状态: 可以根据数据信息进行确认数据是否重复消费, 这时候可以使用手动提交的最少一次消费语义实现, 即使消费的数据有重复, 可以通过状态进行数据去重, 以达到幂等的效果

存储数据容器具备幂等性: 在数据存入的容器具备天然的幂等(比如 ElasticSearch 的 put 操作具备幂等性, 相同的数据多次执行 Put 操作和一次执行 Put 操作的结果是一致的), 这样的场景也可以使用手动提交的最少一次消费语义实现, 由存储数据端来进行数据去重

数据无状态, 并且存储容器不具备幂等: 这种场景需要自行控制 offset 的准确性, 今天文章主要说明这种场景下的处理方式, 这里数据不具备状态, 存储使用关系型数据库, 比如 MySQL。

这里简单说明一下实现思路

1) 利用 consumer api 的 seek 方法可以指定 offset 进行消费, 在启动消费者时查询数据库中记录的 offset 信息, 如果是第一次启动, 那么数据库中将没有 offset 信息, 需要进行消费的元数据插入, 然后从 offset=0 开始消费。

2) 关系型数据库具备事务的特性, 当数据入库时, 同时也将 offset 信息更新, 借用关系型数据库事务的特性保证数据入库和修改 offset 记录这两个操作是在同一个事务中进行。

3) 使用 ConsumerRebalanceListener 来完成在分配分区时和 Relalance 时作出相应的处理逻辑。

设置 kafka 信息表

```
create table kafka_info(
    topic_group_partition varchar(32) primary key, //主题+组名+分区号 这里冗余设计方便通过这个主键进行更新
提升效率
    topic_group varchar(30), //主题和组名
    partition_num tinyint, //分区号
    offsets bigint default 0 //offset 信息
);
```

代码实现

```
package com.huawei.kafka.consumer;

import com.alibaba.fastjson.JSON;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.TopicPartition;
import java.sql.Connection;
```

```

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.time.Duration;
import java.util.*;

/**
 * @author: xuqiangnj@163.com
 * @date: 2019/5/3 14:36
 * @description:精确一次消费实现
 */
public class AccurateConsumer {

    private static final Properties props = new Properties();
    private static final String GROUP_ID = "Test";
    static {
        props.put("bootstrap.servers", "192.168.142.139:9092");
        props.put("group.id", GROUP_ID);
        props.put("enable.auto.commit", false);//注意这里设置为手动提交方式
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    }
    final KafkaConsumer<String, String> consumer;

    //用于记录每次消费时每个 partition 的最新 offset
    private Map<TopicPartition, Long> partitionOffsetMap;
    //用于缓存接受消息，然后进行批量入库
    private List<Message> list;
    private volatile boolean isRunning = true;
    private final String topicName;
    private final String topicNameAndGroupId;

    public AccurateConsumer(String topicName) {
        this.topicName = topicName;
        topicNameAndGroupId = topicName + "_" + GROUP_ID;
        consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList(topicName), new HandleRebalance());
        list = new ArrayList<>(100);
        partitionOffsetMap = new HashMap<>();
    }

    //这里使用异步提交和同步提交的组合方式
    public void receiveMsg() {
        try {
            while (isRunning) {
                ConsumerRecords<String, String> consumerRecords = consumer.poll(Duration.ofSeconds(1));
                if (!consumerRecords.isEmpty()) {
                    for (TopicPartition topicPartition : consumerRecords.partitions()) {
                        List<ConsumerRecord<String, String>> records = consumerRecords.records(topicPartition);
                        for (ConsumerRecord<String, String> record : records) {
                            //使用 fastjson 将记录中的值转换为 Message 对象，并添加到 list 中
                            list.addAll(JSON.parseArray(record.value(), Message.class));
                        }
                    }
                    //将 partition 对应的 offset 信息添加到 map 中，入库时将 offset-partition 信息一起入库
                    //记住这里一定要加 1，因为下次消费的位置就是从+1 的位置开始
                    partitionOffsetMap.put(topicPartition, records.get(records.size() - 1).offset() + 1);
                }
            }
        }
    }
}

```

```

        }
        //如果 list 中存在有数据,则进行入库操作
        if (list.size() > 0) {
            boolean isSuccess = insertIntoDB(list, partitionOffsetMap);
            if (isSuccess) {
                //将缓存数据清空,并将 offset 信息清空
                list.clear();
                partitionOffsetMap.clear();
            }
        }
    }
} catch (Exception e) {
    //处理异常
} finally {
    //offset 信息由我们自己保存, 提交 offset 其实没有什么必要
    //consumer.commitSync();
    close();
}
}

private boolean insertIntoDB(List<Message> list, Map<TopicPartition, Long> partitionOffsetMap) {
    Connection connection = getConnection(); //获取数据库连接 自行实现
    boolean flag = false;
    try {
        //设置手动提交, 让插入数据和更新 offset 信息在一个事务中完成
        connection.setAutoCommit(false);
        insertMessage(list); //将数据进行入库 自行实现
        updateOffset(partitionOffsetMap); //更新 offset 信息 自行实现
        connection.commit();
        flag = true;
    } catch (SQLException e) {
        try {
            //出现异常则回滚事务
            connection.rollback();
        } catch (SQLException e1) {
            //处理异常
        }
    }
    return flag;
}

//获取数据库连接 自行实现
private Connection getConnection() {
    return null;
}

public void close() {
    isRunning = false;
    if (consumer != null) {
        consumer.close();
    }
}

private class HandleRebalance implements ConsumerRebalanceListener {
    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        //发生 Rebalance 时, 需要将 list 中数据和记录 offset 信息清空
    }
}

```

```

//这里为什么要清除数据？因为在 Rebalance 的时候有可能还有一批缓存数据在内存中没有进行入库
//并且 offset 信息也没有更新，如果不清除，那么下一次还会重新 poll 一次这些数据，导致数据重复
list.clear();
partitionOffsetMap.clear();
}

@Override
public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
    //获取对应 Topic 的分区数
    List<PartitionInfo> partitionInfos = consumer.partitionsFor(topicName);
    Map<TopicPartition, Long> partitionOffsetMapFromDB =
        getPartitionOffsetMapFromDB(partitionInfos.size());
    //在分配分区时指定消费位置
    for (TopicPartition partition : partitions) {
        //如果在数据库中有对应 partition 的信息则使用，否则将默认从 offset=0 开始消费
        if (partitionOffsetMapFromDB.get(partition) != null) {
            consumer.seek(partition, partitionOffsetMapFromDB.get(partition));
        } else {
            consumer.seek(partition, 0L);
        }
    }
}
/**
 * 从数据库中查询分区和 offset 信息
 * @param size 分区数量
 * @return 分区号和 offset 信息
 */
private Map<TopicPartition, Long> getPartitionOffsetMapFromDB(int size) {
    Map<TopicPartition, Long> partitionOffsetMapFromDB = new HashMap<>();
    //从数据库中查询出对应信息
    Connection connection = getConnection(); //获取数据库连接 自行实现
    PreparedStatement preparedStatement = null;
    ResultSet resultSet = null;
    String querySql = "SELECT partition_num,offsets from kafka_info WHERE topic_group = ?";
    try {
        preparedStatement = connection.prepareStatement(querySql);
        preparedStatement.setString(1, topicNameAndGroupId);
        resultSet = preparedStatement.executeQuery();
        while (resultSet.next()) {
            partitionOffsetMapFromDB.put(new TopicPartition(topicName, resultSet.getInt(1)),
                resultSet.getLong(2));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    //判断数据库是否存在所有的分区的信息,如果没有,则需要进行初始化
    if (partitionOffsetMapFromDB.size() < size) {
        connection.setAutoCommit(false);
        StringBuilder sqlBuilder = new StringBuilder();
        //partition 分区号是从 0 开始,如果有 10 个分区,那么分区号就是 0-9
        /*这里拼接插入数据 格式 INSERT INTO kafka_info(topic_group_partition,topic_group,partition_num) VALUES
        (topicNameAndGroupId_0,topicNameAndGroupId_0),(topicNameAndGroupId_1, topicNameAndGroupId_1),...*/
        for (int i = 0; i < size; i++) {
            sqlBuilder.append("(").append
                (topicNameAndGroupId).append("_").append(i).append(",").append
                (topicNameAndGroupId).append(",").append(i).append(",");
        }
        //将最后一个逗号去掉加上分号结束
        sqlBuilder.deleteCharAt(sqlBuilder.length() - 1).append(";");
    }
}

```

```

        preparedStatement = connection.prepareStatement("INSERT INTO kafa_info" +
                "(topic_group_partition,topic_group,partition_num) VALUES " + sqlBuilder.toString());
        preparedStatement.execute();
        connection.commit();
    }
} catch (SQLException e) {
    //处理异常 回滚事务 这里应该结束程序 排查错误
    try {
        connection.rollback();
    } catch (SQLException e1) {
        //打印日志 排查错误信息
    }
} finally {
    try {
        if (resultSet != null) {
            resultSet.close();
        }
        if (preparedStatement != null) {
            preparedStatement.close();
        }
        if (connection != null) {
            connection.close();
        }
    } catch (SQLException e) {
        //处理异常 打印日志即可 关闭资源失败
    }
}
}
return partitionOffsetMapFromDB;
}
}

```

数据对象

```

package com.huawei.kafka.consumer;

/**
 * @author: xuqiangnj@163.com
 * @date: 2019/5/3 14:07
 * @description: Message 对象 这里模拟数据
 */
public class Message {

    private String id;

    private String name;

    private String desc;

    private Date time;

    //get set toString 方法省略
}

```

这种实现方式对于以下故障场景测试通过，虽然不能说所有故障场景均可以保证精确一次消费，但目前基本覆盖大部分故障场景：

- 1) 一个消费者组中的某个消费者频繁加入组或离开组
- 2) 直接 kill 消费应用程序
- 3) 故障 kafka 集群中某个节点
- 4) 故障客户端网络，使其不能连接 kafka server 端

5) 直接重启应用程序所在虚拟机

总结：这里主要使用自己管理 offset 的方式来确保数据和 offset 信息是同时变化的，通过数据库事务的特性来保证一致性和原子性

3.4 Kafka 高效读写数据

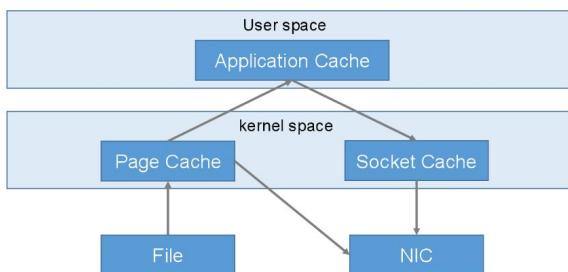
1) 顺序写磁盘

Kafka 的 producer 生产数据，要写入到 log 文件中，写的过程是一直追加到文件末端，为顺序写。官网有数据表明，同样的磁盘，顺序写能到到 600M/s，而随机写只有 100k/s。这与磁盘的机械机构有关，顺序写之所以快，是因为其省去了大量磁头寻址的时间。

2) 零拷贝技术

数据直接在内核完成输入和输出，不需要拷贝到用户空间再写出去。kafka 数据写入磁盘前，数据先写到进程的内存空间。MMap 和 sendFile 技术。

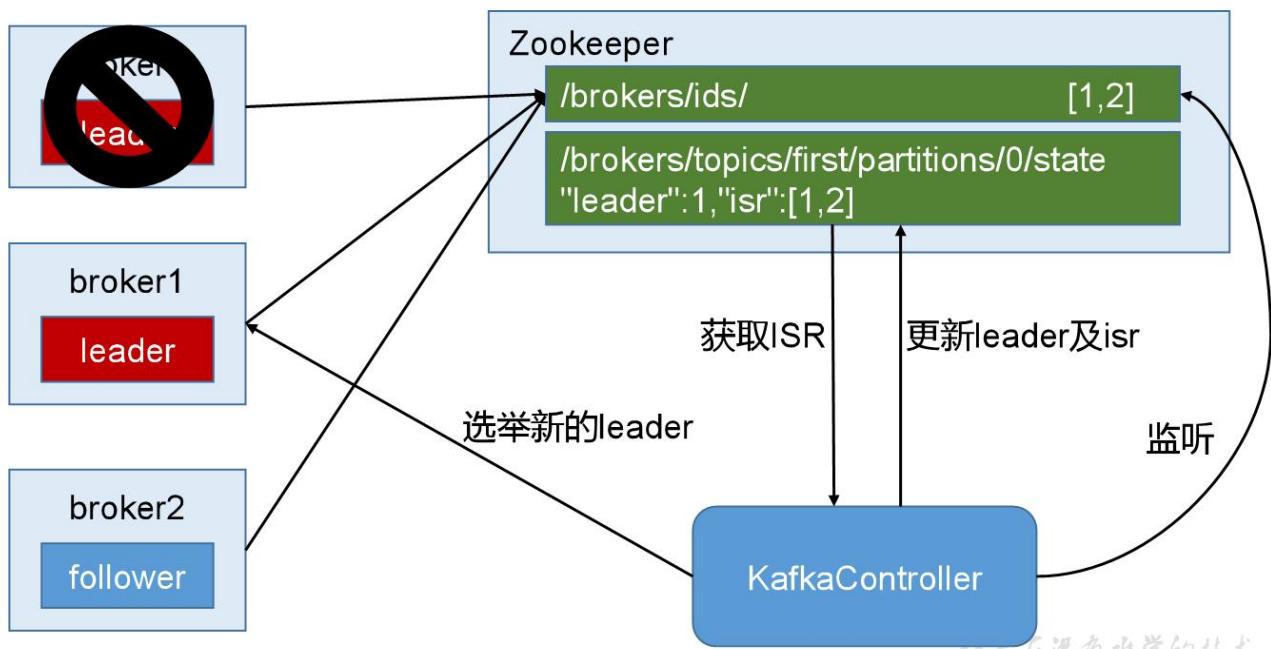
零拷贝



3.5 Zookeeper 在 Kafka 中的作用

Kafka 集群中有一个 broker 会被选举为 Controller，负责管理集群 broker 的上下线，所有 topic 的分区副本分配和 leader 选举等工作。而 Controller 的管理工作依赖于 Zookeeper。Zookeeper 负责监听 kafkaController，维护 ISR 和 leader。

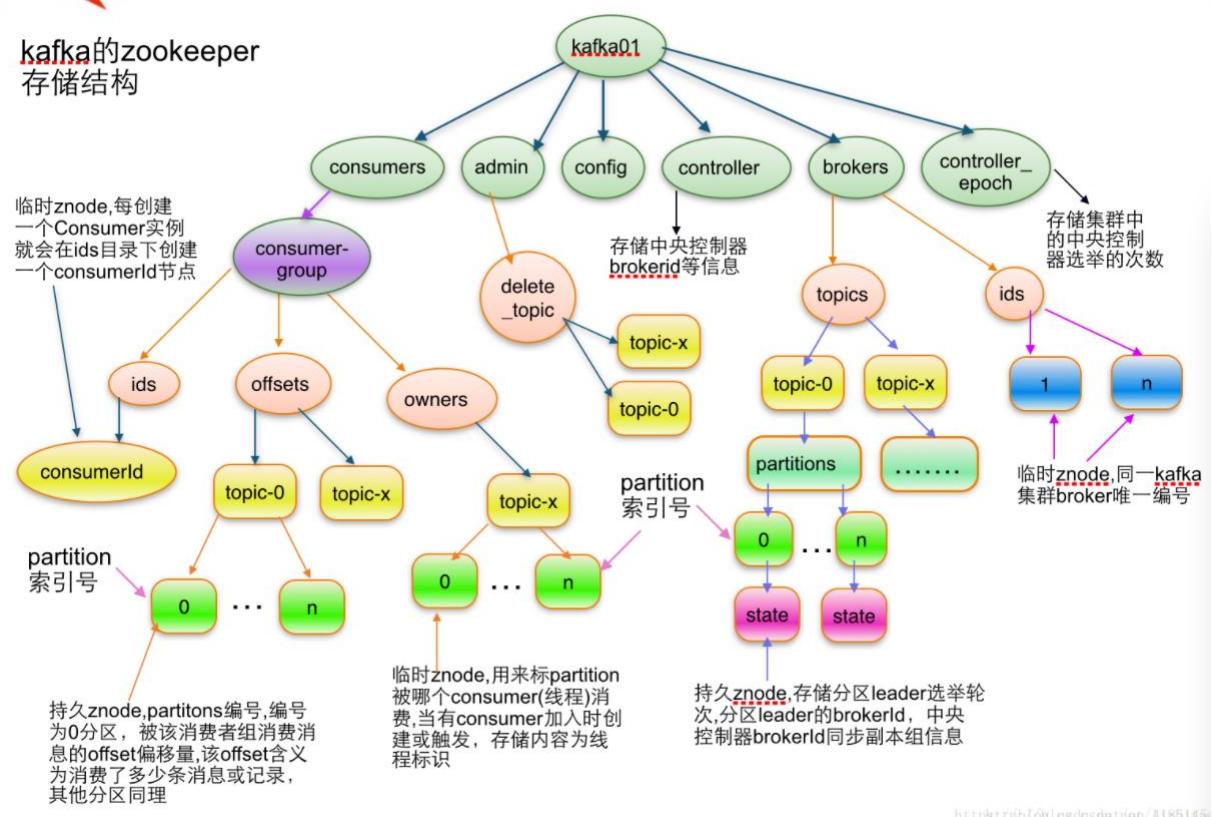
以下为 partition 的 leader 选举过程：



总结：kafka 的部分数据会存放到 zookeeper 中，比如 broker 的信息（brokerid）、broker 中的 topic 信息（leader，

ISR 集合), 0.9 版本之前还存放消费组的 offset, 0.9 版本之后存放到 kafka 的一个 topic 中。

3.6 kafka 元数据管理



参考

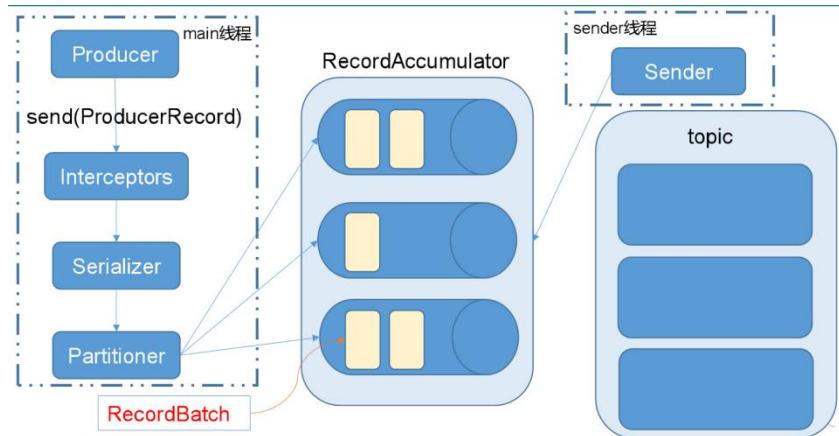
https://blog.csdn.net/qq_41851454/article/details/80239602?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommentFromMachineLearnPai2-1.control&dist_request_id=&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommentFromMachineLearnPai2-1.control

第 4 章 Kafka API

4.1 Producer API

4.1.1 消息发送流程

Kafka 的 Producer 发送消息采用的是异步发送的方式。在消息发送的过程中，涉及到了两个线程——**main 线程** 和 **Sender 线程**，以及一个线程共享变量——**RecordAccumulator**。main 线程将消息发送给 RecordAccumulator，Sender 线程不断从 RecordAccumulator 中拉取消息发送到 Kafka broker。



相关参数：

batch.size: 只有数据积累到 batch.size 之后，sender 才会发送数据。

linger.ms: 如果数据迟迟未达到 batch.size，sender 等待 linger.time 之后就会发送数据。

4.1.2 异步发送 API

1) 导入依赖

```
<dependency>
<groupId>org.apache.kafka</groupId>
<artifactId>kafka-clients</artifactId>
<version>0.11.0.0</version>
</dependency>
```

2) 编写代码

需要用到的类：

KafkaProducer: 需要创建一个生产者对象，用来发送数据

ProducerConfig: 获取所需的一系列配置参数

ProducerRecord: 每条数据都要封装成一个 ProducerRecord 对象

1.不带回调函数的 API

```
import org.apache.kafka.clients.producer.*;
import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class CustomProducer {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        Properties props = new Properties();
        props.put("bootstrap.servers", "hadoop102:9092");//kafka 集群, broker-list
        props.put("acks", "all"); // -1
        props.put("retries", 1);//重试次数
        props.put("batch.size", 16384);//批次大小
        props.put("linger.ms", 1);//等待时间
        props.put("buffer.memory", 33554432);//RecordAccumulator 缓冲区大小
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        Producer<String, String> producer = new KafkaProducer<>(props);
        for (int i = 0; i < 100; i++) {
            producer.send(new ProducerRecord<String, String>("first", Integer.toString(i), Integer.toString(i)));
        }
        producer.close();
    }
}
```

2.带回调函数的 API

回调函数会在 producer 收到 ack 时调用，为异步调用，该方法有两个参数，分别是 RecordMetadata 和 Exception，如果 Exception 为 null，说明消息发送成功，如果 Exception 不为 null，说明消息发送失败。

注意：消息发送失败会自动重试，不需要我们在回调函数中手动重试。

```
import org.apache.kafka.clients.producer.*;
import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class CustomProducer {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        Properties props = new Properties();
        props.put("bootstrap.servers", "hadoop102:9092");//kafka 集群, broker-list
        props.put("acks", "all"); // -1
        props.put("retries", 1);//重试次数
        props.put("batch.size", 16384);//批次大小
```

```

props.put("linger.ms", 1); //等待时间
props.put("buffer.memory", 33554432); //RecordAccumulator 缓冲区大小
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);
for (int i = 0; i < 100; i++) {
    producer.send(new ProducerRecord<String, String>("first", Integer.toString(i), Integer.toString(i)), new C
allback() {
    //回调函数，该方法会在 Producer 收到 ack 时调用，为异步调用
    @Override
    public void onCompletion(RecordMetadata metadata, Exception exception) {
        if (exception == null) {
            System.out.println("success->" + metadata.offset());
        } else {
            exception.printStackTrace();
        }
    }
});
}
producer.close();
}
}

```

4.1.3 同步发送 API

同步发送的意思是一条消息发送之后会阻塞当前线程，直至返回 ack。

由于 send 方法返回的是一个 Future 对象，根据 Future 对象的特点，我们也可以实现同步发送的效果，只需在调用 Future 对象的 get 方法即可。

```

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class CustomProducer {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        Properties props = new Properties();
        props.put("bootstrap.servers", "hadoop102:9092"); //kafka 集群，broker-list
        props.put("acks", "all");
        props.put("retries", 1); //重试次数
        props.put("batch.size", 16384); //批次大小
        props.put("linger.ms", 1); //等待时间
        props.put("buffer.memory", 33554432); //RecordAccumulator 缓冲区大小
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        Producer<String, String> producer = new KafkaProducer<>(props);
        for (int i = 0; i < 100; i++) {
            producer.send(new ProducerRecord<String, String>("first", Integer.toString(i), Integer.toString(i))).get();
        }
        producer.close();
    }
}

```

4.2 Consumer API

Consumer 消费数据时的可靠性是很容易保证的，因为数据在 Kafka 中是持久化的，故不用担心数据丢失问题。

由于 consumer 在消费过程中可能会出现断电宕机等故障，consumer 恢复后，需要从故障前的位置的继续消费，所以 consumer 需要实时记录自己消费到了哪个 offset，以便故障恢复后继续消费。

所以 offset 的维护是 Consumer 消费数据是必须考虑的问题。

4.2.1 手动提交 offset

1) 导入依赖

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.0</version>
</dependency>
```

2) 编写代码

需要用到的类：

KafkaConsumer: 需要创建一个消费者对象，用来消费数据
ConsumerConfig: 获取所需的一系列配置参数
ConsumerRecord: 每条数据都要封装成一个 ConsumerRecord 对象

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import java.util.Arrays;
import java.util.Properties;

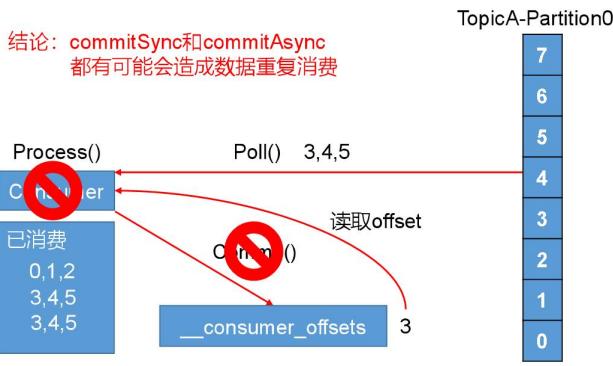
public class CustomConsumer {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "hadoop102:9092");
        props.put("group.id", "test");//消费者组，只要 group.id 相同，就属于同一个消费者组
        props.put("enable.auto.commit", "false");//禁止自动提交 offset
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList("first"));
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(100);
            for (ConsumerRecord<String, String> record : records) {
                System.out.printf("offset=%d, key=%s, value=%s%n", record.offset(), record.key(), record.value());
            }
            consumer.commitSync();
        }
    }
}
```

3) 代码分析：

手动提交 offset 的方法有两种：分别是 commitSync（同步提交）和 commitAsync（异步提交）。两者的相同点是，都会将本次 poll 的一批数据最高的偏移量提交；不同点是，commitSync 会失败重试，一直到提交成功（如果由于不可恢复原因导致，也会提交失败）；而 commitAsync 则没有失败重试机制，故有可能提交失败。

3) 数据重复消费问题



4.2.2 自动提交 offset

为了使我们能够专注于自己的业务逻辑，Kafka 提供了自动提交 offset 的功能。

自动提交 offset 的相关参数：

enable.auto.commit: 是否开启自动提交 offset 功能

auto.commit.interval.ms: 自动提交 offset 的时间间隔

以下为自动提交 offset 的代码：

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.util.Arrays;
import java.util.Properties;

public class CustomConsumer {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "hadoop102:9092");
        props.put("group.id", "test");
        props.put("enable.auto.commit", "true"); // 自动提交
        props.put("auto.commit.interval.ms", "1000");
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList("first"));
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(100);
            for (ConsumerRecord<String, String> record : records)
                System.out.printf("offset=%d, key=%s, value=%s%n", record.offset(), record.key(), record.value());
        }
    }
}
```

4.3 拦截器 Interceptor

4.3.1 拦截器原理

Producer 拦截器(interceptor)是在 Kafka 0.10 版本被引入的，主要用于实现 clients 端的定制化控制逻辑。

对于 producer 而言，interceptor 使得用户在消息发送前以及 producer 回调逻辑前有机会对消息做一些定制化需求，比如修改消息等。同时，producer 允许用户指定多个 interceptor 按序作用于同一条消息从而形成一个拦截链(interceptor chain)。Interceptpor 的实现接口是 org.apache.kafka.clients.producer.ProducerInterceptor，其定义的方法包括：

(1) configure(configs)

获取配置信息和初始化数据时调用。

(2) onSend(ProducerRecord):

该方法封装进 KafkaProducer.send 方法中，即它运行在用户主线程中。Producer 确保在消息被序列化以及计算分区前调用该方法。用户可以在该方法中对消息做任何操作，但最好保证不要修改消息所属的 topic 和分区，否则会影响目标分区的计算。

(3) onAcknowledgement(RecordMetadata, Exception):

该方法会在消息从 RecordAccumulator 成功发送到 Kafka Broker 之后，或者在发送过程中失败时调用。并且通常都是在 producer 回调逻辑触发之前。onAcknowledgement 运行在 producer 的 IO 线程中，因此不要在该方法中放入很重的逻辑，否则会拖慢 producer 的消息发送效率。

(4) close:

关闭 interceptor，主要用于执行一些资源清理工作

如前所述，interceptor 可能被运行在多个线程中，因此在具体实现时用户需要自行确保线程安全。另外倘若指定了多个 interceptor，则 producer 将按照指定顺序调用它们，并仅仅是捕获每个 interceptor 可能抛出的异常记录到错误日志中而非在向上传递。这在使用过程中要特别留意。

4.3.2 拦截器案例

1) 需求：

实现一个简单的双 interceptor 组成的拦截链。第一个 interceptor 会在消息发送前将时间戳信息加到消息 value 的最前部；第二个 interceptor 会在消息发送后更新成功发送消息数或失败发送消息数。

Kafka 拦截器：

发送的数据	TimeInterceptor	CounterInterceptor	InterceptorProducer
	1) 实现ProducerInterceptor 2) 获取record数据，并在 value前增加时间戳	1) 返回record 2) 统计发送成功是失败次数	1) 构建拦截器链 2) 发送数据 3) 关闭producer时，打印统计次数 <code>success:10 error:0</code>
)			
message0	1502102979120,message0	1502102979120,message0	
message1	1502102979242,message1	1502102979242,message1	
...	
message9	1502102979242,message9	1502102979242,message9	
message10	1502102979242,message10	1502102979242,message10	

2) 案例实操

(1) 增加时间戳拦截器

```
package com.atguigu.kafka.interceptor;
import java.util.Map;
import org.apache.kafka.clients.producer.ProducerInterceptor;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

public class TimeInterceptor implements ProducerInterceptor<String, String> {

    @Override
    public void configure(Map<String, ?> configs) {
    }

    //producer 发送给 kafka 之前的数据拦截逻辑
    @Override
    public ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {
        // 创建一个新的 record，把时间戳写入消息体的最前部
    }
}
```

```

        return new ProducerRecord(record.topic(), record.partition(), record.timestamp(), record.key(), System.currentTimeMillis() + "," + record.value().toString());
    }

    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {

    }

    @Override
    public void close() {

    }
}

```

(2) 统计发送消息成功和发送失败消息数，并在 producer 关闭时打印这两个计数器

```

import java.util.Map;
import org.apache.kafka.clients.producer.ProducerInterceptor;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

public class CounterInterceptor implements ProducerInterceptor<String, String>{
    private int errorCounter = 0;
    private int successCounter = 0;

    @Override
    public void configure(Map<String, ?> configs) {

    }

    @Override
    public ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {
        return record;
    }

    // producer 发送数据后，根据 exception 是否为空对数据进行处理
    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {
        // 统计成功和失败的次数
        if (exception == null) {
            successCounter++;
        } else {
            errorCounter++;
        }
    }

    // 关闭 producer 之前最后做的事
    @Override
    public void close() {
        // 保存结果
        System.out.println("Successful sent: " + successCounter);
        System.out.println("Failed sent: " + errorCounter);
    }
}

```

(3) producer 主程序

```

package com.atguigu.kafka.interceptor;
import java.util.ArrayList;
import java.util.List;

```

```

import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;

public class InterceptorProducer {
    public static void main(String[] args) throws Exception {
        // 1 设置配置信息
        Properties props = new Properties();
        props.put("bootstrap.servers", "hadoop102:9092");
        props.put("acks", "all");
        props.put("retries", 0);
        props.put("batch.size", 16384);
        props.put("linger.ms", 1);
        props.put("buffer.memory", 33554432);
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        // 2 构建拦截链
        List<String> interceptors = new ArrayList<>();
        interceptors.add("com.atguigu.kafka.interceptor.TimeInterceptor");
        interceptors.add("com.atguigu.kafka.interceptor.CounterInterceptor");
        props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, interceptors);

        String topic = "first";
        Producer<String, String> producer = new KafkaProducer<>(props);

        // 3 发送消息
        for (int i = 0; i < 10; i++) {
            ProducerRecord<String, String> record = new ProducerRecord<>(topic, "message" + i);
            producer.send(record);
        }

        // 4 一定要关闭 producer，这样才会调用 interceptor 的 close 方法
        producer.close();
    }
}

```

3) 测试

(1) 在 kafka 上启动消费者，然后运行客户端 java 程序。

```
[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh \
--bootstrap-server hadoop102:9092 --from-beginning --topic first
```

```

1501904047034,message0
1501904047225,message1
1501904047230,message2
1501904047234,message3
1501904047236,message4
1501904047240,message5
1501904047243,message6
1501904047246,message7
1501904047249,message8
1501904047252,message9

```

第 5 章 Flume 对接 Kafka

1) 配置 flume(flume-kafka.conf)

```
# define
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# source
a1.sources.r1.type = exec
a1.sources.r1.command = tail -F -c +0 /opt/module/datas/flume.log
a1.sources.r1.shell = /bin/bash -c

# sink
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.bootstrap.servers = hadoop102:9092,hadoop103:9092,hadoop104:9092
a1.sinks.k1.kafka.topic = first
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.linger.ms = 1 # linger 继续留存

# channel
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000 # capacity 容积, 容量
a1.channels.c1.transactionCapacity = 100 # capacity 交易

# bind
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

2) 启动 kafkaIDEA 消费者

3) 进入 flume 根目录下, 启动 flume

```
$ bin/flume-ng agent -c conf/ -n a1 -f jobs/flume-kafka.conf
```

4) 向 /opt/module/datas/flume.log 里追加数据, 查看 kafka 消费者消费情况

```
$ echo hello >> /opt/module/datas/flume.log
```

第 6 章 Kafka 监控

6.1 Kafka Monitor

Kafka Monitor 是 kafka 的监控工具, 安装和页面都简单, 主要是用来监控消费者和 offset, 对 kafka 集群的支持较少。

1.上传 jar 包 KafkaOffsetMonitor-assembly-0.4.6.jar 到集群

2.在/opt/module/下创建 kafka-offset-console 文件夹

3.将上传的 jar 包放入刚创建的目录下

4.在/opt/module/kafka-offset-console 目录下创建启动脚本 start.sh, 内容如下:

```
#!/bin/bash
java -cp KafkaOffsetMonitor-assembly-0.4.6-SNAPSHOT.jar \
com.quantifind.kafka.offsetapp.OffsetGetterWeb \
--offsetStorage kafka \
--kafkaBrokers hadoop102:9092,hadoop103:9092,hadoop104:9092 \
--kafkaSecurityProtocol PLAINTEXT \
--zk hadoop102:2181,hadoop103:2181,hadoop104:2181 \
--port 8086 \
--refresh 10.seconds \
--retain 2.days \
--dbName offsetapp_kafka &
```

5. 在 /opt/module/kafka-offset-console 目录下创建 mobile-logs 文件夹

```
mkdir /opt/module/kafka-offset-console/mobile-logs
```

6. 启动 KafkaMonitor

```
./start.sh
```

7. 登录页面 hadoop102:8086 端口查看详情

6.2 Kafka Manager

1. 上传压缩包 kafka-manager-1.3.3.15.zip 到集群

2. 解压到 /opt/module

3. 修改配置文件 conf/application.conf

```
kafka-manager.zkhosts="kafka-manager-zookeeper:2181"
```

修改为：

```
kafka-manager.zkhosts="hadoop102:2181,hadoop103:2181,hadoop104:2181"
```

4. 启动 kafka-manager

```
bin/kafka-manager
```

5. 登录 hadoop102:9000 页面查看详细信息

6.3 Kafka Eagle

第 7 章 Kafka 面试题

7.1 面试问题

1. Kafka 中的 ISR、AR 又代表什么？

ISR 是一个动态维护的与 leader 保持联系的 follower 集合

2. Kafka 中的 HW、LEO 等分别代表什么？

HW 是该分区所有副本最小的 LEO

LEO 是副本最后一条消息的 offset

3. Kafka 中是怎么体现消息顺序性的？

每个分区内，每条消息都有一个 offset，故只能保证分区内有序。

4. Kafka 中的分区器、序列化器、拦截器是否了解？它们之间的处理顺序是什么？

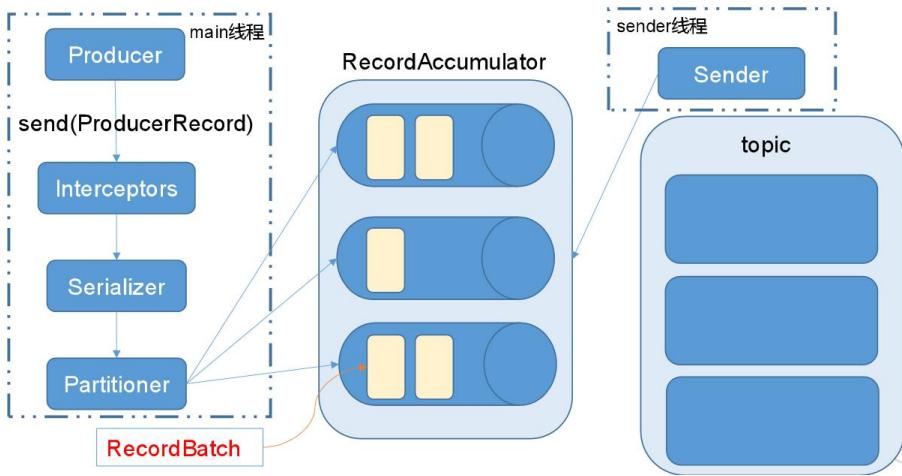
分区器 partitioner 用来定义分区方式

序列化器用来序列化数据，方便网络传输

拦截器是 producer 对发送到 kafka 的数据进行加工，可以形成拦截链。

拦截器 -> 序列化器 -> 分区器

5. Kafka 生产者客户端的整体结构是什么样子的？使用了几个线程来处理？分别是什么？



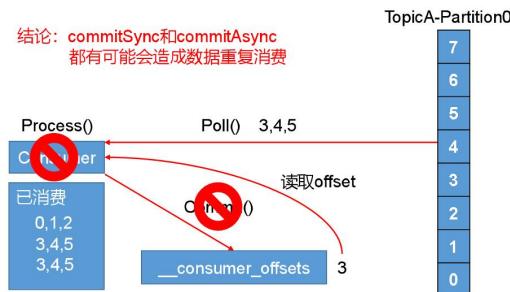
6. “消费组中的消费者个数如果超过 topic 的分区，那么就会有消费者消费不到数据”这句话是否正确？

正确

7. 消费者提交消费位移时提交的是当前消费到的最新消息的 offset 还是 offset+1？

提交的是 offset+1

8. 有哪些情形会造成重复消费？



消费成功但提交 offset 失败，无论同步还是异步提交

9. 那些情景会造成消息漏消费？

先提交 offset 后消费，有可能造成数据漏消费

Producer 发送到 kafka 时候，ack 等于 0 或者 1 时，leader 挂了就会导致漏消费

Consumer 在消费时候，消费数据能力低于拉取的数据量，此时没来的及消费的数据就会丢失

10. 当你使用 kafka-topics.sh 创建（删除）了一个 topic 之后，Kafka 背后会执行什么逻辑？

- 1) 会在 zookeeper 中的/brokers/topics 节点下创建一个新的 topic 节点，如：/brokers/topics/first
- 2) 触发 Controller 的监听程序
- 3) kafka Controller 负责 topic 的创建工作，并更新 metadata cache

11. topic 的分区数可不可以增加？如果可以怎么增加？如果不可以，那又是为什么？

可以增加

12. topic 的分区数可不可以减少？如果可以怎么减少？如果不可以，那又是为什么？

不能减少，减少分区会导致数据丢失，同时加剧其他分区的消费压力

13. Kafka 有内部的 topic 吗？如果有是什么？有什么用？

有一个存放 offset 的 topic, __consumer_offsets, 维护 offset

14. Kafka 分区分配的概念？

一个 topic 多个分区，一个消费者组多个消费者，故需要将分区分配给消费者(roundrobin、range)

15. 简述 Kafka 的日志目录结构？

partition--segment--log 和.index

16. 如果我指定了一个 offset，Kafka Controller 怎么查找到对应的消息？

17. 聊一聊 Kafka Controller 的作用？

负责管理集群 broker 的上下线，所有 topic 的分区副本分配和 leader 选举等工作。

18.Kafka 中有那些地方需要选举？这些地方的选举策略又有哪些？

partition leader(ISR)和 kafka controller(先到先得)

19.失效副本是指什么？有那些应对措施？

不能及时与 leader 同步的 follower，暂时踢出 ISR，等其追上 leader 之后再重新加入

20.Kafka 的那些设计让它有如此高的性能？

分区，顺序读取，零复制

7.2 参考答案



Kafka相关面试题
及答案.docx

SparkCore

第 1 章 RDD 概述

1.1 什么是 RDD

RDD (Resilient Distributed Dataset) 叫做弹性分布式数据集，是 Spark 中最基本的数据抽象。代码中是一个抽象类，它代表一个不可变、可分区、里面的元素可并行计算的集合。弹性指的是 RDD 分区可变多变少，分布式是 RDD 的数据可以物理分布在不同的地方。

1.2 RDD 的属性

* Internally, each RDD is characterized by five main properties:

- * - A list of partitions
- * - A function for computing each split
- * - A list of dependencies on other RDDs
- * - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
- * - Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)

- 1) 一组分区 (Partition)，即数据集的基本组成单位；
- 2) 一个计算每个分区的函数；
- 3) RDD 之间的依赖关系；
- 4) 一个 Partitioner，即 RDD 的分片函数；
- 5) 一个列表，存储取每个 Partition 的优先位置 (preferred location)。

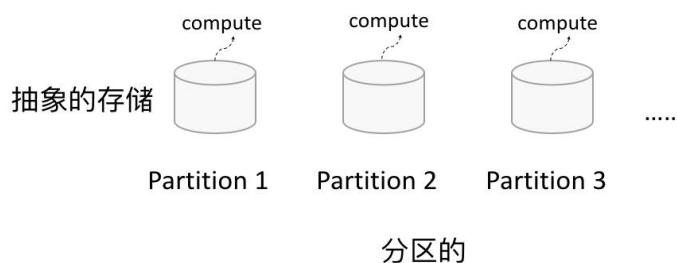
1.3 RDD 特点

RDD 表示只读的分区的数据集，对 RDD 进行改动，只能通过 RDD 的转换操作，由一个 RDD 得到一个新的 RDD，新的 RDD 包含了从其他 RDD 衍生所必需的信息。RDDs 之间存在依赖，RDD 的执行是按照血缘关系延时计算的。如果血缘关系较长，可以通过持久化 RDD 来切断血缘关系。

1.3.1 分区

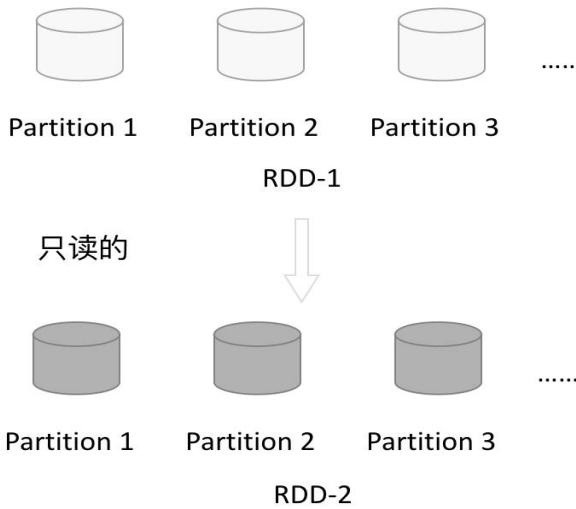
分区是 RDD 内部并行计算的一个计算单元，RDD 的数据集在逻辑上被划分为多个分片，每一个分片称为分区，分区的格式决定了并行计算的粒度，而每个分区的数值计算都是在一个任务中进行的，因此任务的个数也是由 RDD (准确来说是作业最后一个 RDD) 的分区数决定。

RDD 逻辑上是分区的，每个分区的数据是抽象存在的，计算的时候会通过一个 compute 函数得到每个分区的数据。如果 RDD 是通过已有的文件系统构建，则 compute 函数是读取指定文件系统中的数据，如果 RDD 是通过其他 RDD 转换而来，则 compute 函数是执行转换逻辑将其他 RDD 的数据进行转换。

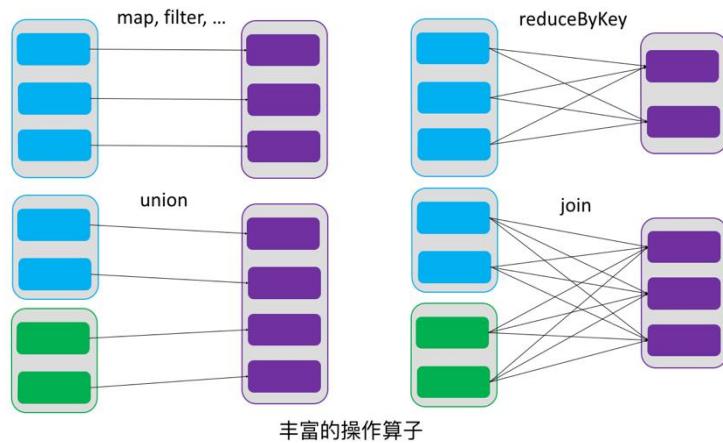


1.3.2 只读

如下图所示，RDD 是只读的，要想改变 RDD 中的数据，只能在现有的 RDD 基础上创建新的 RDD。



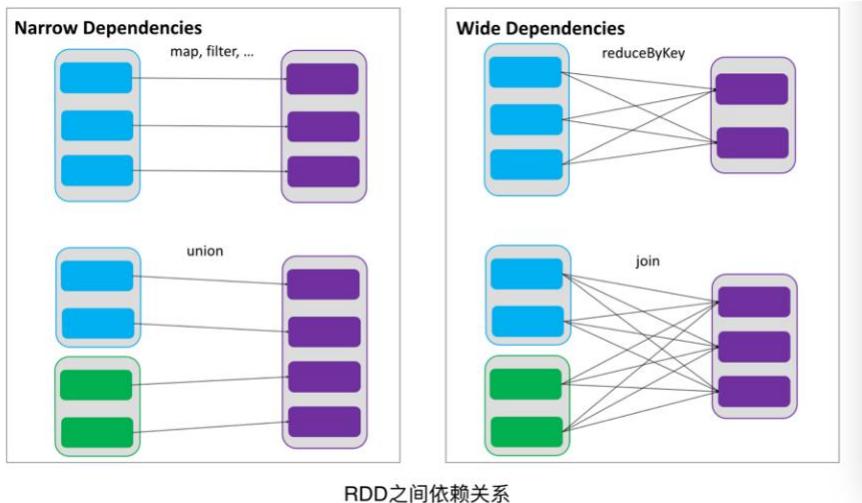
由一个 RDD 转换到另一个 RDD，可以通过丰富的操作算子实现，不再像 MapReduce 那样只能写 map 和 reduce 了，如下图所示。



RDD 的操作算子包括两类，一类叫做 **transformations**，它是用来将 RDD 进行转化，构建 RDD 的血缘关系；另一类叫做 **actions**，它是用来触发 RDD 的计算，得到 RDD 的相关计算结果或者将 RDD 保存的文件系统中。下图是 RDD 所支持的操作算子列表。

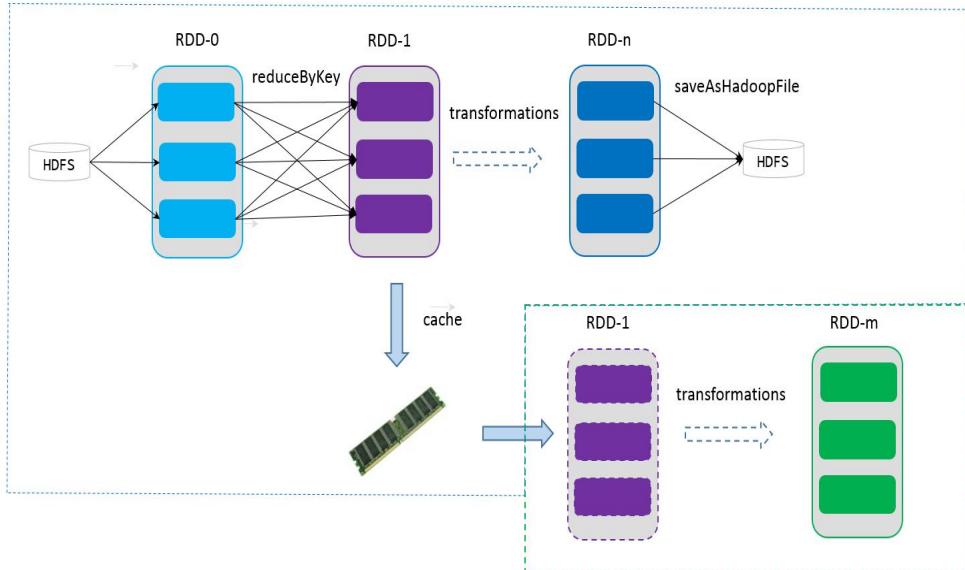
1.3.3 依赖

RDDs 通过操作算子进行转换，转换得到的新 RDD 包含了从其他 RDDs 衍生所必需的信息，RDDs 之间维护着这种血缘关系，也称之为依赖。如下图所示，依赖包括两种，一种是窄依赖，RDDs 之间分区是一一对应的，另一种是宽依赖，下游 RDD 的每个分区与上游 RDD(也称之为父 RDD)的每个分区都有关，是多对多的关系。



1.3.4 缓存

如果在应用程序中多次使用同一个 RDD，可以将该 RDD 缓存起来，该 RDD 只有在第一次计算的时候会根据血缘关系得到分区的数据，在后续其他地方用到该 RDD 的时候，会直接从缓存处取而不用再根据血缘关系计算，这样就加速后期的重用。如下图所示，RDD-1 经过一系列的转换后得到 RDD-n 并保存到 hdfs，RDD-1 在这一过程中会有个中间结果，如果将其缓存到内存，那么在随后的 RDD-1 转换到 RDD-m 这一过程中，就不会计算其之前的 RDD-0 了。



1.3.5 CheckPoint

RDD 的血缘关系可以实现容错，即当 RDD 的某个分区数据失败或丢失可以通过血缘关系重建。但是对于长时间迭代的应用来说，随着迭代的进行，RDDs 之间的血缘关系会越来越长，一旦在后续迭代过程中出错，则需要通过非常长的血缘关系去重建，势必影响性能。为此，RDD 支持 checkpoint 将数据持久化到存储中，这样就可以切断之前的血缘关系，因为 checkpoint 后的 RDD 不需要知道它的父 RDDs 了，它可以从 checkpoint 处拿到数据。

1.3.6 缓存和 checkpoint 的区别

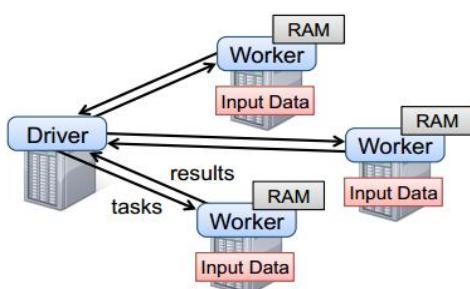
缓存是为了 RDD 复用，checkpoint 是防止数据丢失导致重新计算影响性能。

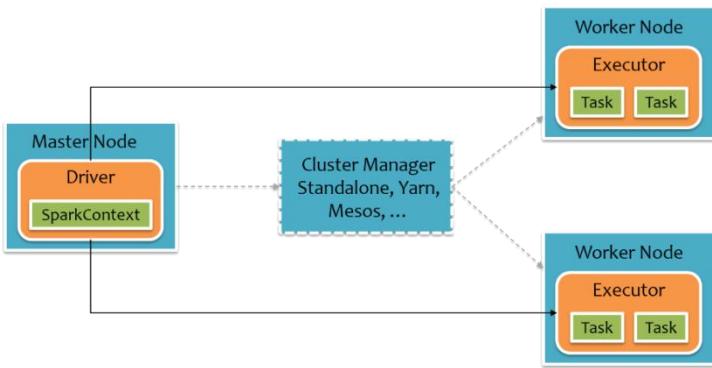
第 2 章 RDD 编程

2.1 编程模型

在 Spark 中，RDD 被表示为对象，通过对对象上的方法调用来对 RDD 进行转换。经过一系列的 transformations 定义 RDD 之后，就可以调用 actions 触发 RDD 的计算，action 可以是向应用程序返回结果(count, collect 等)，或者是向存储系统保存数据(saveAsTextFile 等)。在 Spark 中，只有遇到 action 才会执行 RDD 的计算(即延迟计算)，这样在运行时可以通过管道的方式传输多个转换。

要使用 Spark，开发者需要编写一个 Driver 程序，它被提交到集群以调度运行 Worker，如下图所示。Driver 中定义了一个或多个 RDD，并调用 RDD 上的 action，Worker 则执行 RDD 分区计算任务。





2.2 RDD 的创建

在 Spark 中创建 RDD 的创建方式可以分为三种：从集合中创建 RDD；从外部存储创建 RDD；从其他 RDD 创建。

2.2.1 从集合中创建

从集合中创建 RDD，Spark 主要提供了两种函数：parallelize 和 makeRDD

```
//使用 parallelize()从集合创建
//def parallelize[T](seq: Seq[T], numSlices: Int = {})
val rdd = spark.sparkContext.parallelize(Array(1,2,3,4,5,6,7,8), 5)
```

//使用 makeRDD()从集合创建

```
//def makeRDD[T](seq: Seq[T], numSlices: Int = {})
val rdd = spark.sparkContext.makeRDD(Array(1,2,3,4,5,6,7,8))
```

2.2.2 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。

```
//def textFile(path: String, minPartitions: Int = {})
val rdd2 = spark.sparkContext.textFile("hdfs://hadoop102:9000/RELEASE", 10)
```

2.2.3 从其他 RDD 创建

详见 2.3 节

2.3 Transformation（面试开发重点）

RDD 整体上分为 Value 类型和 Key-Value 类型

2.3.1 Value 类型

2.3.1.1 map(func) 案例

1. 作用：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
2. 需求：创建一个 1-10 数组的 RDD，将所有元素*2 形成新的 RDD

(1) 创建

```
var source = spark.sparkContext.parallelize(1 to 10)
```

(2) 打印

```
source.collect()
```

(3) 将所有元素*2

```
scala> val mapadd = source.map(_ * 2)
mapadd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[9] at map at <console>:26
```

(4) 打印最终结果

```
scala> mapadd.collect()
res8: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

2.3.1.2 mapPartitions(func) 案例

1. 作用：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]`。假设有 N 个元素，有 M 个分区，那么 map 的函数将被调用 N 次，而 mapPartitions 被调用 M 次，一个函数一次处理一个分区。
2. 需求：创建一个 RDD，使每个元素*2 组成新的 RDD

```
//创建一个 RDD
```

```
val rdd = spark.sparkContext.parallelize(Array(1,2,3,4))
//使每个元素*2 组成新的 RDD
val res3 = rdd.mapPartitions(x=>x.map(_*2))
//打印新的 RDD
res3.collect
```

2.3.1.3 mapPartitionsWithIndex(func) 案例

- 作用：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]；

这个不仅能拿到分区的迭代器，还能拿到分区编号。

- 需求：创建一个 RDD，使每个元素跟所在分区形成一个元组组成一个新的 RDD

```
//创建一个 RDD
val rdd = spark.sparkContext.parallelize(Array(1,2,3,4))
//使每个元素跟所在分区形成一个元组组成一个新的 RDD
val indexRdd = rdd.mapPartitionsWithIndex((index,items)=>(items.map((index,_))))
```

//打印新的 RDD

2.3.1.4 flatMap(func) 案例

- 作用：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）

- 需求：创建一个元素为 1-5 的 RDD，运用 flatMap 创建一个新的 RDD，新的 RDD 为原 RDD 的每个元素的 2 倍 (2, 4, 6, 8, 10)

(1) 创建

```
val sourceFlat = spark.sparkContext.parallelize(1 to 5)
```

(2) 打印

```
scala> sourceFlat.collect()
res11: Array[Int] = Array(1, 2, 3, 4, 5)
```

(3) 根据原 RDD 创建新 RDD (1->1,2->1,2.....5->1,2,3,4,5)

```
scala> val flatMap = sourceFlat.flatMap(1 to _)
flatMap: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[13] at flatMap at <console>:26
```

(4) 打印新 RDD

```
scala> flatMap.collect()
res12: Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5)
```

2.3.1.5 map()和 mapPartition()的区别

- map()：每次处理一条数据。
- mapPartition()：每次处理一个分区的数据，这个分区的数据处理完后，原 RDD 中分区的数据才能释放，可能导致 OOM。（分区数据加载到内存里，全部处理才会释放掉）
- 开发指导：当内存空间较大的时候建议使用 mapPartition()，以提高处理效率。

2.3.1.6 glom 案例

- 作用：将每一个分区形成一个数组，形成新的 RDD 类型时 RDD[Array[T]]

- 需求：创建一个 4 个分区的 RDD，并将每个分区的数据放到一个数组

(1) 创建

```
scala> val rdd = sc.parallelize(1 to 16, 4)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[65] at parallelize at <console>:24
```

(2) 将每个分区的数据放到一个数组并收集到 Driver 端打印

```
scala> rdd.glom().collect()
res25: Array[Array[Int]] = Array(Array(1, 2, 3, 4), Array(5, 6, 7, 8), Array(9, 10, 11, 12), Array(13, 14, 15, 16))
```

2.3.1.7 groupBy(func)案例

- 作用：分组，按照传入函数的返回值进行分组。将相同的 key 对应的值放入一个迭代器。

- 需求：创建一个 RDD，按照元素模以 2 的值进行分组。

(1) 创建

```
scala> val rdd = sc.parallelize(1 to 4)
```

```
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[65] at parallelize at <console>:24
```

(2) 按照元素模以 2 的值进行分组

```
scala> val group = rdd.groupBy(_%2)
```

```
group: org.apache.spark.rdd.RDD[(Int, Iterable[Int])] = ShuffledRDD[2] at groupBy at <console>:26
```

(3) 打印结果

```
scala> group.collect
```

```
res0: Array[(Int, Iterable[Int])] = Array((0,CompactBuffer(2, 4)), (1,CompactBuffer(1, 3)))
```

2.3.1.8 filter(func) 案例

1. 作用：过滤。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。

2. 需求：创建一个 RDD（由字符串组成），过滤出一个新 RDD（包含"xiao"子串）

(1) 创建

```
scala> var sourceFilter = sc.parallelize(Array("xiaoming","xiaojiang","xiaohe","dazhi"))
```

```
sourceFilter: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[10] at parallelize at <console>:24
```

(2) 打印

```
scala> sourceFilter.collect()
```

```
res9: Array[String] = Array(xiaoming, xiaojiang, xiaohe, dazhi)
```

(3) 过滤出含"xiao"子串的形成一个新的 RDD

```
scala> val filter = sourceFilter.filter(_.contains("xiao"))
```

```
filter: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[11] at filter at <console>:26
```

(4) 打印新 RDD

```
scala> filter.collect()
```

```
res10: Array[String] = Array(xiaoming, xiaojiang, xiaohe)
```

2.3.1.9 sample(withReplacement, fraction, seed) 案例

1. 作用：以指定的随机种子随机抽样出占比为 fraction 的数据，fraction 为 0.4 指抽出 40%的数据，withReplacement 表示是抽出的数据是否放回，true 为有放回的抽样，false 为无放回的抽样，seed 用于指定随机数生成器种子。

2. 需求：创建一个 RDD（1-10），从中选择放回和不放回抽样

(1) 创建 RDD

```
scala> val rdd = sc.parallelize(1 to 10)
```

```
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[20] at parallelize at <console>:24
```

(2) 打印

```
scala> rdd.collect()
```

```
res15: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

(3) 放回抽样

```
scala> var sample1 = rdd.sample(true, 0.4, 2)
```

```
sample1: org.apache.spark.rdd.RDD[Int] = PartitionwiseSampledRDD[21] at sample at <console>:26
```

(4) 打印放回抽样结果

```
scala> sample1.collect()
```

```
res16: Array[Int] = Array(1, 2, 2, 7, 7, 8, 9)
```

(5) 不放回抽样

```
scala> var sample2 = rdd.sample(false, 0.2, 3)
```

```
sample2: org.apache.spark.rdd.RDD[Int] = PartitionwiseSampledRDD[22] at sample at <console>:26
```

(6) 打印不放回抽样结果

```
scala> sample2.collect()
```

```
res17: Array[Int] = Array(1, 9)
```

2.3.1.10 distinct([numTasks]) 案例

1. 作用：对源 RDD 进行去重后返回一个新的 RDD。默认情况下，只有 8 个并行任务来操作，但是可以传入一个可选的 numTasks 参数改变它。

2. 需求：创建一个 RDD，使用 distinct() 对其去重。

(1) 创建一个 RDD

```
scala> val distinctRdd = sc.parallelize(List(1,2,1,5,2,9,6,1))
```

```
distinctRdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[34] at parallelize at <console>:24
```

(2) 对 RDD 进行去重（不指定并行度）

```
scala> val unionRDD = distinctRDD.distinct()
unionRDD: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[37] at distinct at <console>:26
(3) 打印去重后生成的新 RDD
scala> unionRDD.collect()
res20: Array[Int] = Array(1, 9, 5, 6, 2)
(4) 对 RDD (指定并行度为 2)
scala> val unionRDD = distinctRDD.distinct(2)
unionRDD: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[40] at distinct at <console>:26
(5) 打印去重后生成的新 RDD
scala> unionRDD.collect()
res21: Array[Int] = Array(6, 2, 1, 9, 5)
```

2.3.1.11 coalesce(numPartitions) 案例

1. 作用：缩减分区数，用于大数据集过滤后，提高小数据集的执行效率。

2. 需求：创建一个 4 个分区的 RDD，对其缩减分区

(1) 创建一个 RDD

```
scala> val rdd = sc.parallelize(1 to 16,4)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[54] at parallelize at <console>:24
(2) 查看 RDD 的分区数
```

```
scala> rdd.partitions.size
res20: Int = 4
```

(3) 对 RDD 重新分区

```
scala> val coalesceRDD = rdd.coalesce(3)
coalesceRDD: org.apache.spark.rdd.RDD[Int] = CoalescedRDD[55] at coalesce at <console>:26
(4) 查看新 RDD 的分区数
```

```
scala> coalesceRDD.partitions.size
res21: Int = 3
```

2.3.1.12 repartition(numPartitions) 案例

1. 作用：根据分区数，重新通过网络随机洗牌所有数据。

2. 需求：创建一个 4 个分区的 RDD，对其重新分区

(1) 创建一个 RDD

```
scala> val rdd = sc.parallelize(1 to 16,4)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[56] at parallelize at <console>:24
(2) 查看 RDD 的分区数
```

```
scala> rdd.partitions.size
res22: Int = 4
```

(3) 对 RDD 重新分区

```
scala> val rerdd = rdd.repartition(2)
rerdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[60] at repartition at <console>:26
(4) 查看新 RDD 的分区数
```

```
scala> rerdd.partitions.size
res23: Int = 2
```

2.3.1.13 coalesce 和 repartition 的区别

1. coalesce 重新分区，可以选择是否进行 shuffle 过程。由参数 shuffle: Boolean = false/true 决定。

2. repartition 实际上是调用的 coalesce，默认是进行 shuffle 的。源码如下：

```
def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T] = withScope {
    coalesce(numPartitions, shuffle = true)
}
```

2.3.1.14 sortBy(func,[ascending], [numTasks]) 案例

1. 作用：使用 func 先对数据进行处理，按照处理后的数据比较结果排序，默认为正序。

2. 需求：创建一个 RDD，按照不同的规则进行排序

(1) 创建一个 RDD

```
scala> val rdd = sc.parallelize(List(2,1,3,4))
```

```
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[21] at parallelize at <console>:24
```

(2) 按照自身大小排序

```
scala> rdd.sortBy(x => x).collect()  
res11: Array[Int] = Array(1, 2, 3, 4)
```

(3) 按照与 3 余数的大小排序

```
scala> rdd.sortBy(x => x%3).collect()  
res12: Array[Int] = Array(3, 4, 1, 2)
```

2.3.1.15 pipe(command, [envVars]) 案例

1. 作用：管道，针对每个分区，都执行一个 shell 脚本，返回输出的 RDD。

注意：脚本需要放在 Worker 节点可以访问到的位置

2. 需求：编写一个脚本，使用管道将脚本作用于 RDD 上。

(1) 编写一个脚本

Shell 脚本

```
#!/bin/sh  
echo "AA"  
while read LINE; do  
    echo ">>>${LINE}  
done
```

(2) 创建一个只有一个分区的 RDD

```
scala> val rdd = sc.parallelize(List("hi","Hello","how","are","you"),1)  
rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[50] at parallelize at <console>:24
```

(3) 将脚本作用该 RDD 并打印

```
scala> rdd.pipe("/opt/module/spark/pipe.sh").collect()  
res18: Array[String] = Array(AA, >>>hi, >>>Hello, >>>how, >>>are, >>>you)
```

(4) 创建一个有两个分区的 RDD

```
scala> val rdd = sc.parallelize(List("hi","Hello","how","are","you"),2)  
rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[52] at parallelize at <console>:24
```

(5) 将脚本作用该 RDD 并打印

```
scala> rdd.pipe("/opt/module/spark/pipe.sh").collect()  
res19: Array[String] = Array(AA, >>>hi, >>>Hello, AA, >>>how, >>>are, >>>you)
```

2.3.2 双 Value 类型交互

2.3.2.1 union(otherDataset) 案例

1. 作用：对源 RDD 和参数 RDD 求并集后返回一个新的 RDD

2. 需求：创建两个 RDD，求并集

(1) 创建第一个 RDD

```
scala> val rdd1 = sc.parallelize(1 to 5)  
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[23] at parallelize at <console>:24
```

(2) 创建第二个 RDD

```
scala> val rdd2 = sc.parallelize(5 to 10)  
rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[24] at parallelize at <console>:24
```

(3) 计算两个 RDD 的并集

```
scala> val rdd3 = rdd1.union(rdd2)  
rdd3: org.apache.spark.rdd.RDD[Int] = UnionRDD[25] at union at <console>:28
```

(4) 打印并集结果

```
scala> rdd3.collect()  
res18: Array[Int] = Array(1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10)
```

2.3.2.2 subtract (otherDataset) 案例

1. 作用：计算差的一种函数，去除两个 RDD 中相同的元素，不同的 RDD 将保留下

2. 需求：创建两个 RDD，求第一个 RDD 与第二个 RDD 的差集

(1) 创建第一个 RDD

```
scala> val rdd = sc.parallelize(3 to 8)  
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[70] at parallelize at <console>:24
```

(2) 创建第二个 RDD

```
scala> val rdd1 = sc.parallelize(1 to 5)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[71] at parallelize at <console>:24
```

(3) 计算第一个 RDD 与第二个 RDD 的差集并打印

```
scala> rdd1.subtract(rdd2).collect()
```

```
res27: Array[Int] = Array(8, 6, 7)
```

2.3.2.3 intersection(otherDataset) 案例

1. 作用：对源 RDD 和参数 RDD 求交集后返回一个新的 RDD

2. 需求：创建两个 RDD，求两个 RDD 的交集

(1) 创建第一个 RDD

```
scala> val rdd1 = sc.parallelize(1 to 7)
```

```
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[26] at parallelize at <console>:24
```

(2) 创建第二个 RDD

```
scala> val rdd2 = sc.parallelize(5 to 10)
```

```
rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[27] at parallelize at <console>:24
```

(3) 计算两个 RDD 的交集

```
scala> val rdd3 = rdd1.intersection(rdd2)
```

```
rdd3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[33] at intersection at <console>:28
```

(4) 打印计算结果

```
scala> rdd3.collect()
```

```
res19: Array[Int] = Array(5, 6, 7)
```

2.3.2.4 cartesian(otherDataset) 案例

1. 作用：笛卡尔积（尽量避免使用）

2. 需求：创建两个 RDD，计算两个 RDD 的笛卡尔积

(1) 创建第一个 RDD

```
scala> val rdd1 = sc.parallelize(1 to 3)
```

```
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[47] at parallelize at <console>:24
```

(2) 创建第二个 RDD

```
scala> val rdd2 = sc.parallelize(2 to 5)
```

```
rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[48] at parallelize at <console>:24
```

(3) 计算两个 RDD 的笛卡尔积并打印

```
scala> rdd1.cartesian(rdd2).collect()
```

```
res17: Array[(Int, Int)] = Array((1,2), (1,3), (1,4), (1,5), (2,2), (2,3), (2,4), (2,5), (3,2), (3,3), (3,4), (3,5))
```

2.3.2.5 zip(otherDataset)案例

1. 作用：将两个 RDD 组合成 Key/Value 形式的 RDD，这里默认两个 RDD 的 partition 数量以及元素数量都相同，否则会抛出异常。

2. 需求：创建两个 RDD，并将两个 RDD 组合到一起形成一个(k,v)RDD

(1) 创建第一个 RDD

```
scala> val rdd1 = sc.parallelize(Array(1,2,3),3)
```

```
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:24
```

(2) 创建第二个 RDD（与 1 分区数相同）

```
scala> val rdd2 = sc.parallelize(Array("a","b","c"),3)
```

```
rdd2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[2] at parallelize at <console>:24
```

(3) 第一个 RDD 组合第二个 RDD 并打印

```
scala> rdd1.zip(rdd2).collect
```

```
res1: Array[(Int, String)] = Array((1,a), (2,b), (3,c))
```

(4) 第二个 RDD 组合第一个 RDD 并打印

```
scala> rdd2.zip(rdd1).collect
```

```
res2: Array[(String, Int)] = Array((a,1), (b,2), (c,3))
```

(5) 创建第三个 RDD（与 1,2 分区数不同）

```
scala> val rdd3 = sc.parallelize(Array("a","b","c"),2)
```

```
rdd3: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[5] at parallelize at <console>:24
```

(6) 第一个 RDD 组合第三个 RDD 并打印

```
scala> rdd1.zip(rdd3).collect
java.lang.IllegalArgumentException: Can't zip RDDs with unequal numbers of partitions: List(3, 2)
  at org.apache.spark.rdd.ZippedPartitionsBaseRDD.getPartitions(ZippedPartitionsRDD.scala:57)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:252)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:250)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.rdd.RDD.partitions(RDD.scala:250)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1965)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1.apply(RDD.scala:936)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
  at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
  at org.apache.spark.rdd.RDD.collect(RDD.scala:935)
... 48 elided
```

2.3.3 Key-Value 类型

2.3.3.1 partitionBy 案例

1. 作用：对 pairRDD 进行分区操作，如果原有的 partitionRDD 和现有的 partitionRDD 是一致的话就不进行分区，否则会生成 ShuffleRDD，即会产生 shuffle 过程。

2. 需求：创建一个 4 个分区的 RDD，对其重新分区

(1) 创建一个 RDD

```
scala> val rdd = sc.parallelize(Array((1,"aaa"),(2,"bbb"),(3,"ccc"),(4,"ddd")),4)
rdd: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[44] at parallelize at <console>:24
```

(2) 查看 RDD 的分区数

```
scala> rdd.partitions.size
res24: Int = 4
```

(3) 对 RDD 重新分区

```
scala> var rdd2 = rdd.partitionBy(new org.apache.spark.HashPartitioner(2))
rdd2: org.apache.spark.rdd.RDD[(Int, String)] = ShuffledRDD[45] at partitionBy at <console>:26
```

(4) 查看新 RDD 的分区数

```
scala> rdd2.partitions.size
res25: Int = 2
```

2.3.3.2 groupByKey 案例

1. 作用：groupByKey 也是对每个 key 进行操作，但只生成一个 sequence。

2. 需求：创建一个 pairRDD，将相同 key 对应值聚合到一个 sequence 中，并计算相同 key 对应值的相加结果。

(1) 创建一个 pairRDD

```
scala> val words = Array("one", "two", "two", "three", "three", "three")
words: Array[String] = Array(one, two, two, three, three, three)
```

```
scala> val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))
```

```
wordPairsRDD: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[4] at map at <console>:26
```

(2) 将相同 key 对应值聚合到一个 sequence 中

```
scala> val group = wordPairsRDD.groupByKey()
group: org.apache.spark.rdd.RDD[(String, Iterable[Int])] = ShuffledRDD[5] at groupByKey at <console>:28
```

(3) 打印结果

```
scala> group.collect()
res1: Array[(String, Iterable[Int])] = Array((two,CompactBuffer(1, 1)), (one,CompactBuffer(1)), (three,CompactBuffer(1, 1)))
```

(4) 计算相同 key 对应值的相加结果

```
scala> group.map(t => (t._1, t._2.sum))
res2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[6] at map at <console>:31
```

(5) 打印结果

```
scala> res2.collect()
res3: Array[(String, Int)] = Array((two,2), (one,1), (three,3))
```

2.3.3.3 reduceByKey(func, [numTasks]) 案例

1. 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，reduce 任务的个数可以通过第二个可选的参数来设置。
2. 需求：创建一个 pairRDD，计算相同 key 对应值的相加结果

(1) 创建一个 pairRDD

```
scala> val rdd = sc.parallelize(List(("female",1),("male",5),("female",5),("male",2)))  
rdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[46] at parallelize at <console>:24
```

(2) 计算相同 key 对应值的相加结果

```
scala> val reduce = rdd.reduceByKey((x,y) => x+y)  
reduce: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[47] at reduceByKey at <console>:26
```

(3) 打印结果

```
scala> reduce.collect()  
res29: Array[(String, Int)] = Array((female,6), (male,7))
```

2.3.3.4 reduceByKey 和 groupByKey 的区别

1. reduceByKey: 按照 key 进行聚合，在 shuffle 之前有 combine（预聚合）操作，返回结果是 RDD[k,v]。
2. groupByKey: 按照 key 进行分组，直接进行 shuffle。
3. 开发指导: reduceByKey 比 groupByKey 更建议使用。但是需要注意是否会影响业务逻辑。

2.3.3.5 aggregateByKey 案例

参数: (zeroValue:U,[partitioner: Partitioner]) (seqOp: (U, V) => U,combOp: (U, U) => U)

1. 作用: 在 kv 对的 RDD 中，按 key 将 value 进行分组合并，合并时，将每个 value 和初始值作为 seq 函数的参数，进行计算，返回的结果作为一个新的 kv 对，然后再将结果按照 key 进行合并，最后将每个分组的 value 传递给 combine 函数进行计算（先将前两个 value 进行计算，将返回结果和下一个 value 传给 combine 函数，以此类推），将 key 与计算结果作为一个新的 kv 对输出。

2. 参数描述:

- (1) zeroValue: 给每一个分区中的每一个 key 一个初始值；
- (2) seqOp: 函数用于在每一个分区中用初始值逐步迭代 value；
- (3) combOp: 函数用于合并每个分区中的结果。

3. 需求: 创建一个 pairRDD，取出每个分区相同 key 对应值的最大值，然后相加

4. 需求分析

aggregateByKey()案例解析

需求: 取出每个分区相同key对应值的最大值，然后相加

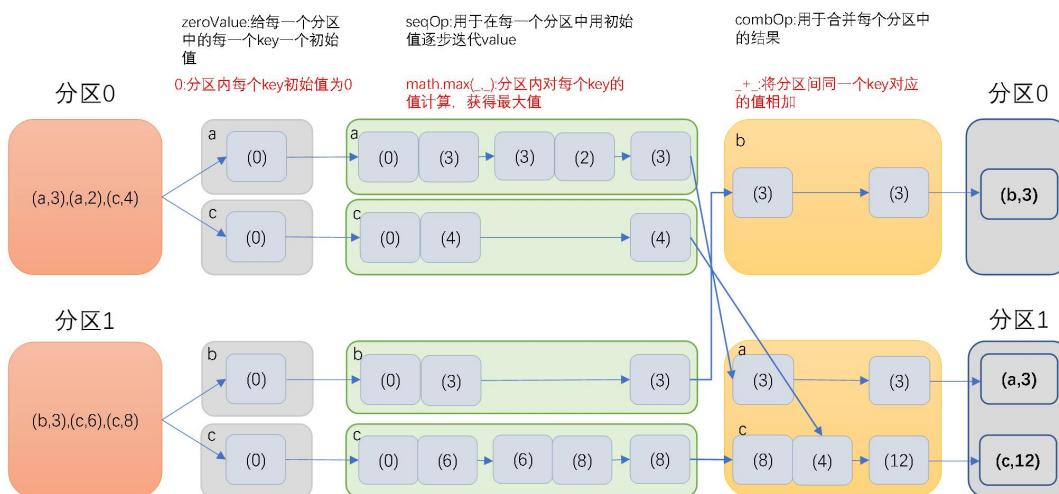


图 1-aggregate 案例分析

(1) 创建一个 pairRDD

```
scala> val rdd = sc.parallelize(List(("a",3),("a",2),("c",4),("b",3),("c",6),("c",8)),2)  
rdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[0] at parallelize at <console>:24
```

(2) 取出每个分区相同 key 对应值的最大值，然后相加

```
scala> val agg = rdd.aggregateByKey(0)(math.max(_,_),_+_)
```

```
agg: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[1] at aggregateByKey at <console>:26
```

(3) 打印结果

```
scala> agg.collect()
```

```
res0: Array[(String, Int)] = Array((b,3), (a,3), (c,12))
```

2.3.3.6 foldByKey 案例

参数: (zeroValue: V)(func: (V, V) => V): RDD[(K, V)]

1. 作用: aggregateByKey 的简化操作, seqop 和 combop 相同

2. 需求: 创建一个 pairRDD, 计算相同 key 对应值的相加结果

(1) 创建一个 pairRDD

```
scala> val rdd = sc.parallelize(List((1,3),(1,2),(1,4),(2,3),(3,6),(3,8)),3)
```

```
rdd: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[91] at parallelize at <console>:24
```

(2) 计算相同 key 对应值的相加结果

```
scala> val agg = rdd.foldByKey(0)(_+_)
```

```
agg: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[92] at foldByKey at <console>:26
```

(3) 打印结果

```
scala> agg.collect()
```

```
res61: Array[(Int, Int)] = Array((3,14), (1,9), (2,3))
```

2.3.3.7 combineByKey[C] 案例

参数: (createCombiner: V => C, mergeValue: (C, V) => C, mergeCombiners: (C, C) => C)

1. 作用: 对相同 K, 把 V 合并成一个集合。

2. 参数描述:

(1) createCombiner: combineByKey() 会遍历分区中的所有元素, 因此每个元素的键要么还没有遇到过, 要么就和之前的某个元素的键相同。如果这是一个新的元素, combineByKey()会使用一个叫作 createCombiner()的函数来创建那个键对应的累加器的初始值

(2) mergeValue: 如果这是一个在处理当前分区之前已经遇到的键, 它会使用 mergeValue()方法将该键的累加器对应的当前值与这个新的值进行合并

(3) mergeCombiners: 由于每个分区都是独立处理的, 因此对于同一个键可以有多个累加器。如果有两个或者更多的分区都有对应同一个键的累加器, 就需要使用用户提供的 mergeCombiners() 方法将各个分区的结果进行合并。

3. 需求: 创建一个 pairRDD, 根据 key 计算每种 key 的均值。(先计算每个 key 出现的次数以及可以对应值的总和, 再相除得到结果)

4. 需求分析:

combineByKey()案例分析

需求: 针对一个pairRDD, 计算每种key对应值的和

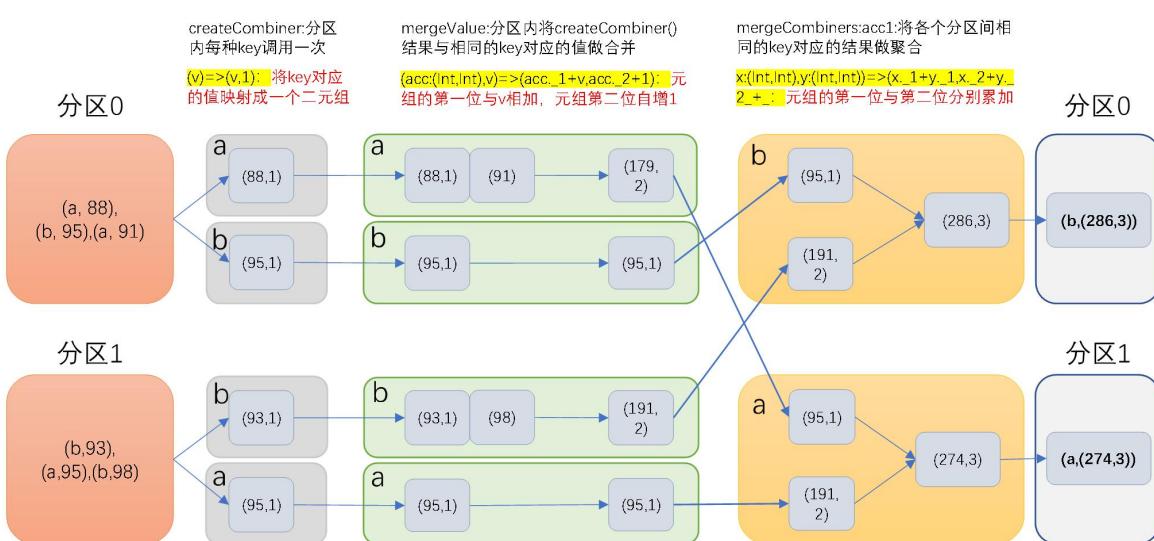


图 2- combineByKey 案例分析

(1) 创建一个 pairRDD

```
scala> val input = sc.parallelize(Array(("a", 88), ("b", 95), ("a", 91), ("b", 93), ("a", 95), ("b", 98)),2)
input: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[52] at parallelize at <console>:26
```

(2) 将相同 key 对应的值相加，同时记录该 key 出现的次数，放入一个二元组

```
scala> val combine = input.combineByKey(
    (_,_1), // 生成新元组
    (acc:(Int,Int),v) => (acc._1+v, acc._2+1), // 分区内的新元组 merge
    (acc1:(Int,Int),acc2:(Int,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2) // 所有分区的新元组 merge
)
combine: org.apache.spark.rdd.RDD[(String, (Int, Int))] = ShuffledRDD[5] at combineByKey at <console>:28
```

(3) 打印合并后的结果

```
scala> combine.collect
res5: Array[(String, (Int, Int))] = Array((b,(286,3)), (a,(274,3)))
```

(4) 计算平均值

```
scala> val result = combine.map{case (key,value) => (key,value._1/value._2.toDouble)}
result: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[54] at map at <console>:30
```

(5) 打印结果

```
scala> result.collect()
res33: Array[(String, Double)] = Array((b,95.33333333333333), (a,91.33333333333333))
```

2.3.3.8 sortByKey([ascending], [numTasks]) 案例

1. 作用：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

2. 需求：创建一个 pairRDD，按照 key 的正序和倒序进行排序

(1) 创建一个 pairRDD

```
scala> val rdd = sc.parallelize(Array((3,"aa"),(6,"cc"),(2,"bb"),(1,"dd")))
rdd: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[14] at parallelize at <console>:24
```

(2) 按照 key 的正序

```
scala> rdd.sortByKey(true).collect()
res9: Array[(Int, String)] = Array((1,dd), (2,bb), (3,aa), (6,cc))
```

(3) 按照 key 的倒序

```
scala> rdd.sortByKey(false).collect()
res10: Array[(Int, String)] = Array((6,cc), (3,aa), (2,bb), (1,dd))
```

2.3.3.9 mapValues 案例

1. 针对于(K,V)形式的类型只对 V 进行操作

2. 需求：创建一个 pairRDD，并将 value 添加字符串"|||"

(1) 创建一个 pairRDD

```
scala> val rdd3 = sc.parallelize(Array((1,"a"),(1,"d"),(2,"b"),(3,"c")))
rdd3: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[67] at parallelize at <console>:24
```

(2) 对 value 添加字符串"|||"

```
scala> rdd3.mapValues(_+"|||").collect()
res26: Array[(Int, String)] = Array((1,a|||), (1,d|||), (2,b|||), (3,c|||))
```

2.3.3.10 join(otherDataset, [numTasks]) 案例

1. 作用：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

2. 需求：创建两个 pairRDD，并将 key 相同的数据聚合到一个元组。

(1) 创建第一个 pairRDD

```
scala> val rdd = sc.parallelize(Array((1,"a"),(2,"b"),(3,"c")))
rdd: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[32] at parallelize at <console>:24
```

(2) 创建第二个 pairRDD

```
scala> val rdd1 = sc.parallelize(Array((1,4),(2,5),(3,6)))
rdd1: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[33] at parallelize at <console>:24
```

(3) join 操作并打印结果

```
scala> rdd.join(rdd1).collect()
res13: Array[(Int, (String, Int))] = Array((1,(a,4)), (2,(b,5)), (3,(c,6)))
```

2.3.3.11 cogroup(otherDataset, [numTasks]) 案例

1. 作用：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

2. 需求：创建两个 pairRDD，并将 key 相同的数据聚合到一个迭代器。

(1) 创建第一个 pairRDD

```
scala> val rdd = sc.parallelize(Array((1,"a"),(2,"b"),(3,"c")))
rdd: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[37] at parallelize at <console>:24
```

(2) 创建第二个 pairRDD

```
scala> val rdd1 = sc.parallelize(Array((1,4),(2,5),(3,6)))
rdd1: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[38] at parallelize at <console>:24
```

(3) cogroup 两个 RDD 并打印结果

```
scala> rdd.cogroup(rdd1).collect()
res14: Array[(Int, (Iterable[String], Iterable[Int]))] = Array((1,(CompactBuffer(a),CompactBuffer(4))), (2,(CompactBuffer(b),CompactBuffer(5))), (3,(CompactBuffer(c),CompactBuffer(6))))
```

2.3.4 案例实操

1. 数据结构：时间戳，省份，城市，用户，广告，中间字段使用空格分割。



agent.log

样本如下：

```
1516609143867 6 7 64 16
1516609143869 9 4 75 18
1516609143869 1 7 87 12
```

2. 需求：统计出每一个省份广告被点击次数的 TOP3

3. 实现过程：

```
package com.package.practice

import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}

//需求：统计出每一个省份广告被点击次数的 TOP3
object Practice {
    def main(args: Array[String]): Unit = {
        //1.初始化 spark 配置信息并建立与 spark 的连接
        val sparkConf = new SparkConf().setMaster("local[*]").setAppName("Practice")
        val sc = new SparkContext(sparkConf)
        //2.读取数据生成 RDD: TS, Province, City, User, AD
        val line = sc.textFile("E:\\IDEAWorkSpace\\SparkTest\\src\\main\\resources\\agent.log")
        //3.按照最小粒度聚合: ((Province,AD),1)
        val provinceAdToOne = line.map { x =>
            val fields: Array[String] = x.split(" ")
            ((fields(1), fields(4)), 1)
        }
        //4.计算每个省中每个广告被点击的总数: ((Province,AD),sum)
        val provinceAdToSum = provinceAdToOne.reduceByKey(_ + _)
        //5.将省份作为 key， 广告加点击数为 value: (Province,(AD,sum))
        val provinceToAdSum = provinceAdToSum.map(x => (x._1._1, (x._1._2, x._2)))
        //6.将同一个省份的所有广告进行聚合(Province,List((AD1,sum1),(AD2,sum2)...))
        val provinceGroup = provinceToAdSum.groupByKey()
        //7.对同一个省份所有广告的集合进行排序并取前 3 条， 排序规则为广告点击总数
        val provinceAdTop3 = provinceGroup.mapValues { x =>
            x.toList.sortWith((x, y) => x._2 > y._2).take(3)
        }
        //8.将数据拉取到 Driver 端并打印
        provinceAdTop3.collect().foreach(println)
    }
}
```

```
//9.关闭与 spark 的连接  
sc.stop()  
}  
}
```

2.4 Action

2.4.1 reduce(func)案例

1. 作用：通过 func 函数聚集 RDD 中的所有元素，先聚合分区内数据，再聚合分区间数据。

2. 需求：创建一个 RDD，将所有元素聚合得到结果。

(1) 创建一个 RDD[Int]

```
scala> val rdd1 = sc.makeRDD(1 to 10,2)  
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[85] at makeRDD at <console>:24
```

(2) 聚合 RDD[Int]所有元素

```
scala> rdd1.reduce(_+_)  
res50: Int = 55
```

(3) 创建一个 RDD[String]

```
scala> val rdd2 = sc.makeRDD(Array(("a",1),("a",3),("c",3),("d",5)))  
rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[86] at makeRDD at <console>:24
```

(4) 聚合 RDD[String]所有数据

```
scala> rdd2.reduce((x,y)=>(x._1 + y._1, x._2 + y._2))  
res51: (String, Int) = (adca,12)
```

2.4.2 collect()案例

1. 作用：在驱动程序中，以数组的形式返回数据集的所有元素。

2. 需求：创建一个 RDD，并将 RDD 内容收集到 Driver 端打印

(1) 创建一个 RDD

```
scala> val rdd = sc.parallelize(1 to 10)  
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24
```

(2) 将结果收集到 Driver 端

```
scala> rdd.collect  
res0: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

2.4.3 count()案例

1. 作用：返回 RDD 中元素的个数

2. 需求：创建一个 RDD，统计该 RDD 的条数

(1) 创建一个 RDD

```
scala> val rdd = sc.parallelize(1 to 10)  
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24
```

(2) 统计该 RDD 的条数

```
scala> rdd.count  
res1: Long = 10
```

2.4.4 first()案例

1. 作用：返回 RDD 中的第一个元素

2. 需求：创建一个 RDD，返回该 RDD 中的第一个元素

(1) 创建一个 RDD

```
scala> val rdd = sc.parallelize(1 to 10)  
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24
```

(2) 获取第 1 个元素

```
scala> rdd.first  
res2: Int = 1
```

2.4.5 take(n)案例

1. 作用：返回一个由 RDD 的前 n 个元素组成的数组

2. 需求：创建一个 RDD，返回前 3 条结果

(1) 创建一个 RDD

```
scala> val rdd = sc.parallelize(Array(2,5,4,6,8,3))
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:24
```

(2) 返回结果

```
scala> rdd.take(3)
res10: Array[Int] = Array(2, 5, 4)
```

2.4.6 takeOrdered(n)案例

1. 作用：返回该 RDD 排序后的前 n 个元素组成的数组

2. 需求：创建一个 RDD，返回排序后的前 3 条结果

(1) 创建一个 RDD

```
scala> val rdd = sc.parallelize(Array(2,5,4,6,8,3))
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:24
```

(2) 返回结果

```
scala> rdd.takeOrdered(3)
res18: Array[Int] = Array(2, 3, 4)
```

2.4.7 aggregate 案例

1. 参数：(zeroValue: U)(seqOp: (U, T) ⇒ U, combOp: (U, U) ⇒ U)

2. 作用：aggregate 函数将每个分区里面的元素通过 seqOp 和初始值进行聚合，然后用 combine 函数将每个分区的结果和初始值(zeroValue)进行 combine 操作。这个函数最终返回的类型不需要和 RDD 中元素类型一致。

3. 需求：创建一个 RDD，将所有元素相加得到结果

(1) 创建一个 RDD

```
scala> var rdd1 = sc.makeRDD(1 to 10,2)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[88] at makeRDD at <console>:24
```

(2) 将该 RDD 所有元素相加得到结果

```
scala> rdd.aggregate(0)(_+_ , _+_)
res22: Int = 55
```

2.4.8 fold(num)(func)案例

1. 作用：折叠操作， aggregate 的简化操作， seqop 和 combop 一样。

2. 需求：创建一个 RDD，将所有元素相加得到结果

(1) 创建一个 RDD

```
scala> var rdd1 = sc.makeRDD(1 to 10,2)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[88] at makeRDD at <console>:24
```

(2) 将该 RDD 所有元素相加得到结果

```
scala> rdd.fold(0)(_+_)
res24: Int = 55
```

2.4.9 saveAsTextFile(path)

作用：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，将它转换为文件中的文本

2.4.10 saveAsSequenceFile(path)

作用：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。

2.4.11 saveAsObjectFile(path)

作用：用于将 RDD 中的元素序列化成对象，存储到文件中。

2.4.12 countByKey()案例

1. 作用：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。

2. 需求：创建一个 PairRDD，统计每种 key 的个数

(1) 创建一个 PairRDD

```
scala> val rdd = sc.parallelize(List((1,3),(1,2),(1,4),(2,3),(3,6),(3,8)),3)
rdd: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[95] at parallelize at <console>:24
```

(2) 统计每种 key 的个数

```
scala> rdd.countByKey
```

```
res63: scala.collection.Map[Int,Long] = Map(3 -> 2, 1 -> 3, 2 -> 1)
```

2.4.13 foreach(func)案例

1. 作用：在数据集的每一个元素上，运行函数 func 进行更新。

2. 需求：创建一个 RDD，对每个元素进行打印

(1) 创建一个 RDD

```
scala> var rdd = sc.makeRDD(1 to 5, 2)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[107] at makeRDD at <console>:24
```

(2) 对该 RDD 每个元素进行打印

```
scala> rdd.foreach(println(_))
3
4
5
1
2
```

2.5 RDD 中的函数传递

在实际开发中我们往往需要自己定义一些对于 RDD 的操作，那么此时需要主要的是，初始化工作是在 **Driver** 端进行的，而实际运行程序是在 **Executor** 端进行的，这就涉及到了跨进程通信，是需要序列化的。下面我们看几个例子：

2.5.1 传递一个方法

1. 创建一个类

```
class Search{
    val query = "rules"

    //过滤出包含字符串的数据
    def isMatch(s: String): Boolean = {
        s.contains(query)
    }

    //过滤出包含字符串的 RDD
    def getMatch1(rdd: RDD[String]): RDD[String] = {
        rdd.filter(isMatch)
    }

    //过滤出包含字符串的 RDD
    def getMatche2(rdd: RDD[String]): RDD[String] = {
        rdd.filter(x => x.contains(query))
    }
}
```

2. 创建 Spark 主程序

```
object SeriTest {
    def main(args: Array[String]): Unit = {
        //1. 初始化配置信息及 SparkContext
        val sparkConf: SparkConf = new SparkConf().setAppName("WordCount").setMaster("local[*]")
        val sc = new SparkContext(sparkConf)

        //2. 创建一个 RDD
        val rdd: RDD[String] = sc.parallelize(Array("hadoop", "spark", "hive", "atguigu"))

        //3. 创建一个 Search 对象
        val search = new Search()

        //4. 运用第一个过滤函数并打印结果
        val match1: RDD[String] = search.getMatche1(rdd)
```

```
    match1.collect().foreach(println)
}
}
```

3. 运行程序

```
Exception in thread "main" org.apache.spark.SparkException: Task not serializable
  at org.apache.spark.util.ClosureCleaner$.ensureSerializable(ClosureCleaner.scala:298)
  at org.apache.spark.util.ClosureCleaner$.org$apache$spark$util$ClosureCleaner$$clean(ClosureCleaner.scala:288)
  at org.apache.spark.util.ClosureCleaner$.clean(ClosureCleaner.scala:108)
  at org.apache.spark.SparkContext.clean(SparkContext.scala:2101)
  at org.apache.spark.rdd.RDD$$anonfun$filter$1.apply(RDD.scala:387)
  at org.apache.spark.rdd.RDD$$anonfun$filter$1.apply(RDD.scala:386)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
  at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
  at org.apache.spark.rdd.RDD.filter(RDD.scala:386)
  at com.atguigu.Search.getMatche1(SeriTest.scala:39)
  at com.atguigu.SeriTest$.main(SeriTest.scala:18)
  at com.atguigu.SeriTest.main(SeriTest.scala)
Caused by: java.io.NotSerializableException: com.atguigu.Search
```

4. 问题说明

```
//过滤出包含字符串的 RDD
def getMatch1 (rdd: RDD[String]): RDD[String] = {
  rdd.filter(isMatch)
}
```

在这个方法中所调用的方法 `isMatch()` 是定义在 `Search` 这个类中的，实际上调用的是 `this.isMatch()`，`this` 表示 `Search` 这个类的对象，程序在运行过程中需要将 `Search` 对象序列化以后传递到 `Executor` 端。

5. 解决方案

使类继承 `scala.Serializable` 即可。

```
class Search() extends Serializable{...}
```

2.5.2 传递一个属性

1. 创建 Spark 主程序

```
object TransmitTest {
  def main(args: Array[String]): Unit = {
    //1. 初始化配置信息及 SparkContext
    val sparkConf: SparkConf = new SparkConf().setAppName("WordCount").setMaster("local[*]")
    val sc = new SparkContext(sparkConf)

    //2. 创建一个 RDD
    val rdd: RDD[String] = sc.parallelize(Array("hadoop", "spark", "hive", "atguigu"))

    //3. 创建一个 Search 对象
    val search = new Search()

    //4. 运用第一个过滤函数并打印结果
    val match1: RDD[String] = search.getMatche2(rdd)
    match1.collect().foreach(println)
  }
}
```

2. 运行程序

```
Exception in thread "main" org.apache.spark.SparkException: Task not serializable
  at org.apache.spark.util.ClosureCleaner$.ensureSerializable(ClosureCleaner.scala:298)
  at org.apache.spark.util.ClosureCleaner$.org$apache$spark$util$ClosureCleaner$$clean(ClosureCleaner.scala:288)
  at org.apache.spark.util.ClosureCleaner$.clean(ClosureCleaner.scala:108)
  at org.apache.spark.SparkContext.clean(SparkContext.scala:2101)
  at org.apache.spark.rdd.RDD$$anonfun$filter$1.apply(RDD.scala:387)
```

```

at org.apache.spark.rdd.RDD$$anonfun$filter$1.apply(RDD.scala:386)
at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
at org.apache.spark.rdd.RDD.filter(RDD.scala:386)
at com.atguigu.Search.getMatche1(SeriTest.scala:39)
at com.atguigu.SeriTest$.main(SeriTest.scala:18)
at com.atguigu.SeriTest.main(SeriTest.scala)
Caused by: java.io.NotSerializableException: com.atguigu.Search

```

3. 问题说明

```

//过滤出包含字符串的 RDD
def getMatche2(rdd: RDD[String]): RDD[String] = {
    rdd.filter(x => x.contains(query))
}

```

在这个方法中所调用的方法 `query` 是定义在 `Search` 这个类中的字段，实际上调用的是 `this.query`，`this` 表示 `Search` 这个类的对象，程序在运行过程中需要将 `Search` 对象序列化以后传递到 `Executor` 端。

4. 解决方案

方案 1 使类继承 `scala.Serializable` 即可。

```
class Search() extends Serializable{...}
```

方案 2 将类变量 `query` 赋值给局部变量

修改 `getMatche2` 为

```

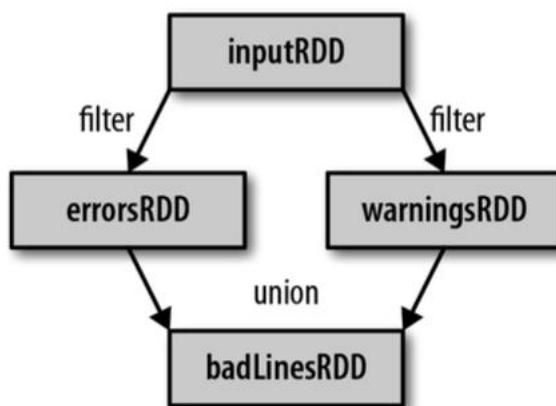
//过滤出包含字符串的 RDD
def getMatche2(rdd: RDD[String]): RDD[String] = {
    val query_ : String = this.query // 将类变量赋值给局部变量
    rdd.filter(x => x.contains(query_))
}

```

2.6 RDD 依赖关系

2.6.1 血缘

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（血缘）记录下来，以便恢复丢失的分区。RDD 的血缘会记录 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。



```

//读取一个 HDFS 文件并将其中内容映射成一个个元组
val wordAndOne = sc.textFile("/fruit.tsv").flatMap(_.split("\t")).map((_,1))
//统计每一种 key 对应的个数
val wordAndCount = wordAndOne.reduceByKey(_+_)
//查看“wordAndOne”的 Lineage
wordAndOne.toDebugString
(2) MapPartitionsRDD[22] at map at <console>:24 []
|  MapPartitionsRDD[21] at flatMap at <console>:24 []
|  /fruit.tsv MapPartitionsRDD[20] at textFile at <console>:24 []
|  /fruit.tsv HadoopRDD[19] at textFile at <console>:24 []

```

```

//查看“wordAndCount”的Lineage
wordAndCount.toDebugString
(2) ShuffledRDD[23] at reduceByKey at <console>:26 []
+- (2) MapPartitionsRDD[22] at map at <console>:24 []
  |  MapPartitionsRDD[21] at flatMap at <console>:24 []
  |  /fruit.tsv MapPartitionsRDD[20] at textFile at <console>:24 []
  |  /fruit.tsv HadoopRDD[19] at textFile at <console>:24 []

//查看“wordAndOne”的依赖类型
wordAndOne.dependencies

//查看“wordAndCount”的依赖类型
wordAndCount.dependencies

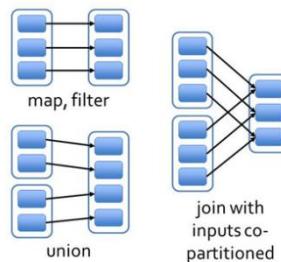
```

注意：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）。

2.6.2 窄依赖

窄依赖指的是父 RDD 的每一个分区最多被子 RDD 的一个分区使用

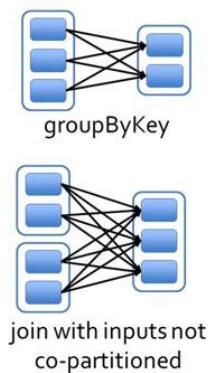
“Narrow” deps:



2.6.3 宽依赖

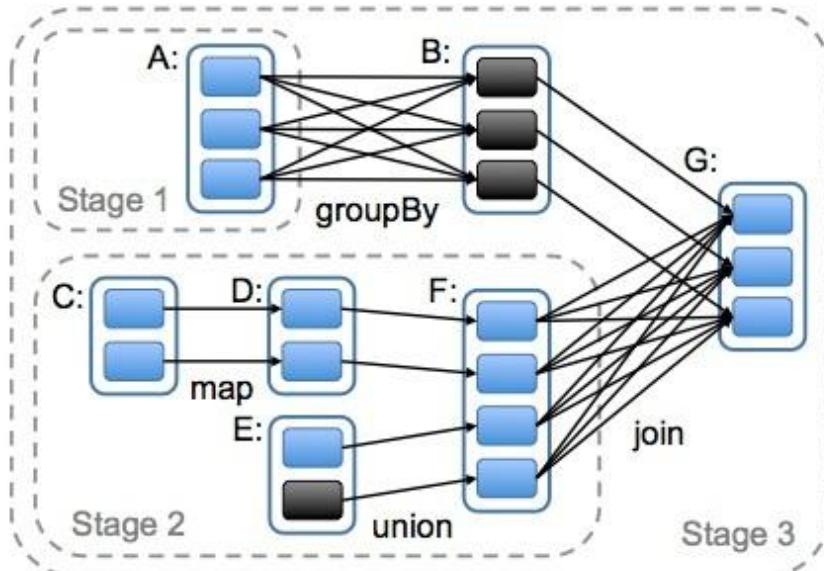
宽依赖指的是子 RDD 的多个分区会依赖父 RDD 的同一个分区，会引起 shuffle

“Wide” (shuffle) deps:



2.6.4 DAG

DAG(Directed Acyclic Graph)叫有向无环图，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。



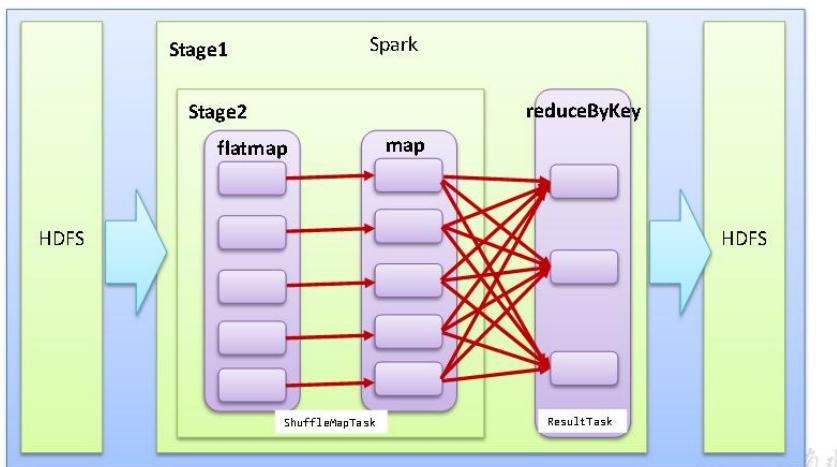
2.6.5 任务划分（面试重点）

RDD 任务划分中间分为：Application、Job、Stage 和 Task

- 1) Application: 初始化一个 `SparkContext` 即生成一个 Application
- 2) Job: 一个 Action 算子就会生成一个 Job
- 3) Stage: 根据 RDD 之间的依赖关系的不同将 Job 划分成不同的 Stage，遇到一个宽依赖则划分一个 Stage。
- 4) Task: Stage 是一个 TaskSet，将 Stage 划分的结果发送到不同的 Executor 执行即为一个 Task。

注意：Application->Job->Stage->Task 每一层都是 1 对 n 的关系。

```
sc.textFile("xx").flatMap(_.split(" ")).map((_,1)).reduceByKey(_+_).saveAsTextFile("xx")
```



2.7 RDD 缓存

RDD 通过 `persist` 方法或 `cache` 方法可以将前面的计算结果缓存，默认情况下 `persist()` 会把数据以序列化的形式缓存在 JVM 的堆空间中。

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。

```
/** Persist this RDD with the default storage level (MEMORY_ONLY). */
def persist(): this.type = persist(StorageLevel.MEMORY_ONLY)
```

```
/** Persist this RDD with the default storage level (MEMORY_ONLY). */
def cache(): this.type = persist()
```

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)
}
```

在存储级别的末尾加上“_2”来把持久化数据存为两份

级别	使用的空间	CPU时间	是否在内存中	是否在磁盘上	备注
MEMORY_ONLY	高	低	是	否	
MEMORY_ONLY_SER	低	高	是	否	
MEMORY_AND_DISK	高	中等	部分	部分	如果数据在内存中放不下，则溢写到磁盘上
MEMORY_AND_DISK_SER	低	高	部分	部分	如果数据在内存中放不下，则溢写到磁盘上。在内存中存放序列化后的数据
DISK_ONLY	低	高	否	是	

缓存有可能丢失，或者存储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。

```
//创建一个 RDD
val rdd = sc.makeRDD(Array("atguigu"))

//将 RDD 转换为携带当前时间戳，不做缓存
val nocache = rdd.map(_.toString+System.currentTimeMillis)
//多次打印结果
nocache.collect
res0: Array[String] = Array(atguigu1538978275359)
nocache.collect
res1: Array[String] = Array(atguigu1538978282416)
nocache.collect
res2: Array[String] = Array(atguigu1538978283199)

//将 RDD 转换为携带当前时间戳，并做缓存
scala> val cache = rdd.map(_.toString+System.currentTimeMillis).cache
cache: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[21] at map at <console>:27
//多次打印做了缓存的结果
cache.collect
res3: Array[String] = Array(atguigu1538978435705)
cache.collect
res4: Array[String] = Array(atguigu1538978435705)
cache.collect
res5: Array[String] = Array(atguigu1538978435705)
```

2.8 RDD CheckPoint

Spark 中对于数据的保存除了持久化操作之外，还提供了一种检查点的机制，检查点（本质是通过将 RDD 写入 Disk 做检查点）是为了通过 lineage 做容错的辅助，lineage 过长会使得容错成本过高，这样就不如在中间阶段做检查点容错，如果之后有节点出现问题而丢失分区，从做检查点的 RDD 开始重做 Lineage，就会减少开销。检查点通过将数据写入到 HDFS 文件系统实现了 RDD 的检查点功能。

为当前 RDD 设置检查点。该函数将会创建一个二进制的文件，并存储到 checkpoint 目录中，该目录是用 Spark Context.setCheckpointDir() 设置的。在 checkpoint 的过程中，该 RDD 的所有依赖于父 RDD 中的信息将全部被移除。对 RDD 进行 checkpoint 操作并不会马上被执行，必须执行 Action 操作才能触发。

案例实操：

```
//设置检查点
sc.setCheckpointDir("hdfs://hadoop102:9000/checkpoint")
//创建一个 RDD
val rdd = sc.parallelize(Array("atguigu"))

//将 RDD 转换为携带当前时间戳，并做 checkpoint
val ch = rdd.map(_+System.currentTimeMillis)
ch.checkpoint

//多次打印结果
ch.collect
res55: Array[String] = Array(atguigu1538981860336)
ch.collect
res56: Array[String] = Array(atguigu1538981860504)
ch.collect
res57: Array[String] = Array(atguigu1538981860504)
ch.collect
res58: Array[String] = Array(atguigu1538981860504)
```

第 3 章 键值对 RDD 数据分区器

Spark 目前支持 Hash 分区和 Range 分区，用户也可以自定义分区，Hash 分区为当前的默认分区，Spark 中分区器直接决定了 RDD 中分区的个数、RDD 中每条数据经过 Shuffle 过程属于哪个分区和 Reduce 的个数。

注意：

- (1) 只有 Key-Value 类型的 RDD 才有分区器的，非 Key-Value 类型的 RDD 分区器的值是 None
- (2) 每个 RDD 的分区 ID 范围：0~numPartitions-1，决定这个值是属于那个分区的。

3.1 获得 RDD 分区

可以通过使用 RDD 的 partitioner 属性来获取 RDD 的分区方式。它会返回一个 scala.Option 对象，通过 get 方法获取其中的值。相关源码如下：

```
def getPartition(key: Any): Int = key match {
  case null => 0
  case _ => Utils.nonNegativeMod(key.hashCode, numPartitions)
}
```

```
def nonNegativeMod(x: Int, mod: Int): Int = {
  val rawMod = x % mod
  rawMod + (if (rawMod < 0) mod else 0)
}
```

使用：

```
//创建一个 pairRDD
val pairs = sc.parallelize(List((1,1),(2,2),(3,3)))
//查看 RDD 的分区器
pairs.partition
//导入 HashPartitioner 类
import org.apache.spark.HashPartitioner
```

```
//使用 HashPartitioner 对 RDD 进行重新分区
val partitioned = pairs.partitionBy(new HashPartitioner(2))
//查看重新分区后 RDD 的分区器
partitioned.partitioner
res2: Option[org.apache.spark.Partitioner] = Some(org.apache.spark.HashPartitioner@2)
```

3.2 Hash 分区

HashPartitioner 分区的原理：对于给定的 key，计算其 hashCode，并除以分区的个数取余，如果余数小于 0，则用余数+分区的个数（否则加 0），最后返回的值就是这个 key 所属的分区 ID。

HashPartitioner 分区弊端：可能导致每个分区中数据量的不均匀，极端情况下会导致某些分区拥有 RDD 的全部数据。

使用 Hash 分区的实操

```
scala> nopar.partitioner
res20: Option[org.apache.spark.Partitioner] = None

scala> val nopar = sc.parallelize(List((1,3),(1,2),(2,4),(2,3),(3,6),(3,8)),8)
nopar: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[10] at parallelize at <console>:24

scala> nopar.mapPartitionsWithIndex((index,iter)=>{ Iterator(index.toString+" : "+iter.mkString(" | ")) }).collect
res0: Array[String] = Array("0 : ", 1 : (1,3), 2 : (1,2), 3 : (2,4), "4 : ", 5 : (2,3), 6 : (3,6), 7 : (3,8))

scala> val hashpar = nopar.partitionBy(new org.apache.spark.HashPartitioner(7))
hashpar: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[12] at partitionBy at <console>:26

scala> hashpar.count
res18: Long = 6

scala> hashpar.partitioner
res21: Option[org.apache.spark.Partitioner] = Some(org.apache.spark.HashPartitioner@7)

scala> hashpar.mapPartitions(iter => Iterator(iter.length)).collect()
res19: Array[Int] = Array(0, 3, 1, 2, 0, 0, 0)
```

3.3 Ranger 分区

RangePartitioner 作用：将一定范围内的数映射到某一个分区内，尽量保证每个分区中数据量的均匀，而且分区与分区之间是有序的，一个分区中的元素肯定都是比另一个分区内的元素小或者大，但是分区内的元素是不能保证顺序的。简单的说就是将一定范围内的数映射到某一个分区内。实现过程为：

第一步：先从整个 RDD 中抽取出样本数据，将样本数据排序，计算出每个分区的最大 key 值，形成一个 Array[KEY]类型的数组变量 rangeBounds；

第二步：判断 key 在 rangeBounds 中所处的范围，给出该 key 值在下一个 RDD 中的分区 id 下标；该分区器要求 RDD 中的 KEY 类型必须是可以排序的

3.4 自定义分区

要实现自定义的分区器，你需要继承 `org.apache.spark.Partitioner` 类并实现下面三个方法。

- (1) `numPartitions: Int`: 返回创建出来的分区数。
- (2) `getPartition(key: Any): Int`: 返回给定键的分区编号(0 到 `numPartitions-1`)。
- (3) `equals(): Java` 判断相等性的标准方法。这个方法的实现非常重要，Spark 需要用这个方法来检查你的分区器对象是否和其他分区器实例相同，这样 Spark 才可以判断两个 RDD 的分区方式是否相同。

需求：将相同后缀的数据写入相同的文件，通过将相同后缀的数据分区到相同的分区并保存输出来实现。

```
//创建一个 pairRDD
val data = sc.parallelize(Array((1,1),(2,2),(3,3),(4,4),(5,5),(6,6)))
//定义一个自定义分区类
class CustomerPartitioner(numParts:Int) extends org.apache.spark.Partitioner{
```

```

// 这个方法需要返回你想要创建分区的个数
override def numPartitions: Int = numParts

// 重写分区号获取函数
// 这个函数需要对输入的 key 做计算，然后返回该 key 的分区 ID，范围一定是 0 到 numPartitions-1
override def getPartition(key: Any): Int = {
    // val ckey: String = key.toString
    // ckey.substring(ckey.length - 1).toInt % numParts

    // 假设有 3 个分区，cba 放到分区 0，nba 放到分区 1，否则放到分区 2
    if ( key.isInstanceOf[String] ) {
        val keyString: String = key.asInstanceOf[String]
        if ( keyString == "cba" ) {
            //自定义将 keystring=cba 的放在 0 号分区中
            0
        }
        else if ( keyString == "nba" ) {
            1
        } else { 2 }
    } else {
        2
    }
}
}

// 将 RDD 使用自定义的分区类进行重新分区
val par = data.partitionBy(new CustomerPartitioner(2))
// 查看重新分区后的数据分布
par.mapPartitionsWithIndex((index,items)=>items.map((index,_))).collect

```

使用自定义的 Partitioner 是很容易的：只要把它传给 partitionBy() 方法即可。Spark 中有许多依赖于数据混洗的方法，比如 join() 和 groupByKey()，它们也可以接收一个可选的 Partitioner 对象来控制输出数据的分区方式。

第 4 章 数据读取与保存

Spark 的数据读取及数据保存可以从两个维度来作区分：文件格式以及文件系统。

文件格式分为：Text 文件、Json 文件、Csv 文件、Sequence 文件以及 Object 文件；

文件系统分为：本地文件系统、HDFS、HBASE 以及数据库。

4.1 文件类数据读取与保存

4.1.1 Text 文件

1) 数据读取: textFile(String)

```
val hdfsFile = sc.textFile("hdfs://hadoop102:9000/fruit.txt")
```

2) 数据保存: saveAsTextFile(String)

```
hdfsFile.saveAsTextFile("/fruitOut")
```

/fruitOut								Go!
Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	
-rw-r--r--	atguigu	supergroup	0 B	2018/10/8 下午3:47:33	1	128 MB	_SUCCESS	
-rw-r--r--	atguigu	supergroup	0 B	2018/10/8 下午3:47:33	1	128 MB	part-00000	

4.1.2 Json 文件

如果 JSON 文件中每一行就是一个 JSON 记录，那么可以通过将 JSON 文件当做文本文件来读取，然后利用相关的 JSON 库对每一条数据进行 JSON 解析。

注意：使用 RDD 读取 JSON 文件处理很复杂，同时 SparkSQL 集成了很好的处理 JSON 文件的方式，所以应用中多是采用 SparkSQL 处理 JSON 文件。

```
(1) 导入解析 json 所需的包
import scala.util.parsing.json.JSON
(2) 上传 json 文件到 HDFS
[atguigu@hadoop102 spark]$ hadoop fs -put ./examples/src/main/resources/people.json /
(3) 读取文件
val json = sc.textFile("/people.json")
(4) 解析 json 数据
val result = json.map(JSON.parseFull)
(5) 打印
result.collect
```

4.1.3 Sequence 文件

SequenceFile 文件是 Hadoop 用来存储二进制形式的 key-value 对而设计的一种平面文件(Flat File)。Spark 有专门用来读取 SequenceFile 的接口。在 `SparkContext` 中，可以调用 `sequenceFile[keyClass, valueClass](path)`。

注意：SequenceFile 文件只针对 `PairRDD`

```
(1) 创建一个 RDD
val rdd = sc.parallelize(Array((1,2),(3,4),(5,6)))
(2) 将 RDD 保存为 Sequence 文件
rdd.saveAsSequenceFile("file:///opt/module/spark/seqFile")
(3) 查看该文件
[atguigu@hadoop102 seqFile]$ pwd
/opt/module/spark/seqFile
[atguigu@hadoop102 seqFile]$ ll
总用量 8
-rw-r--r-- 1 atguigu atguigu 108 10月 9 10:29 part-00000
-rw-r--r-- 1 atguigu atguigu 124 10月 9 10:29 part-00001
-rw-r--r-- 1 atguigu atguigu 0 10月 9 10:29 _SUCCESS
[atguigu@hadoop102 seqFile]$ cat part-00000
SEQ org.apache.hadoop.io.IntWritable org.apache.hadoop.io.IntWritable
(4) 读取 Sequence 文件
val seq = sc.sequenceFile[Int,Int]("file:///opt/module/spark/seqFile")
(5) 打印读取后的 Sequence 文件
seq.collect
```

4.1.4 对象文件

对象文件是将对象序列化后保存的文件，采用 Java 的序列化机制。可以通过 `objectFile[k,v](path)` 函数接收一个路径，读取对象文件，返回对应的 `RDD`，也可以通过调用 `saveAsObjectFile()` 实现对对象文件的输出。因为是序列化，所以要指定类型。

```
(1) 创建一个 RDD
val rdd = sc.parallelize(Array(1,2,3,4))
(2) 将 RDD 保存为 Object 文件
rdd.saveAsObjectFile("file:///opt/module/spark/objectFile")
(3) 查看该文件
[atguigu@hadoop102 objectFile]$ pwd
/opt/module/spark/objectFile
[atguigu@hadoop102 objectFile]$ ll
总用量 8
-rw-r--r-- 1 atguigu atguigu 142 10月 9 10:37 part-00000
-rw-r--r-- 1 atguigu atguigu 142 10月 9 10:37 part-00001
-rw-r--r-- 1 atguigu atguigu 0 10月 9 10:37 _SUCCESS

[atguigu@hadoop102 objectFile]$ cat part-00000
SEQ!org.apache.hadoop.io.NullWritable"org.apache.hadoop.io.BytesWritableW@`I
(4) 读取 Object 文件
val objFile = sc.objectFile[Int]("file:///opt/module/spark/objectFile")
(5) 打印读取后的 Sequence 文件
```

```
objFile.collect  
res19: Array[Int] = Array(1, 2, 3, 4)
```

4.2 文件系统类数据读取与保存

4.2.1 HDFS

Spark 的整个生态系统与 Hadoop 是完全兼容的，所以对于 Hadoop 所支持的文件类型或者数据库类型，Spark 也同样支持。另外，由于 Hadoop 的 API 有新旧两个版本，所以 Spark 为了能够兼容 Hadoop 所有的版本，也提供了两套创建操作接口。对于外部存储创建操作而言，`hadoopRDD` 和 `newHadoopRDD` 是最为抽象的两个函数接口，主要包含以下四个参数。

1) 输入格式(`InputFormat`): 制定数据输入的类型，如 `TextInputFormat` 等，新旧两个版本所引用的版本分别是 `org.apache.hadoop.mapred.InputFormat` 和 `org.apache.hadoop.mapreduce.InputFormat(NewInputFormat)`

2) 键类型: 指定[K,V]键值对中 K 的类型

3) 值类型: 指定[K,V]键值对中 V 的类型

4) 分区值: 指定由外部存储生成的 RDD 的 `partition` 数量的最小值，如果没有指定，系统会使用默认值 `defaultMinSplits`

注意: 其他创建操作的 API 接口都是为了方便最终的 Spark 程序开发者而设置的，是这两个接口的高效实现版本。例如，对于 `textFile` 而言，只有 `path` 这个指定文件路径的参数，其他参数在系统内部指定了默认值。

1. 在 Hadoop 中以压缩形式存储的数据，不需要指定解压方式就能够进行读取，因为 Hadoop 本身有一个解压器会根据压缩文件的后缀推断解压算法进行解压。

2. 如果用 Spark 从 Hadoop 中读取某种类型的数据不知道怎么读取的时候，上网查找一个使用 `map-reduce` 的时候是怎么读取这种数据的，然后再将对应的读取方式改写成上面的 `hadoopRDD` 和 `newAPIHadoopRDD` 两个类就行了。

4.2.2 MySQL 数据库连接

支持通过 Java JDBC 访问关系型数据库，需要通过 `JdbcRDD` 进行，示例如下：

(1) 添加依赖

```
<dependency>  
    <groupId>mysql</groupId>  
    <artifactId>mysql-connector-java</artifactId>  
    <version>5.1.27</version>  
</dependency>
```

(2) Mysql 读取：

```
import java.sql.DriverManager  
import org.apache.spark.rdd.JdbcRDD  
import org.apache.spark.{SparkConf, SparkContext}  
  
object MysqlRDD {  
    def main(args: Array[String]): Unit = {  
        //1. 创建 spark 配置信息  
        val sparkConf: SparkConf = new SparkConf().setMaster("local[*]").setAppName("JdbcRDD")  
  
        //2. 创建 SparkContext  
        val sc = new SparkContext(sparkConf)  
  
        //3. 定义连接 mysql 的参数  
        val driver = "com.mysql.jdbc.Driver"  
        val url = "jdbc:mysql://hadoop102:3306/rdd"  
        val userName = "root"  
        val passWd = "000000"  
  
        // 创建 JdbcRDD  
        val rdd = new JdbcRDD(sc, () => {  
            Class.forName(driver)
```

```

        DriverManager.getConnection(url, userName, passWd)
    },
    "select * from `rddtable` where `id`>=?",
    1,
    10,
    1,
    r => (r.getInt(1), r.getString(2))
)

//打印最后结果
println(rdd.count())
rdd.foreach(println)

sc.stop()
}
}

```

Mysql 写入：

```

def main(args: Array[String]) {
    val sparkConf = new SparkConf().setMaster("local[2]").setAppName("HBaseApp")
    val sc = new SparkContext(sparkConf)
    val data = sc.parallelize(List("Female", "Male", "Female"))

    data.foreachPartition(insertData)
}

def insertData(iterator: Iterator[String]): Unit = {
    Class.forName("com.mysql.jdbc.Driver").newInstance() //newInstance() 用类加载机制创建对象
    val conn = java.sql.DriverManager.getConnection("jdbc:mysql://hadoop102:3306/rdd", "root", "000000")
    iterator.foreach(data => {
        val ps = conn.prepareStatement("insert into rddtable(name) values (?)")
        ps.setString(1, data)
        ps.executeUpdate()
    })
}

```

注：Class.forName("com.mysql.jdbc.Driver") 就已经初始化了一个 Driver 对象，不需要使用 newInstance()，因为 com.mysql.jdbc.Driver 这个类中有一块静态代码块，在 Class.forName 加载完驱动类，开始执行静态初始化代码时就会自动创建一个 Driver 对象。

4.2.3 HBase 数据库

由于 org.apache.hadoop.hbase.mapreduce.TableInputFormat 类的实现，Spark 可以通过 Hadoop 输入格式访问 HBase。这个输入格式会返回键值对数据，其中键的类型为 org.apache.hadoop.io.ImmutableBytesWritable，而值的类型为 org.apache.hadoop.hbase.client.Result。

(1) 添加依赖

```

<dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-server</artifactId>
    <version>1.3.1</version>
</dependency>

```

```

<dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-client</artifactId>
    <version>1.3.1</version>
</dependency>

```

(2) 从 HBase 读取

```

import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.hbase.{HBaseConfiguration, HColumnDescriptor, HTableDescriptor, TableName}

```

```

import org.apache.hadoop.hbase.client.{HBaseAdmin, Put, Result}
import org.apache.hadoop.hbase.io.ImmutableBytesWritable
import org.apache.hadoop.hbase.mapred.TableOutputFormat
import org.apache.hadoop.hbase.mapreduce.TableInputFormat
import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.hadoop.hbase.util.Bytes
import org.apache.hadoop.mapred.JobConf

object HBaseSpark {
  def main(args: Array[String]): Unit = {
    //创建 spark 配置信息
    val sparkConf: SparkConf = new SparkConf().setMaster("local[*]").setAppName("JdbcRDD")

    //创建 SparkContext
    val sc = new SparkContext(sparkConf)

    //构建 HBase 配置信息
    val conf: Configuration = HBaseConfiguration.create()
    conf.set("hbase.zookeeper.quorum", "hadoop102,hadoop103,hadoop104") //zookeeper 数
    conf.set(TableInputFormat.INPUT_TABLE, "rddtable") //设定读取的 HBase 的表是 rddtable

    //从 HBase 读取数据形成 RDD
    val hbaseRDD: RDD[(ImmutableBytesWritable, Result)] = sc.newAPIHadoopRDD(
      conf,
      classOf[TableInputFormat],
      classOf[ImmutableBytesWritable], //返回的键的类型
      classOf[Result]) //返回的值的类型

    val count: Long = hbaseRDD.count()
    println(count)

    //对 hbaseRDD 进行处理
    hbaseRDD.foreach {
      //匹配的变量模式， result 的意思是等同于下划线_， 同时为_赋上别名 result， 这样方便取出值来用
      case (_, result) =>
        val key: String = Bytes.toString(result.getRow)
        val name: String = Bytes.toString(result.getValue(Bytes.toBytes("info"), Bytes.toBytes("name")))
        val color: String = Bytes.toString(result.getValue(Bytes.toBytes("info"), Bytes.toBytes("color")))
        println("RowKey:" + key + ",Name:" + name + ",Color:" + color)
    }
  }

  //关闭连接
  sc.stop()
}

}

```

3) 输出到 HBase

```

def main(args: Array[String]) {
  //获取 Spark 配置信息并创建与 spark 的连接
  val sparkConf = new SparkConf().setMaster("local[*]").setAppName("HBaseApp")
  val sc = new SparkContext(sparkConf)

  //创建 HBaseConf
  val conf = HBaseConfiguration.create()
  val jobConf = new JobConf(conf)
  jobConf.setOutputFormat(classOf[TableOutputFormat])
  jobConf.set(TableOutputFormat.OUTPUT_TABLE, "fruit_spark")
}

```

```

//构建 Hbase 表描述器
val fruitTable = TableName.valueOf("fruit_spark")
val tableDescr = new HTableDescriptor(fruitTable)
tableDescr.addFamily(new HColumnDescriptor("info".getBytes))

//创建 Hbase 表
val admin = new HBaseAdmin(conf)
if (admin.tableExists(fruitTable)) {
    admin.disableTable(fruitTable)
    admin.deleteTable(fruitTable)
}
admin.createTable(tableDescr)

//定义往 Hbase 插入数据的方法
def convert(triple: (Int, String, Int)) = {
    val put = new Put(Bytes.toBytes(triple._1))
    put.addImmutable(Bytes.toBytes("info"), Bytes.toBytes("name"), Bytes.toBytes(triple._2))
    put.addImmutable(Bytes.toBytes("info"), Bytes.toBytes("price"), Bytes.toBytes(triple._3))
    (new ImmutableBytesWritable, put)
}

//创建一个 RDD
val initialRDD = sc.parallelize(List((1,"apple",11), (2,"banana",12), (3,"pear",13)))

//将 RDD 内容写到 HBase
val localData = initialRDD.map(convert)

localData.saveAsHadoopDataset(jobConf)
}

```

第 5 章 RDD 编程进阶

5.1 累加器

累加器用来对信息进行聚合，通常在向 Spark 传递函数时，比如使用 `map()` 函数或者用 `filter()` 传条件时，可以使用户驱动器程序中定义的变量，但是集群中运行的每个任务都会得到这些变量的一份新的副本，更新这些副本的值也不会影响驱动器中的对应变量。如果我们想实现所有分片处理时更新共享变量的功能，那么累加器可以实现我们想要的效果。

5.1.1 系统累加器

针对一个输入的日志文件，如果我们想计算文件中所有空行的数量，我们可以编写以下程序：

```

scala> val notice = sc.textFile("./NOTICE")
notice: org.apache.spark.rdd.RDD[String] = ./NOTICE MapPartitionsRDD[40] at textFile at <console>:32

scala> val blanklines = sc.accumulator(0)
warning: there were two deprecation warnings; re-run with -deprecation for details
blanklines: org.apache.spark.Accumulator[Int] = 0

scala> val tmp = notice.flatMap(line => {
    |     if (line == "") {
    |         blanklines += 1
    |     }
    |     line.split(" ")
    | })
tmp: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[41] at flatMap at <console>:36

```

```
scala> tmp.count()
res31: Long = 3213

scala> blanklines.value
res32: Int = 171
```

累加器的用法如下所示。

通过在驱动器中调用 `SparkContext.accumulator(initialValue)` 方法，创建出存有初始值的累加器。返回值为 `org.apache.spark.Accumulator[T]` 对象，其中 `T` 是初始值 `initialValue` 的类型。Spark 闭包里的执行器代码可以使用累加器的 `+=` 方法(在 Java 中是 `add`)增加累加器的值。驱动器程序可以调用累加器的 `value` 属性(在 Java 中使用 `value()` 或 `setvalue()`)来访问累加器的值。

注意：工作节点上的任务不能访问累加器的值。从这些任务的角度来看，累加器是一个只写变量。

对于要在 `action` 中使用的累加器，Spark 只会把每个任务对各累加器的修改应用一次。因此，如果想要一个无论在失败还是重复计算时都绝对可靠的累加器，我们必须把它放在 `foreach()` 这样的 `action` 中。`Transformation` 中累加器可能会发生不止一次更新。

(累加器最好用在 `foreach` 这样的 `rdd` 中，在 `transformation` 中很容易重复计算)

5.1.2 自定义累加器

自定义累加器类型的功能在 1.X 版本中就已经提供了，但是使用起来比较麻烦，在 2.0 版本后，累加器的易用性有了较大的改进，而且官方还提供了一个新的抽象类：`AccumulatorV2` 来提供更加友好的自定义类型累加器的实现方式。实现自定义类型累加器需要继承 `AccumulatorV2` 并至少覆写下例中出现的方法，下面这个累加器可以用于在程序运行过程中收集一些文本类信息，最终以 `Set[String]` 的形式返回。

```
import org.apache.spark.util.AccumulatorV2
import org.apache.spark.{SparkConf, SparkContext}
import scala.collection.JavaConversions._

class LogAccumulator extends org.apache.spark.util.AccumulatorV2[String, java.util.Set[String]] {
    private val _logArray: java.util.Set[String] = new java.util.HashSet[String]()

    //累加器初始化
    override def isZero: Boolean = {
        _logArray.isEmpty
    }

    //累加器重置
    override def reset(): Unit = {
        _logArray.clear()
    }

    //自定义累加规则
    override def add(v: String): Unit = {
        _logArray.add(v)
    }

    //回到 Driver 端聚合
    override def merge(other: org.apache.spark.util.AccumulatorV2[String, java.util.Set[String]]): Unit = {
        other match {
            case o: LogAccumulator => _logArray.addAll(o.value)
        }
    }

    //调用 value 方法即可得到最终的 sum 值
    override def value: java.util.Set[String] = {
        java.util.Collections.unmodifiableSet(_logArray)
    }
}
```

```

//复制新的累加器
override def copy():org.apache.spark.util.AccumulatorV2[String, java.util.Set[String]] = {
    val newAcc = new LogAccumulator()
    _logArray.synchronized{
        newAcc._logArray.addAll(_logArray)
    }
    newAcc
}

// 过滤掉带字母的
object LogAccumulator {
    def main(args: Array[String]) {
        val conf=new SparkConf().setAppName("LogAccumulator")
        val sc=new SparkContext(conf)

        val accum = new LogAccumulator
        sc.register(accum, "logAccum")
        val sum = sc.parallelize(Array("1", "2a", "3", "4b", "5", "6", "7cd", "8", "9"), 2).filter(line => {
            val pattern = """^-[0-9]+"""
            val flag = line.matches(pattern)
            if (!flag) {
                accum.add(line)
            }
            flag
        }).map(_.toInt).reduce(_ + _)

        println("sum: " + sum)
        for (v <- accum.value) print(v + "")
        println()
        sc.stop()
    }
}

```

自定义累加器-简单版

```

import org.apache.spark.util.AccumulatorV2

class MyAccumulator extends AccumulatorV2[(String,Long),Long]{
    private var sum = 0L

    //累加器初始化
    override def isZero: Boolean = {
        sum == 0L
    }

    //复制新的累加器
    override def copy(): AccumulatorV2[(String, Long), Long] = new MyAccumulator

    //重置累加器
    override def reset(): Unit = {
        sum = 0L
    }

    //最重要的:自定义累加规则
    override def add(v: (String, Long)): Unit = {
        sum += v._2
    }
}

```

```

//回到 Driver 端聚合
override def merge(other: AccumulatorV2[(String, Long), Long]): Unit = {
    sum += other.value
}

//调用 value 方法即可得到最终的 sum 值
override def value: Long = {
    sum
}
}

```

调用：

```

//创建累加器
val myaccu = new MyAccumulator

//注册累加器,让 spark 知道我的存在
sc.register(myaccu)
listRDD.foreach{
    data=>{
        myaccu.add(data)
    }
}
println(myaccu.value)

```

5.2 广播变量（调优策略）

广播变量用来高效分发较大的对象。向所有工作节点发送一个较大的只读值，以供一个或多个 Spark 操作使用。比如，如果你的应用需要向所有节点发送一个较大的只读查询表，甚至是机器学习算法中的一个很大的特征向量，广播变量用起来都很顺手。在需要跨多个 Stage 的多个 Task 中使用相同数据的情况下，广播特别的有用

```

scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(35)

scala> broadcastVar.value
res33: Array[Int] = Array(1, 2, 3)

```

//广播变量使用完了以后，可以使用 unpersist 删除数据

```

broadcastVar.unpersist
//删除数据以后，可以使用 destroy 销毁变量，释放内存空间
broadcastVar.destroy

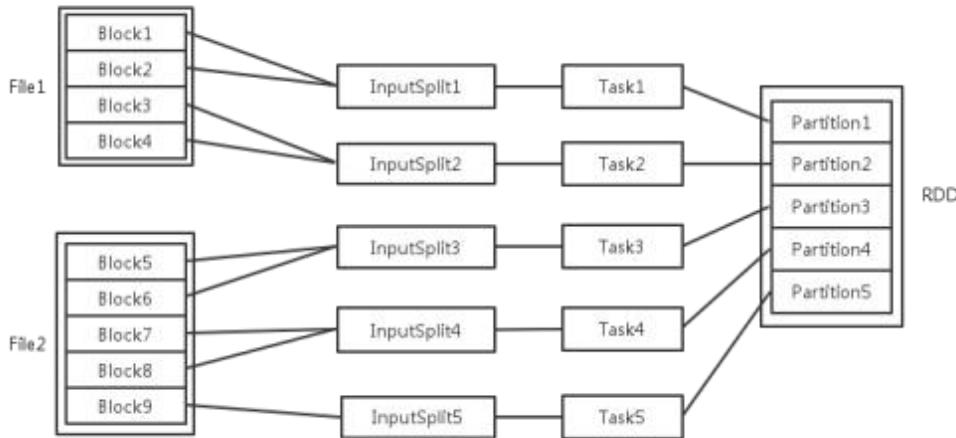
```

使用广播变量的过程如下：

- (1) 通过对一个类型 T 的对象调用 `SparkContext.broadcast` 创建出一个 `Broadcast[T]` 对象，任何可序列化的类型都可以这么实现。
- (2) 通过 `value` 属性访问该对象的值(在 Java 中为 `value()`方法)。
- (3) 变量只会被发到各个节点一次，应作为只读值处理(修改这个值不会影响到别的节点)。

第 6 章 扩展

RDD 相关概念关系



输入可能以多个文件的形式存储在 HDFS 上，每个 File 都包含了很多块，称为 Block。当 Spark 读取这些文件作为输入时，会根据具体数据格式对应的 `InputFormat` 进行解析，一般是将若干个 Block 合并成一个输入分片，称为 `InputSplit`，注意 `InputSplit` 不能跨越文件。随后将为这些输入分片生成具体的 Task。`InputSplit` 与 `Task` 是一一对应的关系。随后这些具体的 `Task` 每个都会被分配到集群上的某个节点的某个 `Executor` 去执行。

- 1) 每个节点可以起一个或多个 `Executor`。
- 2) 每个 `Executor` 由若干 `core` 组成，每个 `Executor` 的每个 `core` 一次只能执行一个 `Task`。
- 3) 每个 `Task` 执行的结果就是生成了目标 `RDD` 的一个 `partition`。

注意：这里的 `core` 是虚拟的 `core` 而不是机器的物理 CPU 核，可以理解为就是 `Executor` 的一个工作线程。而 `Task` 被执行的并发度 = `Executor` 数目 * 每个 `Executor` 核数。

至于 `partition` 的数目：

- 1) 对于数据读入阶段，例如 `sc.textFile`，输入文件被划分为多少 `InputSplit` 就会需要多少初始 `Task`。
- 2) 在 `Map` 阶段 `partition` 数目保持不变。
- 3) 在 `Reduce` 阶段，`RDD` 的聚合会触发 `shuffle` 操作，聚合后的 `RDD` 的 `partition` 数目跟具体操作有关，例如 `repartition` 操作会聚合成指定分区数，还有一些算子是可配置的。

`RDD` 在计算的时候，每个分区都会起一个 `task`，所以 `rdd` 的分区数目决定了总的的 `task` 数目。申请的计算节点（`Executor`）数目和每个计算节点核数，决定了你同一时刻可以并行执行的 `task`。

比如的 `RDD` 有 100 个分区，那么计算的时候就会生成 100 个 `task`，你的资源配置为 10 个计算节点，每个计算节点有两 2 个核，同一时刻可以并行的 `task` 数目为 20，计算这个 `RDD` 就需要 5 个轮次。如果计算资源不变，你有 101 个 `task` 的话，就需要 6 个轮次，在最后一轮中，只有一个 `task` 在执行，其余核都在空转。如果资源不变，你的 `RDD` 只有 2 个分区，那么同一时刻只有 2 个 `task` 运行，其余 18 个核空转，造成资源浪费。这就是在 `spark` 调优中，增大 `RDD` 分区数目，增大任务并行度的做法。

SparkSQL

第1章 Spark SQL 概述

1.1 什么是 Spark SQL

Spark SQL 是 Spark 用来处理结构化数据的一个模块，它提供了 2 个编程抽象：DataFrame 和 DataSet，并且作为分布式 SQL 查询引擎的作用。

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！

1.2 Spark SQL 的特点

1) 易整合

Integrated

Seamlessly mix SQL queries with Spark programs.

Spark SQL lets you query structured data inside Spark programs, using either SQL or a familiar [DataFrame API](#). Usable in Java, Scala, Python and R.

```
context = HiveContext(sc)
results = context.sql(
    "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

Apply functions to results of SQL queries.

2) 统一的数据访问方式

Uniform Data Access

Connect to any data source the same way.

DataFrames and SQL provide a common way to access a variety of data sources, including Hive, Avro, Parquet, ORC, JSON, and JDBC. You can even join data across these sources.

```
context.jsonFile("s3n://...")
    .registerTempTable("json")
results = context.sql(
    """SELECT *
       FROM people
      JOIN json ...""")
```

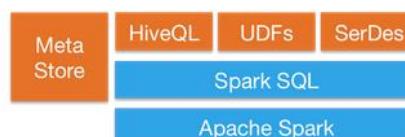
Query and join different data sources.

3) 兼容 Hive

Hive Compatibility

Run unmodified Hive queries on existing data.

Spark SQL reuses the Hive frontend and metastore, giving you full compatibility with existing Hive data, queries, and UDFs. Simply install it alongside Hive.



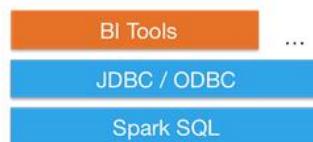
Spark SQL can use existing Hive metastores, SerDes, and UDFs.

4) 标准的数据连接

Standard Connectivity

Connect through JDBC or ODBC.

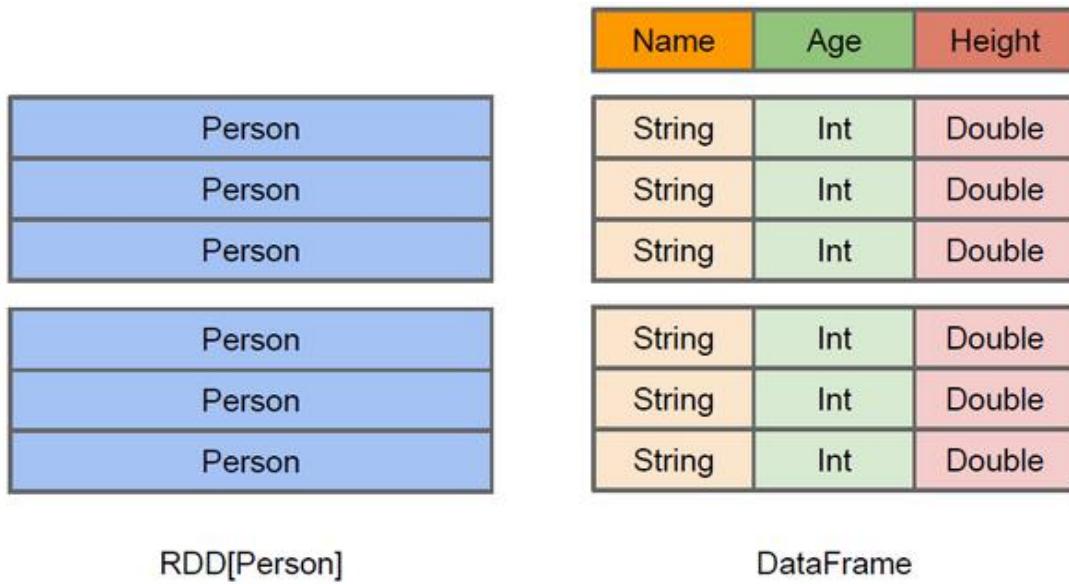
A server mode provides industry standard JDBC and ODBC connectivity for business intelligence tools.



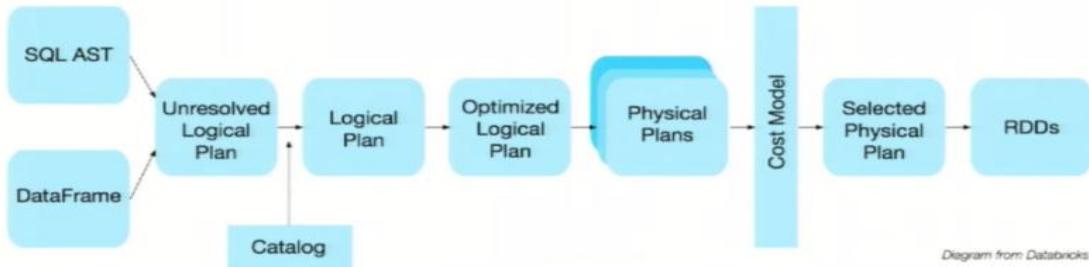
Use your existing BI tools to query big data.

1.3 什么是 DataFrame

与 RDD 类似，DataFrame 也是一个分布式数据集。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，即 schema。同时，与 Hive 类似，DataFrame 也支持嵌套数据类型（struct、array 和 map）。从 API 易用性的角度上看，DataFrame API 提供的是一套高层的关系操作，比函数式的 RDD API 要更加友好，门槛更低。

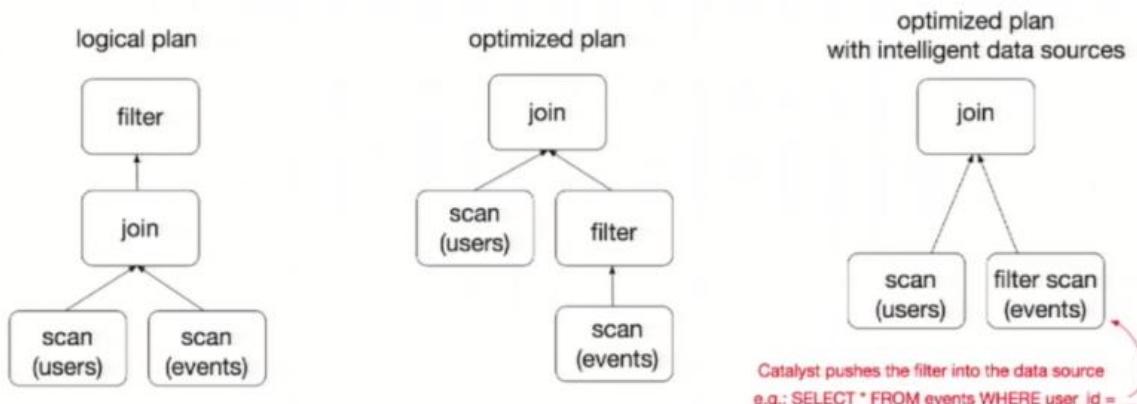


上图直观地体现了 DataFrame 和 RDD 的区别。左侧的 RDD[Person] 虽然以 Person 为类型参数，但 Spark 框架本身不了解 Person 类的内部结构。而右侧的 DataFrame 却提供了详细的结构信息，使得 Spark SQL 可以清楚地知道该数据集中包含哪些列，每列的名称和类型各是什么。DataFrame 是为数据提供了 Schema 的视图。可以把它当做数据库中的一张表来对待，DataFrame 也是懒执行的。性能上比 RDD 要高，主要原因：优化的执行计划查询计划通过 Spark catalyst optimiser 进行优化：



比如下面一个例子：

```
users.join(events, users("id") === events("uid")).filter(events("date") > "2015-01-01")
```



为了说明查询优化，我们来看上图展示的人口数据分析的示例。图中构造了两个 DataFrame，将它们 join 之后又做了一次 filter 操作。如果原封不动地执行这个执行计划，最终的执行效率是不高的。因为 join 是一个代价较大

的操作，也可能会产生一个较大的数据集。如果我们能将 filter 下推到 join 下方，先对 DataFrame 进行过滤，再 join 过滤后的较小的结果集，便可以有效缩短执行时间。而 Spark SQL 的查询优化器正是这样做的。简而言之，逻辑查询计划优化就是一个利用基于关系代数的等价变换，将高成本的操作替换为低成本操作的过程。

1.4 什么是 DataSet

- 1) 是 Dataframe API 的一个扩展，是 Spark 最新的数据抽象。
- 2) 用户友好的 API 风格，既具有类型安全检查也具有 Dataframe 的查询优化特性。
- 3) Dataset 支持编解码器，当需要访问非堆上的数据时可以避免反序列化整个对象，提高了效率。
- 4) 样例类被用来在 Dataset 中定义数据的结构信息，样例类中每个属性的名称直接映射到 DataSet 中的字段名称。
- 5) Dataframe 是 Dataset 的特列，即 DataFrame=Dataset[Row]，可以通过 as 方法将 Dataframe 转换为 Dataset。Row 是一个类型，跟 Car、Person 这些的类型一样，所有的表结构信息都用 Row 来表示。
- 6) DataSet 是强类型的。比如可以有 Dataset[Car], Dataset[Person]。
- 7) DataFrame 只是知道字段，但是不知道字段的类型，所以在执行这些操作的时候是没办法在编译的时候检查是否类型失败的，比如你可以对一个 String 进行减法操作，在执行的时候才报错，而 DataSet 不仅仅知道字段，而且知道字段类型，所以有更严格的错误检查。就跟 JSON 对象和类对象之间的类比。

1.5 RDD、DataFrame 和 DataSet 的区别

(1) 相同点：

都是分布式数据集

DataFrame 底层是 RDD，但是 DataSet 不是，不过他们最后都是转换成 RDD 运行

DataSet 和 DataFrame 的相同点都是有数据特征、数据类型的分布式数据集(schema)

(2) 不同点：

(a) schema 信息：

RDD 中的数据是没有数据类型的

DataFrame 中的数据是弱数据类型，不会做数据类型检查，虽然有 schema 规定了数据类型，但是编译时是不会报错的，运行时才会报错

DataSet 中的数据类型是强数据类型，会做数据类型检查。

(b) 序列化机制：

RDD 和 DataFrame 默认的序列化机制是 java 的序列化，也可以用 Kryo 的机制

DataSet 使用自定义的数据编码器进行序列化和反序列化

第 2 章 Spark SQL 编程

2.1 SparkSession 新的起始点

在老的版本中，SparkSQL 提供两种 SQL 查询起始点：

SQLContext，用于 Spark 自己提供的 SQL 查询；

HiveContext，用于连接 Hive 的查询。

新的版本中，使用 SparkSession 作为 SQL 查询起始点，实质上是 SQLContext 和 HiveContext 的组合，所以在 SQLContext 和 HiveContext 上可用的 API 在 SparkSession 上同样是可以使用的。SparkSession 内部封装了 sparkContext，所以计算实际上是由 sparkContext 完成的。

2.2 DataFrame

2.2.1 创建

在 Spark SQL 中 SparkSession 是创建 DataFrame 和执行 SQL 的入口，创建 DataFrame 有三种方式：通过 Spark 的数据源进行创建；从一个存在的 RDD 进行转换；还可以从 Hive Table 进行查询返回。

1) 从 Spark 数据源进行创建

```

//Spark 数据源的文件格式
csv    format    jdbc    json    load    option    options    orc    parquet    schema    table    text    textFile

//读取 json 文件创建 DataFrame
val df = spark.read.json("src/main/resources/people.json")

//展示结果
scala> df.show
+---+----+
| age| name|
+---+----+
| null| Michael|
| 30| Andy|
| 19| Justin|
+---+----+

```

2) 从 RDD 进行转换

2.5 节我们专门讨论

3) 从 Hive Table 进行查询返回

3.5 节我们专门讨论

2.2.2 SQL 风格语法(主要)

```

//创建 DataFrame
val df = spark.read.json("/opt/module/spark/examples/src/main/resources/people.json")
//创建临时表
df.createOrReplaceTempView("people")
//通过 SQL 语句实现查询全表
val sqlDF = spark.sql("SELECT * FROM people")
//结果展示
sqlDF.show
+---+----+
| age| name|
+---+----+
| null| Michael|
| 30| Andy|
| 19| Justin|
+---+----+

```

注意：临时表是 Session 范围内的，Session 退出后表就失效了。如果想应用范围内有效，可以使用全局表。注意使用全局表时需要全路径访问，如：global_temp.people

```

// 创建一个全局表
df.createGlobalTempView("people")
//通过 SQL 语句实现查询全表
spark.sql("SELECT * FROM global_temp.people").show()
+---+----+
| age| name|
+---+----+
| null| Michael|
| 30| Andy|
| 19| Justin|
+---+----+
spark.newSession().sql("SELECT * FROM global_temp.people").show()
+---+----+
| age| name|
+---+----+
| null| Michael|
| 30| Andy|
| 19| Justin|
+---+----+

```

+-----+ 2.2.3 DSL 风格语法(次要)

```
//创建 DataFrame  
val df = spark.read.text(path)  
  
//查看 DataFrame 的 Schema 信息  
df.printSchema()  
  
root  
|-- age: long (nullable = true)  
|-- name: string (nullable = true)  
  
//只查看"name"列数据  
scala> df.select("name").show()  
+---+  
| name |  
+---+  
| Michael |  
| Andy |  
| Justin |  
+---+  
  
//查看"name"列数据以及"age+1"数据  
df.select($"name", $"age" + 1).show()  
+---+---+  
| name | (age + 1) |  
+---+---+  
| Michael | null |  
| Andy | 31 |  
| Justin | 20 |  
+---+---+  
  
//查看"age"大于"21"的数据  
df.filter($"age" > 21).show()  
+---+---+  
| age | name |  
+---+---+  
| 30 | Andy |  
+---+---+  
  
//按照"age"分组，查看数据条数  
df.groupBy("age").count().show()  
  
+---+---+  
| age | count |  
+---+---+  
| 19 | 1 |  
| null | 1 |  
| 30 | 1 |  
+---+---+
```

2.2.4 RDD 转换为 DataFrame

如果需要 RDD 与 DF 或者 DS 之间操作，那么都需要引入 `import session.implicits._` 【`session` 不是包名，而是 `sparkSession` 对象的名称】

前置条件：导入隐式转换并创建一个 RDD

```
val session = SparkSession.builder()  
  .config("spark.sql.shuffle.partitions", "1")
```

```

.config("spark.master", "local[*]")
.appName("SparkSqlMysqlSession")
.getOrCreate()
val sc = session.sparkContext

import session.implicits._

val peopleRDD = sc.textFile("examples/src/main/resources/people.txt")

```

方式 1、通过手动确定转换，自动推断类型？

```

peopleRDD.map{x=>
    val para = x.split(",");
    (para(0), para(1).trim.toInt)
}.toDF("name", "age")

```

方式 2、通过反射确定（需要用到样例类）

```

case class People(name:String, age:Int)

peopleRDD.map{x =>
    val para = x.split(",");
    People(para(0), para(1).trim.toInt)
}.toDF

```

方式 3、通过编程的方式

```

import org.apache.spark.sql.types._
//创建 Schema
val arr = new ArrayBuffer[StructField]()
arr += DataTypes.createStructField("name", DataTypes.StringType, true) // (字段名, 类型, 是否允许为空)
arr += DataTypes.createStructField("age", DataTypes.IntegerType, false)
val structType = DataTypes.createStructType(arr.toArray)
//根据给定的类型创建二元组 RDD
import org.apache.spark.sql.Row
val data = peopleRDD.map{ x =>
    val para = x.split(",");
    Row(para(0), para(1).trim.toInt)
}
//根据数据及给定的 schema 创建 DataFrame
scala> val dataFrame = session.createDataFrame(data, structType)

```

2.2.5 DataFrame 转换为 RDD

直接调用 rdd 即可

```

//创建一个 DataFrame
val df = spark.read.json("/opt/module/spark/examples/src/main/resources/people.json")
//DataFrame 转 RDD
val dfToRDD = df.rdd
//打印 RDD
dfToRDD.collect

```

2.3 DataSet

Dataset 是具有强类型的数据集合，需要提供对应的类型信息。

2.3.1 创建

```

//创建一个样例类
case class Person(name: String, age: Long)
//创建 DataSet
val caseClassDS = Seq(Person("Andy", 32)).toDS()

```

2.3.2 RDD 转换为 DataSet

SparkSQL 能够自动将包含有 case 类的 RDD 转换成 DataFrame, case 类定义了 table 的结构, case 类属性通过反射变成了表的列名。

```
//创建 RDD  
val peopleRDD = sc.textFile("examples/src/main/resources/people.txt")  
//创建一个样例类  
case class Person(name: String, age: Long)  
//将 RDD 转化为 DataSet (和 rdd 转 df 一样)  
peopleRDD.map(line => {  
    val para = line.split(",")  
    Person(para(0),para(1).trim.toInt)  
}).toDS()
```

2.3.3 DataSet 转换为 RDD

调用 rdd 方法即可。

```
//创建 DataSet  
val DS = Seq(Person("Andy", 32)).toDS()  
//将 DataSet 转换为 RDD  
DS.rdd
```

2.4 DataFrame 与 DataSet 的互操作

1. DataFrame-->DataSet

```
//创建 DataFrame  
val df = spark.read.json("examples/src/main/resources/people.json")  
//创建一个样例类  
case class Person(name: String, age: Long)  
//将 DataFrame 转化为 DataSet  
df.as[Person]
```

2. DataSet-->DataFrame

```
//创建一个样例类  
case class Person(name: String, age: Long)  
//创建 DataSet  
val ds = Seq(Person("Andy", 32)).toDS()  
//将 DataSet 转化为 DataFrame  
val df = ds.toDF  
//展示  
df.show  
+---+---+  
|name|age|  
+---+---+  
|Andy| 32|  
+---+---+
```

2.4.1 DataSet 转 DataFrame

这个很简单, 因为只是把 case class 封装成 Row

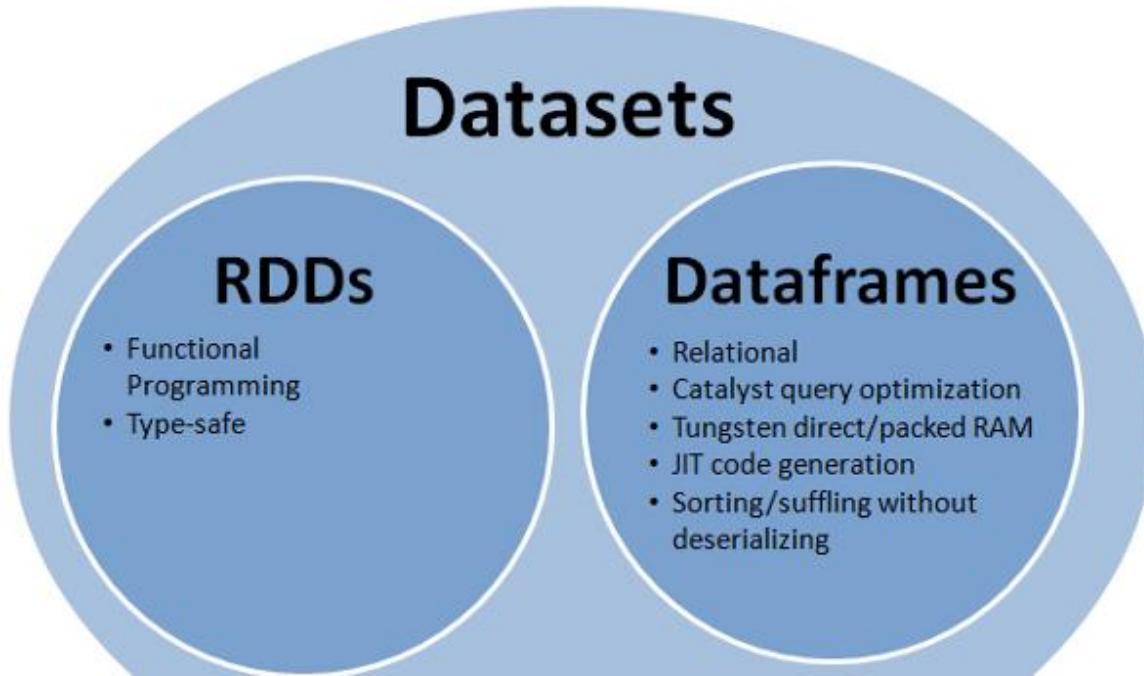
```
//导入隐式转换  
import spark.implicits._  
//转换  
val testDF = testDS.toDF
```

2.4.2 DataFrame 转 DataSet

```
//导入隐式转换  
import spark.implicits._  
//创建样例类  
case class Coltest(col1:String,col2:Int)extends Serializable //定义字段名和类型  
//转换  
val testDS = testDF.as[Coltest]
```

这种方法就是在给出每一列的类型后，使用 `as` 方法，转成 `Dataset`，这在数据类型是 `DataFrame` 又需要针对各个字段处理时极为方便。在使用一些特殊的操作时，一定要加上 `import spark.implicits._` 不然 `toDF`、`toDS` 无法使用。

2.5 RDD、DataFrame、DataSet



在 SparkSQL 中 Spark 为我们提供了两个新的抽象，分别是 `DataFrame` 和 `DataSet`。他们和 `RDD` 有什么区别呢？

首先从版本的产生上来看：`RDD` (`Spark1.0`) → `Dataframe`(`Spark1.3`) → `Dataset`(`Spark1.6`)。如果同样的数据都给到这三个数据结构，他们分别计算之后，都会给出相同的结果。不同的是他们的执行效率和执行方式。在后期的 Spark 版本中，`DataSet` 会逐步取代 `RDD` 和 `DataFrame` 成为唯一的 API 接口。

2.5.1 三者的共性

- 1、`RDD`、`DataFrame`、`Dataset` 全都是 spark 平台下的弹性分布式数据集，为处理超大型数据提供便利
- 2、三者都有惰性机制，在进行创建、转换，如 `map` 方法时，不会立即执行，只有在遇到 Action 如 `foreach` 时，三者才会开始遍历运算。
- 3、三者都会根据 spark 的内存情况自动缓存运算，这样即使数据量很大，也不用担心会内存溢出。
- 4、三者都有分区概念
- 5、三者有许多共同的函数，如 `filter`，排序等
- 6、在对 `DataFrame` 和 `Dataset` 进行操作许多操作都需要这个包进行支持
`import session.implicits._`
- 7、`DataFrame` 和 `Dataset` 均可使用模式匹配获取各个字段的值和类型

`DataFrame`:

```
testDF.map{  
    case Row(col1:String,col2:Int)=>  
        println(col1);println(col2)  
        col1  
    case _=>  
        ""  
}
```

`Dataset`:

```
case class Coltest(col1:String,col2:Int)extends Serializable //定义字段名和类型  
testDS.map{  
    case Coltest(col1:String,col2:Int)=>  
        println(col1);println(col2)  
        col1
```

```
    case _ =>
      """
    }
```

2.5.2 三者的区别

1. RDD:

- 1) RDD 一般和 spark mli (机器学习) 同时使用
- 2) RDD 不支持 spark sql 操作

2. DataFrame:

- 1) 与 RDD 和 Dataset 不同, DataFrame 每一行的类型固定为 Row, 每一列的值没法直接访问, 只有通过解析才能获取各个字段的值, 如:

```
testDF.foreach{
  line =>
    val col1=line.getAs[String]("col1")
    val col2=line.getAs[String]("col2")
}
```

- 2) DataFrame 与 Dataset 一般不与 spark mlib 同时使用

- 3) DataFrame 与 Dataset 均支持 sparksql 的操作, 比如 select, groupby 之类, 还能注册临时表/视窗, 进行 sql 语句操作, 如:

```
dataDF.createOrReplaceTempView("tmp")
spark.sql("select ROW,DATE from tmp where DATE is not null order by DATE").show(100,false)
```

- 4) DataFrame 与 Dataset 支持一些特别方便的保存方式, 比如保存成 csv, 可以带上表头, 这样每一列的字段名一目了然

```
//保存
val saveoptions = Map("header" -> "true", "delimiter" -> "\t", "path" -> "hdfs://hadoop102:9000/test")
dataDF.write.format("com.atguigu.spark.csv").mode(SaveMode.Overwrite).options(saveoptions).save()
//读取
val options = Map("header" -> "true", "delimiter" -> "\t", "path" -> "hdfs://hadoop102:9000/test")
val dataDF= spark.read.options(options).format("com.atguigu.spark.csv").load()
```

利用这样的保存方式, 可以方便的获得字段名和列的对应, 而且分隔符 (delimiter) 可以自由指定。

3. Dataset:

- 1) Dataset 和 DataFrame 拥有完全相同的成员函数, 区别只是每一行的数据类型不同。

2) DataFrame 也可以叫 Dataset[Row], 每一行的类型是 Row, 不解析, 每一行究竟有哪些字段, 各个字段又是什么类型都无从得知, 只能用上面提到的 getAS 方法或者共性中的第七条提到的模式匹配拿出特定字段。而 Dataset 中, 每一行是什么类型是不确定的, 在自定义了 case class 之后可以很自由的获得每一行的信息。

```
case class Coltest(col1:String,col2:Int) extends Serializable //定义字段名和类型
/**
rdd
  ("a", 1)
  ("b", 1)
  ("a", 1)
*/
val test: Dataset[Coltest]=rdd.map{line=>
  Coltest(line._1,line._2)
}.toDS
test.map{
  line=>
    println(line.col1)
    println(line.col2)
}
```

可以看出, Dataset 在需要访问列中的某个字段时是非常方便的, 然而, 如果要写一些适配性很强的函数时, 如果使用 Dataset, 行的类型又不确定, 可能是各种 case class, 无法实现适配, 这时候用 DataFrame 即 Dataset[Row] 就能比较好的解决问题

2.6 IDEA 创建 SparkSQL 程序

IDEA 中程序的打包和运行方式都和 SparkCore 类似，Maven 依赖中需要添加新的依赖项：

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.1.1</version>
</dependency>
```

程序如下：

```
package com.atguigu.sparksq

import org.apache.spark.sql.SparkSession
import org.apache.spark.{SparkConf, SparkContext}
import org.slf4j.LoggerFactory

object HelloWorld {
    def main(args: Array[String]) {
        //创建 sparksession 对象
        val spark = SparkSession
            .builder()
            .appName("Spark SQL basic example")
            .config("spark.some.config.option", "some-value")
            .getOrCreate()

        //导入隐式转换
        import spark.implicits._

        val df = spark.read.json("data/people.json")

        // 输出展示
        df.show()
        df.filter($"age" > 21).show()
        df.createOrReplaceTempView("persons")
        spark.sql("SELECT * FROM persons where age > 21").show()

        spark.stop()
    }
}
```

2.7 用户自定义函数

在 Shell 窗口中可以通过 spark.udf 功能用户可以自定义函数。

2.7.1 用户自定义 UDF 函数

```
val df = spark.read.json("examples/src/main/resources/people.json")
df.show()
+---+-----+
| age|    name|
+---+-----+
|null|Michael|
|  30|    Andy|
|  19|    Justin|
+---+-----+

//注册自定义函数
spark.udf.register("addName", (x:String)=> "Name:"+x)

df.createOrReplaceTempView("people")
```

```

spark.sql("Select addName(name), age from people").show()
+-----+---+
|UDF:addName(name)| age|
+-----+---+
|  Name:Michael |null|
|  Name:Andy    |30 |
|  Name:Justin   |19 |
+-----+---+

```

2.7.2 用户自定义聚合函数

强类型的 Dataset 和弱类型的 DataFrame 都提供了相关的聚合函数，如 count()，countDistinct()，avg()，max()，min()。除此之外，用户还可以设定自己的自定义聚合函数。

弱类型用户自定义聚合函数：通过继承 UserDefinedAggregateFunction 来实现用户自定义聚合函数。下面展示一个求平均工资的自定义聚合函数。::追加进 List，Nil 是空 List。

```

import org.apache.spark.sql.expressions.MutableAggregationBuffer
import org.apache.spark.sql.expressions.UserDefinedAggregateFunction
import org.apache.spark.sql.types._
import org.apache.spark.sql.Row
import org.apache.spark.sql.SparkSession

object MyAverage extends UserDefinedAggregateFunction {
    // 聚合函数输入参数的数据类型
    def inputSchema = StructType(StructField("inputColumn", LongType) :: Nil)
    // 聚合缓冲区中的数据类型
    def bufferSchema = {
        StructType(StructField("sum", LongType) :: StructField("count", LongType) :: Nil)
    }
    // 返回值的数据类型
    def dataType = DoubleType
    // 对于相同的输入是否一直返回相同的输出。
    def deterministic = true
    // 初始化
    def initialize(buffer: MutableAggregationBuffer): Unit = {
        //存工资的总额
        buffer(0) = 0L
        // 存工资的个数
        buffer(1) = 0L
    }
    // 相同 Execute 间的数据合并。
    def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
        if (!input.isNullAt(0)) {
            buffer(0) = buffer.getLong(0) + input.getLong(0)
            buffer(1) = buffer.getLong(1) + 1
        }
    }
    // 不同 Execute 间的数据合并
    def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
        buffer1(0) = buffer1.getLong(0) + buffer2.getLong(0)
        buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1)
    }
    // 计算最终结果
    def evaluate(buffer: Row) = buffer.getLong(0).toDouble / buffer.getLong(1)
}

// 注册函数
spark.udf.register("myAverage", MyAverage)

```

```

val df = spark.read.json("examples/src/main/resources/employees.json")
df.createOrReplaceTempView("employees")
df.show()
// +-----+
// |   name|salary|
// +-----+
// | Michael|  3000|
// |   Andy|  4500|
// | Justin|  3500|
// |  Berta|  4000|
// +-----+


val result = spark.sql("SELECT myAverage(salary) as average_salary FROM employees")
result.show()
// +-----+
// |average_salary|
// +-----+
// |      3750.0|
// +-----+

```

强类型用户自定义聚合函数：通过继承 `Aggregator` 来实现强类型自定义聚合函数，同样是求平均工资

```

import org.apache.spark.sql.expressions.Aggregator
import org.apache.spark.sql.Encoder
import org.apache.spark.sql.Encoders
import org.apache.spark.sql.SparkSession

// 既然是强类型，可能有 case 类
case class Employee(name: String, salary: Long)
case class Average(var sum: Long, var count: Long)

object MyAverage extends Aggregator[Employee, Average, Double] {
    // 定义一个数据结构，保存工资总数和工资总个数，初始都为 0
    def zero: Average = Average(0L, 0L)
    // Combine two values to produce a new value. For performance, the function may modify `buffer`
    // and return it instead of constructing a new object
    def reduce(buffer: Average, employee: Employee): Average = {
        buffer.sum += employee.salary
        buffer.count += 1
        buffer
    }
    // 聚合不同 execute 的结果
    def merge(b1: Average, b2: Average): Average = {
        b1.sum += b2.sum
        b1.count += b2.count
        b1
    }
    // 计算输出
    def finish(reduction: Average): Double = reduction.sum.toDouble / reduction.count
    // 设定之间值类型的编码器，要转换成 case 类
    // Encoders.product 是进行 scala 元组和 case 类转换的编码器
    def bufferEncoder: Encoder[Average] = Encoders.product
    // 设定最终输出值的编码器
    def outputEncoder: Encoder[Double] = Encoders.scalaDouble
}

import spark.implicits._

val ds = spark.read.json("examples/src/main/resources/employees.json").as[Employee]

```

```

ds.show()
// +-----+
// |   name|salary|
// +-----+
// | Michael|  3000|
// |   Andy|  4500|
// | Justin|  3500|
// |  Berta|  4000|
// +-----+

// Convert the function to a `TypedColumn` and give it a name
val averageSalary = MyAverage.toColumn.name("average_salary")
val result = ds.select(averageSalary)
result.show()
// +-----+
// |average_salary|
// +-----+
// |          3750.0|
// +-----+

```

第 3 章 Spark SQL 数据源

3.1 通用加载/保存方法

3.1.1 手动指定选项

Spark SQL 的 DataFrame 接口支持多种数据源的操作。一个 DataFrame 可以进行 RDDs 方式的操作，也可以被注册为临时表。把 DataFrame 注册为临时表之后，就可以对该 DataFrame 执行 SQL 查询。

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

```

val df = session.read.load("examples/src/main/resources/users.parquet")
df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")

```

当数据源格式不是 parquet 格式文件时，需要手动指定数据源的格式。数据源格式需要指定全名（例如：org.apache.spark.sql.parquet），如果数据源格式为内置格式，则只需要指定简称定 json, parquet, jdbc, orc, libsvm, csv, text 来指定数据的格式。

可以通过 SparkSession 提供的 read.load 方法用于通用加载数据，使用 write 和 save 保存数据。

```

val peopleDF = session.read.format("json").load("examples/src/main/resources/people.json")
peopleDF.write.format("parquet").save("hdfs://hadoop102:9000/namesAndAges.parquet")

```

除此之外，可以直接运行 SQL 在文件上：

```

val sqlDF = session.sql("SELECT * FROM parquet.`hdfs://hadoop102:9000/namesAndAges.parquet`")
sqlDF.show()

```

```

val peopleDF = session.read.format("json").load("examples/src/main/resources/people.json")
peopleDF.write.format("parquet").save("hdfs://hadoop102:9000/namesAndAges.parquet")

```

```

peopleDF.show()
+---+-----+
| age|    name|
+---+-----+
| null|Michael|
|  30|   Andy|
|  19| Justin|
+---+-----+

```

```

val sqlDF = session.sql("SELECT * FROM parquet.`hdfs://hadoop102:9000/namesAndAges.parquet`")
scala> sqlDF.show()

```

age	name
null	Michael
30	Andy
19	Justin

3.1.2 文件保存选项

可以采用 `SaveMode` 执行存储操作, `SaveMode` 定义了对数据的处理模式。需要注意的是, 这些保存模式不使用任何锁定, 也不是原子操作。此外, 当使用 `Overwrite` 方式执行时, 在输出新数据之前原数据就已经被删除。`SaveMode` 详细介绍如下表:

```
sql.write.format("json").mode("overwrite").save("MyFile/output/avg")
sql.write.format("text").mode("append").save("MyFile/output/avg")
```

Scala/Java	Any Language	Meaning
<code>SaveMode.ErrorIfExists(default)</code>	"error"(default)	如果文件存在, 则报错
<code>SaveMode.Append</code>	"append"	追加
<code>SaveMode.Overwrite</code>	"overwrite"	覆盖
<code>SaveMode.Ignore</code>	"ignore"	数据存在, 则忽略

3.2 JSON 文件

Spark SQL 能够自动推测 JSON 数据集的结构, 并将它加载为一个 `Dataset[Row]`。可以通过 `SparkSession.read.json()` 去加载一个一个 JSON 文件。

注意: 这个 JSON 文件不是一个传统的 JSON 文件, 每一行都得是一个 JSON 串。

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}

// Primitive types (Int, String, etc) and Product types (case classes) encoders are
// supported by importing this when creating a Dataset.
// 在创建数据集时通过导入原始类型 (Int, String 等) 和产品类型 (样例类) 编码器来支持该编码器。
import spark.implicits._

// A JSON dataset is pointed to by path.
// The path can be either a single text file or a directory storing text files
// 路径指向 JSON 数据集, 路径可以是单个文本文件, 也可以是存储文本文件的目录
val path = "examples/src/main/resources/people.json"
val peopleDF = spark.read.json(path)

// The inferred schema can be visualized using the printSchema() method
// 可以使用 printSchema() 方法可视化推断的架构
peopleDF.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Creates a temporary view using the DataFrame
// 使用 DataFrame 创建一个临时视图
peopleDF.createOrReplaceTempView("people")

// SQL statements can be run by using the sql methods provided by spark
// 可以使用 spark 提供的 sql 方法运行 SQL 语句
val teenagerNamesDF = session.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19")
teenagerNamesDF.show()
```

```

// +---+
// |   name|
// +---+
// |Justin|
// +---+


// Alternatively, a DataFrame can be created for a JSON dataset represented by
// a Dataset[String] storing one JSON object per string
// 或者，可以为由 Dataset [String]表示的 JSON 数据集创建一个 DataFrame，每个字符串存储一个 JSON 对象
val otherPeopleDataset = session.createDataset(
    """{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""": Nil)
val otherPeople = session.read.json(otherPeopleDataset)
otherPeople.show()
// +-----+---+
// |      address|name|
// +-----+---+
// |[Columbus,Ohio]| Yin|

```

3.3 Parquet 文件

Parquet 是一种流行的列式存储格式，可以高效地存储具有嵌套字段的记录。Parquet 格式经常在 Hadoop 生态圈中被使用，它也支持 Spark SQL 的全部数据类型。Spark SQL 提供了直接读取和存储 Parquet 格式文件的方法。

```

importing spark.implicits._
import spark.implicits._

val peopleDF = spark.read.json("examples/src/main/resources/people.json")
peopleDF.write.parquet("hdfs://hadoop102:9000/people.parquet")

val parquetFileDF = spark.read.parquet("hdfs://hadoop102:9000/people.parquet")
parquetFileDF.createOrReplaceTempView("parquetFile")

val namesDF = spark.sql("SELECT name FROM parquetFile WHERE age BETWEEN 13 AND 19")
namesDF.map(attributes => "Name: " + attributes(0)).show()
// +-----+
// |      value|
// +-----+
// |Name: Justin|
// +-----+

```

3.4 JDBC

Spark SQL 可以通过 JDBC 从关系型数据库中读取数据的方式创建 DataFrame，通过对 DataFrame 一系列的计算后，还可以将数据再写回关系型数据库中。

注意：需要将相关的数据库驱动放到 spark 的类路径下。

(1) 启动 spark-shell

```
$ bin/spark-shell
```

(2) 从 Mysql 数据库加载数据方式一

```

val jdbcDF = session.read
.format("jdbc")
.option("url", "jdbc:mysql://hadoop102:3306/rdd")
.option("dbtable", "rddtable")
.option("user", "root")
.option("password", "000000")
.load()
// option: 可选参数

```

(3) 从 Mysql 数据库加载数据方式二

```
val connectionProperties = new Properties()
```

```
connectionProperties.put("user", "root")
connectionProperties.put("password", "000000")
val jdbcDF2 = session.read.jdbc("jdbc:mysql://hadoop102:3306/rdd", "rddtable", connectionProperties)
```

(4) 将数据写入 Mysql 方式一

```
jdbcDF.write  
.format("jdbc")  
.option("url", "jdbc:mysql://hadoop102:3306/rdd")  
.option("dbtable", "dftable")  
.option("user", "root")  
.option("password", "000000")  
.save()
```

(5) 将数据写入 Mysql 方式二

```
jdbcDF2.write.jdbc("jdbc:mysql://hadoop102:3306/rdd", "db", connectionProperties)
```

3.5 Hive 数据库

Apache Hive 是 Hadoop 上的 SQL 引擎，Spark SQL 编译时可以包含 Hive 支持，也可以不包含。包含 Hive 支持的 Spark SQL 可以支持 Hive 表访问、UDF(用户自定义函数)以及 Hive 查询语言(HiveQL/HQL)等。需要强调的一点是，如果要在 Spark SQL 中包含 Hive 的库，并不需要事先安装 Hive。一般来说，最好还是在编译 Spark SQL 时引入 Hive 支持，这样就可以使用这些特性了。如果你下载的是二进制版本的 Spark，它应该已经在编译时添加了 Hive 支持。

若要把 Spark SQL 连接到一个部署好的 Hive 上，你必须把 `hive-site.xml` 复制到 Spark 的配置文件目录中(`$SPARK_HOME/conf`)。即使没有部署好 Hive，Spark SQL 也可以运行。需要注意的是，如果你没有部署好 Hive，Spark SQL 会在当前的工作目录中创建出自己的 Hive 元数据仓库，叫作 `metastore_db`。此外，如果你尝试使用 HiveQL 中的 `CREATE TABLE`(并非 `CREATE EXTERNAL TABLE`)语句来创建表，这些表会被放在你默认的文件系统中的`/user/hive/warhouse` 目录中(如果你的 classpath 中有配好的 `hdfs-site.xml`，默认的文件系统就是 HDFS，否则就是本地文件系统)。

3.5.1 内嵌 Hive 应用

如果要使用内嵌的 Hive，什么都不用做，直接用就可以了。可以通过添加参数初次指定数据仓库地址：

```
--conf spark.sql.warehouse.dir=hdfs://hadoop102/spark-warehouse
```

注意：如果你使用的是内部的 Hive，在 Spark2.0 之后，spark.sql.warehouse.dir 用于指定数据仓库的地址，如果你需要是用 HDFS 作为路径，那么需要将 core-site.xml 和 hdfs-site.xml 加入到 Spark conf 目录，否则只会创建 master 节点上的 warehouse 目录，查询时会出现文件找不到的问题，这时需要使用 HDFS，则需要将 metastore 删除，重启集群。

3.5.2 外部 Hive 应用

如果想连接外部已经部署好的 Hive，需要通过以下几个步骤。

1) 将 Hive 中的 hive-site.xml 拷贝或者软连接到 Spark 安装目录下的 conf 目录下。

2) 打开 spark shell，注意带上访问 Hive 元数据库的 JDBC 客户端

```
$ bin/spark-shell --jars mysql-connector-java-5.1.27-bin.jar
```

3.5.3 运行 Spark SQL CLI

Spark SQL CLI 可以很方便的在本地运行 Hive 元数据服务以及从命令行执行查询任务。在 Spark 目录下执行如下命令启动 Spark SQL CLI：

```
./bin/spark-sql
```

3.5.4 代码中使用 Hive

(1) 添加依赖：

```
<!-- https://mvnrepository.com/artifact/org.apache.spark/spark-hive -->
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-hive_2.11</artifactId>
    <version>2.1.1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.hive/hive-exec -->
<dependency>
    <groupId>org.apache.hive</groupId>
    <artifactId>hive-exec</artifactId>
    <version>1.2.1</version>
</dependency>
```

(2) 创建 SparkSession 时需要添加 hive 支持（红色部分）

```
val warehouseLocation: String = new File("spark-warehouse").getAbsolutePath

val session= SparkSession
.builder()
.appName("Spark Hive Example")
.config("spark.sql.warehouse.dir", warehouseLocation)
.enableHiveSupport()
.getOrCreate()
```

注意：蓝色部分为使用内置 Hive 需要指定一个 Hive 仓库地址。若使用的是外部 Hive，则需要将 hive-site.xml 添加到 ClassPath 下。

第 4 章 Spark SQL 之 DataFrame 操作

4.1 DataFrame 创建方式

1 通过 Seq 生成

```
val spark = SparkSession
.builder()
.appName(this.getClass.getSimpleName).master("local")
.getOrCreate()

val df = spark.createDataFrame(Seq(
    ("ming", 20, 15552211521L),
    ("hong", 19, 13287994007L),
    ("zhi", 21, 15552211523L)
)) toDF("name", "age", "phone")
```

```
df.show()
```

2 读取 Json 文件生成

json 文件内容

```
{"name":"ming","age":20,"phone":15552211521}  
{"name":"hong", "age":19,"phone":13287994007}  
{"name":"zhi", "age":21,"phone":15552211523}
```

代码

```
val dfJson = spark.read.format("json").load("/Users/HollySys/Repository/sparkLearn/data/student.json")  
dfJson.show()
```

3 读取 csv 文件生成

csv 文件

```
name,age,phone  
ming,20,15552211521  
hong,19,13287994007  
zhi,21,15552211523
```

代码:

```
val dfCsv = spark.read.format("csv").option("header", true).load("/Users/shirukai/Desktop/HollySys/Repository/sparkLearn/data/students.csv")  
dfCsv.show()
```

4 通过 Json 格式的 DataSet 生成

//第五种：通过 Json 格式的 DataSet 生成

```
val jsonDataSet = spark.createDataset(Array(  
    "{\"name\":\"ming\", \"age\":20, \"phone\":15552211521}\",  
    "{\"name\":\"hong\", \"age\":19, \"phone\":13287994007}\",  
    "{\"name\":\"zhi\", \"age\":21, \"phone\":15552211523}\",  
)  
val jsonDataSetDf = spark.read.json(jsonDataSet)  
jsonDataSetDf.show()
```

5 通过 csv 格式的 DataSet 生成

```
val scvDataSet = spark.createDataset(Array(  
    "ming,20,15552211521",  
    "hong,19,13287994007",  
    "zhi,21,15552211523",  
)  
spark.read.csv(scvDataSet).toDF("name", "age", "phone").show()
```

6 动态创建 schema

```
val schema = StructType(List(  
    StructField("name", StringType, true),  
    StructField("age", IntegerType, true),  
    StructField("phone", LongType, true)  
)  
val dataList = new util.ArrayList[Row]()  
dataList.add(Row("ming", 20, 15552211521L))  
dataList.add(Row("hong", 19, 13287994007L))  
dataList.add(Row("zhi", 21, 15552211523L))  
spark.createDataFrame(dataList, schema).show()
```

7 通过 jdbc 创建

```
//第八种：读取数据库（mysql）  
val options = new util.HashMap[String, String]()  
options.put("url", "jdbc:mysql://localhost:3306/spark")  
options.put("driver", "com.mysql.jdbc.Driver")  
options.put("user", "root")  
options.put("password", "hollysys")  
options.put("dbtable", "user")
```

```
spark.read.format("jdbc").options(options).load().show()
```

4.2 DataFrame 对象的 Action 操作

1) show

df.show(): 展示数据

df.show(10): 展示 10 条数据

df.show(True): 是否最多只显示 20 个字符, 默认为 true

df.show(3, False): 显示记录条数, 以及对过长字符串的显示格式

2) collect

获取所有数据到数组, 不同于前面的 show 方法, 这里的 collect 方法会将 jdbcDF 中的所有数据都获取到, 并返回一个 Array 对象。

```
df.collect()
```

3) collectAsList

功能和 collect 类似, 只不过将返回结构变成了 List 对象, 使用方法如下:

```
df.collectAsList()
```

4) describe(cols: String*):

获取指定字段的统计信息

这个方法可以动态的传入一个或多个 String 类型的字段名, 结果仍然为 DataFrame 对象, 用于统计数值类型字段的统计值, 比如 count, mean, stddev, min, max 等。使用方法如下, 其中 c1 字段为字符类型, c2 字段为整型, c4 字段为浮点型

```
df.describe("c1", "c2", "c4").show()
```

5) first, head, take, takeAsList

获取若干行记录

这里列出的四个方法比较类似, 其中

(1) first 获取第一行记录

(2) head 获取第一行记录, head(n: Int) 获取前 n 行记录

(3) take(n: Int) 获取前 n 行数据

(4) takeAsList(n: Int) 获取前 n 行数据, 并以 List 的形式展现

以 Row 或者 Array[Row] 的形式返回一行或多行数据。first 和 head 功能相同。

take 和 takeAsList 方法会将获得的数据返回到 Driver 端, 所以, 使用这两个方法时需要注意数据量, 以免 Driver 发生 OutOfMemoryError

4.3 DataFrame 对象的条件查询和 join 操作

1 where 条件相关

(1) where(conditionExpr: String)

SQL 语言中 where 关键字后的条件

```
df.where("id = 1 or c1 = 'b'").show()
```

(2) filter

根据字段进行筛选

传入筛选条件表达式, 得到 DataFrame 类型的返回结果, 和 where 使用条件相同

```
df.filter("id = 1 or c1 = 'b'").show()
```

2 查询指定字段

(1) select

获取指定字段值

根据传入的 String 类型字段名, 获取指定字段的值, 以 DataFrame 类型返回

```
示例: df.select("id", "c3").show(false)
```

还有一个重载的 select 方法，不是传入 String 类型参数，而是传入 Column 类型参数。可以实现 select id, id+1 from test 这种逻辑：df.select(jdbcDF("id"), jdbcDF("id") + 1).show(false)

(2) selectExpr

可以对指定字段进行特殊处理

可以直接对指定字段调用 UDF 函数，或者指定别名等。传入 String 类型参数，得到 DataFrame 对象。

示例，查询 id 字段， c3 字段取别名 time， c4 字段四舍五入：

```
df.selectExpr("id" , "c3 as time" , "round(c4)" ).show(false)
```

(3) col

获取指定字段

只能获取一个字段，返回对象为 Column 类型。

```
val idCol = jdbcDF.col( "id" )果略。
```

(4) apply

获取指定字段

只能获取一个字段，返回对象为 Column 类型

示例：

```
val idCol1 = df.apply("id")
```

```
val idCol2 = df("id")
```

(5) drop

去除指定字段，保留其他字段

返回一个新的 DataFrame 对象，其中不包含去除的字段，一次只能去除一个字段。

示例：

```
df.drop("id")
```

```
df.drop(jdbcDF("id"))
```

(6) dropDuplicates(colNames: Array[String])

删除相同的列，返回一个 dataframe

3、limit

limit 方法获取指定 DataFrame 的前 n 行记录，得到一个新的 DataFrame 对象。和 take 与 head 不同的是， limit 方法不是 Action 操作。

```
df.limit(3).show( false )
```

4 order by

(1) orderBy 和 sort

按指定字段排序，默认为升序

示例 1，按指定字段排序。加个-表示降序排序。sort 和 orderBy 使用方法相同

```
df.orderBy(- df("c4")).show(false)
```

//或者

```
df.orderBy(df("c4").desc).show(false)
```

示例 2，按字段字符串升序排序

```
df.orderBy("c4").show(false)
```

(2) sortWithinPartitions

和上面的 sort 方法功能类似，区别在于 sortWithinPartitions 方法返回的是按 Partition 排好序的 DataFrame 对象。

5 group by

(1) groupBy

根据字段进行 group by 操作，groupBy 方法有两种调用方式，可以传入 String 类型的字段名，也可传入 Column 类型的对象。使用方法如下，

```
df.groupBy("c1" )
```

```
df.groupBy( jdbcDF( "c1" ))
```

(2) cube 和 rollup

group by 的扩展，功能类似于 SQL 中的 group by cube/rollup，略。

(3) GroupedData 对象

该方法得到的是 GroupedData 类型对象，在 GroupedData 的 API 中提供了 group by 之后的操作，比如

`max(colNames: String*)`: 获取分组中指定字段或者所有的数字类型字段的最大值，只能作用于数字型字段

`min(colNames: String*)`: 获取分组中指定字段或者所有的数字类型字段的最小值，只能作用于数字型字段

`mean(colNames: String*)`: 获取分组中指定字段或者所有的数字类型字段的平均值，只能作用于数字型字段

`sum(colNames: String*)`: 获取分组中指定字段或者所有的数字类型字段的和值，只能作用于数字型字段

`count()`: 获取分组中的元素个数

6 distinct

(1) distinct

返回一个不包含重复记录的 DataFrame，即返回当前 DataFrame 中不重复的 Row 记录。该方法和接下来的 dropDuplicates() 方法不传入指定字段时的结果相同。

示例：`df.distinct()`

(2) dropDuplicates

根据指定字段去重，类似于 select distinct a, b 操作，示例：`df.dropDuplicates(Seq("c1"))`

7 agg

聚合操作调用的是 agg 方法，该方法有多种调用方式。一般与 groupBy 方法配合使用。

以下示例其中最简单直观的一种用法，对 id 字段求最大值，对 c4 字段求和。

```
df.agg("id" -> "max", "c4" -> "sum")
df.agg(max("age"), avg("salary"))
df.agg(Map("age" -> "max", "salary" -> "avg"))
```

8 union

unionAll 方法：对两个 DataFrame 进行组合，类似于 SQL 中的 UNION ALL 操作。

示例：`df.unionAll(df.limit(1))`

9 join

重点来了。在 SQL 语言中用得很多的就是 join 操作，DataFrame 中同样也提供了 join 的功能。

接下来隆重介绍 join 方法。在 DataFrame 中提供了六个重载的 join 方法。

(1) 笛卡尔积

```
df1.join(df2)
```

(2) using 一个字段形式

下面这种 join 类似于 a join b using column1 的形式，需要两个 DataFrame 中有相同的一个列名，

```
df1.join(df2, "id")
```

df1 和 df2 根据字段 id 进行 join 操作，结果如下，using 字段只显示一次。

(3) using 多个字段形式

除了上面这种 using 一个字段的情况外，还可以 using 多个字段，如下

```
df1.join(df2, Seq("id", "name"))
```

(4) 指定 join 类型

两个 DataFrame 的 join 操作有 inner, outer, left_outer, right_outer, leftsemi 类型。在上面的 using 多个字段的 join 情况下，可以写第三个 String 类型参数，指定 join 的类型，如下所示

```
df1.join(df2, Seq("id", "name"), "inner")
```

(5) 使用 Column 类型来 join

如果不使用 using 模式，灵活指定 join 字段的话，可以使用如下形式

```
df1.join(df2, df1("id") === df2("t1_id"))
```

(6) 在指定 join 字段同时指定 join 类型

```
示例：df1.join(df2, df1("id") === df2("t1_id"), "inner")
```

10 stat

stat 方法可以用于计算指定字段或指定字段之间的统计信息，比如方差，协方差等。这个方法返回一个 DataFramesStatFunctions 类型对象。

下面代码演示根据 c4 字段，统计该字段值出现频率在 30%以上的内容。在 df 中字段 c1 的内容为 "a, b, a, c, d, b"。其中 a 和 b 出现的频率为 2 / 6，大于 0.3。

```
df.stat.freqItems(Seq("c1"), 0.3).show()
```

11 intersect

intersect 方法可以计算出两个 DataFrame 中相同的记录，

```
df.intersect(df.limit(1)).show(false)
```

12 except

获取一个 DataFrame 中有另一个 DataFrame 中没有的记录

```
示例: df.except(df.limit(1)).show(false)
```

13 操作字段名

(1) withColumnRenamed:

重命名 DataFrame 中的指定字段名，如果指定的字段名不存在，不进行任何操作。下面示例中将 jdbcDF 中的 id 字段重命名为 idx。

```
df.withColumnRenamed("id", "idx")
```

(2) withColumn:

往当前 DataFrame 中新增一列，withColumn(colName: String, col: Column)方法根据指定 colName 往 DataFrame 中新增一列，如果 colName 已存在，则会覆盖当前列。

以下代码往 jdbcDF 中新增一个名为 id2 的列，

```
df.withColumn("id2", df("id")).show(false)
```

14、explode

行转列，有时候需要根据某个字段内容进行分割，然后生成多行，这时可以使用 explode 方法。

下面代码中，根据 c3 字段中的空格将字段内容进行分割，分割的内容存储在新的字段 c3_ 中，如下所示

```
df.explode("c3", "c3_"){time: String => time.split(" ")}
```

15 na

对两个数据表如 A, B 取 JOIN 操作的时候，其结果往往会出现 NULL 值的出现。这种情况是非常不利于后续的分析与计算的，特别是当涉及到对这个数值列进行各种聚合函数计算的时候。

Spark 为此提供了一个高级操作，就是：na.fill 的函数。其处理过程就是先构建一个 MAP，如下： val map = Map("列名 1" -> 指定数字, "列名 2" -> 指定数字,) 然后执行 df.na.fill(map)，即可实现对 NULL 值的填充。

(1) dropna

dropna()比较简单粗暴，他会把包含 Na 的那一行删除掉，当然也可以设置 dropna()中的 how 参数，来设定删除的规则，如 how='all' 就表示删除全部数据都是 Na 的哪一行： df.dropna(how='all')

(2) fillna

我们也可以使用 fillna 方法来对 Na 进行填充，一般的，我们可以指定填充的方法，一般用一个字典来制定方法。代码如下：

```
df.fillna({'gender':'M', 'student':'unknown', 'score':0})
```

把 gender 列的 Na 全部改为 M， student 列的 Na 全部改为 unknown， score 列的 Na 全部改为 0。

```
df.fillna(0)
```

只填写一个参数 0，所有的 Na 都被修改为了 0

16 分区

问题：发现 spark DataFrame.write 无论 format("csv").save(hdfsPath) 中是 csv、parquet、json，或者使用 write.csv() write.json() 保存之后都是一个目录，下面生成很多个文件，只有设置分区为一个时，才能在目录下只有一个.success 文件和一个分区数据文件（即小文件数据文件个数与分区个数对应）

所以要弄明白几个分区的意思：

(1) **repartition(numPartitions, *cols)**

重新分区(内存中, 用于并行度), 用于分区数变多, 设置变小一般也只是 action 算子时才开始 shuffling; 而且当参数 numPartitions 小于当前分区个数时会保持当前分区个数等于失效。

```
df.repartition(100,"month")
```

(2) **partitionBy(*cols)**

根据指定列进行分区 (主要磁盘存储, 影响生成磁盘文件数量, 后续再从存储中载入影响并行度), 相似的在一个区, 并没有参数来指定多少个分区, 而且仅用于 PairRdd

```
df.map(lambda x:(x[0],x[1],x[2])).toDF(["month","day","value"]).partitionBy(["month","day"])
```

(3) **coalesce(numPartitions)**

联合分区, 用于将分区变少。不能指定按某些列联合分区。

```
df.coalesce(1).rdd.getNumPartitions()
```

17 alias

别名

18 pivot

透视函数, 只可意会

```
import spark.implicits._
```

```
val ds = spark.createDataset(list)
```

```
val df = ds.toDF("year", "month", "num")
```

```
val res:org.apache.spark.sql.DataFrame = df.groupBy("year").pivot("month").sum("num")
```

```
df.show
```

```
+---+---+---+
|year|month|num|
+---+---+---+
|2017|    1|100|
|2017|    1| 50|
|2017|    2|100|
|2017|    3| 50|
|2018|    2|200|
|2018|    2|100|
+---+---+---+
```

```
res.show
```

```
+---+---+---+
|year|    1|    2|    3|
+---+---+---+
|2018| null|300|null|
|2017| 150|100|  50|
+---+---+---+
```

19 udf

调用 sqlcontext 里面的 udf 函数

```
// 对 test 这个 String 计算它的长度
```

```
sqlContext.udf.register("str",(_:String).length)
```

```
sqlContext.sql("select str('test')")
```

20 as(alias: String)

返回一个新的 dataframe 类型, 就是原来的一个别名

21 explode

```
explode[A, B](inputColumn: String, outputColumn: String)(f: (A) ⇒ TraversableOnce[B])(implicit arg0:
```

```
scala.reflect.api.JavaUniverse.TypeTag[B])
```

返回值是 dataframe 类型, 这个 api 将一个字段进行更多行的拆分

```
df.explode("name","names") {name :String=> name.split(" ")}.show();
```

第 5 章 Spark SQL 实战

5.1 数据说明

数据集是货品交易数据集。



每个订单可能包含多个货品，每个订单可以产生多次交易，不同的货品有不同的单价。

5.2 加载数据

```
tbStock:  
case class tbStock(ordernumber:String,locationid:String,dateid:String) extends Serializable  
  
val tbStockRdd = spark.sparkContext.textFile("tbStock.txt")  
val tbStockDS = tbStockRdd.map(_.split(",")).map(attr=>tbStock(attr(0),attr(1),attr(2))).toDS  
  
scala> tbStockDS.show()  
+-----+-----+  
| ordernumber|locationid|    dataid|  
+-----+-----+  
|BYSL00000893|      ZHAO|2007-8-23|  
|BYSL00000897|      ZHAO|2007-8-24|  
|BYSL00000898|      ZHAO|2007-8-25|  
|BYSL00000899|      ZHAO|2007-8-26|  
|BYSL00000900|      ZHAO|2007-8-26|  
|BYSL00000901|      ZHAO|2007-8-27|  
|BYSL00000902|      ZHAO|2007-8-27|  
|BYSL00000904|      ZHAO|2007-8-28|  
|BYSL00000905|      ZHAO|2007-8-28|  
|BYSL00000906|      ZHAO|2007-8-28|  
|BYSL00000907|      ZHAO|2007-8-29|  
|BYSL00000908|      ZHAO|2007-8-30|  
|BYSL00000909|      ZHAO|2007-9-1|  
|BYSL00000910|      ZHAO|2007-9-1|  
|BYSL00000911|      ZHAO|2007-8-31|  
|BYSL00000912|      ZHAO|2007-9-2|
```

```

|BYSL00000913| ZHAO| 2007-9-3|
|BYSL00000914| ZHAO| 2007-9-3|
|BYSL00000915| ZHAO| 2007-9-4|
|BYSL00000916| ZHAO| 2007-9-4|
+-----+
only showing top 20 rows

```

tbStockDetail:

```

case class tbStockDetail(ordernumber:String, rownum:Int, itemid:String, number:Int, price:Double, amount:Double) extends Serializable

val tbStockDetailRdd = spark.sparkContext.textFile("tbStockDetail.txt")
val tbStockDetailDS = tbStockDetailRdd.map(_.split(",")).map(attr=> tbStockDetail(attr(0).trim().toInt,attr(2).trim().toInt,attr(3).trim().toDouble, attr(4).trim().toDouble, attr(5).trim().toDouble)).toDS

scala> tbStockDetailDS.show()
+-----+-----+-----+-----+
| ordernumber|rownum| itemid |number|price|amount|
+-----+-----+-----+-----+
|BYSL00000893| 0|FS527258160501| -1|268.0|-268.0|
|BYSL00000893| 1|FS527258169701| 1|268.0| 268.0|
|BYSL00000893| 2|FS527230163001| 1|198.0| 198.0|
|BYSL00000893| 3|24627209125406| 1|298.0| 298.0|
|BYSL00000893| 4|K9527220210202| 1|120.0| 120.0|
|BYSL00000893| 5|01527291670102| 1|268.0| 268.0|
|BYSL00000893| 6|QY527271800242| 1|158.0| 158.0|
|BYSL00000893| 7|ST040000010000| 8| 0.0| 0.0|
|BYSL00000897| 0|04527200711305| 1|198.0| 198.0|
|BYSL00000897| 1|MY627234650201| 1|120.0| 120.0|
|BYSL00000897| 2|01227111791001| 1|249.0| 249.0|
|BYSL00000897| 3|MY627234610402| 1|120.0| 120.0|
|BYSL00000897| 4|01527282681202| 1|268.0| 268.0|
|BYSL00000897| 5|84126182820102| 1|158.0| 158.0|
|BYSL00000897| 6|K9127105010402| 1|239.0| 239.0|
|BYSL00000897| 7|QY127175210405| 1|199.0| 199.0|
|BYSL00000897| 8|24127151630206| 1|299.0| 299.0|
|BYSL00000897| 9|G1126101350002| 1|158.0| 158.0|
|BYSL00000897| 10|FS527258160501| 1|198.0| 198.0|
|BYSL00000897| 11|ST040000010000| 13| 0.0| 0.0|
+-----+
only showing top 20 rows

```

tbDate:

```

case class tbDate(dateid:String, years:Int, theyear:Int, month:Int, day:Int, weekday:Int, week:Int, quarter:Int, period:Int, halfmonth:Int) extends Serializable

val tbDateRdd = spark.sparkContext.textFile("tbDate.txt")

val tbDateDS = tbDateRdd.map(_.split(",")).map(attr=> tbDate(attr(0).trim().toInt, attr(1).trim().toInt, attr(2).trim().toInt, attr(3).trim().toInt, attr(4).trim().toInt, attr(5).trim().toInt, attr(6).trim().toInt, attr(7).trim().toInt, attr(8).trim().toInt, attr(9).trim().toInt)).toDS

scala> tbDateDS.show()
+-----+-----+-----+-----+-----+-----+-----+-----+
| dateid| years|theyear|month|day|weekday|week|quarter|period|halfmonth|
+-----+-----+-----+-----+-----+-----+-----+
| 2003-1-1|200301| 2003| 1| 1| 3| 1| 1| 1| 1|

```

2003-1-2 200301 2003 1 2 4 1 1 1 1
2003-1-3 200301 2003 1 3 5 1 1 1 1
2003-1-4 200301 2003 1 4 6 1 1 1 1
2003-1-5 200301 2003 1 5 7 1 1 1 1
2003-1-6 200301 2003 1 6 1 2 1 1 1
2003-1-7 200301 2003 1 7 2 2 1 1 1
2003-1-8 200301 2003 1 8 3 2 1 1 1
2003-1-9 200301 2003 1 9 4 2 1 1 1
2003-1-10 200301 2003 1 10 5 2 1 1 1
2003-1-11 200301 2003 1 11 6 2 1 2 1
2003-1-12 200301 2003 1 12 7 2 1 2 1
2003-1-13 200301 2003 1 13 1 3 1 2 1
2003-1-14 200301 2003 1 14 2 3 1 2 1
2003-1-15 200301 2003 1 15 3 3 1 2 1
2003-1-16 200301 2003 1 16 4 3 1 2 2
2003-1-17 200301 2003 1 17 5 3 1 2 2
2003-1-18 200301 2003 1 18 6 3 1 2 2
2003-1-19 200301 2003 1 19 7 3 1 2 2
2003-1-20 200301 2003 1 20 1 4 1 2 2

only showing top 20 rows

注册表：

```
scala> tbStockDS.createOrReplaceTempView("tbStock")
scala> tbDateDS.createOrReplaceTempView("tbDate")
scala> tbStockDetailDS.createOrReplaceTempView("tbStockDetail")
```

5.3 计算所有订单中每年的销售单数、销售总额

统计所有订单中每年的销售单数、销售总额

三个表连接后以 `count(distinct a.ordernumber)` 计销售单数，`sum(b.amount)` 计销售总额



```
SELECT c.theyear, COUNT(DISTINCT a.ordernumber), SUM(b.amount)
FROM tbStock a
JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
JOIN tbDate c ON a.dateid = c.dateid
GROUP BY c.theyear
ORDER BY c.theyear
```

```
spark.sql("SELECT c.theyear, COUNT(DISTINCT a.ordernumber), SUM(b.amount) FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear ORDER BY c.theyear").show
```

结果如下：

theyear	count(DISTINCT ordernumber)	sum(amount)
2003	20	1000

2004	1094	3268115.499199999
2005	3828	1.3257564149999991E7
2006	3772	1.3680982900000006E7
2007	4885	1.6719354559999993E7
2008	4861	1.467429530000001E7
2009	2619	6323697.189999999
2010	94	210949.659999999997

5.4 计算所有订单每年最大金额订单的销售额

目标：统计每年最大金额订单的销售额：



1) 统计每年，每个订单一共有多少销售额

```
SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount
FROM tbStock a
JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
GROUP BY a.dateid, a.ordernumber
```

```
spark.sql("SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber GROUP BY a.dateid, a.ordernumber").show
```

结果如下：

dateid	ordernumber	SumOfAmount
2008-4-9 BYSL00001175		350.0
2008-5-12 BYSL00001214		592.0
2008-7-29 BYSL00011545		2064.0
2008-9-5 DGSL00012056		1782.0
2008-12-1 DGSL00013189		318.0
2008-12-18 DGSL00013374		963.0
2009-8-9 DGSL00015223		4655.0
2009-10-5 DGSL00015585		3445.0
2010-1-14 DGSL00016374		2934.0
2006-9-24 GCSL00000673	3556.1000000000004	
2007-1-26 GCSL00000826	9375.19999999999	
2007-5-24 GCSL00001020	6171.300000000002	
2008-1-8 GCSL00001217		7601.6
2008-9-16 GCSL00012204		2018.0
2006-7-27 GHSL00000603		2835.6
2006-11-15 GHSL00000741		3951.94
2007-6-6 GHSL00001149		0.0
2008-4-18 GHSL00001631		12.0
2008-7-15 GHSL00011367		578.0
2009-5-8 GHSL00014637		1797.6

2) 以上一步查询结果为基础表, 和表 tbDate 使用 dateid join, 求出每年最大金额订单的销售额

```
SELECT theyear, MAX(c.SumOfAmount) AS SumOfAmount
FROM (SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount
      FROM tbStock a
      JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
     GROUP BY a.dateid, a.ordernumber
    ) c
   JOIN tbDate d ON c.dateid = d.dateid
GROUP BY theyear
ORDER BY theyear DESC
```

```
spark.sql("SELECT theyear, MAX(c.SumOfAmount) AS SumOfAmount FROM (SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber GROUP BY a.dateid, a.ordernumber ) c JOIN tbDate d ON c.dateid = d.dateid GROUP BY theyear ORDER BY theyear DESC").show
```

结果如下:

theyear	SumOfAmount
2010	13065.280000000002
2009	25813.200000000008
2008	55828.0
2007	159126.0
2006	36124.0
2005	38186.399999999994
2004	23656.799999999997

5.5 计算所有订单中每年最畅销货品

目标: 统计每年最畅销货品(哪个货品销售额 amount 在当年最高, 哪个就是最畅销货品)



第一步、求出每年每个货品的销售额

```
SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount
FROM tbStock a
   JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
   JOIN tbDate c ON a.dateid = c.dateid
GROUP BY c.theyear, b.itemid
```

```
spark.sql("SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid").show
```

结果如下:

theyear	itemid	SumOfAmount
---------	--------	-------------

2004 43824480810202	4474.72
2006 YA214325360101	556.0
2006 BT624202120102	360.0
2007 AK215371910101	24603.639999999992
2008 AK216169120201	29144.19999999997
2008 YL526228310106	16073.09999999999
2009 KM529221590106	5124.800000000001
2004 HT224181030201	2898.600000000004
2004 SG224308320206	7307.06
2007 04426485470201	14468.800000000001
2007 84326389100102	9134.11
2007 B4426438020201	19884.2
2008 YL427437320101	12331.79999999997
2008 MH215303070101	8827.0
2009 YL629228280106	12698.4
2009 BL529298020602	2415.8
2009 F5127363019006	614.0
2005 24425428180101	34890.74
2007 YA214127270101	240.0
2007 MY127134830105	11099.92

第二步、在第一步的基础上，统计每年单个货品中的最大金额

```
SELECT d.theyear, MAX(d.SumOfAmount) AS MaxOfAmount
FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount
      FROM tbStock a
      JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
      JOIN tbDate c ON a.dateid = c.dateid
      GROUP BY c.theyear, b.itemid
     ) d
GROUP BY d.theyear
```

```
spark.sql("SELECT d.theyear, MAX(d.SumOfAmount) AS MaxOfAmount FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid ) d GROUP BY d.theyear").show
```

结果如下：

thetyear	MaxOfAmount
2007	70225.1
2006	113720.6
2004 53401.75999999995	
2009	30029.2
2005 56627.32999999994	
2010	4494.0
2008 98003.6000000003	

第三步、用最大销售额和统计好的每个货品的销售额 join，以及用年 join，集合得到最畅销货品那一行信息

```
SELECT DISTINCT e.theyear, e.itemid, f.MaxOfAmount
FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount
      FROM tbStock a
      JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
      JOIN tbDate c ON a.dateid = c.dateid
      GROUP BY c.theyear, b.itemid
     ) e
JOIN (SELECT d.theyear, MAX(d.SumOfAmount) AS MaxOfAmount
      FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount
```

```

        FROM tbStock a
            JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
            JOIN tbDate c ON a.dateid = c.dateid
            GROUP BY c.theyear, b.itemid
        ) d
        GROUP BY d.theyear
    ) f ON e.theyear = f.theyear
        AND e.SumOfAmount = f.MaxOfAmount
ORDER BY e.theyear

```

```

spark.sql("SELECT DISTINCT e.theyear, e.itemid, f.maxofamount FROM (SELECT c.theyear, b.itemid, SUM(b.amount)
AS sumofamount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.
dateid = c.dateid GROUP BY c.theyear, b.itemid ) e JOIN (SELECT d.theyear, MAX(d.sumofamount) AS maxofamou
nt FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS sumofamount FROM tbStock a JOIN tbStockDetail b ON
a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid ) d GROU
P BY d.theyear ) f ON e.theyear = f.theyear AND e.sumofamount = f.maxofamount ORDER BY e.theyear").show

```

结果如下：

thetyear	itemid	maxofamount
2004	JY424420810101	53401.759999999995
2005	24124118880102	56627.32999999994
2006	JY425468460101	113720.6
2007	JY425468460101	70225.1
2008	E2628204040101	98003.60000000003
2009	YL327439080102	30029.2
2010	SQ429425090101	4494.0

SparkStreaming

第 1 章 SparkStreaming 概述

1.1 SparkStreaming 是什么

Spark Streaming 用于流式数据的处理。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。



和 Spark 基于 RDD 的概念很相似，Spark Streaming 使用离散化流(discretized stream)作为抽象表示，叫作 DStream。DStream 是随时间推移而收到的数据的序列。在内部，每个时间区间收到的数据都作为 RDD 存在，而 DStream 是由这些 RDD 所组成的序列(因此得名“离散化”)。

1.2 SparkStreaming 特点

1.易用

Ease of Use

Build applications through high-level operators.

Spark Streaming brings Spark's [language-integrated API](#) to stream processing, letting you write streaming jobs the same way you write batch jobs. It supports Java, Scala and Python.

```
TwitterUtils.createStream(...)  
.filter(_.getText.contains("spark"))  
.countByWindow(Seconds(5))
```

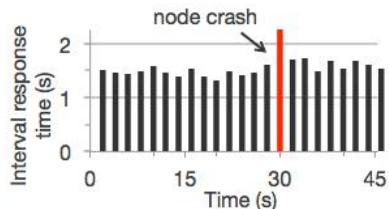
Counting tweets on a sliding window

2.容错

Fault Tolerance

Stateful exactly-once semantics out of the box.

Spark Streaming recovers both lost work and operator state (e.g. sliding windows) out of the box, without any extra code on your part.



3.易整合到 Spark 体系

Spark Integration

Combine streaming with batch and interactive queries.

By running on Spark, Spark Streaming lets you reuse the same code for batch processing, join streams against historical data, or run ad-hoc queries on stream state. Build powerful interactive applications, not just analytics.

```
stream.join(historicCounts).filter {  
    case (word, (curCount, oldCount)) =>  
        curCount > oldCount  
}
```

Find words with higher frequency than historic data

1.3 SparkStreaming 架构

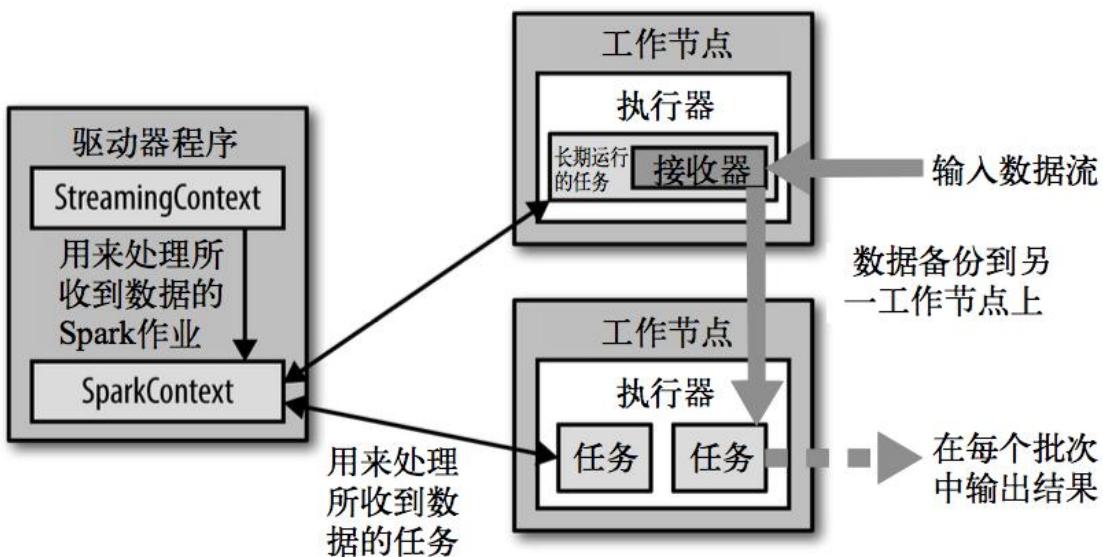


图 1-1 SparkStreaming 架构图

第 2 章 Dstream 入门

2.1 WordCount 案例实操

1. 需求：使用 netcat 工具向 9999 端口不断的发送数据，通过 SparkStreaming 读取端口数据并统计不同单词出现的次数

2. 添加依赖

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.11</artifactId>
    <version>2.1.1</version>
</dependency>
```

3. 编写代码

```
import org.apache.spark.streaming.DStream, ReceiverInputDStream
import org.apache.spark.streaming.Seconds, StreamingContext
import org.apache.spark.SparkConf

object StreamWordCount {
    def main(args: Array[String]): Unit = {
        //1. 初始化 Spark 配置信息
        val sparkConf = new SparkConf().setMaster("local[*]").setAppName("StreamWordCount")

        //2. 初始化 SparkStreamingContext，设置窗口时间
        val ssc = new StreamingContext(sparkConf, Seconds(5))

        //3. 通过监控端口创建 DStream，读进来的数据为一行行
        val lineStreams = ssc.socketTextStream("hadoop102", 9999)
        //将每一行数据做切分，形成一个个单词
        val wordStreams = lineStreams.flatMap(_.split(" "))
        //将单词映射成元组 (word, 1)
        val wordAndOneStreams = wordStreams.map((_, 1))
        //将相同的单词次数做统计
        val wordAndCountStreams = wordAndOneStreams.reduceByKey(_ + _)
        //打印
        wordAndCountStreams.print()

        //启动 SparkStreamingContext
        ssc.start()
        ssc.awaitTermination()
    }
}
```

4. 启动程序并通过 NetCat 发送数据：

```
[atguigu@hadoop102 spark]$ nc -lk 9999
hello atguigu
```

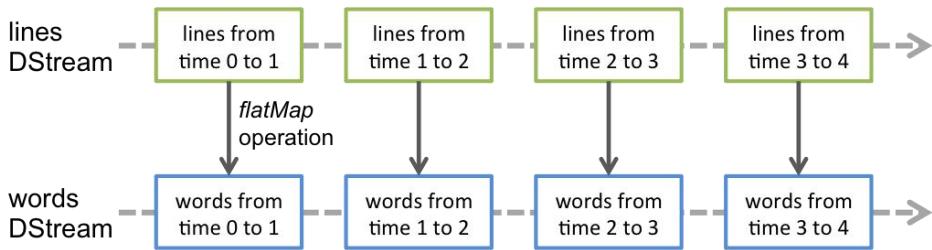
注意：如果程序运行时，log 日志太多，可以将 spark conf 目录下的 log4j 文件里面的日志级别改成 WARN。

2.2 WordCount 解析

Discretized Stream 是 Spark Streaming 的基础抽象，代表持续性的数据流和经过各种 Spark 原语操作后的结果数据流。在内部实现上，DStream 是一系列连续的 RDD 来表示。每个 RDD 含有一段时间间隔内的数据，如下图：



对数据的操作也是按照 RDD 为单位来进行的



计算过程由 Spark engine 来完成



第 3 章 Dstream 创建

Spark Streaming 原生支持一些不同的数据源。一些“核心”数据源已经被打包到 Spark Streaming 的 Maven 工件中，而其他的一些则可以通过 spark-streaming-kafka 等附加工作件获取。每个接收器都以 Spark 执行器程序中一个长期运行的任务的形式运行，因此会占据分配给应用的 CPU 核心。此外，我们还需要有可用的 CPU 核心来处理数据。这意味着如果要运行多个接收器，就必须至少有和接收器数目相同的核心数，还要加上用来完成计算所需要的核心数。例如，如果我们想要在流计算应用中运行 10 个接收器，那么至少需要为应用分配 11 个 CPU 核心。所以如果在本地模式运行，不要使用 local[1]。

3.1 文件数据源

3.1.1 用法及说明

文件数据流：能够读取所有 HDFS API 兼容的文件系统文件，通过 `fileStream` 方法进行读取，Spark Streaming 将会监控 `dataDirectory` 目录并不断处理移动进来的文件，记住目前不支持嵌套目录。

```
streamingContext.textFileStream(dataDirectory)
```

注意事项：

- 1) 文件需要有相同的数据格式；
- 2) 文件进入 `dataDirectory` 的方式需要通过移动或者重命名来实现；
- 3) 一旦文件移动进目录，则不能再修改，即便修改了也不会读取新数据；

3.1.2 案例实操

(1) 在 HDFS 上建好目录

```
[atguigu@hadoop102 spark]$ hadoop fs -mkdir /fileStream
```

(2) 在 /opt/module/data 创建三个文件

```
[atguigu@hadoop102 data]$ touch a.tsv
[atguigu@hadoop102 data]$ touch b.tsv
[atguigu@hadoop102 data]$ touch c.tsv
```

添加如下数据：

```
Hello atguigu
Hello spark
```

(3) 编写代码

```
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.DStream
```

```

object FileStream {
    def main(args: Array[String]): Unit = {
        //1.初始化 Spark 配置信息
        Val sparkConf = new SparkConf().setMaster("local[*]").setAppName("StreamWordCount")
        //2.初始化 SparkStreamingContext
        val ssc = new StreamingContext(sparkConf, Seconds(5))
        //3.监控文件夹创建 DStream
        val dirStream = ssc.textFileStream("hdfs://hadoop102:9000/fileStream")
        //4.将每一行数据做切分，形成一个个单词
        val wordStreams = dirStream.flatMap(_.split("\t"))
        //5.将单词映射成元组 (word,1)
        val wordAndOneStreams = wordStreams.map((_, 1))
        //6.将相同的单词次数做统计
        val wordAndCountStreams] = wordAndOneStreams.reduceByKey(_ + _)
        //7.打印
        wordAndCountStreams.print()
        //8.启动 SparkStreamingContext
        ssc.start()
        ssc.awaitTermination()
    }
}

```

(4) 启动程序并向 fileStream 目录上传文件

```

[atguigu@hadoop102 data]$ hadoop fs -put ./a.tsv /fileStream
[atguigu@hadoop102 data]$ hadoop fs -put ./b.tsv /fileStream
[atguigu@hadoop102 data]$ hadoop fs -put ./c.tsv /fileStream

```

(5) 获取计算结果

```
Time: 1539073810000 ms
```

```
Time: 1539073815000 ms
```

```
(Hello,4)
(spark,2)
(atguigu,2)
```

```
Time: 1539073820000 ms
```

```
(Hello,2)
(spark,1)
(atguigu,1)
```

```
Time: 1539073825000 ms
```

3.2 RDD 队列（了解）

3.2.1 用法及说明

测试过程中，可以通过使用 `ssc.queueStream(queueOfRDDs)` 来创建 DStream，每一个推送到这个队列中的 RDD，都会作为一个 DStream 处理。

3.2.2 案例实操

- 1) 需求：循环创建几个 RDD，将 RDD 放入队列。通过 SparkStream 创建 Dstream，计算 WordCount
- 2) 编写代码

```

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

```

```

import org.apache.spark.streaming.dstream.{DStream, InputDStream}
import org.apache.spark.streaming.{Seconds, StreamingContext}
import scala.collection.mutable

object RDDStream {
    def main(args: Array[String]) {
        //1. 初始化 Spark 配置信息
        val conf = new SparkConf().setMaster("local[*]").setAppName("RDDStream")

        //2. 初始化 SparkStreamingContext
        val ssc = new StreamingContext(conf, Seconds(4))

        //3. 创建 RDD 队列
        val rddQueue = new mutable.Queue[RDD[Int]]()

        //4. 创建 QueueInputDStream
        val inputStream = ssc.queueStream(rddQueue, oneAtATime = false)

        //5. 处理队列中的 RDD 数据
        val mappedStream = inputStream.map(_._1)
        val reducedStream = mappedStream.reduceByKey(_ + _)

        //6. 打印结果
        reducedStream.print()

        //7. 启动任务
        ssc.start()

        //8. 循环创建并向 RDD 队列中放入 RDD
        for (i <- 1 to 5) {
            rddQueue += ssc.sparkContext.makeRDD(1 to 300, 10) //1 to 300 是数据, 10 是分区数
            Thread.sleep(2000)
        }
        ssc.awaitTermination()
    }
}

```

3) 结果展示

Time: 1539075280000 ms

(224,2)
(160,2)
(296,2)
(96,2)
(56,2)
(112,2)
(120,2)
(280,2)
(16,2)
(184,2)

Time: 1539075284000 ms

(224,2)
(160,2)
(296,2)
(96,2)

```
(56,2)
(112,2)
(120,2)
(280,2)
(16,2)
(184,2)
```

```
Time: 1539075288000 ms
```

```
(224,2)
(160,2)
(296,2)
(96,2)
(56,2)
(112,2)
(120,2)
(280,2)
(16,2)
(184,2)
```

```
Time: 1539075292000 ms
```

3.3 Socket 数据源

3.3.1 用法及说明

需要继承 Receiver，并实现 onStart、onStop 方法来自定义数据源采集。

3.3.2 案例实操

1) 需求：Socket 数据源，实现监控某个端口号，获取该端口号内容。

2) 代码实现

```
import java.io.{BufferedReader, InputStreamReader}
import java.net.Socket
import java.nio.charset.StandardCharsets
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.receiver.Receiver

class CustomerReceiver(host: String, port: Int) extends Receiver[String](StorageLevel.MEMORY_ONLY) {
    //最初启动的时候，调用该方法，作用是调用 receive()
    override def onStart(): Unit = {
        new Thread("Socket Receiver") {
            override def run() {
                receive()
            }
        }.start()
    }

    //读数据并将数据发送给 Spark
    def receive(): Unit = {
        //创建一个 Socket
        var socket: Socket = new Socket(host, port)
        //定义一个变量，用来接收端口传过来的数据
        var input: String = null
        //创建一个 BufferedReader 用于读取端口传来的数据
        val reader = new BufferedReader(new InputStreamReader(socket.getInputStream, StandardCharsets.UTF_8))
        //读取数据
        input = reader.readLine()
    }
}
```

```

//当 receiver 没有关闭并且输入数据不为空，则循环发送数据给 Spark
while (!isStopped() && input != null) {
    store(input) // 父类 Reciver 的方法，调用进程方法给子进程推送数据
    input = reader.readLine()
}
//跳出循环则关闭资源
reader.close()
socket.close()

//重启任务
restart("restart")
}
override def onStop(): Unit = {}
}

```

3) 使用自定义的数据源采集数据

```

import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.DStream

object FileStream {
    def main(args: Array[String]): Unit = {
        val sparkConf = new SparkConf().setMaster("local[*]").setAppName("StreamWordCount")
        val ssc = new StreamingContext(sparkConf, Seconds(5))

        // 创建自定义 receiver 的 Streaming
        val lineStream = ssc.receiverStream(new CustomerReceiver("hadoop102", 9999))

        // 数据处理
        val wordStreams = lineStream.flatMap(_.split("\t"))
        val wordAndOneStreams = wordStreams.map((_, 1))
        val wordAndCountStreams] = wordAndOneStreams.reduceByKey(_ + _)

        // 打印
        wordAndCountStreams.print()

        // 启动 SparkStreamingContext
        ssc.start()
        ssc.awaitTermination()
    }
}

```

3.4 Kafka 数据源（重点）

3.4.1 用法及说明

在工程中需要引入 Maven 工件 spark-streaming-kafka_2.10 来使用它。包内提供的 KafkaUtils 对象可以在 StreamingContext 和 JavaStreamingContext 中以你的 Kafka 消息创建出 DStream。由于 KafkaUtils 可以订阅多个主题，因此它创建出的 DStream 由成对的主题和消息组成。要创建出一个流数据，需要使用 StreamingContext 实例、一个由逗号隔开的 ZooKeeper 主机列表字符串、消费者组的名字(唯一名字)，以及一个从主题到针对这个主题的接收器线程数的映射表来调用 createStream()方法。

3.4.2 案例实操

需求：通过 SparkStreaming 从 Kafka 读取数据，并将读取过来的数据做简单计算(WordCount)，最终打印到控制台。

(1) 导入依赖

```

<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming-kafka-0-8_2.11</artifactId>

```

```
<version>2.1.1</version>
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.2</version>
</dependency>
```

(2) 编写代码

```
import kafka.serializer.StringDecoder
import org.apache.kafka.clients.consumer.ConsumerConfig
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.dstream.ReceiverInputDStream
import org.apache.spark.streaming.kafka.KafkaUtils
import org.apache.spark.streaming.{Seconds, StreamingContext}

object KafkaSparkStreaming {
    def main(args: Array[String]): Unit = {
        //1.创建 SparkConf 并初始化 SSC
        val sparkConf: SparkConf = new SparkConf().setMaster("local[*]").setAppName("KafkaSparkStreaming")
        val ssc = new StreamingContext(sparkConf, Seconds(5))

        //2.定义 kafka 参数
        val brokers = "hadoop102:9092,hadoop103:9092,hadoop104:9092"
        val topic = "source"
        val consumerGroup = "spark"

        //3.将 kafka 参数映射为 map
        val kafkaParam: Map[String, String] = Map[String, String](
            ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG -> "org.apache.kafka.common.serialization.StringDeserializer",
            ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG -> "org.apache.kafka.common.serialization.StringDeserializer",
            ConsumerConfig.GROUP_ID_CONFIG -> consumerGroup,
            ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG -> brokers
        )

        //4.通过 KafkaUtil 创建 kafkaDSteam
        val kafkaDSteam: ReceiverInputDStream[(String, String)] = KafkaUtils.createStream[String, String, StringDecoder, StringDecoder](
            ssc,
            kafkaParam,
            Set(topic),
            StorageLevel.MEMORY_ONLY
        )

        //5.对 kafkaDSteam 做计算 (WordCount)
        kafkaDSteam.foreachRDD {
            rdd => {
                val word: RDD[String] = rdd.flatMap(_.split(" "))
                val wordAndOne: RDD[(String, Int)] = word.map(_ + 1)
                val wordAndCount: RDD[(String, Int)] = wordAndOne.reduceByKey(_ + _)
                wordAndCount.collect().foreach(println)
            }
        }
    }
}
```

```

    //6.启动 SparkStreaming
    ssc.start()
    ssc.awaitTermination()
}
}

```

3.5 Kafka 数据源高级开发（重点）

3.5.1 代码需求

需求：

- 1、SparkStreaming 多 DirectStream 方式去对接 kafka 数据源
- 2、更新广播变量替代 cogroup fileStream 达到更改配置的目的
- 3、累加器重置

3.5.2 代码实现

```

import java.io.{BufferedReader, InputStreamReader}
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.{FSDataInputStream, FileStatus, FileSystem, Path}
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.TopicPartition
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.SparkConf
import org.apache.spark.broadcast.Broadcast
import org.apache.spark.streaming.dstream.{DStream, InputDStream}
import org.apache.spark.streaming.kafka010.{ConsumerStrategies, ConsumerStrategy, KafkaUtils, LocationStrategies}
import org.apache.spark.streaming.{Durations, StreamingContext}
import org.apache.spark.util.LongAccumulator

import scala.collection.mutable
import scala.collection.mutable.Map
import scala.collection.mutable.ListBuffer

object SparkStreamingKafkaBroadCastUpdate {
  def main(args: Array[String]): Unit = {
    val topic = "hainiu_test"
    val brokers = "s1.hadoop:9092,s2.hadoop:9092,s3.hadoop:9092,s4.hadoop:9092,s5.hadoop:9092"

    //这里设置 cpu cores 为 1 的时候也能运行，说明 KafkaUtils.createDirectStream 是不需要 receiver 占用一个 cpu cores 的，而 KafkaUtils.createStream 是需要的，这种模式是出现在 sparkStreaming-kafka-0.80 版本的，0.10.x 版本已经抛弃了。这个的 cpu 为什么设置为 25？因为对于流式计算来说目的是快速完成每个任务的运算，所以在任务生成的时候每个 task 都应有对应的 cpu，不让其等待，运算速度才会更快；再者由于是本地环境，这个 25 个 cpu 中有一个 cpu 是为了 driver 设置的，在现实的分布式环境中设置 executor 为 24，就是 kafka 中 topic 的 partition 数据就可以了，然后 driver 单独设置
    val conf: SparkConf = new SparkConf().setAppName("SparkStreamingKafka").setMaster("local[25]")

    //设置 blockInterval 来调整并发数，也就是 spark 的 RDD 分区数或者说是 task 的数量，这个配置对于 Streaming Context 创建带 receiver 的流是起作用的，比如 socket 或者 KafkaUtils.createStream，但是对于 KafkaUtils.createDirect Stream（kafka 的直连模式）创建的流是不起作用的，因为直连模式是根据 topic 的分区数来决定并发度的，也就是 task 会直接连接到 kafka 中的 topic 的 partition 上，所以这里设置 blockInterval 对 RDD 的 partition 的划分是不起作用的
    //conf.set("spark.streaming.blockInterval","1000ms")
    val streamingContext = new StreamingContext(conf, Durations.seconds(5))

    //配置 kafka 信息
    val kafkaParams = new mutable.HashMap[String, Object]()
    kafkaParams += "bootstrap.servers" -> brokers
    kafkaParams += "group.id" -> "group15"
  }
}

```

```

//这两个的 deserializer 是 sparkStreaming 做为 consumer 时使用的
kafkaParams += "key.deserializer" -> classOf[StringDeserializer].getName
kafkaParams += "value.deserializer" -> classOf[StringDeserializer].getName
//这两个的 serializer 是 sparkStreaming 做为 producer 时使用的
//kafkaParams += "key.serializer" -> classOf[StringSerializer].getName
//kafkaParams += "value.serializer" -> classOf[StringSerializer].getName

//这里的作用是让每个 DStream 指定负责那些 partition，这里使用的分配的方式让每个 DStream 读取指定的 partition，因为我们的 topic 有 24 个 partition，所以我们这里设置 3 个 DStream，之后每个 DStream 配置成读取 8 个 Partition，但是对于直连流来讲 RDD 的 Partition 是被动由 Kafka 的 Topic 的 Partition 来决定的，所以也无法改变接受速度
val directDStreamList = new ListBuffer[InputDStream[ConsumerRecord[String, String]]]
for (i <- 0 until 3) {
    val partitions = new ListBuffer[TopicPartition]
    for (ii <- (8 * i) until (8 * i) + 8) {
        val partition = new TopicPartition(topic, ii)
        partitions += partition
    }
    val value: ConsumerStrategy[String, String] = ConsumerStrategies.Assign(partitions, kafkaParams)
    val directDStream: InputDStream[ConsumerRecord[String, String]] = KafkaUtils.createDirectStream(streamingContext, LocationStrategies.PreferConsistent, value)
    directDStreamList += directDStream
}

val unionDStream: DStream[ConsumerRecord[String, String]] = streamingContext.union(directDStreamList)
val words: DStream[String] = unionDStream.flatMap(_.value().split(" "))

//创建广播变量
var mapBroadCast: Broadcast[Map[String, String]] = streamingContext.sparkContext.broadcast(Map[String, String]())

//配置更新广播变量数据间隔的时间
val updateInterval = 10000L
//用于存储最后一次更新的时候
var lastUpdateTime = 0L
//声明两个累加器，用于每批次数据的累加统计，而不是历史上所有的批次，所以这两个累加器在每批次使用完成之后要进行重置
val matchAcc: LongAccumulator = streamingContext.sparkContext.longAccumulator
val noMatchAcc: LongAccumulator = streamingContext.sparkContext.longAccumulator

words.foreachRDD(r => {
    //第一次启动的时候先进行广播变量的更新操作，以后的每次更新操作是根据时间的间隔来进行的。这里的代码是在 Driver 端进行广播变量重建的
    if (mapBroadCast.value.isEmpty || System.currentTimeMillis() - lastUpdateTime >= updateInterval) {
        val map: mutable.Map[String, String] = Map[String, String]()
        val dictFilePath = "/Users/leohe/Data/input/updateSparkBroadCast" //字典文件路径
        val fs: FileSystem = FileSystem.get(new Configuration())
        val fileStatuses: Array[FileStatus] = fs.listStatus(new Path(dictFilePath)) //字典文件路径下的所有文件
        for (f <- fileStatuses) {
            val filePath: Path = f.getPath // 获取文件路径
            val stream: FSDataInputStream = fs.open(filePath) // 打开
            val reader = new BufferedReader(new InputStreamReader(stream)) // 包装缓冲的读取方式
            var line: String = reader.readLine() //读
            //数据处理，存到 map 中
            while (line != null) {
                val strings: Array[String] = line.split("\t")
                val code: String = strings(0)

```

```

    val countryName: String = strings(1)
    map += code -> countryName
    line = reader.readLine()
}
}

//手动取消广播变量的持久化
mapBroadCast.unpersist()
//重新创建广播变量
mapBroadCast = streamingContext.sparkContext.broadcast(map)
//修改最后一次的更新时间
lastUpdateTime = System.currentTimeMillis()
}

println(s"broadCast:${mapBroadCast.value}")

if(!r.isEmpty()){
    r.foreachPartition(it => {
        //这里的代码是属于 RDD 算子中的 function 的，所以函数中的代码是在集群运行的
        val cast: mutable.Map[String, String] = mapBroadCast.value
        it.foreach(f => {
            println(s"kafka:${f}")

            //scala 中 continue 的写法
            import scala.util.control.Breaks._
            breakable {
                if(f == null){
                    break()
                }
                if(cast.contains(f)){
                    matchAcc.add(1L)
                }else{
                    noMatchAcc.add(1L)
                }
            }
        })
    })
}

//统计每个批次的累加器结果
val matchA: Long = matchAcc.count
val noMatchA: Long = noMatchAcc.count

//这里是使用每个批次累加器的值，比如可以保存到 mysql 中，用于流式计算的统计，最后可以实时的进行报表的展示
println(s"match:${matchA},noMatch:${noMatchA}")

//累加器清 0
matchAcc.reset()
noMatchAcc.reset()
}
}

streamingContext.start()
streamingContext.awaitTermination()
}
}

```

3.6 offset 管理

3.6.1 保存 offset 到 ZK

代码需求：

- 1、offset 保存到 zookeeper
- 2、不使用 DStream 的 transform 等其他算子
- 3、将 DStream 数据处理方式转成 Spark-core 数据处理方式
- 4、SparkStreaming 程序长时间中断，再次消费 kafka 的时候数据过期
- 5、处理上次消费记录的 offset 丢失了的问题

代码实现：

```
import kafka.api.PartitionOffsetRequestInfo
import kafka.common.TopicAndPartition
import kafka.javaapi.consumer.SimpleConsumer
import kafka.javaapi.{OffsetRequest, PartitionMetadata, TopicMetadataRequest, TopicMetadataResponse}
import kafka.utils.{ZKGroupTopicDirs, ZkUtils}
import org.I0Itec.zkclient.ZkClient
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.TopicPartition
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.dstream.InputDStream
import org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe
import org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent
import org.apache.spark.streaming.kafka010.{HasOffsetRanges, KafkaUtils, OffsetRange}
import org.apache.spark.streaming.{Durations, StreamingContext}

import scala.collection.mutable
import scala.collection.mutable.HashMap

/**
 * 偏移量保存到 zk 中
 * 不使用 DStream 的 transform 等其它算子
 * 将 DStream 数据处理方式转成纯正的 spark-core 的数据处理方式
 * 由于 SparkStreaming 程序长时间中断，再次消费时 kafka 中数据已过时，
 * 上次记录消费的 offset 已丢失的问题处理
 */
object SparkStreamingKafkaOffsetZKRecovery {
    def main(args: Array[String]): Unit = {
        //创建 SparkConf
        val conf = new SparkConf().setAppName("SparkStreamingKafkaOffsetZK").setMaster("local[49]")
        //创建 SparkStreaming，设置批时间
        val ssc = new StreamingContext(conf, Durations.seconds(5))
        //指定 topic
        val topic = "hainiu_html"
        //指定 consumer
        val group = "qingniu23"

        //指定 kafka 的 broker 地址，SparkStream 的 Task 直连到 kafka 的分区上，用底层的 API 消费，效率更高
        val brokerList = "s1.hadoop:9092, s2.hadoop:9092, s3.hadoop:9092, s4.hadoop:9092, s5.hadoop:9092"

        //指定 zk 的地址，更新消费的偏移量时使用（也可以使用 Redis 和 MySQL 来记录偏移量）
        val zkQuorum = "nn1.hadoop:2181, nn2.hadoop:2181, s1.hadoop:2181"

        //SparkStreaming 时使用的 topics 集合，可同时消费多个 topic
    }
}
```

```

val topics: Set[String] = Set(topic)

//topic 在 zk 里的数据路径, 用于保存偏移量
val topicDirs = new ZKGroupTopicDirs(group, topic)

//得到 zk 中的数据路径 例如: "/consumers/${group}/offsets/${topic}"
val zkTopicPath = s"${topicDirs.consumerOffsetDir}"

//kafka 参数
//earliest: 当各分区下有已提交的 offset 时, 从提交的 offset 开始消费; 无提交的 offset 时, 从头开始消费。
//latest: 当各分区下有已提交的 offset 时, 从提交的 offset 开始消费; 无提交的 offset 时消费该分区下新产生的数据。
//none: topic 各分区都存在已提交的 offset 时, 从 offset 后开始消费; 只要有一个分区不存在已提交的 offset, 则抛出异常
val kafkaParams = Map(
    "bootstrap.servers" -> brokerList,
    "group.id" -> group,
    "key.deserializer" -> classOf[StringDeserializer],
    "value.deserializer" -> classOf[StringDeserializer],
    "enable.auto.commit" -> (false: java.lang.Boolean),
    "auto.offset.reset" -> "latest"
)

//定义一个空的 kafkaStream, 之后根据是否有关历史的偏移量进行选择
var kafkaStream: InputDStream[ConsumerRecord[String, String]] = null

//如果存在历史的偏移量, 那使用 fromOffsets 来存放存储在 zk 中的每个 TopicPartition 对应的 offset
var fromOffsets = new HashMap[TopicPartition, Long]

//创建 zk 客户端, 可以从 zk 中读取偏移量数据, 并更新偏移量
val zkClient = new ZkClient(zkQuorum)

//从 zk 中查询该数据路径下是否有每个 partition 的 offset, 这个 offset 是我们自己根据每个 topic 的不同 partition 生成的, 数据路径例子: /consumers/${group}/offsets/${topic}/${partitionId}/${offset}"
//zkTopicPath = /consumers/qingniu/offsets/hainiu_qingniu/
val children = zkClient.countChildren(zkTopicPath)

//判断 zk 中是否保存过历史的 offset
if (children > 0) {
    for (i <- 0 until children) {
        // /consumers/qingniu/offsets/hainiu_qingniu/0
        val partitionOffset = zkClient.readData[String](s"$zkTopicPath/${i}")
        // hainiu_qingniu/0
        val tp = new TopicPartition(topic, i)
        //将每个 partition 对应的 offset 保存到 fromOffsets 中
        // hainiu_qingniu/0 -> 888
        fromOffsets += tp -> partitionOffset.toLong
    }
}

//用于解决 SparkStreaming 程序长时间中断, 再次消费时已记录的 offset 丢失导致程序启动报错问题
import scala.collection.mutable.Map
//存储 kafka 集群中每个 partition 当前最早的 offset
var clusterEarliestOffsets = Map[Long, Long]()
val consumer: SimpleConsumer = new SimpleConsumer("s1.hadoop", 9092, 100000, 64 * 1024, "leaderLookup" + System.currentTimeMillis())

```

```

//使用隐式转换进行 java 和 scala 的类型的互相转换
import scala.collection.convert.WrapAll._

val request: TopicMetadataRequest = new TopicMetadataRequest(topics.toList)
val response: TopicMetadataResponse = consumer.send(request)
consumer.close()

val metadata: mutable.Buffer[PartitionMetadata] = response.topicsMetadata.flatMap(f => f.partitionsMetadata)

//从 kafka 集群中得到当前每个 partition 最早的 offset 值
metadata.map(f => {
    val partitionId: Int = f.partitionId
    val leaderHost: String = f.leader.host
    val leaderPort: Int = f.leader.port
    val clientName: String = "Client_" + topic + "_" + partitionId
    val consumer: SimpleConsumer = new SimpleConsumer(leaderHost, leaderPort, 100000, 64 * 1024, clientName)
    val topicAndPartition = new TopicAndPartition(topic, partitionId)
    var requestInfo = new HashMap[TopicAndPartition, PartitionOffsetRequestInfo]()
    requestInfo.put(topicAndPartition, new PartitionOffsetRequestInfo(kafka.api.OffsetRequest.EarliestTime, 1))
    val request = new OffsetRequest(requestInfo, kafka.api.OffsetRequest.CurrentVersion, clientName)
    val response = consumer.getOffsetsBefore(request)
    val offsets: Array[Long] = response.offsets(topic, partitionId)
    consumer.close()
    clusterEarliestOffsets += ((partitionId, offsets(0)))
}
)

//和历史的进行对比
val nowOffset: mutable.HashMap[TopicPartition, Long] = fromOffsets.map(owner => {
    val clusterEarliestOffset = clusterEarliestOffsets(owner._1.partition())
    if (owner._2 >= clusterEarliestOffset) {
        owner
    } else {
        (owner._1, clusterEarliestOffset)
    }
})

//通过 KafkaUtils 创建直连的 DStream，并使用 fromOffsets 中存储的历史偏移量来继续消费数据
kafkaStream = KafkaUtils.createDirectStream[String, String](ssc, PreferConsistent, Subscribe[String, String](topics, kafkaParams, nowOffset))
} else {
    //如果 zk 中没有该 topic 的历史 offset，那就根据 kafkaParam 的配置用最新(latest)或最旧的(earliest)的 offset
    kafkaStream = KafkaUtils.createDirectStream[String, String](ssc, PreferConsistent, Subscribe[String, String](topics, kafkaParams))
}

//通过 rdd 转换得到偏移量的范围
var offsetRanges = Array[OffsetRange]()

//迭代 DStream 中的 RDD，将每一个时间间隔对应的 RDD 拿出来，这个方法是在 driver 端执行
//在 foreachRDD 方法中就跟开发 spark-core 是同样的流程了，当然也可以使用 spark-sql
kafkaStream.foreachRDD(kafkaRDD => {
    if (!kafkaRDD.isEmpty()) {
        //得到该 RDD 对应 kafka 消息的 offset, 该 RDD 是一个 KafkaRDD，所以可以获得偏移量的范围，不使用 transform 可以直接在 foreachRDD 中得到这个 RDD 的偏移量，这种方法适用于 DStream 不经过任何的转换，直接进行 foreachRDD，因为如果 transformation 了那就不是 KafkaRDD 了，就不能强转成 HasOffsetRanges 了，从而就得不
    }
})

```

到 kafka 的偏移量了。

```
offsetRanges = kafkaRDD.asInstanceOf[HasOffsetRanges].offsetRanges
val dataRDD: RDD[String] = kafkaRDD.map(_.value())

//执行这个 rdd 的 action, 这里 rdd 的算子是在集群上执行的
dataRDD.foreachPartition(partition =>
    partition.foreach(x => {
        println(x)
    })
)

for (o <- offsetRanges) {
    // /consumers/qingniu/offsets/hainiu_qingniu/0
    val zkPath = s"${topicDirs.consumerOffsetDir}/${o.partition}"
    //将该 partition 的 offset 保存到 zookeeper
    // /consumers/qingniu/offsets/hainiu_qingniu/888
    // println(s"${zkPath}__${o.untilOffset.toString}")
    ZkUtils(zkClient, false).updatePersistentPath(zkPath, o.untilOffset.toString)
}
}

ssc.start()
ssc.awaitTermination()

}

}
```

3.6.2 手动更新 offset

```
package main.Kafka.SparkStreamingKafkaOffset

import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.SparkConf
import org.apache.spark.streaming.DStream
import org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe
import org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent
import org.apache.spark.streaming.kafka010._
import org.apache.spark.streaming.{Seconds, StreamingContext}

/**
 * 手动更新偏移量
 */
object SparkStreamingKafkaOffsetNotAutoCommit {
    def main(args: Array[String]): Unit = {

        val group = "qingniu8888888"
        val topic = "hainiu_test"
        val conf = new SparkConf().setAppName("sparkstreamingkafkaoffset").setMaster("local[*]")
        val streamingContext = new StreamingContext(conf, Seconds(5))

        //可以通过 streamingContext 中的 sparkContext 来设置日志级别或者其他参数
        //streamingContext.sparkContext.setLogLevel("info")

        //kafka 的参数
        val kafkaParams = Map[String, Object](
            "bootstrap.servers" -> "s1.hadoop:9092,s3.hadoop:9092,s4.hadoop:9092,s5.hadoop:9092",
            "socket.timeout.ms" -> "5000",
            "max.poll.records" -> "1000"
        )
    }
}
```

```

"key.deserializer" -> classOf[StringDeserializer],
"value.deserializer" -> classOf[StringDeserializer],
"group.id" -> group,
//earliest 当各分区下有已提交的 offset 时，从提交的 offset 开始消费；无提交的 offset 时，从头开始消费
//latest 当各分区下有已提交的 offset 时，从提交的 offset 开始消费；无提交的 offset 时，消费新产生的该
分区下的数据
//none topic 各分区都存在已提交的 offset 时，从 offset 后开始消费；只要有一个分区不存在已提交的 off
set，则抛出异常
"auto.offset.reset" -> "earliest",
//修改为手动提交偏移量
"enable.auto.commit" -> (false: java.lang.Boolean)
)

val topics = Array(topic)
//这种方式是在 Kafka 中记录读取偏移量
val stream = KafkaUtils.createDirectStream[String, String](
  streamingContext,
  //位置策略
  PreferConsistent,
  //订阅的策略
  Subscribe[String, String](topics, kafkaParams)
)

//迭代 DStream 中的 RDD，将每一个时间间隔对应的 RDD 拿出来，这个方法是在 driver 端执行
//在 foreachRDD 方法中就跟开发 spark-core 是同样的流程了，当然也可以使用 spark-sql
stream.foreachRDD { rdd =>
  //获取该 RDD 对应的偏移量，记住只有 kafka 的 rdd 才能强转成 HasOffsetRanges 类型
  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
  //执行这个 rdd 的 action，这里 rdd 的算子是在集群上执行的
  rdd.foreach { line =>
    println(line.key() + " " + line.value())
  }
}

//foreach 和 foreachPartition 的区别
//foreachPartition 不管有没有数据都会执行自己的 function
//foreach 只在有数据时执行自己的 function
//      rdd.foreachPartition(it =>{
//        val list: List[ConsumerRecord[String, String]] = it.toList
//        println(list)
//      })

for (o <- offsetRanges) {
  val zkPath = s"partitioner:${o.partition}_offset:${o.untilOffset.toString}||"
  print(zkPath)
}

println()

//更新偏移量
//在上面所有的流程走完之后，再更新偏移量，为什么放到后面？因为如果上面抛异常了这里就不用更新了
//这里如果不提交，那在下次启动的时候 DirectStream 将在原来的位置重新处理
stream.asInstanceOf[CanCommitOffsets].commitAsync(offsetRanges)
}

streamingContext.start()
streamingContext.awaitTermination()

```

```
}
```

3.7 从 checkpoint 恢复数据

```
import org.apache.spark.SparkConf
import org.apache.spark.streaming.dstream.{DStream, ReceiverInputDStream}
import org.apache.spark.streaming.{Durations, StreamingContext}

object SparkStreamingSocketPortUpdateState {
    def main(args: Array[String]): Unit = {
        val checkPoint = "/Users/leohe/Data/output/sparkstreamingsocketportupdatestat"

        val createFunction: () => StreamingContext = () => {
            val conf: SparkConf = new SparkConf().setAppName("SparkStreamingSocketPort").setMaster("local[2]")
            val streamingContext = new StreamingContext(conf, Durations.seconds(5))

            //使用 updateStateByKey 必须得设置一个 checkPoint 地址用于保存历史的数据
            streamingContext.checkpoint(checkPoint)

            //在有 checkpoint 数据的时候，修改地址和端口是不生效的
            val lines: ReceiverInputDStream[String] = streamingContext.socketTextStream("op.hadoop", 6666)

            val flatMap: DStream[String] = lines.flatMap(_.split(" "))
            val mapToPair: DStream[(String, Int)] = flatMap.map((_, 1))
            val reduceByKey: DStream[(String, Int)] = mapToPair.reduceByKey(_ + _)

            val updateStateByKey: DStream[(String, Int)] = reduceByKey.updateStateByKey(
                //这个参数 a 是本批次的数据，b 是这个 key 上次的历史结果
                (a: Seq[Int], b: Option[Int]) => {
                    var total = 0
                    for (i <- a) {
                        total += i
                    }
                    //这里为什么要判断一下，因为这个 key 有可能是第一次进入，也就是说没有历史数据，那此时应该给个初始值 0
                    val last = if (b.isDefined) b.get else 0
                    val now = last + total
                    Some(now)
                })
            }

            //在有 checkpoint 数据的时候，修改算子中的 function 是有效的，所以使用了 checkpoint 恢复 streamingContext 的程序，这样的程序并不影响你后续的升级代码
            updateStateByKey.foreachRDD((r, t) => {
                //可以用 rdd 的 isEmpty 方法，判断此批次是否有数据，如果有数据再执行
                if (!r.isEmpty()) {
                    println(s"count time:${t}, ${r.collect().toList}, ha ha ha")
                }
            })
            streamingContext
        }
    }

    //使用 getOrCreate 从 checkPoint 里恢复最后一次的 StreamingContext 状态，如果没有 checkPoint 地址，那就新建一个 StreamingContext
    val strc: StreamingContext = StreamingContext.getOrCreate(checkPoint, createFunction)
```

```

//这里必须得使用 getOrCreate 判断好（要么是历史的，要么是新 create 的）的那个 streamingContext 进行启动
strc.start()
strc.awaitTermination()
}
}

```

3.8 多个 receiver 源 union

```

import org.apache.spark.SparkConf
import org.apache.spark.streaming.dstream.{DStream, ReceiverInputDStream}
import org.apache.spark.streaming.{Durations, StreamingContext}
import scala.collection.mutable.ListBuffer

object SparkStreamingMuiReceiver {
    def main(args: Array[String]): Unit = {
        //这个地方设置 local[3]，为什么呢？因为这个程序里面有两个 socket 的 receiver，每个 receiver 都占用了一个 cpu，所以设置 cpu 必须大于 2，不然没有可以负责任务运行的 cpu 了
        val conf: SparkConf = new SparkConf().setAppName("SparkStreamingFile").setMaster("local[3]")
        val streamingContext = new StreamingContext(conf, Durations.seconds(5))

        //生成两个 socket 流
        val lines1: ReceiverInputDStream[String] = streamingContext.socketTextStream("op.hadoop", 6666)
        val lines2: ReceiverInputDStream[String] = streamingContext.socketTextStream("op.hadoop", 6667)

        //合拼两个 socket 流
        val bu = new ListBuffer[DStream[String]]
        bu += lines1
        bu += lines2
        val unionStream: DStream[String] = streamingContext.union(bu)

        //合拼了的两个流使用了统一的业务处理逻辑
        unionStream.foreachRDD(r => {
            r foreach println
        })

        streamingContext.start()
        streamingContext.awaitTermination()
    }
}

```

第 4 章 DStream 转换

DStream 上的原语与 RDD 的类似，分为 Transformations（转换）和 Output Operations（输出）两种，此外转换操作中还有一些比较特殊的原语，如：updateStateByKey()、transform()以及各种 Window 相关的原语。

4.1 无状态转化操作

无状态转化操作就是把简单的 RDD 转化操作应用到每个批次上，也就是转化 DStream 中的每一个 RDD。部分无状态转化操作列在了下表中。注意，针对键值对的 DStream 转化操作(比如 reduceByKey())要添加 import StreamingContext._ 才能在 Scala 中使用。

函数名称	目的	Scala示例	用来操作DStream[T]的用户自定义函数的函数签名
map()	对 DStream 中的每个元素应用给定函数，返回由各元素输出的元素组成的 DStream。	ds.map(x => x + 1)	f: (T) -> U
flatMap()	对 DStream 中的每个元素应用给定函数，返回由各元素输出的迭代器组成的 DStream。	ds.flatMap(x => x.split(" "))	f: T -> Iterable[U]
filter()	返回由给定 DStream 中通过筛选的元素组成的 DStream。	ds.filter(x => x != 1)	f: T -> Boolean
repartition()	改变 DStream 的分区数。	ds.repartition(10)	N/A
reduceByKey()	将每个批次中键相同的记录归约。	ds.reduceByKey((x, y) => x + y)	f: T, T -> T
groupByKey()	将每个批次中的记录根据键分组。	ds.groupByKey()	N/A

需要记住的是，尽管这些函数看起来像作用在整个流上一样，但事实上每个 DStream 在内部是由许多 RDD(批次)组成，且无状态转化操作是分别应用到每个 RDD 上的。例如，reduceByKey()会归约每个时间区间中的数据，但不会归约不同区间之间的数据。

举个例子，在之前的 wordcount 程序中，我们只会统计 5 秒内接收到的数据的单词个数，而不会累加。

无状态转化操作也能在多个 DStream 间整合数据，不过也是在各个时间区间内。例如，键值对 DStream 拥有和 RDD 一样的与连接相关的转化操作，也就是 cogroup()、join()、leftOuterJoin() 等。我们可以在 DStream 上使用这些操作，这样就对每个批次分别执行了对应的 RDD 操作。

我们还可以像在常规的 Spark 中一样使用 DStream 的 union() 操作将它和另一个 DStream 的内容合并起来，也可以使用 StreamingContext.union() 来合并多个流。

4.2 有状态转化操作（重点）

4.2.1 UpdateStateByKey

UpdateStateByKey 原语用于记录历史记录，有时我们需要在 DStream 中跨批次维护状态（例如流计算中累加 wordcount）。针对这种情况，updateStateByKey() 为我们提供了对一个状态变量的访问，用于键值对形式的 DStream。给定一个由（键，事件）对构成的 DStream，并传递一个指定如何根据新的事件更新每个键对应状态的函数，它可以构建出一个新的 DStream，其内部数据为（键，状态）对。

updateStateByKey 的结果会是一个新的 DStream，其内部的 RDD 序列是由每个时间区间对应的(键，状态)对组成的。

updateStateByKey 操作使得我们可以在用新信息进行更新时保持任意的状态。为使用这个功能，你需要做下面两步：

1. 定义状态，状态可以是一个任意的数据类型。
2. 定义状态更新函数，用此函数阐明如何使用之前的状态和如何对来自输入流的新值对状态进行更新。

使用 updateStateByKey 需要对检查点目录进行配置，会使用检查点来保存状态。

更新版的 wordcount：

(1) 编写代码

```
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}

object WordCount {
    def main(args: Array[String]) {
        // 定义更新状态方法，参数 values 为当前批次单词频度，state 为以往批次单词频度
        val updateFunc = (values: Seq[Int], state: Option[Int]) => {
            val currentCount = values.foldLeft(0)(_ + _)
            val previousCount = state.getOrElse(0)
            Some(currentCount + previousCount) // Some 表示一定有值，没值会报错
        }
    }
}
```

```

}

val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(3))
ssc.checkpoint("hdfs://hadoop102:9000/streamCheck")

// 创建一个 socket 流
val lines = ssc.socketTextStream("hadoop102", 9999)

// 数据压平
val words = lines.flatMap(_.split(" "))

//import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3
val pairs = words.map(_, 1))

// 使用 updateStateByKey 来更新状态，统计从运行开始以来单词总的次数
val stateDstream = pairs.updateStateByKey[Int](updateFunc)
stateDstream.print()

//val wordCounts = pairs.reduceByKey(_ + _)
//Print the first ten elements of each RDD generated in this DStream to the console
//wordCounts.print()

ssc.start()          // 开始计算
ssc.awaitTermination() // 等待计算结束
//ssc.stop()
}
}

```

(2) 启动程序并向 9999 端口发送数据

```
[atguigu@hadoop102 kafka]$ nc -lk 9999
ni shi shui
ni hao ma
```

(3) 结果展示

```
Time: 1504685175000 ms
```

```
Time: 1504685181000 ms
```

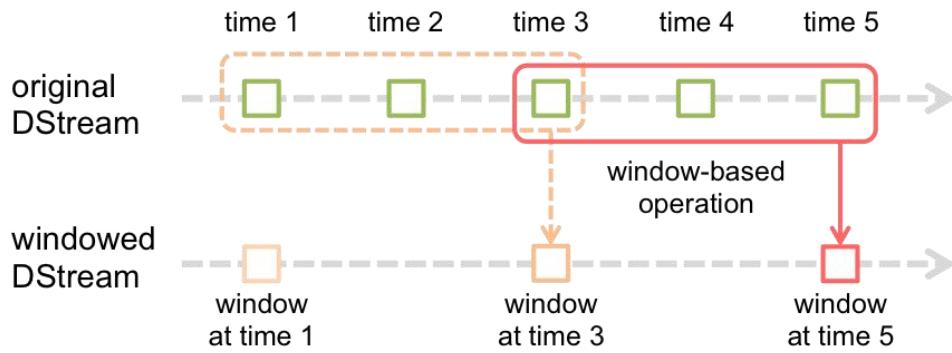
```
(shi,1)
(shui,1)
(ni,1)
```

```
Time: 1504685187000 ms
```

```
(shi,1)
(ma,1)
(hao,1)
(shui,1)
(ni,2)
```

4.2.2 Window Operations

Window Operations 可以设置窗口的大小和滑动窗口的间隔来动态的获取当前 Streaming 的允许状态。基于窗口的操作会在一个比 StreamingContext 的批次间隔更长的时间范围内，通过整合多个批次的结果，计算出整个窗口的结果。

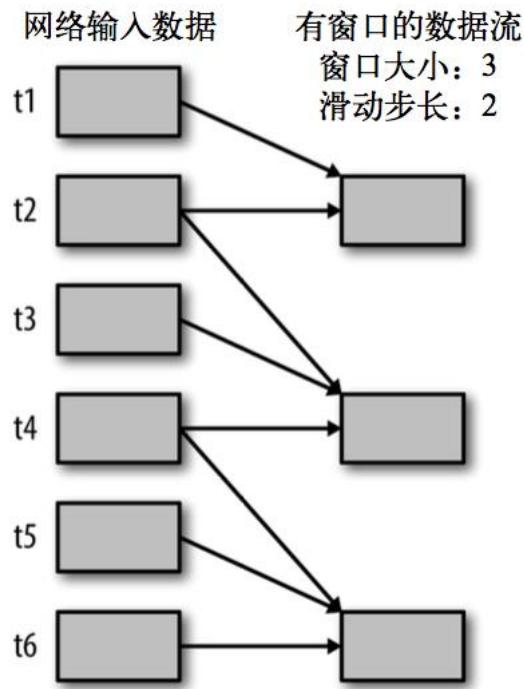


注意：所有基于窗口的操作都需要两个参数，分别为窗口时长以及滑动步长，两者都必须是 StreamContext 的批次间隔的整数倍。

窗口时长控制每次计算最近的多少个批次的数据，其实就是最近的 windowDuration/batchInterval 个批次。如果有一个以 10 秒为批次间隔的源 DStream，要创建一个最近 30 秒的时间窗口(即最近 3 个批次)，就应当把 window Duration 设为 30 秒。而滑动步长的默认值与批次间隔相等，用来控制对新的 DStream 进行计算的间隔。如果源 DStream 批次间隔为 10 秒，并且我们只希望每两个批次计算一次窗口结果，就应该把滑动步长设置为 20 秒。

假设，你想拓展前例从而每隔十秒对持续 30 秒的数据生成 word count。为做到这个，我们需要在持续 30 秒数据的(word,1)对 DStream 上应用 reduceByKey。使用操作 reduceByKeyAndWindow。

```
# reduce last 30 seconds of data, every 10 second
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x - y, 30, 20)
```



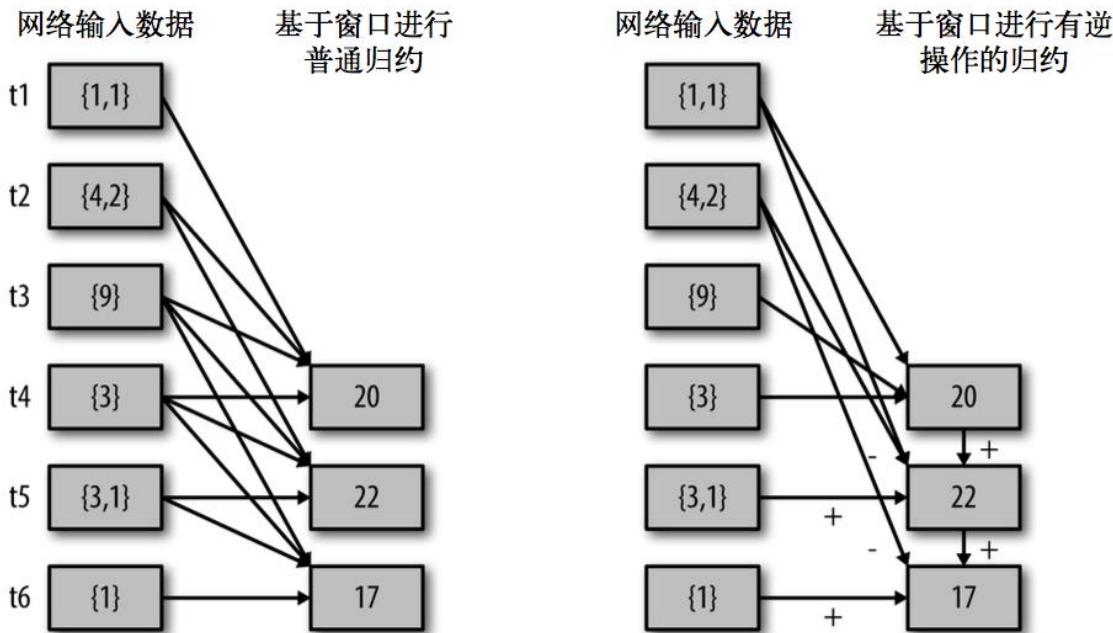
窗口转化操作：

- (1) **window(windowLength, slideInterval)**: 基于对源 DStream 窗化的批次进行计算返回一个新的 Dstream
- (2) **countByWindow(windowLength, slideInterval)**: 返回一个滑动窗口计数流中的元素。
- (3) **reduceByWindow(func, windowLength, slideInterval)**: 通过使用自定义函数整合滑动区间流元素来创建一个新的单元素流。
- (4) **reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])**: 当在一个(K,V)对的 DStream 上调用此函数，会返回一个新(K,V)对的 DStream，此处通过对滑动窗口中批次数据使用 reduce 函数来整合每个 key 的 value 值。Note: 默认情况下，这个操作使用 Spark 的默认数量并行任务(本地是 2)，在集群模式中依据配置属性(spark.default.parallelism)来做 grouping。你可以通过设置可选参数 numTasks 来设置不同数量的 tasks。

(5) **reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])**: 这个函数是上述函数的更高效版本，每个窗口的 reduce 值都是通过用前一个窗的 reduce 值来递增计算。通过 reduce 进入到滑动窗口数据并“反向 reduce”离开窗口的旧数据来实现这个操作。一个例子是随着窗口滑动对 keys 的“加”“减”计数。通过前边介绍可以想到，这个函数只适用于“可逆的 reduce 函数”，也就是这些 reduce 函数有相应的“反 reduce”函数(以参数 invFunc 形式传入)。如前述函数，reduce 任务的数量通过可选参数来配置。注意：为了使用这个操作，检查点必须可用。

(6) **countByValueAndWindow(windowLength,slideInterval, [numTasks])**: 对(K,V)对的 DStream 调用，返回(K,Long)对的新 DStream，其中每个 key 的值是其在滑动窗口中频率。如上，可配置 reduce 任务数量。

`reduceByWindow()`和 `reduceByKeyAndWindow()`让我们可以对每个窗口更高效地进行归约操作。它们接收一个归约函数，在整个窗口上执行，比如 `+`。除此以外，它们还有一种特殊形式，通过只考虑新进入窗口的数据和离开窗口的数据，让 Spark 增量计算归约结果。这种特殊形式需要提供归约函数的一个逆函数，比如 `+` 对应的逆函数为 `-`。对于较大的窗口，提供逆函数可以大大提高执行效率



```
val ipDStream = accessLogsDStream.map(logEntry => (logEntry.getIpAddress(), 1))
val ipCountDStream = ipDStream.reduceByKeyAndWindow(
  {(x, y) => x + y},
  {(x, y) => x - y},
  Seconds(30),
  Seconds(10))
// 加上新进入窗口的批次中的元素 // 移除离开窗口的老批次中的元素 // 窗口时长// 滑动步长
```

`countByWindow()`和 `countByValueAndWindow()`作为对数据进行计数操作的简写。`countByWindow()`返回一个表示每个窗口中元素个数的 DStream，而 `countByValueAndWindow()`返回的 DStream 则包含窗口中每个值的个数。

```
val ipDStream = accessLogsDStream.map{entry => entry.getIpAddress()}
val ipAddressRequestCount = ipDStream.countByValueAndWindow(Seconds(30), Seconds(10))
val requestCount = accessLogsDStream.countByWindow(Seconds(30), Seconds(10))
```

WordCount 第三版：3 秒一个批次，窗口 12 秒，滑步 6 秒。

```
package com.atguigu.streaming

import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}

object WordCount {
  def main(args: Array[String]) {
    // 定义更新状态方法，参数 values 为当前批次单词频度，state 为以往批次单词频度
    val updateFunc = (values: Seq[Int], state: Option[Int]) => {
      val currentState = state.getOrElse(0)
      val currentCount = values.foldLeft(0)(_ + _)
      Some(currentState + currentCount)
    }
  }
}
```

```

    val previousCount = state.getOrElse(0)
    Some(currentCount + previousCount)
}

val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(3)) //3 秒一个批次
ssc.checkpoint(".")

val lines = ssc.socketTextStream("hadoop102", 9999)

val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
//窗口 12 秒， 滑步 6 秒
val wordCounts = pairs.reduceByKeyAndWindow((a,b) => (a + b), Seconds(12), Seconds(6))

// 打印前 10 个结果
wordCounts.print()

ssc.start()           // Start the computation
ssc.awaitTermination() // Wait for the computation to terminate
//ssc.stop()
}
}

```

4.3 其他重要操作

4.3.1 Transform

Transform 原语允许 DStream 上执行任意的 RDD-to-RDD 函数。即使这些函数并没有在 DStream 的 API 中显示出来，通过该函数可以方便的扩展 Spark API。该函数每一批次调度一次。其实也就是对 DStream 中的 RDD 应用转换。

比如下面的例子，在进行单词统计的时候，想要过滤掉 spam 的信息。

```

val spamInfoRDD = spark.sparkContext.newAPIHadoopRDD(...) // 包含脏数据的 RDD

val cleanedDStream = wordCounts.transform { rdd =>
    rdd.join(spamInfoRDD).filter(...) // 文件流与 spamInfoRDD 连接后进行数据过滤
}

```

4.3.2 Join

连接操作（leftOuterJoin, rightOuterJoin, fullOuterJoin 也可以），可以连接 Stream-Stream, windows-stream to windows-stream、stream-dataset

Stream-Stream：普通流与普通流

```

val stream1: DStream[String, String] = ...
val stream2: DStream[String, String] = ...
val joinedStream = stream1.join(stream2)

```

windows-stream to windows-stream：窗口流与窗口流

```

val windowedStream1 = stream1.window(Seconds(20))
val windowedStream2 = stream2.window(Minutes(1))
val joinedStream = windowedStream1.join(windowedStream2)

```

Stream-dataset joins：普通流与数据集

```

val dataset: RDD[String, String] = ...
val windowedStream = stream.window(Seconds(20))...
val joinedStream = windowedStream.transform { rdd => rdd.join(dataset) }

```

第 5 章 DStream 输出

输出操作指定了对流数据经转化操作得到的数据所要执行的操作(例如把结果推入外部数据库或输出到屏幕上)。与 RDD 中的惰性求值类似，如果一个 DStream 及其派生出的 DStream 都没有被执行输出操作，那么这些 DStream 就都不会被求值。如果 StreamingContext 中没有设定输出操作，整个 context 就都不会启动。

输出操作如下：

- (1) **print()**: 在运行流程序的驱动结点上打印 DStream 中每一批次数据的最开始 10 个元素。这用于开发和调试。在 Python API 中，同样的操作叫 print()。
- (2) **saveAsTextFiles(prefix, [suffix])**: 以 text 文件形式存储这个 DStream 的内容。每一批次的存储文件名基于参数中的 prefix 和 suffix。“prefix-Time_IN_MS[.suffix]”。
- (3) **saveAsObjectFiles(prefix, [suffix])**: 以 Java 对象序列化的方式将 Stream 中的数据保存为 SequenceFiles，每一批次的存储文件名基于参数中的为"prefix-TIME_IN_MS[.suffix]"。Python 中目前不可用。
- (4) **saveAsHadoopFiles(prefix, [suffix])**: 将 Stream 中的数据保存为 Hadoop files，每一批次的存储文件名基于参数中的为"prefix-TIME_IN_MS[.suffix]"。Python 中目前不可用。
- (5) **foreachRDD(func)**: 这是最通用的输出操作，即将函数 func 用于产生于 stream 的每一个 RDD。其中参数传入的函数 func 应该实现将每一个 RDD 中数据推送到外部系统，如将 RDD 存入文件或者通过网络将其写入数据库。注意：函数 func 在运行流应用的驱动中被执行，同时其中一般函数 RDD 操作从而强制其对于流 RDD 的运算。

通用的输出操作 foreachRDD(), 它用来对 DStream 中的 RDD 运行任意计算。这和 transform() 有些类似，都可让我们访问任意 RDD。在 foreachRDD() 中，可以重用我们在 Spark 中实现的所有行动操作。

比如，常见的用例之一是把数据写到诸如 MySQL 的外部数据库中。 注意：

- (1) 连接不能写在 driver 层面；
- (2) 如果写在 foreach，则每个 RDD 都创建，得不偿失；
- (3) 增加 foreachPartition，在分区创建。

5.1 输出到 HDFS

5.1.1 小文件问题

1、考虑的问题是可能会产生很多小文件，小文件带来的问题是什么？

- (1) 占用很大的 namenode 的元数据空间，
- (2) 下游使用小文件的 JOB 会产生很多个 partition，如果是 mr 任务就会有很多个 map，如果是 spark 任务就会有很多个 task，

2、如何解决小文件问题？

- (1) 增加 batch 大小
- (2) 使用批处理任务进行小文件的合并
- (3) 使用 HDFS 的 append 方式
- (4) 使用 coalesce 重新设置分区数（此处用于合并分区，减少分区数）

(1) 和 (2) 是不建议使用的

5.1.2 代码实现

注意得使用集群的 core-site.xml 和 hdfs-site.xml

```
import java.text.SimpleDateFormat
import java.util.Date
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.{FSDataOutputStream, FileSystem, Path}
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.{Durations, StreamingContext}
import org.apache.spark.streaming.dstream.{DStream, ReceiverInputDStream}

import scala.collection.mutable.ArrayBuffer
object SparkStreamingSocketPortHDFS {
    def main(args: Array[String]): Unit = {
        val conf: SparkConf = new SparkConf().setAppName("sparkstreamingsocketporthdfs").setMaster("local[2]")
        val streamingContext = new StreamingContext(conf, Durations.seconds(5))
        val lines: ReceiverInputDStream[String] = streamingContext.socketTextStream("op.hadoop", 6666)
        val reduceByKey: DStream[(String, Int)] = lines.flatMap(_.split(" ")).map(_._1).reduceByKey(_ + _)
```

```

reduceByKey.foreachRDD(r => {
    if (!r.isEmpty()) {
        //可以使用 coalesce 减少输出的文件数，但是不建议设置为 1，因为如果为 1 就把并行计算变成单机的了，这里为什么用 mapPartitionsWithIndex 呢？因为可以每个 partition 的 task 拿自己的 partitionId，从而把 partitionId 拼接到输出文件的文件名上，以防止各个 task 之间写文件的冲突
        val value: RDD[String] = r.coalesce(1).mapPartitionsWithIndex((partitionID, f) => {
            val configuration = new Configuration()
            val fs: FileSystem = FileSystem.get(configuration)

            val list: List[(String, Int)] = f.toList
            if (list.length > 0) {
                //这里为什么生成每小时的字符串呢？因为可以拼接到文件名上，用以判断每小时的文件是否存在如果存在就 append，不存在就创建
                val format: String = new SimpleDateFormat("yyyyMMddHH").format(new Date)
                val path = new Path(s"hdfs://ns1/user/qingniu/sparkstreamingsocketporthdfs/${partitionID}_${format}")
            }
        })
    }
}

//存在就是 append 流，不存在就是 create 流，生成 create 流的时候会自动创建文件，append 流只支持集群的 hdfs，不支持 local 模式的 hdfs，所以如果 local 模式的 hdfs，那在后续的 append 的数据就会给你报 not support append
val outputStream: FSDataOutputStream = if (fs.exists(path)) {
    fs.append(path)
} else {
    fs.create(path)
}

list.foreach(ff => {
    outputStream.write(s"${ff._1}\t${ff._2}\n".getBytes("UTF-8"))
})
outputStream.close()
}

//这里为什么用一个 action，因为刚才的 mapPartitionsWithIndex 是个 transformation，如果没有一个 action，咱们的任务将不会执行，而刚才的所有逻辑都写在 mapPartitionsWithIndex 的 function 中，所以必须得写一个 action 用于任务的启动，这样 mapPartitionsWithIndex 的 function 的逻辑才会被执行
new ArrayBuffer[String]().toIterator
})
value.foreach(f => Unit)
}
}
streamingContext.start()
streamingContext.awaitTermination()
}
}

```

Spark 内核

第 1 章 Spark 内核概述

Spark 内核泛指 Spark 的核心运行机制，包括 Spark 核心组件的运行机制、Spark 任务调度机制、Spark 内存管理机制、Spark 核心功能的运行原理等，熟练掌握 Spark 内核原理，能够帮助我们更好地完成 Spark 代码设计，并能够帮助我们准确锁定项目运行过程中出现的问题的症结所在。

1.1 Spark 核心组件回顾

1.1.1 Driver

Spark 驱动器节点，用于执行 Spark 任务中的 main 方法，负责实际代码的执行工作。Driver 在 Spark 作业执行时主要负责：

1. 将用户程序转化为作业（job）；
2. 在 Executor 之间调度任务(task)；
3. 跟踪 Executor 的执行情况；
4. 通过 UI 展示查询运行情况；

1.1.2 Executor

Spark Executor 节点是一个 JVM 进程，负责在 Spark 作业中运行具体任务，任务彼此之间相互独立。Spark 应用启动时，Executor 节点被同时启动，并且始终伴随着整个 Spark 应用的生命周期而存在。如果有 Executor 节点发生了故障或崩溃，Spark 应用也可以继续执行，会将出错节点上的任务调度到其他 Executor 节点上继续运行。

Executor 有两个核心功能：

1. 负责运行组成 Spark 应用的任务，并将结果返回给驱动器进程；
2. 它们通过自身的块管理器（Block Manager）为用户程序中要求缓存的 RDD 提供内存式存储。RDD 是直接缓存在 Executor 进程内的，因此任务可以在运行时充分利用缓存数据加速运算。

1.2 Spark 通用运行流程概述

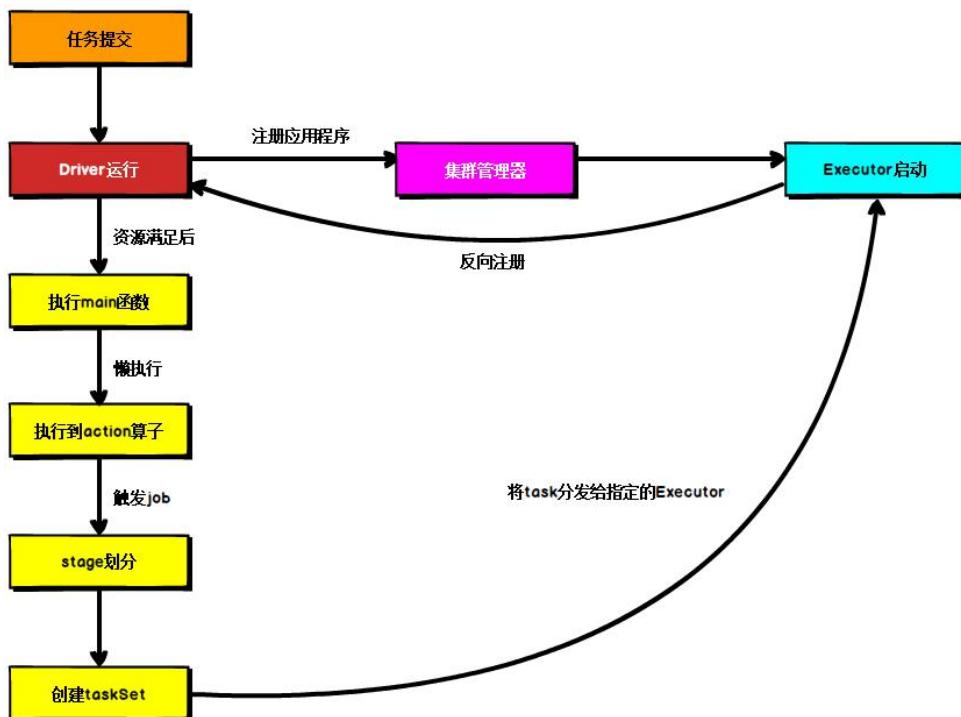


图 1-1 Spark 核心运行流程

图 1-1 为 Spark 通用运行流程，不论 Spark 以何种模式进行部署，任务提交后，都会先启动 Driver 进程，随后 Driver 进程向集群管理器注册应用程序，之后集群管理器根据此任务的配置文件分配 Executor 并启动，当 Driver 所需

的资源全部满足后，Driver 开始执行 main 函数，Spark 查询为懒执行，当执行到 action 算子时开始反向推算，根据宽依赖进行 stage 的划分，随后每一个 stage 对应一个 taskset，taskset 中有多个 task，根据本地化原则，task 会被分发到指定的 Executor 去执行，在任务执行的过程中，Executor 也会不断与 Driver 进行通信，报告任务运行情况。

第 2 章 Spark 部署模式

Spark 支持 3 种集群管理器（Cluster Manager），分别为：

1. **Standalone**: 独立模式，Spark 原生的简单集群管理器，自带完整的服务，可单独部署到一个集群中，无需依赖任何其他资源管理系统，使用 Standalone 可以很方便地搭建一个集群；
2. **Apache Mesos**: 一个强大的分布式资源管理框架，它允许多种不同的框架部署在其上，包括 yarn；
3. **Hadoop YARN**: 统一的资源管理机制，在上面可以运行多套计算框架，如 map reduce、storm 等，根据 driver 在集群中的位置不同，分为 yarn client 和 yarn cluster。

实际上，除了上述这些通用的集群管理器外，Spark 内部也提供了一些方便用户测试和学习的简单集群部署模式。由于在实际工厂环境下使用的绝大多数的集群管理器是 Hadoop YARN，因此我们关注的重点是 Hadoop YARN 模式下的 Spark 集群部署。

Spark 的运行模式取决于传递给 SparkContext 的 MASTER 环境变量的值，个别模式还需要辅助的程序接口来配合使用，目前支持的 Master 字符串及 URL 包括：

表 2-1 Spark 运行模式配置

Master URL	Meaning
local	在本地运行，只有一个工作进程，无并行计算能力。
local[K]	在本地运行，有 K 个工作进程，通常设置 K 为机器的 CPU 核心数量。
local[*]	在本地运行，工作进程数量等于机器的 CPU 核心数量。
spark://HOST:PORT	以 Standalone 模式运行，这是 Spark 自身提供的集群运行模式，默认端口号：7077。 详细文档见：Spark standalone cluster。
mesos://HOST:PORT	在 Mesos 集群上运行，Driver 进程和 Worker 进程运行在 Mesos 集群上，部署模式必须使用固定值：--deploy-mode cluster。详细文档见：MesosClusterDispatcher。
yarn-client	在 Yarn 集群上运行，Driver 进程在本地，Executor 进程在 Yarn 集群上，部署模式必须使用固定值：--deploy-mode client。Yarn 集群地址必须在 HADOOP_CONF_DIR 或 YARN_CONF_DIR 变量里定义。
yarn-cluster	在 Yarn 集群上运行，Driver 进程在 Yarn 集群上，Worker 进程也在 Yarn 集群上，部署模式必须使用固定值：--deploy-mode cluster。Yarn 集群地址必须在 HADOOP_CONF_DIR 或 YARN_CONF_DIR 变量里定义。

用户在提交任务给 Spark 处理时，以下两个参数共同决定了 Spark 的运行方式。

- - master MASTER_URL: 决定了 Spark 任务提交给哪种集群处理。
- - deploy-mode DEPLOY_MODE: 决定了 Driver 的运行方式，可选值为 Client 或者 Cluster。

2.1 Standalone 模式运行机制

Standalone 集群有四个重要组成部分，分别是：

- 1) **Driver**: 是一个进程，我们编写的 Spark 应用程序就运行在 Driver 上，由 Driver 进程执行；
- 2) **Master(RM)**: 是一个进程，主要负责资源的调度和分配，并进行集群的监控等职责；
- 3) **Worker(NM)**: 是一个进程，一个 Worker 运行在集群中的一台服务器上，主要负责两个职责，一个是用自己的内存存储 RDD 的某个或某些 partition；另一个是启动其他进程和线程（Executor），对 RDD 上的 partition 进行并行的处理和计算。
- 4) **Executor**: 是一个进程，一个 Worker 上可以运行多个 Executor，Executor 通过启动多个线程（task）来执行对 RDD 的 partition 进行并行计算，也就是执行我们对 RDD 定义的例如 map、flatMap、reduce 等算子操作。

2.1.1 Standalone Client 模式

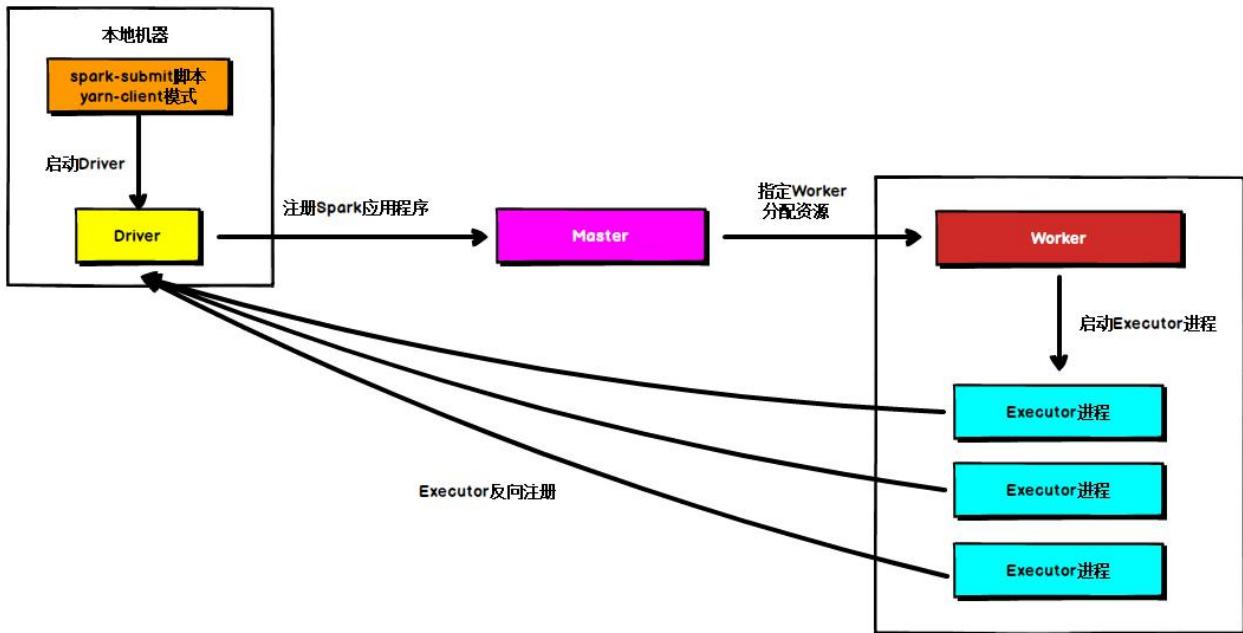


图 2-1 Standalone Client 模式

在 Standalone Client 模式下，Driver 在任务提交的本地机器上运行，Driver 启动后向 Master 注册应用程序，Master 根据 submit 脚本的资源需求找到内部资源至少可以启动一个 Executor 的所有 Worker，然后在这些 Worker 之间分配 Executor，Worker 上的 Executor 启动后会向 Driver 反向注册，所有的 Executor 注册完成后，Driver 开始执行 main 函数，之后执行到 Action 算子时，开始划分 stage，每个 stage 生成对应的 taskSet，之后将 task 分发到各个 Executor 上执行。

2.1.2 Standalone Cluster 模式

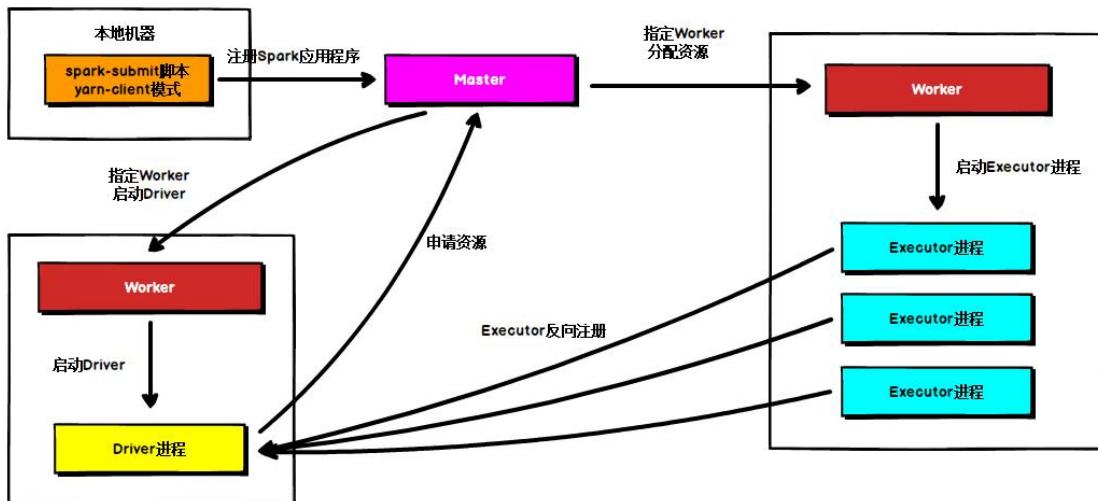


图 2-2 Standalone Cluster 模式

在 Standalone Cluster 模式下，任务提交后，Master 会找到一个 Worker 启动 Driver 进程，Driver 启动后向 Master 注册应用程序，Master 根据 submit 脚本的资源需求找到内部资源至少可以启动一个 Executor 的所有 Worker，然后在这些 Worker 之间分配 Executor，Worker 上的 Executor 启动后会向 Driver 反向注册，所有的 Executor 注册完成后，Driver 开始执行 main 函数，之后执行到 Action 算子时，开始划分 stage，每个 stage 生成对应的 taskSet，之后将 task 分发到各个 Executor 上执行。

注意，Standalone 的两种模式下（client/Cluster），Master 在接到 Driver 注册 Spark 应用程序的请求后，会获取其所管理的剩余资源能够启动一个 Executor 的所有 Worker，然后在这些 Worker 之间分发 Executor，此时的分发只考虑 Worker 上的资源是否足够使用，直到当前应用程序所需的所有 Executor 都分配完毕，Executor 反向注册完毕后，Driver 开始执行 main 程序。

2.2 YARN 模式运行机制

2.2.1 YARN Client 模式

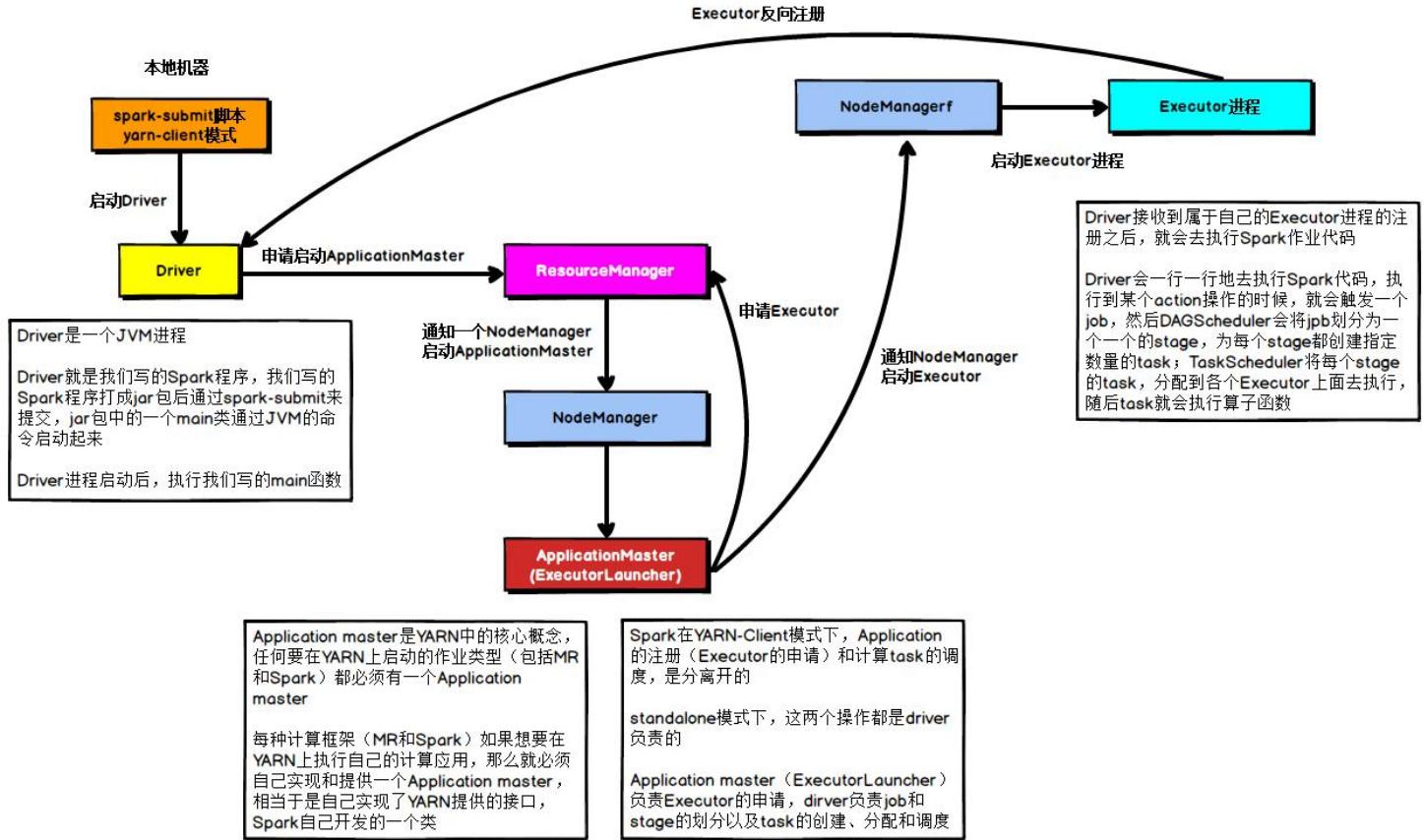
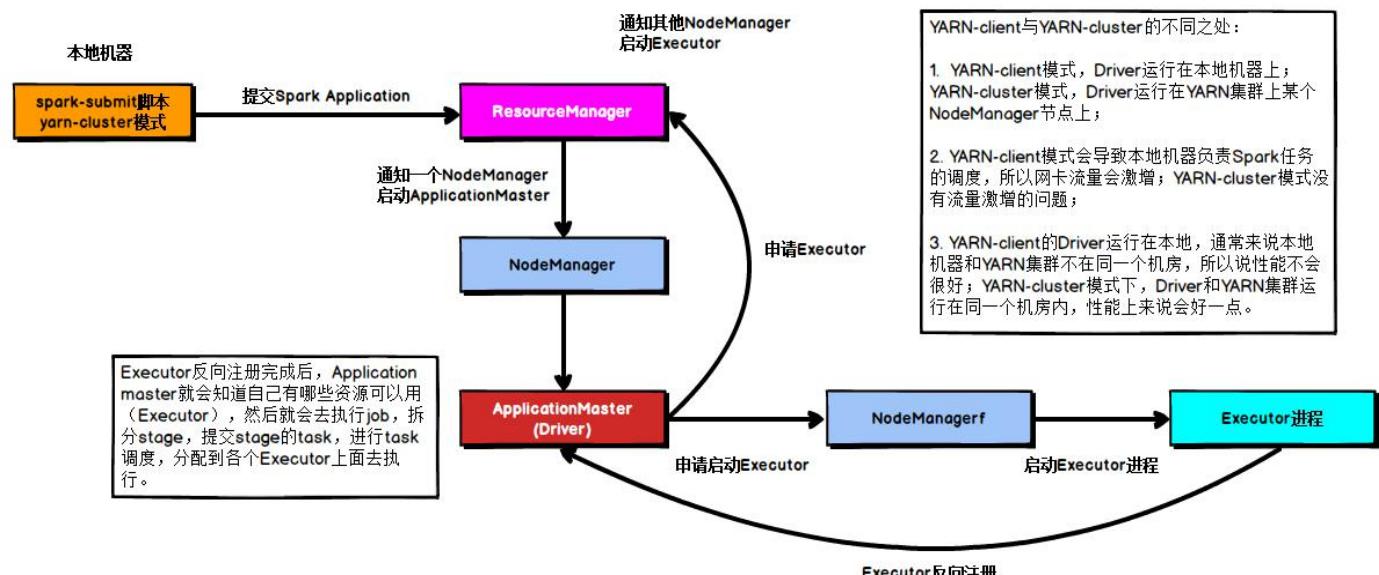


图 2-3 YARN Client 模式

在 YARN Client 模式下，Driver 在任务提交的本地机器上运行，Driver 启动后会和 ResourceManager 通讯申请启动 ApplicationMaster，随后 ResourceManager 分配 container，在合适的 NodeManager 上启动 ApplicationMaster，此时的 ApplicationMaster 的功能相当于一个 ExecutorLaucher，只负责向 ResourceManager 申请 Executor 内存。

ResourceManager 接到 ApplicationMaster 的资源申请后会分配 container，然后 ApplicationMaster 在资源分配指定的 NodeManager 上启动 Executor 进程，Executor 进程启动后会向 Driver 反向注册，Executor 全部注册完成后 Driver 开始执行 main 函数，之后执行到 Action 算子时，触发一个 job，并根据宽依赖开始划分 stage，每个 stage 生成对应的 taskSet，之后将 task 分发到各个 Executor 上执行。

2.2.2 YARN Cluster 模式



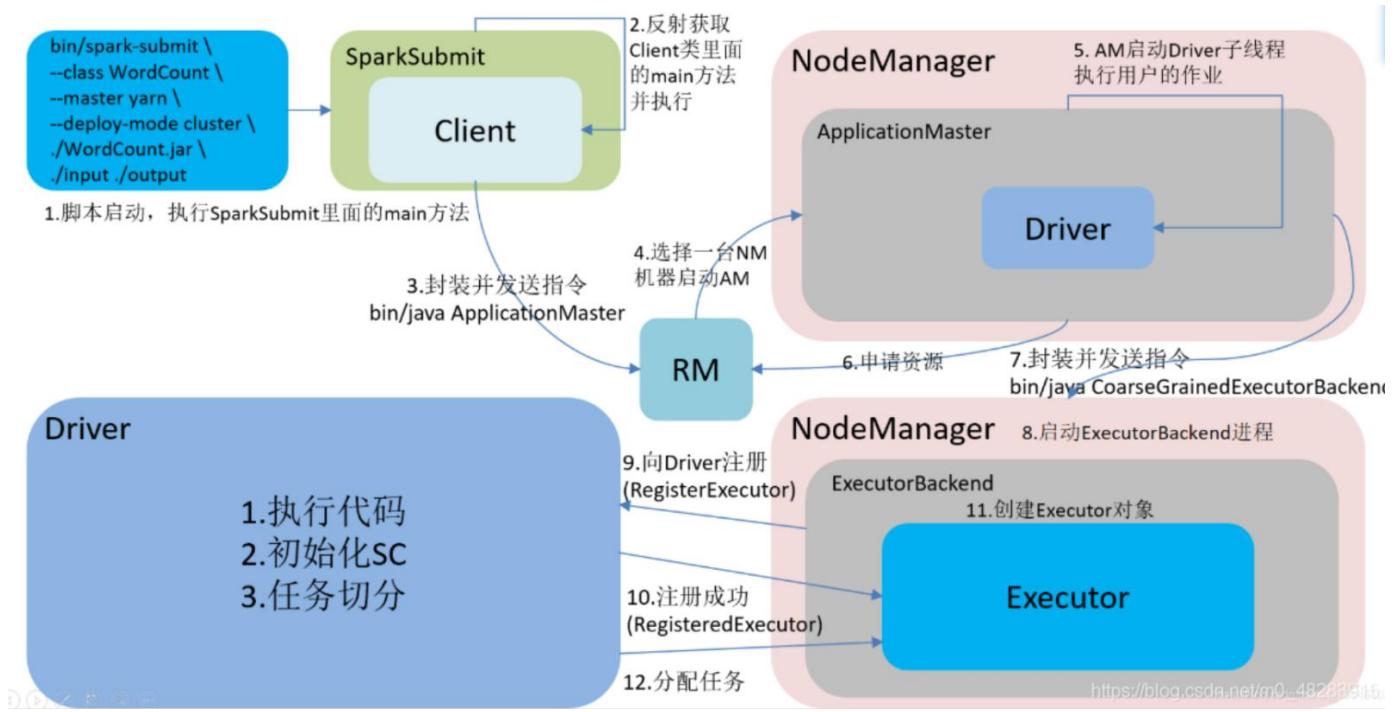


图 2-4 YARN Cluster 模式

在 YARN Cluster 模式下，任务提交后会和 ResourceManager 通讯申请启动 ApplicationMaster，随后 ResourceManager 分配 container，在合适的 NodeManager 上启动 ApplicationMaster，此时的 ApplicationMaster 就是 Driver。

Driver 启动后向 ResourceManager 申请 Executor 内存，ResourceManager 接到 ApplicationMaster 的资源申请后会分配 container，然后在合适的 NodeManager 上启动 Executor 进程，Executor 进程启动后会向 Driver 反向注册，Executor 全部注册完成后 Driver 开始执行 main 函数，之后执行到 Action 算子时，触发一个 job，并根据宽依赖开始划分 stage，每个 stage 生成对应的 taskSet，之后将 task 分发到各个 Executor 上执行。

第 3 章 Spark 通讯架构

3.1 Spark 通信架构概述

Spark2.x 版本使用 Netty 通讯框架作为内部通讯组件。spark 基于 netty 新的 rpc 框架借鉴了 Akka 中的设计，它是基于 Actor 模型，如下图所示：

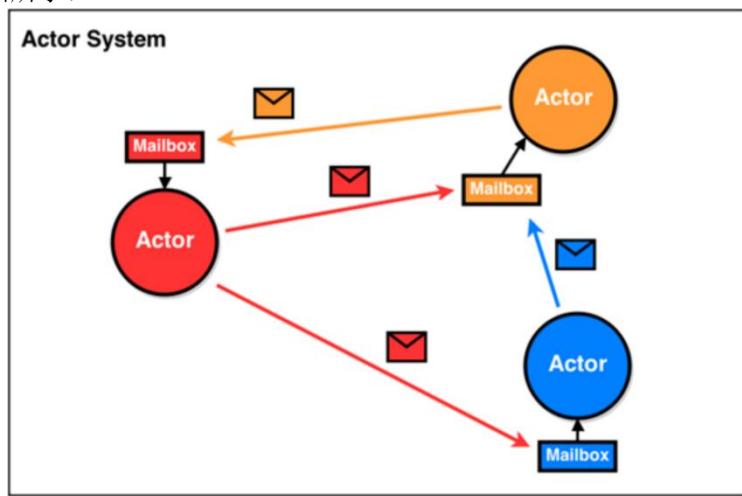


图 4-1 Actor 模型

Spark 通讯框架中各个组件（Client/Master/Worker）可以认为是一个个独立的实体，各个实体之间通过消息来进行通信。具体各个组件之间的关系图如下：

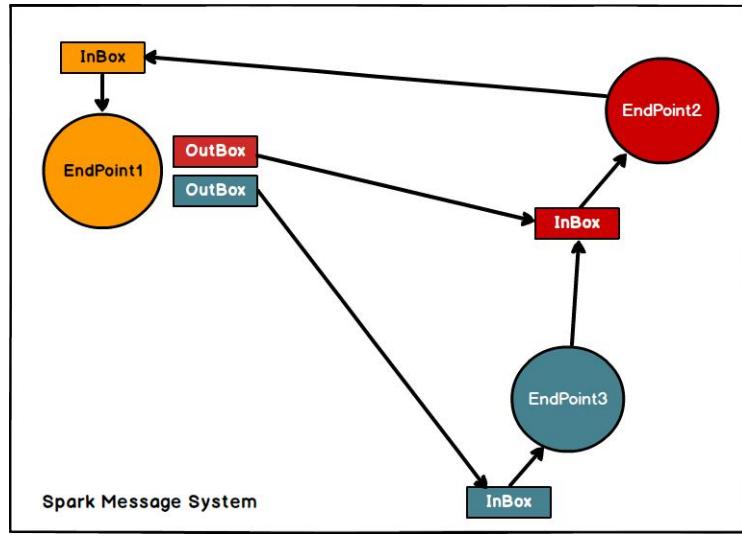


图 4-2 Spark 通讯架构

Endpoint（Client/Master/Worker）有1个InBox和N个OutBox（ $N \geq 1$ ，N取决于当前Endpoint与多少其他的Endpoint进行通信，一个与其通讯的其他Endpoint对应一个OutBox），Endpoint接收到的消息被写入InBox，发送出去的消息写入OutBox并被发送到其他Endpoint的InBox中。

3.2 Spark 通讯架构解析

Spark 通信架构如下图所示：

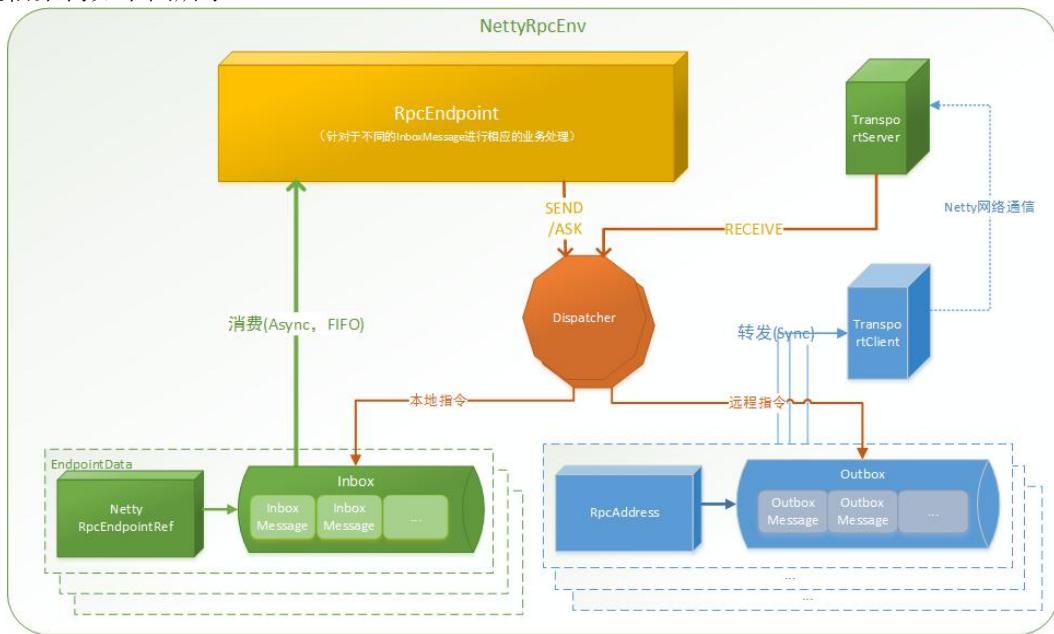


图 4-3 Spark 通讯架构

1) **RpcEndpoint:** RPC 端点，Spark 针对每个节点（Client/Master/Worker）都称之为一个 Rpc 端点，且都实现 RpcEndpoint 接口，内部根据不同端点的需求，设计不同的消息和不同的业务处理，如果需要发送（询问）则调用 Dispatcher；

2) **RpcEnv:** RPC 上下文环境，每个 RPC 端点运行时依赖的上下文环境称为 RpcEnv；

3) **Dispatcher:** 消息分发器，针对 RPC 端点需要发送消息或者从远程 RPC 接收到的消息，分发至对应的指令收件箱/发件箱。如果指令接收方是自己则存入收件箱，如果指令接收方不是自己，则放入发件箱；

4) **Inbox:** 指令消息收件箱，一个本地 RpcEndpoint 对应一个收件箱，Dispatcher 在每次向 Inbox 存入消息时，都将对应 EndpointData 加入内部 ReceiverQueue 中，另外 Dispatcher 创建时会启动一个单独线程进行轮询 Receiver Queue，进行收件箱消息消费；

5) **RpcEndpointRef:** RpcEndpointRef 是对远程 RpcEndpoint 的一个引用。当我们需要向一个具体的 RpcEndpoint 发送消息时，一般我们需要获取到该 RpcEndpoint 的引用，然后通过该应用发送消息。

6) **OutBox**: 指令消息发件箱, 对于当前 **RpcEndpoint** 来说, 一个目标 **RpcEndpoint** 对应一个发件箱, 如果向多个目标 **RpcEndpoint** 发送信息, 则有多个 **OutBox**。当消息放入 **Outbox** 后, 紧接着通过 **TransportClient** 将消息发送出去。消息放入发件箱以及发送过程是在同一个线程中进行;

7) **RpcAddress**: 表示远程的 **RpcEndpointRef** 的地址, **Host + Port**。

8) **TransportClient**: Netty 通信客户端, 一个 **OutBox** 对应一个 **TransportClient**, **TransportClient** 不断轮询 **OutBox**, 根据 **OutBox** 消息的 **receiver** 信息, 请求对应的远程 **TransportServer**;

9) **TransportServer**: Netty 通信服务端, 一个 **RpcEndpoint** 对应一个 **TransportServer**, 接受远程消息后调用 **Dispatcher** 分发消息至对应收发件箱;

根据上面的分析, Spark 通信架构的高层视图如下图所示:

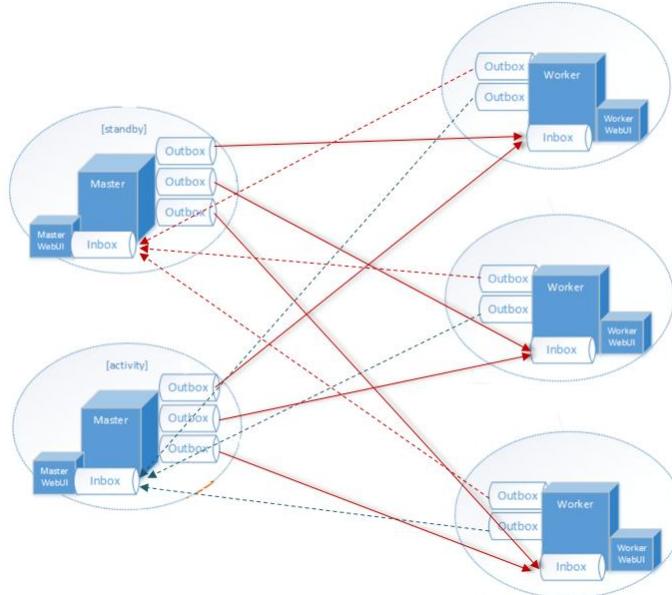


图 4-4 Spark 通信框架高层视图

第 4 章 Spark 任务调度机制

在生产环境下, Spark 集群的部署方式一般为 YARN-Cluster 模式, 之后的内核分析内容中我们默认集群的部署方式为 YARN-Cluster 模式。

4.1 Spark 任务提交流程

在上一章中我们讲解了 Spark YARN-Cluster 模式下的任务提交流程, 如下图所示:

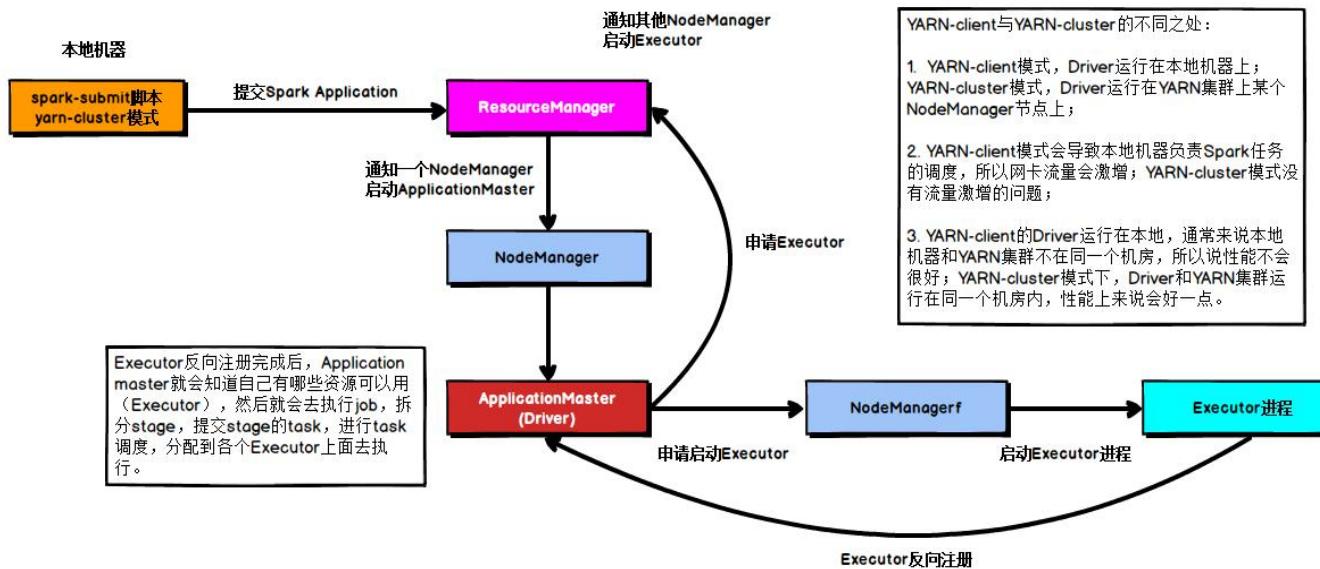


图 4-1 YARN-Cluster 任务提交流程

下面的时序图清晰地说明了一个 Spark 应用程序从提交到运行的完整流程：

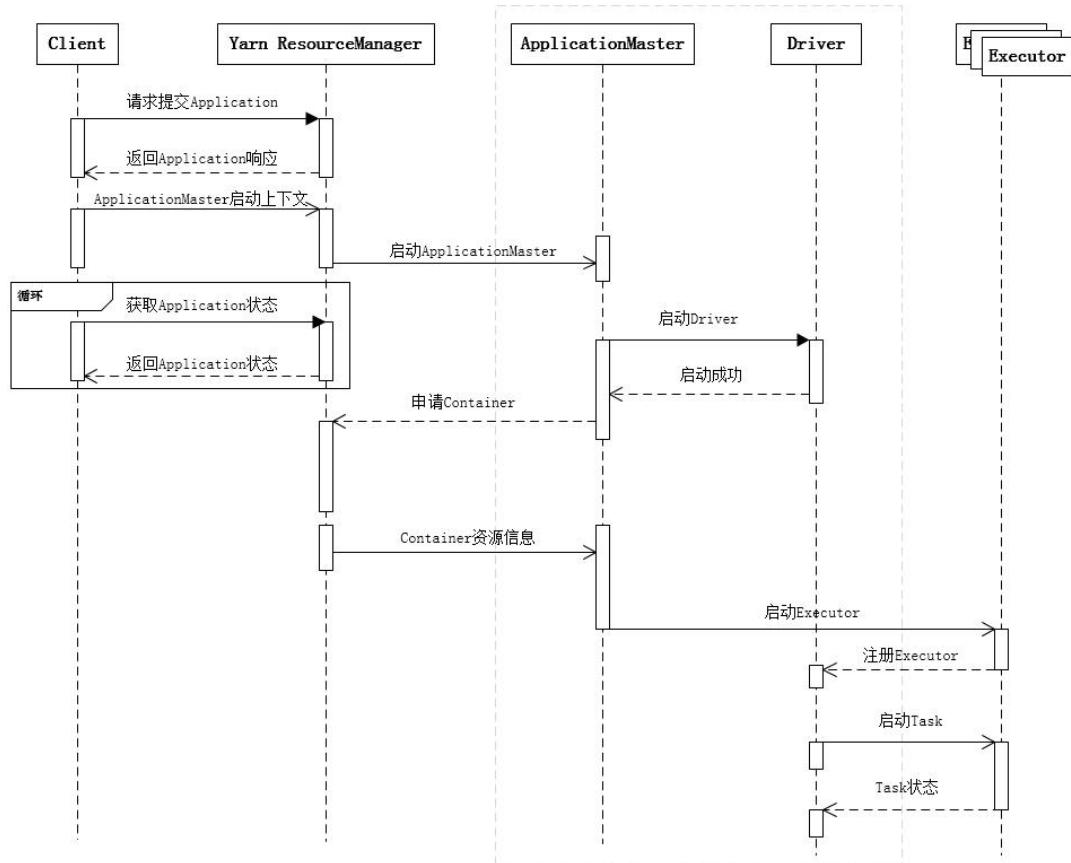


图 4-2 Spark 任务提交时序图

提交一个 Spark 应用程序，首先通过 Client 向 ResourceManager 请求启动一个 Application，同时检查是否有足够的资源满足 Application 的需求，如果资源条件满足，则准备 ApplicationMaster 的启动上下文，交给 ResourceManager，并循环监控 Application 状态。

当提交的资源队列中有资源时，ResourceManager 会在某个 NodeManager 上启动 ApplicationMaster 进程，ApplicationMaster 会单独启动 Driver 后台线程，当 Driver 启动后，ApplicationMaster 会通过本地的 RPC 连接 Driver，并开始向 ResourceManager 申请 Container 资源运行 Executor 进程（一个 Executor 对应与一个 Container），当 ResourceManager 返回 Container 资源，ApplicationMaster 则在对应的 Container 上启动 Executor。

Driver 线程主要是初始化 SparkContext 对象，准备运行所需的上下文，然后一方面保持与 ApplicationMaster 的 RPC 连接，通过 ApplicationMaster 申请资源，另一方面根据用户业务逻辑开始调度任务，将任务下发到已有的空闲 Executor 上。

当 ResourceManager 向 ApplicationMaster 返回 Container 资源时，ApplicationMaster 就尝试在对应的 Container 上启动 Executor 进程，Executor 进程起来后，会向 Driver 反向注册，注册成功后保持与 Driver 的心跳，同时等待 Driver 分发任务，当分发的任务执行完毕后，将任务状态上报给 Driver。

从上述时序图可知，Client 只负责提交 Application 并监控 Application 的状态。对于 Spark 的任务调度主要是集中在两个方面：资源申请和任务分发，其主要是通过 ApplicationMaster、Driver 以及 Executor 之间来完成。

4.2 Spark 任务调度概述

当 Driver 起来后，Driver 则会根据用户程序逻辑准备任务，并根据 Executor 资源情况逐步分发任务。在详细阐述任务调度前，首先说明下 Spark 里的几个概念。一个 Spark 应用程序包括 Job、Stage 以及 Task 三个概念：

- Job 是以 Action 方法为界，遇到一个 Action 方法则触发一个 Job；
- Stage 是 Job 的子集，以 RDD 宽依赖(即 Shuffle)为界，遇到 Shuffle 做一次划分；
- Task 是 Stage 的子集，以并行度(分区数)来衡量，分区数是多少则有多少个 task。

Spark 的任务调度总体来说分两路进行，一路是 Stage 级的调度，一路是 Task 级的调度，总体调度流程如下图所示：

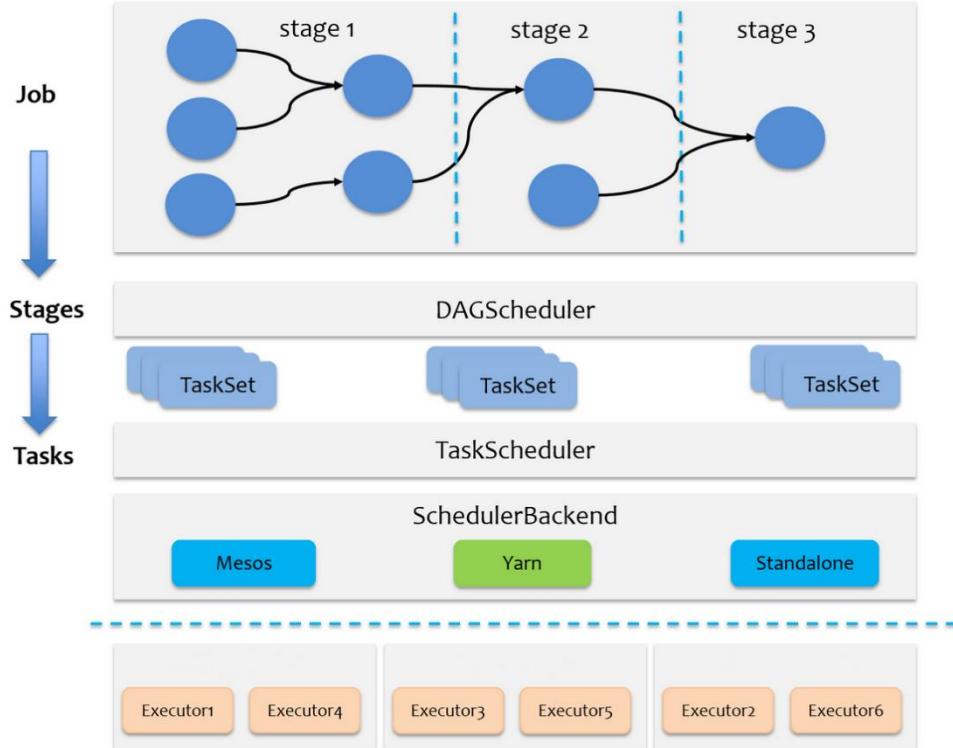


图 4-3 Spark 任务调度概览

Spark RDD 通过其 Transactions 操作，形成了 RDD 血缘关系图，即 DAG，最后通过 Action 的调用，触发 Job 并调度执行。**DAGScheduler** 负责 Stage 级的调度，主要是将 job 切分成若干 Stages，并将每个 Stage 打包成 TaskSet 交给 TaskScheduler 调度。**TaskScheduler** 负责 Task 级的调度，将 DAGScheduler 给过来的 TaskSet 按照指定的调度策略分发到 Executor 上执行，调度过程中 **SchedulerBackend** 负责提供可用资源，其中 **SchedulerBackend** 有多种实现，分别对接不同的资源管理系统。有了上述感性的认识后，下面这张图描述了 Spark-On-Yarn 模式下在任务调度期间，ApplicationMaster、Driver 以及 Executor 内部模块的交互过程：

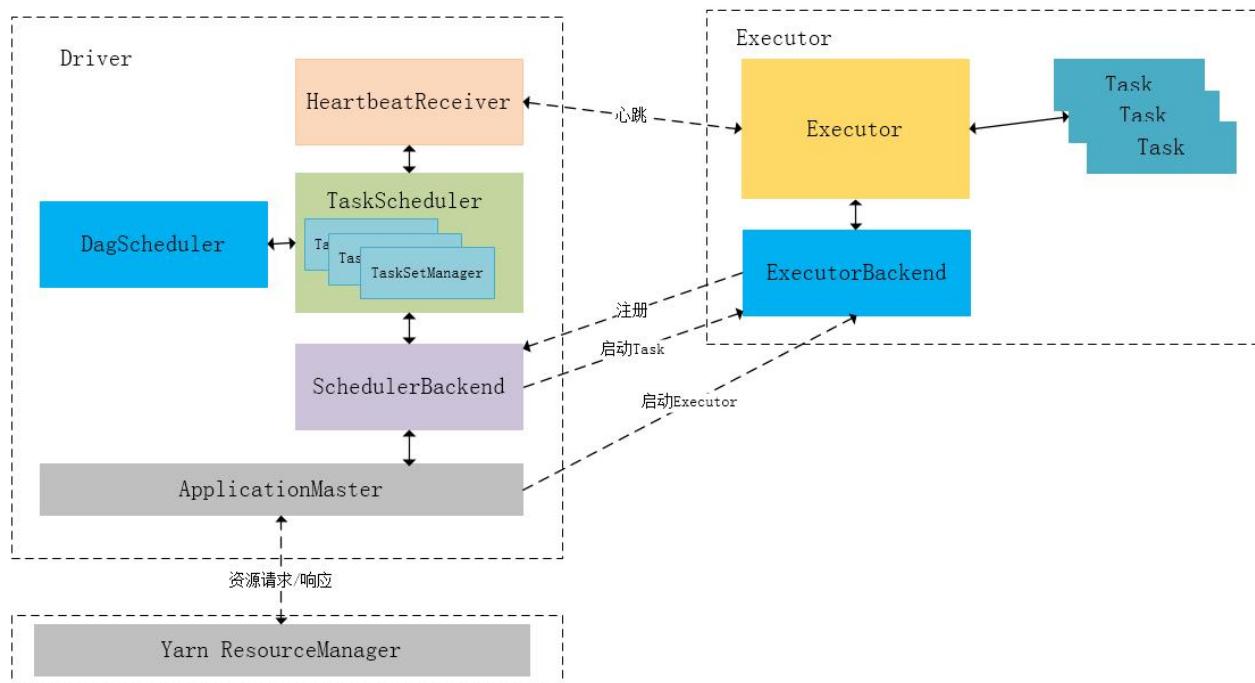
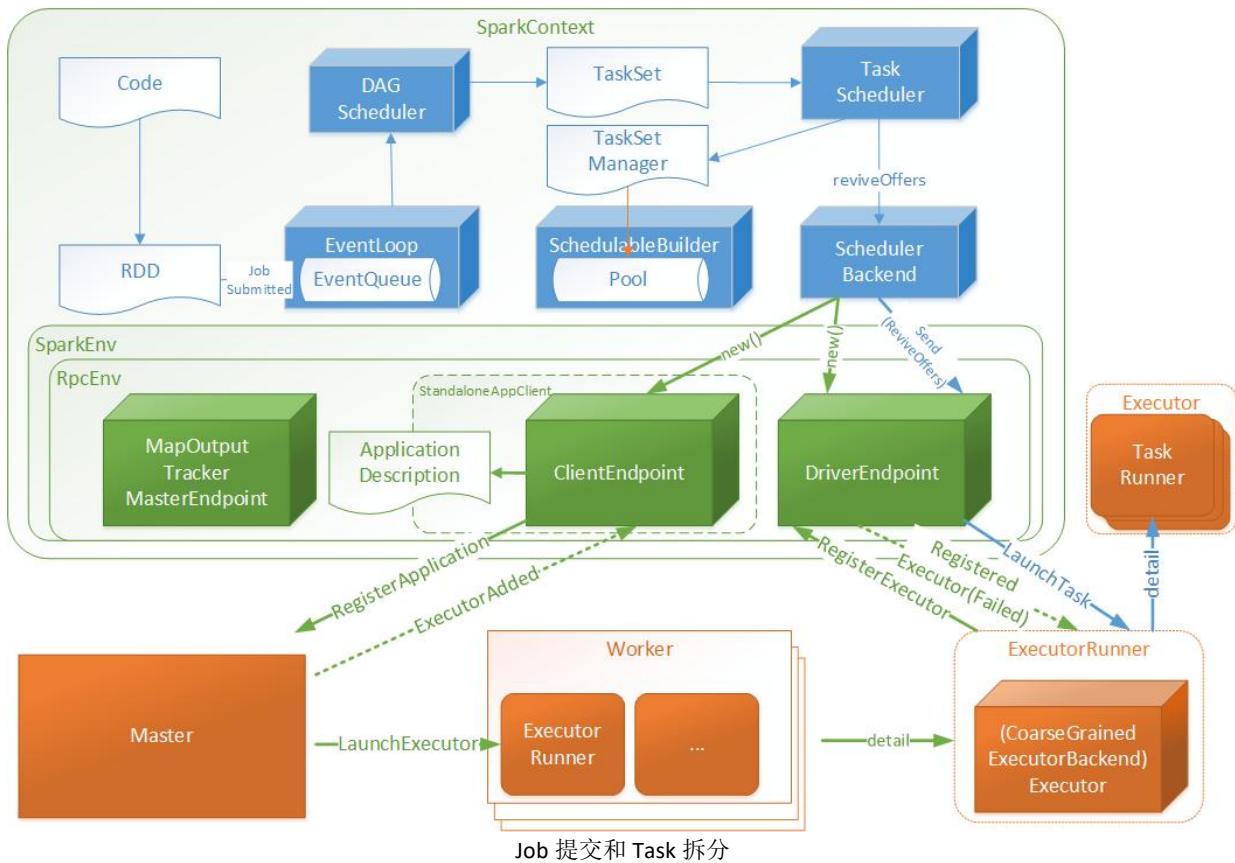


图 4-4 模块交互过程



Driver 初始化 SparkContext 过程中，会分别初始化 DAGScheduler、TaskScheduler、SchedulerBackend 以及 HeartbeatReceiver，并启动 SchedulerBackend 以及 HeartbeatReceiver。SchedulerBackend 通过 ApplicationMaster 申请资源，并不断从 TaskScheduler 中拿到合适的 Task 分发到 Executor 执行。HeartbeatReceiver 负责接收 Executor 的心跳信息，监控 Executor 的存活状况，并通知到 TaskScheduler。

4.3 Spark Stage 级调度

Spark 的任务调度是从 DAG 切割开始，主要是由 DAGScheduler 来完成。当遇到一个 Action 操作后就会触发一个 Job 的计算，并交给 DAGScheduler 来提交，下图是涉及到 Job 提交的相关方法调用流程图。

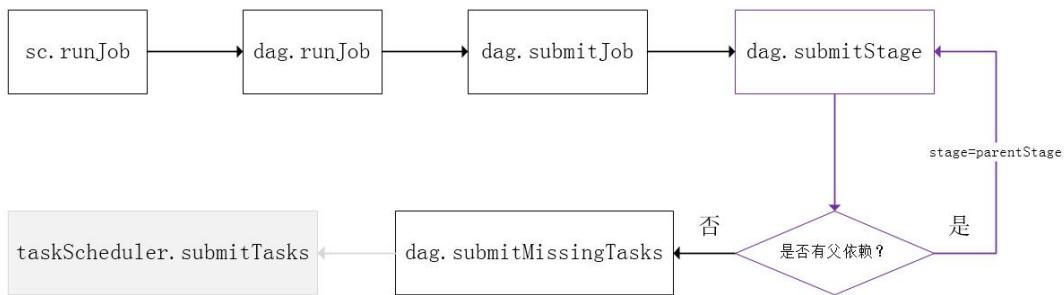


图 4-5 Job 提交调用栈

Job 由最终的 RDD 和 Action 方法封装而成，SparkContext 将 Job 交给 DAGScheduler 提交，它会根据 RDD 的血缘关系构成的 DAG 进行切分，将一个 Job 划分为若干 Stages，具体划分策略是，由最终的 RDD 不断通过依赖回溯判断父依赖是否是宽依赖，即以 Shuffle 为界划分 Stage，窄依赖的 RDD 之间被划分到同一个 Stage 中，可以进行 pipeline 式的计算，如上图紫色流程部分。划分的 Stages 分两类，一类叫做 ResultStage，为 DAG 最下游的 Stage，由 Action 方法决定，另一类叫做 ShuffleMapStage，为下游 Stage 准备数据，下面看一个简单的例子 WordCount。

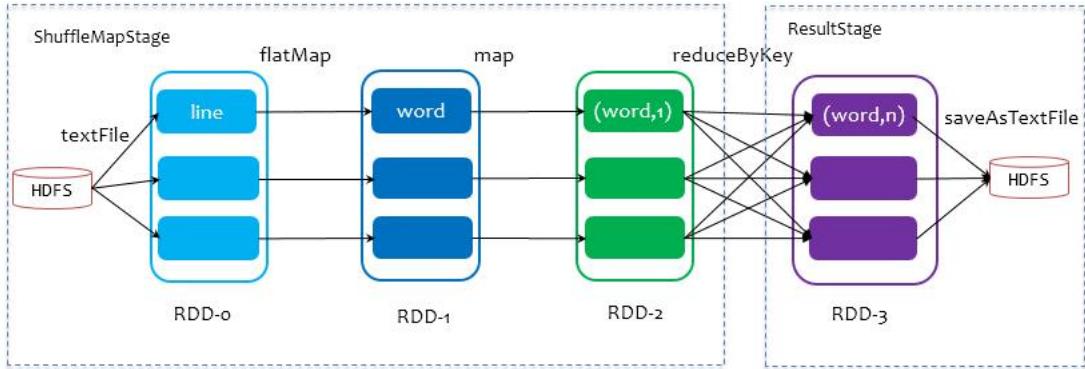


图 4-6 WordCount 实例

Job 由 saveAsTextFile 触发，该 Job 由 RDD-3 和 saveAsTextFile 方法组成，根据 RDD 之间的依赖关系从 RDD-3 开始回溯搜索，直到没有依赖的 RDD-0，在回溯搜索过程中，RDD-3 依赖 RDD-2，并且是宽依赖，所以在 RDD-2 和 RDD-3 之间划分 Stage，RDD-3 被划到最后一个 Stage，即 ResultStage 中，RDD-2 依赖 RDD-1，RDD-1 依赖 RDD-0，这些依赖都是窄依赖，所以将 RDD-0、RDD-1 和 RDD-2 划分到同一个 Stage，即 ShuffleMapStage 中，实际执行的时候，数据记录会一气呵成地执行 RDD-0 到 RDD-2 的转化。不难看出，其本质上是一个深度优先搜索算法。

一个 Stage 是否被提交，需要判断它的父 Stage 是否执行，只有在父 Stage 执行完毕才能提交当前 Stage，如果一个 Stage 没有父 Stage，那么从该 Stage 开始提交。Stage 提交时会将 Task 信息（分区信息以及方法等）序列化并被打包成 TaskSet 交给 TaskScheduler，一个 Partition 对应一个 Task，另一方面 TaskScheduler 会监控 Stage 的运行状态，只有 Executor 丢失或者 Task 由于 Fetch 失败才需要重新提交失败的 Stage 以调度运行失败的任务，其他类型的 Task 失败会在 TaskScheduler 的调度过程中重试。

相对来说 DAGScheduler 做的事情较为简单，仅仅是在 Stage 层面上划分 DAG，提交 Stage 并监控相关状态信息。TaskScheduler 则相对较为复杂，下面详细阐述其细节。

4.4 Spark Task 级调度

Spark Task 的调度是由 TaskScheduler 来完成，由前文可知，DAGScheduler 将 Stage 打包到 TaskSet 交给 TaskScheduler，TaskScheduler 会将 TaskSet 封装为 TaskSetManager 加入到调度队列中，TaskSetManager 结构如下图所示。

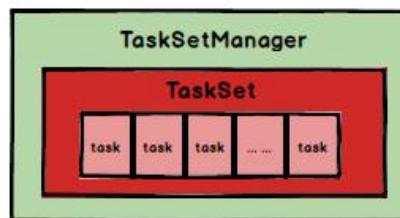
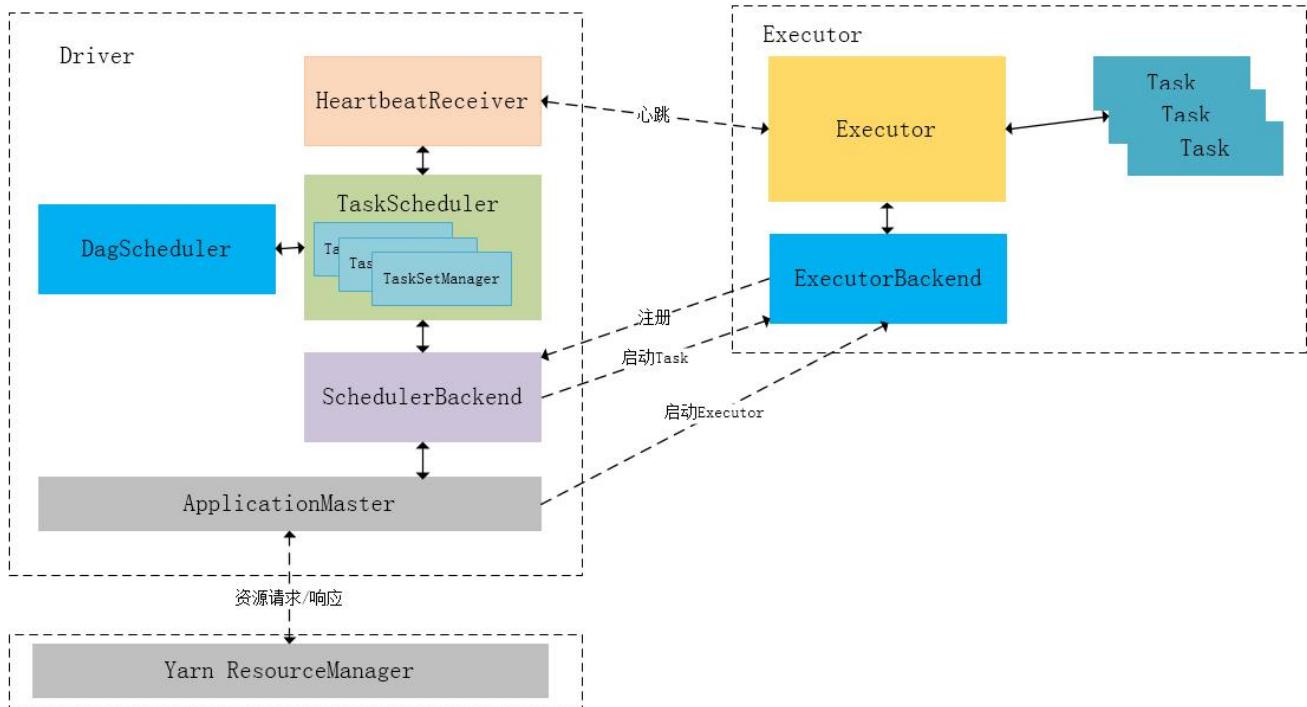


图 4-7 TaskManager 结构

TaskSetManager 负责监控管理同一个 Stage 中的 Tasks，TaskScheduler 就是以 TaskSetManager 为单元来调度任务。



前面也提到，**TaskScheduler** 初始化后会启动 **SchedulerBackend**，它负责跟外界打交道，接收 **Executor** 的注册信息，并维护 **Executor** 的状态，所以说 **SchedulerBackend** 是管“粮食”的，同时它在启动后会定期地去“询问”**TaskScheduler**有没有任务要运行，也就是说，它会定期地“问”**TaskScheduler**“我有这么余量，你要不要啊”，**TaskScheduler** 在 **SchedulerBackend**“问”它的时候，会从调度队列中按照指定的调度策略选择 **TaskSetManager** 去调度运行，大致方法调用流程如下图所示：

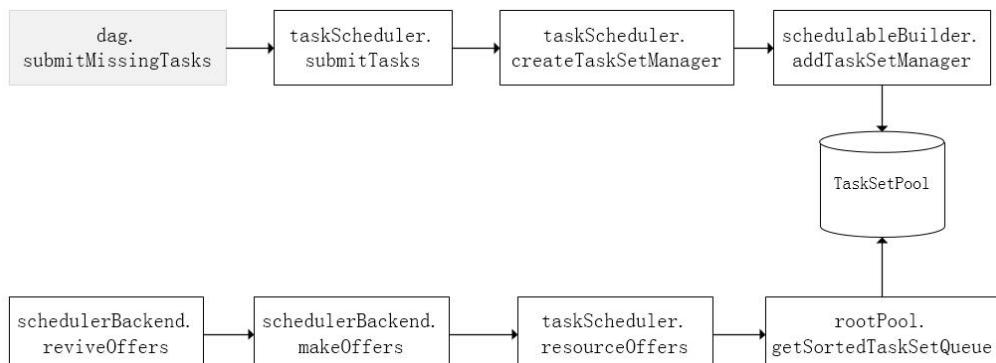


图 4-8 task 调度流程

图 3-7 中，将 **TaskSetManager** 加入 **rootPool** 调度池中之后，调用 **SchedulerBackend** 的 **reviveOffers** 方法给 **driver Endpoint** 发送 **ReviveOffer** 消息；**driverEndpoint** 收到 **ReviveOffer** 消息后调用 **makeOffers** 方法，过滤出活跃状态的 **Executor**（这些 **Executor** 都是任务启动时反向注册到 **Driver** 的 **Executor**），然后将 **Executor** 封装成 **WorkerOffer** 对象；准备好计算资源（**WorkerOffer**）后，**taskScheduler** 基于这些资源调用 **resourceOffer** 在 **Executor** 上分配 **task**。

4.4.1 调度策略

前面讲到，**TaskScheduler** 会先把 **DAGScheduler** 给过来的 **TaskSet** 封装成 **TaskSetManager** 扔到任务队列里，然后再从任务队列里按照一定的规则把它们取出来在 **SchedulerBackend** 给过来的 **Executor** 上运行。这个调度过程实际上还是比较粗粒度的，是面向 **TaskSetManager** 的。

TaskScheduler 是以树的方式来管理任务队列，树中的节点类型为 **Schedulable**，叶子节点为 **TaskSetManager**，非叶子节点为 **Pool**，下图是它们之间的继承关系。

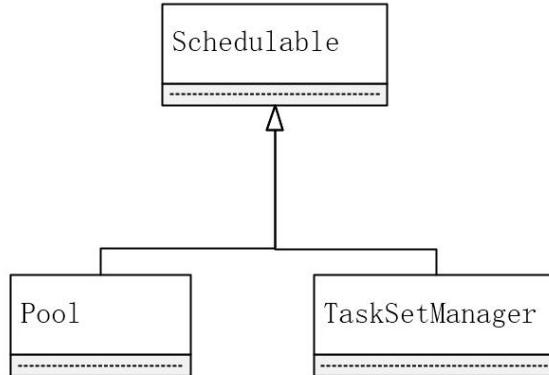


图 4-9 任务队列继承关系

TaskScheduler 支持两种调度策略，一种是 **FIFO**，也是默认的调度策略，另一种是 **FAIR**。在 TaskScheduler 初始化过程中会实例化 rootPool，表示树的根节点，是 Pool 类型。

1. FIFO 调度策略

如果是采用 FIFO 调度策略，则直接简单地将 TaskSetManager 按照先来先到的方式入队，出队时直接拿出最先进队的 TaskSetManager，其树结构如下图所示，TaskSetManager 保存在一个 FIFO 队列中。

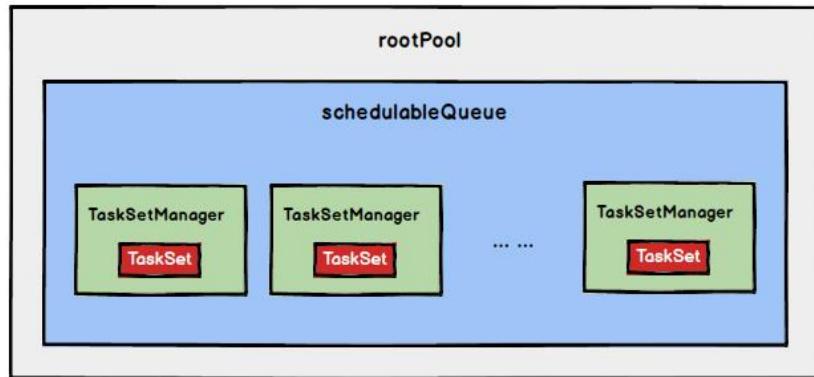


图 4-10 FIFO 调度策略内存结构

2. FAIR 调度策略

FAIR 调度策略的树结构如下图所示：

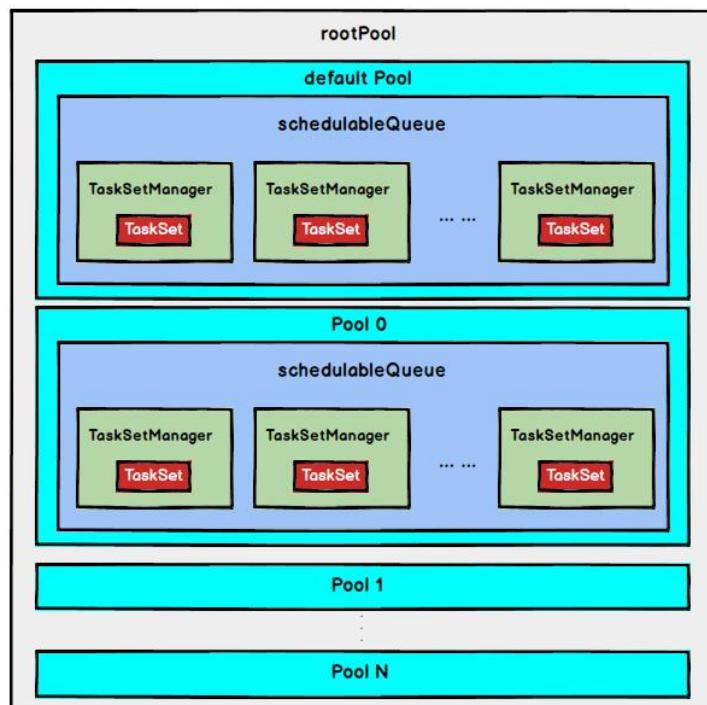


图 4-11 FAIR 调度策略内存结构

FAIR 模式中有一个 rootPool 和多个子 Pool，各个子 Pool 中存储着所有待分配的 TaskSetMagager。

在 FAIR 模式中，需要先对子 Pool 进行排序，再对子 Pool 里面的 TaskSetManager 进行排序，因为 Pool 和 TaskSetManager 都继承了 **Schedulable** 特质，因此使用相同的排序算法。

排序过程的比较是基于 **Fair-share** 来比较的，每个要排序的对象包含三个属性：runningTasks 值（正在运行的任务数）、minShare 值、weight 值，比较时会综合考量 runningTasks 值、minShare 值以及 weight 值。

注意，minShare、weight 的值均在公平调度配置文件 fairscheduler.xml 中被指定，调度池在构建阶段会读取此文件的相关配置。

- 1) 如果 A 对象的 runningTasks 大于它的 minShare，B 对象的 runningTasks 小于它的 minShare，那么 B 排在 A 前面；（runningTasks 比 minShare 小的先执行）
- 2) 如果 A、B 对象的 runningTasks 都小于它们的 minShare，那么就比较 runningTasks 与 minShare 的比值（minShare 使用率），谁小谁排前面；（minShare 使用率低的先执行）
- 3) 如果 A、B 对象的 runningTasks 都大于它们的 minShare，那么就比较 runningTasks 与 weight 的比值（权重使用率），谁小谁排前面。（权重使用率低的先执行）
- 4) 如果上述比较均相等，则比较名字。

整体上来说就是通过 **minShare** 和 **weight** 这两个参数控制比较过程，可以做到让 **minShare** 使用率和权重使用率少（实际运行 task 比例较少）的先运行。

FAIR 模式排序完成后，所有的 TaskSetManager 被放入一个 ArrayBuffer 里，之后依次被取出并发送给 Executor 执行。

从调度队列中拿到 TaskSetManager 后，由于 TaskSetManager 封装了一个 Stage 的所有 Task，并负责管理调度这些 Task，那么接下来的工作就是 TaskSetManager 按照一定的规则一个个取出 Task 给 TaskScheduler，TaskScheduler 再交给 SchedulerBackend 去发到 Executor 上执行。

4.4.2 本地化调度

DAGScheduler 切割 Job，划分 Stage，通过调用 submitStage 来提交一个 Stage 对应的 tasks，submitStage 会调用 submitMissingTasks，submitMissingTasks 确定每个需要计算的 task 的 preferredLocations，通过调用 getPreferredLocations() 得到 partition 的优先位置，由于一个 partition 对应一个 task，此 partition 的优先位置就是 task 的优先位置，对于要提交到 TaskScheduler 的 TaskSet 中的每一个 task，该 task 优先位置与其对应的 partition 对应的优先位置一致。

从调度队列中拿到 TaskSetManager 后，那么接下来的工作就是 TaskSetManager 按照一定的规则一个个取出 task 给 TaskScheduler，TaskScheduler 再交给 SchedulerBackend 去发到 Executor 上执行。前面也提到，TaskSetManager 封装了一个 Stage 的所有 task，并负责管理调度这些 task。

根据每个 task 的优先位置，确定 task 的 Locality 级别，Locality 一共有五种，优先级由高到低顺序：

名称	解析
PROCESS_LOCAL	进程本地化，task 和数据在同一个 Executor 中，性能最好。
NODE_LOCAL	节点本地化，task 和数据在同一个节点中，但是 task 和数据不在同一个 Executor 中，数据需要在进程间进行传输。
RACK_LOCAL	机架本地化，task 和数据在同一个机架的两个节点上，数据需要通过网络在节点之间进行传输。
NO_PREF	对于 task 来说，从哪里获取都一样，没有好坏之分。
ANY	task 和数据可以在集群的任何地方，而且不在一个机架中，性能最差。

表 4-1 Spark 本地化等级

在调度执行时，Spark 调度总是会尽量让每个 task 以最高的本地性级别来启动，当一个 task 以 X 本地性级别启动，但是该本地性级别对应的所有节点都没有空闲资源而启动失败，此时并不会马上降低本地性级别启动而是在某个时间长度内再次以 X 本地性级别来启动该 task，若超过限时时间则降级启动，去尝试下一个本地性级别，依次类推。

可以通过调大每个类别的最大容忍延迟时间，在等待阶段对应的 Executor 可能就会有相应的资源去执行此 task，这就在一定程度上提到了运行性能。

4.4.3 失败重试与黑名单机制

除了选择合适的 Task 调度运行外，还需要监控 Task 的执行状态，前面也提到，与外部打交道的是 SchedulerBackend，Task 被提交到 Executor 启动执行后，Executor 会将执行状态上报给 SchedulerBackend，SchedulerBackend 则告诉 TaskScheduler，TaskScheduler 找到该 Task 对应的 TaskSetManager，并通知到该 TaskSetManager，这样 TaskSet Manager 就知道 Task 的失败与成功状态，对于失败的 Task，会记录它失败的次数，如果失败次数还没有超过最大重试次数，那么就把它放回待调度的 Task 池子中，否则整个 Application 失败。

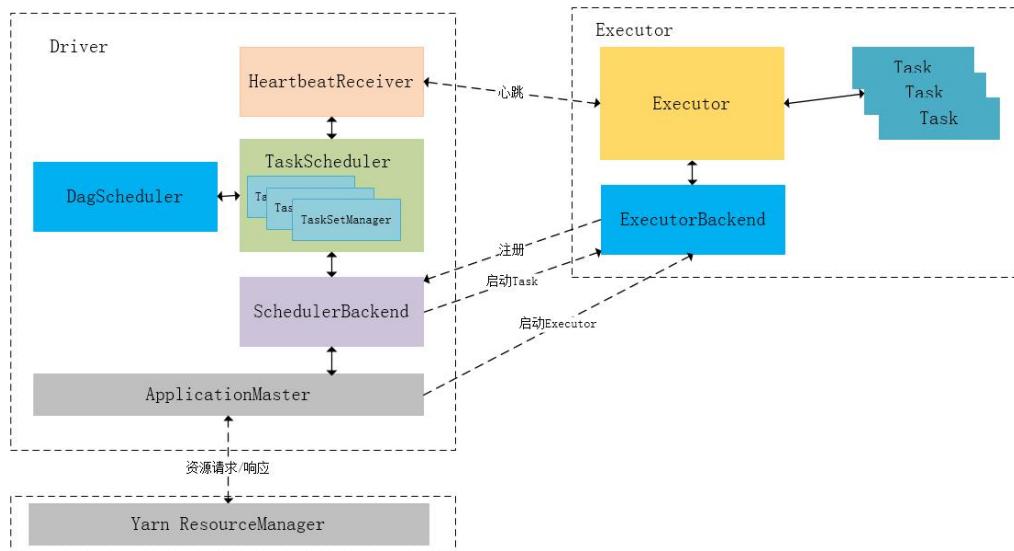


图 4-12 模块交互过程

在记录 Task 失败次数过程中，会记录它上一次失败所在的 Executor Id 和 Host，这样下次再调度这个 Task 时，会使用黑名单机制，避免它被调度到上一次失败的节点上，起到一定的容错作用。黑名单记录 Task 上一次失败所在的 Executor Id 和 Host，以及其对应的“拉黑”时间，“拉黑”时间是指这段时间内不要再往这个节点上调度这个 Task 了。

第 5 章 Spark Shuffle 解析

5.1 Shuffle 的核心要点

5.1.1 ShuffleMapStage 与 ResultStage

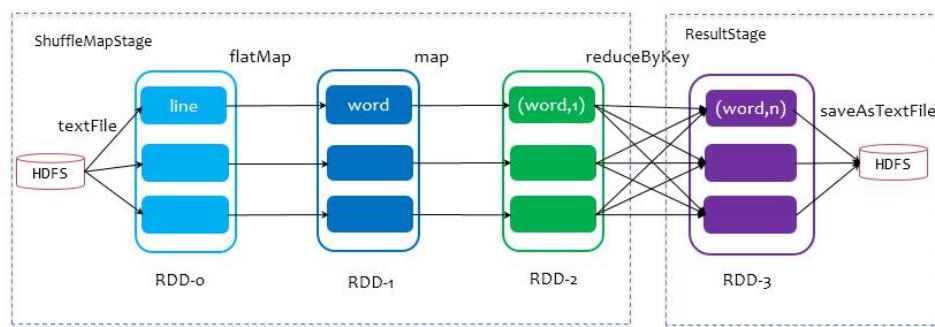


图 5-1 ShuffleMapStage 与 ResultStage

在划分 stage 时，最后一个 stage 称为 finalStage，它本质上是一个 ResultStage 对象，前面的所有 stage 被称为 ShuffleMapStage。

ShuffleMapStage 的结束伴随着 shuffle 文件的写磁盘。

ResultStage 基本上对应代码中的 action 算子，即将一个函数应用在 RDD 的各个 partition 的数据集上，意味着一个 job 的运行结束。

5.1.2 Shuffle 中的任务个数

我们知道，Spark Shuffle 分为 map 阶段和 reduce 阶段，或者称之为 ShuffleRead 阶段和 ShuffleWrite 阶段，那么对于一次 Shuffle，map 过程和 reduce 过程都会由若干个 task 来执行，那么 map task 和 reduce task 的数量是如何确定的呢？

假设 Spark 任务从 HDFS 中读取数据，那么初始 RDD 分区个数由该文件的 split 个数决定，也就是一个 split 对应生成的 RDD 的一个 partition，我们假设初始 partition 个数为 N。

初始 RDD 经过一系列算子计算后（假设没有执行 repartition 和 coalesce 算子进行重分区，则分区个数不变，仍为 N，如果经过重分区算子，那么分区个数变为 M），我们假设分区个数不变，当执行到 Shuffle 操作时，map 端的 task 个数和 partition 个数一致，即 map task 为 N 个。

reduce 端的 stage 默认取 spark.default.parallelism 这个配置项的值作为分区数，如果没有配置，则以 map 端的最后一个 RDD 的分区数作为其分区数（也就是 N），那么分区数就决定了 reduce 端的 task 的个数。

5.1.3 reduce 端数据的读取

根据 stage 的划分我们知道，map 端 task 和 reduce 端 task 不在相同的 stage 中，map task 位于 ShuffleMapStage，reduce task 位于 ResultStage，map task 会先执行，那么后执行的 reduce task 如何知道从哪里去拉取 map task 落盘后的数据呢？

reduce 端的数据拉取过程如下：

1. map task 执行完毕后会将计算状态以及磁盘小文件位置等信息封装到 MapStatus 对象中，然后由本进程中的 MapOutputTrackerWorker 对象将 mapStatus 对象发送给 Driver 进程的 MapOutputTrackerMaster 对象；
2. 在 reduce task 开始执行之前会先让本进程中的 MapOutputTrackerWorker 向 Driver 进程中的 MapOutputTrackerMaster 发动请求，请求磁盘小文件位置信息；
3. 当所有的 Map task 执行完毕后，Driver 进程中的 MapOutputTrackerMaster 就掌握了所有的磁盘小文件的位置信息。此时 MapOutputTrackerMaster 会告诉 MapOutputTrackerWorker 磁盘小文件的位置信息；
4. 完成之前的操作之后，由 BlockTransferService 去 Executor0 所在的节点拉数据，默认会启动五个子线程。每次拉取的数据量不能超过 48M（reduce task 每次最多拉取 48M 数据，将拉来的数据存储到 Executor 内存的 20% 内存中）。

5.2 HashShuffle 解析

以下的讨论都假设每个 Executor 有 1 个 CPU core。

1. 未经优化的 HashShuffleManager

shuffle write 阶段，主要就是在在一个 stage 结束计算之后，为了下一个 stage 可以执行 shuffle 类的算子（比如 reduceByKey），而将每个 task 处理的数据按 key 进行“划分”。所谓“划分”，就是对相同的 key 执行 hash 算法，从而将相同 key 都写入同一个磁盘文件中，而每一个磁盘文件都只属于下游 stage 的一个 task。在将数据写入磁盘之前，会先将数据写入内存缓冲中，当内存缓冲填满之后，才会溢写到磁盘文件中去。

下一个 stage 的 task 有多少个，当前 stage 的每个 task 就要创建多少份磁盘文件。比如下一个 stage 总共有 10 0 个 task，那么当前 stage 的每个 task 都要创建 100 份磁盘文件。如果当前 stage 有 50 个 task，总共有 10 个 Executor，每个 Executor 执行 5 个 task，那么每个 Executor 上总共就要创建 500 个磁盘文件，所有 Executor 上会创建 500 0 个磁盘文件。由此可见，未经优化的 shuffle write 操作所产生的磁盘文件的数量是极其惊人的。

shuffle read 阶段，通常就是一个 stage 刚开始时要做的事情。此时该 stage 的每一个 task 就需要将上一个 stage 的计算结果中的所有相同 key，从各个节点上通过网络都拉取到自己所在的节点上，然后进行 key 的聚合或连接等操作。由于 shuffle write 的过程中，map task 给下游 stage 的每个 reduce task 都创建了一个磁盘文件，因此 shuffle read 的过程中，每个 reduce task 只要从上游 stage 的所有 map task 所在节点上，拉取属于自己的那一个磁盘文件即可。

shuffle read 的拉取过程是一边拉取一边进行聚合的。每个 shuffle read task 都会有一个自己的 buffer 缓冲，每次都只能拉取与 buffer 缓冲相同大小的数据，然后通过内存中的一个 Map 进行聚合等操作。聚合完一批数据后，

再拉取下一批数据，并放到 buffer 缓冲中进行聚合操作。以此类推，直到最后将所有数据到拉取完，并得到最终的结果。

未优化的 HashShuffleManager 工作原理如图 1-7 所示：

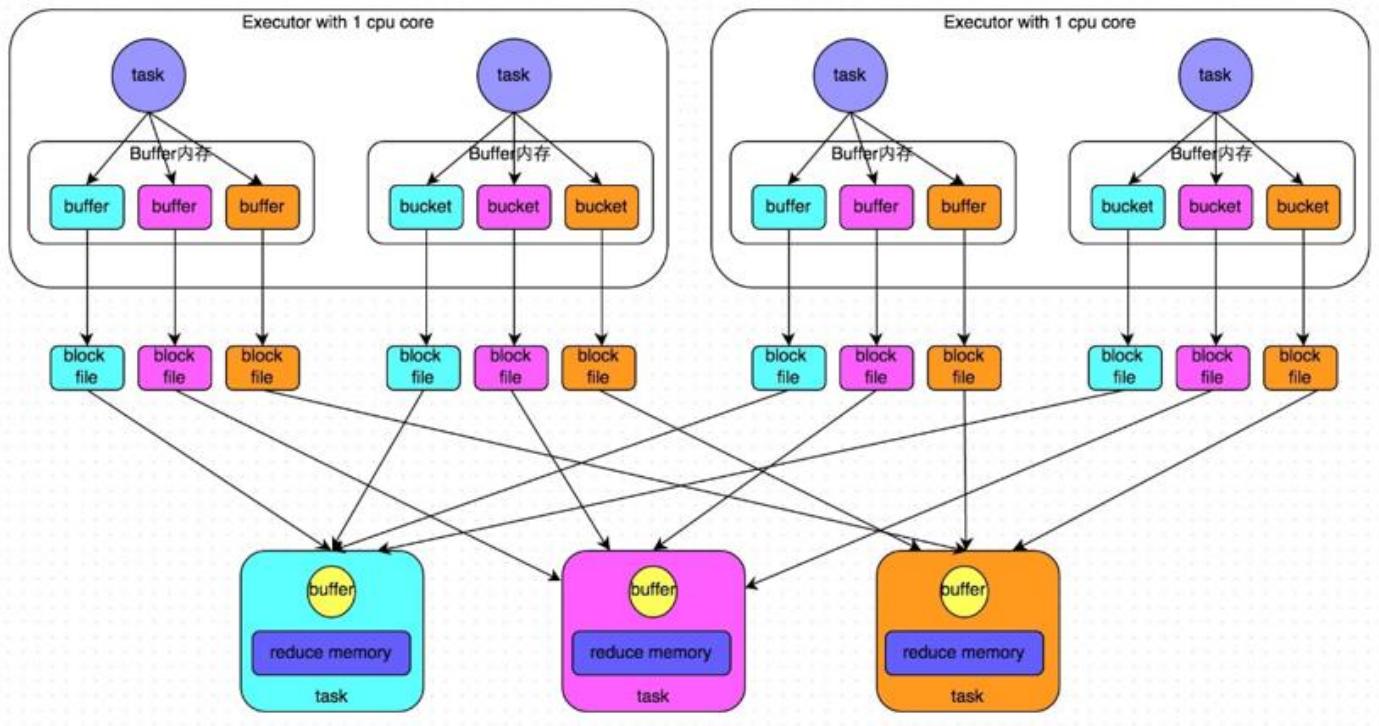


图 1-7 未优化的 HashShuffleManager 工作原理

2. 优化后的 HashShuffleManager

为了优化 HashShuffleManager 我们可以设置一个参数，`spark.shuffle.consolidateFiles`，该参数默认值为 `false`，将其设置为 `true` 即可开启优化机制，通常来说，如果我们使用 HashShuffleManager，那么都建议开启这个选项。

开启 `consolidate` 机制之后，在 `shuffle write` 过程中，task 就不是为下游 stage 的每个 task 创建一个磁盘文件了，此时会出现 **shuffleFileGroup** 的概念，每个 `shuffleFileGroup` 会对应一批磁盘文件，磁盘文件的数量与下游 stage 的 task 数量是相同的。一个 Executor 上有多少个 CPU core，就可以并行执行多少个 task。而第一批并行执行的每个 task 都会创建一个 `shuffleFileGroup`，并将数据写入对应的磁盘文件内。

当 Executor 的 CPU core 执行完一批 task，接着执行下一批 task 时，下一批 task 就会复用之前已有的 `shuffleFileGroup`，包括其中的磁盘文件，也就是说，此时 task 会将数据写入已有的磁盘文件中，而不会写入新的磁盘文件中。因此，`consolidate` 机制允许不同的 task 复用同一批磁盘文件，这样就可以有效将多个 task 的磁盘文件进行一定程度上的合并，从而大幅度减少磁盘文件的数量，进而提升 `shuffle write` 的性能。

假设第二个 stage 有 100 个 task，第一个 stage 有 50 个 task，总共还是有 10 个 Executor (Executor CPU 个数为 1)，每个 Executor 执行 5 个 task。那么原本使用未经优化的 HashShuffleManager 时，每个 Executor 会产生 500 个磁盘文件，所有 Executor 会产生 5000 个磁盘文件的。但是此时经过优化之后，每个 Executor 创建的磁盘文件的数量的计算公式为：CPU core 的数量 * 下一个 stage 的 task 数量，也就是说，每个 Executor 此时只会创建 100 个磁盘文件，所有 Executor 只会创建 1000 个磁盘文件。

优化后的 HashShuffleManager 工作原理如图 1-8 所示：

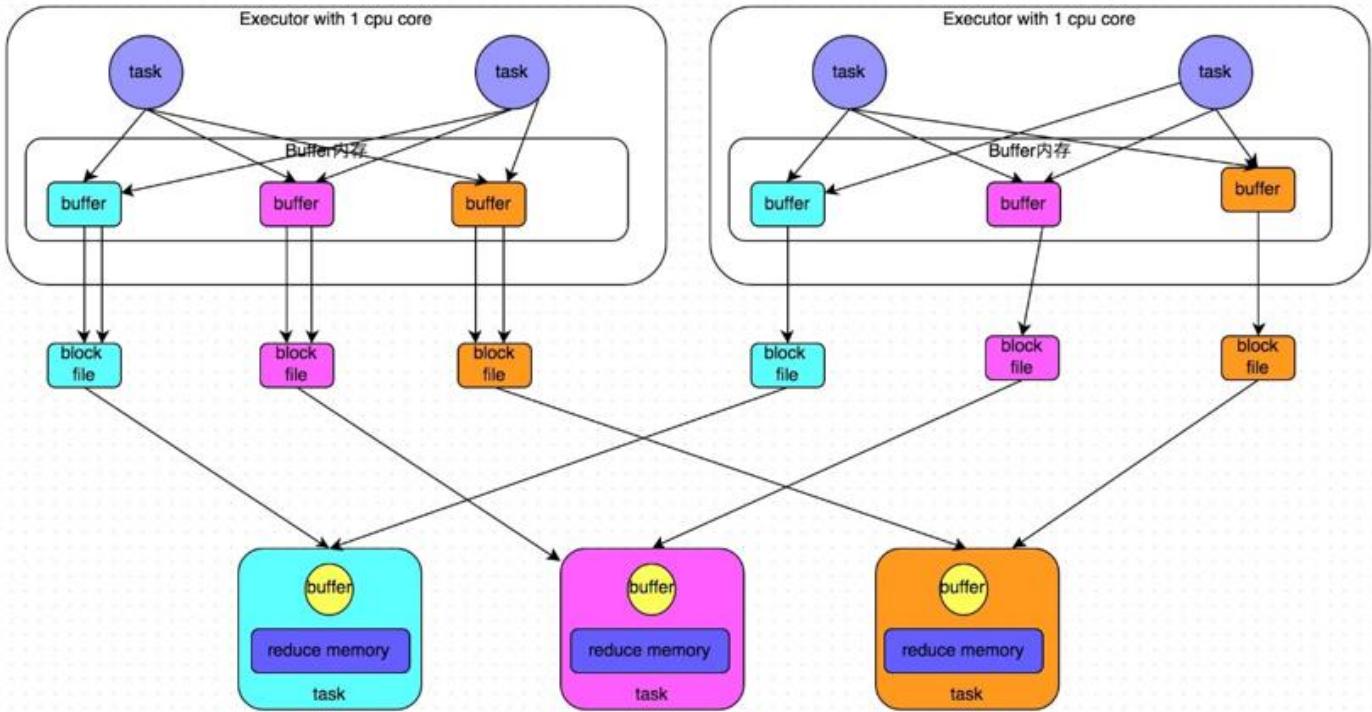


图 1-8 优化后的 HashShuffleManager 工作原理

5.3 SortShuffle 解析

SortShuffleManager 的运行机制主要分成两种，一种是普通运行机制，另一种是 bypass 运行机制。当 shuffle read task 的数量小于等于 spark.shuffle.sort.bypassMergeThreshold 参数的值时（默认为 200），就会启用 bypass 机制。

1. 普通运行机制

在该模式下，数据会先写入一个内存数据结构中，此时根据不同的 shuffle 算子，可能选用不同的数据结构。如果是 reduceByKey 这种聚合类的 shuffle 算子，那么会选用 Map 数据结构，一边通过 Map 进行聚合，一边写入内存；如果是 join 这种普通的 shuffle 算子，那么会选用 Array 数据结构，直接写入内存。接着，每写一条数据进入内存数据结构之后，就会判断一下，是否达到了某个临界阈值。如果达到临界阈值的话，那么就会尝试将内存数据结构中的数据溢写到磁盘，然后清空内存数据结构。

在溢写到磁盘文件之前，会先根据 key 对内存数据结构中已有的数据进行排序。排序过后，会分批将数据写入磁盘文件。默认的 batch 数量是 10000 条，也就是说，排序好的数据，会以每批 1 万条数据的形式分批写入磁盘文件。写入磁盘文件是通过 Java 的 BufferedOutputStream 实现的。BufferedOutputStream 是 Java 的缓冲输出流，首先会将数据缓冲在内存中，当内存缓冲满溢之后再一次写入磁盘文件中，这样可以减少磁盘 IO 次数，提升性能。

一个 task 将所有数据写入内存数据结构的过程中，会发生多次磁盘溢写操作，也就会产生多个临时文件。最后会将之前所有的临时磁盘文件都进行合并，这就是 merge 过程，此时会将之前所有临时磁盘文件中的数据读取出来，然后依次写入最终的磁盘文件之中。此外，由于一个 task 就只对应一个磁盘文件，也就意味着该 task 为下游 stage 的 task 准备的数据都在这一个文件中，因此还会单独写一份索引文件，其中标识了下游各个 task 的数据在文件中的 start offset 与 end offset。

SortShuffleManager 由于有一个磁盘文件 merge 的过程，因此大大减少了文件数量。比如第一个 stage 有 50 个 task，总共有 10 个 Executor，每个 Executor 执行 5 个 task，而第二个 stage 有 100 个 task。由于每个 task 最终只有一个磁盘文件，因此此时每个 Executor 上只有 5 个磁盘文件，所有 Executor 只有 50 个磁盘文件。

普通运行机制的 SortShuffleManager 工作原理如图 1-9 所示：

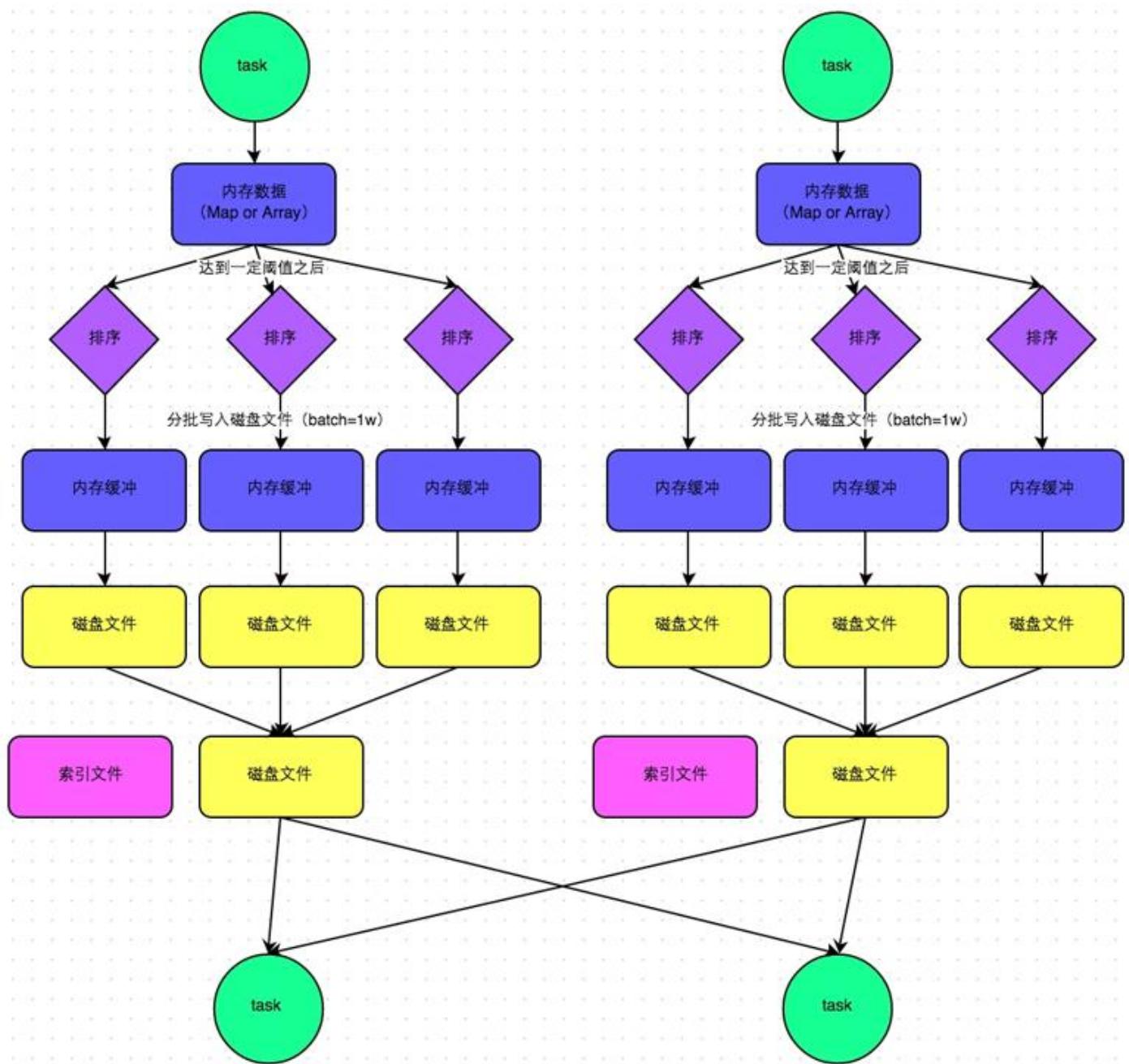


图 1-9 普通运行机制的 SortShuffleManager 工作原理

2. bypass 运行机制

bypass 运行机制的触发条件如下：

- shuffle map task 数量小于 spark.shuffle.sort.bypassMergeThreshold 参数的值。
- 不是聚合类的 shuffle 算子。

此时，每个 task 会为每个下游 task 都创建一个临时磁盘文件，并将数据按 key 进行 hash 然后根据 key 的 hash 值，将 key 写入对应的磁盘文件之中。当然，写入磁盘文件时也是先写入内存缓冲，缓冲写满之后再溢写到磁盘文件的。最后，同样会将所有临时磁盘文件都合并成一个磁盘文件，并创建一个单独的索引文件。

该过程的磁盘写机制其实跟未经优化的 HashShuffleManager 是一模一样的，因为都要创建数量惊人的磁盘文件，只是在最后会做一个磁盘文件的合并而已。因此少量的最终磁盘文件，也让该机制相对未经优化的 HashShuffleManager 来说，shuffle read 的性能会更好。

而该机制与普通 SortShuffleManager 运行机制的不同在于：第一，磁盘写机制不同；第二，不会进行排序。也就是说，启用该机制的最大好处在于，shuffle write 过程中，不需要进行数据的排序操作，也就节省掉了这部分的性能开销。

bypass 运行机制的 SortShuffleManager 工作原理如图 1-10 所示：

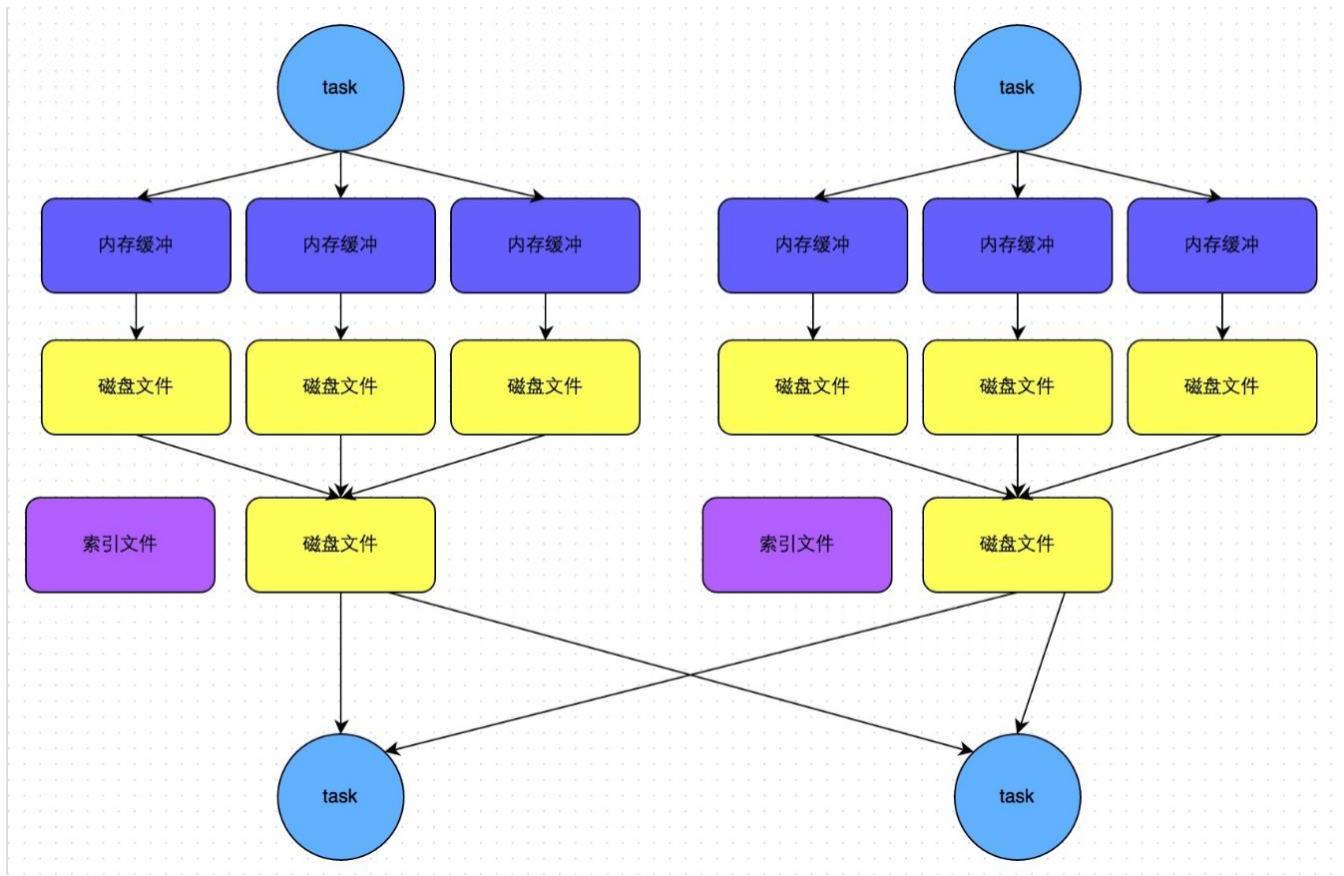


图 1-10 bypass 运行机制的 SortShuffleManager 工作原理

第 6 章 Spark 内存管理

在执行 Spark 的应用程序时，Spark 集群会启动 Driver 和 Executor 两种 JVM 进程，前者为主控进程，负责创建 Spark 上下文，提交 Spark 作业（Job），并将作业转化为计算任务（Task），在各个 Executor 进程间协调任务的调度，后者负责在工作节点上执行具体的计算任务，并将结果返回给 Driver，同时为需要持久化的 RDD 提供存储功能。由于 Driver 的内存管理相对来说较为简单，本节主要对 Executor 的内存管理进行分析，下文中的 Spark 内存均特指 Executor 的内存。

6.1 堆内和堆外内存规划

作为一个 JVM 进程，Executor 的内存管理建立在 JVM 的内存管理之上，Spark 对 JVM 的堆内（On-heap）空间进行了更为详细的分配，以充分利用内存。同时，Spark 引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，进一步优化了内存的使用。

堆内内存受到 JVM 统一管理，堆外内存是直接向操作系统进行内存的申请和释放。

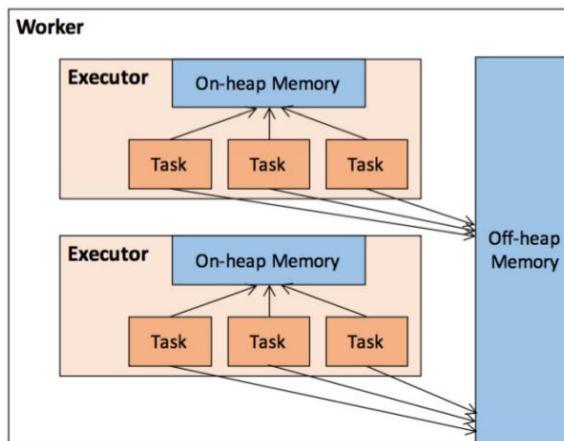


图 1-1 Executor 堆内与堆外内存

1. 堆内存

堆内存的大小，由 Spark 应用程序启动时的 `-executor-memory` 或 `spark.executor.memory` 参数配置。Executor 内运行的并发任务共享 JVM 堆内存，这些任务在缓存 RDD 数据和广播（Broadcast）数据时占用的内存被规划为存储（Storage）内存，而这些任务在执行 Shuffle 时占用的内存被规划为执行（Execution）内存，剩余的部分不做特殊规划，那些 Spark 内部的对象实例，或者用户定义的 Spark 应用程序中的对象实例，均占用剩余的空间。不同的管理模式下，这三部分占用的空间大小各不相同。

Spark 对堆内存的管理是一种逻辑上的“规划式”的管理，因为对象实例占用内存的申请和释放都由 JVM 完成，Spark 只能在申请后和释放前记录这些内存，我们来看其具体流程：

申请内存流程如下：

1. Spark 在代码中 new 一个对象实例；
2. JVM 从堆内存分配空间，创建对象并返回对象引用；
3. Spark 保存该对象的引用，记录该对象占用的内存。

释放内存流程如下：

1. Spark 记录该对象释放的内存，删除该对象的引用；
2. 等待 JVM 的垃圾回收机制释放该对象占用的堆内存。

我们知道，JVM 的对象可以以序列化的方式存储，序列化的过程是将对象转换为二进制字节流，本质上可以理解为将非连续空间的链式存储转化为连续空间或块存储，在访问时则需要进行序列化的逆过程——反序列化，将字节流转化为对象，序列化的方式可以节省存储空间，但增加了存储和读取时候的计算开销。

对于 Spark 中序列化的对象，由于是字节流的形式，其占用的内存大小可直接计算，而对于非序列化的对象，其占用的内存是通过周期性地采样近似估算而得，即并不是每次新增的数据项都会计算一次占用的内存大小，这种方法降低了时间开销但是有可能误差较大，导致某一时刻的实际内存有可能远远超出预期。此外，在被 Spark 标记为释放的对象实例，很有可能在实际上并没有被 JVM 回收，导致实际可用的内存小于 Spark 记录的可用内存。所以 Spark 并不能准确记录实际可用的堆内存，从而也就无法完全避免内存溢出（OOM, Out of Memory）的异常。

虽然不能精准控制堆内存的申请和释放，但 Spark 通过对存储内存和执行内存各自独立的规划管理，可以决定是否要在存储内存里缓存新的 RDD，以及是否为新的任务分配执行内存，在一定程度上可以提升内存的利用率，减少异常的出现。

2. 堆外内存

为了进一步优化内存的使用以及提高 Shuffle 时排序的效率，Spark 引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，存储经过序列化的二进制数据。

堆外内存意味着把内存对象分配在 Java 虚拟机的堆以外的内存，这些内存直接受操作系统管理（而不是虚拟机）。这样做的结果就是能保持一个较小的堆，以减少垃圾收集对应用的影响。

利用 JDK Unsafe API（从 Spark 2.0 开始，在管理堆外的存储内存时不再基于 Tachyon，而是与堆外的执行内存一样，基于 JDK Unsafe API 实现），Spark 可以直接操作系统堆外内存，减少了不必要的内存开销，以及频繁的 GC 扫描和回收，提升了处理性能。堆外内存可以被精确地申请和释放（堆外内存之所以能够被精确的申请和释放，是由于内存的申请和释放不再通过 JVM 机制，而是直接向操作系统申请，JVM 对于内存的清理是无法准确指定时间点的，因此无法实现精确的释放），而且序列化的数据占用的空间可以被精确计算，所以相比堆内存来说降低了管理的难度，也降低了误差。

在默认情况下堆外内存并不启用，可通过配置 `spark.memory.offHeap.enabled` 参数启用，并由 `spark.memory.offHeap.size` 参数设定堆外空间的大小。除了没有 other 空间，堆外内存与堆内存的划分方式相同，所有运行中的并发任务共享存储内存和执行内存。

(*该部分内存主要用于程序的共享库、Perm Space、线程 Stack 和一些 Memory mapping 等，或者类 C 方式 allocate object)

6.2 内存空间分配

1. 静态内存管理

在 Spark 最初采用的静态内存管理机制下，存储内存、执行内存和其他内存的大小在 Spark 应用程序运行期间均为固定的，但用户可以应用程序启动前进行配置，堆内内存的分配如图 2 所示：

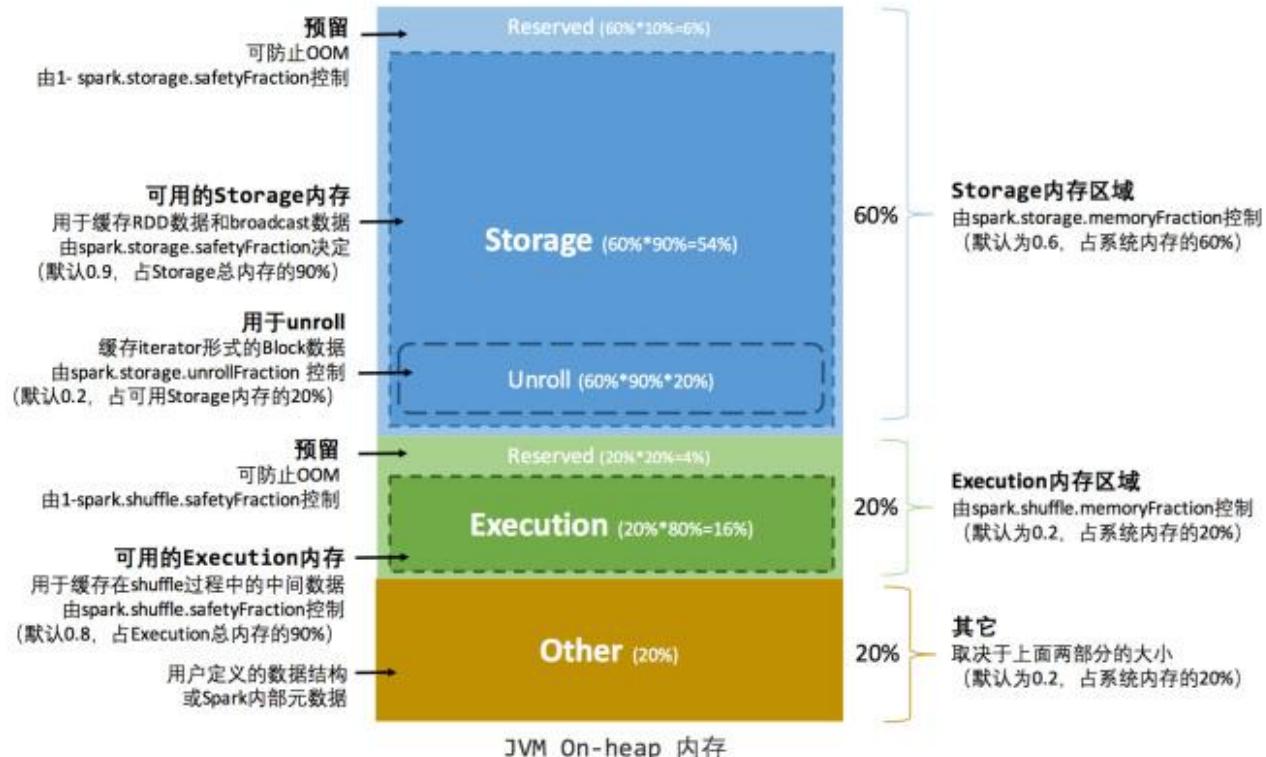


图 1-2 静态内存管理——堆内内存

可以看到，可用的堆内内存的大小需要按照代码清单 1-1 的方式计算：

代码清单 1-1 堆内内存计算公式

```
可用的存储内存 = systemMaxMemory * spark.storage.memoryFraction * spark.storage.safetyFraction  
可用的执行内存 = systemMaxMemory * spark.shuffle.memoryFraction * spark.shuffle.safetyFraction
```

其中 `systemMaxMemory` 取决于当前 JVM 堆内内存的大小，最后可用的执行内存或者存储内存要在此基础上与各自的 `memoryFraction` 参数和 `safetyFraction` 参数相乘得出。上述计算公式中的两个 `safetyFraction` 参数，其意义在于在逻辑上预留出 $1 - \text{safetyFraction}$ 这么一块保险区域，降低因实际内存超出当前预设范围而导致 OOM 的风险（上文提到，对于非序列化对象的内存采样估算会产生误差）。值得注意的是，这个预留的保险区域仅仅是一种逻辑上的规划，在具体使用时 Spark 并没有区别对待，和“其它内存”一样交给了 JVM 去管理。

Storage 内存和 Execution 内存都有预留空间，目的是防止 OOM，因为 Spark 堆内内存大小的记录是不准确的，需要留出保险区域。

堆外的空间分配较为简单，只有存储内存和执行内存，如图 1-3 所示。可用的执行内存和存储内存占用的空间大小直接由参数 `spark.memory.storageFraction` 决定，由于堆外内存占用的空间可以被精确计算，所以无需再设定保险区域。

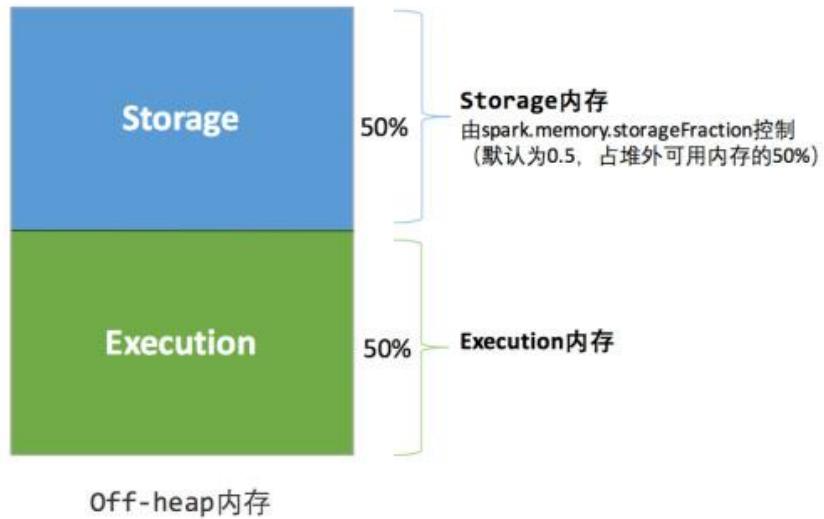


图 1-3 静态内存管理

静态内存管理机制实现起来较为简单，但如果用户不熟悉 Spark 的存储机制，或没有根据具体的数据规模和计算任务或做相应的配置，很容易造成“一半海水，一半火焰”的局面，即存储内存和执行内存中的一方剩余大量的空间，而另一方却早早被占满，不得不淘汰或移出旧的内容以存储新的内容。由于新的内存管理机制的出现，这种方式目前已经很少有开发者使用，出于兼容旧版本的应用程序的目的，Spark 仍然保留了它的实现。

2. 统一内存管理

Spark 1.6 之后引入的统一内存管理机制，与静态内存管理的区别在于存储内存和执行内存共享同一块空间，可以动态占用对方的空闲区域，统一内存管理的堆内内存结构如图 1-4 所示：

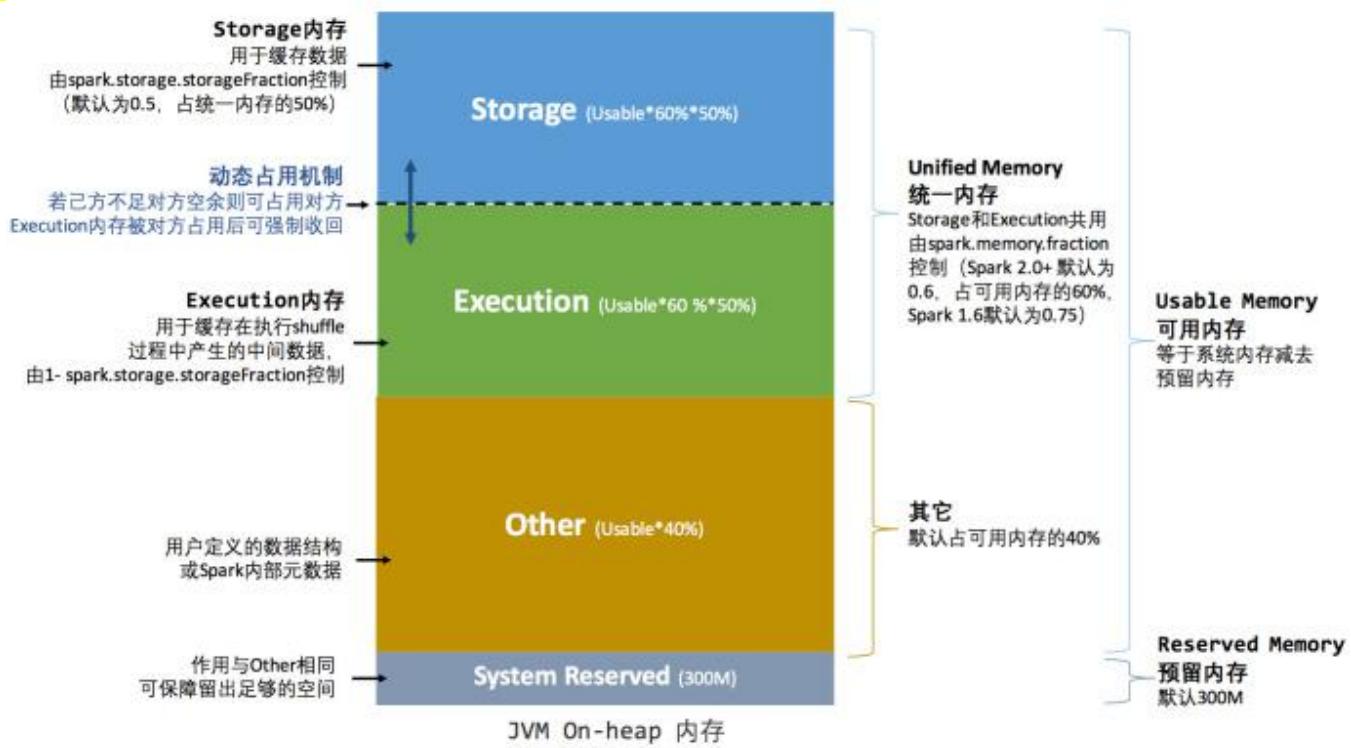


图 1-4 统一内存管理——堆内内存

统一内存管理的堆外内存结构如图 1-5 所示：

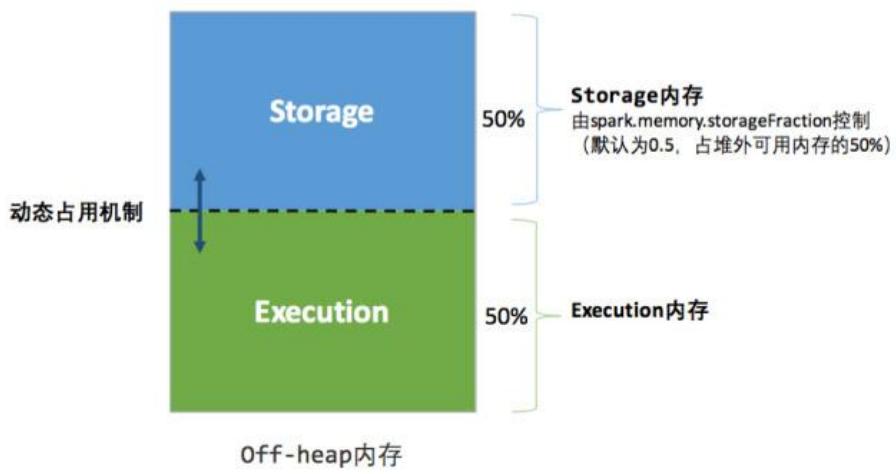


图 1-5 统一内存管理——堆外内存

其中最重要的优化在于动态占用机制，其规则如下：

1. 设定基本的存储内存和执行内存区域（spark.storage.storageFraction 参数），该设定确定了双方各自拥有的空间的范围；
2. 双方的空间都不足时，则存储到硬盘；若己方空间不足而对方空余时，可借用对方的空间；（存储空间不足是指不足以放下一个完整的 Block）
3. 执行内存的空间被对方占用后，可让对方将占用的部分转存到硬盘，然后“归还”借用的空间；
4. 存储内存的空间被对方占用后，无法让对方“归还”，因为需要考虑 Shuffle 过程中的很多因素，实现起来较为复杂。

统一内存管理的动态占用机制如图 1-6 所示：

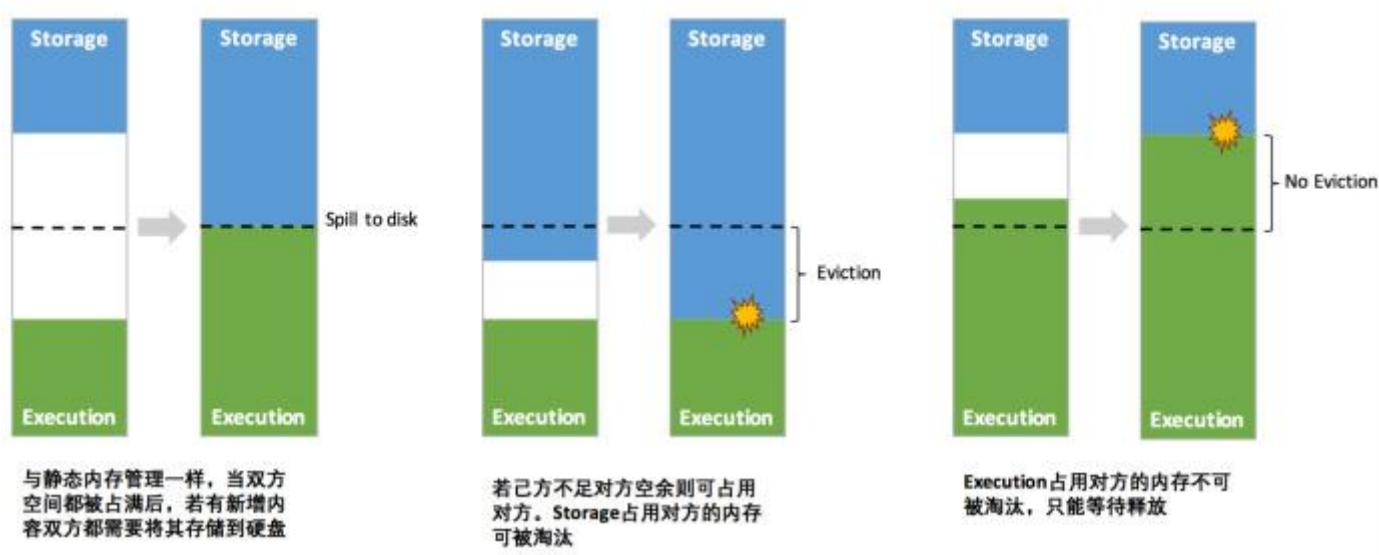


图 1-6 同一内存管理——动态占用机制

凭借统一内存管理机制，Spark 在一定程度上提高了堆内和堆外内存资源的利用率，降低了开发者维护 Spark 内存的难度，但并不意味着开发者可以高枕无忧。如果存储内存的空间太大或者说缓存的数据过多，反而会导致频繁的全量垃圾回收，降低任务执行时的性能，因为缓存的 RDD 数据通常都是长期驻留内存的。所以要想充分发挥 Spark 的性能，需要开发者进一步了解存储内存和执行内存各自的管理方式和实现原理。

6.3 存储内存管理

1. RDD 的持久化机制

弹性分布式数据集（RDD）作为 Spark 最根本的数据抽象，是只读的分区记录（Partition）的集合，只能基于在稳定物理存储中的数据集上创建，或者在其他已有的 RDD 上执行转换（Transformation）操作产生一个新的 RDD。转换后的 RDD 与原始的 RDD 之间产生的依赖关系，构成了血统（Lineage）。凭借血统，Spark 保证了每一个 RDD

都可以被重新恢复。但 RDD 的所有转换都是惰性的，即只有当一个返回结果给 Driver 的行动（Action）发生时，Spark 才会创建任务读取 RDD，然后真正触发转换的执行。

Task 在启动之初读取一个分区时，会先判断这个分区是否已经被持久化，如果没有则需要检查 Checkpoint 或按照血统重新计算。所以如果一个 RDD 上要执行多次行动，可以在第一次行动中使用 persist 或 cache 方法，在内存或磁盘中持久化或缓存这个 RDD，从而在后面的行动时提升计算速度。

事实上，cache 方法是使用默认的 MEMORY_ONLY 的存储级别将 RDD 持久化到内存，故缓存是一种特殊的持久化。堆内和堆外存储内存的设计，便可以对缓存 RDD 时使用的内存做统一的规划和管理。

RDD 的持久化由 Spark 的 Storage 模块负责，实现了 RDD 与物理存储的解耦合。Storage 模块负责管理 Spark 在计算过程中产生的数据，将那些在内存或磁盘、在本地或远程存取数据的功能封装了起来。在具体实现时 Driver 端和 Executor 端的 Storage 模块构成了主从式的架构，即 Driver 端的 BlockManager 为 Master，Executor 端的 BlockManager 为 Slave。

Storage 模块在逻辑上以 Block 为基本存储单位，RDD 的每个 Partition 经过处理后唯一对应一个 Block (BlockId 的格式为 rdd_RDD-ID_PARTITION-ID)。Driver 端的 Master 负责整个 Spark 应用程序的 Block 的元数据信息的管理和维护，而 Executor 端的 Slave 需要将 Block 的更新等状态上报到 Master，同时接收 Master 的命令，例如新增或删除一个 RDD。

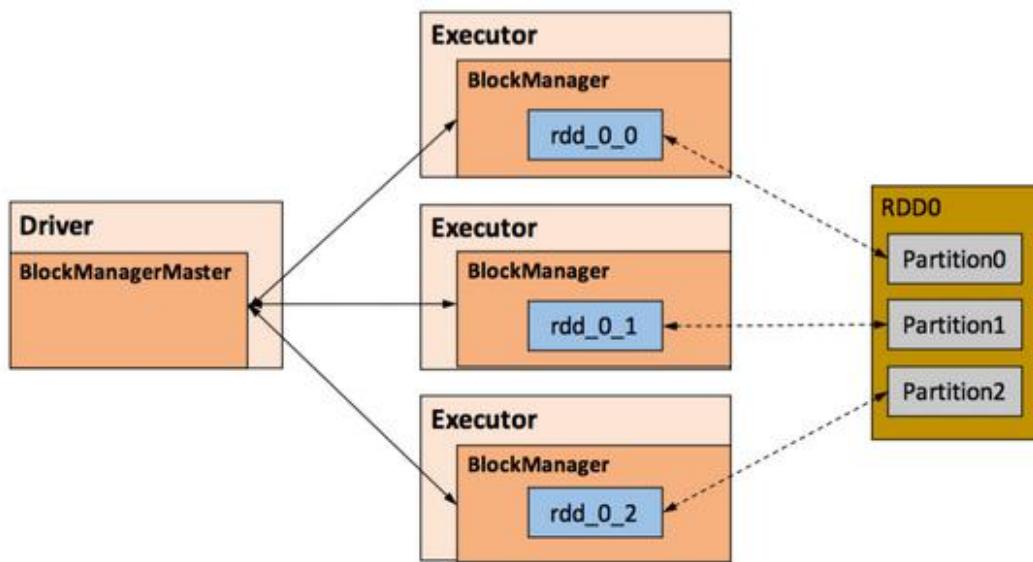


图 5-1 Storage 模块示意图

在对 RDD 持久化时，Spark 规定了 MEMORY_ONLY、MEMORY_AND_DISK 等 7 种不同的存储级别，而存储级别是以下 5 个变量的组合：

```
class StorageLevel private{
    private var _useDisk: Boolean, //磁盘
    private var _useMemory: Boolean, //这里其实是指堆内内存
    private var _useOffHeap: Boolean, //堆外内存
    private var _deserialized: Boolean, //是否为非序列化
    private var _replication: Int = 1 //副本个数
}
```

代码清单 5-1 resourceOffer 代码

Spark 中 7 种存储级别如下：

表 5-1 Spark 持久化级别

持久化级别	含义
MEMORY_ONLY	以非序列化的 Java 对象的方式持久化在 JVM 内存中。如果内存无法完全存储 RDD 所有的 partition，那么那些没有持久化的 partition 就会在下一次需要使用它们的时候，重新被计算

MEMORY_AND_DISK	同上，但是当某些 partition 无法存储在内存中时，会持久化到磁盘中。下次需要使用这些 partition 时，需要从磁盘上读取
MEMORY_ONLY_SER	同 MEMORY_ONLY，但是会使用 Java 序列化方式，将 Java 对象序列化后进行持久化。可以减少内存开销，但是需要进行反序列化，因此会加大 CPU 开销
MEMORY_AND_DISK_SER	同 MEMORY_AND_DISK，但是使用序列化方式持久化 Java 对象
DISK_ONLY	使用非序列化 Java 对象的方式持久化，完全存储到磁盘上
MEMORY_ONLY_2 MEMORY_AND_DISK_2 等等	如果是尾部加了 2 的持久化级别，表示将持久化数据复用一份，保存到其他节点，从而在数据丢失时，不需要再次计算，只需要使用备份数据即可

通过对数据结构的分析，可以看出存储级别从三个维度定义了 RDD 的 Partition（同时也就是 Block）的存储方式：

- 1) **存储位置**: 磁盘 / 堆内内存 / 堆外内存。如 MEMORY_AND_DISK 是同时在磁盘和堆内内存上存储，实现了冗余备份。OFF_HEAP 则是只在堆外内存存储，目前选择堆外内存时不能同时存储到其他位置。
- 2) **存储形式**: Block 缓存到存储内存后，是否为非序列化的形式。如 MEMORY_ONLY 是非序列化方式存储，OFF_HEAP 是序列化方式存储。
- 3) **副本数量**: 大于 1 时需要远程冗余备份到其他节点。如 DISK_ONLY_2 需要远程备份 1 个副本。

2. RDD 的缓存过程

RDD 在缓存到存储内存之前，Partition 中的数据一般以迭代器（Iterator）的数据结构来访问，这是 Scala 语言中一种遍历数据集合的方法。通过 Iterator 可以获取分区中每一条序列化或者非序列化的数据项(Record)，这些 Record 的对象实例在逻辑上占用了 JVM 堆内内存的 other 部分的空间，同一 Partition 的不同 Record 的存储空间并不连续。

RDD 在缓存到存储内存之后，Partition 被转换成 Block，Record 在堆内或堆外存储内存中占用一块连续的空间。将 Partition 由不连续的存储空间转换为连续存储空间的过程，Spark 称之为“展开”（Unroll）。

Block 有序列化和非序列化两种存储格式，具体以哪种方式取决于该 RDD 的存储级别。非序列化的 Block 以一种 DeserializedMemoryEntry 的数据结构定义，用一个数组存储所有的对象实例，序列化的 Block 则以 SerializedMemoryEntry 的数据结构定义，用字节缓冲区（ByteBuffer）来存储二进制数据。每个 Executor 的 Storage 模块用一个链式 Map 结构（LinkedHashMap）来管理堆内和堆外存储内存中所有的 Block 对象的实例，对这个 LinkedHashMap 新增和删除间接记录了内存的申请和释放。

因为不能保证存储空间可以一次容纳 Iterator 中的所有数据，当前的计算任务在 Unroll 时要向 MemoryManager 申请足够的 Unroll 空间来临时占位，空间不足则 Unroll 失败，空间足够时可以继续进行。

对于序列化的 Partition，其所需的 Unroll 空间可以直接累加计算，一次申请。

对于非序列化的 Partition 则要在遍历 Record 的过程中依次申请，即每读取一条 Record，采样估算其所需的 Unroll 空间并进行申请，空间不足时可以中断，释放已占用的 Unroll 空间。

如果最终 Unroll 成功，当前 Partition 所占用的 Unroll 空间被转换为正常的缓存 RDD 的存储空间，如下图所示。

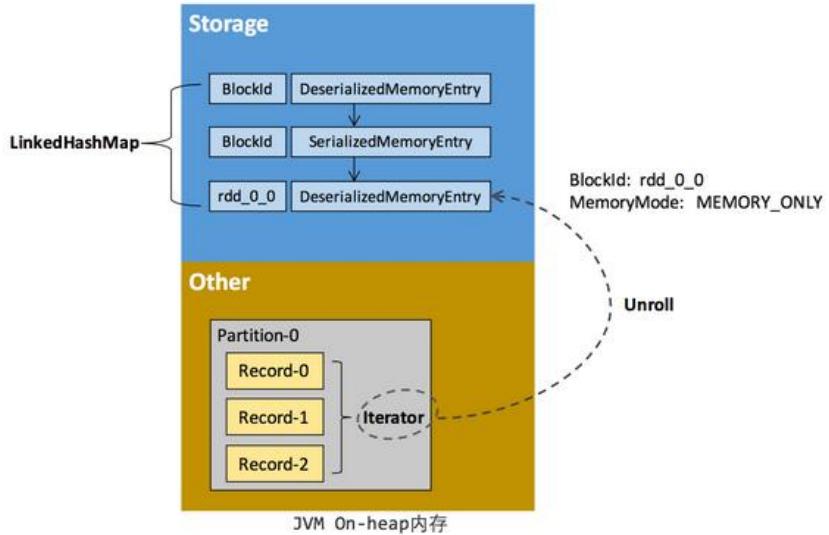


图 5-2 Spark Unroll

在静态内存管理时，Spark 在存储内存中专门划分了一块 **Unroll** 空间，其大小是固定的，统一内存管理时则没有对 **Unroll** 空间进行特别区分，当存储空间不足时会根据动态占用机制进行处理。

1. 淘汰与落盘

由于同一个 **Executor** 的所有的计算任务共享有限的存储内存空间，当有新的 **Block** 需要缓存但是剩余空间不足且无法动态占用时，就要对 **LinkedHashMap** 中的旧 **Block** 进行淘汰（**Eviction**），而被淘汰的 **Block** 如果其存储级别中同时包含存储到磁盘的要求，则要对其进行落盘（**Drop**），否则直接删除该 **Block**。

存储内存的淘汰规则为：

- 被淘汰的旧 **Block** 要与新 **Block** 的 **MemoryMode** 相同，即同属于堆外或堆内内存；
- 新旧 **Block** 不能属于同一个 **RDD**，避免循环淘汰；
- 旧 **Block** 所属 **RDD** 不能处于被读状态，避免引发一致性问题；
- 遍历 **LinkedHashMap** 中 **Block**，按照最近最少使用（LRU）的顺序淘汰，直到满足新 **Block** 所需的空间。其中 LRU 是 **LinkedHashMap** 的特性。

落盘的流程则比较简单，如果其存储级别符合 **_useDisk** 为 **true** 的条件，再根据其 **_deserialized** 判断是否是非序列化形式，若是则对其进行序列化，最后将数据存储到磁盘，在 **Storage** 模块中更新其信息。

6.4 执行内存管理

执行内存主要用来存储任务在执行 **Shuffle** 时占用的内存，**Shuffle** 是按照一定规则对 **RDD** 数据重新分区的过程，我们来看 **Shuffle** 的 **Write** 和 **Read** 两阶段对执行内存的使用：

● Shuffle Write

- 1) 若在 **map** 端选择普通的排序方式，会采用 **ExternalSorter** 进行外排，在内存中存储数据时主要占用堆内执行空间。
- 2) 若在 **map** 端选择 **Tungsten** 的排序方式，则采用 **ShuffleExternalSorter** 直接对以序列化形式存储的数据排序，在内存中存储数据时可以占用堆外或堆内执行空间，取决于用户是否开启了堆外内存以及堆外执行内存是否足够。

● Shuffle Read

- 1) 在对 **reduce** 端的数据进行聚合时，要将数据交给 **Aggregator** 处理，在内存中存储数据时占用堆内执行空间。
- 2) 如果需要进行最终结果排序，则要将再次将数据交给 **ExternalSorter** 处理，占用堆内执行空间。

在 **ExternalSorter** 和 **Aggregator** 中，Spark 会使用一种叫 **AppendOnlyMap** 的哈希表在堆内执行内存中存储数据，但在 **Shuffle** 过程中所有数据并不能都保存到该哈希表中，当这个哈希表占用的内存会进行周期性地采样估算，当其大到一定程度，无法再从 **MemoryManager** 申请到新的执行内存时，Spark 就会将其全部内容存储到磁盘文件中，这个过程被称为溢存(**Spill**)，溢存到磁盘的文件最后会被归并(**Merge**)。

Shuffle Write 阶段中用到的 **Tungsten** 是 Databricks 公司提出的对 Spark 优化内存和 CPU 使用的计划(钨丝计划)，解决了一些 **JVM** 在性能上的限制和弊端。Spark 会根据 **Shuffle** 的情况来自动选择是否采用 **Tungsten** 排序。

Tungsten 采用的页式内存管理机制建立在 MemoryManager 之上，即 Tungsten 对执行内存的使用进行了一步的抽象，这样在 Shuffle 过程中无需关心数据具体存储在堆内还是堆外。

每个内存页用一个 MemoryBlock 来定义，并用 Object obj 和 long offset 这两个变量统一标识一个内存页在系统内存中的地址。

堆内的 MemoryBlock 是以 long 型数组的形式分配的内存，其 obj 的值为是这个数组的对象引用，offset 是 long 型数组在 JVM 中的初始偏移地址，两者配合使用可以定位这个数组在堆内的绝对地址；堆外的 MemoryBlock 是直接申请到的内存块，其 obj 为 null，offset 是这个内存块在系统内存中的 64 位绝对地址。**Spark** 用 **MemoryBlock** 巧妙地将堆内和堆外内存页统一抽象封装，并用页表(pageTable)管理每个 Task 申请到的内存页。

Tungsten 页式管理下的所有内存用 64 位的逻辑地址表示，由页号和页内偏移量组成：

- 页号：占 13 位，唯一标识一个内存页，Spark 在申请内存页之前要先申请空闲页号。
- 页内偏移量：占 51 位，是在使用内存页存储数据时，数据在页内的偏移地址。

有了统一的寻址方式，Spark 可以用 64 位逻辑地址的指针定位到堆内或堆外的内存，整个 Shuffle Write 排序的过程只需要对指针进行排序，并且无需反序列化，整个过程非常高效，对于内存访问效率和 CPU 使用效率带来了明显的提升。

Spark 的存储内存和执行内存有着截然不同的管理方式：对于存储内存来说，Spark 用一个 LinkedHashMap 来集中管理所有的 Block，Block 由需要缓存的 RDD 的 Partition 转化而成；而对于执行内存，Spark 用 AppendOnlyMap 来存储 Shuffle 过程中的数据，在 Tungsten 排序中甚至抽象成为页式内存管理，开辟了全新的 JVM 内存管理机制。

第 7 章 Spark 核心组件解析

7.1 BlockManager 数据存储与管理机制

BlockManager 是整个 Spark 底层负责数据存储与管理的一个组件，Driver 和 Executor 的所有数据都由对应的 BlockManager 进行管理。

Driver 上有 BlockManagerMaster，负责对各个节点上的 BlockManager 内部管理的数据的元数据进行维护，比如 block 的增删改等操作，都会在这里维护好元数据的变更。

每个节点都有一个 BlockManager，每个 BlockManager 创建之后，第一件事即使去向 BlockManagerMaster 进行注册，此时 BlockManagerMaster 会为其长句对应的 BlockManagerInfo。

BlockManager 运行原理如下图所示：

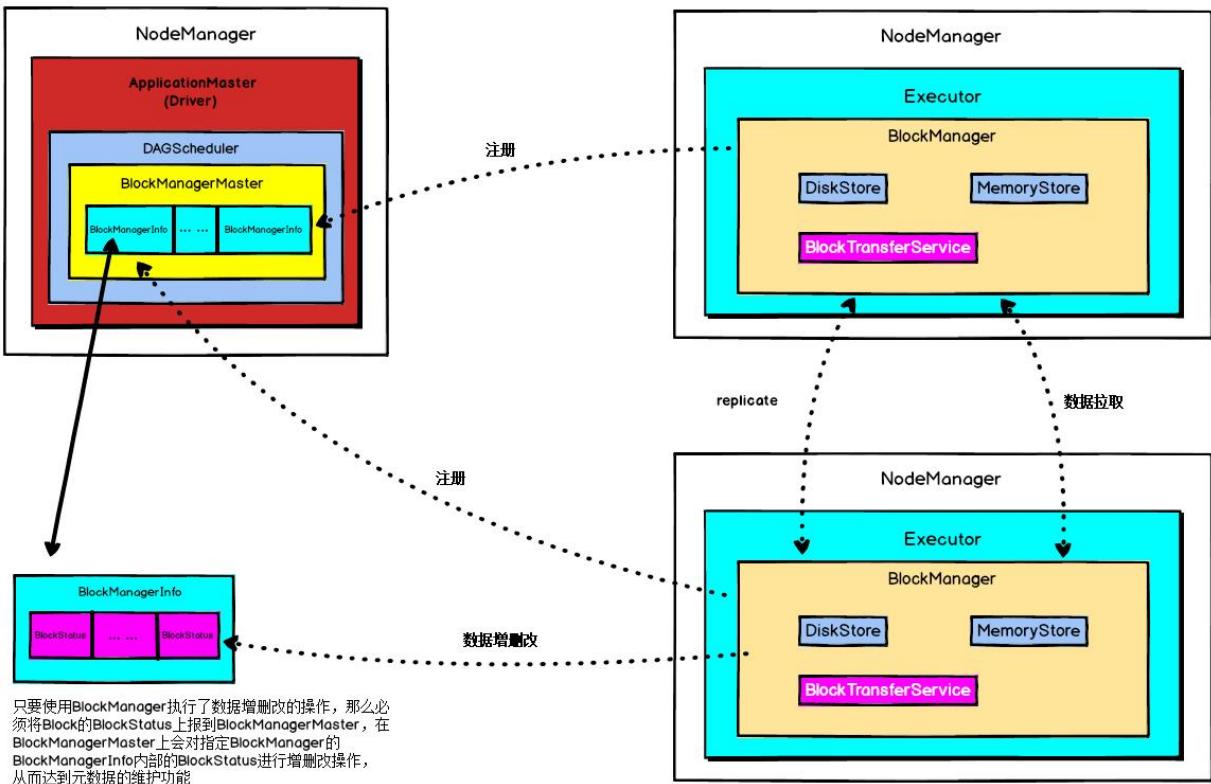


图 7-1 BlockManager 原理

BlockManagerMaster 与 BlockManager 的关系非常像 NameNode 与 DataNode 的关系，BlockManagerMaster 中保存中 BlockManager 内部管理数据的元数据，进行维护，当 BlockManager 进行 Block 增删改等操作时，都会在 Block ManagerMaster 中进行元数据的变更，这与 NameNode 维护 DataNode 的元数据信息，DataNode 中数据发生变化时 NameNode 中的元数据信息也会相应变化是一致的。

每个节点上都有一个 BlockManager，BlockManager 中有 3 个非常重要的组件：

- DiskStore：负责对磁盘数据进行读写；
- MemoryStore：负责对内存数据进行读写；
- BlockTransferService：负责建立 BlockManager 到远程其他节点的 BlockManager 的连接，负责对远程其他节点的 BlockManager 的数据进行读写；

每个 BlockManager 创建之后，做的第一件事就是想 BlockManagerMaster 进行注册，此时 BlockManagerMaster 会为其创建对应的 BlockManagerInfo。

使用 BlockManager 进行写操作时，比如说，RDD 运行过程中的一些中间数据，或者我们手动指定了 persist()，会优先将数据写入内存中，如果内存大小不够，会使用自己的算法，将内存中的部分数据写入磁盘；此外，如果 persist() 指定了要 replica，那么会使用 BlockTransferService 将数据 replicate 一份到其他节点的 BlockManager 上去。

使用 BlockManager 进行读操作时，比如说，shuffleRead 操作，如果能从本地读取，就利用 DiskStore 或者 MemoryStore 从本地读取数据，但是本地没有数据的话，那么会用 BlockTransferService 与有数据的 BlockManager 建立连接，然后用 BlockTransferService 从远程 BlockManager 读取数据；例如，shuffle Read 操作中，很有可能要拉取的数据在本地没有，那么此时就会到远程有数据的节点上，找那个节点的 BlockManager 来拉取需要的数据。

只要使用 BlockManager 执行了数据增删改的操作，那么必须将 Block 的 BlockStatus 上报到 BlockManagerMaster，在 BlockManagerMaster 上会对指定 BlockManager 的 BlockManagerInfo 内部的 BlockStatus 进行增删改操作，从而达到元数据的维护功能。

7.2 Spark 共享变量底层实现

Spark 一个非常重要的特性就是共享变量。

默认情况下，如果在一个算子的函数中使用到了某个外部的变量，那么这个变量的值会被拷贝到每个 task 中，此时每个 task 只能操作自己的那份变量副本。如果多个 task 想要共享某个变量，那么这种方式是做不到的。

Spark 为此提供了两种共享变量，一种是 **Broadcast Variable**（广播变量），另一种是 **Accumulator**（累加变量）。**Broadcast Variable** 会将用到的变量，仅仅为每个节点拷贝一份，即每个 **Executor** 拷贝一份，更大的用途是优化性能，减少网络传输以及内存损耗。**Accumulator** 则可以让多个 **task** 共同操作一份变量，主要可以进行累加操作。**Broadcast Variable** 是共享读变量，**task** 不能去修改它，而 **Accumulator** 可以让多个 **task** 操作一个变量。

7.2.1 广播变量

广播变量允许编程者在每个 **Executor** 上保留外部数据的只读变量，而不是给每个任务发送一个副本。

每个 **task** 都会保存一份它所使用的外部变量的副本，当一个 **Executor** 上的多个 **task** 都使用一个大型外部变量时，对于 **Executor** 内存的消耗是非常大的，因此，我们可以将大型外部变量封装为广播变量，此时一个 **Executor** 保存一个变量副本，此 **Executor** 上的所有 **task** 共用此变量，不再是一个 **task** 单独保存一个副本，这在一定程度上降低了 **Spark** 任务的内存占用。

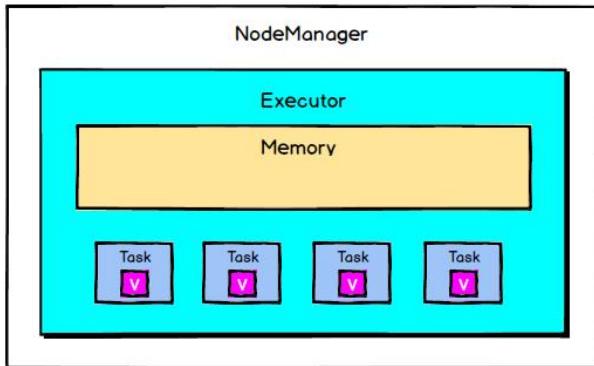


图 7-2 task 使用外部变量

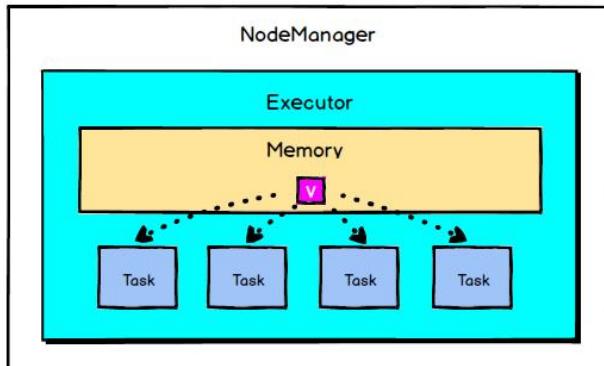


图 7-3 使用广播变量

Spark 还尝试使用高效的广播算法分发广播变量，以降低通信成本。

Spark 提供的 **Broadcast Variable** 是只读的，并且在每个 **Executor** 上只会有一个副本，而不会为每个 **task** 都拷贝一份副本，因此，它的最大作用，就是减少变量到各个节点的网络传输消耗，以及在各个节点上的内存消耗。此外，Spark 内部也使用了高效的广播算法来减少网络消耗。

可以通过调用 **SparkContext** 的 **broadcast()** 方法来针对每个变量创建广播变量。然后在算子的函数内，使用到广播变量时，每个 **Executor** 只会拷贝一份副本了，每个 **task** 可以使用广播变量的 **value()** 方法获取值。

在任务运行时，**Executor** 并不获取广播变量，当 **task** 执行到使用广播变量的代码时，会向 **Executor** 的内存中请求广播变量，如下图所示：

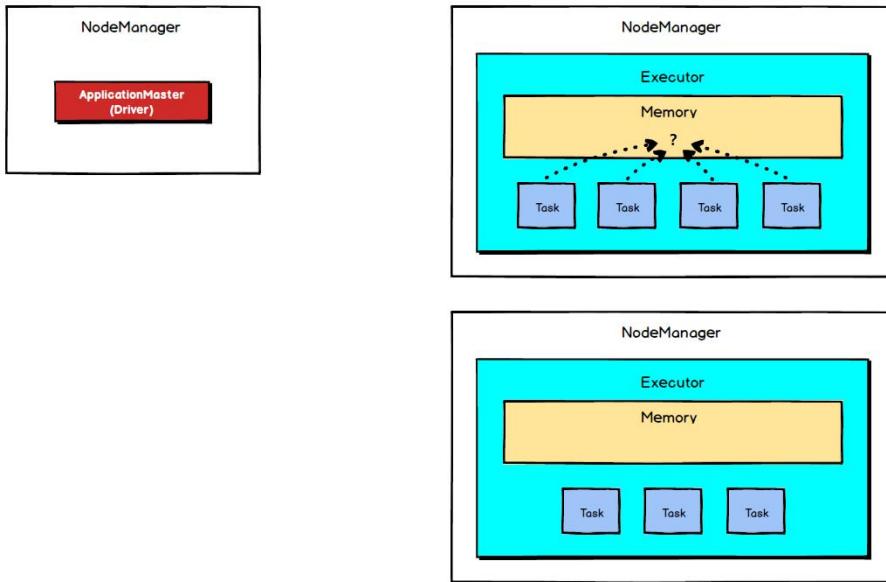


图 7-4 task 向 Executor 请求广播变量

之后 Executor 会通过 BlockManager 向 Driver 拉取广播变量，然后提供给 task 进行使用，如下图所示：

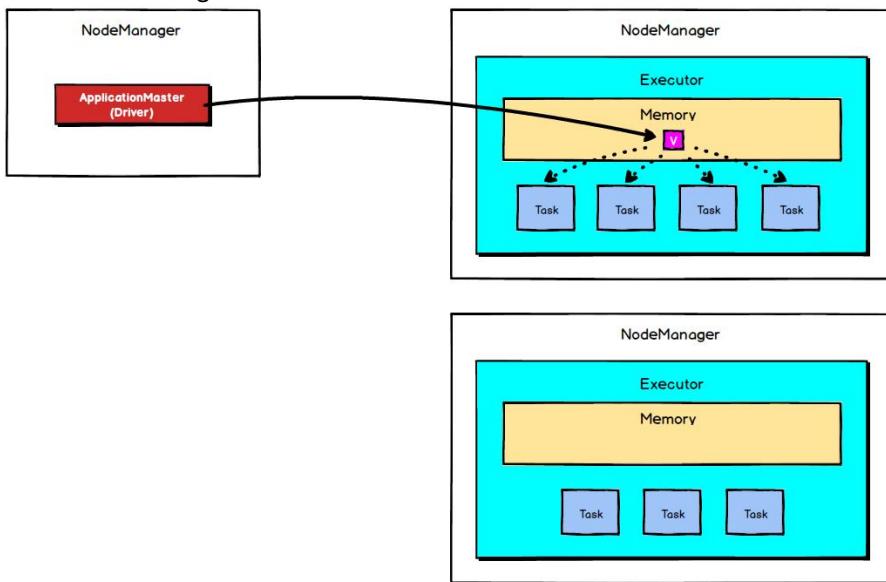


图 7-5 Executor 从 Driver 拉取广播变量

广播大变量是 Spark 中常用的基础优化方法，通过减少内存占用实现任务执行性能的提升。

7.2.2 累加器

累加器（accumulator）：Accumulator 是仅仅被相关操作累加的变量，因此可以在并行中被有效地支持。它们可用于实现计数器（如 MapReduce）或总和计数。

Accumulator 是存在于 Driver 端的，集群上运行的 task 进行 Accumulator 的累加，随后把值发到 Driver 端，在 Driver 端汇总（Spark UI 在 SparkContext 创建时被创建，即在 Driver 端被创建，因此它可以读取 Accumulator 的数值），由于 Accumulator 存在于 Driver 端，从节点读取不到 Accumulator 的数值。

Spark 提供的 Accumulator 主要用于多个节点对一个变量进行共享性的操作。Accumulator 只提供了累加的功能，但是却给我们提供了多个 task 对于同一个变量并行操作的功能，但是 task 只能对 Accumulator 进行累加操作，不能读取它的值，只有 Driver 程序可以读取 Accumulator 的值。

Accumulator 的底层原理如下图所示：

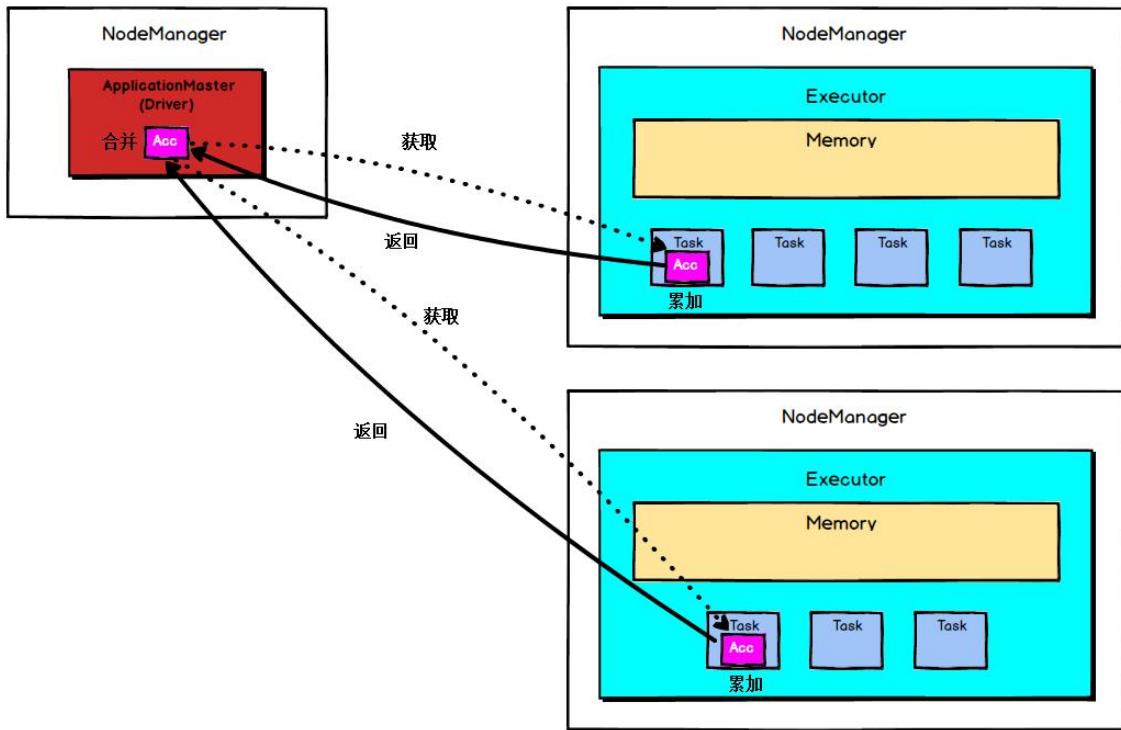


图 7-6 累加器原理

第 8 章 Spark 反压机制

8.1 产生背景

在默认情况下，Spark Streaming 通过 `receivers` (或者是 `Direct` 方式) 以生产者生产数据的速率接收数据。当 `batch processing time > batch interval` 的时候，也就是每个批次数据处理的时间要比 Spark Streaming 批处理间隔时间长；越来越多的数据被接收，但是数据的处理速度没有跟上，导致系统开始出现数据堆积，可能进一步导致 `Executor` 端出现 OOM 问题而出现失败的情况。

而在 Spark 1.5 版本之前，为了解决这个问题，对于 `Receiver-based` 数据接收器，我们可以通过配置 `spark.streaming.receiver.maxRate` 参数来限制每个 `receiver` 每秒最大可以接收的记录的数据；对于 `Direct Approach` 的数据接收，我们可以通过配置 `spark.streaming.kafka.maxRatePerPartition` 参数来限制每次作业中每个 Kafka 分区最多读取的记录条数。这种方法虽然可以通过限制接收速率，来适配当前的处理能力，但这种方式存在以下几个问题：

- 1) 我们需要事先估计好集群的处理速度以及消息数据的产生速度；
- 2) 这两种方式需要人工参与，修改完相关参数之后，我们需要手动重启 Spark Streaming 应用程序；
- 3) 如果当前集群的处理能力高于我们配置的 `maxRate`，而且 `producer` 产生的数据高于 `maxRate`，这会导致集群资源利用率低下，而且也会导致数据不能够及时处理。

Active Batches (4)

Batch Time	Input Size	Scheduling Delay	Processing Time	Output Ops: Succeeded/Total	Status
2018/05/25 13:20:00	8047011 records	-	-	0/1	queued
2018/05/25 13:00:00	7454116 records	-	-	0/1	queued
2018/05/25 12:40:00	7310575 records	-	-	0/1	queued
2018/05/25 12:20:00	8212326 records	1 ms	-	微信公众号: iteblog_hadoop 0/1	processing

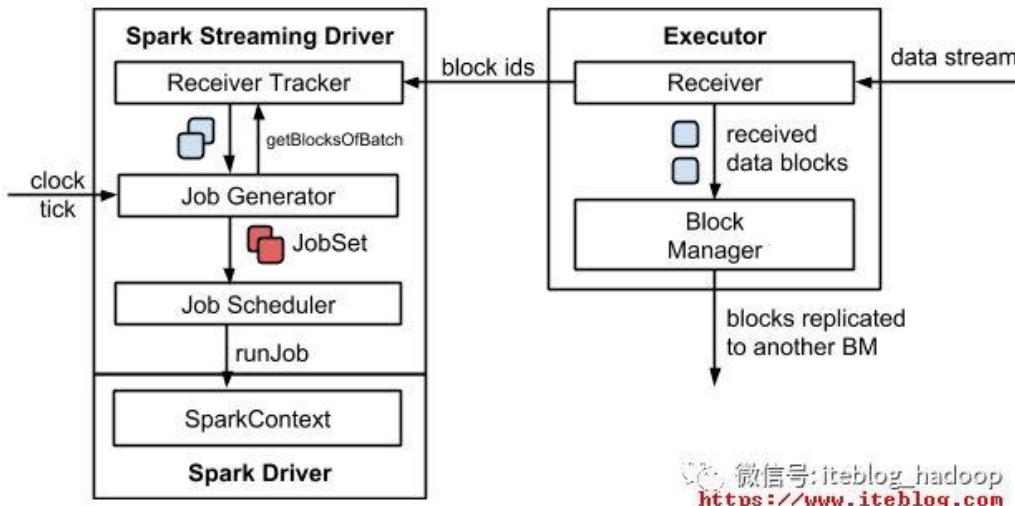
8.2 反压机制

那么有没有可能不需要人工干预，Spark Streaming 系统自动处理这些问题呢？当然有了！Spark 1.5 引入了反压（Back Pressure）机制，其通过动态收集系统的一些数据来自动地适配集群数据处理能力。详细的记录请参见 SPARK-7398 里面的说明。

8.2.1 Spark1.5 之前版本

在 Spark 1.5 版本之前，Spark Streaming 的体系结构如下所示：

Spark 数据堆积

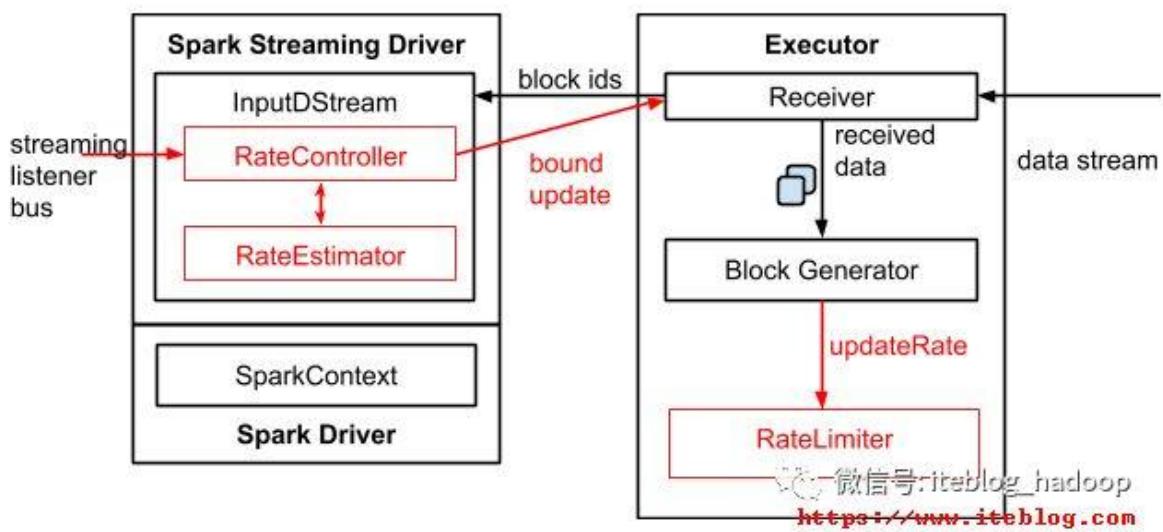


微信号:iteblog_hadoop
<https://www.iteblog.com>

- 1) 数据是源源不断的通过 receiver 接收，当数据被接收后，其将这些数据存储在 Block Manager 中；为了不丢失数据，其还将数据备份到其他的 Block Manager 中；
- 2) Receiver Tracker 收到被存储的 Block IDs，然后其内部会维护一个时间到这些 block IDs 的关系；
- 3) Job Generator 会每隔 batchInterval 的时间收到一个事件，其会生成一个 JobSet；
- 4) Job Scheduler 运行上面生成的 JobSet。

8.2.1 Spark1.5 之后版本

Spark Streaming 1.5 之后的体系结构



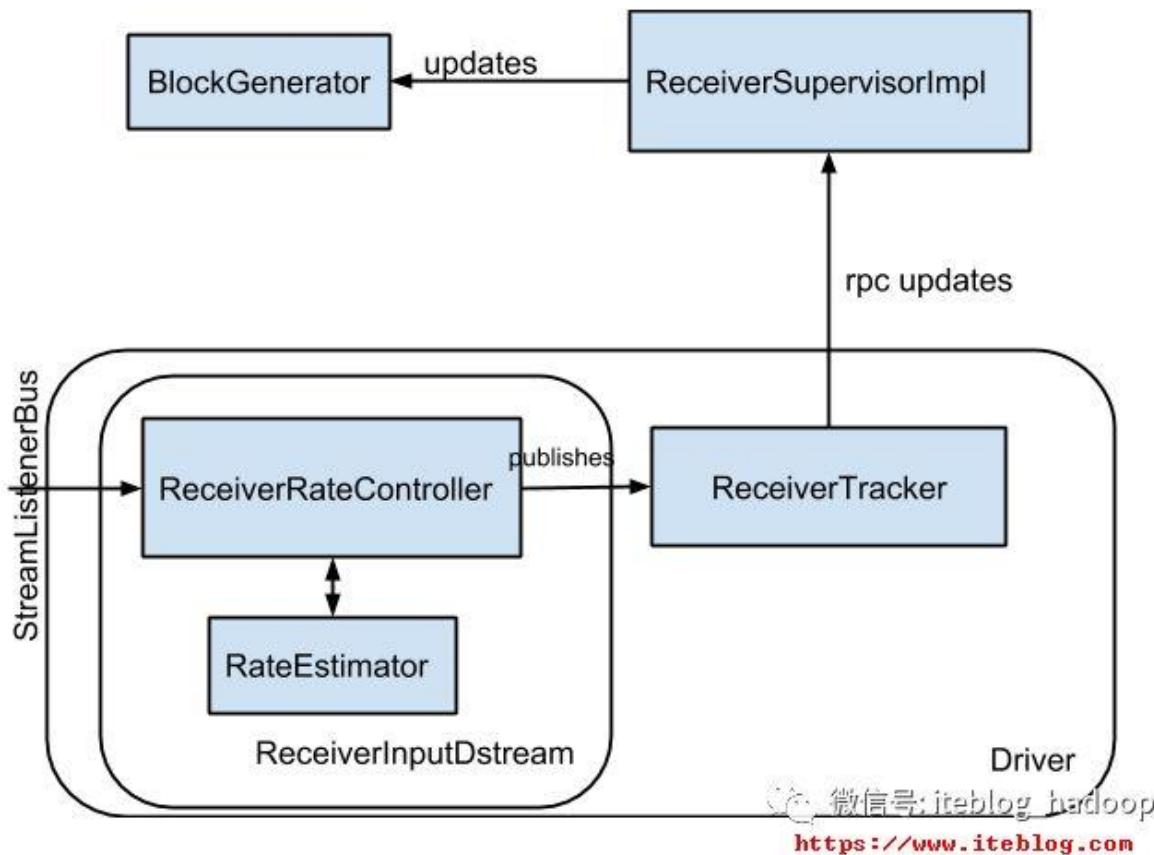
微信号:iteblog_hadoop
<https://www.iteblog.com>

为了实现自动调节数据的传输速率，在原有的架构上新增了一个名为 RateController 的组件，这个组件继承自 StreamingListener，其监听所有作业的 onBatchCompleted 事件，并且基于 processingDelay、schedulingDelay、当前 Batch 处理的记录条数以及处理完成事件来估算出一个速率；这个速率主要用于更新流每秒能够处理的最大记录的条数。速率估算器（RateEstimator）可以又多种实现，不过目前的 Spark 2.2 只实现了基于 PID 的速率估算器。

InputDStreams 内部的 RateController 里面会存下计算好的最大速率，这个速率会在处理完 onBatchCompleted 事件之后将计算好的速率推送到 ReceiverSupervisorImpl，这样接收器就知道下一步应该接收多少数据了。

如果用户配置了 spark.streaming.receiver.maxRate 或 spark.streaming.kafka.maxRatePerPartition，那么最后到底接收多少数据取决于三者的最小值。也就是说每个接收器或者每个 Kafka 分区每秒处理的数据不会超过 spark.streaming.receiver.maxRate 或 spark.streaming.kafka.maxRatePerPartition 的值。

详细的过程如下图所示：



8.3 反压机制的使用

在 **Spark** 启用反压机制很简单，只需要将 `spark.streaming.backpressure.enabled` 设置为 `true` 即可，这个参数的默认值为 `false`。反压机制还涉及以下几个参数，包括文档中没有列出来的：

- 1) `spark.streaming.backpressure.initialRate`: 启用反压机制时每个接收器接收第一批数据的初始最大速率。默认值没有设置。
- 2) `spark.streaming.backpressure.rateEstimator`: 速率估算器类，默认值为 `pid`，目前 **Spark** 只支持这个，大家可以根据自己的需要实现。
- 3) `spark.streaming.backpressure.pid.proportional`: 用于响应错误的权重（最后批次和当前批次之间的更改）。默认值为 1，只能设置成非负值。weight for response to "error" (change between last batch and this batch)
- 4) `spark.streaming.backpressure.pid.integral`: 错误积累的响应权重，具有抑制作用（有效阻尼）。默认值为 0.2，只能设置成非负值。weight for the response to the accumulation of error. This has a dampening effect.
- 5) `spark.streaming.backpressure.pid.derived`: 对错误趋势的响应权重。这可能会引起 `batch size` 的波动，可以帮助快速增加/减少容量。默认值为 0，只能设置成非负值。weight for the response to the trend in error. This can cause arbitrary/noise-induced fluctuations in batch size, but can also help react quickly to increased/reduced capacity.
- 6) `spark.streaming.backpressure.pid.minRate`: 可以估算的最低费率是多少，默认值为 100，只能设置成非负值。

Spark 调优

第 1 章 Spark 性能调优

1.1 常规性能调优

1.1.1 最优资源配置

Spark 性能调优的第一步，就是为任务分配更多的资源，在一定范围内，增加资源的分配与性能的提升是成正比的，实现了最优的资源配置后，在此基础上再考虑进行后面论述的性能调优策略。

资源的分配在使用脚本提交 Spark 任务时进行指定，标准的 Spark 任务提交脚本如代码清单 2-1 所示：

```
/usr/opt/modules/spark/bin/spark-submit \
--class com.atguigu.spark.Analysis \
--num-executors 80 \
--driver-memory 6g \
--executor-memory 6g \
--executor-cores 3 \
/usr/opt/modules/spark/jar/spark.jar \
```

代码清单 2-1 标准 Spark 提交脚本

可以进行分配的资源如表 2-1 所示：

名称	说明
--num-executors	配置 Executor 的数量
--driver-memory	配置 Driver 内存（影响不大）
--executor-memory	配置每个 Executor 的内存大小
--executor-cores	配置每个 Executor 的 CPU core 数量

表 2-1 可分配资源表

调节原则：尽量将任务分配的资源调节到可以使用的资源的最大限度。

对于具体资源的分配，我们分别讨论 Spark 的两种 Cluster 运行模式：

第一种是 Spark Standalone 模式，你在提交任务前，一定知道或者可以从运维部门获取到你可以使用的资源情况，在编写 submit 脚本的时候，就根据可用的资源情况进行资源的分配，比如说集群有 15 台机器，每台机器为 8 G 内存，2 个 CPU core，那么就指定 15 个 Executor，每个 Executor 分配 8G 内存，2 个 CPU core。

第二种是 Spark Yarn 模式，由于 Yarn 使用资源队列进行资源的分配和调度，在编写 submit 脚本的时候，就根据 Spark 作业要提交到的资源队列，进行资源的分配，比如资源队列有 400G 内存，100 个 CPU core，那么指定 50 个 Executor，每个 Executor 分配 8G 内存，2 个 CPU core。

对表 2-1 中的各项资源进行了调节后，得到的性能提升如表 2-2 所示：

名称	解析
增加 Executor 的个数	在资源允许的情况下，增加 Executor 的个数可以提高执行 task 的并行度。比如有 4 个 Executor，每个 Executor 有 2 个 CPU core，那么可以并行执行 8 个 task，如果将 Executor 的个数增加到 8 个（资源允许的情况下），那么可以并行执行 16 个 task，此时的并行能力提升了一倍。
增加每个 Executor 的 CPU core 个数	在资源允许的情况下，增加每个 Executor 的 Cpu core 个数，可以提高执行 task 的并行度。比如有 4 个 Executor，每个 Executor 有 2 个 CPU core，那么可以并行执行 8 个 task，如果将每个 Executor 的 CPU core 个数增加到 4 个（资源允许的情况下），那么可以并行执行 16 个 task，此时的并行能力提升了一倍。
增加每个 Executor 的内存量	在资源允许的情况下，增加每个 Executor 的内存量以后，对性能的提升有三点： <ol style="list-style-type: none">1. 可以缓存更多的数据（即对 RDD 进行 cache），写入磁盘的数据相应减少，甚至可以不写入磁盘，减少了可能的磁盘 IO；2. 可以为 shuffle 操作提供更多内存，即有更多空间来存放 reduce 端拉取的数据，写入磁盘的数据相应减少，甚至可以不写入磁盘，减少了可能的磁盘 IO；3. 可以为 task 的执行提供更多内存，在 task 的执行过程中可能创建很多对象，内存较小时会引发频繁的 GC，增加内存后，可以避免频繁的 GC，提升整体性能。

表 2-2 资源调节后的性能提升

补充：生产环境 Spark submit 脚本配置

```
/usr/local/spark/bin/spark-submit \
```

```
--class com.atguigu.spark.WordCount \
--num-executors 80 \
--driver-memory 6g \
--executor-memory 6g \
--executor-cores 3 \
--master yarn-cluster \
--queue root.default \
--conf spark.yarn.executor.memoryOverhead=2048 \
--conf spark.core.connection.ack.wait.timeout=300 \
/usr/local/spark/spark.jar
```

参数配置参考值：

- num-executors: 50~100
- driver-memory: 1G~5G
- executor-memory: 6G~10G
- executor-cores: 3
- master: 实际生产环境一定使用 yarn-cluster

1.1.2 RDD 优化

1.2.1 RDD 复用

在对 RDD 进行算子时，要避免相同的算子和计算逻辑之下对 RDD 进行重复的计算，如图 2-1 所示：

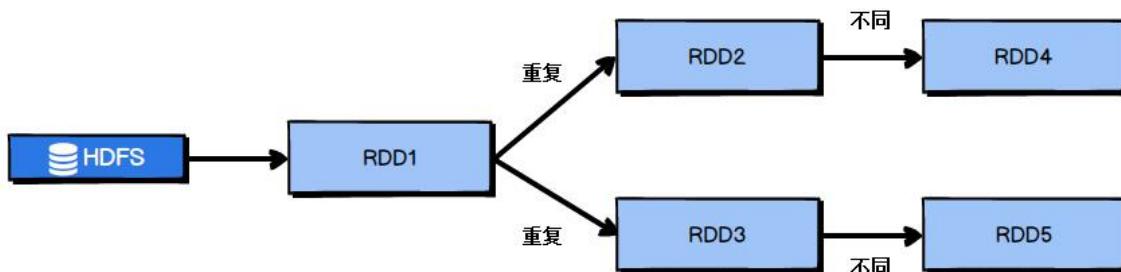


图 2-1 RDD 的重复计算

对图 2-1 中的 RDD 计算架构进行修改，得到如图 2-2 所示的优化结果：

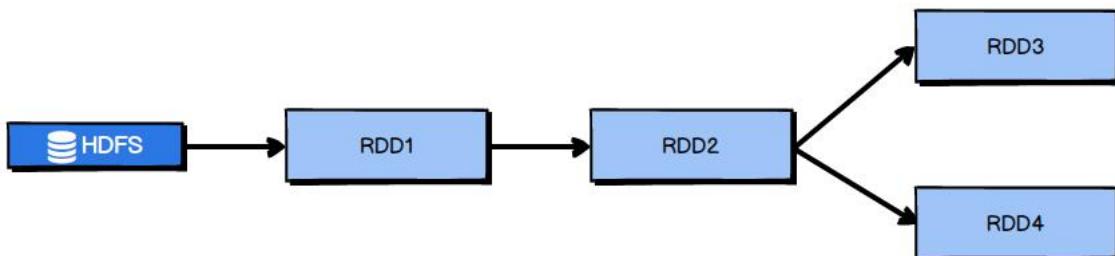


图 2-2 RDD 架构优化

1.2.2 RDD 持久化

在 Spark 中，当多次对同一个 RDD 执行算子操作时，每一次都会对这个 RDD 以之前的父 RDD 重新计算一次，这种情况是必须要避免的，对同一个 RDD 的重复计算是对资源的极大浪费，因此，必须对多次使用的 RDD 进行持久化，通过持久化将公共 RDD 的数据缓存到内存/磁盘中，之后对于公共 RDD 的计算都会从内存/磁盘中直接获取 RDD 数据。

对于 RDD 的持久化，有两点需要说明：

第一，RDD 的持久化是可以进行序列化的，当内存无法将 RDD 的数据完整的进行存放的时候，可以考虑使用序列化的方式减小数据体积，将数据完整存储在内存中。

第二，如果对于数据的可靠性要求很高，并且内存充足，可以使用副本机制，对 RDD 数据进行持久化。当持久化启用了复本机制时，对于持久化的每个数据单元都存储一个副本，放在其他节点上面，由此实现数据的容错，一旦一个副本数据丢失，不需要重新计算，还可以使用另外一个副本。

1.2.3 RDD 瘦身

获取到初始 RDD 后，应该考虑尽早地过滤掉不需要的数据，进而减少对内存的占用，从而提升 Spark 作业的运行效率。

1.1.3 并行度调节

Spark 作业中的并行度指各个 stage 的 task 的数量。

如果并行度设置不合理而导致并行度过低，会导致资源的极大浪费，例如 20 个 Executor，每个 Executor 分配 3 个 CPU core，而 Spark 作业有 40 个 task，这样每个 Executor 分配到的 task 个数是 2 个，这就使得每个 Executor 有一个 CPU core 空闲，导致资源的浪费。

理想的并行度设置，应该是让并行度与资源相匹配，简单来说就是在资源允许的前提下，并行度要设置的尽可能大，达到可以充分利用集群资源。合理的设置并行度，可以提升整个 Spark 作业的性能和运行速度。

Spark 官方推荐，task 数量应该设置为 Spark 作业总 CPU core 数量的 2~3 倍。之所以没有推荐 task 数量与 CPU core 总数相等，是因为 task 的执行时间不同，有的 task 执行速度快而有的 task 执行速度慢，如果 task 数量与 CPU core 总数相等，那么执行快的 task 执行完成后，会出现 CPU core 空闲的情况。如果 **task 数量设置为 CPU core 总数的 2~3 倍**，那么一个 task 执行完毕后，CPU core 会立刻执行下一个 task，降低了资源的浪费，同时提升了 Spark 作业运行的效率。

Spark 作业并行度的设置如代码清单 2-2 所示：

```
val conf = new SparkConf().set("spark.default.parallelism", "500")
```

代码清单 2-2 Spark 作业并行度设置

1.1.4 广播变量

默认情况下，task 中的算子中如果使用了外部的变量，每个 task 都会获取一份变量的副本，这就造成了内存的极大消耗。一方面，如果后续对 RDD 进行持久化，可能就无法将 RDD 数据存入内存，只能写入磁盘，磁盘 IO 将会严重消耗性能；另一方面，task 在创建对象的时候，也许会发现堆内存无法存放新创建的对象，这就会导致频繁的 GC，GC 会导致工作线程停止，进而导致 Spark 暂停工作一段时间，严重影响 Spark 性能。

假设当前任务配置了 20 个 Executor，指定 500 个 task，有一个 20M 的变量被所有 task 共用，此时会在 500 个 task 中产生 500 个副本，耗费集群 10G 的内存，如果使用了广播变量，那么每个 Executor 保存一个副本，一共消耗 400M 内存，内存消耗减少了 5 倍。

广播变量在每个 Executor 保存一个副本，此 Executor 的所有 task 共用此广播变量，这让变量产生的副本数量大大减少。

在初始阶段，广播变量只在 Driver 中有一份副本。task 在运行的时候，想要使用广播变量中的数据，此时首先会在自己本地的 Executor 对应的 BlockManager 中尝试获取变量，如果本地没有，BlockManager 就会从 Driver 或者其他节点的 BlockManager 上远程拉取变量的副本，并由本地的 BlockManager 进行管理；之后此 Executor 的所有 task 都会直接从本地的 BlockManager 中获取变量。

1.1.5 Kryo 序列化

默认情况下，Spark 使用 Java 的序列化机制。Java 的序列化机制使用方便，不需要额外的配置，在算子中使用的变量实现 Serializable 接口即可，但是 Java 序列化机制的效率不高，序列化速度慢并且序列化后的数据所占用的空间依然较大。

Kryo 序列化机制比 Java 序列化机制性能提高 10 倍左右，Spark 之所以没有默认使用 Kryo 作为序列化类库，是因为它不支持所有对象的序列化，同时 Kryo 需要用户在使用前注册需要序列化的类型，不够方便，但从 Spark 2.0.0 版本开始，简单类型、简单类型数组、字符串类型的 Shuffling RDDs 已经默认使用 Kryo 序列化方式了。

Kryo 序列化注册方式的实例代码如代码清单 2-3 所示：

```
public class MyKryoRegistrar implements KryoRegistrar
{
    @Override
    public void registerClasses(Kryo kryo)
    {
        kryo.register(StartupReportLogs.class);
    }
}
```

代码清单 2-3 Kryo 序列化机制配置代码

配置 Kryo 序列化方式的实例代码如代码清单 2-4 所示：

```
//创建 SparkConf 对象
val conf = new SparkConf().setMaster(...).setAppName(...)
//使用 Kryo 序列化库，如果要使用 Java 序列化库，需要把该行屏蔽掉
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
//在 Kryo 序列化库中注册自定义的类集合，如果要使用 Java 序列化库，需要把该行屏蔽掉
conf.set("spark.kryo.registrator", "atguigu.com.MyKryoRegistrar");
```

代码清单 2-4 Kryo 序列化机制配置代码

1.1.6 调节本地化等待时长

Spark 作业运行过程中，Driver 会对每一个 stage 的 task 进行分配。根据 Spark 的 task 分配算法，Spark 希望 task 能够运行在它要计算的数据算在的节点（数据本地化思想），这样就可以避免数据的网络传输。通常来说，task 可能不会被分配到它处理的数据所在的节点，因为这些节点可用的资源可能已经用尽，此时，Spark 会等待一段时间，默认 3s，如果等待指定时间后仍然无法在指定节点运行，那么会自动降级，尝试将 task 分配到比较差的本地化级别所对应的节点上，比如将 task 分配到离它要计算的数据比较近的一个节点，然后进行计算，如果当前级别仍然不行，那么继续降级。

当 task 要处理的数据不在 task 所在节点上时，会发生数据的传输。task 会通过所在节点的 BlockManager 获取数据，BlockManager 发现数据不在本地时，会通过网络传输组件从数据所在节点的 BlockManager 处获取数据。

网络传输数据的情况是我们不愿意看到的，大量的网络传输会严重影响性能，因此，我们希望通过调节本地化等待时长，如果在等待时长这段时间内，目标节点处理完成了一部分 task，那么当前的 task 将有机会得到执行，这样就能够改善 Spark 作业的整体性能。

Spark 的本地化等级如表 2-3 所示：

表 2-3 Spark 本地化等级

名称	解析
PROCESS_LOCAL	进程本地化，task 和数据在同一个 Executor 中，性能最好。（一个节点可以起 1 到多个 executor，一个 executor 有若干个 core，一个 core 一次执行一个 task）
NODE_LOCAL	节点本地化，task 和数据在同一个节点中，但是 task 和数据不在同一个 Executor 中，数据需要在进程间进行传输。
RACK_LOCAL	机架本地化，task 和数据在同一个机架的两个节点上，数据需要通过网络在节点之间进行传输。
NO_PREF	对于 task 来说，从哪里获取都一样，没有好坏之分。
ANY	task 和数据可以在集群的任何地方，而且不在一个机架中，性能最差。

在 Spark 项目开发阶段，可以使用 client 模式对程序进行测试，此时，可以在本地看到比较全的日志信息，日志信息中有明确的 task 数据本地化的级别，如果大部分都是 PROCESS_LOCAL，那么就无需进行调节，但是如果发现很多的级别都是 NODE_LOCAL、ANY，那么需要对本地化的等待时长进行调节，通过延长本地化等待时长，看看 task 的本地化级别有没有提升，并观察 Spark 作业的运行时间有没有缩短。

注意，过犹不及，不要将本地化等待时长延长地过长，导致因为大量的等待时长，使得 Spark 作业的运行时间反而增加了。

Spark 本地化等待时长的设置如代码清单 2-5 所示：

```
val conf = new SparkConf().set("spark.locality.wait", "6")
```

代码清单 2-5 Spark 本地化等待时长设置示例

1.2 算子调优

1.2.1 mapPartitions

普通的 map 算子对 RDD 中的每一个元素进行操作，而 mapPartitions 算子对 RDD 中每一个分区进行操作。如果是普通的 map 算子，假设一个 partition 有 1 万条数据，那么 map 算子中的 function 要执行 1 万次，也就是对每个元素进行操作。

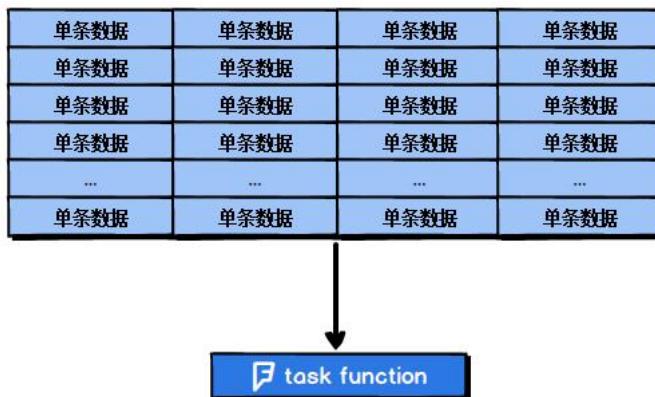


图 2-3 map 算子

如果是 mapPartition 算子，由于一个 task 处理一个 RDD 的 partition，那么一个 task 只会执行一次 function，function 一次接收所有的 partition 数据，效率比较高。

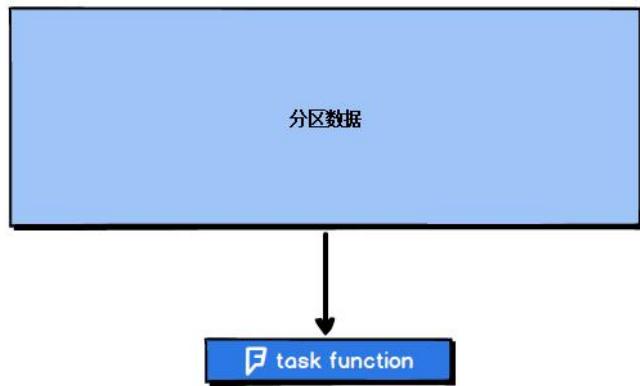


图 2-4 mapPartitions 算子

比如，当要把 RDD 中的所有数据通过 JDBC 写入数据，如果使用 map 算子，那么需要对 RDD 中的每一个元素都创建一个数据库连接，这样对资源的消耗很大，如果使用 mapPartitions 算子，那么针对一个分区的数据，只需要建立一个数据库连接。

mapPartitions 算子也存在一些缺点：对于普通的 map 操作，一次处理一条数据，如果在处理了 2000 条数据后内存不足，那么可以将已经处理完的 2000 条数据从内存中垃圾回收掉；但是如果使用 mapPartitions 算子，但数据量非常大时，function 一次处理一个分区的数据，如果一旦内存不足，此时无法回收内存，就可能会 OOM，即内存溢出。

因此，mapPartitions 算子适用于数据量不是特别大的时候，此时使用 mapPartitions 算子对性能的提升效果还是不错的。（当数据量很大的时候，一旦用 mapPartitions 算子就会直接 OOM）

在项目中，应该首先估算一下 RDD 的数据量、每个 partition 的数据量，以及分配给每个 Executor 的内存资源，如果资源允许，可以考虑使用 mapPartitions 算子代替 map。

1.2.2 foreachPartition

foreachPartition 优化数据库操作

在生产环境中，通常使用 foreachPartition 算子来完成数据库的写入，通过 foreachPartition 算子的特性，可以优化写数据库的性能。

如果使用 foreach 算子完成数据库的操作，由于 foreach 算子是遍历 RDD 的每条数据，因此，每条数据都会建立一个数据库连接，这是对资源的极大浪费，因此，对于写数据库操作，我们应当使用 foreachPartition 算子。

与 mapPartitions 算子非常相似，foreachPartition 是将 RDD 的每个分区作为遍历对象，一次处理一个分区的数据，也就是说，如果涉及数据库的相关操作，一个分区的数据只需要创建一次数据库连接，如图 2-5 所示：

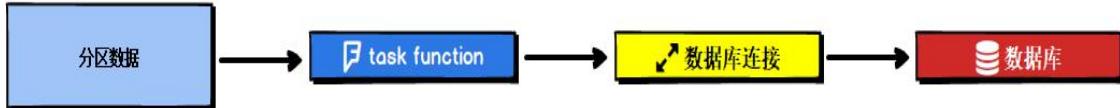


图 2-5 foreachPartition 算子

使用了 foreachPartition 算子后，可以获得以下的性能提升：

1. 对于我们写的 function 函数，一次处理一整个分区的数据；
2. 对于一个分区内的数据，创建唯一的数据库连接；
3. 只需要向数据库发送一次 SQL 语句和多组参数；

在生产环境中，全部都会使用 foreachPartition 算子完成数据库操作。foreachPartition 算子存在一个问题，与 mapPartitions 算子类似，如果一个分区的数据量特别大，可能会造成 OOM，即内存溢出。

1.2.3 filter 与 coalesce 的配合使用

在 Spark 任务中我们经常会使用 filter 算子完成 RDD 中数据的过滤，在任务初始阶段，从各个分区中加载到的数据量是相近的，但是一旦进过 filter 过滤后，每个分区的数据量有可能会存在较大差异，如图 2-6 所示：

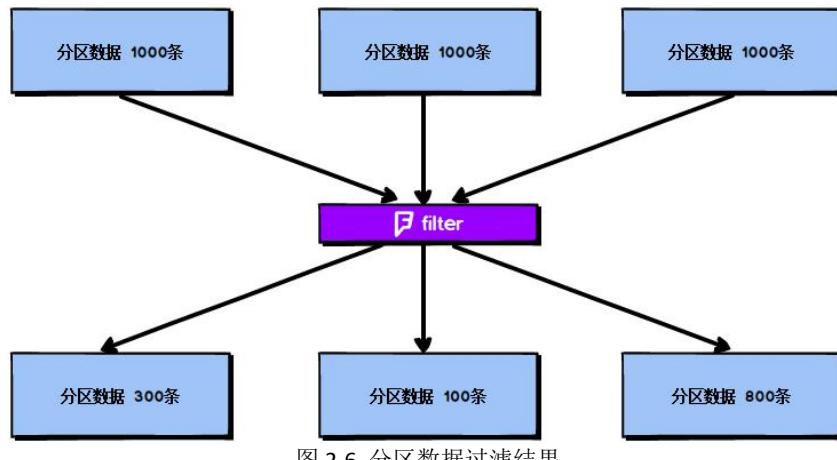


图 2-6 分区数据过滤结果

根据图 2-6 我们可以发现两个问题：

1. 每个 `partition` 的数据量变小了，如果还按照之前与 `partition` 相等的 `task` 个数去处理当前数据，有点浪费 `task` 的计算资源；
2. 每个 `partition` 的数据量不一样，会导致后面的每个 `task` 处理每个 `partition` 数据的时候，每个 `task` 要处理的数据量不同，这很有可能导致数据倾斜问题。

如图 2-6 所示，第二个分区的数据过滤后只剩 100 条，而第三个分区的数据过滤后剩下 800 条，在相同的处理逻辑下，第二个分区对应的 `task` 处理的数据量与第三个分区对应的 `task` 处理的数据量差距达到了 8 倍，这也会导致运行速度可能存在数倍的差距，这也就是数据倾斜问题。

针对上述的两个问题，我们分别进行分析：

1. 针对第一个问题，既然分区的数据量变小了，我们希望可以对分区数据进行重新分配，比如将原来 4 个分区的数据转化到 2 个分区中，这样只需要用后面的两个 `task` 进行处理即可，避免了资源的浪费。
2. 针对第二个问题，解决方法和第一个问题的解决方法非常相似，对分区数据重新分配，让每个 `partition` 中的数据量差不多，这就避免了数据倾斜问题。

那么具体应该如何实现上面的解决思路？我们需要 `coalesce` 算子。

`repartition` 与 `coalesce` 都可以用来进行重分区，其中 `repartition` 只是 `coalesce` 接口中 `shuffle` 为 `true` 的简易实现，`coalesce` 默认情况下不进行 `shuffle`，但是可以通过参数进行设置。

假设我们希望将原本的分区个数 A 通过重新分区变为 B，那么有以下几种情况：

1. $A > B$ （多数分区合并为少数分区）

① A 与 B 相差值不大

此时使用 `coalesce` 即可，无需 `shuffle` 过程。

② A 与 B 相差值很大

此时可以使用 `coalesce` 并且不启用 `shuffle` 过程，但是会导致合并过程性能低下，所以推荐设置 `coalesce` 的第二个参数为 `true`，即启动 `shuffle` 过程。

2. $A < B$ （少数分区分解为多数分区）

此时使用 `repartition` 即可，如果使用 `coalesce` 需要将 `shuffle` 设置为 `true`，否则 `coalesce` 无效。

我们可以在 `filter` 操作之后，使用 `coalesce` 算子针对每个 `partition` 的数据量各不相同的情况，压缩 `partition` 的数量，而且让每个 `partition` 的数据量尽量均匀紧凑，以便于后面的 `task` 进行计算操作，在某种程度上能够在一定程度上提升性能。

注意：`local` 模式是进程内模拟集群运行，已经对并行度和分区数量有了一定的内部优化，因此不用去设置并行度和分区数量。

1.2.4 repartition

`Repartition` 解决 Spark SQL 低并行度问题：

在第一节的常规性能调优中我们讲解了并行度的调节策略，但是并行度的设置对于 Spark SQL 是不生效的，用户设置的并行度只对于 Spark SQL 以外的所有 Spark 的 stage 生效。

Spark SQL 的并行度不允许用户自己指定，Spark SQL 自己会默认根据 hive 表对应的 HDFS 文件的 `split` 个数自动设置 Spark SQL 所在的那个 stage 的并行度，用户自己通过 `spark.default.parallelism` 参数指定的并行度，只会在没 Spark SQL 的 stage 中生效。

由于 Spark SQL 所在 stage 的并行度无法手动设置，如果数据量较大并且此 stage 中后续的 transformation 操作有着复杂的业务逻辑，而 Spark SQL 自动设置的 task 数量很少，这就意味着每个 task 要处理为数不少的数据量，然

后还要执行非常复杂的处理逻辑，这就可能表现为第一个有 Spark SQL 的 stage 速度很慢，而后续的没有 Spark SQL 的 stage 运行速度非常快。

为了解决 Spark SQL 无法设置并行度和 task 数量的问题，我们可以使用 `repartition` 算子。

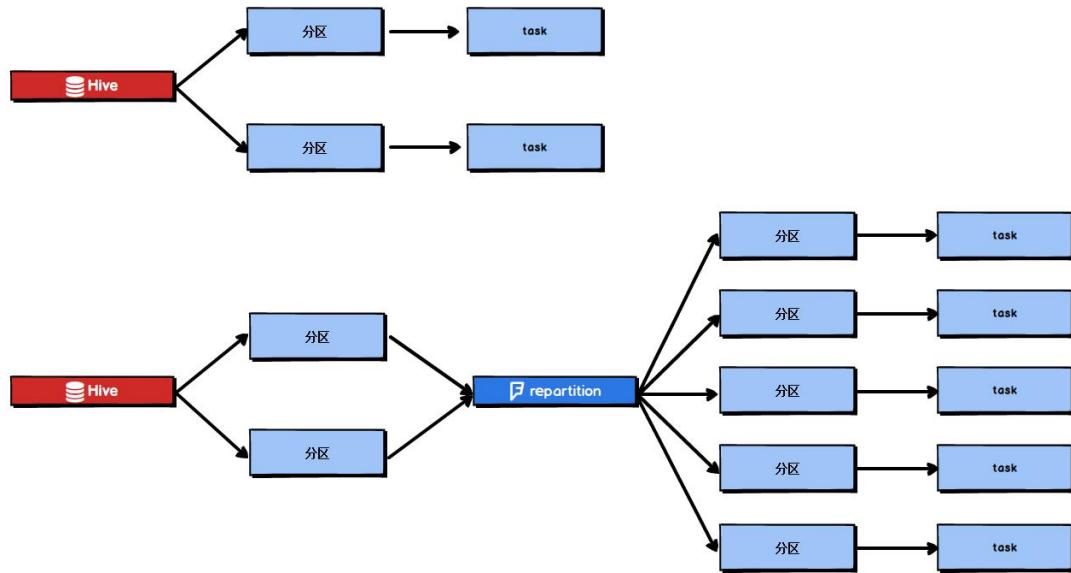


图 2-7 repartition 算子使用前后对比图

Spark SQL 这一步的并行度和 task 数量肯定是没有办法去改变了，但是对于 Spark SQL 查询出来的 RDD，立即使用 `repartition` 算子去重新进行分区，可以重新分区为多个 partition，从 `repartition` 之后的 RDD 操作。由于不再设计 Spark SQL，因此 stage 的并行度就会等于你手动设置的值，这样就避免了 Spark SQL 所在的 stage 只能用少量的 task 去处理大量数据并执行复杂的算法逻辑。使用 `repartition` 算子的前后对比如图 2-7 所示。

1.2.5 reduceByKey 本地聚合

`reduceByKey` 相较于普通的 `shuffle` 操作一个显著的特点就是会进行 `map` 端的本地聚合，`map` 端会先对本地的数据进行 `combine` 操作，然后将数据写入给下个 stage 的每个 task 创建的文件中，也就是在 `map` 端，对每一个 key 对应的 value，执行 `reduceByKey` 算子函数。`reduceByKey` 算子的执行过程如图 2-8 所示：

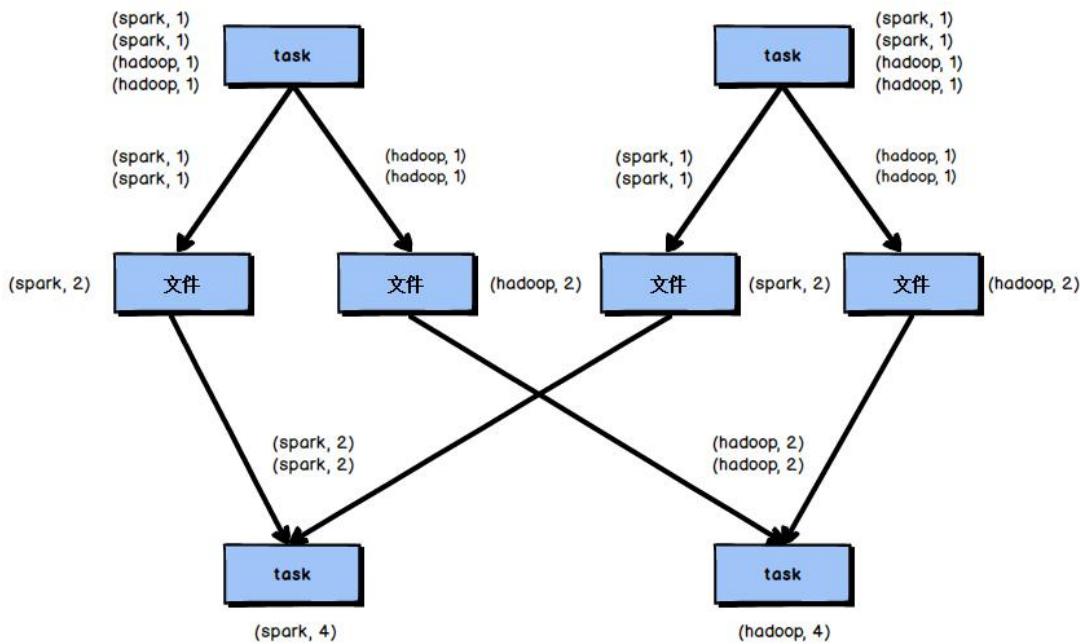


图 2-8 reduceByKey 算子执行过程

使用 `reduceByKey` 对性能的提升如下：

1. 本地聚合后，在 `map` 端的数据量变少，减少了磁盘 IO 和对磁盘空间的占用；
2. 本地聚合后，下一个 stage 拉取的数据量变少，减少了网络传输的数据量；
3. 本地聚合后，在 `reduce` 端进行数据缓存的内存占用减少；
4. 本地聚合后，在 `reduce` 端进行聚合的数据量减少。

基于 `reduceByKey` 的本地聚合特征，我们应该考虑使用 `reduceByKey` 代替其他的 `shuffle` 算子，例如 `groupByKey`。`reduceByKey` 与 `groupByKey` 的运行原理如图 2-9 和图 2-10 所示：

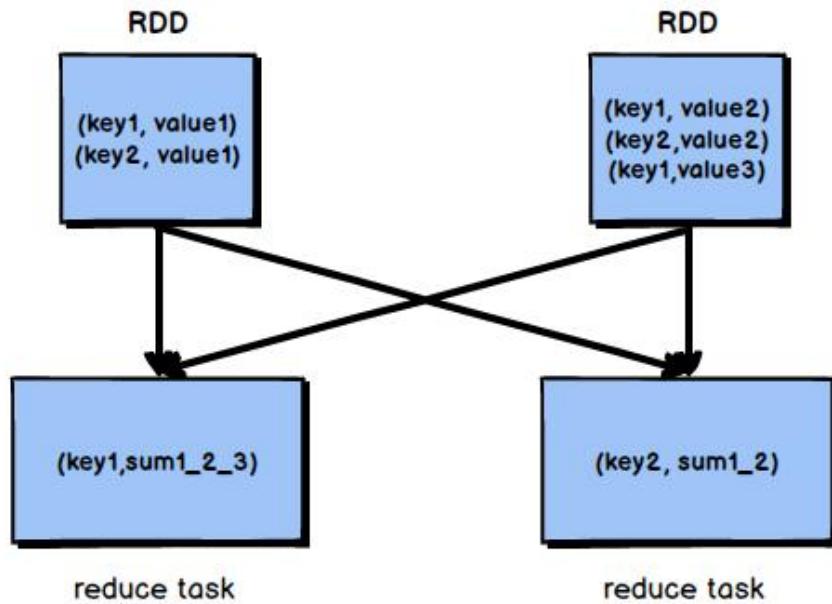


图 2-9 groupByKey 原理

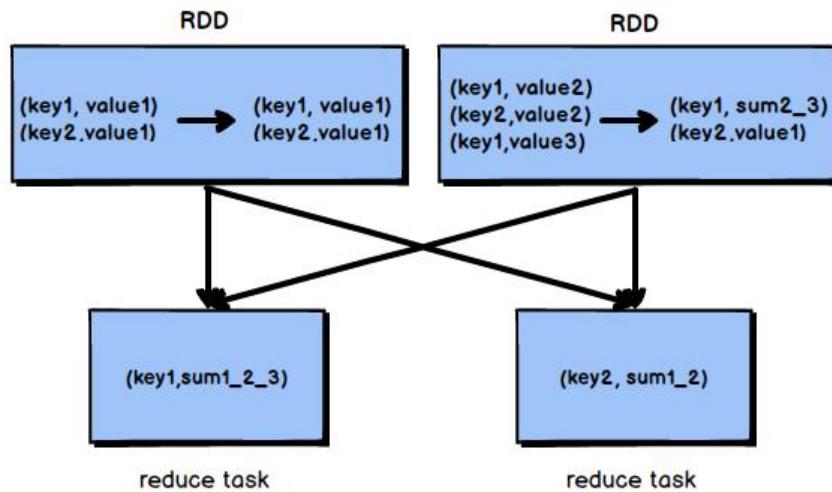


图 2-10 reduceByKey 原理

根据上图可知，groupByKey 不会进行 map 端的聚合，而是将所有 map 端的数据 shuffle 到 reduce 端，然后在 reduce 端进行数据的聚合操作。由于 reduceByKey 有 map 端聚合的特性，使得网络传输的数据量减小，因此效率要明显高于 groupByKey。

1.3 Shuffle 调优

1.3.1 调节 map 端缓冲区大小

在 Spark 任务运行过程中，如果 shuffle 的 map 端处理的数据量比较大，但是 map 端缓冲的大小是固定的，可能会出现 map 端缓冲数据频繁 spill 溢写到磁盘文件中的情况，使得性能非常低下，通过调节 map 端缓冲的大小，可以避免频繁的磁盘 IO 操作，进而提升 Spark 任务的整体性能。

map 端缓冲的默认配置是 32KB，如果每个 task 处理 640KB 的数据，那么会发生 $640/32 = 20$ 次溢写，如果每个 task 处理 64000KB 的数据，机会发生 $64000/32=2000$ 此溢写，这对于性能的影响是非常严重的。

map 端缓冲的配置方法如代码清单 2-7 所示：

代码清单 2-7 map 端缓冲配置

```
val conf = new SparkConf().set("spark.shuffle.file.buffer", "64")
```

1.3.2 调节 reduce 端拉取数据缓冲区大小

Spark Shuffle 过程中，shuffle reduce task 的 buffer 缓冲区大小决定了 reduce task 每次能够缓冲的数据量，也就是每次能够拉取的数据量，如果内存资源较为充足，适当增加拉取数据缓冲区的大小，可以减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。

reduce 端数据拉取缓冲区的大小可以通过 spark.reducer.maxSizeInFlight 参数进行设置，默认为 48MB，该参数的设置方法如代码清单 2-8 所示：

代码清单 2-8 reduce 端数据拉取缓冲区配置

```
val conf = new SparkConf().set("spark.reducer.maxSizeInFlight", "96")
```

1.3.3 调节 reduce 端拉取数据重试次数

Spark Shuffle 过程中, reduce task 拉取属于自己的数据时, 如果因为网络异常等原因导致失败会自动进行重试。对于那些包含了特别耗时的 shuffle 操作的作业, 建议增加重试最大次数(比如 60 次), 以避免由于 JVM 的 full gc 或者网络不稳定等因素导致的数据拉取失败。在实践中发现, 对于针对超大数据量(数十亿~上百亿)的 shuffle 过程, 调节该参数可以大幅度提升稳定性。

reduce 端拉取数据重试次数可以通过 spark.shuffle.io.maxRetries 参数进行设置, 该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功, 就可能会导致作业执行失败, 默认为 3, 该参数的设置方法如代码清单 2-9 所示:

代码清单 2-9 reduce 端拉取数据重试次数配置

```
val conf = new SparkConf().set("spark.shuffle.io.maxRetries", "6")
```

1.3.4 调节 reduce 端拉取数据等待间隔

Spark Shuffle 过程中, reduce task 拉取属于自己的数据时, 如果因为网络异常等原因导致失败会自动进行重试, 在一次失败后, 会等待一定的时间间隔再进行重试, 可以通过加大间隔时长(比如 60s), 以增加 shuffle 操作的稳定性。

reduce 端拉取数据等待间隔可以通过 spark.shuffle.io.retryWait 参数进行设置, 默认值为 5s, 该参数的设置方法如代码清单 2-10 所示:

代码清单 2-10 reduce 端拉取数据等待间隔配置

```
val conf = new SparkConf().set("spark.shuffle.io.retryWait", "60s")
```

1.3.5 调节 SortShuffle 排序操作阈值

对于 SortShuffleManager, 如果 shuffle reduce task 的数量小于某一阈值则 shuffle write 过程中不会进行排序操作, 而是直接按照未经优化的 HashShuffleManager 的方式去写数据, 但是最后会将每个 task 产生的所有临时磁盘文件都合并成一个文件, 并会创建单独的索引文件。

当你使用 SortShuffleManager 时, 如果的确不需要排序操作, 那么建议将这个参数调大一些, 大于 shuffle read task 的数量, 那么此时 map-side 就不会进行排序了, 减少了排序的性能开销, 但是这种方式下, 依然会产生大量的磁盘文件, 因此 shuffle write 性能有待提高。

SortShuffleManager 排序操作阈值的设置可以通过 spark.shuffle.sort.bypassMergeThreshold 这一参数进行设置, 默认值为 200, 该参数的设置方法如代码清单 2-11 所示:

代码清单 2-10 reduce 端拉取数据等待间隔配置

```
val conf = new SparkConf().set("spark.shuffle.sort.bypassMergeThreshold", "400")
```

1.4 JVM 调优

对于 JVM 调优, 首先应该明确 full gc 和 minor gc 都会导致 JVM 的工作线程停止工作, 即 stop the world。

1.4.1 降低 cache 操作的内存占比

1. 静态内存管理机制

根据 Spark 静态内存管理机制, 堆内存被划分为了两块: Storage 和 Execution。Storage 主要用于缓存 RDD 数据和 broadcast 数据, Execution 主要用于缓存在 shuffle 过程中产生的中间数据, Storage 占系统内存的 60%, Execution 占系统内存的 20%, 并且两者完全独立。

在一般情况下, Storage 的内存都提供给了 cache 操作, 但是如果在某些情况下 cache 操作内存不是很紧张, 而 task 的算子中创建的对象很多, Execution 内存又相对较小, 这回导致频繁的 minor gc, 甚至于频繁的 full gc, 进而导致 Spark 频繁的停止工作, 性能影响会很大。

在 Spark UI 中可以查看每个 stage 的运行情况, 包括每个 task 的运行时间、gc 时间等等, 如果发现 gc 太频繁, 时间太长, 就可以考虑调节 Storage 的内存占比, 让 task 执行算子函数式, 有更多的内存可以使用。

Storage 内存区域可以通过 spark.storage.memoryFraction 参数进行指定, 默认为 0.6, 即 60%, 可以逐级向下递减, 如代码清单 2-6 所示:

代码清单 2-6 Storage 内存占比设置

```
val conf = new SparkConf().set("spark.storage.memoryFraction", "0.4")
```

2. 统一内存管理机制

Spark 1.6 之后引入了统一内存管理机制, 堆内存被划分为了两块: Storage 和 Execution。Storage 主要用于缓存数据, Execution 主要用于缓存在 shuffle 过程中产生的中间数据, 两者所组成的内存部分称为统一内存, Storage 和

Execution 各占统一内存的 50%，由于动态占用机制的实现，shuffle 过程需要的内存过大时，会自动占用 Storage 的内存区域，因此无需手动进行调节。

1.4.2 调节 Executor 堆外内存

Executor 的堆外内存主要用于程序的共享库、Perm Space、线程 Stack 和一些 Memory mapping 等，或者类 C 方式 allocate object。

有时，如果你的 Spark 作业处理的数据量非常大，达到几亿的数据量，此时运行 Spark 作业会时不时地报错，例如 shuffle output file cannot find, executor lost, task lost, out of memory 等，这可能是 Executor 的堆外内存不太够用，导致 Executor 在运行的过程中内存溢出。

stage 的 task 在运行的时候，可能要从一些 Executor 中去拉取 shuffle map output 文件，但是 Executor 可能已经由于内存溢出挂掉了，其关联的 BlockManager 也没有了，这就可能会报出 shuffle output file cannot find, executor lost, task lost, out of memory 等错误，此时，就可以考虑调节一下 Executor 的堆外内存，也就可以避免报错，与此同时，堆外内存调节的比较大的时候，对于性能来讲，也会带来一定的提升。

默认情况下，Executor 堆外内存上限大概为 300 多 MB，在实际的生产环境下，对海量数据进行处理的时候，这里都会出现问题，导致 Spark 作业反复崩溃，无法运行，此时就会去调节这个参数，到至少 1G，甚至于 2G、4G。

Executor 堆外内存的配置需要在 spark-submit 脚本里配置，如代码清单 2-7 所示：

代码清单 2-7 Executor 堆外内存配置

```
--conf spark.yarn.executor.memoryOverhead=2048
```

以上参数配置完成后，会避免掉某些 JVM OOM 的异常问题，同时可以提升整体 Spark 作业的性能。

1.4.3 调节连接等待时长

在 Spark 作业运行过程中，Executor 优先从自己本地关联的 BlockManager 中获取某份数据，如果本地 BlockManager 没有的话，会通过 TransferService 远程连接其他节点上 Executor 的 BlockManager 来获取数据。

如果 task 在运行过程中创建大量对象或者创建的对象较大，会占用大量的内存，这回导致频繁的垃圾回收，但是垃圾回收会导致工作现场全部停止，也就是说，垃圾回收一旦执行，Spark 的 Executor 进程就会停止工作，无法提供相应，此时，由于没有响应，无法建立网络连接，会导致网络连接超时。

在生产环境下，有时会遇到 file not found、file lost 这类错误，在这种情况下，很有可能是 Executor 的 Block Manager 在拉取数据的时候，无法建立连接，然后超过默认的连接等待时长 60s 后，宣告数据拉取失败，如果反复尝试都拉取不到数据，可能会导致 Spark 作业的崩溃。这种情况也可能会导致 DAGScheduler 反复提交几次 stage，TaskScheduler 返回提交几次 task，大大延长了我们的 Spark 作业的运行时间。

此时，可以考虑调节连接的超时时长，连接等待时长需要在 spark-submit 脚本中进行设置，设置方式如代码清单 2-8 所示：

代码清单 2-8 连接等待时长配置

```
--conf spark.core.connection.ack.wait.timeout=300
```

调节连接等待时长后，通常可以避免部分的 XX 文件拉取失败、XX 文件 lost 等报错。

第 2 章 Spark 数据倾斜

Spark 中的数据倾斜问题主要指 shuffle 过程中出现的数据倾斜问题，是由于不同的 key 对应的数据量不同导致的不同 task 所处理的数据量不同的问题。

例如，reduce 一共要处理 100 万条数据，第一个和第二个 task 分别被分配到了 1 万条数据，计算 5 分钟内完成，第三个 task 分配到了 98 万数据，此时第三个 task 可能需要 10 个小时完成，这使得整个 Spark 作业需要 10 个小时才能运行完成，这就是数据倾斜所带来的后果。

注意，要区分开数据倾斜与数据量过量这两种情况，数据倾斜是指少数 task 被分配了绝大多数的数据，因此少数 task 运行缓慢；数据过量是指所有 task 被分配的数据量都很大，相差不多，所有 task 都运行缓慢。

数据倾斜的表现：

1. Spark 作业的大部分 task 都执行迅速，只有有限的几个 task 执行的非常慢，此时可能出现了数据倾斜，作业可以运行，但是运行得非常慢；

2. Spark 作业的大部分 task 都执行迅速，但是有的 task 在运行过程中会突然报出 OOM，反复执行几次都在某一个 task 报出 OOM 错误，此时可能出现了数据倾斜，作业无法正常运行。

定位数据倾斜问题：

1. 查阅代码中的 shuffle 算子，例如 reduceByKey、countByKey、groupByKey、join 等算子，根据代码逻辑判断此处是否会出现数据倾斜；

2. 查看 Spark 作业的 log 文件，log 文件对于错误的记录会精确到代码的某一行，可以根据异常定位到的代码位置来明确错误发生在第几个 stage，对应的 shuffle 算子是哪一个；

2.1 聚合原数据

1. 避免 shuffle 过程

绝大多数情况下，Spark 作业的数据来源都是 Hive 表，这些 Hive 表基本都是经过 ETL 之后的昨天的数据。

为了避免数据倾斜，我们可以考虑避免 shuffle 过程，如果避免了 shuffle 过程，那么从根本上就消除了发生数据倾斜问题的可能。

如果 Spark 作业的数据来源于 Hive 表，那么可以先在 Hive 表中对数据进行聚合，例如按照 key 进行分组，将同一 key 对应的所有 value 用一种特殊的格式拼接到一个字符串里去，这样，一个 key 就只有一条数据了；之后，对一个 key 的所有 value 进行处理时，只需要进行 map 操作即可，无需再进行任何的 shuffle 操作。通过上述方式就避免了执行 shuffle 操作，也就不可能会发生任何的数据倾斜问题。

对于 Hive 表中数据的操作，不一定是拼接成一个字符串，也可以是直接对 key 的每一条数据进行累计计算。

要区分开，处理的数据量大和数据倾斜的区别。

2. 缩小 key 粒度（增大数据倾斜可能性，降低每个 task 的数据量）

key 的数量增加，可能使数据倾斜更严重。

3. 增大 key 粒度（减小数据倾斜可能性，增大每个 task 的数据量）

如果没有办法对每个 key 聚合出来一条数据，在特定场景下，可以考虑扩大 key 的聚合粒度。

例如，目前有 10 万条用户数据，当前 key 的粒度是（省，城市，区，日期），现在我们考虑扩大粒度，将 key 的粒度扩大为（省，城市，日期），这样的话，key 的数量会减少，key 之间的数据量差异也有可能会减少，由此可以减轻数据倾斜的现象和问题。（此方法只针对特定类型的数据有效，当应用场景不适宜时，会加重数据倾斜）

2.2 过滤导致倾斜的 key

如果在 Spark 作业中允许丢弃某些数据，那么可以考虑将可能导致数据倾斜的 key 进行过滤，滤除可能导致数据倾斜的 key 对应的数据，这样在 Spark 作业中就不会发生数据倾斜了。

2.3 提高 shuffle 操作中的 reduce 并行度

当方案一和方案二对于数据倾斜的处理没有很好的效果时，可以考虑提高 shuffle 过程中的 reduce 端并行度，reduce 端并行度的提高就增加了 reduce 端 task 的数量，那么每个 task 分配到的数据量就会相应减少，由此缓解数据倾斜问题。

1. reduce 端并行度的设置

在大部分的 shuffle 算子中，都可以传入一个并行度的设置参数，比如 reduceByKey(500)，这个参数会决定 shuffle 过程中 reduce 端的并行度，在进行 shuffle 操作的时候，就会对应着创建指定数量的 reduce task。对于 Spark SQL 中的 shuffle 类语句，比如 group by、join 等，需要设置一个参数，即 spark.sql.shuffle.partitions，该参数代表了 shuffle read task 的并行度，该值默认是 200，对于很多场景来说都有点过小。

增加 shuffle read task 的数量，可以让原本分配给一个 task 的多个 key 分配给多个 task，从而让每个 task 处理比原来更多的数据。举例来说，如果原本有 5 个 key，每个 key 对应 10 条数据，这 5 个 key 都是分配给一个 task 的，那么这个 task 就要处理 50 条数据。而增加了 shuffle read task 以后，每个 task 就分配到一个 key，即每个 task 就处理 10 条数据，那么自然每个 task 的执行时间都会变短了。

2. reduce 端并行度设置存在的缺陷

提高 reduce 端并行度并没有从根本上改变数据倾斜的本质和问题（方案一和方案二从根本上避免了数据倾斜的发生），只是尽可能地去缓解和减轻 shuffle reduce task 的数据压力，以及数据倾斜的问题，适用于有较多 key 对应的数据量都比较大的情况。

该方案通常无法彻底解决数据倾斜，因为如果出现一些极端情况，比如某个 key 对应的数据量有 100 万，那么无论你的 task 数量增加到多少，这个对应着 100 万数据的 key 肯定还是会分配到一个 task 中去处理，因此注定还是会发生在数据倾斜的。所以这种方案只能说是在发现数据倾斜时尝试使用的第一种手段，尝试去用最简单的方法缓解数据倾斜而已，或者是和其他方案结合起来使用。

在理想情况下，reduce 端并行度提升后，会在一定程度上减轻数据倾斜的问题，甚至基本消除数据倾斜；但是，在一些情况下，只会让原来由于数据倾斜而运行缓慢的 task 运行速度稍有提升，或者避免了某些 task 的 OOM 问题，但是，仍然运行缓慢，此时，要及时放弃方案三，开始尝试后面的方案。

2.4 使用随机 key 实现双重聚合

当使用了类似于 groupByKey、reduceByKey 这样的算子时，可以考虑使用随机 key 实现双重聚合，如图 3-1 所示：

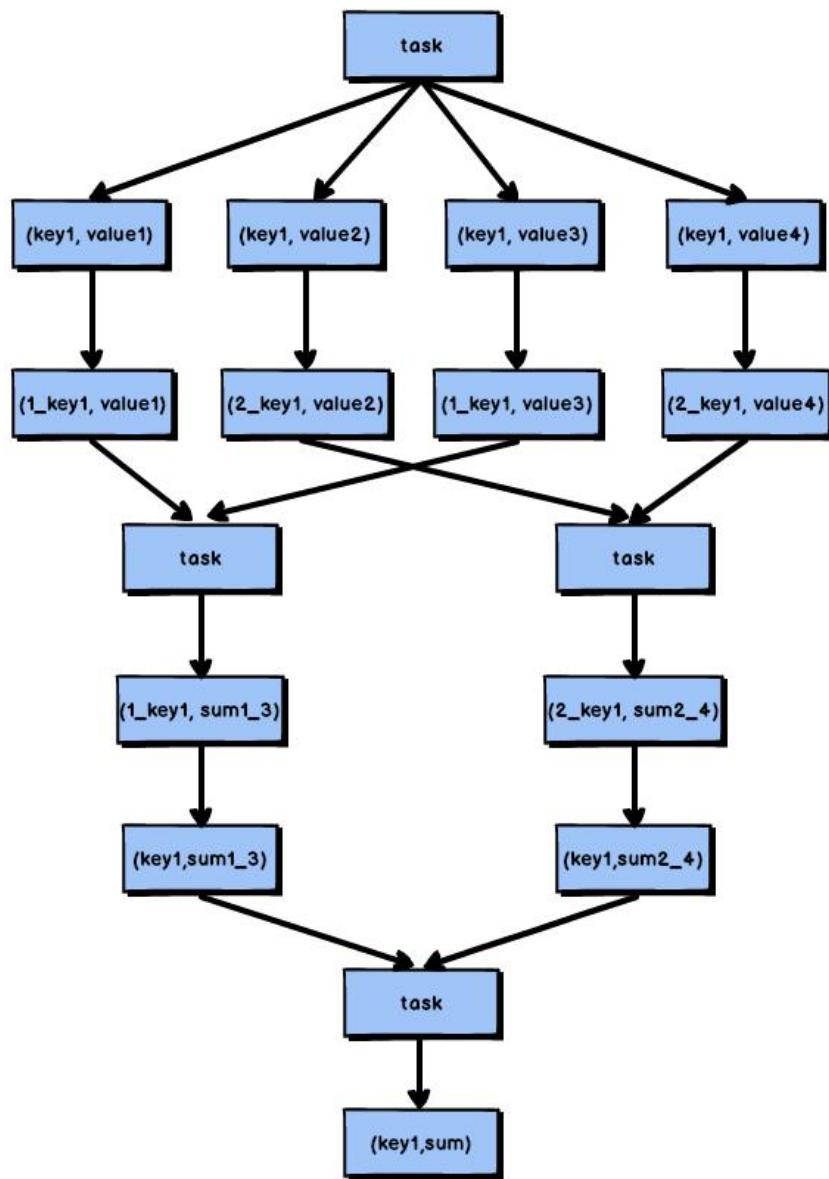


图 3-1 随机 key 实现双重聚合

首先，通过 map 算子给每个数据的 key 添加随机数前缀，对 key 进行打散，将原先一样的 key 变成不一样的 key，然后进行第一次聚合，这样就可以让原本被一个 task 处理的数据分散到多个 task 上去做局部聚合；随后，去掉每个 key 的前缀，再次进行聚合。

此方法对于由 `groupByKey`、`reduceByKey` 这类算子造成的数据倾斜有比较好的效果，仅仅适用于聚合类的 `shuffle` 操作，适用范围相对较窄。如果是 `join` 类的 `shuffle` 操作，还得用其他的解决方案。

此方法也是前几种方案没有比较好的效果时要尝试的解决方案。

2.5 map join 代替 reduce join

正常情况下，`join` 操作都会执行 `shuffle` 过程，并且执行的是 `reduce join`，也就是先将所有相同的 key 和对应的 value 汇聚到一个 `reduce task` 中，然后再进行 `join`。普通 `join` 的过程如下图所示：

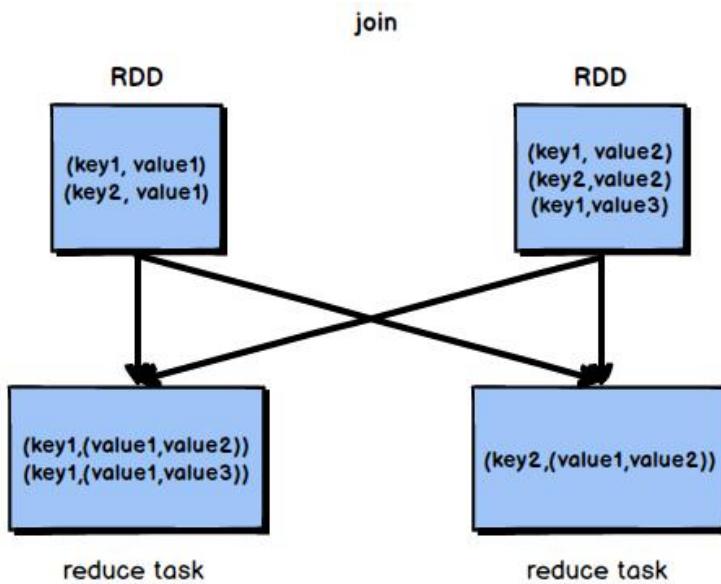


图 3-2 普通 join 过程

普通的 join 是会走 shuffle 过程的，而一旦 shuffle，就相当于会将相同 key 的数据拉取到一个 shuffle read task 中再进行 join。但是如果一个 RDD 是比较小的，则可以采用广播小 RDD 全量数据+map 算子来实现与 join 同样的效果，也就是 map join，此时就不会发生 shuffle 操作，也就不会发生数据倾斜。

(注意，RDD 是并不能进行广播的，只能将 RDD 内部的数据通过 collect 拉取到 Driver 内存然后再进行广播)

1. 核心思路：

不使用 join 算子进行连接操作，而使用 **Broadcast** 变量与 **map** 类算子实现 join 操作，进而完全规避掉 shuffle 类的操作，彻底避免数据倾斜的发生和出现。将较小 RDD 中的数据直接通过 collect 算子拉取到 Driver 端的内存中来，然后对其创建一个 Broadcast 变量；接着对另外一个 RDD 执行 map 类算子，在算子函数内，从 Broadcast 变量中获取较小 RDD 的全量数据，与当前 RDD 的每一条数据按照连接 key 进行比对，如果连接 key 相同的话，那么就将两个 RDD 的数据用你需要的方式连接起来。

根据上述思路，根本不会发生 shuffle 操作，从根本上杜绝了 join 操作可能导致的数据倾斜问题。

当 join 操作有数据倾斜问题并且其中一个 RDD 的数据量较小时，可以优先考虑这种方式，效果非常好。map join 的过程如图 3-3 所示：

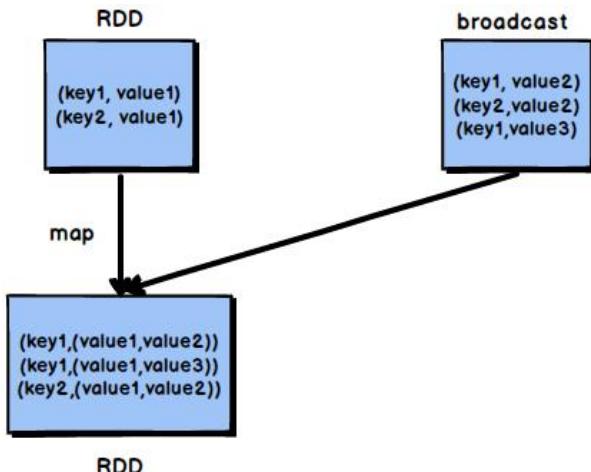


图 3-3 map join 过程

2. 不适用场景分析：

由于 Spark 的广播变量是在每个 Executor 中保存一个副本，如果两个 RDD 数据量都比较大，那么如果将一个数据量比较大的 RDD 做成广播变量，那么很有可能会造成内存溢出。

2.6 sample 采样对倾斜 key 单独进行 join

在 Spark 中，如果某个 RDD 只有一个 key，那么在 shuffle 过程中会默认将此 key 对应的数据打散，由不同的 reduce 端 task 进行处理。

当由单个 key 导致数据倾斜时，可将发生数据倾斜的 key 单独提取出来，组成一个 RDD，然后用这个原本会导致倾斜的 key 组成的 RDD 和其他 RDD 单独 join，此时，根据 Spark 的运行机制，此 RDD 中的数据会在 shuffle 阶段被分散到多个 task 中去进行 join 操作。倾斜 key 单独 join 的流程如图 3-4 所示：

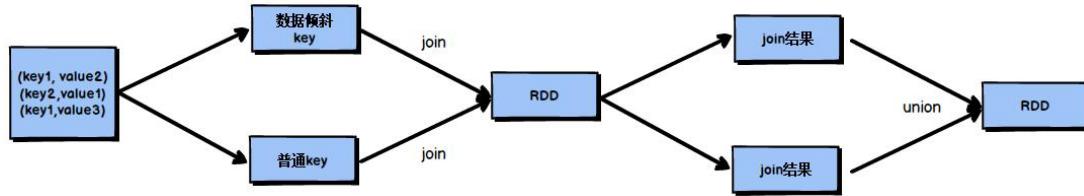


图 3-4 倾斜 key 单独 join 流程

1. 适用场景分析：

对于 RDD 中的数据，可以将其转换为一个中间表，或者是直接使用 `countByKey()` 的方式，看一个这个 RDD 中各个 key 对应的数据量，此时如果你发现整个 RDD 就一个 key 的数据量特别多，那么就可以考虑使用这种方法。

当数据量非常大时，可以考虑使用 `sample` 采样获取 10% 的数据，然后分析这 10% 的数据中哪个 key 可能会导致数据倾斜，然后将这个 key 对应的数据单独提取出来。

2. 不适用场景分析：

如果一个 RDD 中导致数据倾斜的 key 很多，那么此方案不适用。

2.7 使用随机数以及扩容进行 join

如果在进行 join 操作时，RDD 中有大量的 key 导致数据倾斜，那么进行分拆 key 也没什么意义，此时就只能使用最后一种方案来解决问题了，对于 join 操作，我们可以考虑对其中一个 RDD 数据进行扩容，另一个 RDD 进行稀释后再 join。

我们会将原先一样的 key 通过附加随机前缀变成不一样的 key，然后就可以将这些处理后的“不同 key”分散到多个 task 中去处理，而不是让一个 task 处理大量的相同 key。这一种方案是针对有大量倾斜 key 的情况，没法将部分 key 拆分出来进行单独处理，需要对整个 RDD 进行数据扩容，对内存资源要求很高。

1. 核心思想：

选择一个 RDD，使用 `flatMap` 进行扩容，对每条数据的 key 添加数值前缀（1~N 的数值），将一条数据映射为多条数据；（扩容）

选择另外一个 RDD，进行 `map` 映射操作，每条数据的 key 都打上一个随机数作为前缀（1~N 的随机数）；（稀释）

将两个处理后的 RDD，进行 join 操作。

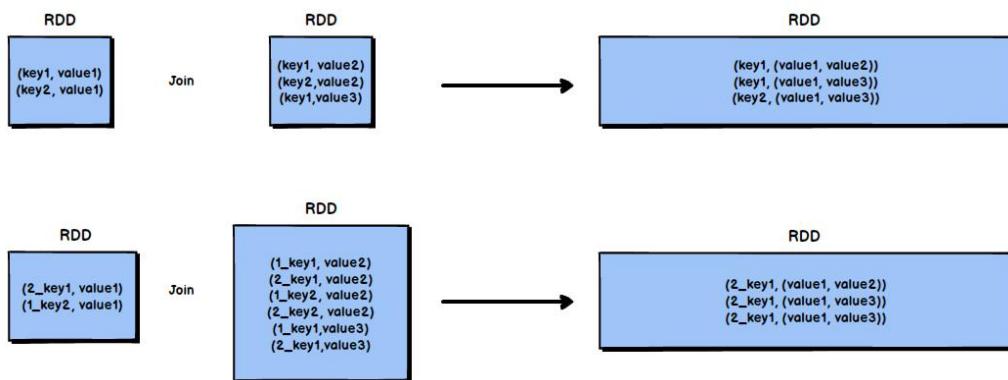


图 3-6 使用随机数以及扩容进行 join

2. 局限性：

如果两个 RDD 都很大，那么将 RDD 进行 N 倍的扩容显然行不通；

使用扩容的方式只能缓解数据倾斜，不能彻底解决数据倾斜问题。

3. 使用方案七对方案六进一步优化分析：

当 RDD 中有几个 key 导致数据倾斜时，方案六不再适用，而方案七又非常消耗资源，此时可以引入方案七的思想完善方案六：

1) 对包含少数几个数据量过大的 key 的那个 RDD，通过 `sample` 算子采样出一份样本来，然后统计一下每个 key 的数量，计算出来数据量最大的是哪几个 key。

2) 然后将这几个 key 对应的数据从原来的 RDD 中拆分出来，形成一个单独的 RDD，并给每个 key 都打上 n 以内的随机数作为前缀，而不会导致倾斜的大部分 key 形成另外一个 RDD。

3) 接着将需要 join 的另一个 RDD, 也过滤出来那几个倾斜 key 对应的数据并形成一个单独的 RDD, 将每条数据膨胀成 n 条数据, 这 n 条数据都按顺序附加一个 0~n 的前缀, 不会导致倾斜的大部分 key 也形成另外一个 RDD。

4) 再将附加了随机前缀的独立 RDD 与另一个膨胀 n 倍的独立 RDD 进行 join, 此时就可以将原先相同的 key 打散成 n 份, 分散到多个 task 中去进行 join 了。

5) 而另外两个普通的 RDD 就照常 join 即可。

6) 最后将两次 join 的结果使用 union 算子合并起来即可, 就是最终的 join 结果。

第 3 章 Spark Troubleshooting

3.1 故障排除一：控制 reduce 端缓冲大小以避免 OOM

在 Shuffle 过程, reduce 端 task 并不是等到 map 端 task 将其数据全部写入磁盘后再去拉取, 而是 map 端写一点数据, reduce 端 task 就会拉取一小部分数据, 然后立即进行后面的聚合、算子函数的使用等操作。

reduce 端 task 能够拉取多少数据, 由 reduce 拉取数据的缓冲区 buffer 来决定, 因为拉取过来的数据都是先放在 buffer 中, 然后再进行后续的处理, buffer 的默认大小为 48MB。

reduce 端 task 会一边拉取一边计算, 不一定每次都会拉满 48MB 的数据, 可能大多数时候拉取一部分数据就处理掉了。

虽然说增大 reduce 端缓冲区大小可以减少拉取次数, 提升 Shuffle 性能, 但是有时 map 端的数据量非常大, 写出的速度非常快, 此时 reduce 端的所有 task 在拉取的时候, 有可能全部达到自己缓冲的最大极限值, 即 48MB, 此时, 再加上 reduce 端执行的聚合函数的代码, 可能会创建大量的对象, 这可难会导致内存溢出, 即 OOM。

如果一旦出现 reduce 端内存溢出的问题, 我们可以考虑减小 reduce 端拉取数据缓冲区的大小, 例如减少为 12 MB。

在实际生产环境中是出现过这种问题的, 这是典型的以性能换执行的原理。reduce 端拉取数据的缓冲区减小, 不容易导致 OOM, 但是相应的, reduce 端的拉取次数增加, 造成更多的网络传输开销, 造成性能的下降。

注意, 要保证任务能够运行, 再考虑性能的优化。

3.2 故障排除二：JVM GC 导致的 shuffle 文件拉取失败

在 Spark 作业中, 有时会出现 shuffle file not found 的错误, 这是非常常见的一个报错, 有时出现这种错误以后, 选择重新执行一遍, 就不再报出这种错误。

出现上述问题可能的原因是 Shuffle 操作中, 后面 stage 的 task 想要去上一个 stage 的 task 所在的 Executor 拉取数据, 结果对方正在执行 GC, 执行 GC 会导致 Executor 内所有的工作现场全部停止, 比如 BlockManager、基于 netty 的网络通信等, 这就会导致后面的 task 拉取数据拉取了半天都没有拉取到, 就会报出 shuffle file not found 的错误, 而第二次再次执行就不会再出现这种错误。

可以通过调整 reduce 端拉取数据重试次数和 reduce 端拉取数据时间间隔这两个参数来对 Shuffle 性能进行调整, 增大参数值, 使得 reduce 端拉取数据的重试次数增加, 并且每次失败后等待的时间间隔加长。

代码清单 4-1 JVM GC 导致的shuffle文件拉取失败

```
val conf = new SparkConf().set("spark.shuffle.io.maxRetries", "60").set("spark.shuffle.io.retryWait", "60s")
```

3.3 故障排除三：解决各种序列化导致的报错

当 Spark 作业在运行过程中报错, 而且报错信息中含有 Serializable 等类似词汇, 那么可能是序列化问题导致的报错。

序列化问题要注意以下三点:

1. 作为 RDD 的元素类型的自定义类, 必须是可以序列化的;
2. 算子函数里可以使用的外部的自定义变量, 必须是可以序列化的;
3. 不可以在 RDD 的元素类型、算子函数里使用第三方的不支持序列化的类型, 例如 Connection。

3.4 故障排除四：解决算子函数返回 NULL 导致的问题

在一些算子函数里, 需要我们有一个返回值, 但是在一些情况下我们不希望有返回值, 此时我们如果直接返回 NULL, 会报错, 例如 Scala.Math(NULL) 异常。

如果你遇到某些情况, 不希望有返回值, 那么可以通过下述方式解决:

1. 返回特殊值, 不返回 NULL, 例如 “-1” ;
2. 在通过算子获取到了一个 RDD 之后, 可以对这个 RDD 执行 filter 操作, 进行数据过滤, 将数值为 -1 的数据给过滤掉;
3. 在使用完 filter 算子后, 继续调用 coalesce 算子进行优化。

3.5 故障排除五：解决 YARN-CLIENT 模式导致的网卡流量激增问题

YARN-client 模式的运行原理如下图所示:

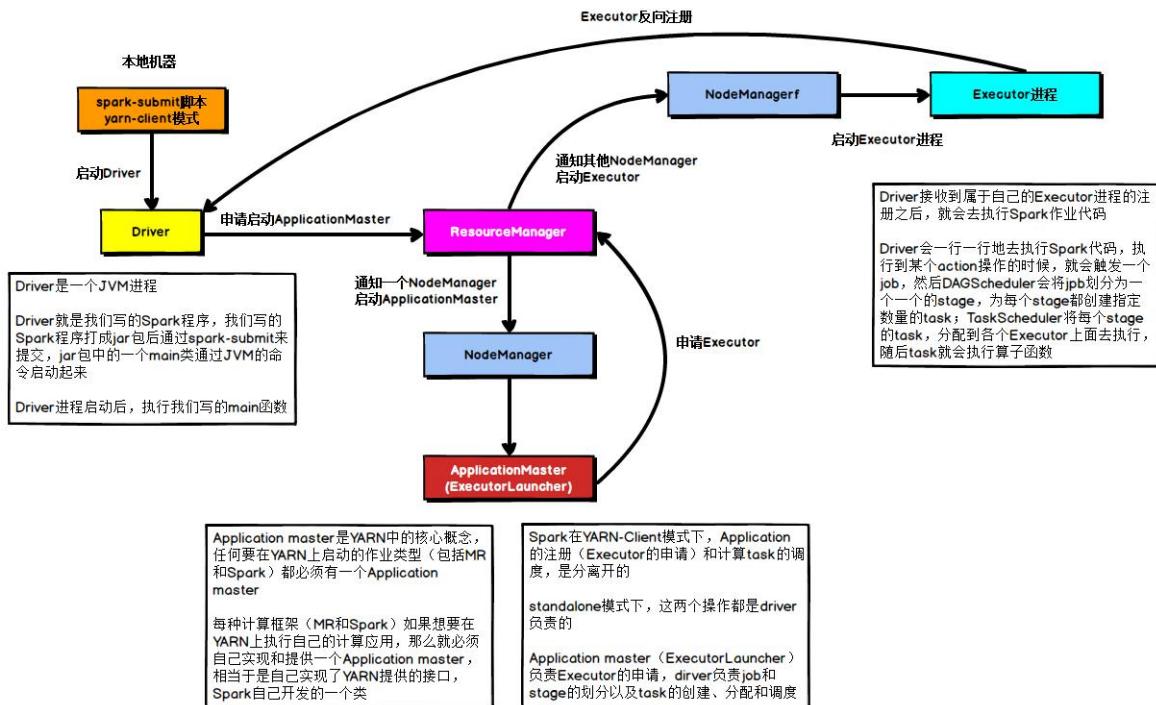


图 4-1 YARN-client 模式运行原理

在 YARN-client 模式下，Driver 启动在本地机器上，而 Driver 负责所有的任务调度，需要与 YARN 集群上的多个 Executor 进行频繁的通信。

假设有 100 个 Executor，1000 个 task，那么每个 Executor 分配到 10 个 task，之后，Driver 要频繁地跟 Executor 上运行的 1000 个 task 进行通信，通信数据非常多，并且通信品类特别高。这就导致有可能在 Spark 任务运行过程中，由于频繁大量的网络通讯，本地机器的网卡流量会激增。

注意，YARN-client 模式只会在测试环境中使用，而之所以使用 YARN-client 模式，是由于可以看到详细全面的 log 信息，通过查看 log，可以锁定程序中存在的问题，避免在生产环境下发生故障。

在生产环境下，使用的一定是 YARN-cluster 模式。在 YARN-cluster 模式下，就不会造成本地机器网卡流量激增问题，如果 YARN-cluster 模式下存在网络通信的问题，需要运维团队进行解决。

3.6 故障排除六：解决 YARN-CLUSTER 模式的 JVM 栈内存溢出无法执行问题

YARN-cluster 模式的运行原理如下图所示：

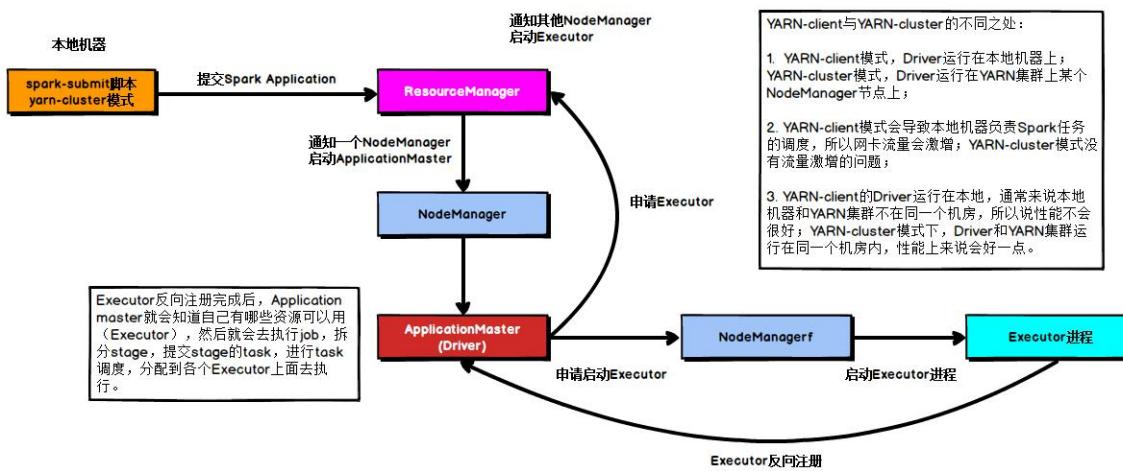


图 4-1 YARN-client 模式运行原理

当 Spark 作业中包含 SparkSQL 的内容时，可能会碰到 YARN-client 模式下可以运行，但是 YARN-cluster 模式下无法提交运行（报出 OOM 错误）的情况。

YARN-client 模式下，Driver 是运行在本地机器上的，Spark 使用的 JVM 的 PermGen 的配置，是本地机器上的 spark-class 文件，JVM 永久代的大小是 128MB，这个是没有问题的，但是在 YARN-cluster 模式下，Driver 运行在 YARN 集群的某个节点上，使用的是没有经过配置的默认设置，PermGen 永久代大小为 82MB。

SparkSQL 的内部要进行很复杂的 SQL 的语义解析、语法树转换等等，非常复杂，如果 sql 语句本身就非常复杂，那么很有可能会导致性能的损耗和内存的占用，特别是对 PermGen 的占用会比较大。

所以，此时如果 PermGen 的占用好过了 82MB，但是又小于 128MB，就会出现 YARN-client 模式下可以运行，YARN-cluster 模式下无法运行的情况。

解决上述问题的方法时增加 PermGen 的容量，需要在 spark-submit 脚本中对相关参数进行设置，设置方法如代码清单 4-2 所示。

代码清单 4-2 配置

```
--conf spark.driver.extraJavaOptions="-XX:PermSize=128M -XX:MaxPermSize=256M"
```

通过上述方法就设置了 Driver 永久代的大小，默认为 128MB，最大 256MB，这样就可以避免上面所说的问题。

3.7 故障排除七：解决 SparkSQL 导致的 JVM 栈内存溢出

当 SparkSQL 的 sql 语句有成百上千的 or 关键字时，就可能会出现 Driver 端的 JVM 栈内存溢出。

JVM 栈内存溢出基本上就是由于调用的方法层级过多，产生了大量的，非常深的，超出了 JVM 栈深度限制的递归。（我们猜测 SparkSQL 有大量 or 语句的时候，在解析 SQL 时，例如转换为语法树或者进行执行计划的生成的时候，对于 or 的处理是递归，or 非常多时，会发生大量的递归）

此时，建议将一条 sql 语句拆分为多条 sql 语句来执行，每条 sql 语句尽量保证 100 个以内的子句。根据实际的生产环境试验，一条 sql 语句的 or 关键字控制在 100 个以内，通常不会导致 JVM 栈内存溢出。

3.8 故障排除八：持久化与 checkpoint 的使用

Spark 持久化在大部分情况下是没有问题的，但是有时数据可能会丢失，如果数据一旦丢失，就需要对丢失的数据重新进行计算，计算完后再缓存和使用，为了避免数据的丢失，可以选择对这个 RDD 进行 checkpoint，也就是将数据持久化一份到容错的文件系统上（比如 HDFS）。

一个 RDD 缓存并 checkpoint 后，如果一旦发现缓存丢失，就会优先查看 checkpoint 数据存不存在，如果有，就会使用 checkpoint 数据，而不用重新计算。也即是说，checkpoint 可以视为 cache 的保障机制，如果 cache 失败，就使用 checkpoint 的数据。

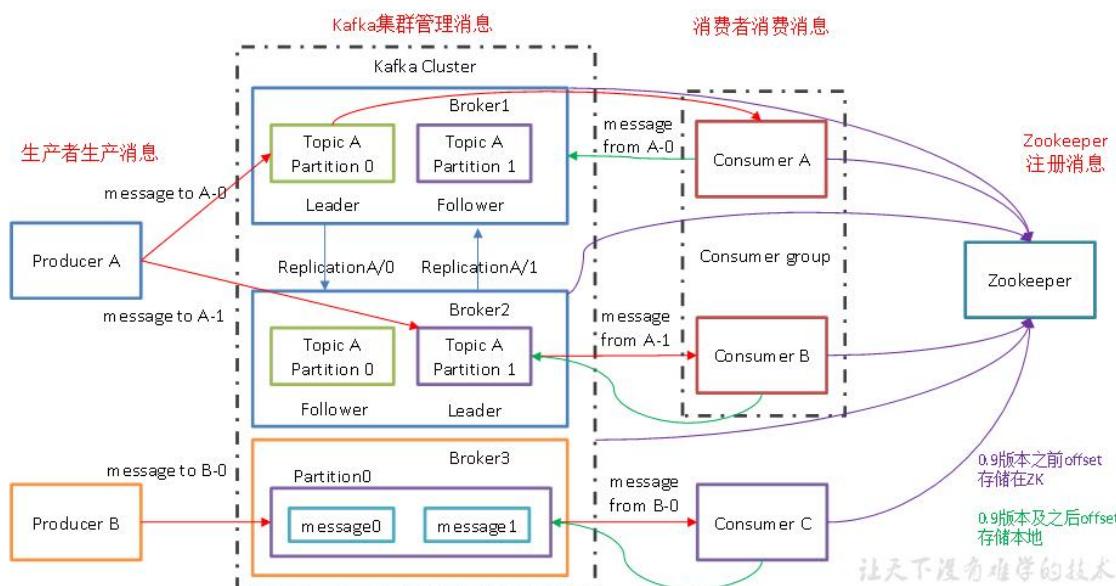
使用 checkpoint 的优点在于提高了 Spark 作业的可靠性，一旦缓存出现问题，不需要重新计算数据，缺点在于，checkpoint 时需要将数据写入 HDFS 等文件系统，对性能的消耗较大。

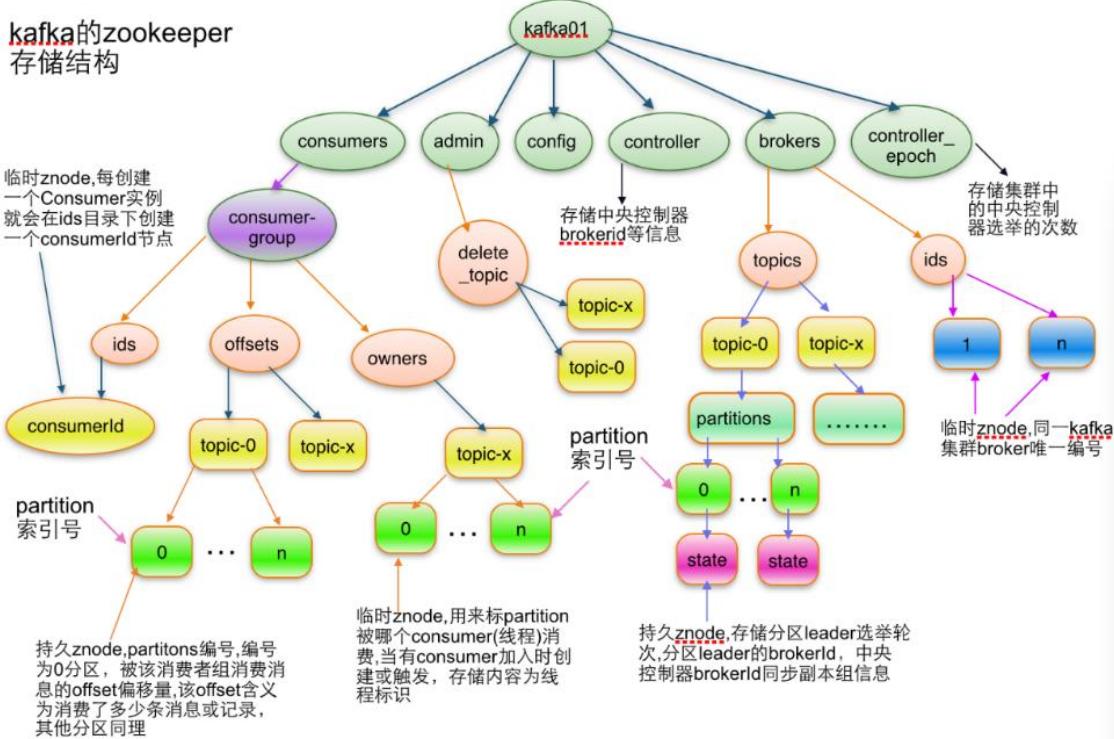
Kafka 面试题

1.1 Kafka 架构

生产者、Broker、消费者、ZK；

注意：Zookeeper 中保存 Broker id 和消费者 offsets 等信息，但是没有生产者信息。





1.2 Kafka 的机器数量

Kafka 机器数量 = $2 * (\text{峰值生产速度} * \text{副本数} / 100) + 1$

1.3 kafka 副本数设定

一般我们设置成 2 个或 3 个，很多企业设置为 2 个。

副本的优点是提高可靠性，缺点是增加了网络 IO 传输。

1.4 Kafka 压测问题

Kafka 官方自带压力测试脚本（`kafka-consumer-perf-test.sh`、`kafka-producer-perf-test.sh`）。Kafka 压测时，可以查看到哪个地方出现了瓶颈（CPU，内存，网络 IO）。一般都是网络 IO 达到瓶颈。

1.5 Kafka 日志保存时间

默认保存 7 天；生产环境建议 3 天

1.6 Kafka 中数据量计算

每天总数据量 100g，每天产生 1 亿条日志， $10000 \text{ 万} / 24 / 60 / 60 = 1150 \text{ 条/每秒钟}$

平均每秒钟：1150 条

低谷每秒钟：50 条

高峰每秒钟： $1150 \text{ 条} * (2\text{-}20 \text{ 倍}) = 2300 \text{ 条} - 23000 \text{ 条}$

每条日志大小：0.5k-2k（取 1k）

每秒多少数据量：2.0M-20MB

1.7 Kafka 的硬盘大小

每天的数据量 $100g * 2 \text{ 个副本} * 3 \text{ 天} / 70\%$

1.8 Kafka 监控器

一般公司有自己开发的监控器；开源的监控器：`KafkaManager`、`Monitor`、`Eagle`

1.9 Kakfa 分区数计算

- 1) 创建一个只有 1 个分区的 topic
 - 2) 测试这个 topic 的 producer 吞吐量和 consumer 吞吐量。
 - 3) 假设他们的值分别是 T_p 和 T_c , 单位可以是 MB/s。
 - 4) 然后假设总的目标吞吐量是 T_t , 那么分区数 = $T_t / \min(T_p, T_c)$
- 例如: producer 吞吐量=20m/s; consumer 吞吐量=50m/s, 期望吞吐量 100m/s;
 分区数=100 / 20 = 5 分区
https://blog.csdn.net/weixin_42641909/article/details/89294698
 分区数一般设置为: 3-10 个

1.10 kafka 的 Topic 数量

通常情况下, 多少个日志类型就多少个 Topic, 也有对日志类型进行合并的。

1.11 Kafka 的 ISR (副本同步队列)

ISR (副本同步队列) : ISR 中包括 Leader 和 Follower。如果 Leader 进程挂掉, 会在 ISR 队列中选择一个服务作为新的 Leader。replica.lag.max.messages (延迟条数) 和 replica.lag.time.max.ms (延迟时间) 两个参数决定一台服务是否可以加入 ISR 副本队列, 在 0.10 版本移除了 replica.lag.max.messages 参数, 防止服务频繁的进去队列。

任意一个维度超过阈值都会把 Follower 剔除出 ISR, 存入 OSR (Outof-Sync Replicas) 列表, 新加入的 Follower 也会先存放在 OSR 中。

1.12 Kafka 分区分配策略

在 Kafka 内部存在两种默认的分区分配策略: Range 和 RoundRobin。

Range 是默认策略。Range 是对每个 Topic 而言的 (即一个 Topic 一个 Topic 分), 首先对同一个 Topic 里面的分区按照序号进行排序, 并对消费者按照字母顺序进行排序。然后用 Partitions 分区的个数除以消费者线程的总数来决定每个消费者线程消费几个分区。如果除不尽, 那么前面几个消费者线程将会多消费一个分区。

例如: 我们有 10 个分区, 两个消费者 (C1, C2), 3 个消费者线程, $10 / 3 = 3$ 而且除不尽。

C1-0 将消费 0, 1, 2, 3 分区

C2-0 将消费 4, 5, 6 分区

C2-1 将消费 7, 8, 9 分区

第一步: 将所有主题分区组成 TopicAndPartition 列表, 然后对 TopicAndPartition 列表按照 hashCode 进行排序, 最后按照轮询的方式发给每一个消费线程。

1.13 Kafka 容机问题

Kafka 是高可用的, 在某 broker 容机的时候, 只要该分区存在副本, 并且副本有在 ISR 集合中维护, 那么消费者可以继续消费数据; 如果是 __consumer_offsets 分区所在容机了, 确实会导致消费者消费不到数据, 即 __consumer_offsets 这个 topic 存在单点问题, 我们要手动修改该主题的副本数量, 可以借助 zookeeper 来删除然后重新设置。

__consumer_offsets 这个 topic 是由 kafka 自动创建的, 默认 50 个, 但是都存在一台 kafka 服务器上, 其副本数默认是 1 (如果我们没有在 server.properties 文件中指定 topic 分区的副本数的话)。

1.14 Kafka 丢不丢数据

Ack=0, 相当于异步发送, 消息发送完毕即 offset 增加, 继续生产。

Ack=1, leader 收到 leader replica 对一个消息的接受 ack 才增加 offset, 然后继续生产。

Ack=-1, leader 收到所有 replica 对一个消息的接受 ack 才增加 offset, 然后继续生产。

1.15 Kafka 数据一致性

幂等性 + ack=-1 + 事务

Kafka 数据重复，可以再下一级：SparkStreaming、redis 或者 hive 中 dwd 层去重，去重的手段：分组、按照 id 开窗只取第一个值；

1.16 Kafka 消息数据积压/消费能力不足怎么处理？

1) 如果是 Kafka 消费能力不足，则可以考虑增加 Topic 的分区数，并且同时提升消费组的消费者数量，消费者数=分区数。（两者缺一不可）

2) 如果是下游的数据处理不及时：提高每批次拉取的数据量。批次拉取数据过少（拉取数据/处理时间<生产速度），使处理的数据小于生产的数据，也会造成数据积压。

1.17 Kafka 参数优化

1) Broker 参数配置 (`server.properties`)

1、日志保留策略配置

```
# 保留三天，也可以更短 (log.cleaner.delete.retention.ms)
```

```
log.retention.hours=72
```

2、Replica 相关配置

```
default.replication.factor:1 默认副本 1 个
```

3、网络通信延时

```
replica.socket.timeout.ms:30000 #当集群之间网络不稳定时,调大该参数
```

```
replica.lag.time.max.ms= 600000# 如果网络不好,或者 kafka 集群压力较大,会出现副本丢失,然后会频繁复制副本,导致集群压力更大,此时可以调大该参数
```

2) Producer 优化 (`producer.properties`)

```
compression.type:none      gzip      snappy      lz4
```

默认发送不进行压缩，推荐配置一种适合的压缩算法，可以大幅度的减缓网络压力和 Broker 的存储压力。

3) Kafka 内存调整 (`kafka-server-start.sh`)

默认内存 1 个 G，生产环境尽量不要超过 6 个 G。

```
export KAFKA_HEAP_OPTS="-Xms4g -Xmx4g"
```

1.18 Kafka 高效读写数据

1) Kafka 本身是分布式集群，同时采用分区技术并发度高；

2) 顺序写磁盘

Kafka 的 producer 生产数据，要写入到 log 文件中，写的过程是一直追加到文件末端，为顺序写。官网有数据表明，同样的磁盘，顺序写能到 600M/s，而随机写只有 100K/s。

3) 零复制技术

1.19 Kafka 单条日志传输大小

kafka 对于消息体的大小默认为单条最大值是 1M 但是在我们应用场景中，常常会出现一条消息大于 1M，如果不对 kafka 进行配置。则会出现生产者无法将消息推送到 kafka 或消费者无法去消费 kafka 里面的数据，这时我们就要对 kafka 进行以下配置：`server.properties`

```
replica.fetch.max.bytes: 1048576 broker 可复制的消息的最大字节数，默认为 1M
```

```
message.max.bytes: 1000012 kafka 接收单个消息 size 的最大限制， 默认为 1M
```

注意：`message.max.bytes` 必须小于等于 `replica.fetch.max.bytes`，否则就会导致 replica 之间数据同步失败。

1.20 Kafka 过期数据清理

保证数据没有被引用（没人消费他）

日志清理保存的策略只有 `delete` 和 `compact` 两种

```
log.cleanup.policy=delete 启用删除策略
```

```
log.cleanup.policy=compact 启用压缩策略
```

<https://www.jianshu.com/p/fa6adeae8eb5>

1.21 Kafka 按时间消费数据

```
Map<TopicPartition, OffsetAndTimestamp> startOffsetMap = KafkaUtil.fetchOffsetsWithTimestamp(topic, sTime, kafkaProp);
```

1.22 Kafka 数据推送/拉取

Producer 推送数据，Consumer 拉取数据

1.23 Kafka 中的数据是有序的吗

单分区内是有序的；多分区时候，分区与分区间无序；

Producer 通过维护 pid 和 sequence num 实现单分区有序，但是单分区并行消费的情况下会出现乱序消费，此时的解决方案是创建多个内存队列，具有相同 key 的数据都路由到同一个内存 队列；然后每个线程分别消费一个内存队列即可，这样就能保证顺序性。参考 https://blog.csdn.net/weixin_42494845/article/details/111408725

Spark 面试题

2.1 Spark 有几种部署方式？

- 1) Local：运行在一台机器上，通常是练手或者测试环境。
- 2) Standalone：构建一个基于 Master+Slaves 的资源调度集群，Spark 任务提交给 Master 运行。是 Spark 自身的一个调度系统。
- 3) Yarn：Spark 客户端直接连接 Yarn，不需要额外构建 Spark 集群。有--deploy-mode client 和--deploy-mode cluster 两种模式，主要区别在于 Driver 程序的运行节点。
- 4) Mesos：国内大环境比较少用。

2.2 Spark 作业提交参数

- 1) 几个重要参数

executor-cores —— 每个 executor 使用的内核数，默认为 1，官方建议 2-5 个，我们企业是 4 个，内核越多并行任务数就多

num-executors —— 启动 executors 的数量，默认为 2

executor-memory —— executor 内存大小，默认 1G

driver-cores —— ApplicationMaster 使用内核数，默认为 1

driver-memory —— driver 内存大小，默认 512M

- 2) 提交任务的样式

```
spark-submit \
```

```
--master local[5] \
```

```
--driver-cores 2 \
```

```
--driver-memory 8g \
--executor-cores 4 \
--num-executors 10 \
--executor-memory 8g \
--class PackageName.ClassName XXXX.jar \
--name "Spark Job Name" \
InputPath      \
OutputPath
```

2.3 简述 Spark 的架构与作业提交流程

Spark 架构: master (rm)、driver、executor、worker (nn)

作业提交流程: yarn-cluster 模式

Spark 任务提交后会和 ResourceManager 通讯申请启动 ApplicationMaster, 随后 ResourceManager 分配 container, 在合适的 NodeManager 上启动 ApplicationMaster, 此时的 ApplicationMaster 就是 Driver。

Driver 启动后向 ResourceManager 申请 Executor 内存, ResourceManager 接到 ApplicationMaster 的资源申请后会分配 container, 然后在合适的 NodeManager 上启动 Executor 进程, Executor 进程启动后会向 Driver 反向注册, Executor 全部注册完成后 Driver 开始执行 main 函数, 之后执行到 Action 算子时, 触发一个 job, 并根据宽依赖开始划分 stage, 每个 stage 生成对应的 taskSet, 之后将 task 分发到各个 Executor 上执行。

2.4 简述 Spark 的宽窄依赖和任务划分

窄依赖: 父 rdd 的一个分区最多只被子 rdd 的一个分区使用

宽依赖: 子 RDD 的多个分区会依赖父 RDD 的同一个分区

任务划分: spark 执行的时候, 每遇到一个 action 算子生成一个 job; 遇到一个宽依赖划分一个 stage, 一个 stage 生成一个 taskset; 一个 rdd 分区生成一个 task; 一个 job 有多个 taskset, 一个 taskset 有多个 task。

2.5 列举 8 个 transformation 算子并简述其作用

map(): 返回一个新 rdd

mapPartitions(): 返回一个新 rdd, 但独立在每一个 rdd 分片上运行

flatMap(): 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

groupBy(): 分组

filter(): 过滤

distinct(): 去重

union(): 求并集返回新 rdd

partitionBy(): 对 pairrdd 进行分区操作

groupByKey(): 对相同 key 的 value 进行聚合

reduceByKey(): 对相同 key 的 value 进行累加

2.6 列举 6 个 action 算子并简述其作用

foreach(): 在数据集的每一个元素上, 进行函数更新

reduce(): 通过函数聚集 RDD 中的所有元素, 先聚合分区内数据, 再聚合分区间数据

first(): 返回 RDD 中的第一个元素

take(): 返回一个由 RDD 的前 n 个元素组成的数组

collect(): 以数组的形式返回数据集的所有元素到 driver 端

saveAsTextFile(): 存储为 textfile 格式

2.7 reduceByKey 和 groupByKey 的区别

reduceByKey: 按照 key 进行聚合, 在 shuffle 之前有 combine (预聚合) 操作, 返回结果是 RDD[k,v]。

`groupByKey`: 按照 key 进行分组，直接进行 shuffle。
`reduceByKey` 在不影响业务的前提下性能更好。

2.8 Repartition 和 Coalesce 关系与区别

两者都是用来改变 RDD 分区数量的，`repartition` 底层调用的就是 `coalesce` 方法：`coalesce(numPartitions, shuffle = true)`。
`repartition` 一定会发生 shuffle，`coalesce` 根据传入的参数来判断是否发生 shuffle。一般情况下增大 rdd 的分区数用 `repartition`，减少分区数时使用 `coalesce`。但是如果数据量过大，分区数过少会出现 OOM 所以 `coalesce` 缩小分区个数也需合理。

2.9 分别简述 Spark 中的缓存机制（cache 和 persist）与 checkpoint 机制

两者都是做 rdd 持久化的。Cache 是持久化到内存，不会截断血缘关系，目的是数据重用；而 checkpoint 是持久化到硬盘，截断血缘关系，目的是减少 rdd 重新计算带来的性能损耗。

2.10 简述 Spark 中广播变量和累加器的基本原理与用途

累加器（`accumulator`）是 Spark 中提供的一种分布式的变量机制，其原理类似于 mapreduce，即分布式的改变，然后聚合这些改变。累加器的一个常见用途是在调试时对作业执行过程中的事件进行计数。而广播变量用来高效分发较大的对象。

共享变量出现的原因：通常在向 Spark 传递函数时，比如使用 `map()` 函数或者用 `filter()` 传条件时，可以使用驱动器程序中定义的变量，但是集群中运行的每个任务都会得到这些变量的一份新的副本，更新这些副本的值也不会影响驱动器中的对应变量。

2.11 如何减少 Spark 运行中的数据库连接数？

使用 `foreachPartition` 代替 `foreach`，在 `foreachPartition` 内获取数据库的连接。

`foreach` 是对 rdd 每个元素的操作，而 `foreachPartition` 是对 rdd 每个分区的操作。

2.12 Spark 如何实现 TopN 的获取

方法 1：

- (1) 按照 key 对数据进行聚合（`groupByKey`）
- (2) 将 value 转换为数组，利用 scala 的 `sortBy` 或者 `sortWith` 进行排序（`mapValues`）数据量太大，会 OOM。

方法 2：

- (1) 取出所有的 key
- (2) 对 key 进行迭代，每次取出一个 key 利用 spark 的排序算子进行排序

方法 3：

- (1) 自定义分区器，按照 key 进行分区，使不同的 key 进到不同的分区
- (2) 对每个分区运用 spark 的排序算子进行排序

2.13 简述 SparkSQL 中 RDD、DataFrame、DataSet 三者的区别与联系？

1) RDD

编译时类型安全，编译时就能检查出类型错误，可以直接通过对对象名点的方式来操作数据。但是序列化和反序列化的性能开销大，无论是集群间的通信，还是 IO 操作都需要对对象的结构和数据进行序列化和反序列化。而且 GC 的性能开销，频繁的创建和销毁对象，势必会增加 GC

2) DataFrame

DataFrame 引入了 schema 和 off-heap。RDD 每一行的数据，结构都是一样的，这个结构就存储在 schema 中。Spark 通过 schema 就能够读懂数据，因此在通信和 IO 时就只需要序列化和反序列化数据，而结构的部分就可以省略了。

3) DataSet

DataSet 结合了 RDD 和 DataFrame 的优点，并带来的一个新的概念 Encoder。当序列化数据时，Encoder 产生字节码与 off-heap 进行交互，能够达到按需访问数据的效果，而不用反序列化整个对象。Spark 还没有提供自定义 Encoder 的 API，但是未来会加入。

2.14 简述 Kryo 序列化

kryo 序列化比 java 序列化更快更紧凑，但 spark 默认的序列化是 java 序列化并不是 kryo 序列化，因为 spark 并不支持所有序列化类型，而且每次使用都必须进行注册，注册只针对于 RDD。在 DataFrames 和 DataSet 当中自动实现了 kryo 序列化。

2.15 简述 BroadCast join

原理：先将小表数据查询出来聚合到 driver 端，再广播到各个 executor 端，使表与表 join 时可以进行本地 join，避免进行网络传输产生 shuffle

使用场景：大表 join 小表，只能广播小表

2.16 SparkSQL 中 join 操作与 left join 操作的区别？

join 和 sql 中的 inner join 操作很相似，返回结果是前面一个集合和后面一个集合中匹配成功的，过滤掉关联不上的。leftJoin 类似于 SQL 中的左外关联 left outer join，返回结果以第一个 RDD 为主，关联不上的记录为空。

部分场景下可以使用 left semi join 替代 left join：因为 left semi join 是 in(keySet) 的关系，遇到右表重复记录，左表会跳过，性能更高，而 left join 则会一直遍历。但是 left semi join 中最后 select 的结果中只许出现左表中的列名，因为右表只有 join key 参与关联计算了

2.17 Spark Streaming 第一次运行不丢数据

kafka 参数 auto.offset.reset 参数设置成 earliest 从最初始偏移量开始消费数据

2.18 Spark Streaming 精准一次消费

- 1) 手动维护偏移量
- 2) 处理完业务数据后，再进行提交偏移量操作

极端情况下，如在提交偏移量时断网或停电会造成 spark 程序第二次启动时重复消费问题，所以在涉及到金额或精确性非常高的场景会使用事物保证精准一次消费。

2.19 Spark Streaming 控制每秒消费数据的速度

通过 spark.streaming.kafka.maxRatePerPartition 参数来设置 Spark Streaming 从 kafka 分区每秒拉取的条数

2.20 Spark Streaming 背压机制

把 spark.streaming.backpressure.enabled 参数设置为 true，开启背压机制后 Spark Streaming 会根据延迟动态去 kafka 消费数据，上限由 spark.streaming.kafka.maxRatePerPartition 参数控制，所以两个参数一般会一起使用

2.21 Spark Streaming 的一个 stage 耗时

Spark Streaming stage 耗时由最慢的 task 决定，所以数据倾斜时某个 task 运行慢会导致整个 Spark Streaming 都运行非常慢。

2.22 Spark Streaming 默认分区个数

Spark Streaming 默认分区个数与所对接的 kafka topic 分区个数一致，Spark Streaming 里一般不会使用 repartition 算子增大分区，因为 repartition 会进行 shuffle 增加耗时

2.23 简述 SparkStreaming 的消费方式以及它们之间的区别

- 1) 基于 Receiver 的方式

这种方式使用 Receiver 来获取数据。Receiver 是使用 Kafka 的高层次 Consumer API 来实现的。receiver 从 Kafka 中获取的数据都是存储在 Spark Executor 的内存中的（如果突然数据暴增，大量 batch 堆积，很容易出现内存溢出的问题），然后 Spark Streaming 启动的 job 会去处理那些数据。

然而，在默认的配置下，这种方式可能会因为底层的失败而丢失数据。如果要启用高可靠机制，让数据零丢失，就必须启用 Spark Streaming 的预写日志机制（Write Ahead Log, WAL）。该机制会同步地将接收到的 Kafka 数据写入分布式文件系统（比如 HDFS）上的预写日志中。所以，即使底层节点出现了失败，也可以使用预写日志中的数据进行恢复。

2) 基于 Direct 的方式

这种新的不基于 Receiver 的直接方式，是在 Spark 1.3 中引入的，从而能够确保更加健壮的机制。替代掉使用 Receiver 来接收数据后，这种方式会周期性地查询 Kafka，来获得每个 topic+partition 的最新的 offset，从而定义每个 batch 的 offset 的范围。当处理数据的 job 启动时，就会使用 Kafka 的简单 consumer api 来获取 Kafka 指定 offset 范围的数据。

优点如下：

简化并行读取：如果要读取多个 partition，不需要创建多个输入 DStream 然后对它们进行 union 操作。Spark 会创建跟 Kafka partition 一样多的 RDD partition，并且会并行从 Kafka 中读取数据。所以在 Kafka partition 和 RDD partition 之间，有一个一对一的映射关系。

高性能：如果要保证零数据丢失，在基于 receiver 的方式中，需要开启 WAL 机制。这种方式其实效率低下，因为数据实际上被复制了两份，Kafka 自己本身就有高可靠的机制，会对数据复制一份，而这里又会复制一份到 WAL 中。而基于 direct 的方式，不依赖 Receiver，不需要开启 WAL 机制，只要 Kafka 中作了数据的复制，那么就可以通过 Kafka 的副本进行恢复。

一次且仅一次的事务机制。

3) 对比：

基于 receiver 的方式，是使用 Kafka 的高阶 API 来在 ZooKeeper 中保存消费过的 offset 的。这是消费 Kafka 数据的传统方式。这种方式配合着 WAL 机制可以保证数据零丢失的高可靠性，但是却无法保证数据被处理一次且仅一次，可能会处理两次。因为 Spark 和 ZooKeeper 之间可能是不同步的。

基于 direct 的方式，使用 kafka 的简单 api，Spark Streaming 自己就负责追踪消费的 offset，并保存在 checkpoint 中。Spark 自己一定是同步的，因此可以保证数据是消费一次且仅消费一次。

在实际生产环境中大都用 Direct 方式。

2.24 简述 SparkStreaming 窗口函数的原理（重点）

窗口函数就是在原来定义的 SparkStreaming 计算批次大小的基础上再次进行封装，每次计算多个批次的数据，同时还需要传递一个滑动步长的参数，用来设置当次计算任务完成之后下一次从什么地方开始计算。

图中 time1 就是 SparkStreaming 计算批次大小，虚线框以及实线大框就是窗口的大小，必须为批次的整数倍。虚线框到大实线框的距离（相隔多少批次），就是滑动步长。

2.25 Spark 数据倾斜

公司一：总用户量 1000 万，5 台 64G 内存的服务器。

公司二：总用户量 10 亿，1000 台 64G 内存的服务器。

1. 公司一的数据分析师在做 join 的时候发生了数据倾斜，会导致有几百万用户的相关数据集中到了一台服务器上，几百万的用户数据，说大也不大，正常字段量的数据的话 64G 还是能轻松处理掉的。

2. 公司二的数据分析师在做 join 的时候也发生了数据倾斜，可能会有 1 个亿的用户相关数据集中到了一台机器上了（相信我，这很常见）。这时候一台机器就很难搞定了，最后会很难算出结果。

2.25.1 数据倾斜表现

1) hadoop 中的数据倾斜表现：

- 有一个或几个 Reduce 卡住，卡在 99.99%，一直不能结束。
- 各种 container 报错 OOM
- 异常的 Reducer 读写的数据量极大，至少远远超过其它正常的 Reducer

- 伴随着数据倾斜，会出现任务被 kill 等各种诡异的表现。

2) hive 中数据倾斜

一般都发生在 Sql 中 group by 和 join on 上，而且和数据逻辑绑定比较深。

3) Spark 中的数据倾斜

Spark 中的数据倾斜，包括 Spark Streaming 和 Spark Sql，表现主要有下面几种：

- Executor lost, OOM, Shuffle 过程出错；
- Driver OOM；
- 单个 Executor 执行时间特别久，整体任务卡在某个阶段不能结束；
- 正常运行的任务突然失败；

2.25.2 数据倾斜产生原因

我们以 Spark 和 Hive 的使用场景为例。

他们在做数据运算的时候会涉及到，count distinct、group by、join on 等操作，这些都会触发 Shuffle 动作。一旦触发 Shuffle，所有相同 key 的值就会被拉到一个或几个 Reducer 节点上，容易发生单点计算问题，导致数据倾斜。

一般来说，数据倾斜原因有以下几方面：

1) key 分布不均匀；

存在一些 key 的数量非常多

2) 建表时考虑不周

我们举一个例子，就说数据默认值的设计吧，假设我们有两张表：

user (用户信息表) : userid, register_ip

ip (IP 表) : ip, register_user_cnt

这可能是两个不同的人开发的数据表。如果我们的数据规范不太完善的话，会出现一种情况：user 表中的 register_ip 字段，如果获取不到这个信息，我们认为 null；但是在 ip 表中，我们在统计这个值的时候，为了方便，我们把获取不到 ip 的用户，统一认为他们的 ip 为 0。两边其实都没有错的，但是一旦我们做关联了，这个任务会在做关联的阶段，也就是 sql 的 on 的阶段卡死。

3) 业务数据激增

比如订单场景，我们在某一天在北京和上海两个城市多了强力的推广，结果可能是这两个城市的订单量增长了 10000%，其余城市的数据量不变。

然后我们要统计不同城市的订单情况，这样做 group 操作，可能直接就数据倾斜了。

2.25.3 解决数据倾斜思路

很多数据倾斜的问题，都可以用和平台无关的方式解决，比如更好的数据预处理，异常值的过滤等。因此，解决数据倾斜的重点在于对数据设计和业务的理解，这两个搞清楚了，数据倾斜就解决了大部分了。

1) 业务逻辑

我们从业务逻辑的层面上来优化数据倾斜，比如上面的两个城市做推广活动导致那两个城市数据量激增的例子，我们可以单独对这两个城市来做 count，单独做时可用两次 MR，第一次打散计算，第二次再最终聚合计算。完成后和其它城市做整合。

2) 程序层面

比如说在 Hive 中，经常遇到 count(distinct) 操作，这样会导致最终只有一个 Reduce 任务。

我们可以先 group by，再在外面包一层 count，就可以了。比如计算按用户名去重后的总用户量：

(1) 优化前只有一个 reduce，先去重再 count 负担比较大：

```
select name,count(distinct name)from user;
```

(2) 优化后

// 设置该任务的每个 job 的 reducer 个数为 3 个。Hive 默认-1，自动推断。

```
set mapred.reduce.tasks=3;
```

// 启动两个 job，一个负责子查询(可以有多个 reduce)，另一个负责 count(1)：

```
select count(1) from (select name from user group by name) tmp;
```

3) 调参方面

4) 从业务和数据上解决数据倾斜

很多数据倾斜都是在数据的使用上造成的。我们举几个场景，并分别给出它们的解决方案。

- 有损的方法：找到异常数据，比如 ip 为 0 的数据，过滤掉
- 无损的方法：对分布不均匀的数据，单独计算
- 先对 key 做一层 hash，先将数据随机打散让它的并行度变大，再汇集
- 数据预处理

2.25.4 定位导致数据倾斜代码

Spark 数据倾斜只会发生在 shuffle 过程中。

这里给大家罗列一些常用的并且可能会触发 shuffle 操作的算子：`distinct`、`groupByKey`、`reduceByKey`、`aggregateByKey`、`join`、`cogroup`、`repartition` 等。

出现数据倾斜时，可能就是你的代码中使用了这些算子中的某一个所导致的。

2.25.4.1 某个 task 执行特别慢的情况

首先要看的，就是数据倾斜发生在第几个 stage 中：如果是用 `yarn-client` 模式提交，那么在提交的机器本地是直接可以看到 log，可以在 log 中找到当前运行到了第几个 stage；如果是用 `yarn-cluster` 模式提交，则可以通过 Spark Web UI 来查看当前运行到了第几个 stage。此外，无论是使用 `yarn-client` 模式还是 `yarn-cluster` 模式，我们都可以在 Spark Web UI 上深入看一下当前这个 stage 各个 task 分配的数据量，从而进一步确定是不是 task 分配的数据不均匀导致了数据倾斜。

其次是看 task 运行时间和数据量：

task 运行时间：比如下图中，倒数第三列显示了每个 task 的运行时间。明显可以看到，有的 task 运行特别快，只需要几秒钟就可以运行完；而有的 task 运行特别慢，需要几分钟才能运行完，此时单从运行时间上看就已经能够确定发生数据倾斜了。

task 数据量：此外，倒数第一列显示了每个 task 处理的数据量，明显可以看到，运行时间特别短的 task 只需要处理几百 KB 的数据即可，而运行时间特别长的 task 需要处理几千 KB 的数据，处理的数据量差了 10 倍。此时更加能够确定是发生了数据倾斜。

推断倾斜代码：知道数据倾斜发生在哪一个 stage 之后，接着我们就需要根据 stage 划分原理，推算出来发生倾斜的那个 stage 对应代码中的哪一部分，这部分代码中肯定会有个 shuffle 类算子。精准推算 stage 与代码的对应关系，需要对 Spark 的源码有深入的理解，这里我们可以介绍一个相对简单实用的推算方法：只要看到 Spark 代码中出现了一个 shuffle 类算子或者是 Spark SQL 的 SQL 语句中出现了会导致 shuffle 的语句（比如 `group by` 语句），那么就可以判定，以那个地方为界限划分出了前后两个 stage。

这里我们就以如下单词计数来举例。

```
val conf = new SparkConf()
val sc = new SparkContext(conf)
val lines = sc.textFile("hdfs://...")
val words = lines.flatMap(_.split(" "))
val pairs = words.map(_,_)
val wordCounts = pairs.reduceByKey(_ + _).wordCounts.collect().foreach(println(_))
```

在整个代码中只有一个 `reduceByKey` 是会发生 shuffle 的算子，也就是说这个算子为界限划分出了前后两个 stage：

stage0，主要是执行从 `textFile` 到 `map` 操作以及 `shuffle write` 操作（对 `pairs` RDD 中的数据进行分区操作，每个 task 处理的数据中，相同的 key 会写入同一个磁盘文件内）。

stage1，主要是执行从 `reduceByKey` 到 `collect` 操作以及 `stage1` 的各个 task 一开始运行，就会首先执行 `shuffle read` 操作（会从 `stage0` 的各个 task 所在节点拉取属于自己处理的那些 key，然后对同一个 key 进行全局性的聚合或 `join` 等操作，在这里就是对 key 的 value 值进行累加）

`stage1` 在执行完 `reduceByKey` 算子之后，就计算出了最终的 `wordCounts` RDD，然后会执行 `collect` 算子，将所有数据拉取到 Driver 上，供我们遍历和打印输出。

123456789

通过对单词计数程序的分析，希望能够让大家了解最基本的 stage 划分的原理，以及 stage 划分后 shuffle 操作是如何在两个 stage 的边界处执行的。然后我们就知道如何快速定位出发生数据倾斜的 stage 对应代码的哪一个部分了。

比如我们在 Spark Web UI 或者本地 log 中发现，stage1 的某几个 task 执行得特别慢，判定 stage1 出现了数据倾斜，那么就可以回到代码中，定位出 stage1 主要包括了 reduceByKey 这个 shuffle 类算子，此时基本就可以确定是该算子导致了数据倾斜问题。

此时，如果某个单词出现了 100 万次，其他单词才出现 10 次，那么 stage1 的某个 task 就要处理 100 万数据，整个 stage 的速度就会被这个 task 拖慢。

2.25.4.2 某个 task 莫名其妙内存溢出的情况

这种情况下去定位出问题的代码就比较容易了。我们建议直接看 yarn-client 模式下本地 log 的异常栈，或者是通过 YARN 查看 yarn-cluster 模式下的 log 中的异常栈。一般来说，通过异常栈信息就可以定位到你的代码中哪一行发生了内存溢出。然后在那行代码附近找找，一般也会有 shuffle 类算子，此时很可能就是这个算子导致了数据倾斜。但是大家要注意的是，不能单纯靠偶然的内存溢出就判定发生了数据倾斜。因为自己编写的代码的 bug，以及偶然出现的数据异常，也可能导致内存溢出。因此还是要按照上面所讲的方法，通过 Spark Web UI 查看报错的那个 stage 的各个 task 的运行时间以及分配的数据量，才能确定是否是由于数据倾斜才导致了这次内存溢出。

2.25.5 查看导致数据倾斜的 key 分布情况

先对 pairs 采样 10% 的样本数据，然后使用 countByKey 算子统计出每个 key 出现的次数，最后在客户端遍历和打印样本数据中各个 key 的出现次数。

```
val sampledPairs = pairs.sample(false, 0.1)
val sampledWordCounts = sampledPairs.countByKey()
sampledWordCounts.foreach(println(_))
```

2.25.6 Spark 数据倾斜的解决方案

2.25.6.1 使用 Hive ETL 预处理数据

1) 适用场景

导致数据倾斜的是 Hive 表。如果该 Hive 表中的数据本身很不均匀（比如某个 key 对应了 100 万数据，其他 key 才对应了 10 条数据），而且业务场景需要频繁使用 Spark 对 Hive 表执行某个分析操作，那么比较适合使用这种技术方案。

2) 实现思路

此时可以评估一下，是否可以通过 Hive 来进行数据预处理（即通过 Hive ETL 预先对数据按照 key 进行聚合，或者是预先和其他表进行 join），然后在 Spark 作业中针对的数据源就不是原来的 Hive 表了，而是预处理后的 Hive 表。此时由于数据已经预先进行过聚合或 join 操作了，那么在 Spark 作业中也就不需要使用原先的 shuffle 类算子执行这类操作了。

3) 方案实现原理

这种方案从根源上解决了数据倾斜，因为彻底避免了在 Spark 中执行 shuffle 类算子，那么肯定就不会有数据倾斜的问题了。但是这里也要提醒一下大家，这种方式属于治标不治本。因为毕竟数据本身就存在分布不均匀的问题，所以 Hive ETL 中进行 group by 或者 join 等 shuffle 操作时，还是会出现数据倾斜，导致 Hive ETL 的速度很慢。我们只是把数据倾斜的发生提前到了 Hive ETL 中，避免 Spark 程序发生数据倾斜而已。

4) 方案优缺点

优点：实现起来简单便捷，效果还非常好，完全规避掉了数据倾斜，Spark 作业的性能会大幅度提升。

缺点：治标不治本，Hive ETL 中还是会发数据倾斜。

5) 方案实践经验

在一些 Java 系统与 Spark 结合使用的项目中，会出现 Java 代码频繁调用 Spark 作业的场景，而且对 Spark 作业的执行性能要求很高，就比较适合使用这种方案。将数据倾斜提前到上游的 Hive ETL，每天仅执行一次，只有那一次是比较慢的，而之后每次 Java 调用 Spark 作业时，执行速度都会很快，能够提供更好的用户体验。

6) 项目实践经验

在美团·点评的交互式用户行为分析系统中使用了这种方案，该系统主要是允许用户通过 Java Web 系统提交数据分析统计任务，后端通过 Java 提交 Spark 作业进行数据分析统计。要求 Spark 作业速度必须要快，尽量在 10 分钟以内，否则速度太慢，用户体验会很差。所以我们将有些 Spark 作业的 shuffle 操作提前到了 Hive ETL 中，从而让 Spark 直接使用预处理的 Hive 中间表，尽可能地减少 Spark 的 shuffle 操作，大幅度提升了性能，将部分作业的性能提升了 6 倍以上。

2.25.6.2 过滤少数导致倾斜的 key

1) 方案适用场景

如果发现导致倾斜的 key 就少数几个，而且对计算本身的影响并不大的话，那么很适合使用这种方案。比如 99% 的 key 就对应 10 条数据，但是只有一个 key 对应了 100 万数据，从而导致了数据倾斜。

2) 方案实现思路

如果我们判断那少数几个数据量特别多的 key，对作业的执行和计算结果不是特别重要的话，那么干脆就直接过滤掉那少数几个 key。比如，在 Spark SQL 中可以使用 where 子句过滤掉这些 key 或者在 Spark Core 中对 RDD 执行 filter 算子过滤掉这些 key。如果需要每次作业执行时，动态判定哪些 key 的数据量最多然后再进行过滤，那么可以使用 sample 算子对 RDD 进行采样，然后计算出每个 key 的数量，取数据量最多的 key 过滤掉即可。

3) 方案实现原理

将导致数据倾斜的 key 给过滤掉之后，这些 key 就不会参与计算了，自然不可能产生数据倾斜。

4) 方案优缺点

优点：实现简单，而且效果也很好，可以完全规避掉数据倾斜。

缺点：适用场景不多，大多数情况下，导致倾斜的 key 还是很多的，并不是只有少数几个。

5) 方案实践经验

在项目中我们也采用过这种方案解决数据倾斜。有一次发现某一天 Spark 作业在运行的时候突然 OOM 了，追查之后发现，是 Hive 表中的某一个 key 在那天数据异常，导致数据量暴增。因此就采取每次执行前先进行采样，计算出样本中数据量最大的几个 key 之后，直接在程序中将那些 key 给过滤掉。

2.25.6.3 提高 shuffle 操作的并行度

1) 方案适用场景

如果我们要对数据倾斜迎难而上，那么建议优先使用这种方案，因为这是处理数据倾斜最简单的一种方案。

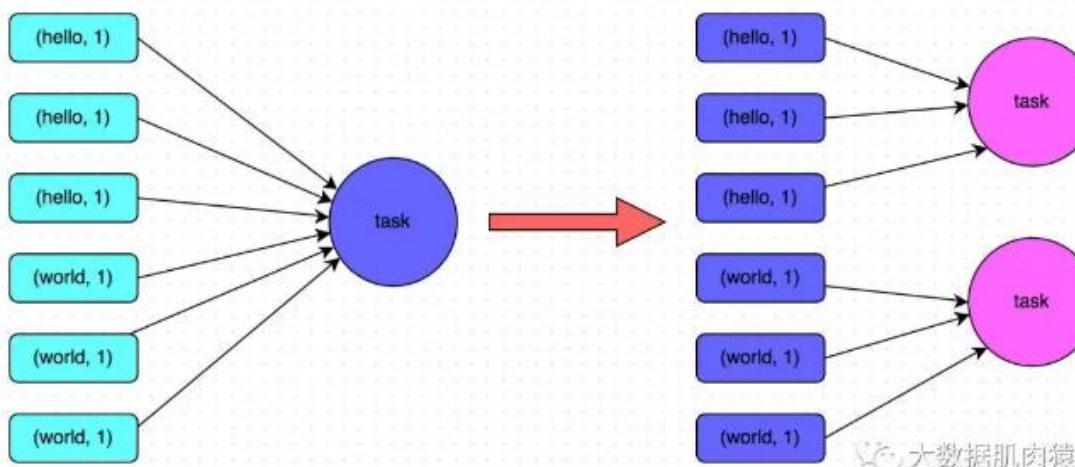
2) 方案实现思路

在对 RDD 执行 shuffle 算子时，给 shuffle 算子传入一个参数，比如 reduceByKey(1000)，该参数就设置了这个 shuffle 算子执行时 shuffle read task 的数量，即 spark.sql.shuffle.partitions，该参数代表了 shuffle read task 的并行度，默认是 200，对于很多场景来说都有点过小。

3) 方案实现原理

增加 shuffle read task 的数量，可以让原本分配给一个 task 的多个 key 分配给多个 task，从而让每个 task 处理比原来更少的数据。举例来说，如果原本有 5 个 key，每个 key 对应 10 条数据，这 5 个 key 都是分配给一个 task 的，那么这个 task 就要处理 50 条数据。

而增加了 shuffle read task 以后，每个 task 就分配到一个 key，即每个 task 就处理 10 条数据，那么自然每个 task 的执行时间都会变短了。具体原理如下图所示。



4) 方案优缺点

优点：实现起来比较简单，可以有效缓解和减轻数据倾斜的影响。

缺点：只是缓解了数据倾斜而已，没有彻底根除问题，根据实践经验来看，其效果有限。

5) 方案实践经验

该方案通常无法彻底解决数据倾斜，因为如果出现一些极端情况，比如某个 key 对应的数据量有 100 万，那么无论你的 task 数量增加到多少，这个对应着 100 万数据的 key 肯定还是会分配到一个 task 中去处理，因此注定还是会发生在数据倾斜的。所以这种方案只能说是在发现数据倾斜时尝试使用的第一种手段，尝试去用最简单的方法缓解数据倾斜而已，或者是和其他方案结合起来使用。

2.25.6.4 两阶段聚合（局部聚合+全局聚合）

1) 方案适用场景

对 RDD 执行 reduceByKey 等聚合类 shuffle 算子或者在 Spark SQL 中使用 group by 语句进行分组聚合时，比较适用这种方案。

2) 方案实现思路

这个方案的核心实现思路就是进行两阶段聚合：

第一次是局部聚合，先给每个 key 都打上一个随机数，比如 10 以内的随机数，此时原先一样的 key 就变成不一样的了，比如(hello, 1) (hello, 1) (hello, 1) (hello, 1)，就会变成(1_hello, 1) (1_hello, 1) (2_hello, 1) (2_hello, 1)。

接着对打上随机数后的数据，执行 reduceByKey 等聚合操作，进行局部聚合，那么局部聚合结果，就会变成了(1_hello, 2) (2_hello, 2)。

然后将各个 key 的前缀给去掉，就会变成(hello,2)(hello,2)，再次进行全局聚合操作，就可以得到最终结果了，比如(hello, 4)。

示例代码如下：

```
// 第一步，给 RDD 中的每个 key 都打上一个随机前缀。
JavaPairRDD<String, Long> randomPrefixRdd = rdd.mapToPair(
    new PairFunction<Tuple2<Long, Long>, String, Long>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<String, Long> call(Tuple2<Long, Long> tuple)
            throws Exception {
            Random random = new Random();
            int prefix = random.nextInt(10);
            return new Tuple2<String, Long>(prefix + "_" + tuple._1, tuple._2);
        }
    });
}

// 第二步，对打上随机前缀的 key 进行局部聚合。
```

```
JavaPairRDD<String, Long> localAggrRdd = randomPrefixRdd.reduceByKey(
    new Function2<Long, Long, Long>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Long call(Long v1, Long v2) throws Exception {
            return v1 + v2;
        }
    });
}

// 第三步，去除 RDD 中每个 key 的随机前缀。
```

```
JavaPairRDD<Long, Long> removedRandomPrefixRdd = localAggrRdd.mapToPair(
    new PairFunction<Tuple2<String, Long>, Long, Long>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<Long, Long> call(Tuple2<String, Long> tuple)
            throws Exception {
```

```

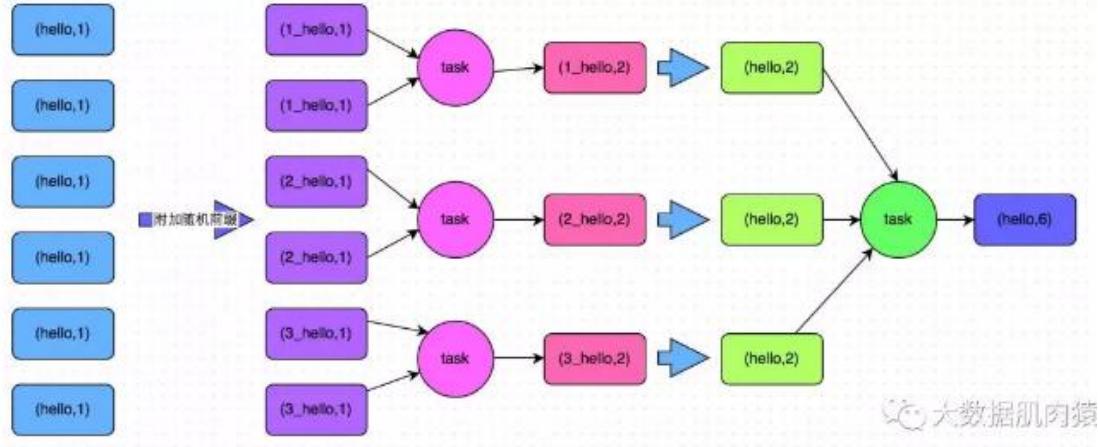
        long originalKey = Long.valueOf(tuple._1.split(" ")[1]);
        return new Tuple2<Long, Long>(originalKey, tuple._2);
    }
});

// 第四步，对去除了随机前缀的 RDD 进行全局聚合。
JavaPairRDD<Long, Long> globalAggrRdd = removedRandomPrefixRdd.reduceByKey(
    new Function2<Long, Long, Long>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Long call(Long v1, Long v2) throws Exception {
            return v1 + v2;
        }
    });

```

3) 方案实现原理

将原本相同的 key 通过附加随机前缀的方式，变成多个不同的 key，就可以让原本被一个 task 处理的数据分散到多个 task 上去做局部聚合，进而解决单个 task 处理数据量过多的问题。接着去除掉随机前缀，再次进行全局聚合，就可以得到最终的结果。具体原理见下图。



4) 方案优缺点

优点

对于聚合类的 shuffle 操作导致的数据倾斜，效果是非常不错的。通常都可以解决掉数据倾斜，或者至少是大幅度缓解数据倾斜，将 Spark 作业的性能提升数倍以上。

缺点

仅仅适用于聚合类的 shuffle 操作，适用范围相对较窄。如果是 join 类的 shuffle 操作，还得用其他的解决方案。

2.25.6.5 将 reduce join 转为 map join

1) 方案适用场景

在对 RDD 使用 join 类操作，或者是在 Spark SQL 中使用 join 语句时，而且 join 操作中的一个 RDD 或表的数据量比较小（比如几百 M 或者一两 G），比较适用此方案。

2) 方案实现思路

不使用 join 算子进行连接操作，而使用 Broadcast 变量与 map 类算子实现 join 操作，进而完全规避掉 shuffle 类的操作，彻底避免数据倾斜的发生和出现。将较小 RDD 中的数据直接通过 collect 算子拉取到 Driver 端的内存中来，然后对其创建一个 Broadcast 变量，广播给其他 Executor 节点；

接着对另外一个 RDD 执行 map 类算子，在算子函数内，从 Broadcast 变量中获取较小 RDD 的全量数据，与当前 RDD 的每一条数据按照连接 key 进行比对，如果连接 key 相同的话，那么就将两个 RDD 的数据用你需要的方式连接起来。示例如下：

```

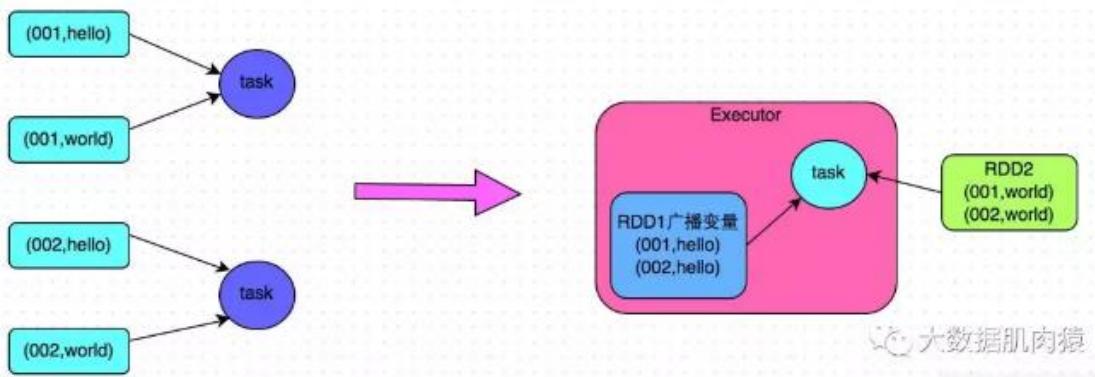
// 首先将数据量比较小的 RDD 的数据，collect 到 Driver 中来。
List<Tuple2<Long, Row>> rdd1Data = rdd1.collect()
// 然后使用 Spark 的广播功能，将小 RDD 的数据转换成广播变量，这样每个 Executor 就只有一份 RDD 的数据。
// 可以尽可能节省内存空间，并且减少网络传输性能开销。
final Broadcast<List<Tuple2<Long, Row>>> rdd1DataBroadcast = sc.broadcast(rdd1Data);

// 对另外一个 RDD 执行 map 类操作，而不再是 join 类操作。
JavaPairRDD<String, Tuple2<String, Row>> joinedRdd = rdd2.mapToPair(
    new PairFunction<Tuple2<Long, String>, String, Tuple2<String, Row>>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<String, Tuple2<String, Row>> call(Tuple2<Long, String> tuple)
            throws Exception {
            // 在算子函数中，通过广播变量，获取到本地 Executor 中的 rdd1 数据。
            List<Tuple2<Long, Row>> rdd1Data = rdd1DataBroadcast.value();
            // 可以将 rdd1 的数据转换为一个 Map，便于后面进行 join 操作。
            Map<Long, Row> rdd1DataMap = new HashMap<Long, Row>();
            for(Tuple2<Long, Row> data : rdd1Data) {
                rdd1DataMap.put(data._1, data._2);
            }
            // 获取当前 RDD 数据的 key 以及 value。
            String key = tuple._1;
            String value = tuple._2;
            // 从 rdd1 数据 Map 中，根据 key 获取到可以 join 到的数据。
            Row rdd1Value = rdd1DataMap.get(key);
            return new Tuple2<String, String>(key, new Tuple2<String, Row>(value, rdd1Value));
        }
    });
// 这里得提示一下。
// 上面的做法，仅仅适用于 rdd1 中的 key 没有重复，全部是唯一的场景。
// 如果 rdd1 中有多个相同的 key，那么就得用 flatMap 类的操作，在进行 join 的时候不能用 map，而是得遍历 rdd1
所有数据进行 join。
// rdd2 中每条数据都可能会返回多条 join 后的数据。

```

3) 方案实现原理

普通的 join 是会走 shuffle 过程的，而一旦 shuffle，就相当于会将相同 key 的数据拉取到一个 shuffle read task 中再进行 join，此时就是 reduce join。但是如果一个 RDD 是比较小的，则可以采用广播小 RDD 全量数据+map 算子来实现与 join 同样的效果，也就是 map join，此时就不会发生 shuffle 操作，也就不会发生数据倾斜。具体原理如下图所示。



4) 方案优缺点

优点：对 join 操作导致的数据倾斜，效果非常好，因为根本就不会发生 shuffle，也就根本不会发生数据倾斜。

缺点：适用场景较少，因为这个方案只适用于一个大表和一个小表的情况。毕竟我们需要将小表进行广播，此时会比较消耗内存资源，driver 和每个 Executor 内存中都会驻留一份小 RDD 的全量数据。如果我们广播出去的 RDD 数据比较大，比如 10G 以上，那么就可能发生内存溢出了。因此并不适合两个都是大表的情况。

2.25.6.6 采样倾斜 key 并分拆 join 操作

1) 方案适用场景

两个 RDD/Hive 表进行 join 的时候，如果数据量都比较大，无法采用“解决方案五”，那么此时可以看一下两个 RDD/Hive 表中的 key 分布情况。

如果出现数据倾斜，是因为其中某一个 RDD/Hive 表中的少数几个 key 的数据量过大，而另一个 RDD/Hive 表中的所有 key 都分布比较均匀，那么采用这个解决方案是比较合适的。

2) 方案实现思路

对包含少数几个数据量过大的 key 的那个 RDD，通过 sample 算子采样出一份样本来，然后统计一下每个 key 的数量，计算出来数据量最大的是哪几个 key。

然后将这几个 key 对应的数据从原来的 RDD 中拆分出来，形成一个单独的 RDD，并给每个 key 都打上 n 以内的随机数作为前缀；

而不会导致倾斜的大部分 key 形成另外一个 RDD。

接着将需要 join 的另一个 RDD，也过滤出来那几个倾斜 key 对应的数据并形成一个单独的 RDD，将每条数据膨胀成 n 条数据，这 n 条数据都按顺序附加一个 0~n 的前缀；

不会导致倾斜的大部分 key 也形成另外一个 RDD。

再将附加了随机前缀的独立 RDD 与另一个膨胀 n 倍的独立 RDD 进行 join，此时就可以将原先相同的 key 打散成 n 份，分散到多个 task 中去进行 join 了。

而另外两个普通的 RDD 就照常 join 即可。

最后将两次 join 的结果使用 union 算子合并起来即可，就是最终的 join 结果。

示例如下：

```
// 首先从包含了少数几个导致数据倾斜 key 的 rdd1 中，采样 10% 的样本数据。
```

```
JavaPairRDD<Long, String> sampledRDD = rdd1.sample(false, 0.1);
```

```
// 对样本数据 RDD 统计出每个 key 的出现次数，并按出现次数降序排序。
```

```
// 对降序排序后的数据，取出 top 1 或者 top 100 的数据，也就是 key 最多的前 n 个数据。
```

```
// 具体取出多少个数据量最多的 key，由大家自己决定，我们这里就取 1 个作为示范。
```

```
// 每行数据变为<key,1>
```

```
JavaPairRDD<Long, Long> mappedSampledRDD = sampledRDD.mapToPair(
```

```
    new PairFunction<Tuple2<Long, String>, Long, Long>() {
```

```
        private static final long serialVersionUID = 1L;
```

```
        @Override
```

```

        public Tuple2<Long, Long> call(Tuple2<Long, String> tuple)
            throws Exception {
            return new Tuple2<Long, Long>(tuple._1, 1L);
        }
    });

// 按 key 累加行数
JavaPairRDD<Long, Long> countedSampledRDD = mappedSampledRDD.reduceByKey(
    new Function2<Long, Long, Long>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Long call(Long v1, Long v2) throws Exception {
            return v1 + v2;
        }
    });
}

// 反转 key 和 value,变为<value,key>
JavaPairRDD<Long, Long> reversedSampledRDD = countedSampledRDD.mapToPair(
    new PairFunction<Tuple2<Long,Long>, Long, Long>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<Long, Long> call(Tuple2<Long, Long> tuple)
            throws Exception {
            return new Tuple2<Long, Long>(tuple._2, tuple._1);
        }
    });
}

// 以行数排序 key, 取最多行数的 key
final Long skewedUserId = reversedSampledRDD.sortByKey(false).take(1).get(0)._2;

// 从 rdd1 中分拆出导致数据倾斜的 key, 形成独立的 RDD。
JavaPairRDD<Long, String> skewedRDD = rdd1.filter(
    new Function<Tuple2<Long,String>, Boolean>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Boolean call(Tuple2<Long, String> tuple) throws Exception {
            return tuple._1.equals(skewedUserId);
        }
    });
}

// 从 rdd1 中分拆出不导致数据倾斜的普通 key, 形成独立的 RDD。
JavaPairRDD<Long, String> commonRDD = rdd1.filter(
    new Function<Tuple2<Long,String>, Boolean>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Boolean call(Tuple2<Long, String> tuple) throws Exception {
            return !tuple._1.equals(skewedUserId);
        }
    });

```

```

    }
});

// rdd2, 就是那个所有 key 的分布相对较为均匀的 rdd。
// 这里将 rdd2 中, 前面获取到的 key 对应的数据, 过滤出来, 分拆成单独的 rdd, 并对 rdd 中的数据使用 flatMap
// 算子都扩容 100 倍。
// 对扩容的每条数据, 都打上 0~100 的前缀。
JavaPairRDD<String, Row> skewedRdd2 = rdd2.filter(
    new Function<Tuple2<Long,Row>, Boolean>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Boolean call(Tuple2<Long, Row> tuple) throws Exception {
            return tuple._1.equals(skewedUserId);
        }
    }).flatMapToPair(new PairFlatMapFunction<Tuple2<Long,Row>, String, Row>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Iterable<Tuple2<String, Row>> call(
            Tuple2<Long, Row> tuple) throws Exception {
            Random random = new Random();
            List<Tuple2<String, Row>> list = new ArrayList<Tuple2<String, Row>>();
            for(int i = 0; i < 100; i++) {
                list.add(new Tuple2<String, Row>(i + "_" + tuple._1, tuple._2));
            }
            return list;
        }
    });

// 将 rdd1 中分拆出来的导致倾斜的 key 的独立 rdd, 每条数据都打上 100 以内的随机前缀。
// 然后将这个 rdd1 中分拆出来的独立 rdd, 与上面 rdd2 中分拆出来的独立 rdd, 进行 join。
JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD1 = skewedRDD.mapToPair(
    new PairFunction<Tuple2<Long,String>, String, String>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<String, String> call(Tuple2<Long, String> tuple)
            throws Exception {
            Random random = new Random();
            int prefix = random.nextInt(100);
            return new Tuple2<String, String>(prefix + "_" + tuple._1, tuple._2);
        }
    })
    .join(skewedUserId2infoRDD)
    .mapToPair(new PairFunction<Tuple2<String,Tuple2<String,Row>>, Long, Tuple2<String, Row>>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<Long, Tuple2<String, Row>> call(

```

```

        Tuple2<String, Tuple2<String, Row>> tuple)
    throws Exception {
    long key = Long.valueOf(tuple._1.split("_")[1]);
    return new Tuple2<Long, Tuple2<String, Row>>(key, tuple._2);
}
});

```

// 将 rdd1 中分拆出来的包含普通 key 的独立 rdd，直接与 rdd2 进行 join。

```
JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD2 = commonRDD.join(rdd2);
```

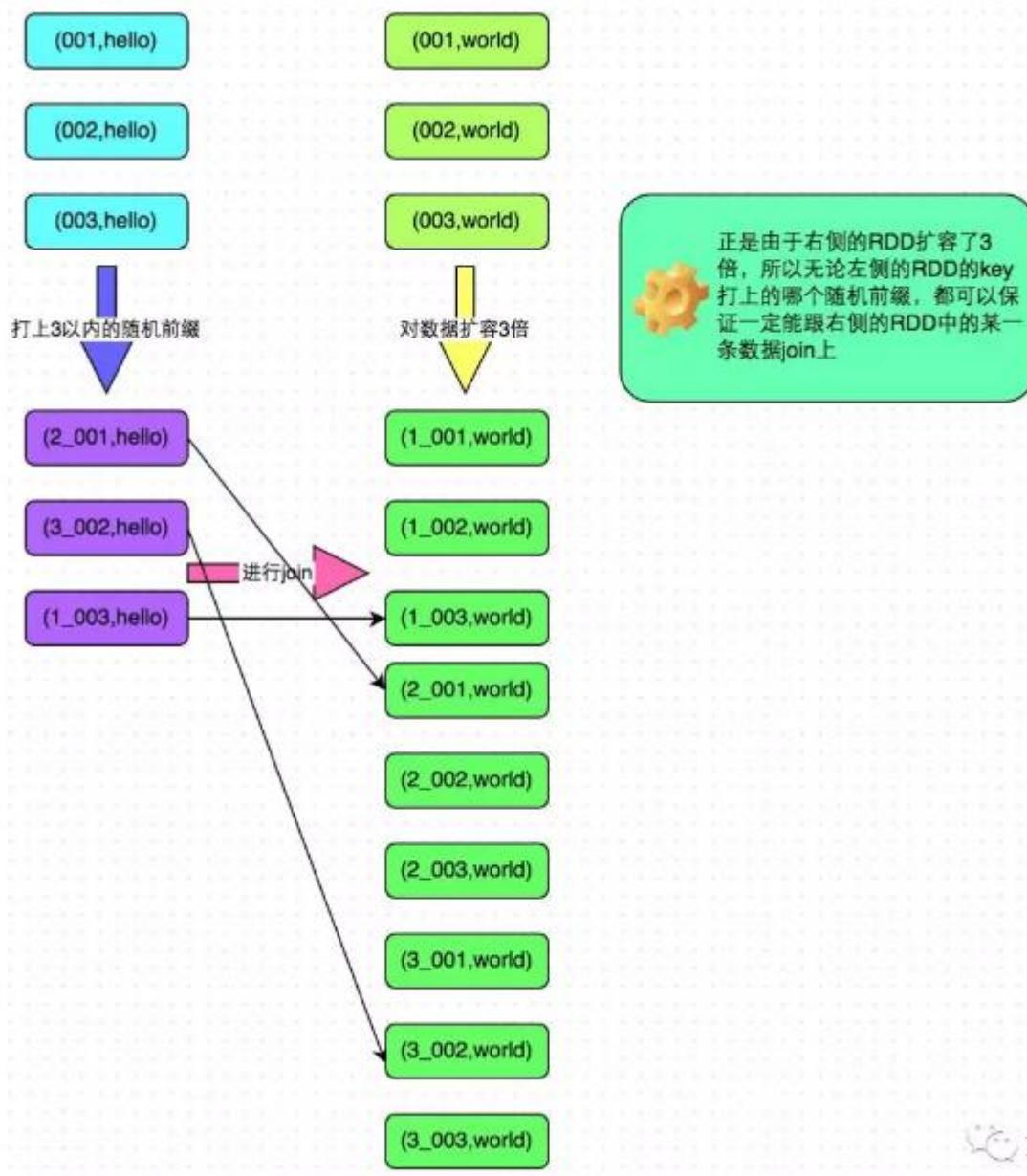
// 将倾斜 key join 后的结果与普通 key join 后的结果， union 起来。

// 就是最终的 join 结果。

```
JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD = joinedRDD1.union(joinedRDD2);
```

3) 方案实现原理

对于 join 导致的数据倾斜，如果只是某几个 key 导致了倾斜，可以将少数几个 key 分拆成独立 RDD，并附加随机前缀打散成 n 份去进行 join，此时这几个 key 对应的数据就不会集中在少数几个 task 上，而是分散到多个 task 进行 join 了。具体原理见下图。



4) 方案优缺点

优点：对于 join 导致的数据倾斜，如果只是某几个 key 导致了倾斜，采用该方式可以用最有效的方式打散 key 进行 join。而且只需要针对少数倾斜 key 对应的数据进行扩容 n 倍，不需要对全量数据进行扩容。避免了占用过多内存。
缺点：如果导致倾斜的 key 特别多的话，比如成千上万个 key 都导致数据倾斜，那么这种方式也不适合。

2.25.6.7 使用随机前缀和扩容 RDD 进行 join

1) 方案适用场景

如果在进行 join 操作时，RDD 中有大量的 key 导致数据倾斜，那么进行分拆 key 也没什么意义，此时就只能使用最后一种方案来解决问题了。

2) 方案实现思路

该方案的实现思路基本和“解决方案六”类似，首先查看 RDD/Hive 表中的数据分布情况，找到那个造成数据倾斜的 RDD/Hive 表，比如有多个 key 都对应了超过 1 万条数据。

然后将该 RDD 的每条数据都打上一个 n 以内的随机前缀。

同时对另外一个正常的 RDD 进行扩容，将每条数据都扩容成 n 条数据，扩容出来的每条数据都依次打上一个 0~n 的前缀。

最后将两个处理后的 RDD 进行 join 即可。

示例代码如下：

```
// 首先将其中一个 key 分布相对较为均匀的 RDD 膨胀 100 倍。
JavaPairRDD<String, Row> expandedRDD = rdd1.flatMapToPair(
    new PairFlatMapFunction<Tuple2<Long, Row>, String, Row>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Iterable<Tuple2<String, Row>> call(Tuple2<Long, Row> tuple)
            throws Exception {
            List<Tuple2<String, Row>> list = new ArrayList<Tuple2<String, Row>>();
            for(int i = 0; i < 100; i++) {
                list.add(new Tuple2<String, Row>(0 + "_" + tuple._1, tuple._2));
            }
            return list;
        }
    });
// 其次，将另一个有数据倾斜 key 的 RDD，每条数据都打上 100 以内的随机前缀。
JavaPairRDD<String, String> mappedRDD = rdd2.mapToPair(
    new PairFunction<Tuple2<Long, String>, String, String>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<String, String> call(Tuple2<Long, String> tuple)
            throws Exception {
            Random random = new Random();
            int prefix = random.nextInt(100);
            return new Tuple2<String, String>(prefix + "_" + tuple._1, tuple._2);
        }
    });
// 将两个处理后的 RDD 进行 join 即可。
JavaPairRDD<String, Tuple2<String, Row>> joinedRDD = mappedRDD.join(expandedRDD);
```

3) 方案实现原理

将原先一样的 key 通过附加随机前缀变成不一样的 key，然后就可以将这些处理后的“不同 key”分散到多个 task 中去处理，而不是让一个 task 处理大量的相同 key。

该方案与“解决方案六”的不同之处就在于，上一种方案是尽量只对少数倾斜 key 对应的数据进行特殊处理，由于处理过程需要扩容 RDD，因此上一种方案扩容 RDD 后对内存的占用并不大；

而这一种方案是针对有大量倾斜 key 的情况，没法将部分 key 拆分出来进行单独处理，因此只能对整个 RDD 进行数据扩容，对内存资源要求很高。

4) 方案优缺点

优点：对 join 类型的数据倾斜基本都可以处理，而且效果也相对比较显著，性能提升效果非常不错。

缺点：该方案更多的是缓解数据倾斜，而不是彻底避免数据倾斜。而且需要对整个 RDD 进行扩容，对内存资源要求很高。

5) 方案实践经验

曾经开发一个数据需求的时候，发现一个 join 导致了数据倾斜。优化之前，作业的执行时间大约是 60 分钟左右；使用该方案优化之后，执行时间缩短到 10 分钟左右，性能提升了 6 倍。

2.25.6.8 多种方案组合使用

在实践中发现，很多情况下，如果只是处理较为简单的数据倾斜场景，那么使用上述方案中的某一种基本就可以解决。但是如果要处理一个较为复杂的数据倾斜场景，那么可能需要将多种方案组合起来使用。

比如说，我们针对出现了多个数据倾斜环节的 Spark 作业，可以先运用解决方案一 HiveETL 预处理和过滤少数导致倾斜的 k，预处理一部分数据，并过滤一部分数据来缓解；

其次可以对某些 shuffle 操作提升并行度，优化其性能；

最后还可以针对不同的聚合或 join 操作，选择一种方案来优化其性能。

大家需要对这些方案的思路和原理都透彻理解之后，在实践中根据各种不同的情况，灵活运用多种方案，来解决自己的数据倾斜问题。

2.26 手写 Spark wordcount

```
val conf: SparkConf = new SparkConf().setMaster("local[*]").setAppName("WordCount")
val sc = new SparkContext(conf)
sc.textFile("/input")
  .flatMap(_.split(" "))
  .map((_, 1))
  .reduceByKey(_ + _)
  .saveAsTextFile("/output")
sc.stop()
```