

第一章 分布式计算框架与资源调度.....	8
1.1 分布式计算框架.....	8
1.1.1编程模型.....	8
1.1.2特殊的组件 partitioner 与 combiner.....	13
1.1.3用 mr 进行数据的排序, 然后求出 topN.....	16
1.1.4MapReduce 求两个人的共同好友算法.....	19
1.2 分布式资源调度框架.....	23
1.2.1yarn 的概念.....	23
1.2.2yarn 的架构.....	24
1.2.3yarn 的工作流程.....	24
1.2.4yarn 的调度器 Scheduler.....	26
1.3 分布式文件存储系统.....	28
1.3.1 架构.....	28
1.3.2原理.....	29
1.3.3API.....	32
1.4 项目.....	47
1.4.1点击流日志模型.....	47
1.4.2推荐系统项目.....	56
1.5 相关面试题.....	59
第2章 数据分析及其步骤.....	61
2.1 数据分析定义.....	61
2.2 数据分析的作用.....	61
2.2.1现状分析.....	61
2.2.2原因分析.....	62
2.2.3预测分析.....	62
2.3 数据分析基本步骤 (重点)	62
2.3.1明确分析目的和思路.....	63
2.3.2数据收集.....	67
2.3.3数据处理.....	68
2.3.4数据分析.....	68
2.3.5数据展现 (数据可视化)	69
2.3.6报告撰写.....	69
2.4 行业前景.....	70
2.4.1蓬勃发展的趋势.....	70
2.4.2数据分析师的职业要求.....	71
2.5 大数据时代.....	72
2.5.1大数据的含义.....	72
2.5.2产生背景.....	72
2.5.3影响.....	73
2.5.4特征.....	75
2.5.5思维变革.....	76
2.5.6第三个思维变革:	76
不是所有的事情都必须知道现象背后的原因, 而是要让数据自己“发声”, 即不是因果关系, 而是相关关系。.....	76

2.5.7	科技发展带来的挑战.....	76
2.5.8	分布式系统.....	77
2.6	面试题:.....	78
2.6.1	分布式和集群有啥区别呢?	78
2.6.2	集群 负载均衡 分布式 有什么区别?	79
2.6.3	Web 流量日志数据自定义采集.....	80
2.7	Hive.....	89
2.7.1	Hive 的基本简介.....	89
2.7.2	Hive 数仓开发的基本流程。	90
2.7.3	Hive sql 知识点。	92
2.8	Hive 面试题.....	96
2.8.1	Hive 常见.....	96
2.8.2	海量数据处理方法.....	153
2.9	Hadoop HA.....	166
2.9.1	知识点.....	166
2.9.2	上课用图.....	176
2.9.3	面试题.....	177
2.10	Hadoop 的联邦机制.....	178
2.10.1 为什么会出现联邦?	178
2.10.2 联邦的实现	178
2.10.3 主要优点:	179
2.10.4配置:	180
2.10.5操作	181
2.11	项目	182
2.11.1 点击流日志模型	182
2.11.2 相关面试题	191
2.11.3 推荐系统项目	191
第 3 章	Storm.....	194
3.1	架构.....	195
3.1.1	Nimbus.....	195
3.1.2	Zookeeper.....	195
3.1.3	supervisor.....	195
3.1.4	worker.....	195
3.2	编程模型.....	196
3.2.1	Spout.....	196
3.2.2	Bolt.....	196

3.2.3	并行度.....	196
3.2.4	消息不丢失.....	197
3.3	Storm 物联网 wifi 项目.....	197
3.3.1	wifi 项目背景.....	197
3.3.2	数据来源(采集).....	198
3.3.3	集群规模配置.....	199
3.3.4	数据处理流程.....	199
3.3.5	数据计算流程.....	201
3.4	storm 实时看板案例.....	201
3.4.1	需求分析.....	201
3.4.2	确定数据源.....	202
3.4.3	确定采集方案.....	202
3.4.4	确定存储.....	203
3.4.5	数据计算.....	203
3.4.6	展现.....	203
3.5	storm 日志监控告警系统.....	203
3.5.1	需求分析.....	203
3.5.2	确定数据源.....	204
3.5.3	确定采集方案.....	204
3.5.4	确定存储.....	204
3.5.5	数据计算.....	205
3.6	Storm 的框架.....	205
3.7	面试题.....	206
第 4 章	Kafka.....	206
4.1	kafka 的介绍.....	206
4.2	kafka 的架构.....	209
4.3	kafka 集群的安装与搭建.....	212
4.4	kafka 的原理.....	215
4.5	kafka 的 API 使用.....	218
4.6	kafka 与其他的整合使用.....	222
第 5 章	HBASE 数据库.....	239
5.1	数据库 OLAP、OLTP 的介绍和比较.....	239
5.2	Hbase 基础.....	240
	什么是 OLTP.....	240
5.2.1	hbase 数据库介绍.....	243
5.2.2	hbase 集群结构.....	246
5.2.3	hbase 集群搭建 (简易步骤).....	247
5.2.4	命令行演示.....	247
5.2.5	hbase 代码开发 (基本, 过滤器查询)	248
5.2.6	hbase 内部原理	249
5.2.7	物理存储	251

5.2.8.....	寻址机制	255
5.2.9.....	读写过程	256
5.2.10.....	Region 管理	258
5.2.11.....	Master 工作机制	259
5.2.12.....	HBase 容错性	259
5.3	Hbase 高级应用.....	260
5.3.1	建表高级属性.....	260
5.3.2	HBase 的设计原则.....	262
5.3.3	rowkey 长度原则.....	262
5.3.4	什么是热点.....	263
1)	加盐.....	263
2)	哈希.....	264
3)	反转.....	264
4)	时间戳反转.....	264
5.4	HBase 性能优化完全版.....	265
5.4.1	垃圾回收优化.....	265
5.4.2	启用压缩，详情自行搜索，暂时未曾尝试，后面持续更新。.....	266
5.4.3	优化 Region 拆分合并以及与拆分 Region.....	266
5.4.4	客户端入库调优.....	267
5.4.5	HBase 配置文件.....	268
5.4.6	HDFS 优化部分.....	272
5.5	大数据 Hbase 面试题.....	273
第 6 章	spark.....	279
6.1	什么是 Spark.....	279
6.1.1	Spark 的特点及相对于 MapReduce 的优势.....	279
6.1.2	Spark HA 高可用部署.....	281
6.1.3	Spark 的角色.....	282
6.1.4	Spark 程序提交任务模式.....	283
6.1.5	Spark-Shell.....	284
6.1.6	在 IDEA 中编写 WordCount 程序.....	285
6.2	Spark Streaming.....	285
6.2.1	Spark Streaming 的特性.....	285
6.2.2	Spark Streaming 对比 Storm.....	285

6.3	算子操作.....	285
6.3.1	Transformations.....	285
6.3.2	OutputOperations.....	287
6.3.3	Spark Streaming 编程实战.....	288
6.3.4	SparkStreaming 整合 flume.....	288
6.3.5	SparkStreaming 整合 Kafka.....	288
6.3.6	Checkpoint.....	289
6.4	Spark SQL.....	290
6.4.1	几个知识点.....	290
6.4.2	002 以编程方式执行 Spark SQL 查询.....	291
6.4.3	JDBC 数据源.....	293
6.5	SparkRDD.....	293
6.5.1 了解 SparkRDD	293
1)	什么是 RDD.....	293
2)	RDD 的属性.....	294
3.	为什么会产生 RDD?	295
4)	RDD 在 Spark 中的地位及作用.....	295
6.5.2	创建 RDD.....	296
6.5.3	RDD 编程 API.....	297
6.5.4	RDD 常用的算子操作.....	299
6.5.5	RDD 的依赖关系.....	302
6.5.6	RDD 的缓存.....	303
6.5.7	DAG 的生成.....	304
6.5.8 Spark 任务调度	305
6.6	RDD 容错机制之 checkpoint.....	307
6.6.1	checkpoint 是什么.....	307
6.6.2	checkpoint 原理机制.....	307
6.6.3	Checkpoint 常见面试问题.....	314
6.7	Spark 运行架构.....	318
6.7.1	Spark 运行基本流程.....	318
6.7.2 Spark 运行架构特点	319
6.8	开发调优.....	324
6.8.1 调优概述	324
6.8.2 原则一：避免创建重复的 RDD	324
6.8.3 原则二：尽可能复用同一个 RDD	325
6.8.4 原则三：对多次使用的 RDD 进行持久化	326
6.8.5 原则四：尽量避免使用 shuffle 类算子	

	329
6.8.6原则五：使用 map-side 预聚合的 shuffle 操作.....	329
6.8.7.....原则六：使用高性能的算子	
331	
6.8.8.....原则七：广播大变量	
332	
6.8.9.....原则八：使用 Kryo 优化序列化性能	
333	
6.8.10.....原则九：优化数据结构	
333	
6.9 资源调优.....	334
6.9.1.....调优概述	
334	
6.9.2.....资源参数调优	
336	
6.9.3.....资源参数参考示例	
339	
6.10 数据倾斜调优.....	339
6.10.1.....调优概述	
339	
6.10.2.....数据倾斜发生时的现象	
339	
6.10.3.....数据倾斜发生的原理	
340	
6.10.4.....如何定位导致数据倾斜的代码	
341	
6.10.5.....查看导致数据倾斜的 key 的数据分布情况	
344	
6.10.6.....数据倾斜的解决方案	
344	
6.11 shuffle 调优.....	356
6.11.1.....调优概述	
356	
6.11.2.....ShuffleManager 发展概述	
357	
6.11.3.....HashShuffleManager 运行原理	
357	
6.11.4.....SortShuffleManager 运行原理	
358	
6.11.5.....shuffle 相关参数调优	
360	
6.12 Spark 面试题汇总.....	363
1. spark 中的 RDD 是什么，有哪些特性.....	363
2. 概述一下 spark 中的常用算子区别（map、mapPartitions、foreach、	

foreachPartition)	363
3. 谈谈 spark 中的宽窄依赖.....	364
4. spark 中如何划分 stage.....	364
5. spark-submit 的时候如何引入外部 jar 包.....	364
6. spark 如何防止内存溢出.....	365
7. spark 中 cache 和 persist 的区别.....	365
8. 简要描述 Spark 分布式集群搭建的步骤.....	365
9. spark 中的数据倾斜的现象、原因、后果.....	365
10. 如何解决 spark 中的数据倾斜问题.....	366
11. flume 整合 sparkStreaming 问题.....	367
12. kafka 整合 sparkStreaming 问题.....	369
6.12.1..... ---- 简答题 ---- 网上资料 ---	371
6.12.2..... -----Spark on Yarn 面试篇	376
6.12.3..... -----spark sql 面试篇	380
6.12.4..... ----选择题 ---	380
6.12.5..... 补充资料: (spark 集群 standalone + spark on yarn)	382

第一章 分布式计算框架与资源调度

1.1 分布式计算框架

1.1.1 编程模型

1. inputformat

在 MapReduce 程序的开发过程中, 往往需要用到 FileInputFormat 与 TextInputFormat, 我们会发现 TextInputFormat 这个类继承自 FileInputFormat, FileInputFormat 这个类继承自 InputFormat, InputFormat 这个类会将文件 file 按照逻辑进行划分, 划分成的每一个 split 切片将会被分配给一个 Mapper 任务, 文件先被切分成 split 块, 而后每一个 split 切片对应一个 Mapper 任务

FileInputFormat 的划分机制:

- A. 简单地按照文件的内容长度进行切片
- B. 切片大小, 默认等于 block 大小

C. 切片时不考虑数据集整体，而是逐个针对每一个文件单独切片

默认情况下， $\text{split size} = \text{block size}$ ，在 hadoop 2.x 中为 128M。

注意： $\text{bytesRemaining}/\text{splitSize} > 1.1$ 不满足的话，那么最后所有剩余的会作为一个切片。从而不会形成例如 129M 文件规划成两个切片的局面。

2. MaTask 端的工作机制

input File 通过 split 被逻辑切分为多个 split 文件，通过 Record 按行读取内容给 map（用户自己实现的）进行处理，数据被 map 处理结束之后交给 OutputCollector 收集器，对其结果 key 进行分区（默认使用 hash 分区），然后写入 buffer，每个 map task 都有一个内存缓冲区，存储着 map 的输出结果，当缓冲区快满的时候需要将缓冲区的数据以一个临时文件的方式存放到磁盘，当整个 map task 结束后再对磁盘中这个 map task 产生的所有临时文件做合并，生成最终的正式输出文件，然后等待 reduce task 来拉数据。Map 端的输入的(k,v)分别是该行的起始偏移量,以及每一行的数据内容,map 端的输出(k,v)可以根据需求进行自定义,但是如果输出的是 javabean 对象,需要对 javabean 继承 writable

3. shuffle 的过程

shuffle 的过程是:Map 产生输出开始到 Reduc 取得数据作为输入之

前的过程称作 shuffle.

1).Collect 阶段：将 MapTask 的结果输出到默认大小为 100M 的环形缓冲区，保存的是 key/value, Partition 分区信息等。

2).Spill 阶段：当内存中的数据量达到一定的阈值的时候，就会将数据写入本地磁盘，在将数据写入磁盘之前需要对数据进行一次排序的操作，如果配置了 combiner，还会将有相同分区号和 key 的数据进行排序。

3).Merge 阶段：把所有溢出的临时文件进行一次合并操作，以确保一个

MapTask 最终只产生一个中间数据文件。

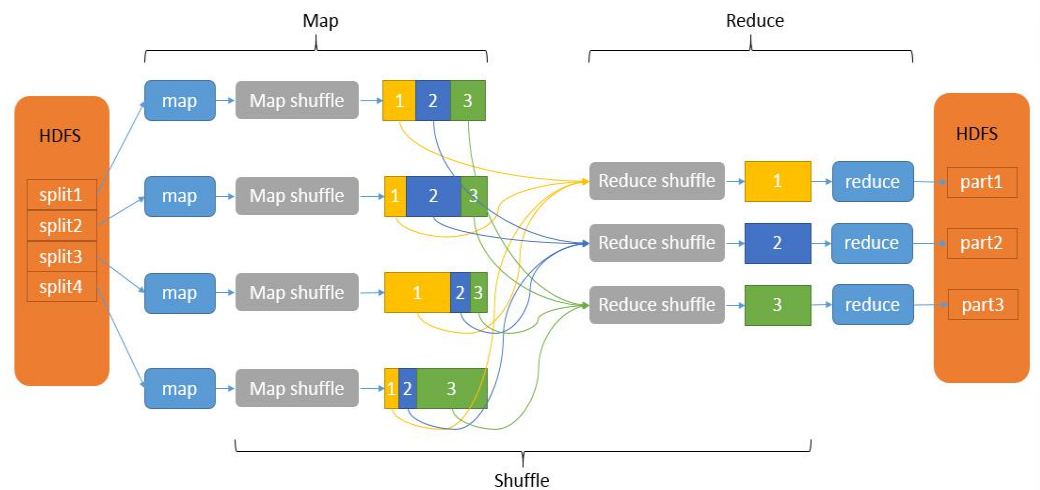
4).Copy 阶段： ReduceTask 启动 Fetcher 线程到已经完成 MapTask 的节点上复制一份属于自己的数据，这些数据默认会保存在内存的缓冲区中，当内存的缓冲区达到一定的阈值的时候，就会将数据写到磁盘之上。

5).Merge 阶段：在 ReduceTask 远程复制数据的同时，会在后台开启两个线

程对内存到本地的数据文件进行合并操作。

6).Sort 阶段：在对数据进行合并的同时，会进行排序操作，由于 MapTask 阶段已经对数据进行了局部的排序，ReduceTask 只需保证 Copy 的数据的最终整体有效性即可。

Shuffle 中的缓冲区大小会影响到 mapreduce 程序的执行效率，原则上说，缓冲区越大，磁盘 io 的次数越少，执行速度就越快缓冲区的大小可以通过参数调整，参数：
io.sort.mb 默认 100M



4. reduceTask

reducer 将已经分好组的数据作为输入，并依次为每个键对应分组执行 reduce 函数。reduce 函数的输入是键以及包含与该键对应的所有值的迭代器。

reduce 端的输入是 map 端的输出,它的输出的(k,v)根据需求进行自定义

reduceset 并行度同样影响整个 job 的执行并发度和执行效率，与 maptask

的并发数由切片数决定不同，Reduceset 数量的决定是可以直接手动设置：

```
job.setNumReduceTasks(4);
```

如果数据分布不均匀，就有可能在 reduce 阶段产生数据倾斜。

默认的 reduceTask 的是 1

*Task 并行度经验之谈：

最好每个 task 的执行时间至少一分钟。如果 job 的每个 map 或者 reduce task 的运行时间都只有 30-40 秒钟，那么就减少该 job 的 map 或者 reduce 数，每一个 task(map|reduce)的 setup 和加入到调度器中进行调度，这个中间的过程可能都要花费几秒钟，所以如果每个 task 都非常快就跑完了，就会在 task 的开始和结束的时候浪费太多的时间。

默认情况下，每一个 task 都是一个新的 JVM 实例，都需要开启和销

毁的开销。在一些情况下，JVM 开启和销毁的时间可能会比实际处理数据的时间

要消耗的长，配置 task 的 M JVM 重用可以改善该问题：

(mapred.job.reuse.jvm.num.tasks, 默认是 1, 表示一个 JVM 上最多可以

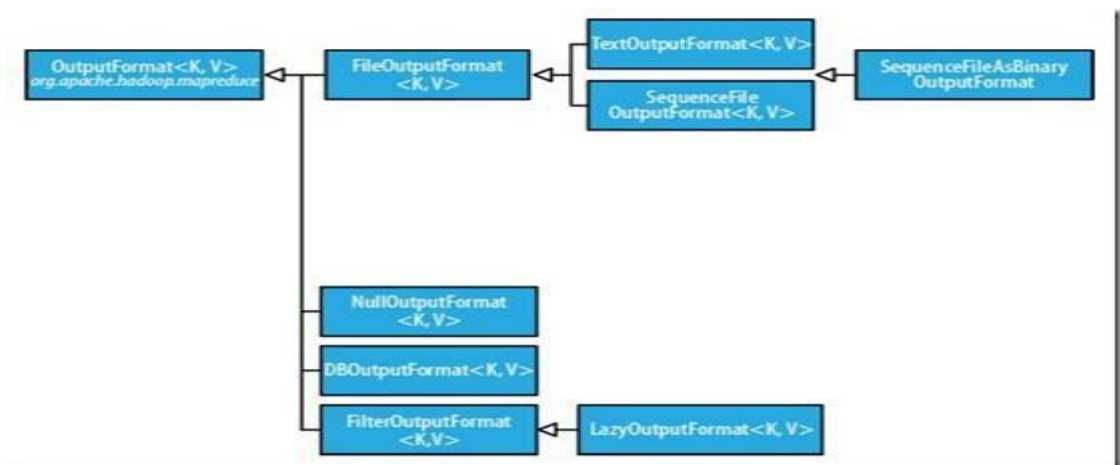
顺序执行的 task 数目 (属于同一个 Job) 是 1。也就是说一个 task 启一个 JVM)

如果 input 的文件非常的大，比如 1TB，可以考虑将 hdfs 上的每个 block

size 设大，比如设成 256MB 或者 512MB

5. outputformat

OutputFormat 主要用于描述输出数据的格式，它能够将用户提供的 key/value 对写入特定格式的文件中。Hadoop 自带了很多 OutputFormat 的实现，它们与 InputFormat 实现相对应，足够满足我们业务的需要。OutputFormat 类的层次结构如下图所示



OutputFormat 是 MapReduce 输出的基类，所有 MapReduce 输出都实现了 OutputFormat 接口,主要有：

TextInputFormat、SequenceFileOutputFormat、MultipleOutputs、DBOutputFormat 等

1.1.2 特殊的组件 partitioner 与 combiner

1. partitioner 定义

partitioner 的作用是将 mapper（如果使用了 combiner 的话就是 combiner）输出的 key/value 拆分为分片（shard），每个 reducer 对应一个分片。默认情况下，partitioner 先计算 key 的散列值（通常为 md5 值）。然后通过 reducer 个数执行取模运算：

key.hashCode%(reducer 个数)。这种方式不仅能够随机地将整个 key 空间平均分发给每个 reducer,同时也能确保不同 mapper 产生的相同 key 能被分发到同一个 reducer。也可以自定义分区去继承 partition<key,value>把不同的结果写入不同的文件中

分区 Partitioner 主要作用在于以下两点

- (1) 根据业务需要, 产生多个输出文件;
- (2) 多个 reduce 任务并发运行, 提高整体 job 的运行效率

适用范围:

需要非常注意的是: 必须提前知道有多少个分区。比如自定义 Partitioner 会返回 5 个不同 int 值, 而 reducer number 设置了小于 5, 那就会报错。所以我们可以通过运行分析任务来确定分区数。

2. map 端的 combine 组件

每一个 map 都可能会产生大量的本地输出, Combiner 的作用就是对 map 端的输出先做一次合并, 以减少在 map 和 reduce 节点之间的数据传输量, 以提高网络 IO 性能, 是 MapReduce 的一种优化手段之一

combiner 是 MR 程序中 Mapper 和 Reducer 之外的一种组件

combiner 组件的父类就是 Reducer

combiner 和 reducer 的区别在于运行的位置:

combiner 是在每一个 maptask 所在的节点运行

reducer 是接收全局所有 Mapper 的输出结果;

combiner 的意义就是对每一个 maptask 的输出进行局部汇总，以减小网络传输量

具体实现步骤：

1)自定义一个 combiner 继承 Reducer，重写 reduce 方法

2)中设置： `job.setCombinerClass(CustomCombiner.class)`

combiner 能够应用的前提是不能影响最终的业务逻辑，而且，combine

输出 kv 应该跟 reducer 的输入 kv 类型要对应起来

Combiner 使用需要注意的是：

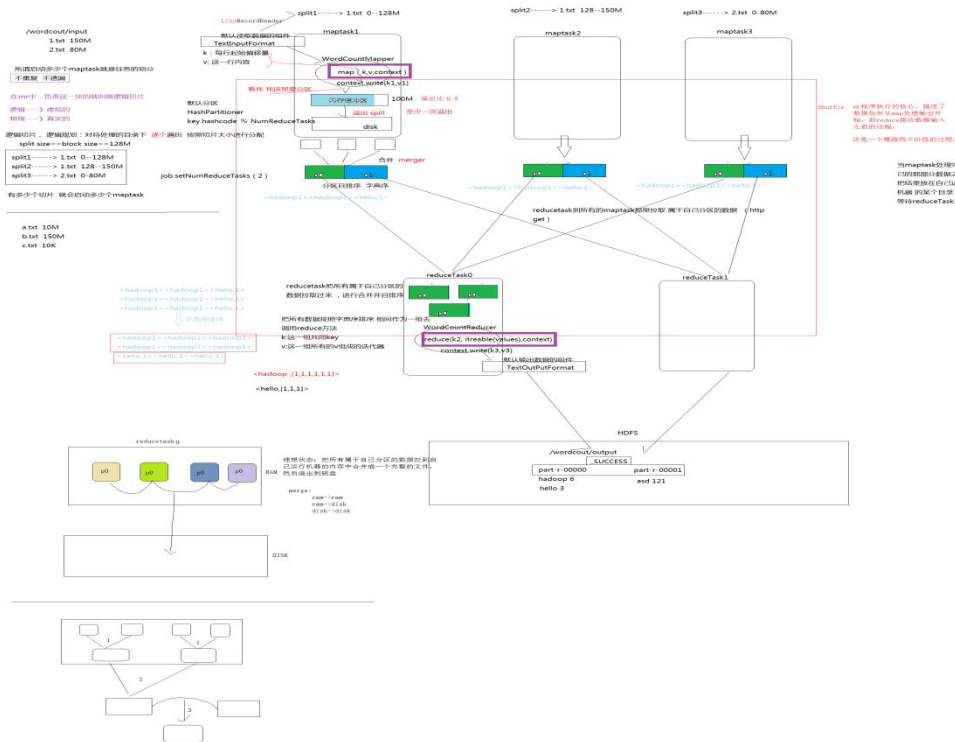
1.有很多人认为这个 combiner 和 map 输出的数据合并是一个过程，其实不然，map 输出的数据合并只会产生在有数据 spill 出的时候，即进行 merge 操作。

2.与 mapper 与 reducer 不同的是, combiner 没有默认的实现，需要显式的设置在 conf 中才有作用。

3.并不是所有的 job 都适用 combiner，只有操作满足结合律的才可设置 combiner。combine 操作类似于：`opt(opt(1, 2, 3), opt(4, 5, 6))`。如果 opt 为求和、求最大值的话，可以使用，但是如果是求中值的话，不适用。

4.一般来说，combiner 和 reducer 它们俩进行同样的操作。

分布式计算的整个流程分析如下图所示：



1.1.3 用 mr 进行数据的排序，然后求出 topN

1.需求

在得出统计每一个用户（手机号）所耗费的总上行流量、下行流量，总流量

结果的基础之上再加一个需求：将统计结果按照总流量倒序排序。

2.分析

基本思路：

实现自定义的 `n bean` 来封装流量信息，并将 `n bean` 作为 `p map` 输出的 `y key` 来传输 MR 程序在处理数据的过程中会对数据排序(map 输出的 `kv` 对传输到 `reduce` 之前，会排序)，排序的依据是 `map` 输出的 `key`。所以，我们如果要想实现自己需要的排序规则，则可以考虑将排序因素放到 `key` 中，让 `key` 实现接口：

WritableComparable, 然后重写 key 的 compareTo 方法。

3.实现自定义的 bean

```
public class FlowBean implements WritableComparable<FlowBean>{
private long upFlow;
private long downFlow;
private long sumFlow;
//这里反序列的时候会用到
public FlowBean() {
}
public FlowBean(long upFlow, long downFlow, long sumFlow) {
this.upFlow = upFlow;
this.downFlow = downFlow;
this.sumFlow = sumFlow;
}
public FlowBean(long upFlow, long downFlow) {
this.upFlow = upFlow;
this.downFlow = downFlow;
this.sumFlow = upFlow+downFlow;
北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090
}
public void set(long upFlow, long downFlow) {
this.upFlow = upFlow;
this.downFlow = downFlow;
this.sumFlow = upFlow+downFlow;
}
@Override
public String toString() {
return upFlow+"\t"+downFlow+"\t"+sumFlow;
}
//这里是序列化方法
@Override
public void write(DataOutput out) throws IOException {
out.writeLong(upFlow);
out.writeLong(downFlow);
}
```

```

out.writeLong(sumFlow);
}
//这里是反序列化方法
@Override
public void readFields(DataInput in) throws IOException {
//注意反序列化的顺序跟序列化的顺序一致
this.upFlow = in.readLong();
this.downFlow = in.readLong();
this.sumFlow = in.readLong();
}
//这里进行 bean 的自定义比较大小
@Override
public int compareTo(FlowBean o) {
//实现按照 sumflow 的大小倒序排序
return this.sumFlow>o.getSumFlow()?-1:1;
}
}
}
北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090
public class FlowSumMapper extends Mapper<LongWritable, Text, Text, FlowBean>{
Text k = new Text();
FlowBean v = new FlowBean();
@Override
protected void map(LongWritable key, Text value,Context context)
throws IOException, InterruptedException {
String line = value.toString();
String[] fields = line.split("\t");
String phoneNum = fields[1];
long upFlow = Long.parseLong(fields[fields.length-3]);
long downFlow = Long.parseLong(fields[fields.length-2]);
k.set(phoneNum);
v.set(upFlow, downFlow);
context.write(k, v);
}
}

public class FlowSumReducer extends Reducer<Text, FlowBean, Text, FlowBean> {
FlowBean v = new FlowBean();
@Override
protected void reduce(Text key, Iterable<FlowBean> values, Context context)
throws IOException, InterruptedException {
long upFlowCount = 0;
long downFlowCount = 0;
for (FlowBean bean : values) {
upFlowCount += bean.getUpFlow();
downFlowCount += bean.getDownFlow();
}
}
}

```

```

}
v.set(upFlowCount, downFlowCount);
context.write(key, v);
}
}

```

1.1.4 MapReduce 求两个人的共同好友算法

需求

以下是 qq 的好友列表数据，冒号前是一个用户，冒号后是该用户的所有好友（数据中的好友关系是单向的）

```

A:B,C,D,F,E,O
B:A,C,E,K
C:F,A,D,I
D:A,E,F,L
E:B,C,D,M,L
F:A,B,C,D,E,O,M
G:A,C,D,E,F
H:A,C,D,E,O
I:A,O
J:B,O
K:A,C,D
L:D,E,F
M:E,F,G
O:A,H,I,J

```

求出哪些人两两之间有共同好友，及他俩的共同好友都有谁？

思路

需求中给出的传入数据格式为：

用户：该用户拥有的好友们

```

user1: frined1, friend2, friend3.....
user2: frined1, friend2, friend3.....
user3: frined1, friend2, friend3.....
user4: frined1, friend2, friend3.....
.....

```

要求传出的格式为：

两个用户：两个用户的共同好友

```

user1-user2: friend1, friend2, friend3.....
user1-user3: friend1, friend2, friend3.....
user1-user4: friend1, friend2, friend3.....
user2-user3: friend1, friend2, friend3.....
user2-user4: friend1, friend2, friend3.....
user3-user4: friend1, friend2, friend3.....

```

.....

我们可以先将传入数据格式转换成：

所有用户的好友们：拥有该好友的用户

friend1: user1, user2, user3, user4.....

friend2: user1, user2, user3, user4.....

friend3: user1, user2, user3, user4.....

.....

再转换成：

两个用户：两个用户的共同好友

user1-user2: friend1, friend2, friend3.....

user1-user3: friend1, friend2, friend3.....

user1-user4: friend1, friend2, friend3.....

user2-user3: friend1, friend2, friend3.....

user2-user4: friend1, friend2, friend3.....

user3-user4: friend1, friend2, friend3.....

.....

代码实现

第一个 Mapper：

```
package com.xianshun.mapper;
```

```
import java.io.IOException;
```

```
import org.apache.hadoop.io.LongWritable;
```

```
import org.apache.hadoop.io.Text;
```

```
import org.apache.hadoop.mapreduce.Mapper;
```

```
public class CommonFriendsStepOneMapper extends Mapper<LongWritable, Text, Text, Text>{
```

```
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, Text>.Context context)
```

```
        throws IOException, InterruptedException {
```

```
        // 传进来的 value: A:B,C,D,F,E,O
```

```
        // 将 value 转换成字符串,line 表示行数据, 为"A:B,C,D,F,E,O"
```

```
        String line = value.toString();
```

```
        // 分割字符串,得到用户和好友们, 用 userAndFriends 表示, 为 {"A","B,C,D,F,E,O"}
```

```
        String[] userAndFriends = line.split(":");
```

```
        // userAndFriends 0 位置为用户, user 为"A"
```

```
        String user = userAndFriends[0];
```


友

```
// userAndFriends 1 位置为好友们，这里以","分割分割字符串，得到每一个好
友

// friend 为{"B","C","D","F","E","O"}
String[] friends = userAndFriends[1].split(",");

// 循环遍历，以<B,A>,<C,A>,<D,A>.....的形式传给 reducer
for (String friend : friends) {

    context.write(new Text(friend), new Text(user));

}

}

}

package com.xianshun.reducer;
import java.io.IOException;
import org.apache.hadoop.io.Text;import org.apache.hadoop.mapreduce.Reducer;

public class CommonFriendsStepOneReducer extends Reducer<Text, Text, Text, Text>{

    protected void reduce(Text friend, Iterable<Text> users, Context context)
        throws IOException, InterruptedException {

        // 传进来的数据    < 好友 , 用户 >
        <B,A>,<C,A>,<D,A>,<A,B>,<C,B>,<E,B>,<K,B>.....
        // 新建 stringBuffer，用于存放 拥有该好友的用户们
        StringBuffer stringBuffer = new StringBuffer();

        // 遍历所有的用户，并将用户放在 stringBuffer 中，以","分隔
        for (Text user : users) {
            stringBuffer.append(user).append(",");
        }

        // 以好友为 key,用户们为 value 传给下一个 mapper
        context.write(friend, new Text(stringBuffer.toString()));

    }

}

}

第一个 MR 结果：
A    I,K,C,B,G,F,H,O,D,
B    A,F,J,E,C    A,E,B,H,F,G,K,D    G,C,K,A,L,F,E,H,E    G,M,L,H,A,F,B,D,
F    L,M,D,C,G,A,
```

```

G    M,
H    O,I    O,C,
J    O,K    B,
L    D,E,
M    E,F,O    A,H,I,J,F,

```

第二个 Mapper:

```

package com.xianshun.mapper;
import java.io.IOException;import java.util.Arrays;
import org.apache.hadoop.io.LongWritable;import org.apache.hadoop.io.Text;import
org.apache.hadoop.mapreduce.Mapper;

```

```

public class CommonFriendsStepTwoMapper extends Mapper<LongWritable, Text, Text,
Text>{

```

```

    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

```

```

        // 传进来的数据      A      I,K,C,B,G,F,H,O,D,
        // 将传进来的数据转换成字符串并以"\t"分割, 得到 fiendsAndUsers
        String[] friendAndUsers = value.toString().split("\t");

```

```

        // fiendsAndUsers 0 位置为好友 "A"
        String friend = friendAndUsers[0];

```

```

        // fiendsAndUsers 1 位置为拥有上面好友用户们
        // 以 ","进行分割字符串, 得到每一个用户,{"I","K",....}
        String[] users = friendAndUsers[1].split(",");
        // 将 user 进行排序, 避免重复
        Arrays.sort(users); // 以用户-用户为 key, 好友们做 value 传给 reducer
        for(int i=0; i<users.length-2; i++) {
            for(int j=i+1; j<users.length-1; j++) {
                context.write(new Text(users[i] + "-" + users[j]), new Text(friend));
            }
        }

```

```

    }

```

```

}

```

第二个 Reducer:

```

package com.xianshun.reducer;
import java.io.IOException;
import org.apache.hadoop.io.Text;import org.apache.hadoop.mapreduce.Reducer;

```

```

public class CommonFriendsStepTwoReducer extends Reducer<Text, Text, Text, Text>{

    protected void reduce(Text user_user, Iterable<Text> friends, Context context)
        throws IOException, InterruptedException {

        // 传进来的数据 <用户 1-用户 2, 好友们>
        // 新建 stringBuffer, 用于用户的共同好友们
        StringBuffer stringBuffer = new StringBuffer();

        // 遍历所有的好友, 并将这些好友放在 stringBuffer 中, 以" "分隔
        for (Text friend : friends) {
            stringBuffer.append(friend).append(" ");
        }

        // 以好友为 key,用户们为 value 传给下一个 mapper
        context.write(user_user, new Text(stringBuffer.toString()));

    }

}

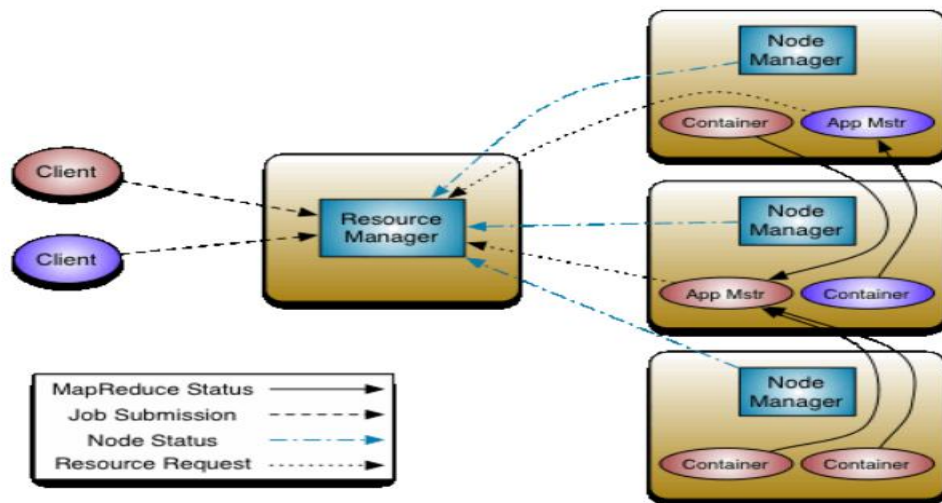
```

1.2 分布式资源调度框架

1.2.1 yarn 的概念

Apache Hadoop YARN (Yet Another Resource Negotiator, 另一种资源协调者) 是一种新的 Hadoop 资源管理器, 它是一个通用资源管理系统和调度平台, 可为上层应用提供统一的资源管理和调度, 它的引入为集群在利用率、资源统一管理和数据共享等方面带来了巨大好处。可以把 yarn 理解为相当于一个分布式的操作系统平台, 而 mapreduce 等运算程序则相当于运行于操作系统之上的应用程序, Yarn 为这些程序提供运算所需的资源 (内存、cpu)。

1.2.2 yarn 的架构



YARN 是一个资源管理、任务调度的框架，主要包含三大模块：
ResourceManager (RM)、
NodeManager (NM)、ApplicationMaster (AM) .

- 1).ResourceManager 负责所有资源的监控、分配和管理；
- 2).ApplicationMaster 负责每一个具体应用程序的调度和协调；
- 3).NodeManager 负责每一个节点的维护。对于所有的 applications, RM 拥有绝对的控制权和对资源的分配权。而每个 AM 则会和 RM 协商资源，同时和 NodeManager 通信来执行和监控 task。

1.2.3 yarn 的工作流程

- 1) client 向 RM 提交应用程序，其中包括启动该应用的 ApplicationMaster 的必须信息，例如 ApplicationMaster 程序、启动 ApplicationMaster 的命令、用户程序等。
- 2) ResourceManager 启动一个 container 用于运行

ApplicationMaster。启动中的 ApplicationMaster 向 ResourceManager 注册自己，启动成功后与 RM 保持心跳。

3) ApplicationMaster 向 ResourceManager 发送请求，申请相应数目的 container。

4) ResourceManager 返回 ApplicationMaster 的申请的 containers 信息。申请成功的

container，由 ApplicationMaster 进行初始化。container 的启动信息初始化后，AM

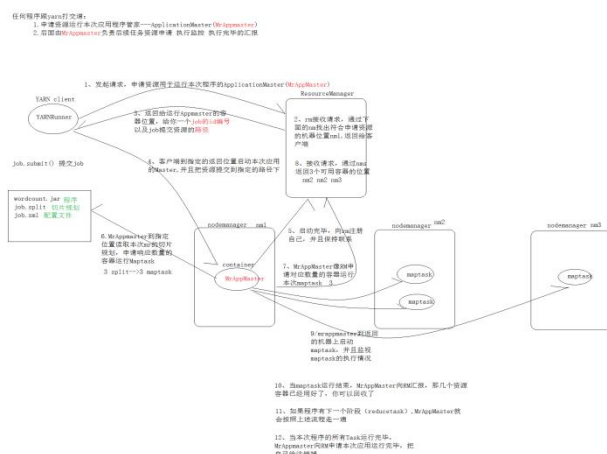
与对应的 NodeManager 通信，要求 NM 启动 container。AM 与 NM 保持心跳，从而对 NM 上运行的任务进行监控和管理。

5) container 运行期间，ApplicationMaster 对 container 进行监控。container 通过 RPC

协议向对应的 AM 汇报自己的进度和状态等信息。

6) 应用运行期间，client 直接与 AM 通信获取应用的状态、进度更新等信息。

7) 应用运行结束后，ApplicationMaster 向 ResourceManager 注销自己，并允许属于它的 container 被收回。

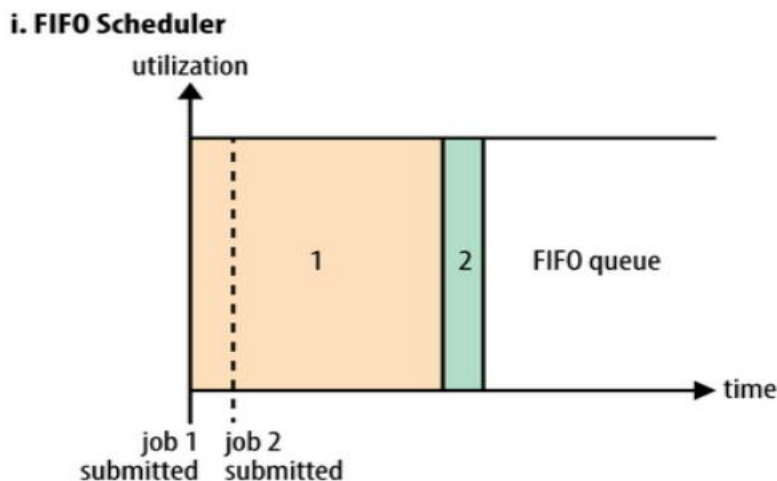


1.2.4 yarn 的调度器 Scheduler

Yarn 中,负责给应用分配资源的就是 Scheduler,三种调度器可以选择:FIFO Scheduler ,Capacity Scheduler,FairScheduler。

1. FIFO Scheduler

FIFO Scheduler 把应用按提交的顺序排成一个队列,这是一个 先进先出队列,在进行资源分配的时候,先给队列中最头上的应用进行分配资源,待最头上的应用需求满足后再给下一个分配,以此类推。



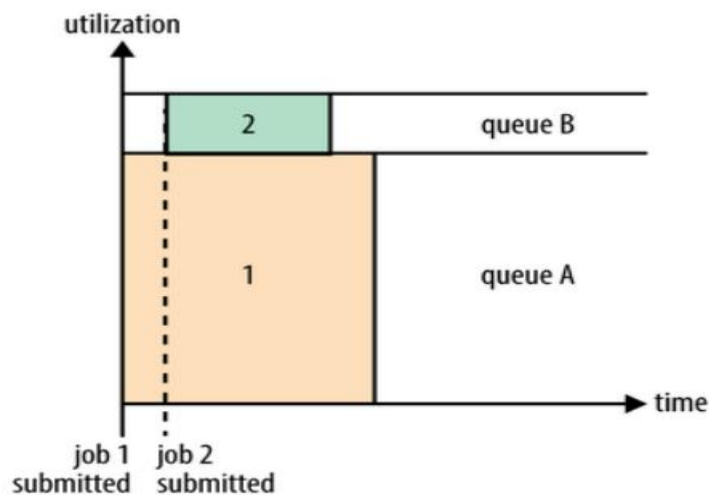
2. Capacity Scheduler

Capacity 调度器允许多个组织共享整个集群,每个组织可以获得集群的一部分计算能力。

通过为每个组织分配专门的队列,然后再为每个队列分配一定的集群资源,这样整个集群就可以通过设置多个队列的方式给多个组织提供服务了。除此之外,队列内部又可以垂直划分,这样一个组织内

部的多个成员就可以共享这个队列资源了，在一个队列内部，资源的调度是采用的是先进先出(FIFO)策略。

ii. Capacity Scheduler



3.Fair Scheduler

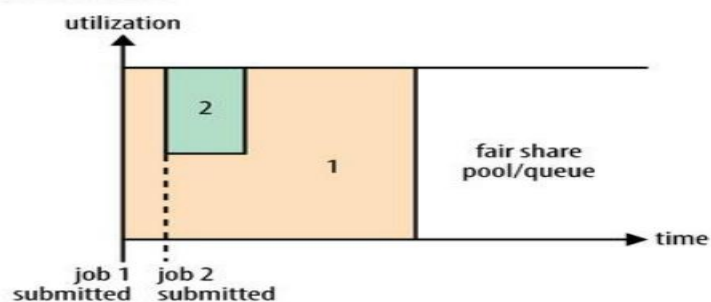
在 Fair 调度器中，我们不需要预先占用一定的系统资源，Fair 调度器会为所有运行的

job 动态的调整系统资源。如下图所示，当第一个大 job 提交时，只有这一个 job 在运行，

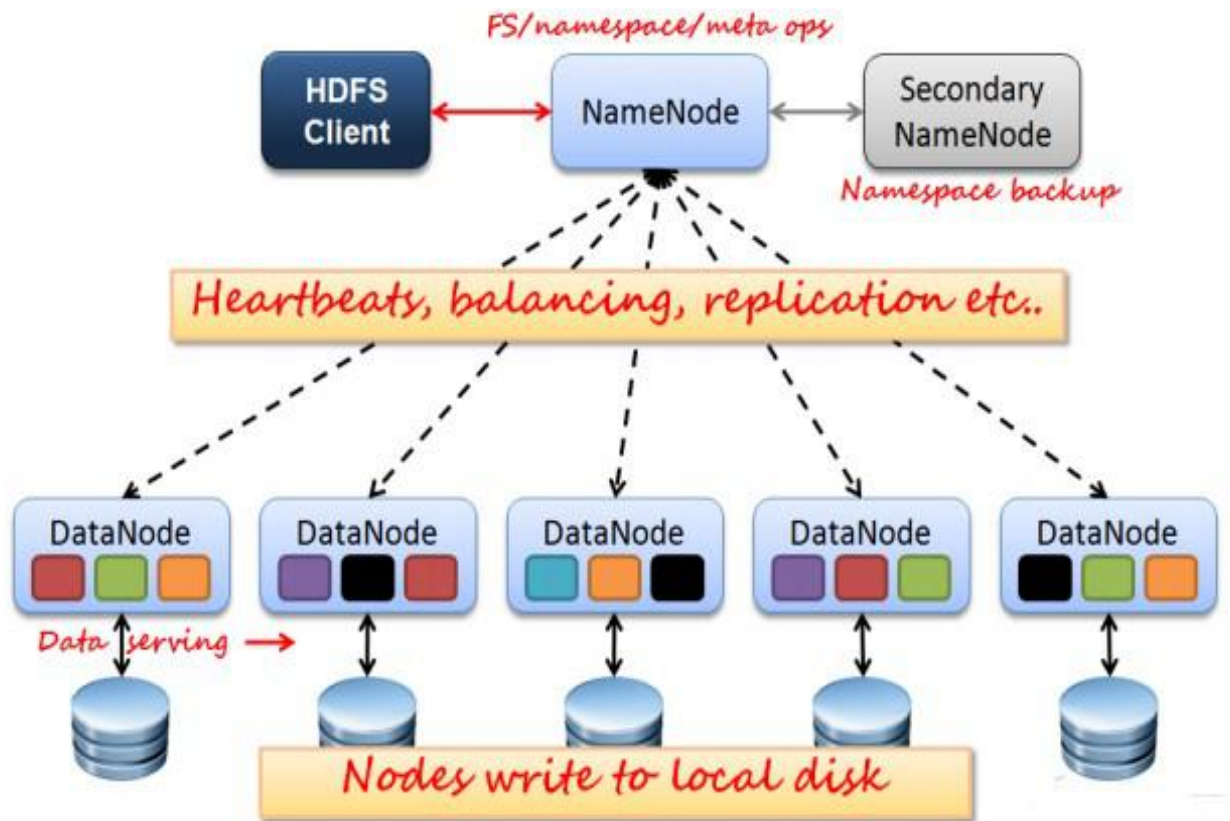
此时它获得了所有集群资源；当第二个小任务提交后，Fair 调度器会分配一半资源给这个

小任务，让这两个任务公平的共享集群资源。

iii. Fair Scheduler



1.3 分布式文件存储系统



1.3.1 架构

如上图所示，HDFS 也是按照 Master 和 Slave 的结构。分 NameNode、SecondaryNameNode、DataNode 这几个角色。

NameNode: 是 Master 节点，是大领导。管理数据块映射；处理客户端的读写请求；配置副本策略；管理 HDFS 的名称空间；

SecondaryNameNode: 是一个小弟，分担大哥 namenode 的一部分工作量；是 NameNode 的冷备份；合并 `fsimage` 和 `fsedit` 然后再发给 namenode。

DataNode: *Slave* 节点，奴隶，干活的。负责存储 client 发来的数据块 **block**；执行数据块的读写操作。

热备份: **b** 是 **a** 的热备份，如果 **a** 坏掉。那么 **b** 马上运行代替 **a** 的工作。

冷备份: **b** 是 **a** 的冷备份，如果 **a** 坏掉。那么 **b** 不能马上代替 **a** 工作。但是 **b** 上存储 **a** 的一些信息，减少 **a** 坏掉之后的损失。

fsimage: 元数据镜像文件（文件系统的目录树。）

edits: 元数据的操作日志（针对文件系统做的修改操作记录）

namenode 内存中存储的是 **=fsimage+edits**。

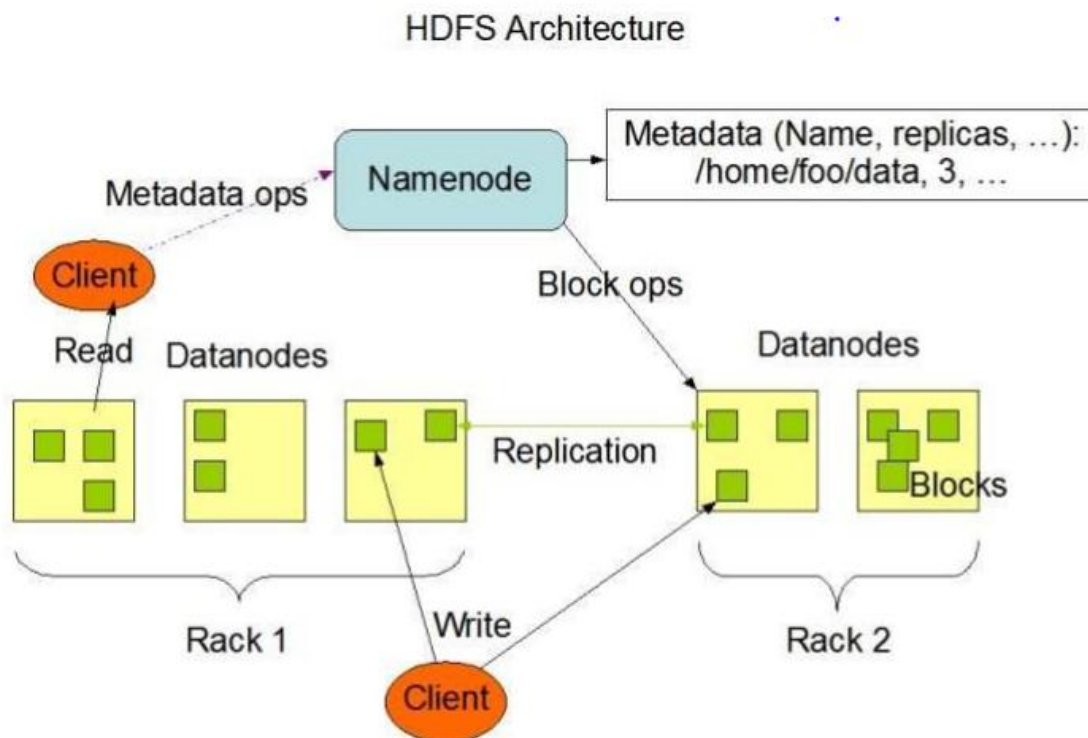
SecondaryNameNode 负责定时默认 1 小时，从 **namenode** 上，获取 **fsimage** 和 **edits** 来进行合并，然后再发送给 **namenode**。减少 **namenode** 的工作量。

1.3.2 原理

1.工作机制

NameNode 负责管理整个文件系统元数据；**DataNode** 负责管理具体文件数据块存储；**Secondary NameNode** 协助 **NameNode** 进行元数据的备份。

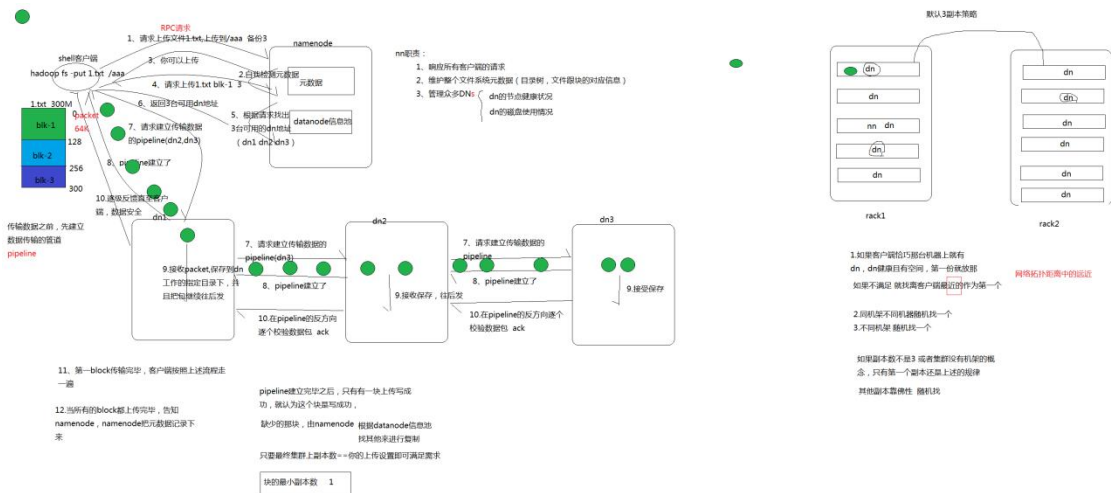
HDFS 的内部工作机制对客户端保持透明，客户端请求访问 **HDFS** 都是通过向 **NameNode** 申请来进行。



2.读写流程

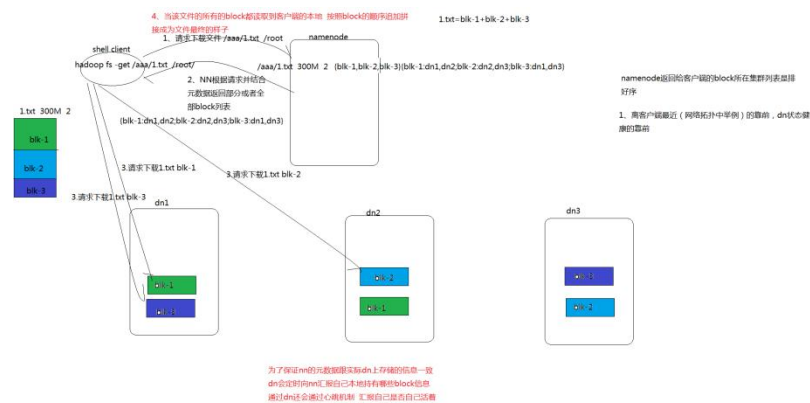
* HDFS 写数据流程

- 1、 client 发起文件上传请求，通过 RPC 与 NameNode 建立通讯，NameNode 检查目标文件是否已存在，父目录是否存在，返回是否可以上传；
 - 2、 client 请求第一个 block 该传输到哪些 DataNode 服务器上；
 - 3、 NameNode 根据配置文件中指定的备份数量及机架感知原理进行文件分配，返回可用的 DataNode 的地址如：A, B, C；
- 注： Hadoop 在设计时考虑到数据的安全与高效，数据文件默认在 HDFS 上存放三份， 存储策略 为本地一份， 同机架内其它某一节点上一份， 不同机架的某一节点上一份。
- 4、 client 请求 3 台 DataNode 中的一台 A 上传数据（本质上是一个 RPC 调用，建立 pipeline）， A 收到请求会继续调用 B， 然后 B 调用 C， 将整个 pipeline 建立完成， 后逐级返回 client；
 - 5、 client 开始往 A 上传第一个 block（先从磁盘读取数据放到一个本地内存缓存）， 以 packet 为单位（默认 64K）， A 收到一个 packet 就会传给 B， B 传给 C； A 每传一个 packet 会放入一个应答队列等待应答。
 - 6、 数据被分割成一个个 packet 数据包在 pipeline 上依次传输， 在 pipeline 反方向上， 逐个发送 ack（命令正确应答）， 最终由 pipeline 中第一个 DataNode 节点 A 将 pipeline ack 发送给 client；
 - 7、 当一个 block 传输完成之后， client 再次请求 NameNode 上传第二个 block 到服务器。



*HDFS 读数据流程

- 1、 Client 向 NameNode 发起 RPC 请求, 来确定请求文件 block 所在的位置;
- 2、 NameNode 会视情况返回文件的部分或者全部 block 列表, 对于每个 block, NameNode 都会返回含有该 block 副本的 DataNode 地址;
- 3、 这些返回的 DN 地址, 会按照集群拓扑结构得出 DataNode 与客户端的距离, 然后进行排序, 排序两个规则: 网络拓扑结构中距离 Client 近的排靠前; 心跳机制中超时汇报的 DN 状态为 STALE, 这样的排靠后;
- 4、 Client 选取排序靠前的 DataNode 来读取 block, 如果客户端本身就是 DataNode,那么将从本地直接获取数据;
- 5、 底层上本质是建立 Socket Stream (FSDataInputStream), 重复的调用父类 DataInputStream 的 read 方法, 直到这个块上的数据读取完毕;
- 6、 当读完列表的 block 后, 若文件读取还没有结束, 客户端会继续向 NameNode 获取下一批的 block 列表;
- 7、 读取完一个 block 都会进行 checksum 验证, 如果读取 DataNode 时出现错误, 客户端会通知 NameNode, 然后再从下一个拥有该 block 副本的 DataNode 继续读。
- 8、 read 方法是并行的读取 block 信息, 不是一块一块的读取; NameNode 只是返回 Client 请求包含块的 DataNode 地址, 并不是返回请求块的数据;
- 9、 最终读取来所有的 block 会合并成一个完整的最终文件。



1.3.3 API

1.shell 定时采集数据至 HDFS

* 技术分析

HDFS SHELL:

hadoop fs -put // 满足上传 文件，不能满足定时、周期性传入。

Linux crontab:

crontab -e

0 0 * * * /shell/ uploadFile2Hdfs.sh //每天凌晨 12: 00 执行一次

* 实现流程

一般日志文件生成的逻辑由业务系统决定，比如每小时滚动一次，或者一定大小滚动一次，避免单个日志文件过大不方便操作。

比如滚动后的文件命名为 access.log.x,其中 x 为数字。正在进行写的日志文件叫做 access.log。这样的话，如果日志文件后缀是 1\2\3 等数字，则该文件满足需求可以上传，就把该文件移动到准备上传的工作区间目录。工作区间有文件之后，可以使用 hadoop put 命令将文件上传。

* 代码实现

```

#读取日志文件的目录，判断是否有需要上传的文件
echo "log_src_dir:"$log_src_dir
ls $log_src_dir | while read fileName
do
    if [[ "$fileName" == access.log.* ]]; then
        # if [ "access.log" = "$fileName" ];then
        date=`date +%Y_%m_%d_%H_%M_%S`
        #将文件移动到待上传目录并重命名
        #打印信息
        echo "moving $log_src_dir$fileName to $log_toupload_dir"xxxxx_click_log_$fileName
        mv $log_src_dir$fileName $log_toupload_dir"xxxxx_click_log_$fileName"$date
        #将待上传的文件path写入一个列表文件willDoing
        echo $log_toupload_dir"xxxxx_click_log_$fileName"$date >> $log_toupload_dir"willDo
    fi
done

```

2.JavaAPI 操作

HDFS 的 JAVA API 操作

HDFS 在生产应用中主要是客户端的开发，其核心步骤是从 HDFS 提供的 api 中构造一个 HDFS 的访问客户端对象，然后通过该客户端对象操作（增删改查）HDFS 上的文件。

1) 搭建开发环境

创建 Maven 工程，引入 pom 依赖

```

<dependencies>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>2.7.4</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-hdfs</artifactId>
        <version>2.7.4</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>2.7.4</version>
    </dependency>
</dependencies>

```

配置 windows 平台 Hadoop 环境

在 windows 上做 HDFS 客户端应用开发，需要设置 Hadoop 环境,而且要求是 windows 平台编译的 Hadoop,不然会报以下的错误:

Failed to locate the winutils binary in the hadoop binary path java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090

为此我们需要进行如下的操作：

A、在 windows 平台下编译 Hadoop 源码（可以参考资料编译，但不推荐）

B、使用已经编译好的 Windows 版本 Hadoop：

hadoop-2.7.4-with-windows.tar.gz

C、解压一份到 windows 的任意一个目录下

D、在 windows 系统中配置 HADOOP_HOME 指向你解压的安装包目录

E、在 windows 系统的 path 变量中加入 HADOOP_HOME 的 bin 目录

2) 构造客户端对象

在 java 中操作 HDFS，主要涉及以下 Class：

Configuration：该类的对象封装了客户端或者服务器的配置；

FileSystem：该类的对象是一个文件系统对象，可以用该对象的一些方法来对文件进行操作，通过 FileSystem 的静态方法 get 获得该对象。

FileSystem fs = FileSystem.get(conf)

get 方法从 conf 中的一个参数 fs.defaultFS 的配置值判断具体是什么类型的文件系统。如果我们的代码中没有指定 fs.defaultFS，并且工程 classpath 下也没有给定相应的配置，conf 中的默认值就来自于 hadoop 的 jar 包中的 core-default.xml，默认值为：file:///，则获取的将不是一个 DistributedFileSystem 的实例，而是一个本地文件系统的客户端对象。

3) 示例代码

```
Configuration conf = new Configuration();
```

```
//这里指定使用的是 hdfs 文件系统
```

```
conf.set("fs.defaultFS", "hdfs://node-21:9000");
```

```
//通过如下的方式进行客户端身份的设置
```

```
System.setProperty("HADOOP_USER_NAME", "root");
```

```
//通过 FileSystem 的静态方法获取文件系统客户端对象
```

```
FileSystem fs = FileSystem.get(conf);
```

```
//也可以通过如下的方式去指定文件系统的类型 并且同时设置用户身份
```

```
//FileSystem fs = FileSystem.get(new URI("hdfs://node-21:9000"), conf, "root");
```

```
//创建一个目录
```

```
fs.create(new Path("/hdfsbyjava-ha"), false);
```

```
//上传一个文件
```

```
fs.copyFromLocalFile(new Path("e:/hello.sh"), new Path("/hdfsbyjava-ha"));
```

```
//关闭我们的文件系统
```

```
fs.close();
```

***Hbase 使用 javaapi 存储数据到 HDFS**

```
public class HbaseDemo {

    private Configuration conf = null;

    @Before
    public void init(){
        conf = HBaseConfiguration.create();
        conf.set("hbase.zookeeper.quorum", "itcast01:2181,itcast02:2181,itcast03:2181");
    }

    @Test
    public void testDrop() throws Exception{
        HBaseAdmin admin = new HBaseAdmin(conf);
        admin.disableTable("account");
        admin.deleteTable("account");
        admin.close();
    }

    @Test
    public void testPut() throws Exception{
        HTable table = new HTable(conf, "person_info");
        Put p = new Put(Bytes.toBytes("person_rk_bj_zhang_000002"));
        p.add("base_info".getBytes(), "name".getBytes(), "zhangwuji".getBytes());
        table.put(p);
        table.close();
    }

    @Test
    public void testDel() throws Exception{
        HTable table = new HTable(conf, "user");
        Delete del = new Delete(Bytes.toBytes("rk0001"));
        del.deleteColumn(Bytes.toBytes("data"), Bytes.toBytes("pic"));
        table.delete(del);
        table.close();
    }

    @Test
    public void testGet() throws Exception{
        HTable table = new HTable(conf, "person_info");
        Get get = new Get(Bytes.toBytes("person_rk_bj_zhang_000001"));
```

```

get.setMaxVersions(5);
Result result = table.get(get);

List<Cell> cells = result.listCells();

for(Cell c:cells){
}

//result.getValue(family, qualifier);  可以从 result 中直接取出一个特定的
value

//遍历出 result 中所有的键值对
List<KeyValue> kvs = result.list();
//kv      --->    f1.title:superise....                f1:author:zhangsan
f1:content:asdfasldgkjsldg
for(KeyValue kv : kvs){
    String family = new String(kv.getFamily());
    System.out.println(family);
    String qualifier = new String(kv.getQualifier());
    System.out.println(qualifier);
    System.out.println(new String(kv.getValue()));
}
table.close();
}

```

*Storm 使用 javaapi 存储数据 HDFS

第一步：导入整合的 jar 包

```

com/artifact/org.apache.storm/storm-hdfs -->
<dependency>
  <groupId>org.apache.storm</groupId>
  <artifactId>storm-hdfs</artifactId>
  <version>1.1.1</version>
  <exclusions>
    <exclusion>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-client</artifactId>
    </exclusion>
    <exclusion>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-auth</artifactId>
    </exclusion>
  </exclusions>
</dependency>

```



```

        </exclusion>
        <exclusion>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-hdfs</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-jdbc</artifactId>
    <version>1.1.1</version>
</dependency>
<!--
https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.38</version>
</dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <encoding>utf-8</encoding>
            </configuration>
        </plugin>
    </plugins>
</build>

```

```

        </configuration>
    </plugin>

    <!-- storm 与 hdfs 的整合，请使用这种打包方式 -->

    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>1.4</version>
        <configuration>
            <createDependencyReducedPom>true</createDependencyReducedPom>
        </configuration>
        <executions>
            <execution>
                <phase>package</phase>
                <goals>
                    <goal>shade</goal>
                </goals>
                <configuration>
                    <transformers>
                        <transformer
                            implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"/>
                        <transformer
                            implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                            <mainClass>cn.itcast.hadoop.stormhadoop.MainTopology</mainClass>
                        </transformer>
                    </transformers>
                </configuration>
            </execution>
        </executions>
    </plugin>

```

```

</plugin>

<!--
    注意 storm 与 hdfs 的整合已经不能使用这种打包方式，使用
    这种打包方式会出错    Error preparing HdfsBolt: No FileSystem for
    scheme: hdfs at org.apache.storm.hdfs.bolt.AbstractHdfs
    storm 与 mysql 的整合请使用这种打包方式
-->
<!--    <plugin>
                                <artifactId>    maven-assembly-plugin
</artifactId>

                                <configuration>
                                    <descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>
                                </descriptorRefs>
                                    <archive>
                                        <manifest>

<mainClass>cn.itcast.storm.demo2.JdbcTopo</mainClass>
                                        </manifest>
                                    </archive>
                                </configuration>
                                <executions>
                                    <execution>
                                        <id>make-assembly</id>
                                        <phase>package</phase>
                                        <goals>
                                            <goal>single</goal>
                                        </goals>
                                    </execution>
                                </executions>
                                </plugin>    -->
</plugins>
</build>

```

第二步：开发我们的 RandomOrder

```

public class RandomOrderSpout extends BaseRichSpout{

    private SpoutOutputCollector collector;
    private PaymentInfo paymentInfo;

    /**
     * 初始化的方法只会被调用一次
     * @param conf
     * @param context
     * @param collector
     */
    @Override
    public void open(Map conf, TopologyContext context,
SpoutOutputCollector collector) {
        this.collector = collector;
        paymentInfo = new PaymentInfo();
    }

    /**
     * 这个方法会反复的不断被调用
     */
    @Override
    public void nextTuple() {
        //发送一个 json 格式的字符串出去
        //collector.emit(new Values(paymentInfo.random()));
        //注意我们这里发送数据多带了一个参数是我们的 msgId,如果
        发送数据的时候，带上了 msgid，就表明我们开启了 storm 的 ack 机制
        String random = paymentInfo.random();
        collector.emit(new Values(random), random);

    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("randomOrder"));
    }
}

```

```

    /**
     * 消息发送失败的时候需要调用这个 fail 方法
     * @param msgId
     */
    @Override
    public void fail(Object msgId) {
        collector.emit(new Values(msgId),msgId);
    }

    /**
     * 消息发送成功时候调用的方法
     * @param msgId
     */
    @Override
    public void ack(Object msgId) {
        // super.ack(msgId);
        System.out.println("我是消息发送成功了");
    }
}

```

第三步：实现我们的 countMoneyBolt

```

public class CountMoneyBolt extends BaseRichBolt{
    //OOM out of memory exception
    private OutputCollector collector;
    private JSONObject jsonObject;

    private static ConcurrentHashMap<String,Long>
    concurrentHashMap = new ConcurrentHashMap<String,Long>();

    /**
     * 初始化的方法，只会被调用一次
     * @param stormConf
     * @param context
     * @param collector
     */
}

```

```

@Override
public void prepare(Map stormConf, TopologyContext context,
OutputCollector collector) {
    this.collector = collector;
    jsonObject = new JSONObject();
}

/**
 * 反复不断的被调用
 * @param input
 */
@Override
public void execute(Tuple input) {
    //获取上游发送的数据过来
    String randomOrder = input.getStringByField("randomOrder");
    PaymentInfo paymentInfo =
jsonObject.parseObject(randomOrder, PaymentInfo.class);
    long payPrice = paymentInfo.getPayPrice();
    if(concurrentHashMap.containsKey("totalMoney")){

concurrentHashMap.put("totalMoney",concurrentHashMap.get("totalMoney")+payPrice);
    }else{
        concurrentHashMap.put("totalMoney",payPrice);
    }
    System.out.println("    当前的总金额是
"+concurrentHashMap.toString());
    collector.emit(new Values(randomOrder));
    //如果继承的是 BaseRichBolt，一定要记得最后调用一下 ack 方法，绝对不会错，如果不调，有可能会错误
    collector.ack(input);
}

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("moneyOrder"));
}

```

```
}  
}
```

第四步：主代码实现

```
public class OrderMain {  
  
    public static void main(String[] args) throws  
InvalidTopologyException, AuthorizationException,  
AlreadyAliveException {  
        // use "|" instead of "," for field delimiter  
        RecordFormat format = new DelimitedRecordFormat()  
            .withFieldDelimiter("|");  
  
        // sync the filesystem after every 1k tuples 数据同步到 hdfs 的策略，每  
        // 执行一千条数据的时候，同步一次到 hdfs 上面去  
        SyncPolicy syncPolicy = new CountSyncPolicy(1000);  
  
        // rotate files when they reach 5MB 每当文件大小达到 5kb 的时候，同  
        // 步到 HDFS 上面去  
        FileRotationPolicy rotationPolicy = new  
        FileSizeRotationPolicy(5.0f, FileSizeRotationPolicy.Units.KB);  
  
        //指定我们的数据上传到 hdfs 的哪个路劲下面去  
        FileNameFormat fileNameFormat = new  
        DefaultFileNameFormat()  
            .withPath("/test/");  
        //FileSystem  
        HdfsBolt bolt = new HdfsBolt()  
            .withFsUrl("hdfs://node01:9000")  
            .withFileNameFormat(fileNameFormat)  
            .withRecordFormat(format)  
            .withRotationPolicy(rotationPolicy)  
            .withSyncPolicy(syncPolicy);  
    }  
}
```

```

        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("randomOrderSpout",new
RandomOrderSpout());
        builder.setBolt("countMoneyBolt",new
CountMoneyBolt()).localOrShuffleGrouping("randomOrderSpout");

builder.setBolt("hdfsBolt",bolt).localOrShuffleGrouping("countMoneyBolt
");

        Config config = new Config();
        //设置我们 ack 的线程的数量，默认是一个，如果数据量比较大，
        需要将 ack 的线程数调大，用于调优
        config.setNumAckers(3);
        //设置我们的内存池里面有多少个状态没有清掉的时候就不要
        再继续发送数据了
        config.setMaxSpoutPending(5000);

        if(args !=null    && args.length > 0){
            config.setDebug(false);

StormSubmitter.submitTopology(args[0],config,builder.createTopology());
        }else{
            config.setDebug(true);
            LocalCluster cluster = new LocalCluster();

            cluster.submitTopology("stormtoHdfs",config,builder.createTopology());
        }
    }
}

```

*Flume 使用 javaapi 存储数据到 HDFS

1. 配置 flume 代理文件

配置一个 flume agent 代理,在此名称为 shaman。配置文件（netcat-memory-hdfs.conf）如下：


```
# Identify the components on agent shaman:
```

```
shaman.sources = netcat_s1
```

```
shaman.sinks = hdfs_w1
```

```
shaman.channels = in-mem_c1# Configure the source:
```

```
shaman.sources.netcat_s1.type = netcat
```

```
shaman.sources.netcat_s1.bind = localhost
```

```
shaman.sources.netcat_s1.port = 44444# Describe the sink:
```

```
shaman.sinks.hdfs_w1.type = hdfs
```

```
shaman.sinks.hdfs_w1.hdfs.path = hdfs://localhost:8020  
/user/root/test
```

```
shaman.sinks.hdfs_w1.hdfs.writeFormat = Text
```

```
shaman.sinks.hdfs_w1.hdfs.fileType = DataStream
```

```
# Configure a channel that buffers events in memory:
```

```
shaman.channels.in-mem_c1.type = memory
```

```
shaman.channels.in-mem_c1.capacity = 20000
```

```
shaman.channels.in-mem_c1.transactionCapacity = 100# Bi
```

nd the source and sink to the channel:

```
shaman.sources.netcat_s1.channels = in-mem_c1
```

```
shaman.sinks.hdfs_w1.channel = in-mem_c1
```

备注:

hdfs://localhost:8020/user/root/test, 其中 `hdfs://localhost:8020` 为 hadoop 配置文件 `core-site.xml` 中 `fs.defaultFS` 属性的值, `root` 为 hadoop 的登陆用户。

2, 启动 flume 代理

```
bin/flume-ng agent -f agent/netcat-memory-hdfs.conf -n  
  shaman -Dflume.root.logger=DEBUG,console -Dorg.apache.  
flume.log.printconfig=true -Dorg.apache.flume.log.rawd  
ata=true
```

3, 打开 telnet 客户端, 输入字母测试

```
telnet localhost 44444
```

然后输入文字

4, 查看 hdfs test 目录

```
hdfs dfs -ls /user/root/test
```

会发现有新的文件出现, 文件里面的内容即是通过 telnet 输入的字母

1.4 项目

1.4.1 点击流日志模型

1.分析什么

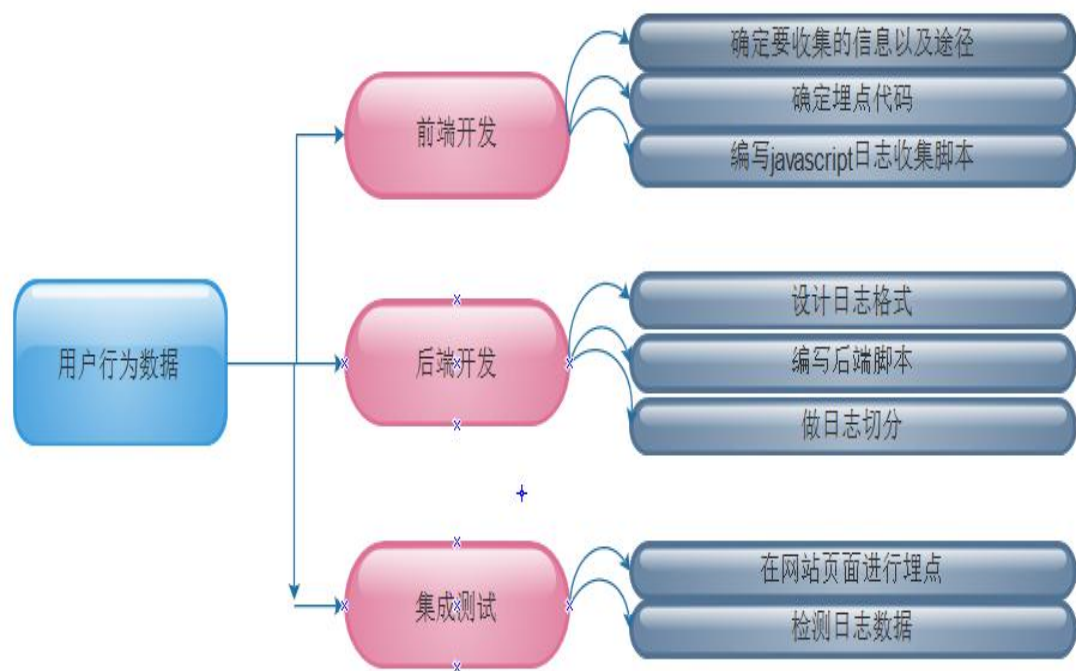
点击流日志模型主要是分析用户在网站上持续访问的轨迹，查看网站的访问量，分析关键路径得到漏斗模型，以此来优化网站，提高用户的网站停留时间。

2.确定数据源

在本次案例中，使用埋点的方式收集数据，埋点是指在网页中插入小段 javascript 代码，这个代码片段会动态创建一个 script 标签，并将 src 属性指向一个单独的 js 文件，此时这个单独的 js 文件会被浏览器请求到并执行，这个 js 往往就是真正的数据收集脚本。

数据收集完成后，js 会请求一个后端的数据收集脚本，这个脚本一般是一个伪装成图片的动态脚本程序，js 会将收集的数据通过 http 参数的方式传递给后端脚本，后端脚本解析参数并按固定格式记录到访问日志，同时可能会在 http 响应中给客户端种植一些用于追踪的 cookie。

设计实现



确定收集信息

名称	途径	备注
访问时间	web server	Nginx \$msec
IP	web server	Nginx \$remote_addr
域名	javascript	document.domain
URL	javascript	document.URL
页面标题	javascript	document.title
分辨率	javascript	window.screen.height & width
颜色深度	javascript	window.screen.colorDepth
Referrer	javascript	document.referrer
浏览客户端	web server	Nginx \$http_user_agent
客户端语言	javascript	navigator.language
访客标识	cookie	Nginx \$http_cookie
网站标识	javascript	自定义对象
状态码	web server	Nginx \$status
发送内容量	web server	Nginx \$body_bytes_sent

1) 确定埋点代码

```
<script type="text/javascript">
    val _map=_map || [];
    _map.push(['_setAccount'], ' UA-XXXXX-X');

//自调用匿名函数只会在运行时执行一次，一般用于初始化
(function() {
    //引入一个外部的 js 文件 ma
    var ma = document.createElement('script');
    ma.type = 'text/javascript';
    //异步调用外部 js 文件
    ma.async = true;
    //根据协议将 src 指向对应的 ma.js
    ma.src = ('https:' == document.location.protocol ?
```

```

        'https://ssl' : 'http://www') + '.google-analytics.com/ma.js';
//将这个元素插入到 dom 树上
var s = document.getElementsByTagName('script')[0];
s.parentNode.insertBefore( ma, s);

})();
</script>

```

2) 前端收集脚本

在第二步配置中的 ma.js 被请求后会被执行，一般要做如下几件事：

- A、通过浏览器内置 js 对象收集信息
- B、解析_map 数组，收集配置信息，这里面可能会包括用户自定义的事件跟踪，业务数据等
- C、将上面两步收集的数据按预定义格式解析并拼接
- D、请求一个后端脚本，将信息放在 http request 参数中携带给后端脚本

示例代码

```

(function(){
    var params={};
    //document 对象数据，实现了 A 步骤
    if(document){
        params.domain=document.domain||'';
        params.url=document.URL||'';
        params.title=document.title||'';
        params.referrer=document.referrer||'';
    }
    //window 对象数据
    if(window&&window.screen){
        params.sh=window.screen.height||0;
        params.sw=window.screen.width||0;
        params.cd=window.screen.colorDepth||0;
    }
    //navigator 对象数据
    if(navigator){
        params.lang=navigator.language||'';
    }
    //接卸 map 数组。收集配置信息
    if(_maq){
        for(var i in _maq){
            switch(_map[i][0]){
                case '_setAccount':
                    params.account=_maq[i][1];
                    break;
                default:
                    break;
            }
        }
    }
}

```

```

    }
}
}
//拼接参数串
var args="";
for(var i in params){
    if(ags!="")args+='&';
    args += i + '=' + encodeURIComponent(params[i]);
}
//通过 image 对象请求后端脚本
var img=new Image(1,1);
//请求后端脚本常用的是 ajax，但是 ajax 是不能跨域请求的，通用的方
//法是 js 脚本创建一个 image 对象，将 image 对象的 src 属性指向后端脚
//本并携带参数,就实现了跨域请求。
img.src='http://xx.xxx.xxx/log.gif?' + args;
})();

```

3)后端脚本

log.gif 是后端脚本，是一个伪装成 gif 图片的脚本，后端脚本一般需要完成以下几件事情

- A、解析 http 请求参数得到信息
- B、从 web 服务器获取一些客户端无法获取的信息
- C、将信息按格式写入 log
- D、生成一副 1*1 的 gif 图片作为响应内容并将响应头的 content-type 设置为 image/gif
- E、在响应头中通过 set-cookie 设置一些需要的 cookie 信息

我们使用 nginx 的 access_log 做日志收集

首先，需要在 nginx 的配置文件中定义日志格式：

```

log_format tick
"$msec|$remote_addr|$status|$body_bytes_sent|$u_domain|$u_url|
|$u_title|$u_referrer|$u_sh|$u_sw|$u_cd|$u_lang|$http_user_ag
ent|$u_account";

```

注意这里以 u_开头的是我们待会会自己定义的变量，其它的是 nginx 内置变量。然后是核心的两个 location：

```

location / log.gif {
#伪装成 gif 文件
default_type image/gif;
#本身关闭 access_log，通过 subrequest 记录 log
access_log off;
access_by_lua "
-- 用户跟踪 cookie 名为__utrace

```

```

local uid = ngx.var.cookie__utrace
if not uid then
-- 如果没有则生成一个跟踪 cookie, 算法为
md5(时间戳+IP+客户端信息)
uid = ngx.md5(ngx.now() ..
ngx.var.remote_addr .. ngx.var.http_user_agent)
end
ngx.header['Set-Cookie'] = {'__utrace=' .. uid ..
'; path=/'}
if ngx.var.arg_domain then
-- 通过 subrequest 子请求 到/i-log 记录日志,
将参数和用户跟踪 cookie 带过去
ngx.location.capture('/i-log?' ..
ngx.var.args .. '&utrace=' .. uid)
end
",
1
#此请求资源本地不缓存
add_header Expires "Fri, 01 Jan 1980 00:00:00 GMT";
add_header Pragma "no-cache";
add_header Cache-Control "no-cache, max-age=0, must-
revalidate";
#返回一个 1×1 的空 gif 图片
empty_gif;
}
location /i-log {
#内部 location, 不允许外部直接访问
internal;
#设置变量, 注意需要 unescape, 来自 ngx_set_misc 模块
set_unescape_uri $u_domain $arg_domain;
set_unescape_uri $u_url $arg_url;
set_unescape_uri $u_title $arg_title;
set_unescape_uri $u_referrer $arg_referrer;
set_unescape_uri $u_sh $arg_sh;
set_unescape_uri $u_sw $arg_sw;
set_unescape_uri $u_cd $arg_cd;
set_unescape_uri $u_lang $arg_lang;
set_unescape_uri $u_account $arg_account;
#打开日志
log_subrequest on;
#记录日志到 ma.log 格式为 tick
access_log /path/to/logs/directory/ma.log tick;
#输出空字符串
echo "";
}

```

4) 日志切分

日志收集系统访问日志时间一长文件变得很大，而且日志放在一个文件不利于管理。通常要按时间段将日志切分，例如每天或每小时切分一个日志。通过 crontab 定时调用一个 shell 脚本实现，如下：

```
_prefix="/path/to/nginx"
time=`date +%Y%m%d%H`
mv ${_prefix}/logs/ma.log ${_prefix}/logs/ma/ma-${time}.log
kill -USR1 `cat ${_prefix}/logs/nginx.pid`
```

这个脚本将 ma.log 移动到指定文件夹并重命名为 ma-{yyyymmddhh}.log，然后向 nginx 发送 USR1 信号令其重新打开日志文件。

USR1 通常被用来告知应用程序重载配置文件，向服务器发送一个 USR1 信号将导致以下步骤的发生：停止接受新的连接，等待当前连接停止，重新载入配置文件，重新打开日志文件，重启服务器，从而实现相对平滑的不关机的更改。

cat \${_prefix}/logs/nginx.pid 取 nginx 的进程号

然后再/etc/crontab 里加入一行：

```
59 * * * * root /path/to/directory/rotatelog.sh
```

在每个小时的 59 分启动这个脚本进行日志轮转操作。

3.确定采集方案

在本次案例中，使用 Flume 日志采集系统来采集数据到 HDFS 系统。在服务器上部署 agent 节点，修改配置文件

配置文件如下

```
a1.sources=r1
```

```
a1.sinks=k1
```

```
a1.channels=c1
```

```
//监视指定的一些文件，将近实时的 tail 这些文件，获取这些文件的新追加的行
```

```
a1.sources.r1.type= TAILDIR
```

```
a1.sources.r1.channels=c1
```

```
//配置检查点文件的路径，检查点文件会以 json 格式保存已经 tail 文件的位置，解决了断点不能续传的缺陷
```

```
a1.sources.r1.positionFile=/export/log/flume/taildir_position.json
```

```
//可以指定要 tail 的组
```

```
a1.sources.r1.filegroups=f1 f2
```

```
//指定每个文件的全路径
```

```
a1.sources.r1.filegroups.f1=/export/log/test1/example.log
```

```
a1.sources.r1.filegroups.f2=/export/log/test2/*.log.*
```

```
//配置 channels
```

```
a1.channels.c1.type=memory
```

```
a1.channels.c1.capacity=1000
```

```
a1.channels.c1.transactionCapacity=100
```

```
//配置下沉点，将文件写到 hdfs 中去
```



```
a1.sinks.k1.type=hdfs
a1.sinks.k1.channel=c1
a1.sinks.k1.hdfs.path=/flume/events/%y-%m-%d/
a1.sinks.k1.hdfs.filePrefix=events-
a1.sinks.k1.hdfs.round=true
a1.sinks.k1.hdfs.roundValue=10
a1.sinks.k1.hdfs.roundUnit=minute
```

4. 确定三层架构

ETL 工作的实质就是从各个数据源提取数据，对数据进行转换，并最终加载填充数据到数据仓库维度建模后的表中。

1)创建 ODS 层数据表

A、创建原始数据表

```
drop table if exists ods_weblog_origin;
create table ods_weblog_origin(
    valid string,
    remote_addr string,
    remote_user string,
    time_local string,
    request string,
    status string,
    body_bytes_sent string,
    http_referer string,
    http_user_agent string)
partitioned by (datestr string)
row format delimited
fields terminated by '\001';
```

B、点击流模型 pageviews 表

```
drop table if exists ods_weblog_origin;
create table ods_weblog_origin(
    valid string,
    remote_addr string,
    remote_user string,
    time_local string,
    request string,
    visit_step string,
    page_staylong string,
    http_referer string,
    http_user_agent string,
```

```

body_bytes_sent string,
status string)
partitioned by(datestr string)
row format delimited fields terminated by '\001';

```

C、点击流 visit 模型

```

drop table if exists ods_click_stream_visit;
create table ods_click_stream_click(
    session string,
    remote_addr string,
    inTime string,
    outTime string,
    inPage string,
    outPage string,
    referral string,
    pageVisits int)
partitioned by (datestr string)
row format delimited fields terminated by '\001';

```

2)导入 ODS 层数据

```

load data inpath '/weblog/preprocessed/' overwrite into table
ods_weblog_origin partition(datastr='20180526');--数据导入
show partitions ods_weblog_origin;--查看分区

```

3)生成 ODS 层明细宽表

5.模块开发---统计分析

1)流量分析

A、多维度统计 PV 总量 按时间维度

(1) 计算每小时的 pv 数

```

select count(*) as pvs ,month,day,hour from ods_weblog_detail
group by month,day,hour;

```

(2) 计算该处理批次一天中的各个小时 pvs

```

drop table dw_pvs_everyhour_oneday;
create table dw_pvs_everyhour_oneday(month string,day
string,hour string ,pvs bigint) partitioned by(datestr);
insert into table dw_pvs_everyhour_oneday

```

```
partition(datestr="20180526") select a.month as month,a.day as
a.day,a.hour as hour,a.count(*) as pvs from ods_weblog_detail
where a.datastr='20180526' group by a.month,a.day,a.hour;
```

(3) 计算每天的 pvs

```
drop table dw_pvs_everyday;
create table dw_pvs_everyday(pvs bigint,month string,day string);
insert into table dw_pvs_everyday select count(*) as pvs,a.month
as month,a.day as day from ods_weblog_detail a group by
a.month,a.day;
```

B、人均浏览量

需求：统计今日所有来访者平均请求的页面数

计算方式：总页面请求数/去重总人数

remote_addr 表示不同的用户，可以先统计出不同的 remote_addr 的 pv 量，然后累加所有 pv 作为总的页面请求数，再 count 所有 remote_addr 作为总的去重总人数。

--总页面请求数/去重总人数

```
drop table dw_avgpv_user_everyday;
create table dw_avgpv_user_everyday(
    day string,
    avgpv string
);
Insert into table dw_avgpv_user_everyday
select '20180526',sum(b.pvs)/count(b.remote_addr)from (select
remote_addr, count(1) as pvs from ods_weblog_detail where
datestr='20180526' group by remote_addr)b;
```

C、统计 pv 总量最大的来源

统计每个小时各来访 host 的产生 pv 数最多的前 N 个 (topN)

```
drop table dw_pvs_refhost_topn_everyhour;
create table dw_pvs_refhost_topn_everyhour(
    hour string,toporder string,ref_host string,ref_host_cnts string)
partitioned by (datestr string);
insert into dw_pvs_refhost_topn_everyhour
select ref_host,ref_host_cnts,concat(month,day,hour),row_number()
over(partition by concat(month,day,hour) order by ref_host_cnts desc)as
od from dw_pvs_refererhost_everyhour where od<=3;
```

2)关键路径转化率分析（漏斗模型）

A、需求分析

在一条指定的业务流程中，各个步骤的完成人数及相对上一个步骤的百分比

B、模型设计

定义好业务流程中的页面标识，下例中的步骤为：

C、模型设计

查询每一个步骤的总访问人数

Create table dw_oute_numbs as

```
select 'step1' as step ,count(distinct remote_addr) as numbs from ods_clik_pageviews where  
datestr='20180526' and request like '/item%'
```

union

```
select 'step2' as step,count(distinct remote_addr) as numbs from ods_clik_pageviews where  
datestr='20180526' and request like '/category%'
```

union

```
select 'step3' as step,count(distinct remote_addr) as numbs from ods_clik_pageviews where  
datestr='20180526' and request like '/order%'
```

union

```
select 'step4' as step,count(distinct remote_addr) as numbs from ods_clik_pageviews where  
datestr='20180526' and request like '/index%'
```

查询每一步相对于路径起点人数的比例

```
select tmp.rnstep,tmp.rnnumbs/tmp.rnumbs as ratio from (select rn.step as  
rnstep,rn.rnumbs as rnumbs,rr.step as rrstep,rr.numbs as rnumbs from dw_oute_numbs rn  
inner join dw_out_numbs rr)tmp where tmp.rrstep='step1';
```

查询每一步相对于上一步骤的漏出率

```
Select tmp.rnstep,tmp.rnnumbs/tmp.rnumbs from (select rn.step as  
rnstep,rn.rnumbs as rnumbs,rr.step as rrstep,rr.numbs as rnumbs from dw_oute_numbs rn  
inner join dw_out_numbs rr )where  
cast(substr(tmp.rnstep,5,1),int)=cast(substr(tmp.rrstep,5,1),int)-1;
```

6 模块开发—结果导出

采用 apache sqoop 将数据导出

7 模块开发—数据可视化

采用 Echarts 对数据进行可视化展示。

相关面试题

01、 电商总体运营指标

02、 网站流量指标

1.4.2 推荐系统项目

推荐系统：本质上是商品售卖系统，和电商网站的目的是一样的，用来售卖商品。

会收集用户的所有行为信息（网站浏览信息、订单信息、关注信息、收藏商品、评论系统、外部信息（微博信息、联盟网站）），通过分析分析用户的历史行为给用户的兴趣建模，从而主动给用户推荐能够满足他们兴趣和需求的信息。

数据源：

基础数据的来源有很多维度，包括用户的访问、浏览、下单、收藏、用户的历史订单数据，评价信息等很多数据。

基础数据主要包括：

1. 要推荐物品或内容的元数据。例如关键字、属性描述等
2. 系统用户的基本信息，例如性别，年龄等
3. 用户对物品或者信息的偏好，包括用户对物品的评分，用户查看物品的记录，用户购买记录等

可以将这些用户的偏好信息分为两类：

显示的用户反馈：这类是用户在网站上自然浏览或者使用网站以外，显式的提供反馈信息，例如用户对物品的评分，或者对物品的评论。

隐式的用户信息：用户在使用网站是产生的数据，隐式的反应了用户对物品的喜好，例如用户查看了某物品的信息等等用户反馈。

数据的采集方案：

数据收集模块：

点击流模块：flume+kafka+storm+redis 处理用户当前浏览的信息，将信息计算出来，保存到 redis 中（比如用户对一个品类的偏好）

订单支付（AMQ）：开发一个消费者程序（storm、JavaAPP），用来计算用户的偏好

外部数据：通过爬虫技术爬取用户的社交网站数据。

合作数据：比如京东和腾讯合作，可以获取一些用户在腾讯上的信息。

1. 将用户产生的数据（浏览商品、关注商品、收藏商品、加入购物车、下订单、评论等）保存到消息队列（kafka）中，将不同类型的数据保存到不同的 topic 中，以作分类。
2. 另一种方式就是将用户产生的数据保存到分库分表的数据库中

数据的存储与计算：

一． 离线推荐

离线推荐计算数据的周期可以是一个月或者 15 天

1. 由 FTP 服务对推送的数据进行校验，将数据库中校验通过的数据定时的上传到 HDFS 文件系统中。
2. 基于 HDFS 建立 hive 数据仓库，将 HDFS 中的数据映射成一张张的表（用户表、收藏表、购物车表、订单表等）保存到 hive 数据仓库中。
3. 将 hive 数据仓库中的数据通过 HSQL 计算和导入到算法中（协同过滤算法）进行计算，得到一些用户的偏好数据。
4. 将计算出来的用户偏好数据定时导入到 hbase/redis 中。

5. 用户通过点击浏览商品，在推荐引擎（Javaweb）中获取用户 id，根据用户的 id 从 redis 或 hbase 中拿去已经计算好的推荐结果（离线计算）推荐给用户。在此过程中，还要对结果数据进行过滤，比如商品上下线状态的判定、推荐结果种类丰富性的判定、推荐结果数量的补足等。最后将计算过滤之后的商品展示给用户。

二．实时推荐

1. 通过 storm, 实时的消费消息队列 kafka 的 topic 中的数据, 实时的计算一些用户的偏好, 同样将计算好的偏好数据实时的导出到 hbase 或 redis 中, 等待用户点击浏览商品, 实时的将计算好的结果展示给用户。

大数据处理平台：

离线计算和实时计算

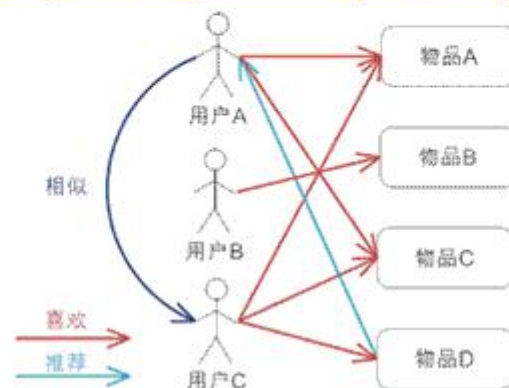
数据的计算：

协同过滤算法：也就是计算相似度，关于相似度的计算，就是计算像个向量的距离，距离越近相似度越大。

1. 基于用户的协同过滤算法

原理：基于用户对物品的偏好找到相邻邻居用户，然后将邻居用户喜欢的推荐给当前用户。

用户/物品	物品A	物品B	物品C	物品D
用户A	√		√	推荐
用户B		√		
用户C	√		√	√



假如：用户 A 喜欢物品 A、物品 C

用户 B 喜欢物品 B

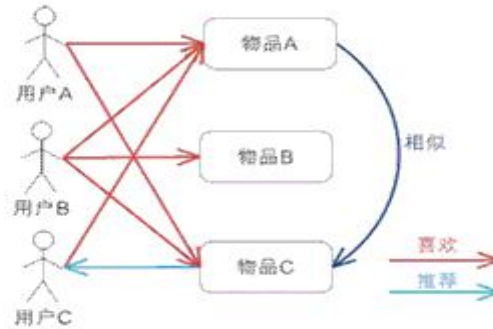
用户 C 喜欢物品 A、物品 C、物品 D

通过计算发现用户 A 和用户 C 相似，所以将用户 C 喜欢的物品 D 推荐给用户 A。

2. 基于物品的协同过滤算法

原理：与基于用户的类似，只是在计算邻居时采用物品本身，而不是从用户角度。基于用户对物品的偏好找到相似的物品，然后根据用户的历史偏好，推荐相似的物品给他。

用户/物品	物品A	物品B	物品C
用户A	√		√
用户B	√	√	√
用户C	√		推荐



如图：根据用户历史数据的偏好计算出喜欢物品 A 的用户大部分都喜欢物品 C，计算出物品 A 和物品 C 比较相似，而用户 C 喜欢物品 A，那么可以推断出用户 C 可能也喜欢物品 C，所以将物品 C 推荐给用户 C。

数据展现：

Web 网站

面试题：

- 1、推荐系统的本质是什么
- 2、推荐系统的指标是什么
- 3、推荐系统的 y 和 x
- 4、推荐系统的样本构造和数据拼接
- 5、推荐系统的场景思考
- 6、推荐系统相关组件

1.5 相关面试题

- 1、hadoop 运行原理
- 2、mapreduce 原理
- 3、mapreduce 的优化
- 4、举一个简单的例子说下 mapreduce 是怎么运行的
- 5、hadoop 中 combiner 的作用

- 6、简述 hadoop 的安装
- 7、请列出 hadoop 的进程名
- 8、简述 hadoop 的调度器
- 9、列出你开发 mapreduce 的语言
- 10、我们开发 job 时是否可以去掉 reduce 阶段
- 11、datanode 在什么情况下不会备份
- 12、combiner 出现在哪个过程
- 13、hdfs 的体系结构
- 14、3 个 datanode 中有一个 datanode 出现错误会怎么样
- 15、描述一下 hadoop 中，有哪些地方用了缓存机制，作用分别是什么？
- 16、如何确定 hadoop 集群的健康状况
- 17、shuffle 阶段，你怎么理解
- 18、mapreduce 的 map 数量和 reduce 数量怎么确定，怎么配置
- 19、简单说一下 mapreduce 的编程模型
- 20、hadoop 的 TextInputFormatter 作用是什么，如何自定义实现
- 21、hadoop 和 spark 都是并行计算，他们有什么相同和区别
- 22、为什么要用 flume 导入 hdfs，hdfs 的架构是怎样的
- 23、简单说一下 hadoop 和 spark 的 shuffle 过程
- 24、hadoop 高并发
- 25、map-reduce 程序运行的时候会有什么比较常见的问题

第 2 章 数据分析及其步骤

2.1 数据分析定义

数据分析离不开数据，计量和记录一起促成了数据的诞生。

数据分析是指用适当的统计分析方法对收集来的数据进行分析，将它们加以汇总和理解并消化，以求最大化地开发数据的功能，发挥数据的作用。

数据分析的目的是把隐藏在一大批看似杂乱无章的数据背后的信息集中和提炼出来，总结出所研究对象的内在规律。

数据分析可划分为：描述性数据分析、探索性数据分析、验证性数据分析。

描述性数据分析属于初级数据分析，另两个属于高级数据分析。其中探索性分析侧重于在数据之中发现新的特征，而验证性数据分析则侧重于验证已有假设的真伪证明。我们日常学习和工作中所涉及的数据分析主要是描述性数据分析。

2.2 数据分析的作用

在商业领域中，数据分析的目的是把隐藏在数据背后的信息集中和提炼出来，总结出所研究对象的内在规律，帮助管理者进行有效的判断和决策。

数据分析在企业日常经营分析中主要有三大作用：

2.2.1 现状分析

简单来说就是告诉你当前的状况。具体体现在：

第一，告诉你企业现阶段的整体运营情况，通过各个指标的完成情况来衡量

企业的运营状态，以说明企业整天运营是好了还是坏了，好的程度如何，坏的程度又到哪里。

第二，告诉你[企业各项业务的构成](#)，让你了解企业各项业务的发展以及变动情况，对企业运营状况有更深入的了解。

2.2.2 原因分析

简单来说就是告诉你[某一现状为什么发生](#)。

经过现状分析，我们对企业的运营情况有了基本了解，但不知道运营情况具体好在哪里，差在哪里，是什么原因引起的。这时就需要开展原因分析，以[进一步确定收入下降的具体原因，对运营策略做出调整与优化](#)。

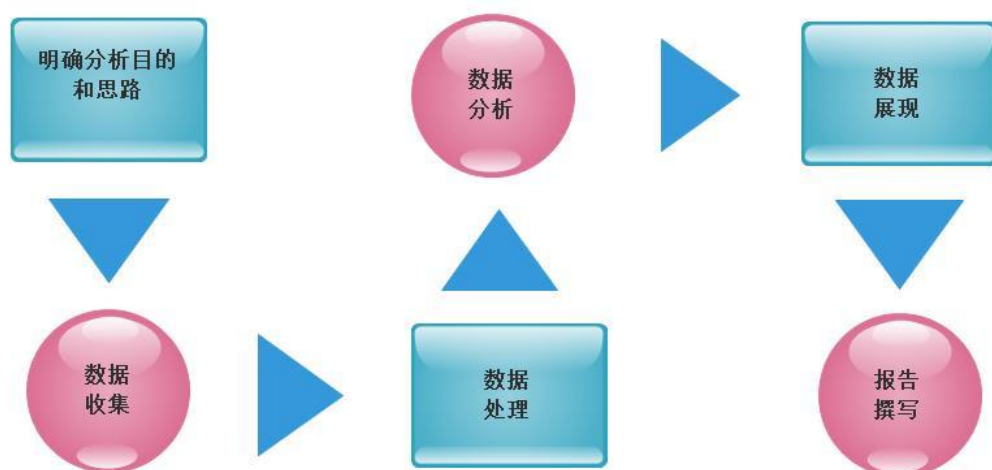
2.2.3 预测分析

简单来说就是告诉你[将来会发生什么](#)。

在了解企业运营现状后，有时还需要对企业未来发展趋势做出预测，[为制订企业运营目标及策略提供有效的参考与决策依据，以保证企业的可持续健康发展](#)。预测分析一般通过专题分析来完成，通常在制订企业季度、年度等计划时进行，其开展的频率没有现状分析及原因分析高。

2.3 数据分析基本步骤（重点）

典型的数据分析包含以下几个步骤：



2.3.1 明确分析目的和思路

确保数据分析过程有效进行的先决条件，[为数据的收集、处理及分析提供清晰的指引方向](#)。

[目的是整个分析流程的起点。目的不明确则会导致方向性的错误。](#)

[思考:为什么要开展数据分析，通过这次数据分析要解决什么问题？](#)

当明确目的后，就要[校理分析思路，并搭建分析框架，把分析目的分解成若干个不同的分析要点，即如何具体开展数据分析。需要从哪几个角度进行分析，采用哪些分析指标。](#)

只有明确了分析目的，分析框架才能跟着确定下来，最后还要确保分析框架的体系化，使分析更具有说服力。

体系化也就是逻辑化，简单来说就是先分析什么，后分析什么，使得各个分析点之间具有逻辑联系。避免不知从哪方面入手以及分析的内容和指标被质疑是否合理、完整。所以[体系化就是为了让你的分析框架具有说服力](#)。要想使分析框架体系化，就需要一些营销、管理等理论为指导，结合着实际的业务情况进行构

建，这样才能保证分析维度的完整性，分析结果的有效性以及正确性。

比如以用户行为 理论为指导，搭建的互联网网站分析指标框架如下：



把跟数据分析相关的营销、管理等理论统称为**数据分析方法论**。比如用户行为理论、PEST 分析法、5W2H 分析法等等

扩展

1) 数据分析方法论与数据分析区别

数据分析方法论主要用来指导数据分析师进行一次完整的数据分析,它更多的是指**数据分析思路**。数据分析方法论主要从**宏观角度**指导如何进行数据分析,它就像是一个数据分析的前期规划,指导着后期数据分析工作的开展。

而**数据分析法**则是指**具体的分析方法**。数据分析法主要从**微观角度**指导如何

进行数据分析。

2) 常用的数据分析方法论

PEST 分析法

PEST 分析法用于对宏观环境的分析。宏观环境又称一般环境，是指影响一切行业和企业的所有宏观力量。对宏观环境因素作分析时。由于不同行业和企业有其自身特点和经营需要，分析的具体内容会有差异，但一般都应对 **政治 (Political)**、**经济 (Economic)**、**技术 (Technological)** 和 **社会 (Social)** 这四类影响企业的主要外部环境因素进行分析，这种方法简称为 PEST 分析法。

5W2H 分析法

5w2H 分析法是以五个 w 开头的英语单词和两个 H 开头的英语单词进行提问，从回答中发现解决问题的线索，即 **何因 (Why)**、**何事(What)**、**何人(Who)**、**何时(When)**、**何地(Where)**、**如何做(How)**、**何价(How much)**，这就构成了 5W2H 分析法的总框架。

逻辑树分析法

逻辑树又称问题树、演绎树或分解树等。它是**将问题的所有问题分层罗列，从最高层开始，并逐步向下扩展。**

把一个已知问题当成树干，然后开始考虑这个问题和哪些相关问题有关。每想到一点，就给这个问题所在的树干加一个“树枝”，并标明这个“树枝”代表什么问题。

逻辑树的使用必须遵循以下三个原则。

要素化：把相同的问题总结归纳成要素。

框架化：将各个要素组织成框架。遵守不重不漏的原则。

关联化：框架内的各要素保持必要的相互关系，简单而不独立。

4P 营销理论

营销组合实际上有几十个要素，这些要素可以概括为 4 类：**产品** (Product) 、 **价格**(Price) 、 **渠道**(Place) 、 **促销**(Promotion)。☂

用户行为理论

网站分析的发展已经较为成熟，有一套成熟的分析指标。比如 IP、PV、页面停留时间、跳出率、回访者、新访问者、回访次数、回访相隔天数、流失率、关键字搜索、转化率、登录率，等等。需要我们梳理指标之间的逻辑关系。

用户使用行为是指用户为获取、使用物品或服务所采取的各种行动，用户对产品首先需要有一个认知、熟悉的过程，然后试用，再决定是否继续消费使用，最后成为忠诚用户。

可利用用户使用行为理论，梳理网站分析的各关键指标之间的逻辑关系，构建符合公司实际业务的网站分析指标体系。

2.3.2 数据收集

数据收集是**按照确定的数据分析框架，收集相关数据的过程，它为数据分析提供了素材和依据。**

包括第一手数据与第二手数据，第一手数据主要指可直接获取的数据，第二手数据主要指经过加工整理后得到的数据。

一般数据来源主要有以下几种方式：

数据库：每个公司都有自己的业务数据库，存放从公司成立以来产生的相关业务数据。这个业务数据库就是一个庞大的数据资源，需要有效地利用起来。

公开出版物：可以用于收集数据的公开出版物包括《中国统计年鉴》《中国社会统计年鉴》《中国人口统计年鉴》《世界经济年鉴》《世界发展报告》等统计年鉴或报告。

互联网：随着互联网的发展，网络上发布的数据越来越多，特别是搜索引擎可以帮助我们快速找到所需要的数据，例如国家及地方统计局网站、行业组织网站、政府机构网站、传播媒体网站、大型综合门户网站等上面都可能有我们需要的数据。

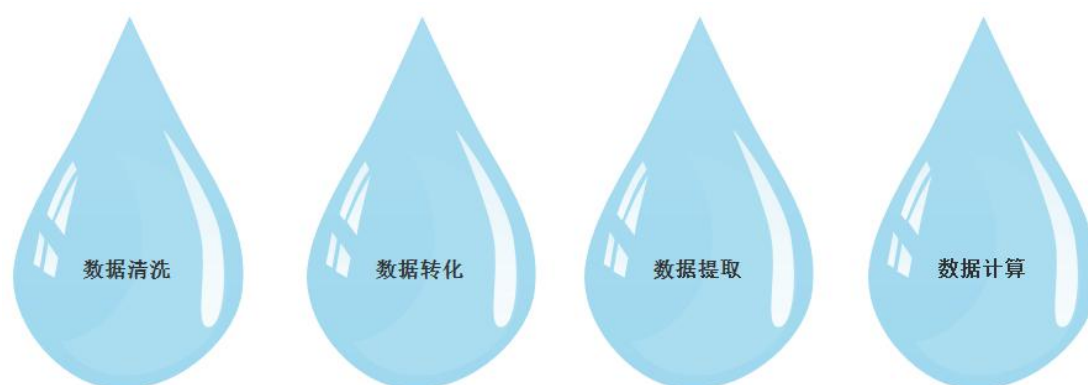
市场调查：市场调查就是指运用科学的方法，有目的、有系统地收集、记录、整理有关市场营销的信息和资料，分析市场情况，了解市场现状及其发展趋势，为市场预测和营销决策提供客观、正确的数据资料。

市场调查可以弥补其他数据收集方式的不足（比如：需要了解用户的想法与需求），但进行市场调查所需的费用较高，而且会存在一定的误差，故仅作参考之用。

2.3.3 数据处理

数据处理是指对收集到的数据进行加工整理，形成适合数据分析的样式，它是数据分析前必不可少的阶段。

基本目的:从大量的、杂乱无章、难以理解的数据中，抽取并推导出对解决问题有价值、有意义的数据。



数据处理主要包括数据清洗、数据转化、数据提取、数据计算等处理方法。

一般拿到手的数据都需要进行一定的处理才能用于后续的数据分析工作，即使再“干净”的原始数据也需要先进行一定的处理才能使用。数据处理是数据分析的基础。通过数据处理，将收集到的原始数据转换为可以分析的形式，并且保证数据的一致性和有效性。

2.3.4 数据分析

数据分析是指用适当的分析方法及工具，对处理过的数据进行分析，提取有价值的信息，形成有效结论的过程。

由于数据分析多是通过软件来完成的，这就要求数据分析师不仅要掌握各种数据分析方法，还要熟悉数据分析软件的操作。

数据挖掘其实是一种高级的数据分析方法，就是从大量的数据中挖掘出有用

的信息，它是根据用户的特定要求，从浩如烟海的数据中找出所需的信息，以满足用户的特定需求。数据挖掘技术是人们长期对数据库技术进行研究和开发的结果。一般来说，数据挖掘侧重解决四类数据分析问题：分类、聚类、关联和预测，重点在寻找模式和规律。数据分析与数据挖掘的本质是一样的，都是从数据里面发现关于业务的知识。

2.3.5 数据展现（数据可视化）

一般情况下，数据是通过表格和图形的方式来呈现的，我们常说用图表说话就是这个意思。常用的数据图表包括饼图、柱形图、条形图、折线图、散点图、雷达图等，当然可以对这些图表进一步整理加工，使之变为我们所需要的图形，例如金字塔图、矩阵图、漏斗图等。大多数情况下，人们更愿意接受图形这种数据展现方式，因为它能更加有效、直观地传递出分析所要表达的观点。

一般情况下能用图说明问题的就不用表格，能用表格说明问题的就不要用文字。

2.3.6 报告撰写

数据分析报告其实是对整个数据分析过程的一个总结与呈现。通过报告，把数据分析的起因、过程、结果及建议完整地呈现出来，供决策者参考。

一份好的数据分析报告，首先需要有一个好的分析框架，并且图文并茂，层次明晰，能够让阅读者一目了然。

另外，数据分析报告需要有明确的结论，没有明确结论的分析称不上分析，同时也失去了报告的意义，因为我们最初就是为寻找或者求证一个结论才进行分析的，所以千万不要舍本求末。

最后，好的分析报告一定要有建议或解决方案。作为决策者，需要的不仅仅是找出问题，更重要的是建议或解决方案，以便他们做决策时作参考。所以，数据分析师不仅需要掌握数据分析方法，而且还要了解和熟悉业务，这样才能根据发现的业务问题，提出具有可行性的建议或解决方案。

2.4 行业前景

2.4.1 蓬勃发展的趋势

市场巨大

许多企业都在讨论大数据分析，也有实际的需求并愿意为此付钱，但是比较零碎尚不系统化。

目前对数据需求最强烈的行业依次是：金融机构，以广告投放及电商为代表的互联网企业等；

尚没出现平台级公司的模式

（这或许往往是大市场或者大机会出现之前的混沌期）

企业技术外包的氛围在国内尚没完全形成

对于一些有能力的技术公司，如果数据需求强烈的话，考虑到自身能力的健全以及数据安全性，往往不会外包或者采用外部模块，而倾向于自建这块业务；

整个行业很大而且需求旺盛

未来 BAT 及京东、58 和滴滴打车等企业，凭借其自身产生的海量数据,必然是数据领域的大玩家。即使没有留给创业公司出现平台级巨型企业的机会，也将留出各种各样的细分市场机会让大家可以获得自己的领地。

2.4.2 数据分析师的职业要求

懂业务：从事数据分析工作的前提就是需要懂业务，即熟悉行业知识、公司业务及流程，最好有自己独特见解，若脱离行业认知和公司业务背景，分析的结果只会是脱了线的风筝，没有太大的实用价值。

懂管理：一方面是搭建数据分析框架的要求。另一方面的作用是针对数据分析结论提出有指导意义的分析建议，如果没有管理理论的支撑，就难以确保分析建议的有效性。

懂分析：是指掌握数据分析的基本原理与一些有效的数据分析方法，并能灵活运用到实践工作中，以便有效地开展数据分析。

懂工具：是指掌握数据分析相关的常用工具。

数据分析工具就是实现数据分析方法理论的工具，面对越来越庞大的数据，依靠计算器进行分析是不现实的，必须利用强大的数据分析工具完成数据分析工作。

懂设计：是指运用图表有效表达数据分析师的分析观点，使分析结果一目了然。

面试题

1. 数据清洗后最后会保留哪些数据？
2. 数据采集过来怎么进行处理的？
3. 数据清洗的规则.
4. 需求分析是什么，需求文档是什么.

5. 关于集群搭建的大概思路

2.5 大数据时代

2.5.1 大数据的含义

早在 1980 年，著名未来学家阿尔文·托夫勒便在《第三次浪潮》一书中，将大数据热情地赞颂为“第三次浪潮的华彩乐章”。从 2009 年开始，“大数据”才成为互联网信息技术行业的流行词汇。2012 年，大数据概念炙手可热，2013 年，大数据走向实践，有的专家称之为“大数据元年”。美国互联网数据中心指出，互联网上的数据每年将增长 50%，每两年将翻一番，而目前世界上 90% 以上的数据是最近几年才产生的。

最早提出“大数据”时代到来的是全球知名咨询公司麦肯锡，麦肯锡称：“数据，已经渗透到当今每一个行业和业务职能领域，成为重要的生产因素。人们对于海量数据的挖掘和运用，预示着新一波生产率增长和消费者盈余浪潮的到来。”“大数据”在物理学、生物学、环境生态学等领域以及军事、金融、通讯等行业存在已有时日，却因为近年来互联网和信息行业的发展而引起人们关注。

2.5.2 产生背景

进入 2012 年，大数据（big data）一词越来越多地被提及，人们用它来描述和定义信息爆炸时代产生的海量数据，并命名与之相关的技术发展与创新。它已经上过《纽约时报》《华尔街日报》的专栏封面，进入美国白宫官网的新闻，现身在国内一些互联网主题的讲座沙龙中，甚至被嗅觉灵敏的国金证券、国泰君安、

银河证券等写进了投资推荐报告。数据正在迅速膨胀并变大，它决定着企业的未来发展，虽然很多企业可能并没有意识到数据爆炸性增长带来问题的隐患，但是随着时间的推移，人们将越来越多的意识到数据对企业的重要性。正如《纽约时报》2012年2月的一篇专栏中所称，“大数据”时代已经降临，在商业、经济及其他领域中，决策将日益基于数据和分析而作出，而并非基于经验和直觉。

哈佛大学社会学教授加里·金说：“这是一场革命，庞大的数据资源使得各个领域开始了量化进程，无论学术界、商界还是政府，所有领域都将开始这种进程。”

2.5.3 影响

1 大数据

现在的社会是一个高速发展的社会，科技发达，信息流通，人们之间的交流越来越密切，生活也越来越方便，大数据就是这个高科技时代的产物。

大数据到底有多大？一组名为“互联网上一天”的数据告诉我们，一天之中，互联网产生的全部内容可以刻满1.68亿张DVD；发出的邮件有2940亿封之多（相当于美国两年的纸质信件数量）；发出的社区帖子达200万个（相当于《时代》杂志770年的文字量）；卖出的手机为37.8万台，高于全球每天出生的婴儿数量37.1万……

2 大数据的精粹

大数据带给我们的三个颠覆性观念转变：是全部数据，而不是随机采样；是大体方向，而不是精确制导；是相关关系，而不是因果关系。

A.不是随机样本，而是全体数据：在大数据时代，我们可以分析更多的数据，

有时候甚至可以处理和某个特别现象相关的所有数据，而不再依赖于随机采样(随机采样，以前我们通常把这看成是理所应当的限制，但高性能的数字技术让我们意识到，这其实是一种人为限制);

B.不是精确性，而是混杂性：研究数据如此之多，以至于我们不再热衷于追求精确度;之前需要分析的数据很少，所以我们必须尽可能精确地量化我们的记录，随着规模的扩大，对精确度的痴迷将减弱;拥有了大数据，我们不再需要对一个现象刨根问底，只要掌握了大体的发展方向即可，适当忽略微观层面上的精确度，会让我们在宏观层面拥有更好的洞察力;

C.不是因果关系，而是相关关系：我们不再热衷于找因果关系，寻找因果关系是人类长久以来的习惯，在大数据时代，我们无须再紧盯事物之间的因果关系，而应该寻找事物之间的相关关系;相关关系也许不能准确地告诉我们某件事情为何会发生，但是它会提醒我们这件事情正在发生。

3 数据价值

十年前，葛大爷曾说过，“21 世纪什么最贵?”——“人才”，深以为然。只是，十年后的今天，大数据时代也带来了身价不断翻番的各种数据。由于急速拓展的网络带宽以及各种穿戴设备所带来的大量数据，数据的增长从未停歇，甚至呈井喷式增长。

一分钟内，微博推特上新发的数据量超过 10 万；社交网络“脸谱”的浏览量超过 600 万……

这些庞大数字，意味着什么？

它意味着，一种全新的致富手段也许就摆在面前，它的价值堪比石油和黄金。

这些数据都能干啥。具体有六大价值：

3.3.1、华尔街根据民众情绪抛售股票；

3.3.2、对冲基金依据购物网站的顾客评论，分析企业产品销售状况；

3.3.3、银行根据求职网站的岗位数量，推断就业率；

3.3.4、投资机构搜集并分析上市企业声明，从中寻找破产的蛛丝马迹；

3.3.5、美国疾病控制和预防中心依据网民搜索，分析全球范围内流感等病疫的传播状况；

3.3.6、美国总统奥巴马的竞选团队依据选民的微博，实时分析选民对总统竞选人的喜好；

4 可视化

“数据是新的石油。”亚马逊前任首席科学家 Andreas Weigend 说。Instagram 以 10 亿美元出售之时，成立于 1881 年的世界最大影像产品及服务商柯达正申请破产。

大数据是如此重要，以至于其获取、储存、搜索、共享、分析，乃至可视化地呈现，都成为了当前重要的研究课题。

2.5.4 特征

大数据的 4V 的特征:Volume(大量)、Variety(多样)、Velocity(高速)、Value(价值)；

第一个特征是数据量大。大数据的起始计量单位至少是 P (1000 个 T)、E (100 万个 T) 或 Z (10 亿个 T)。

第二个特征是数据类型繁多。包括网络日志、音频、视频、图片、地理位置

信息等等，多类型的数据对数据的处理能力提出了更高的要求。

第三个特征是处理速度快，时效性要求高。这是大数据区分于传统数据挖掘最显著的特征。

第四个特征是数据价值密度相对较低。如随着物联网的广泛应用，信息感知无处不在，信息海量，但价值密度较低，如何通过强大的机器算法更迅速地完成任务的价值“提纯”，是大数据时代亟待解决的难题。

2.5.5 思维变革

当数据的处理技术发生翻天覆地的变化时，大数据时代，我们的思维也要变革。

第一个思维变革：利用所有的数据，而不再仅仅依靠部分数据，即不是随机样本，而是全体数据。

第二个思维变革：我们唯有接受不精确性，才有机会打开一扇新的世界之窗，即不是精确性，而是混杂性。

2.5.6 第三个思维变革：

不是所有的事情都必须知道现象背后的原因，而是要让数据自己“发声”，即不是因果关系，而是相关关系。

2.5.7 科技发展带来的挑战

在科技的快速发展推动下，在 IT 领域，企业会面临两个方面的问题。

一是如何实现网站的高可用、易伸缩、可扩展、高安全等目标。为了解决这样一系列问题，迫使网站的架构在不断发展。从单一架构迈向高可用架构，这过程中不得不提的就是分布式。

二是用户规模越来越大，由此产生的数据也在以指数倍增长，俗称数据大爆炸。海量数据处理的场景也越来越多。

2.5.8 分布式系统

1 概述：

分布式系统是一个硬件或软件组件分布在不同的网络计算机上，彼此之间仅仅通过消息传递进行通信和协调的系统。简单来说就是一群独立计算机集合共同对外提供服务，但是对于系统的用户来说，就像是一台计算机在提供服务一样。

2 特征：

分布性：分布式系统中的多台计算机之间在空间位置上可以随意分布，系统中的多台计算机之间没有主、从之分，即没有控制整个系统的主机，也没有受控的从机。

透明性：系统资源被所有计算机共享。每台计算机不仅可以使⽤本机的资源，还可以使⽤分布式系统中其他计算机的资源(包括 CPU、文件、打印机等)。

同一性：系统中的若干台计算机可以互相协作来完成一个共同的任务，或者说一个程序可以分布在几台计算机上并行地运行。

通信性：系统中任意两台计算机都可以通过通信来交换信息。

3 常用分布式方案：

A.分布式应用和服务

将应用和服务进行分层和分割，然后将应用和服务模块进行分布式部署。这样做不仅可以提高并发访问能力、减少数据库连接和资源消耗，还能使不同应用复用共同的服务，使业务易于扩展。比如：分布式服务框架 Dubbo。

B.分布式静态资源

对网站的静态资源如 JS、CSS、图片等资源进行分布式部署可以减轻应用服

务器的负载压力，提高访问速度。比如：CDN。

C.分布式数据和存储

大型网站常常需要处理海量数据，单台计算机往往无法提供足够的内存空间，可以对这些数据进行分布式存储。比如 Apache Hadoop HDFS。

D.分布式计算

随着计算技术的发展，有些应用需要非常巨大的计算能力才能完成，如果采用集中式计算，需要耗费相当长的时间来完成。分布式计算将该应用分解成许多小的部分，分配给多台计算机进行处理。这样可以节约整体计算时间，大大提高计算效率。比如 Apache Hadoop MapReduce。

6.1.4 分布式、集群

分布式（distributed）是指在多台不同的服务器中部署不同的服务模块，通过远程调用协同工作，对外提供服务。

集群（cluster）是指在多台不同的服务器中部署相同应用或服务模块，构成一个集群，通过负载均衡设备对外提供服务。

2.6 面试题:

2.6.1 分布式和集群有啥区别呢？

集群和分布式都是由多个节点组成，但是集群之间的通信协调基本不需要，而分布式各个节点的通信协调必不可少。

集群主要是为了应对请求压力的分担，从而有了 LB，负载均衡集群；为了应对可用性，从而有了 HA，高可用性集群；为了更强的性能，从而有了 HP，高性能集群；为了高并发大规模性能，从而有分布式系统集群。

2.6.2 集群 负载均衡 分布式 有什么区别？

服务器集群：

服务器集群就是指将很多服务器集中起来一起进行同一种服务，在客户端看来就像是只有一个服务器。集群可以利用多个计算机进行并行计算从而获得很高的计算速度，也可以用多个计算机做备份，从而使得任何一个机器坏了整个系统还是能正常运行。

服务器负载均衡：

负载均衡（Load Balancing）建立在现有网络结构之上，它提供了一种廉价有效透明的方法扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的灵活性和可用性。

分布式服务器：

所谓分布式资源共享服务器就是指数据和程序可以不位于一个服务器上，而是分散到多个服务器，以网络上分散分布的地理信息数据及受其影响的数据库操作为研究对象的一种理论计算模型服务器形式。分布式有利于任务在整个计算机系统上进行分配与优化，克服了传统集中式系统会导致中心主机资源紧张与响应瓶颈的缺陷，解决了网络 GIS 中存在的异构、数据共享、运算复杂等问题，是地理信息系统技术的一大进步。

这三种架构都是常见的服务器架构，集群的主要是 IT 公司在做，可以保障重要数据安全；负载均衡主要是为了分担访问量，避免临时的网络堵塞，主要用于电子商务类型的网站；分布式服务器主要是解决跨区域，多个单个节点达到高速访问的目的，一般是类似 CDN 的用途的话，会采用分布式服务器。

2.6.3 Web 流量日志数据自定义采集

1 网站流量日志数据分析系统

数据分析：用适当的统计分析方法对收集来的数据进行分析，将这些数据加以汇总和理解消化，追求最大化开发数据的功能，发挥数据的作用。

本小节主要介绍点击流日志采集。

网站流量日志数据分析系统主要目的：帮助网站管理员、运营员、推广员等实时获取网站流量信息，从而提高网站流量，提升网站用户体验，让更多的访客转沉淀下来称为会员或客户，以更少的投入获取最大化的收入

网站流量日志数分析系统图如图 1 所示。

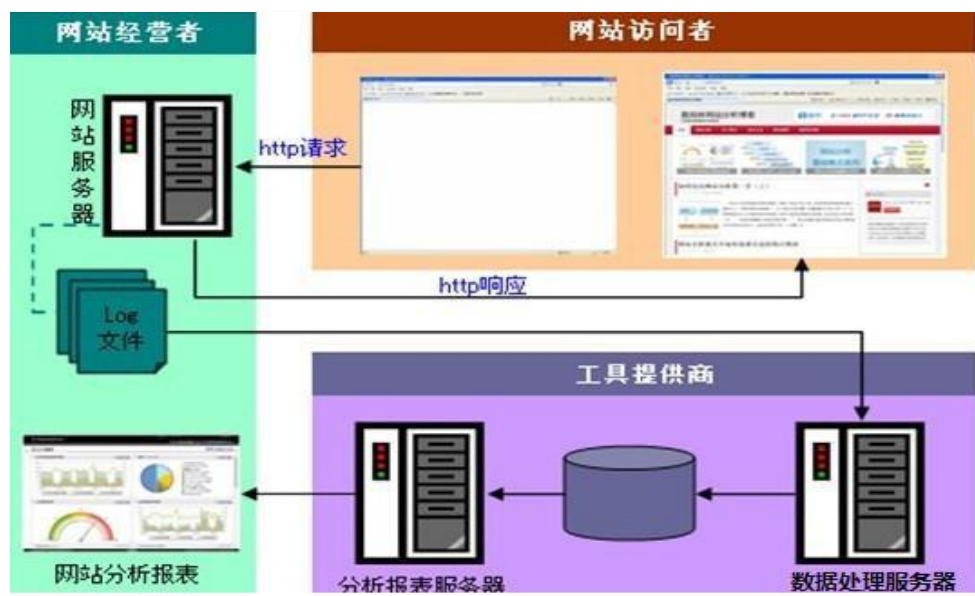


图 1 网站流量日志数分析系统

Web 访问日志：用户访问网站时所有访问、浏览、点击行为数据。比如点击了哪一个链接，打开了哪一个页面，采用了哪个搜索项、总体会话时间等。而所有这些信息都可通过网站日志保存下来。通过分析这些数据，可以获知许多对网站运营至关重要的信息。采集的数据越全面，分析就能越精准。

日志生成渠道主要包括以下两种：

一是：web 服务器软件(httpd、nginx、tomcat)自带的日志记录功能，如 Nginx

的 access.log 日志;

二是: 自定义采集用户行为数据, 通过在页面嵌入自定义的 javascript 代码来获取用户的访问行为 (比如鼠标悬停的位置, 点击的页面组件等), 然后通过 ajax 请求到后台记录日志, 这种方式所能采集的信息会更加全面。

Tips:

tomcat 服务器的资源存放在 webapp 目录下;

nginx 服务器资源存放在 html 目录下;

https 服务器的资源存放在 /var/www/html/目录下。

在实际操作中, 有以下几个方面的数据可以自定义的采集:

- ✧ 系统特征: 比如所采用的操作系统、浏览器、域名和访问速度等。
- ✧ 访问特征: 包括停留时间、点击的 URL、所点击的“页面标签<a>”及标签的属性等。
- ✧ 来源特征: 包括来访 URL, 来访 IP 等。
- ✧ 产品特征: 包括所访问的产品编号、产品类别、产品颜色、产品价格、产品利润、产品数量和特价等级等。

2 网站流量日志数据自定义采集

用户的行为会触发浏览器对被统计页面的一个 http 请求, 比如打开某网页。当网页被打开, 页面中的埋点 javascript 代码会被执行。

埋点: 在网页中预先加入小段 javascript 代码, 这个代码片段一般会动态创建一个 script 标签, 并将 src 属性指向一个单独的 js 文件, 此时这个单独的 js 文件 (图中绿色节点) 会被浏览器请求到并执行, 这个 js 往往就是真正的数据收集脚本。网站流量日志数据自定义采集分析图如图 2 所示。

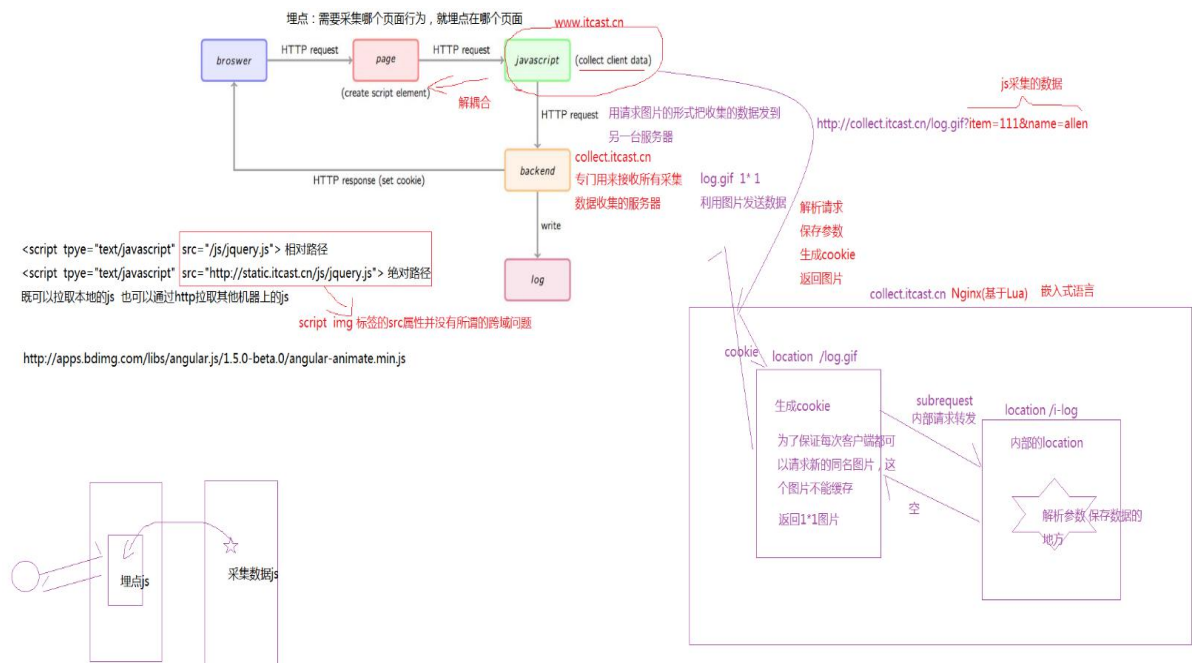
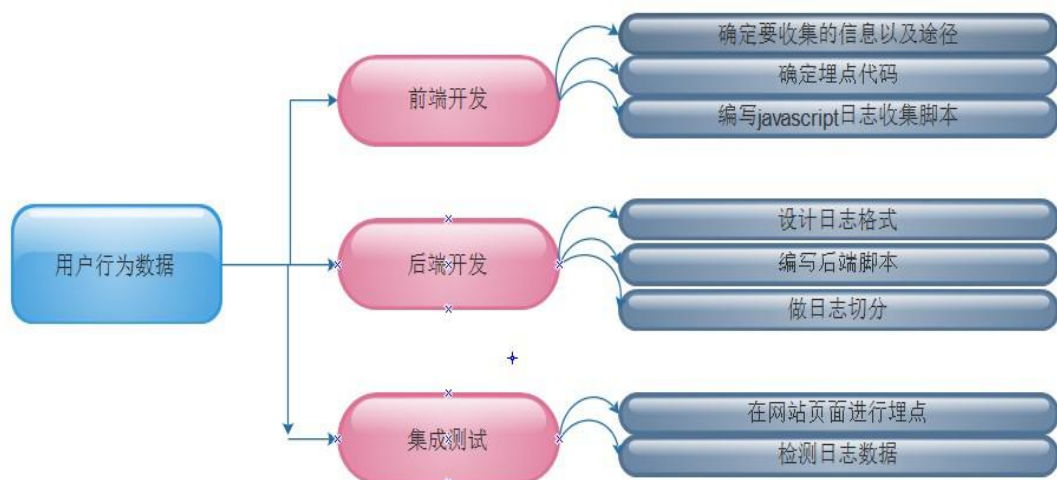


图2 网站流量日志数据自定义采集分析图

数据收集完成后 js 会请求一个后端的数据收集脚本（图中的 **backend**），这个脚本一般是一个伪装成图片的动态脚本程序，js 会将收集到的数据通过 **http** 参数的方式传递给后端脚本，后端脚本解析参数并按固定格式记录到访问日志，同时可能会在 **http** 响应中给客户端种植一些用于追踪的 **cookie**。

3. 设计实现

根据原理分析并结合 **Google Analytics**，想搭建一个自定义日志数据采集系统，要做以下几件事：



1. 确定收集信息

例如访问时间、ip、域名、URL、页面标题、浏览客户端、客户端语言、方可标识、状态码、发送内容量等信息

2. 确定埋点代码

在需要进行数据采集的关键点植入统计代码

js 自调用匿名函数

格式： `(function({})){`

第一对括号向脚本返回未命名的函数；后一对空括号立即执行返回的未命名函数，括号内为匿名函数的参数。

自调用匿名函数的好处是，避免重名，自调用匿名函数只会在运行时执行一次，一般用于初始化。

3. 前端收集脚本

数据收集脚本被请求后会被执行，一般需要做以下几件事：

- 1) 通过浏览器内置 javascript 对象收集信息，如页面 title（通过 `document.title`）、referrer（上一跳 url，通过 `document.referrer`）、用户显示器分辨率（通过 `windows.screen`）、cookie 信息（通过 `document.cookie`）等等一些信息。
- 2) 收集配置信息
- 3) 将上面两步收集的数据按预定义格式解析并拼接（get 请求参数）
- 4) 请求一个后端脚本，将信息放在 `http request` 参数中携带给后端脚本

4. 后端脚本

`log.gif` 是后端脚本，是一个伪装成 gif 图片的脚本。后端脚本一般需要完成以下几件事情：

- 1) 解析 `http` 请求参数得到信息。
- 2) 从 Web 服务器中获取一些客户端无法获取的信息，如访客 ip 等。
- 3) 将信息按格式写入 `log`。
- 4) 生成一副 1×1 的空 gif 图片作为响应内容并将响应头的 `Content-type` 设为 `image/gif`。
- 5) 在响应头中通过 `Set-cookie` 设置一些需要的 cookie 信息。之所以要

设置 cookie 是因为如果要跟踪唯一访客，通常做法是如果在请求时发现客户端没有指定的跟踪 cookie，则根据规则生成一个全局唯一的 cookie 并种植给用户，否则 Set-cookie 中放置获取到的跟踪 cookie 以保持同一用户 cookie 不变。这种做法虽然不是完美的（例如用户清掉 cookie 或更换浏览器会被认为是两个用户），但是目前被广泛使用的手段。

我们使用 nginx 的 access_log 做日志收集，不过有个问题就是 nginx 配置本身的逻辑表达能力有限，所以选用 OpenResty 做这个事情。

OpenResty 是一个基于 Nginx 扩展出的高性能应用开发平台，内部集成了诸多有用的模块，其中的核心是通过 ngx_lua 模块集成了 Lua，从而在 nginx 配置文件中可以通过 Lua 来表述业务。

Lua 是一种轻量小巧的脚本语言，用标准 C 语言编写并以源代码形式开放，其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。需要在 nginx 的配置文件中定义日志格式。

5. 日志格式

日志格式主要考虑日志分隔符，一般会有以下几种选择：

固定数量的字符、制表符分隔符、空格分隔符、其他一个或多个字符、特定的开始和结束文本。

6. 日志切分

日志收集系统访问日志时间一长文件变得很大，而且日志放在一个文件不便于管理。通常要按时间段将日志切分，例如每天或每小时切分一个日志。

Tips:

1. 如果埋点 js 涉及频繁更新，则需要让埋点 js 代码与页面解耦合，让更新采集数据的 js 不影响页面。
2. 为了高效地采集数据，同时不影响 web 服务器的运行压力，则需要把后端收集数据的服务单独部署。
3. ajax 是不能跨域请求的。一种通用的方法是 js 脚本创建一个 Image 对象，将 Image 对象的 src 属性指向后端脚本并携带参数，此时即实现了跨域请求后端。这也是后端脚本为什么通常伪装成 gif 文件的原因。

跨域问题的其他解决方案：

https://blog.csdn.net/qq_31617637/article/details/72955239

4. 环境部署

本次搭建两台服务器，一台用于正常的 web 服务器，一台用于收集数据，需要在该台服务器部署基于 lua 的 nginx。

1. 服务器中安装依赖

```
yum -y install gcc perl pcre-devel openssl openssl-devel
```

2. 上传 LuaJIT-2.0.4.tar.gz 并安装 LuaJIT

```
tar -zxvf LuaJIT-2.0.4.tar.gz -C /usr/local/src/
```

```
cd /usr/local/src/LuaJIT-2.0.4/
```

```
make && make install PREFIX=/usr/local/luajit
```

3. 设置 LuaJIT 环境变量

vi /etc/profile 添加如下内容：

```
export LUAJIT_LIB=/usr/local/luajit/lib
```

```
export LUAJIT_INC=/usr/local/luajit/include/luajit-2.0
```

```
source /etc/profile
```

4. 创建 modules 保存 nginx 的模块

```
mkdir -p /usr/local/nginx/modules
```

5. 上传依赖的模块

➤ set-misc-nginx-module-0.29.tar.gz

➤ lua-nginx-module-0.10.0.tar.gz

➤ ngx_devel_kit-0.2.19.tar.gz

➤ echo-nginx-module-0.58.tar.gz

6. 将依赖的模块直接解压到/usr/local/nginx/modules 目录

```
tar -zxvf lua-nginx-module-0.10.0.tar.gz -C /usr/local/nginx/modules/
```

```
tar -zxvf set-misc-nginx-module-0.29.tar.gz -C /usr/local/nginx/modules/
```

```
tar -zxvf ngx_devel_kit-0.2.19.tar.gz -C /usr/local/nginx/modules/
```

```
tar -zxvf echo-nginx-module-0.58.tar.gz -C /usr/local/nginx/modules/
```

7. 安装 openresty

下载对应的安装包：openresty-1.9.7.3.tar.gz

解压：tar -zxvf openresty-1.9.7.3.tar.gz -C /usr/local/src/

编译安装 openresty:

```
cd /usr/local/src/openresty-1.9.7.3/
```

执行命令：

```
./configure --prefix=/usr/local/openresty --with-luajit && make && make  
install
```

8. 安装 nginx

下载对应的安装包：nginx-1.8.1.tar.gz

解压：tar -zxvf nginx-1.8.1.tar.gz -C /usr/local/src/

9. 编译 nginx 并支持其他模块

进入到 nginx 的安装目录 cd /usr/local/src/nginx-1.8.1/

执行如下命令：

```
./configure --prefix=/usr/local/nginx \  
--with-ld-opt="-Wl,-rpath,/usr/local/luajit/lib" \  
--add-module=/usr/local/nginx/modules/ngx_devel_kit-0.2.19 \  
--add-module=/usr/local/nginx/modules/lua-nginx-module-0.10.0 \  
--add-module=/usr/local/nginx/modules/set-misc-nginx-module-0.29 \  
--add-module=/usr/local/nginx/modules/echo-nginx-module-0.58
```

make -j2 && make install

5. 自定义采集数据实现

方案一：基本功能实现

- a) 创建页面 index.html，添加埋点代码，放入 nginx 默认目录 nginx/html 下。
- b) 在默认目录 nginx/html 下添加一个数据采集脚本 ma.js。
- c) 修改 nginx 的配置文件，添加自定义相关业务逻辑。
- d) 启动 nginx

```
sbin/nginx -c conf/nginx.conf
```

- e) 通过浏览器访问 nginx
- f) 观察自定义日志采集文件是否有对应的内容输出

```
tail -f logs/user_defined.log
```

此时还可以观察 nginx 默认的输出日志文件

```
tail -f logs/access.log
```

停止 nginx:

```
sbin/nginx -s stop
```

问题：解决 nginx 在记录 post 数据时，中文字符转成 16 进制的问题

解决方法:

/usr/local/src/nginx-1.8.1/src/http/modules/ ngx_http_log_module.c

修改源码如下图所示

```

971 static uintptr_t
972 ngx_http_log_escape(u_char *dst, u_char *src, size_t size)
973 {
974     ngx_uint_t    n;
975     /* static u_char  hex[] = "0123456789ABCDEF"; */
976     static uint32_t escape[] = {
977         0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
978
979         /* ?>=< ;:98 7654 3210 /.-, +*)( '%$$ #"! */
980         0x00000004, /* 0000 0000 0000 0000 0000 0000 0000 0100 */
981
982         /* _^]\ [ZYX WVUT SRQP ONML KJIH GFED CBA@ */
983         0x10000000, /* 0001 0000 0000 0000 0000 0000 0000 0000 */
984
985         /* ~}| {zyx wvut srqp onml kjih gfed cba` */
986         0x80000000, /* 1000 0000 0000 0000 0000 0000 0000 0000 */
987
988         0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
989         0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
990         0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
991         0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
992     };
993
994     if (dst == NULL) {
995
996         /* find the number of the characters to be escaped */
997
998         n = 0;
999
1000         while (size) {
1001             if (escape[*src >> 5] & (1U << (*src & 0x1f))) {
1002                 n++;
1003             }
1004             src++;
1005             size--;
1006         }
1007
1008         return (uintptr_t) n;
1009     }
1010
1011     while (size) {
1012         /*if (escape[*src >> 5] & (1U << (*src & 0x1f))) {
1013             *dst++ = '\\';
1014             *dst++ = 'x';
1015             *dst++ = hex[*src >> 4];
1016             *dst++ = hex[*src & 0xf];
1017             src++;
1018
1019         } else {
1020             *dst++ = *src++;
1021         } */
1022         *dst++ = *src++;
1023         size--;
1024     }

```

然后重新编译，安装 nginx

```
./configure --prefix=/usr/local/nginx \
```

```
--with-ld-opt="-Wl,-rpath,/usr/local/luajit/lib" \
```

```
--add-module=/usr/local/nginx/modules/ngx_devel_kit-0.2.19 \
```

```
--add-module=/usr/local/nginx/modules/lua-nginx-module-0.10.0 \
--add-module=/usr/local/nginx/modules/set-misc-nginx-module-0.29 \
--add-module=/usr/local/nginx/modules/echo-nginx-module-0.58
```

```
make -j2 && make install
```

2.7 Hive

2.7.1 Hive 的基本简介

1) Hive 是什么?

Hive 是基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供类 SQL 查询功能。Hive 利用 HDFS 存储数据，利用 MapReduce 查询分析数据。

本质是将 SQL 转换为 MapReduce 程序，比直接用 MapReduce 开发效率更高。

2) Hive 的源数据存储?

通常是存储在关系数据库如 mysql/derby 中。Hive 将元数据存储在数据库中。Hive 中的元数据包括表的名字，表的列和分区及其属性，表的属性（是否为外部表等），表的数据所在目录等。

3) Hive 与传统 DB 的区别?

传统数据库：OLTP-->面向事务(Transaction) 操作型处理 就是关系型数据库：mysql oracle sqlserver db2 主要是支持业务，面向业务。

Hive：OLAP-->面向分析 (Analytical) 分析型处理 就是数据仓库 面对的是历史数据（历史数据中的一部分就来自于数据库） 开展分析

	Hive	RDBMS
查询语言	HQL	SQL
数据存储	HDFS	Raw Device or Local FS
执行	MapReduce	Excutor
执行延迟	高	低
处理数据规模	大	小
索引	0.8版本后加入位图索引	有复杂的索引

2.7.2 Hive 数仓开发的基本流程。

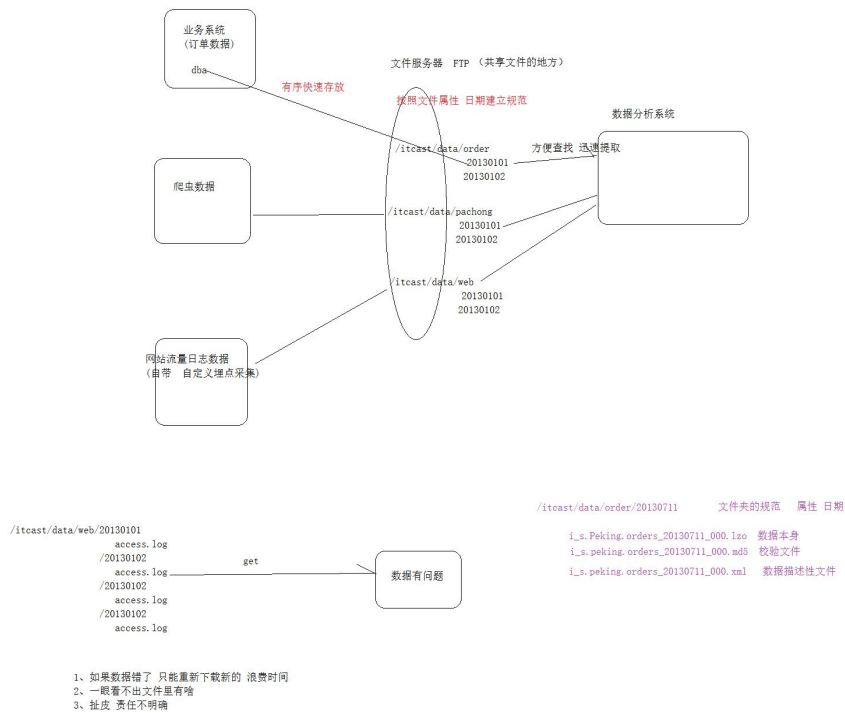
1) 从业务系统获取数据

Sqoop 导入数据库的数据：spoop 可以在 Hive 与传统的数据库间进行数据的传递，可以将一个关系型数据库中的数据导进到 Hadoop 的 HDFS 中，也可以将 HDFS 的数据导进到关系型数据库中。

Flume 采集文本数据：Flume 可以将各类型的文件进行采集，存放入 hdfs 中。

Ftp 文件服务器：从文件服务器上下载分析所需的源数据（增量数据、全量数据）。如下图所示：

规范:企业内部的规范



2) 数据存储

数据仓库分层 ods 层、dw 层、da 层：

源数据层 ODS :直接引用外围的数据 没有统一格式化的 不会直接应用使用 不利于分析

数据仓库层 DW :来自于 ODS 要经过 ETL 的过程 格式统一 数据规整 干净整洁

数据应用层 DA : 要去用 DW 层数据 真正的数据使用者。

数据集市：

数据集市也叫数据市场，数据集市就是满足特定的部门或者用户的需求，按照多维的方式进行存储，包括定义维度、需要计算的指标、维度的层次等，生成面向决策分析需求的数据立方体。

3) 写 sql 开发需求

4) 配置调度系统

5) 导出数据&展示

2.7.3 Hive sql 知识点。

1) DML&DDL

DML：数据操纵语言，负责对数据库对象运行数据访问工作的指令集

DELETE FROM table_name;

DDL：数据定义语言，用语定义和管理数据库中的对象。

TRUNCATE TABLE table_name;

2) 外表和内表

内部表：

create table student(Sno int,Sname string,Sex string,Sage int,Sdept string) row format delimited fields terminated by ',';

未被 external 修饰、表数据由 Hive 自身管理、表数据存储的位置是默认的 hive.metastore.warehouse.dir、删表会直接删除元数据及存储数据、对表的修改会将修改直接同步给元数据的是内部表

外部表：

create external table student_ext(Sno int,Sname string,Sex string,Sage int,Sdept string) row format delimited fields terminated by ',' location '/stu';

被 external 修饰的、表数据由 HDFS 管理、表数据的存储位置由自己指定、删表仅仅会删除元数据，HDFS 上的文件并不会被删除的是外部表。

3) 分区、分桶

分区（重点）

为了避免 select 查询时候的全表扫描问题 hive 提出了分区表 (partitioned by) 概念 给文件归类 打上标识（标识不能为表中已有的字段）

静态分区：

单分区建表 create table par_tab (name string, nation string) partitioned by (sex string) row format delimited fields terminated by ',';

加载 load data local inpath '/home/hadoop/files/par_tab.txt' into table par_tab partition (sex='man');

在新建分区表的时候, 系统会在 hive 数据仓库默认路径/user/hive/warehouse/下创建一个目录 (表名), 再创建目录的子目录 sex=man (分区名), 最后在分区名下存放实际的数据文件。

多分区建表 `create table par_tab_muilt (name string, nation string) partitioned by (sex string, dt string) row format delimited fields terminated by ',' ;`

加载: `load data local inpath '/home/hadoop/files/par_tab.txt' into table par_tab_muilt partition (sex='man', dt='2018-05-26');`

新建表的时候定义的分区顺序, 决定了文件目录顺序 (谁是父目录谁是子目录), 因为有了这个层级关系, **当我们查询所有 man 的时候, man 以下的所有日期下的数据都会被查出来。**如果只查询日期分区, 但父目录 sex=man 和 sex=woman 都有该日期的数据, 那么 Hive 会对输入路径进行修剪, **从而只扫描日期分区, 性别分区不作过滤 (即查询结果包含了所有性别)。**

动态分区

静态分区在插入的时候必须首先要知道有什么分区类型, 而且每个分区写一个 load data, 太烦人。使用动态分区可解决以上问题, 其可以根据查询得到的数据动态分配到分区里。其实动态分区与静态分区区别就是不指定分区目录, 由系统自己选择。

开启动态分区功能 `set hive.exec.dynamic.partition=true;`

加载数据 `insert overwrite table par_dnm partition (sex='man', dt) select name, nation, dt from par_tab;`

注:

1. 动态分区不允许主分区采用动态列而副分区采用静态列, 这样将导致所有的主分区都要创建副分区静态列所定义的分区。
2. 动态分区可以允许所有的分区列都是动态分区列, 但是要首先设置一个参数 `hive.exec.dynamic.partition.mode` 。

`set hive.exec.dynamic.partition.mode;`

`hive.exec.dynamic.partition.mode=strict`

它的**默认值是 strict, 即不允许分区列全部是动态的**, 这是为了防止用户有可能原意是只在子分区内进行动态建分区, 但是由于疏忽忘记为主分区列指定值了, 这将导致一个 dml 语句在短时间内创建大量的新的分区 (对应大量新的文件夹), 对系统性能带来影响。修改后即可。

`set hive.exec.dynamic.partition.mode=nostrict;`

总结:

- 1、分区表是为了减少查询时候的全表扫描而出现。

2、分区表的现象就是在表的文件夹下多了一个文件夹 而文件夹的名字就是分区 字段=分区值

3、分区字段的值在查询的时候 会显示出来 但是并不代表结构化数据中有这个字段 分区字段是一个虚拟字段 只是用来标识文件

方便用户查询的时候 根据这个标识进行过滤 从而减少了全局扫描

4、分区表的数据通过 load data 的方式加载 加载的时候要指定分区的值 (这个分区的值就是这批数据文件夹名字的值)

5、关于分区字段一定不会是表中存在的字段 如果是直接编译报错 体会什么叫给数据打标识

分桶

1. 为了提高 join 查询时的效率 减少了笛卡尔积的数量, 分桶表出现。

clustered by (字段) into num_buckets buckets (如: clustered by (sex) into 2 buckets)

2. 分桶的功能默认不开启 需要自己手动开启 set hive.enforce.bucketing = true;

3. 分成几桶 也需要自己指定 set mapreduce.job.reduces=N;

4. 分桶表导入数据的方式: insert+select insert 数据来自于后 select 查询的结果

```
insert overwrite table stu_buck
select * from student cluster by(Sno);
```

4) UDF

1. 什么是 UDF

当 Hive 提供的内置函数无法满足你的业务处理需要时, 此时就可以考虑使用用户自定义函数 (UDF: user-defined function) 。

2. UDF 开发步骤示例

新建 JAVA maven 项目

添加 hive-exec-1.2.1.jar 和 hadoop-common-2.7.4.jar 依赖 (见参考资料)

1、 写一个 java 类, 继承 UDF, 并重载 evaluate 方法

```
package cn.itcast.bigdata.udf
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;
public class Lower extends UDF{
    public Text evaluate(Text s){
        if(s==null){return null;}
        return new Text(s.toString().toLowerCase());
    }
}
```

2、打成 jar 包上传到服务器

3、将 jar 包添加到 hive 的 classpath

hive>add JAR /home/hadoop/udf.jar;

4、 创建临时函数与开发好的 java class 关联

```
create temporary function tolowercase as
'cn.itcast.bigdata.udf.ToProvince';
```

5、 即可在 hql 中使用自定义的函数 tolowercase ip

```
Select tolowercase(name),age from t_test;
```

2.8 Hive 面试题

2.8.1 Hive 常见

1) hive 的使用，内外部表的区别，分区作用，UDF 和 Hive 优化

(1)hive 使用：仓库、工具

(2)hive 内部表：加载数据到 hive 所在的 hdfs 目录，删除时，元数据和数据文件都删除

外部表：不加载数据到 hive 所在的 hdfs 目录，删除时，只删除表结构。

(3)分区作用：防止数据倾斜

(4)UDF 函数：用户自定义的函数（主要解决格式，计算问题），需要继承 UDF 类

java 代码实现

```
class TestUDFHive extends UDF {  
    public String evalute(String str){  
        try{  
            return "hello"+str  
        }catch(Exception e){  
            return str+"error"  
        }  
    }  
}
```

(5)Hive 优化：看做 mapreduce 处理

排序优化：sort by 效率高于 order by

分区：使用静态分区 (statu_date="20160516",location="beijin")，每个分区对应 hdfs 上

的一个目录，减少 job 和 task 数量：使用表链接操作，解决 groupby 数据倾斜问题：设

置 hive.groupby.skewindata=true，那么 hive 会自动负载均衡，小文件合并成大文件：表

连接操作，使用 UDF 或 UDAF 函数：

<http://www.cnblogs.com/ggjucheng/archive/2013/02/01/2888819.html>

2) 使用 Hive 或者自定义 MR 实现如下逻辑

product_no	lac_id	moment	start_time	user_id	county_id	staytime	city_id
13429100031	22554	8	2013-03-11 08:55:19.151754088	571	571	282	571
13429100082	22540	8	2013-03-11 08:58:20.152622488	571	571	270	571
13429100082	22691	8	2013-03-11 08:56:37.149593624	571	571	103	571
13429100087	22705	8	2013-03-11 08:56:51.139539816	571	571	220	571
13429100087	22540	8	2013-03-11 08:55:45.150276800	571	571	66	571
13429100082	22540	8	2013-03-11 08:55:38.140225200	571	571	133	571
13429100140	26642	9	2013-03-11 09:02:19.151754088	571	571	18	571
13429100082	22691	8	2013-03-11 08:57:32.151754088	571	571	287	571
13429100189	22558	8	2013-03-11 08:56:24.139539816	571	571	48	571
13429100349	22503	8	2013-03-11 08:54:30.152622440	571	571	211	571

字段解释：

product_no： 用户手机号；

lac_id： 用户所在基站；

start_time： 用户在此基站的开始时间；

staytime： 用户在此基站的逗留时间。

需求描述：

根据 lac_id 和 start_time 知道用户当时的位置，根据 staytime 知道用户各个基站的逗留时长。根据轨迹合并连续基站的 staytime。最终得到每一个用户按时间排序在每一个基站驻留时长。

期望输出举例：

13429100082	22540	8	2013-03-11 08:58:20.152622488	571	571	270	571
13429100082	22691	8	2013-03-11 08:56:37.149593624	571	571	390	571
13429100082	22540	8	2013-03-11 08:55:38.140225200	571	571	133	571
13429100087	22705	8	2013-03-11 08:56:51.139539816	571	571	220	571
13429100087	22540	8	2013-03-11 08:55:45.150276800	571	571	66	571

3) 你们的数据是用什么导入到数据库的? 导入到什么数据库?

4) 你们业务数据量多大? 有多少行数据? (面试了三家, 都问这个问题)

5) 你们写 hive 的 hql 语句, 大概有多少条?

6) hive 跟 hbase 的区别是?

共同点:

1.hbase 与 hive 都是架构在 hadoop 之上的。都是用 hadoop 作为底层存储

区别:

2.Hive 是建立在 Hadoop 之上为了减少 MapReduce jobs 编写工作的批处理系统, HBase 是为了支持弥补 Hadoop 对实时操作的缺陷的项目。

3.想象你在操作 RMDB 数据库, 如果是全表扫描, 就用 Hive+Hadoop,如果是索引访问, 就用 HBase+Hadoop。

4.Hive query 就是 MapReduce jobs 可以从 5 分钟到数小时不止, HBase 是非常高效的, 肯定比 Hive 高效的多。

5.Hive 本身不存储和计算数据, 它完全依赖于 HDFS 和 MapReduce, Hive 中的表纯逻辑。

6.hive 借用 hadoop 的 MapReduce 来完成一些 hive 中的命令的执行

7.hbase 是物理表, 不是逻辑表, 提供一个超大的内存 hash 表, 搜索引擎通过它来存储索引, 方便查询操作。

8.hbase 是列存储。

9.hdfs 作为底层存储, hdfs 是存放文件的系统, 而 Hbase 负责组织文件。

10.hive 需要用到 hdfs 存储文件, 需要用到 MapReduce 计算框架。

- 7) 你们处理数据是直接读数据库的数据还是读文本数据?
- 8) 你们提交的 job 任务大概有多少个? 这些 job 执行完大概用多少时间?
- 9) reduce 后输出的数据量有多大?
- 10) 你自己写过 udf 函数么? 写了哪些? 作用是什么?
- 11) 你在项目中主要的工作任务是?
- 12) hive 底层与数据库交互原理
- 13) 使用 Hive 进行手机流量统计

问题导读

1. hive 实现统计的查询语句是什么?
2. 生产环境中为什么建议使用外部表?
3. hadoop mapreduce 创建类 DataWritable 的作用是什么?
4. 为什么创建 类 类 DataWritable ?
5. 如何实现统计手机流量?
6. 对比 hive 与 与 mapreduce 统计手机流量的区别?

很多公司在使用 hive 对数据进行处理。hive 是 hadoop 家族成员,是一种解析 like sql 语句的框架。它封装了常用 MapReduce 任务,让你像执行 sql 一样操作存储在 HDFS 的表。

hive 的表分为两种,内表和外表。Hive 创建内部表时,会将数据移动到数据仓库指向的路径;若创建外部表,仅记录数据所在的路径,不对数据的位置做任何改变。在删除表的时候,内部表的元数据和数据会被一起删除,而外部表只删除元数据,不删除数据。这样外部表相对来说更加安全些,数据组织也更加灵活,方便共享源数据。

Hive 的内外表,还有一个 Partition 的分区知识点,用于避免全表扫描,快速检索。后期的文章会提到。

原始数据

1363157985066	13726230503	00-FD-07-A4-72-B8:CMCC	120.196.100.82	i02.c.aliimg.com		
24	27	2481	24681	200		
1363157995052	13826544101	5C-0E-8B-C7-F1-E0:CMCC	120.197.40.4		4	0
264	0	200				
1363157991076	13926435656	20-10-7A-28-CC-0A:CMCC	120.196.100.99		2	4
132	1512	200				
1363154400022	13926251106	5C-0E-8B-8B-B1-50:CMCC	120.197.40.4		4	0
240	0	200				
1363157993044	18211575961	94-71-AC-CD-E6-18:CMCC-EASY	120.196.100.99	iface.qiyi.com		
瑞.?.细..	15	2	1527	2106	200	
1363157995074	84138413	5C-0E-8B-8C-E8-20:7DaysInn	120.197.40.4	122.72.52.12		
20	16	4116	1432	200		
1363157993055	13560439658	C4-17-FE-BA-DE-D9:CMCC	120.196.100.99		18	1
5	1116	954	200			
1363157995033	15920133257	5C-0E-8B-C7-BA-20:CMCC	120.197.40.4	sug.so.360.cn	洪℃.	
渝..	20	20	156	2936	200	

操作步骤

1. #配置好 Hive 之后，使用 `hive` 命令启动 `hive` 框架。`hive` 启动属于懒加载模式，会比较慢
2. `hive;`
3. #使用 `show databases` 命令查看当前数据库信息
4. `hive> show databases;`
5. OK
6. `default`
7. `hive`
8. `Time taken: 3.389 seconds`
9. #使用 `use hive` 命令，使用指定的数据库 `hive` 数据库是我之前创建的
10. `use hive;`
11. #创建表，这里是创建内表。内表加载 `hdfs` 上的数据，会将被加载文件中的内容剪切走。
12. #外表没有这个问题，所以在实际的生产环境中，建议使用外表。
13. `create table ll(reportTime string,msisdn string,apmac string,acmac string,host string,siteType string,upPackNum bigint,downPackNum bigint,upPayLoad bigint,downPayLoad bigint,httpStatus string)row format delimited fields terminated by '\t';`
14. #加载数据，这里是从 `hdfs` 加载数据，也可用 `linux` 下加载数据 需要 `local` 关键字
15. `load data inpath '/HTTP_20130313143750.dat' into table ll;`
16. #数据加载完毕之后，`hdfs` 的
17. #执行 `hive` 的 `like sql` 语句,对数据进行统计
18. `select msisdn,sum(uppcknum),sum(downpacknum),sum(uppayload),sum(downpayload) from ll group by msisdn;`

执行结果如下


```

1.  hive> select msisdn,sum(uppcknum),sum(downpacknum),sum(uppayload),sum(downpayload)
    from ll group by msisdn;

2.  Total MapReduce jobs = 1

3.  Launching Job 1 out of 1

4.  Number of reduce tasks not specified. Estimated from input data size: 1

5.  In order to change the average load for a reducer (in bytes):

6.    set hive.exec.reducers.bytes.per.reducer=<number>

7.  In order to limit the maximum number of reducers:

8.    set hive.exec.reducers.max=<number>

9.  In order to set a constant number of reducers:

10.   set mapred.reduce.tasks=<number>

11. Starting Job = job_201307160252_0006, Tracking URL =
    http://hadoop0:50030/jobdetails.jsp?jobid=job_201307160252_0006

12. Kill Command = /usr/local/hadoop/libexec/bin/hadoop
    job -Dmapred.job.tracker=hadoop0:9001 -kill job_201307160252_0006

13. Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1

14. 2013-07-17 19:51:42,599 Stage-1 map = 0%, reduce = 0%

15. 2013-07-17 19:52:40,474 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 48.5 sec

16. 2013-07-17 19:52:41,690 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 48.5 sec

17. 2013-07-17 19:52:42,693 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 48.5 sec

18. 2013-07-17 19:52:43,698 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 48.5 sec

19. 2013-07-17 19:52:44,702 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 48.5 sec

20. 2013-07-17 19:52:45,707 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 48.5 sec

21. 2013-07-17 19:52:46,712 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 48.5 sec

22. 2013-07-17 19:52:47,715 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 48.5 sec

23. 2013-07-17 19:52:48,721 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 48.5 sec

24. 2013-07-17 19:52:49,758 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 48.5 sec

25. 2013-07-17 19:52:50,763 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 48.5 sec

26. 2013-07-17 19:52:51,772 Stage-1 map = 100%, reduce = 100%, Cumulative CPU 50.0 sec

27. 2013-07-17 19:52:52,775 Stage-1 map = 100%, reduce = 100%, Cumulative CPU 50.0 sec

28. 2013-07-17 19:52:53,779 Stage-1 map = 100%, reduce = 100%, Cumulative CPU 50.0 sec

29. MapReduce Total cumulative CPU time: 50 seconds 0 msec

30. Ended Job = job_201307160252_0006

31. MapReduce Jobs Launched:

32. Job 0: Map: 1 Reduce: 1 Cumulative CPU: 50.0 sec HDFS Read: 2787075 HDFS Write: 16518
    SUCCESS

33. Total MapReduce CPU Time Spent: 50 seconds 0 msec

34. OK

35. 13402169727      171      108      11286      130230

36. 13415807477      2067      1683      169668      1994181

37. 13416127574      1501      1094      161963      802756

38. 13416171820      113       99      10630      32120

39. 13417106524      160       128      18688      13088

40. 13418002498      240       256      22136      86896

41. 13418090588      456       351      98934      67470

42. 13418117364      264       152      29436      49966

43. 13418173218      37680     48348     2261286     73159722

44. 13418666750      22432     26482     1395648     39735552

45. 13420637670       20        20      1480       1480

46. ....

47. Time taken: 75.24 seconds

```

14) 表大概有 2T 左右，对表数据转换

TRLOG:

CREATE TABLE TRLOG

(PLATFORM string,

USER_ID int,

CLICK_TIME string,

CLICK_URL string)

row format delimited

fields terminated by '\t';

数据

PLATFORM	USER_ID	CLICK_TIME	CLICK_URL
WEB	12332321	2013-03-21 13:48:31.324	/home/
WEB	12332321	2013-03-21 13:48:32.954	/selectcat/er/
WEB	12332321	2013-03-21 13:48:46.365	/er/viewad/12.html
WEB	12332321	2013-03-21 13:48:53.651	/er/viewad/13.html
WEB	12332321	2013-03-21 13:49:13.435	/er/viewad/24.html
WEB	12332321	2013-03-21 13:49:35.876	/selectcat/che/
WEB	12332321	2013-03-21 13:49:56.398	/che/viewad/93.html
WEB	12332321	2013-03-21 13:50:03.143	/che/viewad/10.html
WEB	12332321	2013-03-21 13:50:34.265	/home/
WAP	32483923	2013-03-21 23:58:41.123	/m/home/
WAP	32483923	2013-03-21 23:59:16.123	/m/selectcat/fang/
WAP	32483923	2013-03-21 23:59:45.123	/m/fang/33.html
WAP	32483923	2013-03-22 00:00:23.984	/m/fang/54.html
WAP	32483923	2013-03-22 00:00:54.043	/m/selectcat/er/
WAP	32483923	2013-03-22 00:01:16.576	/m/er/49.html
.....

需要把上述数据处理为如下结构的表 ALLOG:

CREATE TABLE ALLOG

(PLATFORM string,

USER_ID int,

SEQ int,

FROM_URL string,

TO_URL string)

row format delimited

fields terminated by '\t';

整理后的数据结构：

PLATFORM	USER_ID	SEQ	FROM_URL	TO_URL
WEB	12332321	1	NULL	/home/
WEB	12332321	2	/home/	/selectcat/er/
WEB	12332321	3	/selectcat/er/	/er/viewad/12.html
WEB	12332321	4	/er/viewad/12.html	/er/viewad/13.html
WEB	12332321	5	/er/viewad/13.html	/er/viewad/24.html
WEB	12332321	6	/er/viewad/24.html	/selectcat/che/
WEB	12332321	7	/selectcat/che/	/che/viewad/93.html
WEB	12332321	8	/che/viewad/93.html	/che/viewad/10.html
WEB	12332321	9	/che/viewad/10.html	/home/
WAP	32483923	1	NULL	/m/home/
WAP	32483923	2	/m/home/	/m/selectcat/fang/
WAP	32483923	3	/m/selectcat/fang/	/m/fang/33.html
WAP	32483923	4	/m/fang/33.html	/m/fang/54.html
WAP	32483923	5	/m/fang/54.html	/m/selectcat/er/
WAP	32483923	6	/m/selectcat/er/	/m/er/49.html
*****	*****	*****	*****	

PLATFORM 和 USER_ID 还是代表平台和用户 ID；SEQ 字段代表用户按时间排序后的访问顺序，FROM_URL 和 TO_URL 分别代表用户从哪一页跳转到哪一页。对于某个平台上某个用户的第一条访问记录，其 FROM_URL 是 NULL（空值）。

面试官说需要用两种办法做出来：

1、实现一个能加速上述处理过程的 Hive Generic UDF，并给出使用此 UDF 实现 ETL 过程的 Hive SQL

给你个 JAVA 写的 RowNumber 方法

```
1.
2. public class RowNumber extends org.apache.hadoop.hive.ql.exec.UDF {
3.
4.     private static int MAX_VALUE = 50;
5.     private static String comparedColumn[] = new String[MAX_VALUE];
6.     private static int rowNum = 1;
7.
8.     public int evaluate(Object... args) {
9.         String columnValue[] = new String[args.length];
10.        for (int i = 0; i < args.length; i++)
11.            columnValue[i] = args[i].toString();
12.        if (rowNum == 1)
13.        {
14.
15.            for (int i = 0; i < columnValue.length; i++)
16.                comparedColumn[i] = columnValue[i];
17.        }
18.
19.        for (int i = 0; i < columnValue.length; i++)
20.        {
21.
22.            if (!comparedColumn[i].equals(columnValue[i]))
23.            {
24.                for (int j = 0; j < columnValue.length; j++)
25.                {
26.                    comparedColumn[j] = columnValue[j];
27.                }
28.                rowNum = 1;
29.                return rowNum++;
30.            }
31.        }
32.        return rowNum++;
33.    }
34. }
```

把这个 JAVA 打包，编译成 JAR 包，比如 RowNumber.jar。然后放到 HIVE 的机器上在 HIVE SHELL 里执行下面两条语句：

```
1. add jar /root/RowNumber.jar;
2. #把 RowNumber.jar 加载到 HIVE 的 CLASSPATH 中
3. create temporary function row_number as 'RowNumber';
4. #在 HIVE 里创建一个新函数，叫 row_number，引用的 CLASS 就是 JAVA 代码里的 RowNumber
```

提示成功后，执行下面这条 HIVE SQL

```
2. #INSERT OVERWRITE TABLE ALLOG 如果要写入 ALLOG 表，可以把注释去掉
3. SELECT t1.platform,t1.user_id,row_number(t1.user_id)seq,t2.click_url
FROM_URL,t1.click_url TO_URL FROM
4. (select *,row_number(user_id)seq from trlog)t1
5. LEFT OUTER JOIN
6. (select *,row_number(user_id)seq from trlog)t2
7. on t1.user_id = t2.user_id and t1.seq = t2.seq + 1;
```

第一题中的 RN 貌似是 HIVE 转译 SQL 的 BUG，你可以把外层的 ROW_NUMBER 去掉，用 T1 的 SEQ，就能发现问题了。

2、实现基于纯 Hive SQL 的 ETL 过程，从 TRLOG 表生成 ALLOG 表；（结果是一套 SQL）

```

2. INSERT OVERWRITE TABLE ALLOG
3. SELECT t1.platform,t1.user_id,t1.seq,t2.click_url FROM_URL,t1.click_url TO_URL FROM
4. (SELECT platform,user_id,click_time,click_url,count(1) seq FROM (SELECT a.*,b.click_time
click_time1,b.click_url click_url2 FROM trlog a left outer join trlog b on a.user_id =
b.user_id)t WHERE click_time>=click_time1 GROUP BY
platform,user_id,click_time,click_url)t1
5. LEFT OUTER JOIN
6. (SELECT platform,user_id,click_time,click_url,count(1) seq FROM (SELECT a.*,b.click_time
click_time1,b.click_url click_url2 FROM trlog a left outer join trlog b on a.user_id =
b.user_id)t WHERE click_time>=click_time1 GROUP BY
platform,user_id,click_time,click_url )t2
7. on t1.user_id = t2.user_id and t1.seq = t2.seq + 1;

```

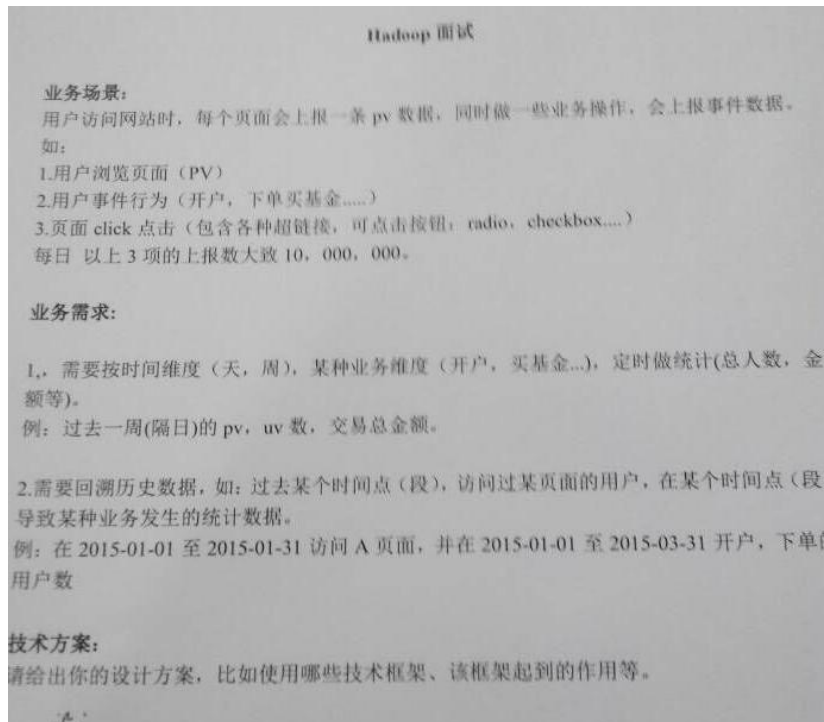
15) hive 有哪些保存元数据的方式，个有什么特点。

- 1、 内存数据库 derby，安装小，但是数据存在内存，不稳定
- 2、 mysql 数据库，数据存储模式可以自己设置，持久化好，查看方便

16) 生产环境中为什么建议使用外部表？

- 1、因为外部表不会加载数据到 hive，减少数据传输、数据还能共享。
- 2、hive 不会修改数据，所以无需担心数据的损坏
- 3、删除表时，只删除表结构、不删除数据。

17) 如何用 hive 分析业务数据，将数据导入到 hive 中



1、用 hive 分析业务数据即可

2、将数据导入到 hive 中

sql 的设计思路：多表关联

1、找到所有在 2015-01-01 到 2015-01-31 时间内访问 A 页面的用户

2、在这些用户中删选在 2015-01-01 到 2015-03-31 下单的用户

3、统计总数

18) 你们数据库怎么导入 hive 的，有没有出现问题

在导入 hive 的时候，如果数据库中有 blob 或者 text 字段会报错，解决方案在 sqoop 笔记中

19) Hive 你们用的是外部表还是内部表，有没有写过 UDF, hive 的版本外部表和内部表的区别

20) 一个 Hadoop 环境，整合了 hbase 和 hive，是否有必要给 HDFS 和 hbase 都分别配置压缩策略？请给出对压缩策略的建议

hdfs 在存储的时候不会将数据进行压缩，如果想进行压缩，我们可以在向 hdfs 上传数据的时候进行压缩。

1、采用压缩流


```
//压缩文件

public static void compress(String codecClassName) throws
Exception{

    Class<?> codecClass = Class.forName(codecClassName);

    Configuration conf = new Configuration();

    FileSystem fs = FileSystem.get(conf);

    CompressionCodec codec =

    (CompressionCodec)ReflectionUtils.newInstance(codecClass, conf);

    //指定压缩文件路径

    FSDataOutputStream outputStream = fs.create(new

    Path("/user/hadoop/text.gz"));

    //指定要被压缩的文件路径

    FSDataInputStream in = fs.open(new

    Path("/user/hadoop/aa.txt"));

    //创建压缩输出流

    CompressionOutputStream out =

    codec.createOutputStream(outputStream);

    IOUtils.copyBytes(in, out, conf);

    IOUtils.closeStream(in);

    IOUtils.closeStream(out);

}
```

2、 采用序列化文件

```

public void testSeqWrite() throws Exception {

Configuration conf = new Configuration();// 创建配置信息

conf.set("fs.default.name", "hdfs://master:9000");//hdfs 默认路径

conf.set("hadoop.job.ugi", "hadoop,hadoop");// 用户和组信息

String uriin = "hdfs://master:9000/ceshi2/";// 文件路径

FileSystem fs = FileSystem.get(URI.create(uriin),conf);// 创建 filesystem

Path path = new Path("hdfs://master:9000/ceshi3/test.seq");// 文件名

IntWritable k = new IntWritable();// key, 相当于 intText v = new Text(); value, 相当于 String

SequenceFile.Writer w = SequenceFile.createWriter(fs,conf, path,k.getClass(),

    v.getClass());// 创建 writer

for (int i = 1; i < 100; i++) { // 循环添加

k.set(i);

v.set("abcd");

w.append(k, v);

}

w.close();

IOUtils.closeStream(w);// 关闭的时候 flush

fs.close();

}

```

hbase 为列存数据库，本身存在压缩机制，所以无需设计。

21) 简述 Hive 中的虚拟列作用是什么?使用他的注意事项

Hive 提供了三个虚拟列：

INPUT_FILE_NAME

BLOCK_OFFSET_INSIDE_FILE

ROW_OFFSET_INSIDE_BLOCK

但 ROW_OFFSET_INSIDE_BLOCK 默认是不可用的，需要设置 hive.exec.rowoffset 为 true 才可以。可以用来排查有问题的输入数据。

INPUT_FILE_NAME, mapper 任务的输出文件名。

BLOCK_OFFSET_INSIDE_FILE, 当前全局文件的偏移量。对于块压缩文件，就是当前块的文件偏移量，即当前块的第一个字节在文件中的偏移量。

```
hive> SELECT INPUT_FILE_NAME, BLOCK_OFFSET_INSIDE_FILE, line
```

```
> FROM hive_text WHERE line LIKE '%hive%' LIMIT 2;
```

```
har://file/user/hive/warehouse/hive_text/folder=docs/
```

```
data.har/user/hive/warehouse/hive_text/folder=docs/README.txt 2243
```

```
har://file/user/hive/warehouse/hive_text/folder=docs/
```

```
data.har/user/hive/warehouse/hive_text/folder=docs/README.txt 3646
```

基站逗留时间

1 使用 Hive 或者自定义 MR 实现如下逻辑

product_no	lac_id	moment	start_time	user_id	county_id	staytime	city_id
13429100031	22554	8	2013-03-11 08:55:19.151754088	571	571	282	571
13429100082	22540	8	2013-03-11 08:58:20.152622488	571	571	270	571
13429100082	22691	8	2013-03-11 08:56:37.149593624	571	571	103	571
13429100087	22705	8	2013-03-11 08:56:51.139539816	571	571	220	571
13429100087	22540	8	2013-03-11 08:55:45.150276800	571	571	66	571
13429100082	22540	8	2013-03-11 08:55:38.140225200	571	571	133	571
13429100140	26642	9	2013-03-11 09:02:19.151754088	571	571	18	571
13429100082	22691	8	2013-03-11 08:57:32.151754088	571	571	287	571

字段解释：

product_no: 用户手机号；

lac_id: 用户所在基站；

start_time: 用户在此基站的开始时间；

staytime: 用户在此基站的逗留时间。

需求

根据 lac_id 和 start_time 知道用户当时的位置，根据 staytime 知道用户各个基站的逗留时长。根据轨迹合并连续基站的 staytime。

最终得到每一个用户按时间排序在每一个基站驻留时长

期望

期望输出举例：

13429100082	22540	8	2013-03-11 08:58:20.152622488	571	571	270	571
13429100082	22691	8	2013-03-11 08:56:37.149593624	571	571	103	571
13429100082	22540	8	2013-03-11 08:55:38.140225200	571	571	133	571
13429100087	22705	8	2013-03-11 08:56:51.139539816	571	571	220	571
13429100087	22540	8	2013-03-11 08:55:45.150276800	571	571	66	571

思路：

将数据导入 hive 表中，查询时，用电话号码和时间排序即可！

22) 你们数据库怎么导入 hive 的,有没有出现问题

使用 sqoop 导入，我们公司的数据库中设计了 text 字段，导致导入的时候出现了缓存不够的情况（见云笔记），开始解决起来感觉很棘手，后来查看了 sqoop 的文档，加上了 limit 属性，解决了

23) Redis,传统数据库,hbase,hive 每个之间的区别(问的非常细)

Redis 是缓存，围绕着内存和缓存说

Hbase 是列式数据库，存在 hdfs 上，围绕着数据量来说

Hive 是数据仓库，是用来分析数据的，不是增删改查数据的。

24) Hive 你们用的是外部表还是内部表,有没有写过 UDF,hive 的版本外部表, udf, udaf 等, hive 版本为 1.0

25) hive partition 分区分区表，动态分区

26) insert into 和 override write 区别？

insert into：将某一张表中的数据写到另一张表中

override write：覆盖之前的内容。

27) 假如一个分区的数据主部错误怎么通过 hivesql 删除 hdfs

```
alter table ptable drop partition (daytime='20140911',city='bj');
```

元数据，数据文件都删除，但目录 daytime= 20140911 还在

28) 请简述一下 Hadoop/MapReduce, Spark, Storm, Hive 的特点及适用场景?

Hadoop : 是一种分布式系统基础架构当处理海量数据的程序, 开始要求高可靠、高扩展、高效、低容错、低成本的场景

MapReduce: MapReduce 是一种编程模型, 用于大规模数据集 (大于 1TB) 的并行运算。MapReduce 的典型应用场景中, 目前日志分析用的比较多, 还有做搜索的索引, 机器学习算法包 mahout 也是之一, 当然它能做的东西还有很多, 比如数据掘、信息提取。

Spark: 拥有 Hadoop MapReduce 所具有的优点; 但不同于 MapReduce 的是 Job 中间输出结果可以保存在内存中, 从而不再需要读写 HDFS, 因此 Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 MapReduce 的算法。数据过于繁杂, 并且需要让计算通过迭代, 并在内存中, 极大地提高效率的场景

Storm: 一个分布式实时计算系统, Storm 是一个任务并行连续计算引擎。Storm 本身并不

典型在 Hadoop 集群上运行, 它使用 Apache ZooKeeper 的和自己的主/从工作进程, 协调拓扑, 主机和工作状态, 保证信息的语义。无论如何, Storm 必定还是可以从 HDFS 文件消费或者从文件写入到 HDFS。

Hive: 基于 Hadoop 的一个数据仓库工具, 可以将结构化的数据文件映射为一张数据库表, 并提供简单的 sql 查询功能, 可以将 sql 语句转换为 MapReduce 任务进行运行。应用场景: 十分适合数据仓库的统计分析。

Hbase: 应用场景: 数据量太大, 以至于传统 RDBMS 无法胜任、联机业务功能开发、离线数据分析 (数据仓库),

29) Hive 的条件判断有几种?

hive 的条件判断 (if、coalesce、case)

30) 请适用 hive 写出查询某网站日志中方位多页面 a 和页面 b 的用户数量的语句:

```
Select count(user) from urla a , urlb b where a.url = b.url ;
```

31) Orcal、hive、mysql 分页

Orcal 分页

第一种: 显示从 5 到 10 的员工

1、table01 按照工资降序排列出所有员工

```
select * from emp order by sal desc
```

2、table02 在员工表 table01 中设定 rownum, rownum<=5

```
select rownum no, e.*  
  
from (select * from emp order by sal desc) e  
  
where rownum <= 5
```

3、在 table02 中限定条件 rownum>3

```
select *  
  
from (select rownum no, e.*  
  
from (select * from emp order by sal desc) e  
  
where rownum <= 10) where no>5
```

第二种：显示从 5 到 10 的员工

```
select *  
  
from  
  
(select rownum no, e.* from (select * from emp order by sal desc) e)  
  
where  
  
no >= 5 and no <= 10
```

hive 中的分页

第一种：

一、借助唯一标识字段

如果分页的表有唯一标识的字段，可以通过这个字段来实现分页：

- 获取第一页数据：

注：同时需要记录这 10 条中最大的 id 为 preId，作为下一页的条件。

```
select * from table order by id asc limit 10;
```

- 获取第二页数据：

注：同时保存数据中最大的 id 替换 preld。

```
select * from table where id >preld order by id asc limit 10;
```

二、使用 row number() 函数

如果分页的表没有唯一标识的字段，可以通过 row number()函数来实现分页。

- 首先使用 row number()函数来给这个表做个递增的唯一标识：

```
create table newtable as select row number(1) as id ,* from table;
```

- 通过 row number 函数给表加了唯一标识之后, 就可以利用 第一个方法 来进行分页。

MySQL 中分页实现

我们来贴例子吧！

```
mysql> select pname from product;
```

```
+-----+
```

```
| pname |
```

```
+-----+
```

```
| 产品 1 |
```

```
| 产品 2 |
```

```
| 产品三 |
```

```
+-----+
```

```
3 rows in set (0.00 sec)
```

这个地方是说，从 product 中选出所有的 pname 来，一共有三条记录。

MySQL 中的分页非常简单，我们可以使用 limit

比如：


```
mysql> select pname from product limit 0,2;
```

```
+-----+
```

```
| pname |
```

```
+-----+
```

```
| 产品 1 |
```

```
| 产品 2 |
```

```
+-----+
```

```
2 rows in set (0.00 sec)
```

Limit 用法如下：

第一个参数是指要开始的地方，第二个参数是指每页显示多少条数据；注意：第一页用 0

表示。

I Mysql 分页：

```
select * from tableName where 条件 limit 当前页码*页面容量
```

```
-1, 页面容量
```

32) Hive 中的排序方法有哪些

order by

order by 会对输入做全局排序，因此只有一个 reducer（多个 reducer 无法保证全局有序）

只有一个 reducer，会导致当输入规模较大时，需要较长的计算时间。

Sql 代码

1. set hive.mapred.mode=nonstrict; (default value / 默认值)

2. set hive.mapred.mode=strict;

order by 和数据库中的 Order by 功能一致，按照某一项 & 几项 排序输出。与数据库中 order by 的区别在于在 hive.mapred.mode = strict 模式下 必须指定 limit 否则执行会报错。

Sql 代码

1. hive> select * from test order by id;

Java 代码

1. FAILED: Error in semantic analysis: 1:28 In strict mode, if ORDER BY is specified, LIMIT must also be specified. Error encountered near token 'id'

原因：在 order by 状态下所有数据会到一台服务器进行 reduce 操作也即只有一个 reduce，如果在数据量大的情况下会出现无法输出结果的情况，如果进行 limit n，那只有 n * map number 条记录而已。只有一个 reduce 也可以处理过来。

sort by

sort by 不是全局排序，其在数据进入 reducer 前完成排序。因此，如果用 sort by 进行排序，并且设置 mapred.reduce.tasks>1，则 sort by 只保证每个 reducer 的输出有序，不保证全局有序。

sort by 不受 hive.mapred.mode 是否为 strict ,nostrict 的影响

sort by 的数据只能保证在同一 reduce 中的数据可以按指定字段排序。使用 sort by 你可以指定执行的 reduce 个数 (set mapred.reduce.tasks=<number>)。对输出的数据再执行归并排序，即可以得到全部结果。注意：可以用 limit 子句大大减少数据量。使用 limit n 后，传输到 reduce 端（单机）的数据记录数

就减少到 $n \times$ (map 个数)。否则由于数据过大可能出不了结果。

distributed by

按照指定的字段对数据进行划分到不同的输出 reduce / 文件中。

```
insert overwrite local directory '/home/hadoop/out' select * from test order by  
name distributed by length(name);
```

此方法会根据 name 的长度划分到不同的 reduce 中，最终输出到不同的文件中。length 是内建函数，也可以指定其他的函数或这使用自定义函数。

Cluster By

cluster by 除了具有 distributed by 的功能外还兼具 sort by 的功能。但是排序只能是倒序排序，不能指定排序规则为 asc 或者 desc

33) hive 和 hbase 方法：项目中采集的哪些指标，

34) 你们提交的 job 任务大概有多少个？这些 job 执行完大概用多少时间？
(面试了三家，都问这个问题)

每天六百个，2, 3 个小时。

35) 你在项目中主要的工作任务是？

利用 hive 分析数据

36) 一个网络商城

- 1.1 天大概产生多少 G 的日志？大概有多少条日志记录（在不清洗的情况下）？一个网络商城 1 天大概产生多少 G 的日志？ 4tb
- 2.大概有多少条日志记录（在不清洗的情况下）？ 7-8 百万条
- 3.日访问量大概有多少个？ 百万
- 4.注册数大概多少？ 不清楚 几十万吧
- 5.我们的日志是不是除了 apache 的访问日志是不是还有其他的日志？
- 6.假设我们有其他的日志是不是可以对这个日志有其他的业务分析？这些业务分析有什么

37) 你认为用 java,streaming,pipe 方式开发 map/reduce ,各有哪些优点就 用过 java 和 hiveQL。

Java 写 mapreduce 可以实现复杂的逻辑, 如果需求简单, 则显得繁琐。HiveQL 基本都是针对 hive 中的表数据进行编写, 但对复杂的逻辑很难进行实现。写起来简单。

38) hive 有哪些方式保存元数据, 各有哪些优点

三种: 内存数据库 derby, 挺小, 不常用。本地 mysql。常用远程端 mysql。不常用。网上找了下专业名称: single user mode..multi user mode...remote user mode

39) hive 内部表和外部表的区别

Hive 创建内部表时, 会将数据移动到数据仓库指向的路径; 若创建外部表, 仅记录数据所在的路径, 不对数据的位置做任何改变。在删除表的时候, 内部表的元数据和数据会被一起删除, 而**外部表**只删除元数据, 不删除数据。这样外部表相对来说更加安全些, 数据组织也更加灵活, 方便共享源数据。

40) hive 底层与数据库交互原理

hive 有一套自己的 sql 解析引擎, 称为 metastore, 存储在 mysql 或者 derby 数据库中, 可以将 sql 语句转化为 mapreducejob 任务执行。

41) sqoop 在导入数据到 mysql 中, 如何不重复导入数据, 如果存在数据问题, sqoop 如何处理?

42) pig , latin , hive 语法有什么不同

43) hive 如何调优

底层是 MapReduce, 所以又可以说是 MapReduce 优化。

1. 小文件都合并成大文件
2. Reducer 数量在代码中介于节点数*reduceTask 的最大数量的 0.95 倍到 1.75 倍
3. 写一个 UDF 函数, 在建表的时候制定好分区
4. 配置文件中, 打开在 map 端的合并
5. 在库表设计的时候, 尽量考虑 rowkey 和 columnfamily 的特性
6. 进行 hbase 集群的调优: 见 hbase 调优

- 44) hive 如何控制权限
- 45) hive 能像关系型数据库那样建多个库吗?
- 46) 假设公司要建一个数据中心, 你会如何处理?
- 47) Hive 中的 metastore 用来做什么的?

metastore 是一套映射工具, 将 sql 语句转换成对应的 job 任务去进行执行。

- 48) hive 中的压缩格式 RCFile、TextFile、SequenceFile 各有什么区别? 以上 3 种格式一样大的文件哪个占用空间大小
- 49) 你们的 hive 处理数据能达到的指标是多少?
- 50) Hive 查询案例

3. 请按如下需求完成查询 (Hive查询)

用户注册信息表, 表名: user_log

id	name	info	create_time	log_date
001	AA	aaa	1483200062	2017-01-01
002	BB	bbb	1483552862	2017-01-05
003	CC	ccc	1483322462	2017-01-02
004	DD	ddd	1483275662	2017-01-01
005	EE	eee	1483153262	2016-12-31
006	FF	fff	1482976862	2016-12-29

用户登录信息表, 表名: login_log

id	login_time	log_date
001	1483200062	2017-01-01
001	1483236062	2017-01-01
005	1483239662	2017-01-01
001	1483275662	2017-01-01
003	1483275662	2017-01-01
003	1483408862	2017-01-03
003	1483448462	2017-01-03
002	1483639262	2017-01-06
005	1483650062	2017-01-06
002	1484100062	2017-01-11

数据表如上, 详细说明:

1. 数据表存在于hive数据仓库之中
2. create_time为注册时间、login_time为登录时间
3. log_date为partition字段
4. 服务器时间与北京时间存在-2小时的时差

4. 有a、b两个文件，每个文件中各存放50亿个URL，每个URL占64个字节，最大可使用内存为4G，如何找出a、b两个文件中同时存在的URL？

查询需求如下：

1. 服务器时间1月1日的活跃用户（扣除当日新增用户），在1月6日仍有登陆的用户总数。请按需写出hql语句，并尝试优化（计算引擎为MapReduce），且给出优化后的hql执行时需要启动多少个mapreduce的job。

2. 不考虑时差，上述两张表如果使用 `user_log LEFT OUTER JOIN login_log`，关联条件及要求如下，请写出对应结果集。

1. `user_log.id = login_log.id`

2. 取每个id的最后登录时间

3. 结果字段：id、name、最后登录时间

1. 请使用shell写出如下几个操作对应的命令，任意能实现的命令即可

1. 过滤hadoop fs -ls xxx 后的结果集，仅保留所有大小为0的文件（假定第5列为文件大小）

2. 按字典序排序a.txt文件并删除相同的行

3. 删除文件a.txt中的空白行

4. 输出文件a.txt中包含字符串“playcrab”的所有行到文件b.txt

5. 对比两个文件a.txt、b.txt

6. 定时任务每日的11点~12点之间，每5分钟执行一次a.sh脚本

2. Hive on MapReduce调优，通常需要注意哪些？

51) 给定 a、b 两个文件，各存放 50 亿个 url，每个 url 各占 64 字节，内存限制是 4G，让你找出 a、b 文件共同的 url？

方案 1：可以估计每个文件安的大小为 $50G \times 64 = 320G$ ，远远大于内存限制的 4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。遍历文件 a，对每个 url 求取，然后根据所取得的值将 url 分别存储到 1000 个小文件（记为 a_i ）中。这样每个小文件的大约为 300M。s 遍历文件 b，采取和 a 相同的方式将 url 分别存储到 1000 个小文件（记为 b_i ）。这样处理后，所有可能相同的 url 都在对应的小文件（ a_i, b_i ）中，不对应的小文件不可能有相同的 url。然后我们只要求出 1000 对小文件中相同的 url 即可。s 求每对小文件中相同的 url 时，可以把其中一个 small 文件的 url 存储到 hash_set 中。然后遍历另一个小文件的每个 url，看其是否在刚才构建的 hash_set 中，如果是，那么就是共同的 url，存到文件里面就可以了。

方案 2：如果允许有一定的错误率，可以使用 Bloom filter，4G 内存大概可以表示 340 亿 bit。将其中一个文件中的 url 使用 Bloom filter 映射为这 340 亿 bit，然后挨个读取另外一个文件的 url，检查是否与 Bloom filter，如果是，那么该 url 应该是共同的 url（注意会有一定的错误率）。

52) 有 10 个文件，每个文件 1G，每个文件的每一行存放的都是用户的 query，每个文件的 query 都可能重复。要求你按照 query 的频度排序。

方案 1：

顺序读取 10 个文件，按照 $\text{hash}(\text{query})\%10$ 的结果将 query 写入到另外 10 个文件（记为 a_i ）中。这样新生成的文件每个的大小大约也 1G（假设 hash 函数是随机的）。找一台内存存在 2G 左右的机器，依次对用 $\text{hash_map}(\text{query}, \text{query_count})$ 来统计每个 query 出现的次数。利用快速/堆/归并排序按照出现次数进行排序。将排序好的 query 和对应的 query_cout 输出到文件中。这样得到了 10 个排好序的文件（记为 b_i ）。对这 10 个文件进行归并排序（内排序与外排序相结合）。

方案 2：

一般 query 的总量是有限的，只是重复的次数比较多而已，可能对于所有的 query，一次性就可以加入到内存了。这样，我们就可以采用 trie 树/hash_map 等直接来统计每个 query 出现的次数，然后按出现次数做快速/堆/归并排序就可以了。

方案 3：

与方案 1 类似，但在做完 hash，分成多个文件后，可以交给多个文件来处理，采用分布式的架构来处理（比如 MapReduce），最后再进行合并。

53) 有一个 1G 大小的一个文件，里面每一行是一个词，词的大小不超过 16 字节，内存限制大小是 1M。返回频数最高的 100 个词。方案 1：顺序读文件中，对于每个词 x，取 $\text{hash}(x)\%5000$ ，然后按照该值存到 5000 个小文件（记为 a_i ）中。这样每个文件大概是 200k 左右。如果其中的有的文件超过了 1M 大小，还可以按照类似的方法继续往下分，知道分解得到的小文件的大小都不超过 1M。对每个小文件，统计每个文件中出现的词以及相应的频率（可以采用 trie 树/hash_map 等），并取出出现频率最大的 100 个词（可以用含 100 个结点的最小堆），并把 100 词及相应的频率存入文件，这样又得到了 5000 个文件。下一步就是把这 5000 个文件进行归并（类似与归并排序）的过程了。

54) 海量日志数据，提取出某日访问百度次数最多的那个 IP。

方案 1：首先是这一天，并且是访问百度的日志中的 IP 取出来，逐个写入到一个大文件中。注意到 IP 是 32 位的，最多有个 IP。同样可以采用映射的方法，比如模 1000，把整个大文件映射为 1000 个小文件，再找出每个小文件中出现频率最大的 IP（可以采用 hash_map 进行频率统计，然后再找出频率最大的几个）及相应的频率。然后再在这 1000 个最大的 IP 中，找出那个频率最大的 IP，即为所求。

55) 在 2.5 亿个整数中找出不重复的整数，注，内存不足以容纳这 2.5 亿个整数。

方案 1：采用 2-Bitmap（每个数分配 2bit，00 表示不存在，01 表示出现一次，10 表示多次，11 无意义）进行，共需内存 $2^{32} * 2 \text{ bit} = 1 \text{ GB}$ 内存，还可以接受。然后扫描这 2.5 亿个整数，查看 Bitmap 中相对应位，如果是 00 变 01，01 变 10，10 保持不变。扫描完后，查看 bitmap，把对应位是 01 的整数输出即可。

方案 2：也可采用与第 1 题类似的方法，进行划分小文件的方法。然后在小文件中找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。

56) 海量数据分布在 100 台电脑中，想个办法高效统计出这批数据的 TOP10。

方案 1：

s 在每台电脑上求出 TOP10，可以采用包含 10 个元素的堆完成（TOP10 小，用最大堆，TOP10 大，用最小堆）。比如求 TOP10 大，我们首先取前 10 个元素调整成最小堆，如果发现，然后扫描后面的数据，并与堆顶元素比较，如果比堆顶元素大，那么用该元素替换堆顶，然后再调整为最小堆。最后堆中的元素就是 TOP10 大。

s 求出每台电脑上的 TOP10 后，然后把这 100 台电脑上的 TOP10 组合起来，共 1000 个数据，再利用上面类似的方法求出 TOP10 就可以了。

57) 怎么在海量数据中找出重复次数最多的一个？

方案 1：先做 hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。然后找出上一步求出的数据中重复次数最多的一个就是所求（具体参考前面的题）。

58) 上千万或上亿数据（有重复），统计其中出现次数最多的前 N 个数据。

方案 1：上千万或上亿的数据，现在的机器的内存应该能存下。所以考虑采用 hash_map/搜索二叉树/红黑树等来进行统计次数。然后就是取出前 N 个出现次数最多的数据了，可以用第 6 题提到的堆机制完成。

59) 1000 万字符串，其中有些是重复的，需要把重复的全部去掉，保留没有重复的字符串。

请怎么设计和实现？

方案 1：这题用 trie 树比较合适，hash_map 也应该能行。

60) 一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前 10 个词，

请给出思想，给出时间复杂度分析。

方案 1：这题是考虑时间效率。用 trie 树统计每个词出现的次数，时间复杂度是 $O(n \cdot l_e)$ (l_e

表示单词的平准长度)。然后是找出出现最频繁的前 10 个词，可以用堆来实现，前面的题中已经讲到了，时间复杂度是 $O(n \cdot \lg 10)$ 。所以总的时间复杂度，是 $O(n \cdot l_e)$ 与 $O(n \cdot \lg 10)$ 中较大的哪一个。

61) 一个文本文件，找出前 10 个经常出现的词，但这次文件比较长，说是上亿行或十亿行，

总之无法一次读入内存，问最优解。

方案 1：首先根据用 hash 并求模，将文件分解为多个小文件，对于单个文件利用上题的方法求出每个文件中 10 个最常出现的词。然后再进行归并处理，找出最终的 10 个最常出现的词。

62) 100w 个数中找出最大的 100 个数。

方案 1：在前面的题中，我们已经提到了，用一个含 100 个元素的最小堆完成。复杂度为 $O(100w \cdot \lg 100)$ 。

方案 2：采用快速排序的思想，每次分割之后只考虑比轴大的一部分，知道比轴大的一部分在比 100 多的时候，采用传统排序算法排序，取前 100 个。复杂度为 $O(100w \cdot 100)$ 。方案

3：采用局部淘汰法。选取前 100 个元素，并排序，记为序列 L。然后一次扫描剩余的元素 x，与排好序的 100 个元素中最小的元素比，如果比这个最小的要大，那么把这个最小的元素删除，并把 x 利用插入排序的思想，插入到序列 L 中。依次循环，知道扫描了所有的元素。复杂度为 $O(100w \cdot 100)$ 。

63) ABC 三个 hive 表，每个表中只有一列 int 类型且列名相同，求三个表中互不重复的数，比如表明：T_a T_b T_c 每个列都叫做 id (int 类型)

64) 系统设计

1.有200亿条数据，每条数据的大小在1K~1M不等，每条数据有一个唯一的
id_int64的

id。请设计一个读取数据系统，能根据id获取数据。要求：

A.内存有限制，16G

B.尽可能利用内存资源

C.尽可能高效的获取数据

D.可以利用磁盘，磁盘容量不受限制

2.C2C网站的商品子系统，包括的关系数据有 分类、属性、商品。

一个商品只能属于一个分类，不同的分类有不同的属性（多个），每个属性有多个

以第三列为 key 进行 map 分发，所有数据分发到一个桶，利用框架的特点进行排序。此方法可行，但一个 reduce 处理 100G 数据，效率极低。

以第三列为 key 进行 map 分发，按照 key 的大小分发到不同的桶，使得所有数据按桶号有序，而桶内的排序利用框架完成。此方法速度快，但如果分桶不均，会变得很慢。

3)

仍然使用 b 方法，但通过先验知识得到第三列的分布，从而据此均匀分桶。分布的获得可以利用历史数据，也可以用当前数据，在排序前先进行一次 map-reduce，得到分布。

主要考察点(被面试官越能抓住这些点，则说明对题目把握越到位)

1) 是否理解 map-reduce 处理的整个过程。2) 是否能利用每个 reduce 的排序功能进行分布式排序。3) 是否理解 partition 的意义和重要性，并提出解决分桶不均的办法。

什么是数据倾斜？Hadoop 框架的特性决定最怕数据倾斜

65) 什么是数据倾斜？hadoop 框架的特性决定最怕数据倾斜

- 由于数据分布不均匀，造成数据大量的集中到一点，造成数据热点。

节点间数据分布不均衡，会造成 map 端每个 map 任务的工作量不同，即 map 端数据倾斜。
Map-reduce，把相同 key 提交给同一个 reduce，如果 key 不均衡就会造成不同的 reduce 的工作量不同。

以京东首页活动为例，曝光率大的是大活动，曝光率小的是小活动：

假如 reduce1 处理的是小活动，reduce2 处理大活动，reduce2 干的活比其他 reduce 多很多，会出现其他 reduce 执行完毕了，reduce2 还在缓慢执行。

症状：map 阶段快，reduce 阶段非常慢；
某些 map 很快，某些 map 很慢；
某些 reduce 很快，某些 reduce 奇慢。

如下情况：

A、数据在节点上分布不均匀

B、join 时 on 关键词中个别值量很大（如 null 值）

C、count(distinct)，在数据量大的情况下，容易数据倾斜，因为 count(distinct)是按 group by 字段分组，按 distinct 字段排序。

其中 A 无法避免。B 见后边的 Join 章节。C 语法上有时无法避免

66) 如何解决数据倾斜? 实际上没有办法避免, 这里的解决只是个别情况起效

有数据倾斜的时候进行负载均衡

`set hive.groupby.skewindata = false;`

当选项设定为 `true`, 生成的查询计划会有两个 MR Job。第一个 MR Job 中, Map 的输出结果会随机分布到 Reduce 中, 每个 Reduce 做部分聚合操作, 并输出结果, 这样处理的结果是相同的 Group By Key 有可能被分发到不同的 Reduce 中, 从而达到负载均衡的目的; 第二个 MR Job 再根据预处理的数据结果按照 Group By Key 分布到 Reduce 中 (这个过程可以保证相同的 Group By Key 被分布到同一个 Reduce 中), 最后完成最终的聚合操作。

67) Hive 中 Join MapJoin Group by 的特点

按照 join 的 key 进行分发, 而在 join 左边的表的部分数据会首先读入内存, 如果左边表的 key 相对分散(或少, 分散的意思是相同 key 的数据量小), join 任务执行会比较快; 而如果左边的表 key 比较集中 (key 的大小量级分化), 而这张表的数据量很大, 那么数据倾斜就会比较严重。

Map 阶段同一 Key 数据分发给同一个 reduce。

Join 原则:

小表 Join 大表, 原因是在 Join 操作的 Reduce 阶段 (不是 Map 阶段), 位于 Join 左边的表的内容会被加载进内存, 将条目少的表放在左边, 可以有效减少发生内存溢出的几率。

多个表关联时, 最好分拆成小段, 避免大 sql (无法控制中间 Job)。

多表 Join on 条件相同时合并为一个 Map-Reduce, 做 OUTER JOIN 的时候也是一样, 查看执行计划 explain

比如查询, 3 个表关联:

```
select pt.page_id, count(t.url) PV
from rpt_page_type pt
join
(select url_page_id, url from trackinfo where ds='2013-10-11') t
on pt.page_id=t.url_page_id
join
(select page_id from rpt_page_kpi_new where ds='2013-10-11') r
on t.url_page_id=r.page_id
group by pt.page_id;
```

比较 2 个表关联:

```
select pt.page_id, count(t.url) PV
from rpt_page_type pt
join
(select url_page_id, url from trackinfo where ds='2013-10-11') t
on pt.page_id=t.url_page_id
group by pt.page_id;
```

利用这个特性, 可以把相同 join on 条件的放在一个 job 处理。

如果 Join 的条件不相同，如：

```
INSERT OVERWRITE TABLE page_pv
select pt.page_id,count(t.url) PV
from rpt_page_type pt
join
(select url_page_id,url,province_id from trackinfo where ds='2013-10-11') t
on pt.page_id=t.url_page_id
join
(select page_id,province_id from rpt_page_kpi_new where ds='2013-10-11') r
on t.province_id=r.province_id
group by pt.page_id;
```

大表 Join 大表

访户未登录时，日志中 userid 是空，在用 user_id 进行 hash 分桶的时候，会将日志中 userid 为空的数据分到一起，导致了过大空 key 造成倾斜。

解决办法：

把空值的 key 变成一个字符串加上随机数，把倾斜的数据分到不同的 reduce 上，由于 null 值关联不上，处理后并不影响最终结果

案例：

End_user 5000 万，纬度表
Trackinfo 每日 2 亿，按日增量表

原写法：

```
select u.id,t.url,t.track_time
from end_user u
join
(select end_user_id,url,track_time from trackinfo where ds='2013-12-01') t
on u.id=t.end_user_id limit 2;
```

调整为:

```
select u.id,t.url,t.track_time
  from end_user u
 join
 (select case when end_user_id='null' or end_user_id is null
            then cast (concat('00000000_',floor(rand()*1000000)) as bigint)
            else end_user_id end end_user_id ,
      url,track_time
   from trackinfo where ds='2013-12-01') t
 on u.id=t.end_user_id limit 2;
```

此例子只是为了说明原理。

当前这个场景不需要这么麻烦,如下即可:

```
select u.id,t.url,t.track_time
  from end_user u

 join
 (select end_user_id,
      url,track_time
   from trackinfo where ds='2013-12-01' and end_user_id is not null) t
 on u.id=t.end_user_id limit 2;
```

例子 2:

```
Select count(distinct end_user_id)
From trackinfo where ds='2013-12-01' ;
```

建议改为:

```
Select count(distinct end_user_id)+1
From trackinfo where ds='2013-12-01' and end_user_id is not null;
```


MapJoin

Join 操作在 Map 阶段完成, 如果需要的数据在 Map 的过程中可以访问到则不再需要 Reduce。

小表关联一个超大表时, 容易发生数据倾斜, 可以用 MapJoin 把小表全部加载到内存存在 map 端进行 join, 避免 reducer 处理。

如:

```
INSERT OVERWRITE TABLE page_pv
select /*+ MAPJOIN(pt) */
      pt.page_id,count(t.url) PV
from rpt_page_type pt
join
(select url_page_id,url from trackinfo where ds='2013-10-11') t
on pt.page_id=t.url_page_id;
```

如果是小表, 能否自动选择 Mapjoin?

set **hive.auto.convert.join**=true; 默认为 false

该参数为 true 时, Hive 自动对左边的表统计量, 如果是小表就加入内存, 即对小表使用 Map join

大表小表的阈值:

```
hive> set hive.mapjoin.smalltable.filesize;
```

```
hive.mapjoin.smalltable.filesize=25000000
```

默认值是 25mb

```
hive.mapjoin.cache.numrows
```

- 说明: mapjoin 存在内存里的数据量
- 默认值: 25000

```
hive.mapjoin.followby.gby.localtask.max.memory.usage
```

- 说明: map join 做 group by 操作时, 可以使用多大的内存来存储数据, 如果数据太大, 则不会保存在内存里
- 默认值: 0.55

```
hive.mapjoin.localtask.max.memory.usage
```

- 说明: 本地任务可以使用内存的百分比
- 默认值: 0.90

笛卡尔积

尽量避免笛卡尔积, join 的时候不加 on 条件, 或者无效的 on 条件, Hive 只能使用 1 个 reducer 来完成笛卡尔积

On 中支持 udf 函数

Group By

Map 端部分聚合:

并不是所有的聚合操作都需要在 Reduce 端完成, 很多聚合操作都可以先在 Map 端进行部分聚合, 最后在 Reduce 端得出最终结果。

基于 Hash

参数包括:

`hive.map.aggr = true` 是否在 Map 端进行聚合, 默认为 True

`hive.groupby.mapaggr.checkinterval = 100000` 在 Map 端进行聚合操作的条目数目

默认情况下, Map 阶段同一 Key 数据分发给一个 reduce, 当一个 key 数据过大时就倾斜了

有数据倾斜的时候进行负载均衡

`hive.groupby.skewindata = false;`

当选项设定为 true, 生成的查询计划会有两个 MR Job。第一个 MR Job 中, Map 的输出结果会随机分布到 Reduce 中, 每个 Reduce 做部分聚合操作, 并输出结果, 这样处理的结果是相同的 Group By Key 有可能被分发到不同的 Reduce 中, 从而达到负载均衡的目的; 第二个 MR Job 再根据预处理的数据结果按照 Group By Key 分布到 Reduce 中 (这个过程可以保证相同的 Group By Key 被分布到同一个 Reduce 中), 最后完成最终的聚合操作。

Count(distinct) 容易倾斜, 当该字段存在大量值为 NULL 或空的记录

解决思路

count distinct 时, 将值为空的数据在 where 里过滤调, 在最后结果中加 1

如:

```
count(distinct end_user_id) as user_num
```

修正为

```
count(distinct end_user_id)+1 user_num
```

```
where end_user_id is not null and query <> "
```

68) Hive 你们用的是外部表还是内部表,有没有写过 UDF,hive 的版本

69) Hive 语句实现 WordCount

1.建表

2.分组 (group by) 统计 wordcount

```
select word,count(1) from table1 group by word;
```


70) 共同朋友

mapred 找共同朋友，数据格式如下

```
1. A B C D E F
2. B A C D E
3. C A B E
4. D A B E
5. E A B C D
6. F A
```

第一字母表示本人，其他是他的朋友，找出有共同朋友的人，和共同朋友是谁

思路：例如 A，他的朋友是 B\C\D\E\F，那么 BC 的共同朋友就是 A。所以将 BC 作为 key，将 A 作为 value，在 map 端输出即可！其他的朋友循环处理。

```
import java.io.IOException;

2. import java.util.Set;

3. import java.util.StringTokenizer;

4. import java.util.TreeSet;

5.

6. import org.apache.hadoop.conf.Configuration;

7. import org.apache.hadoop.fs.Path;

8. import org.apache.hadoop.io.Text;

9. import org.apache.hadoop.mapreduce.Job;

10. import org.apache.hadoop.mapreduce.Mapper;

11. import org.apache.hadoop.mapreduce.Reducer;

12. import org.apache.hadoop.mapreduce.Mapper.Context;

13. import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
```

```
14. import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
15. import org.apache.hadoop.util.GenericOptionsParser;
16.
17. public class FindFriend {
18.
19. public static class ChangeMapper extends Mapper<Object, Text, Text,
Text>{
20. @Override
21. public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
22. StringTokenizer itr = new StringTokenizer(value.toString());
23. Text owner = new Text();
24. Set<String> set = new TreeSet<String>();
25. owner.set(itr.nextToken());
26. while (itr.hasMoreTokens()) {
27. set.add(itr.nextToken());
28.}
29. String[] friends = new String[set.size()];
30. friends = set.toArray(friends);
31.
32. for(int i=0;i<friends.length;i++){
33. for(int j=i+1;j<friends.length;j++){
```

```
34. String outputkey = friends[i]+friends[j];
35. context.write(new Text(outputkey),owner);
36. }
37. }
38. }
39. }
40.
41. public static class FindReducer extends Reducer<Text,Text,Text,Text>
{
42. public void reduce(Text key, Iterable<Text> values,
43. Context context) throws IOException,
InterruptedException {
44. String commonfriends = "";
45. for (Text val : values) {
46. if(commonfriends == ""){
47. commonfriends = val.toString();
48. }else{
49. commonfriends =
commonfriends+"."+val.toString();
50. }
51. }
52. context.write(key, new
```

```
Text(commonfriends));

53.                }

54.        }

55.

56.

57.        public static void main(String[] args) throws IOException,
58.        InterruptedException, ClassNotFoundException {

59.

60.                Configuration conf = new Configuration();

61.                String[] otherArgs = new GenericOptionsParser(conf,
args).getRemainingArgs();

62.                if (otherArgs.length < 2) {

63.                        System.err.println("args error");

64.                        System.exit(2);

65.                }

66.                Job job = new Job(conf, "word count");

67.                job.setJarByClass(FindFriend.class);

68.                job.setMapperClass(ChangeMapper.class);

69.                job.setCombinerClass(FindReducer.class);

70.                job.setReducerClass(FindReducer.class);

71.                job.setOutputKeyClass(Text.class);

72.                job.setOutputValueClass(Text.class);
```

```
73.         for (int i = 0; i < otherArgs.length - 1; ++i) {
74.             FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
75.         }
76.         FileOutputFormat.setOutputPath(job,
77.             new Path(otherArgs[otherArgs.length - 1]));
78.         System.exit(job.waitForCompletion(true) ? 0 : 1);
79.
80.     }
81.
82. }
```

结果：

1. AB	E:C:D
2. AC	E:B
3. AD	B:E
4. AE	C:B:D
5. BC	A:E
6. BD	A:E
7. BE	C:D:A
8. BF	A
9. CD	E:A:B
10. CE	A:B
11. CF	A

```
12. DE      B:A
```

```
13. DF      A
```

```
14. EF      A
```

71) 简要描述数据库中的 null, 说出 null 在 hive 底层如何存储, 并解释 `select a.* from t1 a left outer join t2 b on a.id=b.id where b.id is null;` 语句的含义。

null 在 hive 底层默认是用 "\N" 来存储的, 所以在 sqoop 到 mysql 之前需要将 null 的数据加工成 其他字符, 否则 sqoop 提示错误。

72) 写出 hive 中 `split`、`coalesce` 及 `collect_list` 函数的用法 (可举例)

`split` 将字符串转化为数组

`coalesce(T v1,T v2,...)` 返回参数中的第一个非空值; 如果所有值都为 null, 那么返回 null

`collect_list` 列出该字段所有的值, 不去重 `select collect_list(id) from table;`

73) 写出 `text.txt` 文件放入 hive 中 `test` 表 '2017-12-12' 分区的语句, `test` 的分区字段是 `l_date`。

```
load data local inpath '/a.txt' overwrite into table test partition(xx='xx')
```

74) hive 中有一个表 `test`, 结构是 `create table arrays (x array<string>)`, 且有:

```
hive>select * from test;
```

```
OK
```

```
["a","b"]
```

```
["c","d","e"]
```

(1) 请写出 `select explode (x) as xx from test;` 语句执行后的结果。

(2) 请写出 `select 'xx', sp from test lateral view explode (split (concat_ws ('', '1', '2'), '')) a as sp;` 语句执行后的结果。

75) 解释一下什么是数据倾斜, 并说明在 hive 中如何避免数据倾斜。

参数调节:

```
hive.map.aggr=true
```

hive.groupby.skewindata=true

有数据倾斜的时候进行负载均衡，当选项设定为 true,生成的查询计划会有两个 MR Job。第一个 MR Job 中，Map 的输出结果集会随机分布到 Reduce 中，每个 Reduce 做部分聚合操作，并输出结果，这样处理的结果是相同 Group By Key 有可能被分发到不同的 Reduce 中，从而达到负载均衡的目的；第二个 MR Job 在根据预处理的数据结果按照 Group By Key 分布到 Reduce 中(这个过程可以保证相同的 Group By Key 被分布到同一个 Reduce 中)，最后完成最终的聚合操作。

2>SQL 语句调节：

1)选用 join key 分布最均匀的表作为驱动表。做好列裁剪和 filter 操作，以达到两表 join 的时候，数据量相对变小的效果。

2)大小表 Join：使用 map join 让小的维度表（1000 条以下的记录条数）先进内存。在 Map 端完成 Reduce。

3)大表 Join 大表：把空值的 Key 变成一个字符串加上一个随机数，把倾斜的数据分到不同的 reduce 上，由于 null 值关联不上，处理后并不影响最终的结果。

4)count distinct 大量相同特殊值：count distinct 时，将值为空的情况单独处理，如果是计算 count distinct，可以不用处理，直接过滤，在做后结果中加 1。如果还有其他计算，需要进行 group by，可以先将值为空的记录单独处理，再和其他计算结果进行 union。

76) 学生表 (students)，有字段学号 (id)，姓名 (name)，性别 (sex) 等字段。课程表 (classes)，有字段课程标识 (id)，课程名称 (class_name)。学生选课分数表 (sc)，有字段学生表的学号 (student_id)，课程标识 (class_id)，分数 (scores)。

(1) 查询平均成绩大于 60 分的，且性别为“男”的同学的学号和平均成绩。

(2) x 代表分数， $x \geq 80$ 表示优秀， $60 \leq x < 80$ 为及格， $x < 60$ 为不及格等三个等级。请查询所有学生的姓名，课程名称及分数等级。

77) Hive 是什么，以及适用场景，Hive 与 mysql 关系，Hive 与 MapReduce 关系，hive 与 Hbase 的区别是？

基于 hadoop 的数据仓库工具，以 hdfs 方式存储，hive 是面向分析的。Mysql 关系型的、Hbase 非关系都是面向存储的。Hive 运行的就是 mapreduce

78) Hive 中创建表有哪几种方式，其区别是什么？

内部表：删除时，删除的时表结构，数据不没

外部表：删除数据也就没了

79) Hive 中的 UDF 函数有哪几种，你写了哪些？

数学函数，字符串函数

字段大小写转换操作

80) 一个 100G 的文件，内存只有 4G，对其进行全排序，如何用普通 java 程序编写处理。先把文件中的 work.Hash () % 100 分成 100 个小文件，work.Hash () 相同的会 进一个文件。再去求一个文件中每个 word 的次數。拿出每个文件最多的，全局比。

81) 用 hive 实现 select a.key,a.value from a where a.key not in (select b.key from b)Key 不在 b 表中一样的

82) sort by ,order by ,cluster by ,distribute by 代表什么意思

order by: reduce 阶段全局排序，所以自有一个 reduce

sort by: map 端输出会做排序

distribute by: 按照指定的字段对数据进行划分，输入到不同的 reduce

cluster by: 具有 distribute by 的功能外还兼具 sort by 的功能

83) Hive 文件压缩格式有哪些，压缩效率如何

开启压缩

```
set hive.exec.compress.output=true;
```

```
set mapred.output.compress=true;
```

```
set
```

```
mapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec;
```

```
set io.compression.codec=org.apache.hadoop.io.compress.GzipCodec;
```

```
set mapred.output.compression.type=BLOCK;
```

TextFile (压缩方式 Gzip,Bzip2 压缩后不支持 split)

SequenceFile

RCFile(存储方式: 数据按行分块, 每块按列存储。结合了行存储和列存储的优点)

ORCFile

84) Hive 的分组方式

row_number() 是没有重复值的排序(即使两天记录相等也是不重复的),可以利用它来实现分页 dense_rank() 是连续排序,两个第二名仍然跟着第三名 rank() 是跳跃排序的,两个第二名下来就是第四名

85) 数据库索引的创建、hive 中 join 的问题

86) 为什么 spark on hive 来使用

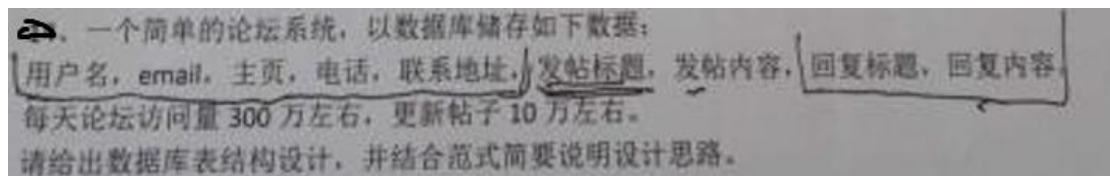
87) Hive 的 UDF

参考链接: <http://blog.csdn.net/bitcarmanlee/article/details/51249260>

88) Hiveserver2

89) Spark 的版本 hive 的版本 hadoop 版本

90) 数据库表结构设计



91) Hive 存储元数据的方式及特点

三种：内存数据库 derby，挺小，不常用。本地 mysql。常用，远程端 mysql。不常用
上网找了下专业名称：single user mode..multi user mode...remote user mode

8. 你认为用 Java, Streaming, pipe 方式开发 map/reduce，各有哪些优缺点。

92)

Java 写 mapreduce 可以实现复杂的逻辑，如果需求简单，则显得繁琐。HiveQL 基本都是针对 hive 中的表数据进行编写，但对复杂的逻辑很难进行实现。写起来简单。

93) Hive 内部表和外部表的区别?

94) hive 底层与数据库交互原理

95) 使用 Hive 或者自定义 MR 实现如下逻辑

product_no	lac_id	moment	start_time	user_id	county_id	staytime	city_id
13429100031	22554	8	2013-03-11 08:55:19.151754088	571	571	282	571
13429100082	22540	8	2013-03-11 08:58:20.152622488	571	571	270	571
13429100082	22691	8	2013-03-11 08:56:37.149593624	571	571	103	571
13429100087	22705	8	2013-03-11 08:56:51.139539816	571	571	220	571
13429100087	22540	8	2013-03-11 08:55:45.150276800	571	571	66	571
13429100082	22540	8	2013-03-11 08:55:38.140225200	571	571	133	571
13429100140	26642	9	2013-03-11 09:02:19.151754088	571	571	18	571
13429100082	22691	8	2013-03-11 08:57:32.151754088	571	571	287	571
13429100189	22558	8	2013-03-11 08:56:24.139539816	571	571	48	571
13429100349	22503	8	2013-03-11 08:54:30.152622440	571	571	211	571

字段解释：

product_no：用户手机号；

lac_id：用户所在基站；

start_time：用户在此基站的开始时间；

staytime：用户在此基站的逗留时间。

需求描述：

根据 lac_id 和 start_time 知道用户当时的位置 根据 staytime 知道用户各个基站的逗留时长。根据轨迹合并连续基站的 staytime。

最终得到每一个用户按时间排序在每一个基站驻留时长

期望输出举例：

13429100082	22540	8	2013-03-11 08:58:20.152622488	571	571	270	571
13429100082	22691	8	2013-03-11 08:56:37.149593624	571	571	390	571
13429100082	22540	8	2013-03-11 08:55:38.140225200	571	571	133	571
13429100087	22705	8	2013-03-11 08:56:51.139539816	571	571	220	571
13429100087	22540	8	2013-03-11 08:55:45.150276800	571	571	66	571

2、Linux 脚本能力考察

2.1 请随意使用各种类型的脚本语言实现：批量将指定目录下的所有文件中的

\$HADOOP_HOME\$替换成/home/ocetl/app/hadoop

2.2 假设有 10 台主机，H1 到 H10，在开启 SSH 互信的情况下，编写一个或多个脚本实现

在所有的远程主机上执行脚本的功能

例如：runRemoteCmd.sh "ls -l"

- 96) hive 如何权限控制?
- 97) hive 能像关系数据库那样, 建多个库吗?
- 98) hive 实现统计的查询语句是什么?
- 99) 生产环境中为什么建议使用外部表?
- 100) hadoop mapreduce 创建类 DataWritable 的作用是什么?
- 101) 为什么创建类 DataWritable?
- 102) 如何实现统计手机流量?
- 103) 对比 hive 与 mapreduce 统计手机流量的区别?
- 104) 如今有 10 个文件夹,每个文件夹都有 1000000 个 url.如今让你找出 top1000000url

不思考歪斜, 功能, 运用 2 个 job, 第一个 job 直接用 filesystem 读取 10 个文件夹作为 map 输入, url 做 key, reduce 计算个 url 的 sum, 下一个 job map 顶用 url 作 key, 运用 -sum 作二次排序, reduce 中取 top10000000 第二种方法, 建 hive 表 A, 挂分区 channel, 每个文件夹是一个分区.
`select x.url,x.c from(select url,count(1) as c from A where channel =" group byurl)x order by x.c desc limie 1000000;`

105) hive 的物理模型跟传统数据库的不同

106) hive 底层与数据库交互原理

Hive 和 Hbase 有各自不同的特征: hive 是高延迟、结构化和面向分析的, hbase 是低延迟、非结构化和面向编程的。Hive 数据仓库在 hadoop 上是高延迟的。Hive 集成 Hbase 就是为了使用 hbase 的一些特性。

Hive 集成 HBase 可以有效利用 HBase 数据库的存储特性, 如行更新和列索引等。在集成的过程中注意维持 HBase jar 包的一致性。Hive 集成 HBase 需要在 Hive 表和 HBase 表之间建立映射关系, 也就是 Hive 表的列(columns)和列类型(column types)与 HBase 表的列族(column families)及列限定词(column qualifiers)建立关联。每一个在 Hive 表中的域都存在于 HBase 中, 而在 Hive 表中不需要包含所有 HBase 中的列。HBase 中的 RowKey 对应到 Hive 中为选择一个域使用:key 来对应, 列族(cf:)映射到 Hive 中的其它所有域, 列为(cf:cq)。

107) 有一个 1G 大小的一个文件, 里面每一行是一个词, 词的大小不超过 16 字节, 内存限制大小是 1M。返回频数最高的 100 个词。

方案: 顺序读文件中, 对于每个词 x , 取 $\text{hash}(x)\%5000$, 然后按照该值存到 5000 个小文件 (记为 $x_0, x_1, \dots, x_{4999}$) 中。这样每个文件大概是 200k 左右。如果其中的有的文件超过了 1M 大小, 还可以按照类似的方法继续往下分, 直到分解得到的小文件的大小都不超过 1M。对每个小文件, 统计每个文件中出现的词以及相应的频率 (可以采用 trie 树/hash_map 等), 并取出出现频率最大的 100 个词 (可以用含 100 个结点的的海军), 并把 100 个词及相应的频率存入文件, 这样又得到了 5000 个文件。下一步就是把这 5000 个文件进行归并 (类似与归并排序) 的过程了。

108) hive 中的压缩格式 RCFile、TextFile、SequenceFile 各有什么区别?

以上 3 种格式一样大的文件哪个占用空间大小

109) 你们的 hive 处理数据能达到的指标是多少?

110) Hive 分桶

在实际生产环境中, 每一个表都可能存储很多的数据, 如果直接进行 join, 通过笛卡尔积会造成大量的 shuffle 溢出, 会延迟 job 完成时间甚至内存不足的想象, 那么就可以对两个表先进行分桶, 然后在 join。这样就可以大大减少笛卡尔积

111) Hive 内部表和外部表的区别:

1、导入数据的时候:

外部表是复制, 内部表是剪切

2、删除表的时候:

外部表只删除元数据, 数据仍然保留

内部表会删除元数据和数据

112) Hive 中的 UDF

先开发一个 java 类, 继承 UDF, 并重载 evaluate 方法

113) 常用的 hive 分析函数

1、row_number:行号递增

2、rank: 行号递增, 如果有相等的, 相当的部分行号一样。然后下面直接递增 N

3、dense_rank: 行号递增, 相等的行号仍然递增

114) join 中出现数据倾斜优化

hive.optimize.skewjoin=true;

如果 join 过程中出现了数据倾斜，应该设置为 true

set hive.skewjoin.key=100000;

这个是 join 的键对应的记录条数超过这个值则会进行优化，优化措施：正常是只有一个 job 的，优化后会有两个 job。当数据量达到 100000 以上的时候，hive 会在启动一个 job，然后将原有的数据的 key 加上一个随机数，将数据打乱。这样数据就会分不到不同的节点上计算。在原打乱的基础上在做一次计算，然后启动一个 job，恢复原来的 key，在做一次计算。

115) Hive 中的 mapjoin

就是将小表加载到缓存中，这样所有的节点都可以访问到这个小表

开启方式：

set hive.auto.convert.join=true;

hive.mapjoin.smalltable.filesize 默认值是 25Mb

这个是自动判断的，当文件小于 25M 的时候，自动启动

或者这样使用：手动

```
select /*+ mapjoin(A)*/ f.a,f.b from A t join B f on ( f.a=t.a and f.ftime=20110802)
```

其中 A 表是小表，B 表是大表，这样让 join 发生在 map 端

mapjoin 的使用场景：

- 1、关联操作中有一张小表
- 2、不等值的链接操作

mapjoin 最好是手动操作

116) hive bucket join

- 1、两个表以相同的方式划分桶
- 2、两个表的桶的个数是倍数关系

```
create table order(cid int , price float) clustered by (cid) into 32 buckets;
```

```
create table customer(id int , first string) clustered by(id) into 32 buckets;
```

```
select price from order t join customer s on t.cid=s.id;
```

117) 关于 where 和 join 的优化

优化前：

```
select o.* from order o join customer c on o.cid=c.id where o.time='2017-01-01'
```

优化后：

```
select o.* from
```

```
(select cid from order where time='2016-01-01')o
```

```
join customer c on o.cid=c.id;
```

优化前是先 join 后再通过 where 进行过滤，这样并没有减轻 reduce 的压力。

优化后是先在 map 端执行 where，过滤数据，然后在 join。这样就会降低计算量

118) Hive group by 优化

1、hive.groupby.skewindata=true;

如果 group by 过程出现倾斜，应该设置为 true

2、set hive.groupby.mapaggr.checkinterval=100000;

group 对应的键对应的记录条数超过这个值则会进行优化

这个仍然是启动两个 job

119) hive 的表优化

1、分区

‘表’相当于一个大目录，分区就是在这个大目录下面的小目录

分区包括静态分区和动态分区

静态分区：就是在建表的时候指定分区的

如果数据量多，一个分区需要一个 insert，就会很麻烦，所以可以使用动态分区

动态分区：

set hive.exec.dynamic.partition=true;

set hive.exec.dynamic.partition.mode=nonstrict;

默认值：strict

描述：strict 是避免全分区字段是动态的，必须有至少一个分区字段是指定有值的

120) hive job 的优化 并行化执行

hive 执行过程中的 job 是按照默认的顺序来执行的，如果没有太大的依赖关系，最好并行执行，减少执行的时间，每个查询被 hive 转化成多个阶段，有些阶段关联性不大，则可以并行执行，减少执行时间。

set hive.exec.parallel=true;

set hive.exec.parallel.thread.numbe=16（默认 8）；

121) hive job 的优化本地化执行

set hive.exec.mode.local.auto=true;

当一个 job 满足如下条件的时候才能真正使用本地模式；

1、job 的输入数据大小必须小于参数：

hive.exec.mode.local.auto.inputbytes.max（默认 128M）

2、job 的 map 数必须小于参数：

hive.exec.mode.local.auto.tasks.max（默认 4）

3、job 的 reduce 数量必须是 0 或者 1

122) hive job 的优化 job 合并输入小文件

在集群中面临这样的问题，集群中很多的小文件，这样会启动很多的（FileInputFormat 会将输入文件分割成 split。split 的个数决定了 map 的个数），而且这些小文件的执行时间特别短，

造成集群的资源没有良好的利用

解决：

```
set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat
```

这样做后，就会把多个 split 分片合并成一个。合并的文件数由 mapred.max.split.size 限制的大小决定

123) hive job 的优化 job 合并输出小文件

```
set hive.merge.smallfiles.avgsize=256000000;当输出文件平均大小小于该值，启动新 job 合并文件
set hive.merge.size.per.task=64000000;设定合并之后的文件大小
```

124) hive job 的优化 JVM 重用

JVM 重用可以使得 JVM 实例在同一个 JOB 中重新使用 N 次

```
set mapred.job.reuse.jvm.num.tasks=10;
```

JVM 重用对 hive 的性能有很大的影响，特别是对小文件的场景或者 task 特别多的场景，可以有效减少执行时间。

当然，这个值不能设置过大，因为有一些 job 是有 reduce 任务的。如果 reduce 任务没有完成，map 任务占用的资源不能都饿到释放，这样其他作业就可能处于等待

125) hive job 的优化压缩数据

中间压缩就是处理 hive 查询的多个 job 之间的数据

中间压缩，减少网络传输的数据量，中间压缩，不需要压缩效果特别好，选择一个节省 CPU 耗时的压缩方式

```
set hive.exec.compress.intermediate=true;
```

```
set hive.intermedia.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
```

```
set hive.intermedia.compression.type=BLOCK; (按照块进行压缩)
```

hive 查询最终输出的压缩

这个阶段可以选择一个压缩效果比较明显的，这样可以降低集群存储的数据量，占用较小的空间

```
set hive.exec.compress.output=true;
```

```
set mapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec;
```

```
set hive.intermedia.compression.type=BLOCK;
```

126) Hive 中的 order by, sort by, distribute by, cluster by 的区别

Order by

会将所有的数据在一个 reducer 上执行，得到的结果是整体有序的。但是由于不能并发执行，所以效率比较低。

sort by

排序操作在多个 reducer 上执行，多只能保证局部有序。无法保证整体有序

而且，使用 sort by 有多个 reduce 的情况下，可能会造成 reduce 处理的数据范围可以重叠
distribute by
可以保证每个 reduce 处理数据范围不重叠。但是不负责排序
cluster by
Cluster by = distribute by + sort by

127) Hive 数据块取样

1、按照数据块取样-----根据百分比取样
SELECT * FROM tableA TABLESAMPLE (50 PERCENT);
2、按照数据块取样-----指定取样大小
SELECT * FROM tableA TABLESAMPLE (30M);
3、按照数据块取样-----指定函数取样
这里指定的行数，每个 Map 中都取样 n ROWS
SELECT * FROM tableA TABLESAMPLE (200 ROWS)

128) Hive 分桶取样

语法：

table_sample: TABLESAMPLE (BUCKET x OUT OF y [ON colname])

其中 x 是要抽样的桶编号，桶编号从 1 开始，colname 表示抽样的列，y 表示桶的数量。

1、假如当前的表没有分过桶

```
SELECT count(1)
FROM tableA TABLESAMPLE (BUCKET 1 OUT OF 10 ON name);
```

将 tableA 中的数据随机分成了 10 个桶，抽取第一个桶

2、当然，如果事先已经分过桶了，我可以这样做，效率会更高一点
仍然是事先分好了 10 个桶

正常的执行方式：

```
SELECT count(1)
FROM tableA TABLESAMPLE (BUCKET 1 OUT OF 10 ON name);
```

但是如果这样执行：

```
SELECT count(1)
FROM tableA TABLESAMPLE (BUCKET 1 OUT OF 20 ON name);
```

就会在第一个桶里面只抽取一半的数据

分桶和未分桶的抽样区别：

已经分桶的表抽样，查询只会扫描相应桶中的数据；

未分桶表的抽样，查询时候需要扫描整表数据，先分桶，再抽样

129) insert into 和 override write 区别

insert into：将某一张表中的数据写到另一张表中

override write: 覆盖之前的内容

130) order by 与 sort by 的区别

a.hive 中的 ORDER BY 语句和关系数据库中的 sql 语法相似。他会查询结果做全局排序，这意味着所有的数据会传送到一个 Reduce 任务上，这样会导致在大数量的情况下，花费大量时间。

b.SORT BY 不是全局排序，其在数据进入 reducer 前完成排序，因此在有多个 reduce 任务情况下，SORT BY 只能保证每个 reduce 的输出有序，而不能保证全局有序 *hive 中通过 set mapred.reduce.tasks=3;来设定 reduce 的数量*

c.一点小知识: DISTRIBUTE BY 可以按指定字段将数据划分到不同的 reduce 中当 DISTRIBUTE BY 的字段和 SORT BY 的字段相同时，可以用 CLUSTER BY 来代替 DISTRIBUTE BY with SORT BY

131) hive 分区 如何将数据定义到哪一个分区中

Create table logs(ts bigint,line string) Partitioned by (dt string,country string);

Load data local inpath '/home/hadoop/par/file01.txt' into table logs partition (dt='2012-06-02',country='cn');

132) HIVE 优化?

1.输出小文件合并

hive.merge.smallfiles.avgsize 设为 5000000

增加 map 数量，可提高 hive 运行速度

set mapred.reduce.tasks=10;

2.map join

大小表 join 时通过使用 hint 的方式指定 join 时使用 mapjoin。

/*+ mapjoin(小表)*/

3.hive 索引

4.优先过滤数据，减少每个阶段的数据量，对分区表加以分区，同时只选择需要使用的字段

5.根据不同的使用目的优化使用方法

6.尽量原子化操作，尽量避免一个 sql 包含复杂逻辑

7.join 操作小表放在 join 的左边

8.如果 union all 的部分个数大于 2，或者 union 部分数据量大，应拆分成多个 insert into 语句。

1.参数优化，小于 6M 自动合并

2.加功能，改成分区表，做 join 写成任务流

3.mapjoin

4.加索引

5.先 where 再 join

6.加小型的 sql

133) hive 有哪些方式保存元数据，各有哪些特点

hive 的数据模型包括：database、table、partition 和 bucket。

1.Database：相当于关系数据库里的命名空间（namespace），它的作用是将用户和数据库的应用隔离到不同的数据库或模式中

2.表（table）：hive 的表逻辑上由存储的数据和描述表格中的数据形式的相关元数据组成。Hive 里的表有两种类型一种叫托管表，这种表的数据文件存储在 hive 的数据仓库里，一种叫外部表，这种表的数据文件可以存放在 hive 数据仓库外部的分布式文件系统上，也可以放到 hive 数据仓库里（注意：hive 的数据仓库也就是 hdfs 上的一个目录，这个目录是 hive 数据文件存储的默认路径，它可以在 hive 的配置文件里进行配置，最终也会存放到元数据仓库里）。

3.分区（partition）：hive 里分区的概念是根据“分区列”的值对表的数据进行粗略划分的机制，在 hive 存储上就体现在表的主目录（hive 的表实际显示就是一个文件夹）下的一个子目录，这个文件夹的名字就是我们定义的分区列的名字，没有实际操作经验的人可能会认为分区列是表的某个字段，其实不是这样，分区列不是表里的某个字段，而是独立的列，我们根据这个列存储表里的数据文件。

4.桶（bucket）：上面的 table 和 partition 都是目录级别的拆分数据，bucket 则是对数据源数据文件本身来拆分数据。

134) hive 与 hbase 的区别

hive：

- 1.数据保存在 hdfs 上，以 hdfs 格式保存数据，映射为 hive 中的表结构
- 2.支持 sql 语言，调用 MR
- 3.不能做实时操作
- 4.相对数据量较小

Hbase：

- 1.Hbase 自己的存储机构
- 2.不支持 sql 语言，不调用 MR
- 3.可以支持实时操作
- 4.相对数据量大，对于反复使用的数据比较适用

135) hive SQL 语句中 select from where group by having order by 的执行顺序?

from--where--group by--having--select--order by,
from:需要从哪个数据表检索数据
where:过滤表中数据的条件
group by:如何将上面过滤出的数据分组
having:对上面已经分组的数据进行过滤的条件
select:查看结果集中的哪个列, 或列的计算结果
order by :按照什么样的顺序来查看返回的数据

136) 有文件 dim_city.txt 如何加载 dim_city 表中

```
load data local inpath"./dim_city.txt" insert into table dim_city;
```

137) hive 如何将下表 table_1 中的数据

```
col1 col2 col3
a b 1,2,3
c d 4,5,6
变为:
col1 col2 col 3
a b 1
a b 2
a b 3
c d 4
c d 5
c d 6
create table table_1(col1 string,col2 string,col3 string)
select col1,col2,name from table_1 LATERAL VIEW explode(split(col3',')) col3 as name;
```

138) HIVE 数据库与 ORACLE 数据库有什么区别,目前 HIVE 数据库不支持哪些函数?

- 查询语言。由于 SQL 被广泛的应用在数据仓库中, 因此, 专门针对 Hive 的特性设计了类 SQL 的查询语言 HQL。熟悉 SQL 开发的开发者可以很方便的使用 Hive 进行开发。
- 数据存储位置。Hive 是建立在 Hadoop 之上的, 所有 Hive 的数据都是存储在 HDFS 中的。而数据库则可以将数据保存在块设备或者本地文件系统中。
- 数据格式。Hive 中没有定义专门的数据格式, 数据格式可以由用户指定, 用户定义数据格式需要指定三个属性: 列分隔符 (通常为空格、"\t"、"\x001")、行分隔符 ("\\n") 以及读取文件数据的方法 (Hive 中默认有三个文件格式 TextFile, SequenceFile 以及 RCFile)。由于在加载数据的过程中, 不需要从用户数据格式到 Hive 定义的数据格式的转换, 因此, Hive 在加载的过程中不会对数据本身进行任何修改, 而只是将数据内容复制或者移动到相应的 HDFS 目录中。而在数据库中, 不同的数据库有不同的存储引擎, 定义了自己的数据格式。所有数据都会按照一定的组织存储, 因此, 数据库加载数据的过程会比较耗时。

d. 数据更新。由于 Hive 是针对数据仓库应用设计的，而数据仓库的内容是读多写少的。因此，Hive 中不支持对数据的改写和添加，所有的数据都是在加载的时候中确定好的。而数据库中的数据通常是需要经常进行修改的，因此可以使用 INSERT INTO ... VALUES 添加数据，使用 UPDATE ... SET 修改数据。

e. 索引。之前已经说过，Hive 在加载数据的过程中不会对数据进行任何处理，甚至不会对数据进行扫描，因此也没有对数据中的某些 Key 建立索引。Hive 要访问数据中满足条件的特定值时，需要暴力扫描整个数据，因此访问延迟较高。由于 MapReduce 的引入，Hive 可以并行访问数据，因此即使没有索引，对于大数据量的访问，Hive 仍然可以体现出优势。数据库中，通常会针对一个或者几个列建立索引，因此对于少量的特定条件的数据的访问，数据库可以有很高的效率，较低的延迟。由于数据的访问延迟较高，决定了 Hive 不适合在线数据查询。

f. 执行。Hive 中大多数查询的执行是通过 Hadoop 提供的 MapReduce 来实现的（类似 select * from tbl 的查询不需要 MapReduce）。而数据库通常有自己的执行引擎。

g. 执行延迟。之前提到，Hive 在查询数据的时候，由于没有索引，需要扫描整个表，因此延迟较高。另外一个导致 Hive 执行延迟高的因素是 MapReduce 框架。由于 MapReduce 本身具有较高的延迟，因此在利用 MapReduce 执行 Hive 查询时，也会有较高的延迟。相对的，数据库的执行延迟较低。当然，这个低是有条件的，即数据规模较小，当数据规模大到超过数据库的处理能力的时候，Hive 的并行计算显然能体现出优势。

h. 可扩展性。由于 Hive 是建立在 Hadoop 之上的，因此 Hive 的可扩展性是和 Hadoop 的可扩展性是一致的（世界上最大的 Hadoop 集群在 Yahoo!，2009 年的规模在 4000 台节点左右）。而数据库由于 ACID 语义的严格限制，扩展行非常有限。目前最先进的并行数据库 Oracle 在理论上的扩展能力也只有 100 台左右。

i. 数据规模。由于 Hive 建立在集群上并可以利用 MapReduce 进行并行计算，因此可以支持很大规模的数据；对应的，数据库可以支持的数据规模较小。

Hive 不支持的函数：decode、rownum、to_char、replace、||、nvl、months_between、add_months、rollup、cube、rank() over、dense_rank() over、row_number() over

139) 使用如下示例数据及数据说明情况,分别实现(1)该数据在 HIVE 库中建表,(2)数据导入到所建表中,(3)使用所建数据表,使用 HQL 统计 2014-12-31 账期手机用户上网总流量.数据说明 文件为 test.txt 字段分割符为|

140) 编写 HIVE 自定义函数实现 ORACLE 数据库中的 addmonths 函数功能,然后封装到 HIVE 函数库中

addmonths(data a,int b)函数功能简单说明,求传入日期 a 经过 B 月后的日期是多少?

141) Hive 的条件判断有几种?

hive 的条件判断 (if、coalesce、case)

142) 请适用 hive 写出查询某网站日志中方位多页面 a 和页面 b 的用户数量的语句:

```
Select count(user) from urla a , urlb b where a.url = b.url ;
```

2.8.2 海量数据处理方法

143) 海量日志数据, 提取出某日访问百度次数最多的那个 IP

此题, 在我之前的一篇文章算法里头有所提到, 当时给出的方案是:

IP 的数目还是有限的, 最多 2^{32} 个,

所以可以考虑使用 hash 将 ip 直接存入内存, 然后进行统计。

再详细介绍下此方案: 首先是这一天, 并且是访问百度的日志中的 IP 取出来, 逐个写入到一个大文件中。

注意到 IP 是 32 位的, 最多有个 2^{32} 个 IP。

同样可以采用映射的方法, 比如模 1000, 把整个大文件映射为 1000 个小文件,

再找出每个小文中出现频率最大的 IP（可以采用 hash_map 进行频率统计，然后再找出频率最大的几个）及相应的频率。然后再在这 1000 个最大的 IP 中，找出那个频率最大的 IP，即为所求。

144) 搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为 1-255 字节。

假设目前有一千万个记录（这些查询串的重复度比较高，虽然总数是 1 千万，但如果除去重复后，不超过 3 百万个。

一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。），请你统计最热门的 10 个查询串，要求使用的内存不能超过 1G。

典型的 Top K 算法，还是在这篇文章里头有所阐述。

文中，给出的最终算法是：第一步、先对这批海量数据预处理，在 $O(N)$ 的时间内用 Hash 表完成排序；然后，第二步、借助堆这个数据结构，找出 TopK，时间复杂度为 $N \log K$ 。

即，借助堆结构，我们可以在 \log 量级的时间内查找和调整/移动。因此，维护一个 K(该题目中是 10)大小的小根堆，然后遍历 300 万的 Query，分别和根元素进行对比所以，我们最终的时间复杂度是： $O(N) + N' * O(\log K)$ ，(N 为 1000 万，N' 为 300 万)。ok，更多，详情，请参考原文。或者：采用 trie 树，关键字域存该查询串出现的次数，没有出现过为 0。最后用 10 个元素的最小堆来对出现频率进行排序。

145) 有一个 1G 过 大小的一个文件，里面每一行是一个词，词的大小不超过 16 字节，内存限制大小是 1M 。返回频数最高的 100 个词。

方案：顺序读文件中，对于每个词 x，取 $\text{hash}(x) \% 5000$ ，然后按照该值存到 5000 个小文件（记为 $x_0, x_1, \dots, x_{4999}$ ）中。这样每个文件大概是 200k 左右。如果其中的有的文件超过了 1M 大小，还可以按照类似的方法继续往下分，直到分解得到的小文件的大小都不超过 1M。对每个小文件，统计每个文件中出现的词以及相应的频率（可以采用 trie 树/hash_map 等），并取出出现频率最大的 100 个词（可以用含 100 个结点的最小堆），并把 100 个词及相应的频率存入文件，这样又得到了 5000 个文件。下一步就是把这 5000 个文件进行归并（类似与归并排序）的过程了。

146) 有 10 个文件，每个文件 1G，每个文件的每一行存放的都是用户的 query，每个文件的 query 都可能重复。要求你按照 query 的频度排序。

还是典型的 TOP K 算法，解决方案如下：

方案 1：顺序读取 10 个文件，按照 $\text{hash}(\text{query})\%10$ 的结果将 query 写入到另外 10 个文件（记为 f_0 到 f_9 ）中。这样新生成的文件每个的大小大约也 1G（假设 hash 函数是随机的）。找一台内存在 2G 左右的机器，依次对用 $\text{hash_map}(\text{query}, \text{query_count})$ 来统计每个 query 出现的次数。利用快速/堆/归并排序按照出现次数进行排序。将排序好的 query 和对应的 query_count 输出到文件中。这样得到了 10 个排好序的文件（记为 f_{0_sort} 到 f_{9_sort} ）。对这 10 个文件进行归并排序（内排序与外排序相结合）。

方案 2：一般 query 的总量是有限的，只是重复的次数比较多而已，可能对于所有的 query，一次性

就可以加入到内存了。这样，我们就可以采用 trie 树/hash_map 等直接来统计每个 query 出现的次数，然

后按出现次数做快速/堆/归并排序就可以了。

方案 3：与方案 1 类似，但在做完 hash，分成多个文件后，可以交给多个文件来处理，采用分布式的架构来处理（比如 MapReduce），最后再进行合并。

147) 给定 a.b 两个文件，各存放 50 亿个 url，每个 url 各占 64 字节，内存限制是 4G，让你找出 a.b 文件共同的 url？

方案 1：可以估计每个文件的大小为 $50 \times 64 = 3200$ GB，远远大于内存限制的 4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。遍历文件 a，对每个 url 求取 $\text{hash}(\text{url})\%1000$ ，然后根据所取得的值将 url 分别存储到 1000 个小文件（记为 a_0, a_1, \dots, a_{999} ）中。这样每个小文件的大小大约为 300M。遍历文件 b，采取和 a 相同的方式将 url 分别存储到 1000 个小文件（记为 b_0, b_1, \dots, b_{999} ）。这样处理后，所有可能相同的 url 都在对应的小文件（ a_0 vs b_0, a_1 vs b_1, \dots, a_{999} vs b_{999} ）中，不对应的小文件不可能有相同的 url。然后我们只要求出 1000 对小文件中相同的 url 即可。求每对小文件中相同的 url 时，可以把其中一个小文件的 url 存储到 hash_set 中。然后遍历另一个小文件的每个 url，看其是否在刚才构建的 hash_set 中，如果是，那么就是共同的 url，存到文件里面就可以了。

方案 2：如果允许有一定的错误率，可以使用 Bloom filter，4G 内存大概可以表示 340 亿 bit。将其中一个文件中的 url 使用 Bloom filter 映射为这 340 亿 bit，然后挨个读取另外一个文件的 url，检查是否与 Bloom filter，如果是，那么该 url 应该是共同的 url（注意会有一定的错误率）。Bloom filter 日后会在本 BLOG 内详细阐述。

148) 在 2.5 亿个整数中找出不重复的整数，注，内存不足以容纳这 2.5 亿个整数

方案 1：采用 2-Bitmap（每个数分配 2bit，00 表示不存在，01 表示出现一次，10 表示多次，11 无意义）进行，共需内存内存，还可以接受。然后扫描这 2.5 亿个整数，查看 Bitmap 中相对应位，如果是 00 变 01，01 变 10，10 保持不变。扫描完后，查看 bitmap，把

对应位是 01 的整数输出即可。

方案 2：也可采用与第 1 题类似的方法，进行划分小文件的方法。然后在小文件中找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。

149) 一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前 10 个词，请给出思想，给出时间复杂度分析。

方案 1：这题是考虑时间效率。用 trie 树统计每个词出现的次数，时间复杂度是 $O(n \cdot le)$ (le 表示单词的平均长度)。然后是找出出现最频繁的前 10 个词，可以用堆来实现，前面的题中已经讲到了，时间复杂度是 $O(n \cdot \lg 10)$ 。所以总的时间复杂度，是 $O(n \cdot le)$ 与 $O(n \cdot \lg 10)$ 中较大的哪一个。附、100w 个数中找出最大的 100 个数。在前面的题中，我们已经提到了，用一个含 100 个元素的最小堆完成。复杂度为 $O(100w \cdot \lg 100)$ 。

方案 2：采用快速排序的思想，每次分割之后只考虑比轴大的一部分，知道比轴大的一部分在比 100 多的时候，采用传统排序算法排序，取前 100 个。复杂度为 $O(100w \cdot 100)$ 。

方案 3：采用局部淘汰法。选取前 100 个元素，并排序，记为序列 L。然后一次扫描剩余的元素 x，与排好序的 100 个元素中最小的元素比，如果比这个最小的要大，那么把这个最小的元素删除，并把 x 利用插入排序的思想，插入到序列 L 中。依次循环，知道扫描了所有的元素。复杂度为 $O(100w \cdot 100)$ 。

150) 有一千万条短信，有重复，以文本文件的形式保存，一行一条，有重复。请用 5 分钟时间，找出重复出现最多的前 10 条。

分析：常规方法是先排序，在遍历一次，找出重复最多的前 10 条。但是排序的算法复杂度最低为 $n \lg n$ 。

可以设计一个 hash_table, `hash_map<string, int>`，依次读取一千万条短信，加载到 hash_table 表中，并且统计重复的次数，与此同时维护一张最多 10 条的短信表。这样遍历一次就能找出最多的前 10 条，算法复杂度为 $O(n)$ 。

实现如下：

```
1. #include<iostream>
2. #include<map>
3. #include<iterator>
4. #include<stdio.h>
5. using namespace std;
6.
7. #define HASH __gnu_cxx
```



```
8. #include<ext/hash_map>

9. #define uint32_t unsigned int

10. #define uint64_t unsigned long int

11. struct StrHash

12. {

13. uint64_t operator()(const std::string& str) const

14. {

15. uint32_t b = 378551;

16. uint32_t a = 63689;

17. uint64_t hash = 0;

18.

19. for(size_t i = 0; i < str.size(); i++)

20. {

21. hash = hash * a + str[i];

22. a = a * b;

23. }

24.

25. return hash;

26. }

27. uint64_t operator()(const std::string& str, uint32_t field) const

28. {

29. uint32_t b = 378551;
```

```
30. uint32_t a = 63689;

31. uint64_t hash = 0;

32. for(size_t i = 0; i < str.size(); i++)

33. {

34. hash = hash * a + str[i];

35. a = a * b;

36. }

37. hash = (hash<<8)+field;

38. return hash;

39. }

40. };

41. struct NameNum{

42. string name;

43. int num;

44. NameNum():num(0),name(""){}

45. };

46. int main()

47. {

48. HASH::hash_map< string, int, StrHash > names;

49. HASH::hash_map< string, int, StrHash >::iterator it;

50. NameNum namenum[10];

51. string l = "";
```

```
52. while(getline(cin, l))
53. {
54. it = names.find(l);
55. if(it != names.end())
56. {
57. names[l] ++;
58. }
59. else
60. {
61. names[l] = 1;
62. names[l] = 1;
63. }
64. }
65. int i = 0;
66. int max = 1;
67. int min = 1;
68. int minpos = 0;
69. for(it = names.begin(); it != names.end(); ++ it)
70. {
71. if(i < 10)
72. {
73. namenum[i].name = it->first;
```

```
74. namenum[i].num = it->second;

75. if(it->second > max)

76. max = it->second;

77. else if(it->second < min)

78. {

79. min = it->second;

80. minpos = i;

81. }

82. }

83. else

84. {

85. if(it->second > min)

86. {

87. namenum[minpos].name = it->first;

88. namenum[minpos].num = it->second;

89. int k = 1;

90. min = namenum[0].num;

91. minpos = 0;

92. while(k < 10)

93. {

94. if(namenum[k].num < min)
```

```
95. {  
96. min = namenum[k].num;  
97. minpos = k;  
98. }  
99. k ++;  
100. }  
101. }  
102. }  
103. i++;  
104.  
105. }  
106. i = 0;  
107. cout << "maxlength (string,num): " << endl;  
108. while( i < 10)  
109. {  
110. cout << "(" << namenum[i].name.c_str() << "," << namenum[i].num << ")" <<  
    endl;  
111. i++;  
112. }  
113. return 0;  
114. }
```

使用 g++ 编译如下:

g++ main.cpp -o main

短信文本文件为：msg.txt

运行：./main < msg.txt

输出结果为：

maxlength (string,num):

(点点母婴坊,4)

(农机配件维修,5)

(红胜超市,6)

(龙溪大酒店,8)

(张记饺子馆,3)

(友谊旅店,3)

(明珠通讯,3)

(金源旅馆,3)

(洞庭山天然泉水,2)

(清源超市,3)

151) 有 20 亿条数据，每条数据的大小在 1K-1M 不等，每条数据有一个唯一的 u_int64 的 id，请设计一个读取数据系统，能根据 id 获取数据，要求：内存有限制，16G,尽可能利用内存资源，尽可能高效的获取数据，可利用磁盘，磁盘容量不受限制（写出主要设计思路）

152) 总结

首先处理大数据的面试题，有些基本概念要清楚：

- (1) **1Gb = 10⁹bytes (1Gb = 10 亿字节)**：1Gb = 1024Mb, 1Mb = 1024Kb, 1Kb = 1024bytes;
- (2) **基本流程**是，分解大问题，解决小问题，从局部最优中选择全局最优；（当然，如果直接放内存里就能解决的话，那就直接想办法求解，不需要分解了。）
- (3) **分解过程**常用方法：hash(x)%m。其中 x 为字符串/url/ip，m 为小问题的数目，比如把一个文件分解为 1000 份，m=1000；
- (4) **解决问题**辅助数据结构：hash_map，Trie 树，bit map，二叉排序树（AVL，SBT，红黑树）；
- (5) **top K 问题**：最大 K 个用最小堆，最小 K 个用最大堆。（至于为什么？自己在纸上写个小栗子，试一下就知道了。）
- (6) 处理大数据**常用排序**：快速排序/堆排序/归并排序/桶排序

一、Bloom filter

适用范围：可以用来实现数据字典，进行数据的判重，或者集合求交集

基本原理及要点：

对于**原理**来说很简单，位数组+k 个独立 hash 函数。将 hash 函数对应的值的位数组置 1，查找时如果发现所有 hash 函数对应位都是 1 说明存在，很明显这个过程并不保证查找的结果是 100%正确的。同时也不支持删除一个已经插入的关键字，因为该关键字对应的位会牵动到其他的关键字。所以一个简单的改进就是 counting Bloom filter，用一个 counter 数组代替位数组，就可以支持删除了。

还有一个比较重要的问题，**如何根据输入元素个数 n，确定位数组 m 的大小及 hash 函数个数**。当 hash 函数个数 $k=(\ln 2)*(m/n)$ 时错误率最小。在错误率不大于 E 的情况下，m 至

少要等于 $n \cdot \lg(1/E)$ 才能表示任意 n 个元素的集合。但 m 还应该更大些, 因为还要保证 bit 数组里至少一半为 0, 则 m 应该 $\geq n \lg(1/E) \cdot \lg e$ 大概就是 $n \lg(1/E) 1.44$ 倍(\lg 表示以 2 为底的对数)。

举个例子我们假设错误率为 0.01, 则此时 m 应大概是 n 的 13 倍。这样 k 大概是 8 个。注意这里 m 与 n 的单位不同, m 是 bit 为单位, 而 n 则是以元素个数为单位(准确的说是不同元素的个数)。通常单个元素的长度都是有很多 bit 的。所以使用 bloom filter 内存上通常都是节省的。

扩展:

Bloom filter 将集合中的元素映射到位数组中, 用 k (k 为哈希函数个数) 个映射位是否全 1 表示元素在不在这个集合中。Counting bloom filter (CBF) 将位数组中的每一位扩展为一个 counter, 从而支持了元素的删除操作。Spectral Bloom Filter (SBF) 将其与集合元素的出现次数关联。SBF 采用 counter 中的最小值来近似表示元素的出现频率。

问题实例:

给你 A,B 两个文件, 各存放 50 亿条 URL, 每条 URL 占用 64 字节, 内存限制是 4G, 让你找出 A,B 文件共同的 URL。如果是三个乃至 n 个文件呢?

根据这个问题我们来计算下内存的占用, $4G = 2^{32}$ 大概是 40 亿*8 大概是 340 亿, $n=50$ 亿, 如果按出错率 0.01 算需要的大概是 650 亿个 bit。现在可用的是 340 亿, 相差并不多, 这样可能会使出错率上升些。另外如果这些 urlip 是一一对应的, 就可以转换成 ip, 则大大简单了。

二、Hashing

适用范围: 快速查找, 删除的基本数据结构, 通常需要总数据量可以放入内存

基本原理及要点:

hash 函数选择, 针对字符串, 整数, 排列, 具体相应的 hash 方法。碰撞处理, 一种是 open hashing, 也称为拉链法; 另一种就是 closed hashing, 也称开地址法, openedaddressing。

扩展:

d-left hashing 中的 d 是多个的意思, 我们先简化这个问题, 看一看 2-left hashing。2-left hashing 指的是将一个哈希表分成长度相等的两半, 分别叫做 T1 和 T2, 给 T1 和 T2 分别配备一个哈希函数, h_1 和 h_2 。在存储一个新的 key 时, 同时用两个哈希函数进行计算, 得出两个地址 $h_1[key]$ 和 $h_2[key]$ 。这时需要检查 T1 中的 $h_1[key]$ 位置和 T2 中的 $h_2[key]$ 位置, 哪一个位置已经存储的 (有碰撞的) key 比较多, 然后将新 key 存储在负载少的位置。如果两边一样多, 比如两个位置都为空或者都存储了一个 key, 就把新 key 存储在左边的 T1 子表中, 2-left 也由此而来。在查找一个 key 时, 必须进行两次 hash, 同时查找两个位置。

问题实例:

1).海量日志数据, 提取出某日访问百度次数最多的那个 IP。

IP 的数目还是有限的, 最多 2^{32} 个, 所以可以考虑使用 hash 将 ip 直接存入内存, 然后进行统计。

三、bit-map

适用范围: 可进行数据的快速查找, 判重, 删除, 一般来说数据范围是 int 的 10 倍以下

基本原理及要点: 使用 bit 数组来表示某些元素是否存在, 比如 8 位电话号码

扩展:

bloom filter 可以看做是对 bit-map 的扩展

问题实例:

1)已知某个文件内包含一些电话号码, 每个号码为 8 位数字, 统计不同号码的个数。

8 位最多 99 999 999，大概需要 99m 个 bit，大概 10 几 m 字节的内存即可。

2)2.5 亿个整数中找出不重复的整数的个数，内存空间不足以容纳这 2.5 亿个整数。

将 bit-map 扩展一下，用 2bit 表示一个数即可，0 表示未出现，1 表示出现一次，2 表示出现 2 次及以上。或者我们不用 2bit 来进行表示，我们用两个 bit-map 即可模拟实现这个 2bit-map。

四、堆

适用范围：海量数据前 n 大，并且 n 比较小，堆可以放入内存

基本原理及要点：最大堆求前 n 小，最小堆求前 n 大。方法，比如求前 n 小，我们比较当前元素与最大堆里的最大元素，如果它小于最大元素，则应该替换那个最大元素。这样最后得到的 n 个元素就是最小的 n 个。适合大数据量，求前 n 小，n 的大小比较小的情况，这样可以扫描一遍即可得到所有的前 n 元素，效率很高。

扩展：

双堆，一个最大堆与一个最小堆结合，可以用来维护中位数。

问题实例：

1)100w 个数中找最大的前 100 个数。

用一个 100 个元素大小的最小堆即可。

五、双层桶划分---- 其实本质上就是【分而治之】的思想，重在分的技巧上！

适用范围：第 k 大，中位数，不重复或重复的数字
基本原理及要点：因为元素范围很大，不能利用直接寻址表，所以通过多次划分，逐步确定范围，然后最后在一个可以接受的范围内进行。可以通过多次缩小，双层只是一个例子。

扩展：

问题实例：

1)2.5 亿个整数中找出不重复的整数的个数，内存空间不足以容纳这 2.5 亿个整数。

有点像鸽巢原理，整数个数为 2^{32} ，也就是，我们可以将这 2^{32} 个数，划分为 2^8 个区域(比如用单个文件代表一个区域)，然后将数据分离到不同的区域，然后不同的区域在利用 bitmap 就可以直接解决了。也就是说只要有足够的磁盘空间，就可以很方便的解决。

2)5 亿个 int 找它们的中位数。

这个例子比上面那个更明显。首先我们将 int 划分为 2^{16} 个区域，然后读取数据统计落到各个区域里的数的个数，之后我们根据统计结果就可以判断中位数落到那个区域，同时知道这个区域中的第几大数刚好是中位数。然后第二次扫描我们只统计落在这个区域中的那些数就可以了。实际上，如果不是 int 是 int64，我们可以经过 3 次这样的划分即可降低到可以接受的程度。即可以先将 int64 分成 2^{24} 个区域，然后确定区域的第几大数，在将该区域分成 2^{20} 个子区域，然后确定是子区域的第几大数，然后子区域里的数的个数只有 2^{20} ，就可以直接利用 direct addr table 进行统计了。

六、数据库索引

适用范围：大数据量的增删改查

基本原理及要点：利用数据的设计实现方法，对海量数据的增删改查进行处理。

七、倒排索引(Inverted index)

适用范围：搜索引擎，关键字查询

基本原理及要点：为何叫倒排索引？一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。以英文为例，下面是要被索引的文本： T0 = "it is what it is" T1 = "what is it" T2 = "it is a banana"

我们就能得到下面的反向文件索引：

"a": {2} "banana": {2} "is": {0, 1, 2} "it": {0, 1, 2} "what": {0, 1}

检索的条件"what","is"和"it"将对应集合的交集。

正向索引开发出来用来存储每个文档的单词的列表。正向索引的查询往往满足每个文档有序频繁的全文查询和每个单词在校验文档中的验证这样的查询。在正向索引中，文档占据了中心的位置，每个文档指向了一个它所包含的索引项的序列。也就是说文档指向了它包含的那些单词，而反向索引则是单词指向了包含它的文档，很容易看到这个反向的关系。

扩展：

问题实例：文档检索系统，查询那些文件包含了某单词，比如常见的学术论文的关键字搜索。

八、外排序

适用范围：大数据的排序，去重

基本原理及要点：外排序的归并方法，置换选择败者树原理，最优归并树

扩展：

问题实例：

1).有一个 1G 大小的一个文件，里面每一行是一个词，词的大小不超过 16 个字节，内存限制大小是 1M。返回频数最高的 100 个词。这个数据具有很明显的特点，词的大小为 16 个字节，但是内存只有 1m 做 hash 有些不够，所以可以用来排序。内存可以当输入缓冲区使用。

九、trie 树

适用范围：数据量大，重复多，但是数据种类小可以放入内存

基本原理及要点：实现方式，节点孩子的表示方式

扩展：压缩实现。

问题实例：

1).有 10 个文件，每个文件 1G，每个文件的每一行都存放的是用户的 query，每个文件的 query 都可能重复。要你按照 query 的频度排序。

2).1000 万字符串，其中有些是相同的(重复),需要把重复的全部去掉，保留没有重复的字符串。请问怎么设计和实现？

3).寻找热门查询：查询串的重复度比较高，虽然总数是 1 千万，但如果除去重复后，不超过 3 百万个，每个不超过 255 字节。

十、分布式处理 mapreduce

适用范围：数据量大，但是数据种类小可以放入内存

基本原理及要点：将数据交给不同的机器去处理，数据划分，结果归约。

扩展：

问题实例：

1).The canonical example application of MapReduce is a process to count the appearances of each

different word in a set of documents:

2).海量数据分布在 100 台电脑中，想个办法高效统计出这批数据的 TOP10。

3).一共有 N 个机器，每个机器上有 N 个数。每个机器最多存 O(N)个数并对它们操作。

如何找到 N^2 个数的中数 (median) ?

经典问题分析

上千万 or 亿数据 (有重复)，统计其中出现次数最多的前 N 个数据,分两种情况：可一次读入内存，不可一次读入。

可用思路：trie 树+堆，数据库索引，划分子集分别统计，hash，分布式计算，近似统计，

外排序

所谓的是否能一次读入内存，实际上应该指去除重复后的数据量。如果去重后数据可以放入内存，我们可以为数据建立字典，比如通过 map, hashmap, trie, 然后直接进行统计即可。当然在更新每条数据的出现次数的时候，我们可以利用一个堆来维护出现次数最多的前 N 个数据，当然这样导致维护次数增加，不如完全统计后在求前 N 大效率高。

如果数据无法放入内存。一方面我们可以考虑上面的字典方法能否被改进以适应这种情形，可以做的改变就是将字典存放到硬盘上，而不是内存，这可以参考数据库的存储方法。当然还有更好的方法，就是可以采用分布式计算，基本上就是 map-reduce 过程，首先可以根据数据值或者把数据 hash(md5)后的值，将数据按照范围划分到不同的机子，最好可以让数据划分后可以一次读入内存，这样不同的机子负责处理各种的数值范围，实际上就是 map。得到结果后，各个机子只需拿出各自的出现次数最多的前 N 个数据，然后汇总，选出所有的数据中出现次数最多的前 N 个数据，这实际上就是 reduce 过程。实际上可能想直接将数据均分到不同的机子上进行处理，这样是无法得到正确的解的。因为一个数据可能被均分到不同的机子上，而另一个则可能完全聚集到一个机子上，同时还可能存在具有相同数目的数据。比如我们要找出现次数最多的前 100 个，我们将 1000 万的数据分布到 10 台机器上，找到每台出现次数最多的前 100 个，归并之后这样不能保证找到真正的第 100 个，因为比如出现次数最多的第 100 个可能有 1 万个，但是它被分到了 10 台机子，这样在每台上只有 1 千个，假设这些机子排名在 1000 个之前的那些都是单独分布在一台机器上的，比如有 1001 个，这样本来具有 1 万个的这个就会被淘汰，即使我们让每台机子选出出现次数最多的 1000 个再归并，仍然会出错，因为可能存在大量个数为 1001 个的发生聚集。因此不能将数据随便均分到不同机子上，而是要根据 hash 后的值将它们映射到不同的机子上处理，让不同的机器处理一个数值范围。

而外排序的方法会消耗大量的 IO，效率不会很高。而上面的分布式方法，也可以用于单机版本，也就是将总的根据值的范围，划分成多个不同的子文件，然后逐个处理。处理完毕之后再对这些单词的及其出现频率进行一个归并。实际上就可以利用一个外排序的归并过程。

另外还可以考虑近似计算，也就是我们可以通过结合自然语言属性，只将那些真正实际中出现最多的那些词作为一个字典，使得这个规模可以放入内存。

2.9 Hadoop HA

2.9.1 知识点

详情参考课件:大数据离线阶段--Day09.pdf

1) Namenode HA

HA(High Available), 高可用, 是保证业务连续性的有效解决方案, 一般有两个或两个以上的节点, 分为 活动节点 (Active) 及 备用节点 (Standby) 。用于实现业务的不中

断或短暂中断

NN 是 HDFS 集群的单点故障点.在 HA 具体实现方法不同情况下, HA 框架的流程是一致的, 不一致的就是如何存储、管理、同步 edits 编辑日志文件。

QJM/Quorum Journal Manager,基本原理就是用 $2N+1$ 台 JournalNode 存储 EditLog, 每次写数据操作有 $\geq N+1$ 返回成功时即认为该次写成功, 数据不会丢失了

在 HA 模式下, datanode 需要确保同一时间有且只有一个 NN 能命令 DN。

FailoverController 主要包括三个组件: :

HealthMonitor: 监控 NameNode 是否处于 unavailable 或 unhealthy 状态.当前通过 RPC 调用 NN 相应的方法完成。

ActiveStandbyElector: 监控 NN 在 ZK 中的状态。

ZKFailoverController: 订阅 HealthMonitor 和 ActiveStandbyElector 的事件, 并管理 NN 的状态,另外 zkfc 还负责解决 fencing (也就是脑裂问题)。

ZKFailoverController 主要职责:

健康监测: 周期性的向它监控的 NN 发送健康探测命令, 从而来确定某个 NameNode 是否处于健康状态, 如果机器宕机, 心跳失败, 那么 zkfc 就会标记它处于一个不健康的状态

会话管理: 如果 NN 是健康的, zkfc 就会在 zookeeper 中保持一个打开的会话, 如果 NameNode 同时还是 Active 状态的, 那么 zkfc 还会在 Zookeeper 中占有一个类型为短暂类型的 znode, 当这个 NN 挂掉时, 这个 znode 将会被删除, 然后备用的 NN 将会得到这把锁, 升级为主 NN, 同时标记状态为 Active

当宕机的 NN 新启动时, 它会再次注册 zookeeper, 发现已经有 znode 锁了, 便会自动变为 Standby 状态, 如此往复循环, 保证高可靠, 需要注意, 目前仅仅支持最多配置 2 个 NN

master 选举: 通过在 zookeeper 中维持一个短暂类型的 znode, 来实现抢占式的锁机制, 从而判断那个 NameNode 为 Active 状态

2) yarn HA

Hadoop 2.4.0 版本开始, Yarn 实现了 ResourceManager HA

由于资源使用情况和 NodeManager 信息都可以通过 NodeManager 的心跳机制重新构建出来, 因此只需要对 ApplicationMaster 相关的信息进行持久化存储即可。

在一个典型的 HA 集群中, 两台独立的机器被配置成 ResourceManager。在任意时间, 有且只允许一个活动的 ResourceManager, 另外一个备用。切换分为两种方式:

手动切换: 在自动恢复不可用时, 管理员可用手动切换状态, 或是从 Active 到 Standby, 或是从 Standby 到 Active。

自动切换: 基于 Zookeeper, 但是区别于 HDFS 的 HA, 2 个节点间无需配置额外的 ZFKC 守护进程来同步数据。

3) HA Hadoop 搭建

参考资料:Hadoop>Hadoop ha 下文档 hadoop2.X-HA 集群搭建.txt

- 1.修改 Linux 主机名
- 2.修改 IP
- 3.修改主机名和 IP 的映射关系 /etc/hosts
- 4.关闭防火墙
- 5.ssh 免登陆
- 6.安装 JDK，配置环境变量等
- 7.注意集群时间要同步

集群部署节点角色的规划（7 节点）

server01	namenode	zkfc		
server02	namenode	zkfc		
server03	resourcemanager			
server04	resourcemanager			
server05	datanode	nodemanager	zookeeper	journal node
server06	datanode	nodemanager	zookeeper	journal node
server07	datanode	nodemanager	zookeeper	journal node

集群部署节点角色的规划（3 节点）

server01	namenode	resourcemanager	zkfc	nodemanager	datanode	zookeeper	journal node
server02	namenode	resourcemanager	zkfc	nodemanager	datanode	zookeeper	journal node
server03	datanode	nodemanager	zookeeper	journal node			

安装步骤:

- 1.安装配置 zookeeper 集群

- 1.1 解压

```
tar -zxvf zookeeper-3.4.5.tar.gz -C /home/hadoop/app/
```

- 1.2 修改配置

```
cd /home/hadoop/app/zookeeper-3.4.5/conf/
```

```
cp zoo_sample.cfg zoo.cfg
```

```
vim zoo.cfg
```

```
修改: dataDir=/home/hadoop/app/zookeeper-3.4.5/tmp
```

在最后添加:

```
server.1=hadoop05:2888:3888
```

```
server.2=hadoop06:2888:3888
```

```
server.3=hadoop07:2888:3888
```

保存退出

然后创建一个 tmp 文件夹

```
mkdir /home/hadoop/app/zookeeper-3.4.5/tmp
```

```
echo 1 > /home/hadoop/app/zookeeper-3.4.5/tmp/myid
```

1.3 将配置好的 zookeeper 拷贝到其他节点(首先分别在 hadoop06、hadoop07 根目录下创建一个 hadoop 目录: `mkdir /hadoop`)

```
scp -r /home/hadoop/app/zookeeper-3.4.5/
```

hadoop06:/home/hadoop/app/

```
scp -r /home/hadoop/app/zookeeper-3.4.5/
```

hadoop07:/home/hadoop/app/

注意: 修改 hadoop06、hadoop07 对应
/hadoop/zookeeper-3.4.5/tmp/myid 内容

hadoop06:

```
echo 2 > /home/hadoop/app/zookeeper-3.4.5/tmp/myid
```

hadoop07:

```
echo 3 > /home/hadoop/app/zookeeper-3.4.5/tmp/myid
```

2.安装配置 hadoop 集群

2.1 解压

```
tar -zxvf hadoop-2.6.4.tar.gz -C /home/hadoop/app/
```

2.2 配置 HDFS (hadoop2.0 所有的配置文件都在\$HADOOP_HOME/etc/hadoop 目录下)

```
#将 hadoop 添加到环境变量中
```

```
vim /etc/profile
```

```
export JAVA_HOME=/usr/java/jdk1.7.0_55
```

```
export HADOOP_HOME=/hadoop/hadoop-2.6.4
```

```
export
```

```
PATH=$PATH:$JAVA_HOME/cluster1n:$HADOOP_HOME/cluster1n
```

```
#hadoop2.0 的配置文件全部在$HADOOP_HOME/etc/hadoop 下
```

```
cd /home/hadoop/app/hadoop-2.6.4/etc/hadoop
```

2.2.1 修改 hadoop-env.sh

```
export JAVA_HOME=/home/hadoop/app/jdk1.7.0_55
```

```
#####  
##
```

2.2.2 修改 core-site.xml

```
<configuration>
```

```

<!-- 集群名称在这里指定！该值来自于 hdfs-site.xml 中的配置 -->
<property>
<name>fs.defaultFS</name>
<value>hdfs://cluster1</value>
</property>
<!-- 这里的路径默认是 NameNode、DataNode、JournalNode 等存放数据的公共目录 -->
<property>
<name>hadoop.tmp.dir</name>
<value>/root/apps/hadoop/tmp</value>
</property>

<!-- ZooKeeper 集群的地址和端口。注意，数量一定是奇数，且不少于三个节点-->
<property>
<name>ha.zookeeper.quorum</name>
<value>hadoop05:2181,hadoop06:2181,hadoop07:2181</value>
</property>
</configuration>

#####
##

```

2.2.3 修改 hdfs-site.xml

```

<configuration>
<!-- 指定 hdfs 的 nameservice 为 cluster1，需要和 core-site.xml 中的保持一致 -->
<property>
<name>dfs.nameservices</name>
<value>cluster1</value>
</property>
<!-- cluster1 下面有两个 NameNode，分别是 nn1，nn2 -->
<property>
<name>dfs.ha.namenodes.cluster1</name>
<value>nn1,nn2</value>
</property>
<!-- nn1 的 RPC 通信地址 -->
<property>
<name>dfs.namenode.rpc-address.cluster1.nn1</name>
<value>mini1:9000</value>
</property>
<!-- nn1 的 http 通信地址 -->
<property>
<name>dfs.namenode.http-address.cluster1.nn1</name>
<value>hadoop00:50070</value>
</property>
<!-- nn2 的 RPC 通信地址 -->

```

```
<property>
<name>dfs.namenode.rpc-address.cluster1.nn2</name>
<value>hadoop01:9000</value>
</property>
<!-- nn2 的 http 通信地址 -->
<property>
<name>dfs.namenode.http-address.cluster1.nn2</name>
<value>hadoop01:50070</value>
</property>
<!-- 指定 NameNode 的 edits 元数据在 JournalNode 上的存放位置 -->
<property>
<name>dfs.namenode.shared.edits.dir</name>
<value>qjournal://hadoop05:8485;hadoop06:8485;hadoop07:8485/cluster1</value>
</property>
<!-- 指定 JournalNode 在本地磁盘存放数据的位置 -->
<property>
<name>dfs.journalnode.edits.dir</name>
<value>/home/hadoop/app/hdpdata/journaldata</value>
</property>
<!-- 开启 NameNode 失败自动切换 -->
<property>
<name>dfs.ha.automatic-failover.enabled</name>
<value>true</value>
</property>
<!-- 指定该集群出故障时，哪个实现类负责执行故障切换 -->
<property>
<name>dfs.client.failover.proxy.provider.cluster1</name>
<value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>
</property>
<!-- 配置隔离机制方法，多个机制用换行分割，即每个机制暂用一行-->
<property>
<name>dfs.ha.fencing.methods</name>
<value>
sshfence
</value>
</property>
<!-- 使用 sshfence 隔离机制时需要 ssh 免登陆 -->
<property>
<name>dfs.ha.fencing.ssh.private-key-files</name>
<value>/home/hadoop/.ssh/id_rsa</value>
</property>
<!-- 配置 sshfence 隔离机制超时时间 -->
<property>
<name>dfs.ha.fencing.ssh.connect-timeout</name>
```

```
<value>30000</value>
</property>
</configuration>
```

```
#####
##
```

2.2.4 修改 mapred-site.xml

```
<configuration>
<!-- 指定 mr 框架为 yarn 方式 -->
<property>
<name>mapreduce.framework.name</name>
<value>yarn</value>
</property>
</configuration>
```

```
#####
##
```

2.2.5 修改 yarn-site.xml

```
<configuration>
<!-- 开启 RM 高可用 -->
<property>
<name>yarn.resourcemanager.ha.enabled</name>
<value>true</value>
</property>
<!-- 指定 RM 的 cluster id -->
<property>
<name>yarn.resourcemanager.cluster-id</name>
<value>yrc</value>
</property>
<!-- 指定 RM 的名字 -->
<property>
<name>yarn.resourcemanager.ha.rm-ids</name>
<value>rm1,rm2</value>
</property>
<!-- 分别指定 RM 的地址 -->
<property>
<name>yarn.resourcemanager.hostname.rm1</name>
<value>node-1</value>
</property>
<property>
<name>yarn.resourcemanager.hostname.rm2</name>
<value>node-2</value>
```



```

</property>
<!-- 指定 zk 集群地址 -->
<property>
<name>yarn.resourcemanager.zk-address</name>
<value>node-1:2181,node-2:2181,node-3:2181</value>
</property>
<property>
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>
</configuration>

```

2.2.6 修改 slaves(slaves 是指定子节点的位置，因为要在 hadoop01 上启动 HDFS、在 hadoop03 启动 yarn，所以 hadoop01 上的 slaves 文件指定的是 datanode 的位置，hadoop03 上的 slaves 文件指定的是 nodemanager 的位置)

hadoop05

hadoop06

hadoop07

2.2.7 配置免密码登陆

#首先要配置 hadoop00 到 hadoop01、hadoop02、hadoop03、hadoop04、hadoop05、hadoop06、hadoop07 的免密码登陆

#在 hadoop01 上生产一对钥匙

ssh-keygen -t rsa

#将公钥拷贝到其他节点，****包括自己****

ssh-copy-id hadoop00

ssh-copy-id hadoop01

ssh-copy-id hadoop02

ssh-copy-id hadoop03

ssh-copy-id hadoop04

ssh-copy-id hadoop05

ssh-copy-id hadoop06

ssh-copy-id hadoop07

#注意：两个 namenode 之间要配置 ssh 免密码登陆 ssh 远程补刀时候需要

###注意：严格按照下面的步骤!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

2.5 启动 zookeeper 集群（分别在 hadoop05、hadoop06、hadoop07 上启动 zk）

bin/zkServer.sh start

```
#查看状态：一个 leader，两个 follower
bin/zkServer.sh status
```

2.6 手动启动 journalnode（分别在在 hadoop05、hadoop06、hadoop07 上执行）

```
hadoop-daemon.sh start journalnode
#运行 jps 命令检验，hadoop05、hadoop06、hadoop07 上多了
```

JournalNode 进程

2.7 格式化 namenode

```
#在 hadoop00 上执行命令:
hdfs namenode -format
#格式化后会在根据 core-site.xml 中的 hadoop.tmp.dir 配置的目录下生
```

成一个 hdfs 初始化文件，

把 hadoop.tmp.dir 配置的目录下所有文件拷贝到另一台 namenode 节点

所在的机器

```
scp -r tmp/ hadoop02:/home/hadoop/app/hadoop-2.6.4/
```

```
##也可以这样，建议 hdfs namenode -bootstrapStandby
```

2.8 格式化 ZKFC(在 active 上执行即可)

```
hdfs zkfc -formatZK
```

2.9 启动 HDFS(在 hadoop00 上执行)

```
start-dfs.sh
```

2.10 启动 YARN

```
start-yarn.sh
还需要手动在 standby 上手动启动备份的 resourcemanager
yarn-daemon.sh start resourcemanager
```

到此，hadoop-2.6.4 配置完毕，可以统计浏览器访问：

```
http://hadoop00:50070
NameNode 'hadoop01:9000' (active)
http://hadoop01:50070
NameNode 'hadoop02:9000' (standby)
```

验证 HDFS HA

```
首先向 hdfs 上传一个文件
hadoop fs -put /etc/profile /profile
hadoop fs -ls /
然后再 kill 掉 active 的 NameNode
kill -9 <pid of NN>
```

通过浏览器访问: <http://192.168.1.202:50070>

NameNode 'hadoop02:9000' (active)

这个时候 hadoop02 上的 NameNode 变成了 active

在执行命令:

```
hadoop fs -ls /
```

```
-rw-r--r--  3 root supergroup      1926 2014-02-06 15:36 /profile
```

刚才上传的文件依然存在!!!

手动启动那个挂掉的 NameNode

```
hadoop-daemon.sh start namenode
```

通过浏览器访问: <http://192.168.1.201:50070>

NameNode 'hadoop01:9000' (standby)

验证 YARN:

运行一下 hadoop 提供的 demo 中的 WordCount 程序:

```
hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.4.1.jar  
wordcount /profile /out
```

OK, 大功告成!!!

测试集群工作状态的一些指令:

```
hdfs dfsadmin -report      查看 hdfs 的各节点状态信息
```

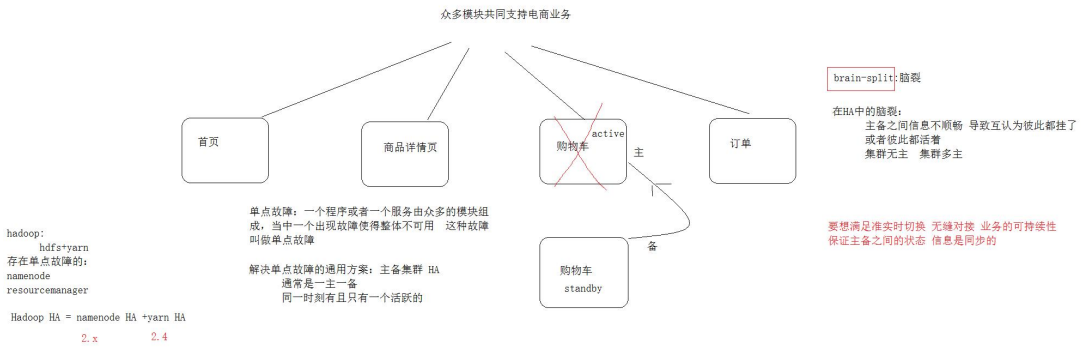
```
cluster1n/hdfs haadmin -getServiceState nn1      获取一个 namenode 节点的 HA 状态
```

```
cluster1n/hadoop-daemon.sh start namenode      单独启动一个 namenode 进程
```

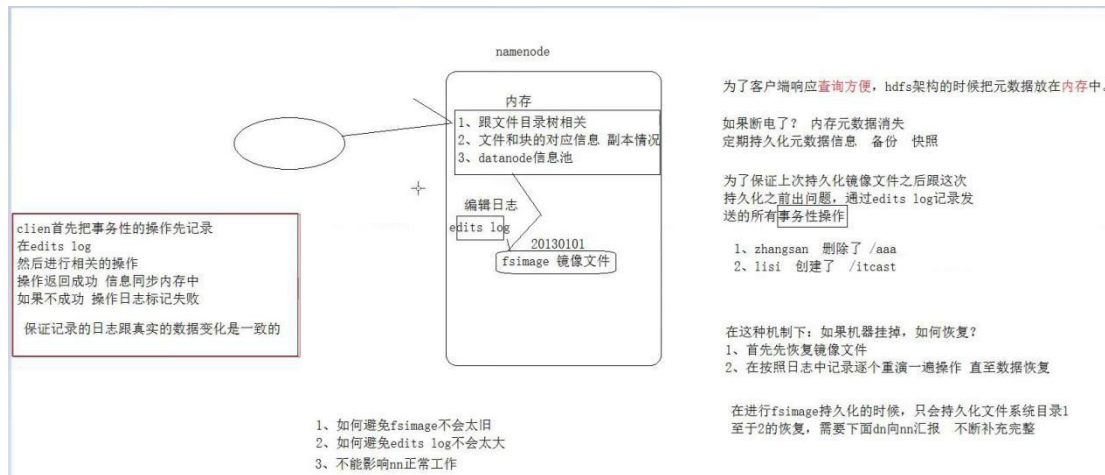
```
./hadoop-daemon.sh start zkfc      单独启动一个 zkfc 进程
```

2.9.2 上课用图

1) 单点故障与“脑裂”



2) client 的事务性操作对 HA 提供了支持



3) QJM

1、QJM是如何避免脑裂问题发生的？

zookeeper :

- zkfc:failover-controller
- 1、监控namenode的健康状态
- 2、可以通过zk主备切换

步骤：

有nn的地方就有zkfc进行监控
监控nn工作的软硬件环境 以及nn本身进程是否正常

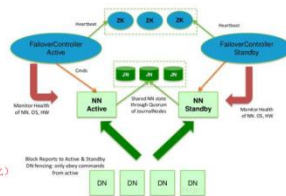
此外我们zkfc还会在zk集群上维护一个节点（短暂、非序列化）

两个zkfc到zk集群注册指定的节点（非序列化 短暂），谁创建成功，谁对应的这台机器上的namenode就是active
创建失败zkfc在此节点设置exist（node）的监听

如果active发现自己监控的nn出现了健康问题，断开自己跟zk的连接 之前所创建的节点消失 触发监听 standby对应的zkfc就会收到监听

standby nn并不会马上把自己切换为active，ssh到active的那台机器上进行远程登陆 补刀 kill -9 xxxx
补刀回来 再把自己的状态切换成active

如果之前的那个active又修复好了 启动之后首先它的zkfc去创建节点 但是节点存在 设置监听 把自己变成standby



2、如何保证主备之间的数据 状态是同步的？

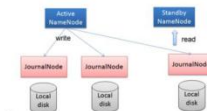
回想：首先写日志 操作成功同步内存

事实上 数据共享的同步的核心：共享操作日志 各跟着主重演一遍操作记录

核心：使用jn集群进行edits log的共享 2n+1 写过半成功即认为成功

步骤：

客户端连接active nn进行hdfs操作，首先把edits log写入到jn集群中，过半认为成功 standby感知jn集群数据的变化 读取edits log 重演操作记录 使得元数据能实时同步



2.9.3 3、面试题

1) hadoop 的 namenode 宕机,怎么解决？

先分析宕机后的损失，宕机后直接导致 client 无法访问，内存中的元数据丢失，但是硬盘中的元数据应该还存在，如果只是节点挂了，重启即可，如果是机器挂了，重启机器后看节点是否能重启，不能重启就要找到原因修复了。但是最终的解决方案应该是在设计集群的初期就考虑到这个问题，做 namenode 的 HA。

2) Hadoop 节点分布

集群部署节点角色的规划（7节点）

server01	namenode	zkfc		
server02	namenode	zkfc		
server03	resourcemanager			
server04	resourcemanager			
server05	datanode	nodemanager	zookeeper	journal node
server06	datanode	nodemanager	zookeeper	journal node
server07	datanode	nodemanager	zookeeper	journal node

集群部署节点角色的规划（3节点）

server01	namenode	resourcemanager	zkfc	nodemanager	datanode	zookeeper	journal node
server02	namenode	resourcemanager	zkfc	nodemanager	datanode	zookeeper	journal node
server03	datanode	nodemanager	zookeeper	journal node			

2.10 Hadoop 的联邦机制

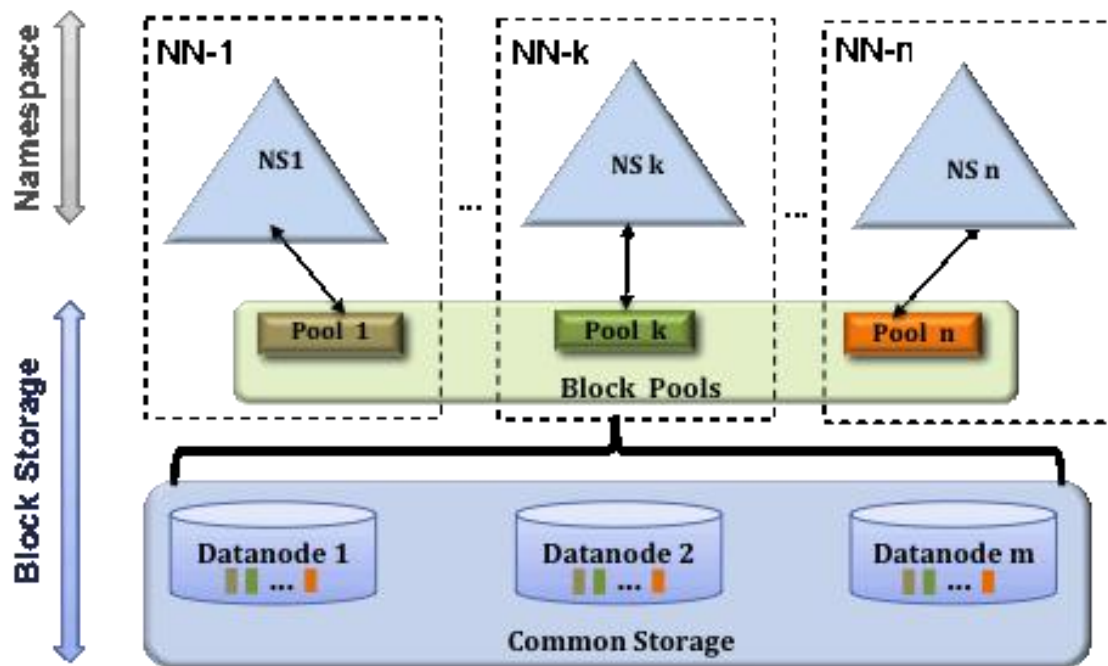
2.10.1 为什么会出现联邦？

Hadoop 的 NN 所使用的资源受所在服务的物理限制，不能满足实际生产需求。

2.10.2 联邦的实现

采用多台 NN 组成联邦。NN 是独立的，NN 之间不需要相互调用。NN 是联合的，同属于一个联邦，所管理的 DN 作为 block 的公共存储。

如下图：



图中概念：

- block pool 的概念，每一个 namespace 都有一个 pool，datanodes 会存储集群中所有的 pool，block pool 之间的管理是独立的，一个 namespace 生成一个 block id 时不需要跟其它 namespace 协调，一个 namenode 的失败也不会影响到 datanode 对其它 namenodes 的服务。
- 一个 namespace 和它的 block pool 作为一个管理单元，删除后，对应于 datanodes 中的 pool 也会被删除。集群升级时，这个管理单元也独立升级。
- 这里引入 clusterID 来标示集群所有节点。当一个 namenode format 之后，这个 id 生成，集群中其它 namenode 的 format 也用这个 id。

2.10.3 主要优点：

- 命名空间可伸缩性——联合添加命名空间水平扩展。DN 也随着 NN 的加入而得到拓展。
- 性能——文件系统吞吐量不是受单个 Namenode 限制。添加更多的 Namenode 集群扩展文件系统读/写吞吐量。
- 隔离——隔离不同类型的程序，一定程度上控制资源的分配

2.10.4 配置：

联邦的配置是向后兼容的，允许在不改变任何配置的情况下让当前运行的单节点环境转换成联邦环境。新的配置方案确保了在集群环境中的所有节点的配置文件都是相同的。

这里引入了 NameServiceID 概念，作为 namenodes 们的后缀。

第一步：配置属性 dfs.nameservices，用于 datanodes 们识别 namenodes。

第二步：为每个 namenode 加入这个后缀。

conf/hdfs-site.xml

```
<configuration>
  <property>
    <name>dfs.nameservices</name>
    <value>ns1,ns2</value>
  </property>
  <property>
    <name>dfs.namenode.rpc-address.ns1</name>
    <value>nn-host1:rpc-port</value>
  </property>
  <property>
    <name>dfs.namenode.http-address.ns1</name>
    <value>nn-host1:http-port</value>
  </property>
  <property>
    <name>dfs.namenode.secondary.http-address.ns1</name>
    <value>snn-host1:http-port</value>
  </property>
  <property>
    <name>dfs.namenode.rpc-address.ns2</name>
    <value>nn-host2:rpc-port</value>
  </property>
  <property>
    <name>dfs.namenode.http-address.ns2</name>
    <value>nn-host2:http-port</value>
  </property>
</configuration>
```



```

<property>
  <name>dfs.namenode.secondary.http-address.ns2</name>
  <value>snn-host2:http-port</value>
</property>

.... Other common configuration ...
</configuration>

```

2.10.5 操作

```

# 创建联邦，不指定 ID 会自动生成
$HADOOP_HOME/bin/hdfs namenode -format [-clusterId <cluster_id>]
# 升级 Hadoop 为集群
$HADOOP_HOME/bin/hdfs start namenode --config $HADOOP_CONF_DIR -upgr
ade -clusterId <cluster_ID>
# 扩展已有联邦
$HADOOP_HOME/bin/hdfs dfsadmin -refreshNamenodes <datanode_host_nam
e>:<datanode_rpc_port>
# 退出联邦
$HADOOP_HOME/sbin/distribute-exclude.sh <exclude_file>
$HADOOP_HOME/sbin/refresh-namenodes.sh

```

什么是 CDH 下载地址：<http://archive.cloudera.com/cdh5/cdh/5/>

CDH (Cloudera's Distribution, including Apache Hadoop), 是 Hadoop 众多分支中的一种，由 Cloudera 维护，基于稳定版本的 Apache Hadoop 构建，并集成了很多补丁，可直接用于生产环境。

CDH 的优点：

- 版本划分清晰
- 版本更新速度快
- 支持 Kerberos 安全认证
- 文档清晰
- 支持多种安装方式（Cloudera Manager、YUM、RPM、Tarball）

什么是 CM

Cloudera Manager 是为了便于在集群中进行 Hadoop 等大数据处理相关的服务安装和监控管理的组件，对集群中主机、Hadoop、Hive、Spark 等服务的安装配置管理做了极大简化。

Cloudera Manager 有四大功能：

- (1) 管理：对集群进行管理，如添加、删除节点等操作。
- (2) 监控：监控集群的健康情况，对设置的各种指标和系统运行情况进行全面监控。
- (3) 诊断：对集群出现的问题进行诊断，对出现的问题给出建议解决方案。

(4) 集成：对 hadoop 的多组件进行整合

2.11 项目

2.11.1 点击流日志模型

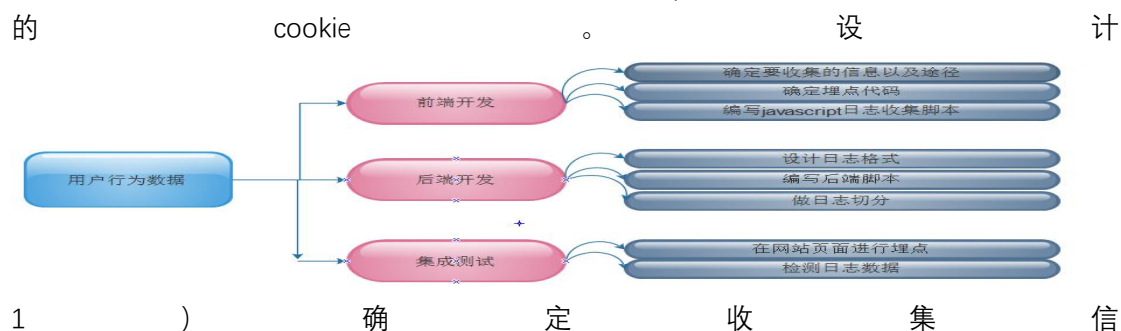
1 分析什么

点击流日志模型主要是分析用户在网站上持续访问的轨迹，查看网站的访问量，分析关键路径得到漏斗模型，以此来优化网站，提高用户的网站停留时间。

2 确定数据源

在本次案例中，使用埋点的方式收集数据，埋点是指在网页中插入小段 javascript 代码，这个代码片段会动态创建一个 script 标签，并将 src 属性指向一个单独的 js 文件，此时这个单独的 js 文件会被浏览器请求到并执行，这个 js 往往就是真正的数据收集脚本。

数据收集完成后，js 会请求一个后端的数据收集脚本，这个脚本一般是一个伪装成图片的动态脚本程序，js 会将收集的数据通过 http 参数的方式传递给后端脚本，后端脚本解析参数并按固定格式记录到访问日志，同时可能会在 http 响应中给客户端种植一些用于追踪的



名称	途径	备注
访问时间	web server	Nginx \$msec
IP	web server	Nginx \$remote_addr
域名	javascript	document.domain
URL	javascript	document.URL
页面标题	javascript	document.title
分辨率	javascript	window.screen.height & width
颜色深度	javascript	window.screen.colorDepth
Referrer	javascript	document.referrer
浏览客户端	web server	Nginx \$http_user_agent
客户端语言	javascript	navigator.language
访客标识	cookie	Nginx \$http_cookie
网站标识	javascript	自定义对象
状态码	web server	Nginx \$status
发送内容量	web server	Nginx \$body_bytes_sent

1) 确定埋点代码

```

<script type="text/javascript">
    val _map=_map || [];
    _map.push(['_setAccount',' UA-XXXXX-X']);

    //自调用匿名函数只会在运行时执行一次，一般用于初始化
    (function() {
        //引入一个外部的 js 文件 ma
        var ma = document.createElement('script');
        ma.type = 'text/javascript';
        //异步调用外部 js 文件
        ma.async = true;
        //根据协议将 src 指向对应的 ma.js
        ma.src = ('https:' == document.location.protocol ?
            'https://ssl' : 'http://www') + '.google-analytics.com/ma.js';
        //将这个元素插入到 dom 树上
        var s = document.getElementsByTagName('script')[0];
        s.parentNode.insertBefore( ma, s);
    })();

```

</script>

2) 前端收集脚本

在第二步配置中的 ma.js 被请求后会被执行，一般要做如下几件事：

E、通过浏览器内置 js 对象收集信息

F、解析_map 数组，收集配置信息，这里面可能会包括用户自定义的事件跟踪，业务数据等

G、将上面两步收集的数据按预定义格式解析并拼接

H、请求一个后端脚本，将信息放在 http request 参数中携带给后端脚本

示例代码

```
(function(){
    var params={};
    //document 对象数据，实现了 A 步骤
    if(document){
        params.domain=document.domain||'';
        params.url=document.URL||'';
        params.title=document.title||'';
        params.referrer=document.referrer||'';
    }
    //window 对象数据
    if(window&&window.screen){
        params.sh=window.screen.height||0;
        params.sw=window.screen.width||0;
        params.cd=window.screen.colorDepth||0;
    }
    //navigator 对象数据
    if(navigator){
        params.lang=navigator.language||'';
    }
    //接卸 map 数组。收集配置信息
    if(_maq){
        for(var i in _maq){
            switch(_map[i][0]){
                case '_setAccount':
                    params.account=_maq[i][1];
                    break;
                default:
                    break;
            }
        }
    }
    //拼接参数串
    var args='';
    for(var i in params){
        if(ags!='')args+='&';
```

```

        args += i + '=' + encodeURIComponent(params[i]);
    }
    //通过 image 对象请求后端脚本
    var img=new Image(1,1);
    //请求后端脚本常用的是 ajax，但是 ajax 是不能跨域请求的，通用的方
    //法是 js 脚本创建一个 image 对象，将 image 对象的 src 属性指向后端脚
    //本并携带参数,就实现了跨域请求。
    img.src='http://xx.xxx.xxx/log.gif?' + args;
})();

```

后端脚本

log.gif 是后端脚本，是一个伪装成 gif 图片的脚本，后端脚本一般需要完成以下几件事情

- F、解析 http 请求参数得到信息
- G、从 web 服务器获取一些客户端无法获取的信息
- H、将信息按格式写入 log
- I、生成一副 1*1 的 gif 图片作为响应内容并将响应头的 content-type 设置为 image/gif
- J、在响应头中通过 set-cookie 设置一些需要的 cookie 信息

我们使用 nginx 的 access_log 做日志收集

首先，需要在 nginx 的配置文件中定义日志格式：

```

log_format tick
"$msec|$remote_addr|$status|$body_bytes_sent|$u_domain|$u_url|
|$u_title|$u_referrer|$u_sh|$u_sw|$u_cd|$u_lang|$http_user_ag
ent|$u_account";

```

注意这里以 u_开头的是我们待会自己定义的变量，其它的是 nginx 内置变量。然后是核心的两个 location：

```

location / log.gif {
#伪装成 gif 文件
default_type image/gif;
#本身关闭 access_log，通过 subrequest 记录 log
access_log off;
access_by_lua "
-- 用户跟踪 cookie 名为 __utrace
local uid = ngx.var.cookie__utrace
if not uid then
-- 如果没有则生成一个跟踪 cookie，算法为
md5(时间戳+IP+客户端信息)
uid = ngx.md5(ngx.now() ..
ngx.var.remote_addr .. ngx.var.http_user_agent)
end
ngx.header['Set-Cookie'] = {'__utrace=' .. uid ..
'; path='/}

```

```

if ngx.var.arg_domain then
-- 通过 subrequest 子请求 到/i-log 记录日志,
将参数和用户跟踪 cookie 带过去
ngx.location.capture('/i-log?' ..
ngx.var.args .. '&utrace=' .. uid)
end
",
#此请求资源本地不缓存
add_header Expires "Fri, 01 Jan 1980 00:00:00 GMT";
add_header Pragma "no-cache";
add_header Cache-Control "no-cache, max-age=0, must-
revalidate";
#返回一个 1×1 的空 gif 图片
empty_gif;
}
location /i-log {
#内部 location, 不允许外部直接访问
internal;
#设置变量, 注意需要 unescape, 来自 ngx_set_misc 模块
set_unescape_uri $u_domain $arg_domain;
set_unescape_uri $u_url $arg_url;
set_unescape_uri $u_title $arg_title;
set_unescape_uri $u_referrer $arg_referrer;
set_unescape_uri $u_sh $arg_sh;
set_unescape_uri $u_sw $arg_sw;
set_unescape_uri $u_cd $arg_cd;
set_unescape_uri $u_lang $arg_lang;
set_unescape_uri $u_account $arg_account;
#打开日志
log_subrequest on;
#记录日志到 ma.log 格式为 tick
access_log /path/to/logs/directory/ma.log tick;
#输出空字符串
echo "";
}

```

日志切分

日志收集系统访问日志时间一长文件变得很大, 而且日志放在一个文件不便于管理。通常要按时间段将日志切分, 例如每天或每小时切分一个日志。通过 crontab 定时调用一个 shell 脚本实现, 如下:

```

_prefix="/path/to/nginx"
time=`date +%Y%m%d%H`
mv ${_prefix}/logs/ma.log ${_prefix}/logs/ma/ma-${time}.log
kill -USR1 `cat ${_prefix}/logs/nginx.pid`

```

这个脚本将 ma.log 移动到指定文件夹并重命名为 ma-{yyyymmddhh}.log,

然后向 nginx 发送 USR1 信号令其重新打开日志文件。

USR1 通常被用来告知应用程序重载配置文件，向服务器发送一个 USR1 信号将导致以下步骤的发生：停止接受新的连接，等待当前连接停止，重新载入配置文件，重新打开日志文件，重启服务器，从而实现相对平滑的不关机的更改。

cat \${_prefix}/logs/nginx.pid 取 nginx 的进程号

然后再/etc/crontab 里加入一行：

```
59 * * * * root /path/to/directory/rotatelog.sh
```

在每个小时的 59 分启动这个脚本进行日志轮转操作。

3 确定采集方案

在本次案例中，使用 Flume 日志采集系统来采集数据到 HDFS 系统。

1) 在服务器上部署 agent 节点，修改配置文件

配置文件如下

```
a1.sources=r1
```

```
a1.sinks=k1
```

```
a1.channels=c1
```

```
//监视指定的一些文件，将近实时的 tail 这些文件，获取这些文件的新追加的行
```

```
a1.sources.r1.type=    TAILDIR
```

```
a1.sources.r1.channels=c1
```

```
//配置检查点文件的路径，检查点文件会以 json 格式保存已经 tail 文件的位置，解决了断点不能续传的缺陷
```

```
a1.sources.r1.positionFile=/export/log/flume/taildir_position.json
```

```
//可以指定要 tail 的组
```

```
a1.sources.r1.filegroups=f1 f2
```

```
//指定每个文件的全路径
```

```
a1.sources.r1.filegroups.f1=/export/log/test1/example.log
```

```
a1.sources.r1.filegroups.f2=/export/log/test2/*.log.*
```

```
//配置 channels
```

```
a1.channels.c1.type=memory
```

```
a1.channels.c1.capacity=1000
```

```
a1.channels.c1.transactionCapacity=100
```

```
//配置下沉点，将文件写到 hdfs 中去
```

```
a1.sinks.k1.type=hdfs
```

```
a1.sinks.k1.channel=c1
```

```
a1.sinks.k1.hdfs.path=/flume/events/%y-%m-%d/
```

```
a1.sinks.k1.hdfs.filePrefix=events-
```

```
a1.sinks.k1.hdfs.round=true
```

```
a1.sinks.k1.hdfs.roundValue=10
```

```
a1.sinks.k1.hdfs.roundUnit=minute
```

4 确定三层架构

ETL 工作的实质就是从各个数据源提取数据，对数据进行转换，并最终加载填充数据到数据仓库维度建模后的表中。

1) 创建 ODS 层数据表

D、创建原始数据表

```
drop table if exists ods_weblog_origin;
create table ods_weblog_origin(
    valid string,
    remote_addr string,
    remote_user string,
    time_local string,
    request string,
    status string,
    body_bytes_sent string,
    http_referer string,
    http_user_agent string)
partitioned by (datestr string)
row format delimited
fields terminated by '\001';
```

E、点击流模型 pageviews 表

```
drop table if exists ods_weblog_origin;
create table ods_weblog_origin(
    valid string,
    remote_addr string,
    remote_user string,
    time_local string,
    request string,
    visit_step string,
    page_staylong string,
    http_referer string,
    http_user_agent string,
    body_bytes_sent string,
    status string)
partitioned by(datestr string)
row format delimited fields terminated by '\001';
```

F、点击流 visit 模型

```
drop table if exists ods_click_stream_visit;
create table ods_click_stream_click(
    session string,
```



```

remote_addr string,
inTime string,
outTime string,
inPage string,
outPage string,
referral string,
pageVisits int)
partitioned by (datastr string)
row format delimited fields terminated by '\001';

```

2) 导入 ODS 层数据

```

load data inpath '/weblog/preprocessed/' overwrite into table
ods_weblog_origin partition(datastr='20180526');--数据导入
show partitions ods_weblog_origin;--查看分区

```

3) 生成 ODS 层明细宽表

5 模块开发---统计分析

流量分析

D、多维度统计 PV 总量 按时间维度

(4) 计算每小时的 pv 数

```

select count(*) as pvs ,month,day,hour from ods_weblog_detail
group by month,day,hour;

```

(5) 计算该处理批次一天中的各个小时 pvs

```

drop table dw_pvs_everyhour_oneday;
create table dw_pvs_everyhour_oneday(month string,day
string,hour string ,pvs bigint) partitioned by(datestr);
insert into table dw_pvs_everyhour_oneday
partition(datestr="20180526") select a.month as month,a.day as
a.day,a.hour as hour,a.count(*) as pvs from ods_weblog_detail
where a.datastr='20180526' group by a.month,a.day,a.hour;

```

(6) 计算每天的 pvs

```

drop table dw_pvs_everyday;
create table dw_pvs_everyday(pvs bigint,month string,day string);
insert into table dw_pvs_everyday select count(*) as pvs,a.month

```

```
as month,a.day as day from ods_weblog_detail a group by
a.month,a.day;
```

E、人均浏览量

需求：统计今日所有来访者平均请求的页面数

计算方式：总页面请求数/去重总人数

remote_addr 表示不同的用户，可以先统计出不同的 remote_addr 的 pv 量，然后累加所有 pv 作为总的页面请求数，再 count 所有 remote_addr 作为总的去重总人数。

--总页面请求数/去重总人数

```
drop table dw_avgpv_user_everyday;
```

```
create table dw_avgpv_user_everyday(
    day string,
    avgpv string
);
```

```
Insert into table dw_avgpv_user_everyday
```

```
select '20180526',sum(b.pvs)/count(b.remote_addr)from (select
remote_addr, count(1) as pvs from ods_weblog_detail where
datestr='20180526' group by remote_addr)b;
```

F、统计 pv 总量最大的来源

统计每个小时各来访 host 的产生 pv 数最多的前 N 个 (topN)

```
drop table dw_pvs_refhost_topn_everyhour;
```

```
create table dw_pvs_refhost_topn_everyhour(
```

```
hour string,toporder string,ref_host string,ref_host_cnts string)
partitioned by (datestr string);
```

```
insert into dw_pvs_refhost_topn_everyhour
```

```
select ref_host,ref_host_cnts,concat(month,day,hour),row_number()
over(partition by concat(month,day,hour) order by ref_host_cnts desc)as
od from dw_pvs_refererhost_everyhour where od<=3;
```

关键路径转化率分析（漏斗模型）

A、需求分析

在一条指定的业务流程中，各个步骤的完成人数及相对上一个步骤的百分比

B、模型设计

定义好业务流程中的页面标识，下例中的步骤为：

C、模型设计

查询每一个步骤的总访问人数

```
Create table dw_oute_numbs as
```

```
select 'step1' as step ,count(distinct remote_addr) as numbs from ods_clik_pageviews where
datestr='20180526' and request like '/item%'
```

```
union
```

```
select 'step2' as step,count(distinct remote_addr) as numbs from ods_clik_pageviews where
datestr='20180526' and request like '/category%'
```

```

union
select 'step3' as step,count(distinct remote_addr) as numbs from ods_clik_pageviews where
datestr='20180526' and request like '/order%'
union
select 'step4' as step,count(distinct remote_addr) as numbs from ods_clik_pageviews where
datestr='20180526' and request like '/index%'
    查询每一步相对于路径起点人数的比例
    select tmp.rnstep,tmp.rnnumbs/tmp.rrnumbs as ratio from (select rn.step as
rnstep,rn.rnumbs as rnnumbs,rr.step as rrstep,rr.numbs as rrnumbs from dw_oute_numbs rn
inner join dw_out_numbs rr)tmp where tmp.rrstep='step1';
    查询每一步相对于上一步骤的漏出率
    Select tmp.rnstep,tmp.rnnumbs/tmp.rrnumbs from (select rn.step as
rnstep,rn.rnumbs as rnnumbs,rr.step as rrstep,rr.numbs as rrnumbs from dw_oute_numbs rn
inner join dw_out_numbs rr)where
cast(substr(tmp.rnstep,5,1),int)=cast(substr(tmp.rrstep,5,1),int)-1;

```

6 模块开发—结果导出

采用 apache sqoop 将数据导出

7 模块开发—数据可视化

采用 Echarts 对数据进行可视化展示。

2.11.2 相关面试题

03、 电商总体运营指标

04、 网站流量指标

2.11.3 推荐系统项目

推荐系统项目：本质上是商品售卖系统，和电商网站的目的一样，用来售卖商品。会收集用户的所有行为信息（网站浏览信息、订单信息、关注信息、收藏商品、评论系统、外部信息（微博信息、联盟网站）），通过分析分析用户的历史行为给用户的兴趣建模，从而主动给用户推荐能够满足他们兴趣和需求的信息。

数据源：

基础数据的来源有很多维度，包括用户的访问、浏览、下单、收藏、用户的历史订单数据，评价信息等很多数据。

基础数据主要包括：

4. 要推荐物品或内容的元数据。例如关键字、属性描述等
5. 系统用户的基本信息，例如性别，年龄等
6. 用户对物品或者信息的偏好，包括用户对物品的评分，用户查看物品的记录，用户购买记录等

可以将这些用户的偏好信息分为两类：

显示的用户反馈：这类是用户在网站上自然浏览或者使用网站以外，显式的提供反馈信息，例如用户对物品的评分，或者对物品的评论。

隐式的用户信息：用户在使用网站是产生的数据，隐式的反应了用户对物品的喜好，例如用户查看了某物品的信息等等用户反馈。

数据的采集方案：

数据收集模块：

点击流模块：flume+kafka+storm+redis 处理用户当前浏览的信息，将信息计算出来，保存到 redis 中（比如用户对一个品类的偏好）

订单支付（AMQ）：开发一个消费者程序（storm、JavaAPP），用来计算用户的偏好

外部数据：通过爬虫技术爬取用户的社交网站数据。

合作数据：比如京东和腾讯合作，可以获取一些用户在腾讯上的信息。

3. 将用户产生的数据（浏览商品、关注商品、收藏商品、加入购物车、下订单、评论等）保存到消息队列（kafka）中，将不同类型的数据保存到不同的 topic 中，以作分类。
4. 另一种方式就是将用户产生的数据保存到分库分表的数据库中

数据的存储与计算：

三．离线推荐

离线推荐计算数据的周期可以是一个月或者 15 天

6. 由 FTP 服务对推送的数据进行校验，将数据库中校验通过的数据定时的上传到 HDFS 文件系统中。
7. 基于 HDFS 建立 hive 数据仓库，将 HDFS 中的数据映射成一张张的表（用户表、收藏表、购物车表、订单表等）保存到 hive 数据仓库中。
8. 将 hive 数据仓库中的数据通过 HSQL 计算和导入到算法中（协同过滤算法）进行计算，得到一些用户的偏好数据。
9. 将计算出来的用户偏好数据定时导入到 hbase/redis 中。
10. 用户通过点击浏览商品，在推荐引擎（Javaweb）中获取用户 id，根据用户的 id 从 redis 或 hbase 中拿去已经计算好的推荐结果（离线计算）推荐给用户。在此过程中，还要对结果数据进行过滤，比如商品上下线状态的判定、推荐结果种类丰富性的判定、推荐结果数量的补足等。最后将计算过滤之后的商品展示给用户。

四．实时推荐

2. 通过 storm, 实时的消费消息队列 kafka 的 topic 中的数据, 实时的计算一些用户的偏好, 同样将计算好的偏好数据实时的导出到 hbase 或 redis 中, 等待用户点击浏览商品, 实时的将计算好的结果展示给用户。

大数据处理平台:

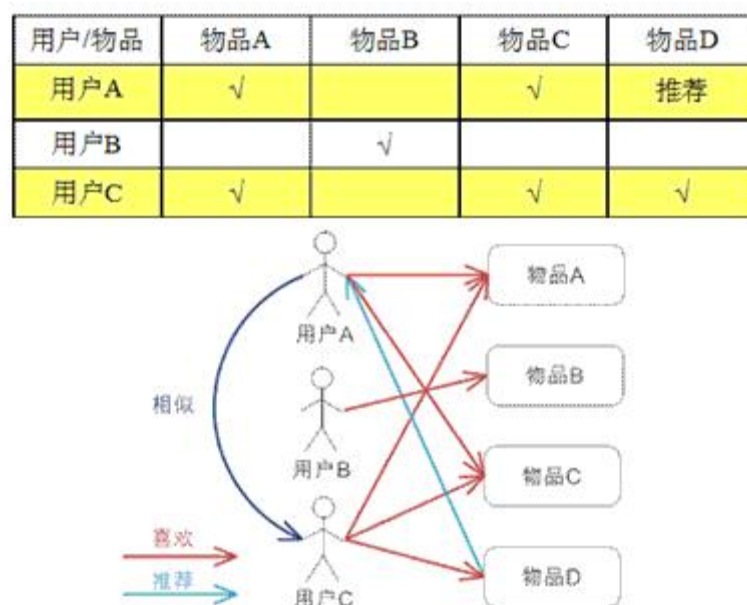
离线计算和实时计算

数据的计算:

协同过滤算法: 也就是计算相似度, 关于相似度的计算, 就是计算像个向量的距离, 距离越近相似度越大。

3. 基于用户的协同过滤算法

原理: 基于用户对物品的偏好找到相邻邻居用户, 然后将邻居用户喜欢的推荐给当前用户。



假如: 用户 A 喜欢物品 A、物品 C

用户 B 喜欢物品 B

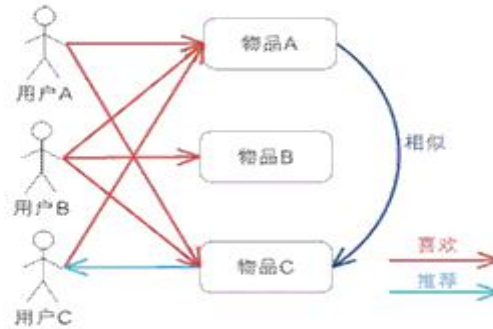
用户 C 喜欢物品 A、物品 C、物品 D

通过计算发现用户 A 和用户 C 相似, 所以将用户 C 喜欢的物品 D 推荐给用户 A。

4. 基于物品的协同过滤算法

原理: 与基于用户的类似, 只是在计算邻居时采用物品本身, 而不是从用户角度。基于用户对物品的偏好找到相似的物品, 然后根据用户的历史偏好, 推荐相似的物品给他。

用户/物品	物品A	物品B	物品C
用户A	√		√
用户B	√	√	√
用户C	√		推荐



如图：根据用户历史数据的偏好计算出喜欢物品 A 的用户大部分都喜欢物品 C，计算出物品 A 和物品 C 比较相似，而用户 C 喜欢物品 A，那么可以推断出用户 C 可能也喜欢物品 C，所以将物品 C 推荐给用户 C。

数据展现：

Web 网站

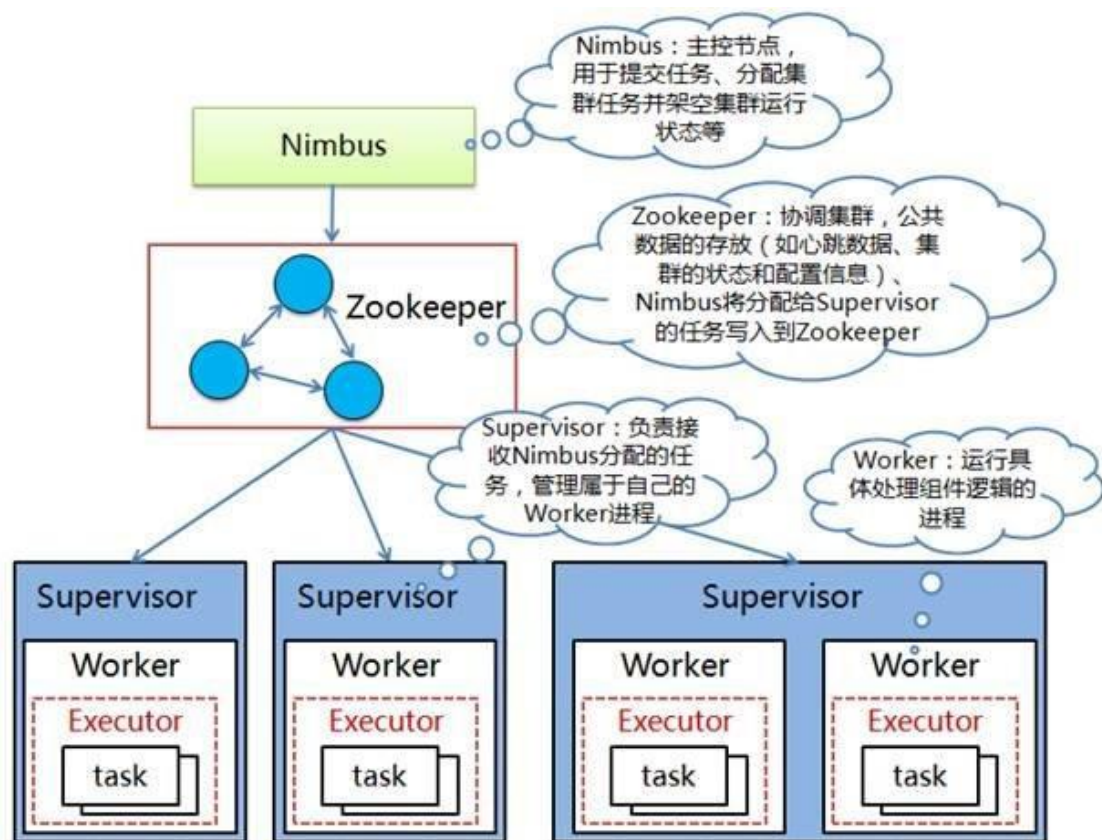
面试题：

- 7、推荐系统的本质是什么
- 8、推荐系统的指标是什么
- 9、推荐系统的 y 和 x
- 10、推荐系统的样本构造和数据拼接
- 11、推荐系统的场景思考
- 12、推荐系统相关组件

第 3 章 Storm

storm 是 twitter 公司开源贡献给 apache 的一款实时流式处理的一个开源软件，主要用于解决数据的实时计算以及实时的处理等方面的问题。Storm 的特点：编程模型简单，可扩展，高可靠性，高容错性，支持多种编程语言，支持本地模式，高效。

3.1 架构



3.1.1 Nimbus

负责资源分配和任务调度。新版本中的 nimbus 节点可以有多个，做主备

3.1.2 Zookeeper

协调集群，公共数据的存放（如心跳数据，集群的状态和配置信息），nimbus 将分配给 Supervisor 的任务写入到 Zookeeper

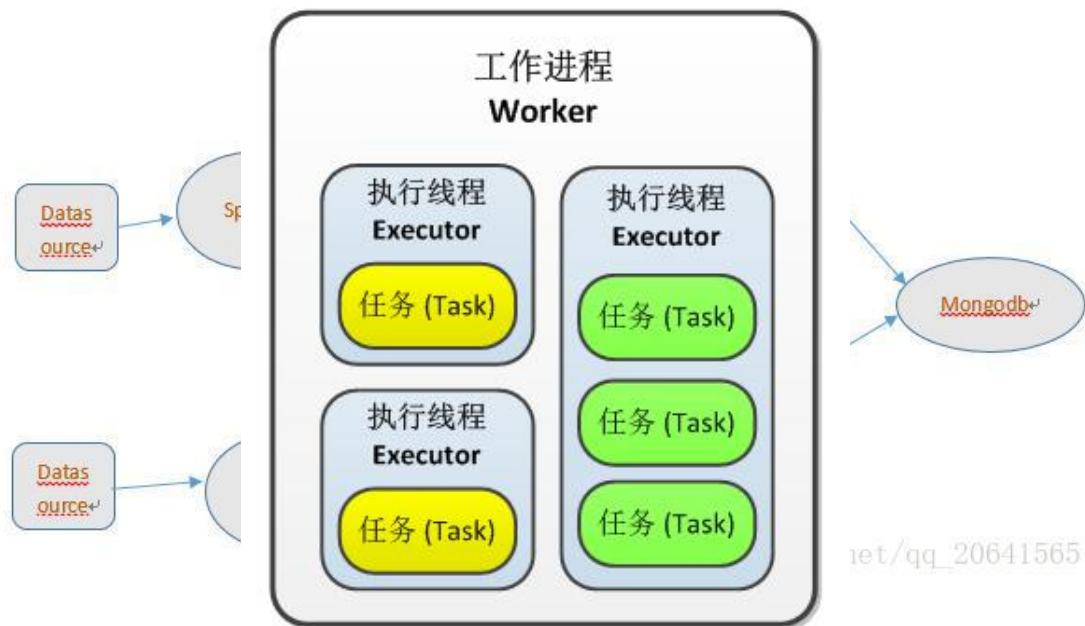
3.1.3 supervisor

负责接受 nimbus 分配的任务，启动和停止属于自己管理的 worker 进程。

3.1.4 worker

运行具体处理组件逻辑的进程。worker 中每一个 spout/bolt 的线程称为一个 task。在 storm0.8 之后，task 不再与物理线程对应，同一个 spout/bolt 的 task 可能会共享一个物理线程，该线程称为 executor。最新版本的 Jstorm 已经废除了 task 的概念

3.2 编程模型



3.2.1 Spout

Spout 是接受外部数据源的组件，将外部数据源转化成 Storm 内部的数据，以 Tuple 为基本的传输单元下发给 Bolt。（Tuple 是 Storm 内部中数据传输的基本单元，里面封装了一个 List 对象，用来保存数据。）

3.2.2 Bolt

Bolt 是接受 Spout 发送的数据，或上游的 bolt 的发送的数据。根据业务逻辑进行处理。发送给下一个 Bolt 或者是存储到某种介质上。介质可以是 mongodb 或 mysql，或者其他。

3.2.3 并行度

Worker：表示一个进程

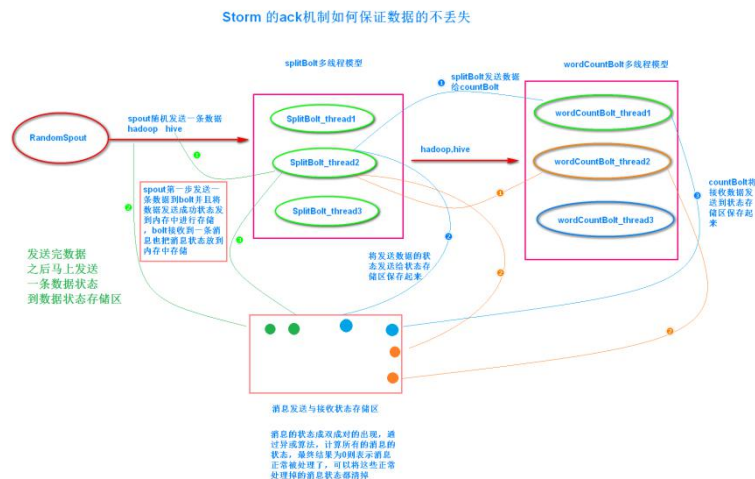
Executor：表示由 worker 启动的线程

Task：实际执行数据处理的最小工作单元（注意，task 并不是线程）

并行度的设置：评估上游 kafka 每秒生产的数据量，分析 topic 每个 partition 每秒的数据量，
partition 的数据量=SpoutTask 接受数据量 SpoutTask 数量=partition 的数量

Worker 的设置：如果数据量大，worker 的数量等于 spouttask 的数量

3.2.4 消息不丢失



ack 机制即， spout 发送的每一条消息，

- 在规定的时间内， spout 收到 Acker 的 ack 响应，即认为该 tuple 被后续 bolt 成功处理
- 在规定的时间内，没有收到 Acker 的 ack 响应 tuple，就触发 fail 动作，即认为该 tuple 处理失败，
- 或者收到 Acker 发送的 fail 响应 tuple，也认为失败，触发 fail 动作。

通过 Ack 机制， spout 发送出去的每一条消息，都可以确定是被成功处理或失败处理，从而可以让开发者采取动作。比如在 Meta 中，成功被处理，即可更新偏移量，当失败时，重复发送数据。因此，通过 Ack 机制，很容易做到保证所有数据均被处理，一条都不漏。

3.3 Storm 物联网 wifi 项目

3.3.1 wifi 项目背景

随着路由器上网的普及，越来越多的人在各个场合选择使用路由器上网，特别是在一些公共场所，例如网吧，酒店，饭店，旅馆，宾馆，洗浴中心等。这些公共场所的网络安全也日益受到各地网安的关注，各种问题也日益凸显。为此特为各地网安推出定制化的路由器，在网安指定的公共地点安装路由器，可以追踪

每个人的上网情况，通过路由器或者嗅探设备的 mac 地址以及经纬度的追踪，可以定位每个人员的上网大致方位，了解每个人的上网内容，做到实时的网页内容监控，地理位置的监控，上网设备的 mac 地址追踪，通过嗅探设备，实现上网设备的实时路线追踪，为各地网安解决各种定制化的任务。

网安对于公共场所上网的人群，突出关注以下几点：

1. 所有流经路由器人群的 mac 地址
2. 上网人群的虚拟身份记录
3. 上网上线记录
4. 搜索关键字记录
5. 网页访问记录
6. 地理位置记录

获取信息的条件:时速小于 90KM/小时,并且开启无线网,嗅探设备就能抓到你的 MAC 地址

3.3.2 2)数据来源(采集)

设备铺设场所: 四川遂宁 吉林白山 四川眉山 内蒙赤峰 山东青岛

线上设备数量 8000+台 各个地方的设备往各个网安后台系统上报数据,网安后台使用 ES+mysql 分表

上报数据的同时,通过 ftp 上报一份到我的大数据平台来,使用的是 ES+hbase 做实时查询功能,hive 整合 HBase 做批量统计功能

注意:关于物联网的项目,这些数据上报一般有两种方式.第一种手动上报,第二种定时上报,如果要做到实时处理数据,定时的时间肯定不长(几秒钟上传一次,每次上传数据量不大),这些数据要么通过 ftp 上传,要么通过 socket 接口传输数据

ftp 服务器是外网,大数据集群是内网(内网关闭防火墙)

五种数据类型,FTP 服务器开通多个 FTP 端口,多个 FTP 目录,用于接收多种设备,多种类型的数据,减轻 FTP 服务器的压力,避免数据的堆积

3.3.3 集群规模配置

每天 FTP 接收数据量大概在 100G 左右

数据存储要求:要求近一年的数据可以随时查询到

服务器集群规划: 12 台服务器组成的小型机集群

服务器硬件配置(最低,公司没钱):每台服务器 12T 硬盘配置

服务器内存配置:32G 内存 4 台, 16G 内存 8 台

网络:外网百兆独享网络,内网千兆交换机

服务器 CPU 配置: 8 核 16 线程 4 台 4 核 8 线程 8 台

注意:服务器如果超过了 50 台,就形成了大型机集群,20-50 台属于中型机集群

20 台以内属于小型机集群,服务器内存一般给 64G

3.3.4 数据处理流程

第一步:使用 flume 从 FTP 服务器拿到数据

第二步:flume 与 kafka 整合,数据收集到 kafka 中

第三步:storm 与 kafka 整合,读取 kafka 数据经过处理,匹配 redis 里的黑白名单,将数据存入实时报警监控系统中的 mysql 表中,并发送实时报警短信.

例:某一个 MAC 地址,或者手机号是 redis 黑名单里面的,哪个地区的 MAC 地址,手机号,

然后通过实时报警系统通知各地的网安

第四步:将处理完的数据整合进入 ES 与 HBase 当中,数据的关键信息存入 ES 集群中,具体的

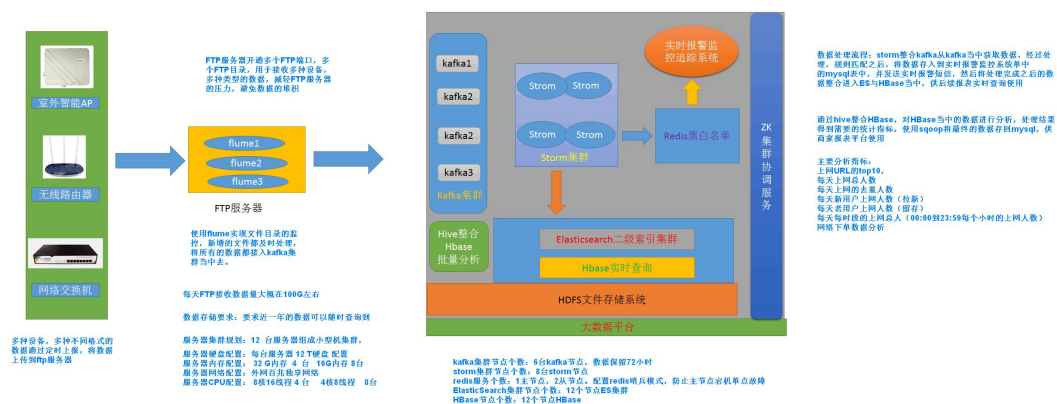
详细信息存入 Hbase 当中,供后续报表实时查询使用

注意:Hbase 条件查询比较弱,如果需要带上各种查询条件,只能使用协处理器,协处理器底层

全部是 MapReduce.

所以把需要查询的条件字段全部存到 ES(空间换时间)

第五步:hive 和 Hbase 整合,实现数据仓库,数据离线分析功能



Kafka 集群节点个数: 6 台 kafkfa 节点,数据保留 72 小时

Storm 集群节点个数:8 台 storm 节点

Redis 服务个数:1 主节点,2 从节点,配置 redis 哨兵模式,防止主节点宕机单点故障

ElasticSecarch 集群节点个数:12 个节点 ES 集群

Hbase 节点个数:12 个节点 Hbase

指标分析

由于与网安合作不挣钱,竞争激烈,转到与美团小商家合作

通过 hive 整合 Hbase,对 Hbase 当中的数据进行分析,处理结果得到需要的统计指标,使用

sqoop 将最终的数据存回 mysql,供商家报表平台使用

主要指标:

上网 URL 的 top10,

每天上网的总人数

每天上网的去重人数

每天新用户上网人数(拉新)

每天老用户上网人数(留存)

每天每时段的上网总人数(00:00 到 23:59 每个小时的上网人数)

网络下单数据分析

3.3.5 数据计算流程

kafkaSpout-----处理 kafka 中的数据

WifiErrorBolt-----把分隔符去掉,把脏数据过滤掉

WiliWarningBolt-----监控黑白名单,告警(没有名单,功能未实现)

WifiTypeBolt-----进行数据存储到 HDFS 上面去,要求实现五种数据类型按照长度分开存放

注意:数据写不到 HDFS 不同的目录,可以先写到本地文件系统,根据字段长度的不同,写到本地不同的文件目录里面去.

本地文件系统如何控制:

首先,写到 128M 的时候上传 HDFS

其次,启动定时任务每隔几秒扫描一下昨天的目录,如果昨天的目录里面有文件,上传 HDFS 对应的文件夹里面去

3.4 storm 实时看板案例

3.4.1 需求分析

平台在活动促销日 (例如双 11) 要求实时展示当日的一些销售信息

我们从三个维度去统计计算:

平台运维角度统计指标: 平台总销售额 平台下单人数 平台商品销售数量

商品销售角度统计指标： 每个商品总销售额 每个商品购买人数 每个商品销售数量

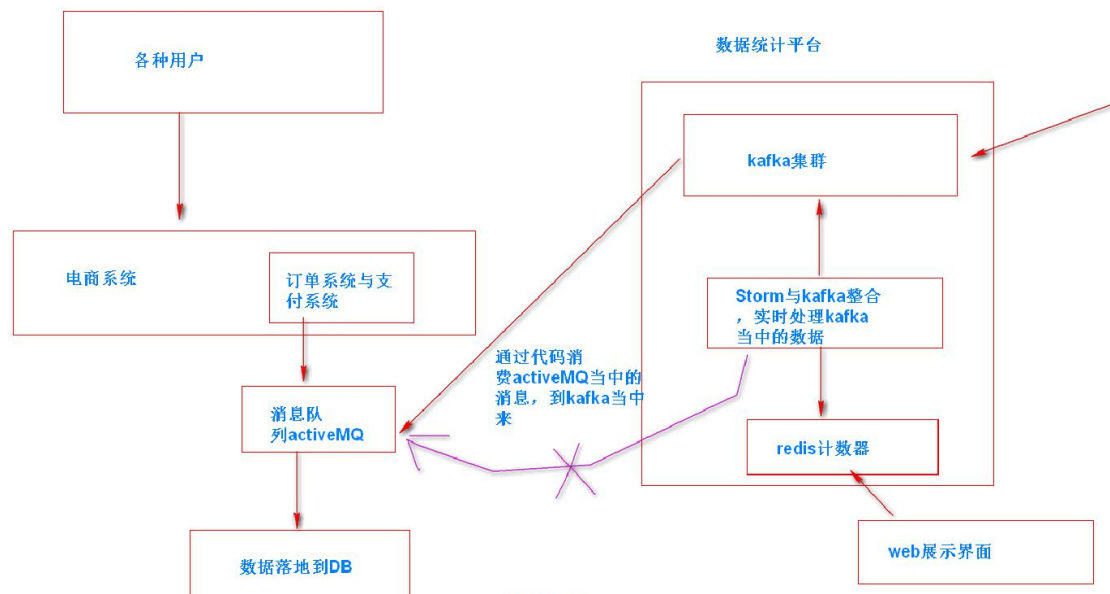
店铺销售角度统计指标： 每个店铺总销售额 每个店铺购买人数 每个店铺销售数量

3.4.2 确定数据源

数据源： 订单系统与支付系统产生的数据。

3.4.3 确定采集方案

订单系统和支付系统产生的数据 通过 ActiveMQ 到 kafka 中，storm 集成 kafka 获取数据。（注：为什么用 ActiveMQ?1.电商业务系统当中需要用到事务支持，只能用比较严谨的 jms 系统来实现，所以考虑到用 ActiveMQ。2.kafka 直接读取数据库，会影响数据库速度。所以订单系统的数据通过 ActiveMQ 这个中间件来落地到数据库，kafka 从 ActiveMQ 中间件中取数据，就可以避免直接读数据库的问题。如下图)



3.4.4 确定存储

我们将 storm 处理后的数据，通过 Redis 来累加计数并存储在 Redis 中。

3.4.5 数据计算

编写 storm 代码：storm 与 kafka 整合来获取到数据，将每一条数据信息，利用 Redis 中 **incrBy** 这个命令来累加计数。并存储在 Redis 中。

3.4.6 展现

可以写一个定时器，定时的去 Redis 中获取结果数据，展示到 web 界面上。

3.5 storm 日志监控告警系统

3.5.1 需求分析

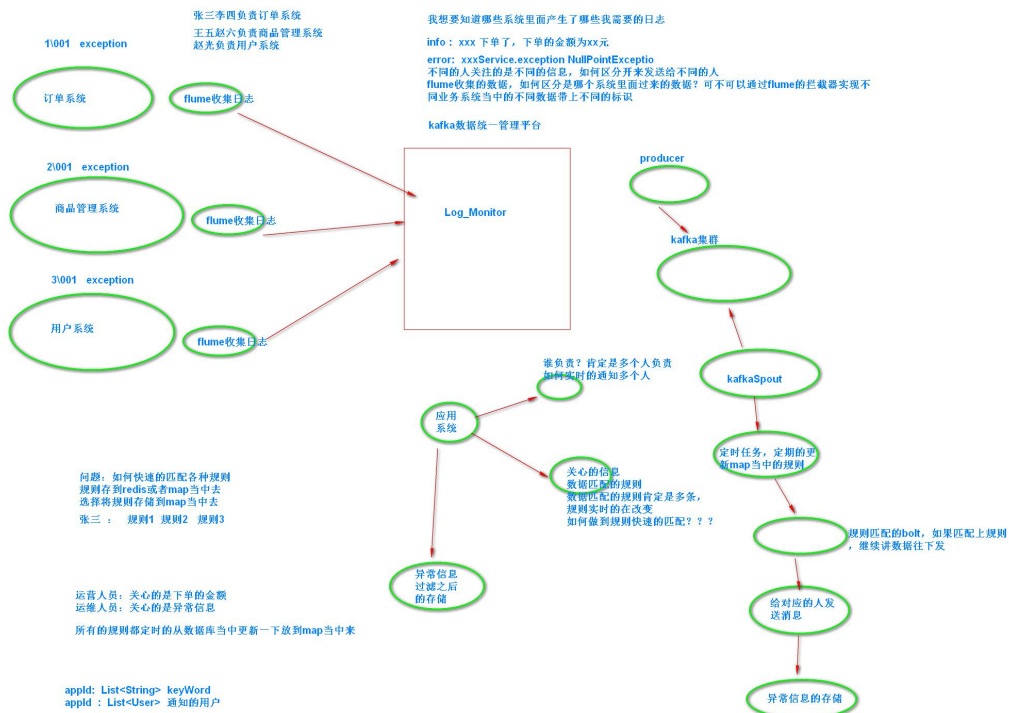
实现项目中日志监控的功能，需要做到日志监控实时告警，例如系统中出现任何异常，触发任何的告警规则，都可以实时通过短信或者邮件告知相关系统负责人。

3.5.2 确定数据源

每个业务系统（如：订单系统，商品管理系统，用户系统）产生的日志信息

3.5.3 确定采集方案

将日志数据 通过 flume 采集到 kafka 中， storm 集成 kafka 获取数据



3.5.4 确定存储

将触发告警规则的数据。存储在 MySQL 数据库中

3.5.5 数据计算

- 通过自定义 flume 拦截器, 给不同系统产生的日志数据前加上一个 appId 来做唯一标识(不同系统对应的告警规则和负责人不同, 所以这里要加一个唯一标识方便我们查找对应的规则和负责人)。
- storm 代码编写步骤:
 1. 获取 kafka 中的数据
 2. 设置定时器, 定时读取在 MySQL 数据库中的告警规则 (数据库中的告警规则我们可以随时去修改, 所以要采用定时器去读取, 保证拿到是最新的规则)
 3. 将获取到的数据, 与告警规则进行匹配, 得到匹配成功的告警数据信息
 4. 将告警信息 以邮件, 或者短信 方式发送给对应的负责人 (对应人的信息, 从 MySQL 数据库中获取) 。
 5. 最后将警告信息存储到我们的 MySQL 数据库中。(方便以后查询异常记录)

3.6 Storm 的框架

- 1) storm 的架构模型
- 2) storm 的编程模型
- 3) storm 的并行度
- 4) Storm 的分发策略
- 5) Storm 原理
- 6) storm 与 kafka 集成
- 7) storm 与 hdfs 的整合使用
- 8) 消息不丢失机制
- 9) storm 的定时器以及与 mysql 的整合使用
- 10) 日志监控告警系统

3.7 面试题

1) 公司技术选型可能利用 **storm** 进行实时计算,讲解一下 **storm**

描述下 storm 的设计模式，是基于 work、excutor、task 的方式运行代码，由 spout、bolt 组成等等

2) **storm** 如果碰上了复杂逻辑,需要算很长的时间,你怎么去优化,怎么保证实时性

拆分复杂的业务到多个 bolt 中，这样可以利用 bolt 的 tree 将速度提升

3) **Spark Streaming** 和 **Storm** 有何区别？

一个实时毫秒一个准实时亚秒，不过 storm 的吞吐率比较低。

第 4 章 Kafka

4.1 kafka 的介绍

1 什么是 kafka？



Apache Kafka 是一个开源消息系统，由 Scala 写成。是由 Apache 软件基金会开发的一个开源消息系统项目。

Kafka 最初是由 LinkedIn 开发，并于 2011 年初开源。2012 年 10 月从 Apache Incubator 毕业。该项目的目标是为处理实时数据提供一个统一、高通量、低等待的平台。

Kafka 是一个分布式消息队列：生产者、消费者的功能。它提供了类似于 JMS 的特性，但是在设计实现上完全不同，此外它并不是 JMS 规范的实现。

Kafka 对消息保存时根据 Topic 进行归类，发送消息者称为 Producer,消息接受者称为 Consumer,此外 kafka 集群有多个 kafka 实例组成，每个实例(server)成为 broker。

无论是 kafka 集群，还是 producer 和 consumer 都依赖于 **zookeeper** 集群保存一些 meta 信息，来保证系统可用性

2 kafka 与传统消息系统的区别

1、在架构模型方面，

RabbitMQ 遵循 AMQP 协议，RabbitMQ 的 broker 由 Exchange,Binding,queue 组成，

其中 exchange 和 binding 组成了消息的路由键；客户端 Producer 通过连接 channel 和 server 进行通信，Consumer 从 queue 获取消息进行消费（长连接，queue 有消息会推送到 consumer 端，consumer 循环从输入流读取数据）。rabbitMQ 以 broker 为中心；有消息的确认机制。

kafka 遵从一般的 MQ 结构，producer, broker, consumer，以 consumer 为中心，消息的消费信息保存的客户端 consumer 上，consumer 根据消费的点，从 broker 上批量 pull 数据；无消息确认机制。

2、在吞吐量，

kafka 具有高的吞吐量，内部采用消息的批量处理，zero-copy 机制，数据的存储和获取是本地磁盘顺序批量操作，具有 $O(1)$ 的复杂度，消息处理的效率很高。

rabbitMQ 在吞吐量方面稍逊于 kafka，他们的出发点不一样，rabbitMQ 支持对消息的可靠的传递，支持事务，不支持批量的操作；基于存储的可靠性的要求存储可以采用内存或者硬盘。

3、在可用性方面，

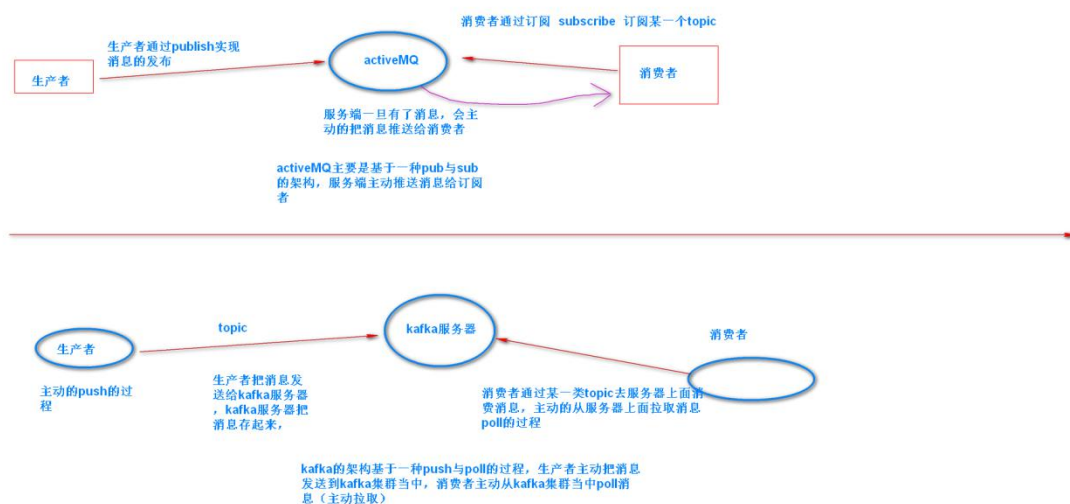
rabbitMQ 支持 miror 的 queue，主 queue 失效，miror queue 接管。

kafka 的 broker 支持主备模式。

4、在集群负载均衡方面

kafka 采用 zookeeper 对集群中的 broker、consumer 进行管理，可以注册 topic 到 zookeeper 上；通过 zookeeper 的协调机制，producer 保存对应 topic 的 broker 信息，可以随机或者轮询发送到 broker 上；并且 producer 可以基于语义指定分片，消息发送到 broker 的某分片上。

3 kafka 与 activeMQ 的区别



Topic:主题,即一个标识,类似于 map 里面的 key,通过它来给消息分类,消息根据 Topic 进行归类。

共同点:都有生产者和消费者两大组件,生产者发送消息给各自的服务器,(发送消息是就会定义一个 topic)并进行存储。

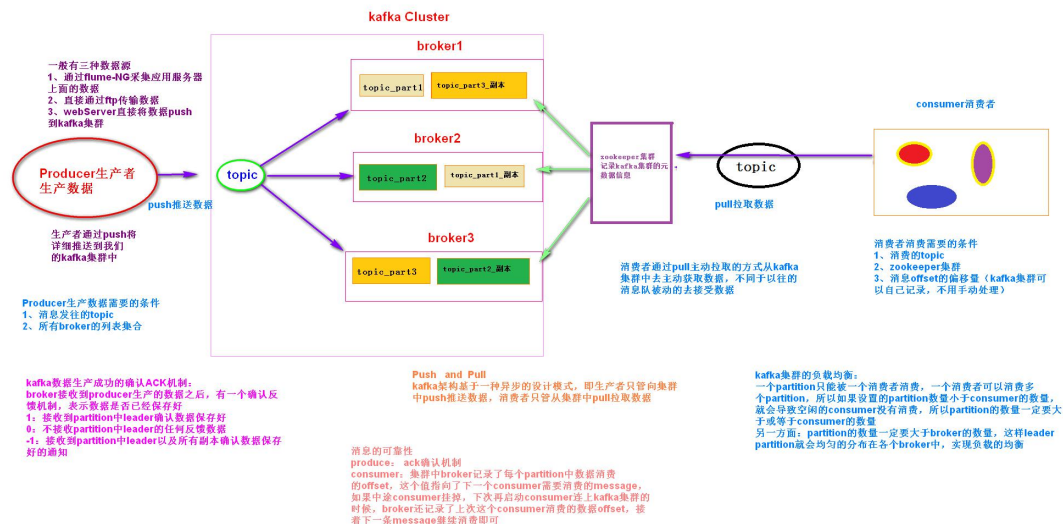
不同点:

activeMQ:消费者会提前订阅自己需要的 topic, 当该 topic 中有了消息以后,activeMQ 服务器会发送消息给消费者,然后消费者再去服务器中拿到自己想要的数据。

Kafka:消费者(指定 topic)会定时去 kafka 服务器中拿该 topic 中的数据。

4.2 kafka 的架构

基于 producer consumer topic broker 等的一个基本架构



kafka 的组件介绍

producer: 生产者, 主要用于我们的消息的生产, 通过 producer 将我们的消息 push 到 kafka 集群当中

topic: 某一类消息的高度抽象, 可以理解成某一类消息的集合, 一类消息, 每个 topic 将被分成多个 partition(区), 在集群的配置文件中配置。

broker: kafka 的服务器, 一个 broker 就代表一个服务器的节点

partition: 分区概念, 一个 topic 当中的消息, 可以拆分成多个 partition 分区, 存放在多个不同的服务器上, 实现数据存放的横向扩展

repliation: 副本, 所有的 partition 都可以指定存放几个副本, 做到数据的冗余, 保证数据的安全

segment: 每个 partiiton 由多个 segment 组成, segment 又包含了两部分, 一个.log 文件, 一个是.index 文件

.log: 存放我们的日志文件, 所有的数据, 最后都以日志文件的形式存放到了 kafka 集群当中

.index : 索引文件, 所有的.log 文件的索引都存放在这里, 便于我们查找某一条日志文件的快速

consumer: 消费者, 消费我们 kafka 集群当中的消息,

问题: 如何知道消费者消费到了哪一条消息来了???

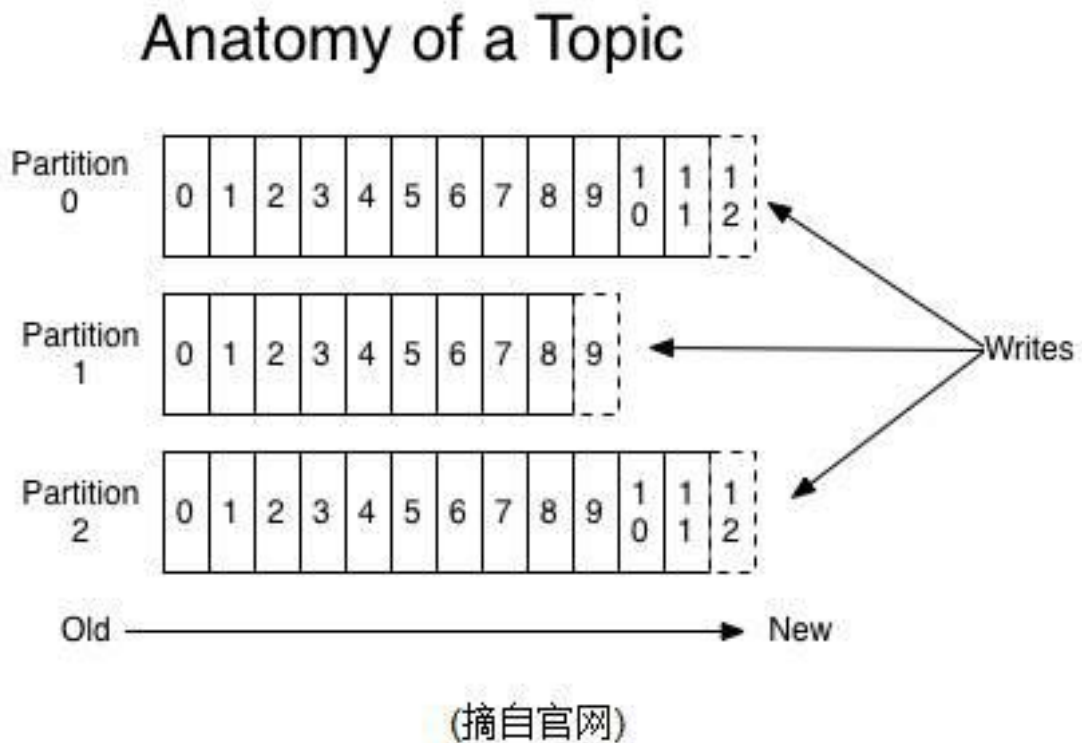
可以通过记录的方式, 记下来每次我们消费的位置

第一种记录方式：kafka 的本地文件系统，比较慢，对应 kafka 的一个慢速消费的方式

第二种记录方式：zookeeper 当中的节点数据记录，比较快，对应 kafka 的一个快速消费的方式

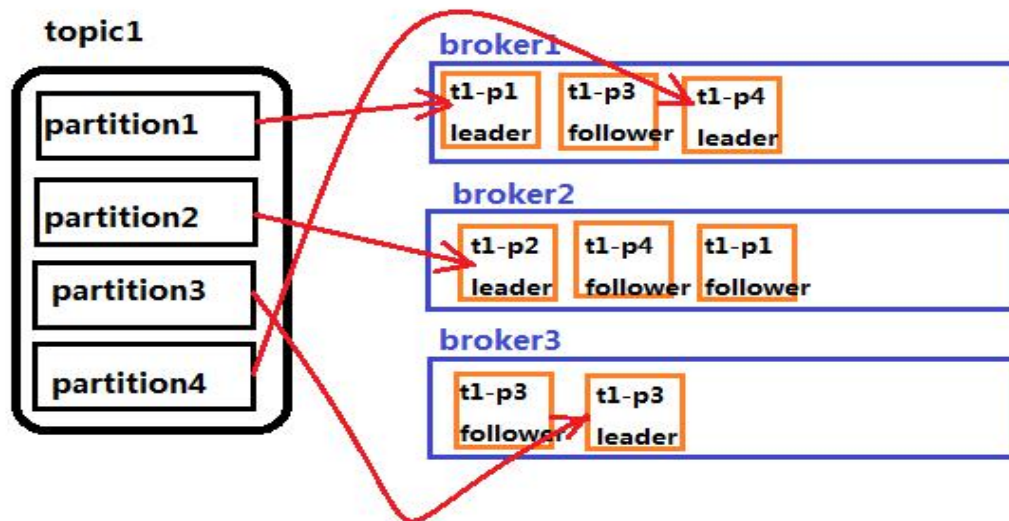
offset：偏移量，就是记录的我们消费到了哪一条数据来了。

发布者发到某个 topic 的消息会被均匀的分布到多个 part 上, broker 收到发布消息往对应 part 的最后一个 segment 上添加该消息。



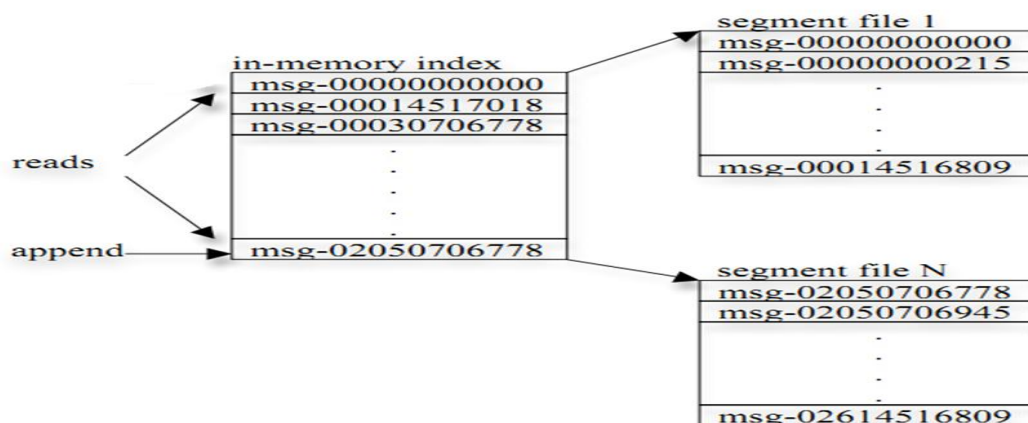
partition 分布

- 1、 partitions 分区到不同的 server 上，一个 partition 保存在一个 server 上，避免一个 server 上的文件过大，同时可以容纳更多的 consumer 消费,有效提升并发消费的能力。
- 2、 这个 server(如果保存的是 partition 的 leader)负责 partition 的读写。可以配置备份。
- 3、 每个 partition 都有一个 server 为"leader", 负责读写，其余的相对备份机为 follower，follower 同步 leader 数据，负责 leader 死了之后的接管。n 个 leader 均衡的分散在每个 server 上。
- 4、 partition 的 leader 和 follower 之间监控通过 zookeeper 完成。



segment

- 1、每个 segment 中存储多条消息，消息 id 由其逻辑位置决定，即从消息 id 可直接定位到消息的存储位置，避免 id 到位置的额外映射。
- 2、当某个 segment 上的消息条数达到配置值或消息发布时间超过阈值时，segment 上的消息会被 flush 到磁盘，只有 flush 到磁盘上的消息订阅者才能订阅到
- 3、segment 达到一定的大小（可以通过配置文件设定,默认 1G）后将不会再往该 segment 写数据，broker 会创建新的 segment。



offset

offset 是每条消息的偏移量。

segment 日志文件中保存了一系列"log entries"(日志条目),每个 log entry 格式为"4 个字节的数字 N 表示消息的长度" + "N 个字节的消息内容";

每个日志文件都有一个 offset 来唯一的标记一条消息,offset 的值为 8 个字节的数字,表示此消息在此 partition 中所处的起始位置.

每个 partition 在物理存储层面,有多个 log file 组成(称为 segment).

segment file 的命名为 "最小 offset".log. 例如 "00000000000.log"; 其中 "最小 offset" 表示此 segment 中起始消息的 offset.

segment1

00000000000.kafka

00000000000 365 发发沙发沙发沙发沙发.....
00000000001 45 fafasdfasdfasdf.....
.....

segment2

00000000010.kafka

00000000010 78 跟他沟通.....
00000000011 67 不敢不敢.....
.....

4.3 kafka 集群的安装与搭建

kafka 的官网

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

下载地址:

https://www.apache.org/dyn/closer.cgi?path=/kafka/1.0.0/kafka_2.11-1.0.0.tgz

第一步: 下载上传压缩包

第二步: 解压

```
tar -zxvf kafka_2.11-1.0.0.tgz -C ../servers/
```

第三步: 安装包的分发

第四步: 修改配置文件

```
cd /export/servers/kafka_2.11-1.0.0/config
```

```
vim server.properties
```

```
df -lh 查看磁盘空间
```

第一台服务器修改配置文件

```
broker.id=0  
num.network.threads=3  
num.io.threads=8  
socket.send.buffer.bytes=102400  
socket.receive.buffer.bytes=102400
```



```
socket.request.max.bytes=104857600
log.dirs=/export/servers/kafka_2.11-1.0.0/logs
num.partitions=2
num.recovery.threads.per.data.dir=1
offsets.topic.replication.factor=1
transaction.state.log.replication.factor=1
transaction.state.log.min.isr=1
log.flush.interval.messages=10000
log.flush.interval.ms=1000
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
zookeeper.connect=node01:2181,node02:2181,node03:2181
zookeeper.connection.timeout.ms=6000
group.initial.rebalance.delay.ms=0
delete.topic.enable=true
host.name=node01
```

第二台服务器修改配置文件

```
broker.id=1
num.network.threads=3
num.io.threads=8
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
log.dirs=/export/servers/kafka_2.11-1.0.0/logs
num.partitions=2
num.recovery.threads.per.data.dir=1
offsets.topic.replication.factor=1
transaction.state.log.replication.factor=1
transaction.state.log.min.isr=1
log.flush.interval.messages=10000
log.flush.interval.ms=1000
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
zookeeper.connect=node01:2181,node02:2181,node03:2181
zookeeper.connection.timeout.ms=6000
```

```
group.initial.rebalance.delay.ms=0
delete.topic.enable=true
host.name=node02
```

第三台服务器修改配置文件

```
broker.id=2
num.network.threads=3
num.io.threads=8
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
log.dirs=/export/servers/kafka_2.11-1.0.0/logs
num.partitions=2
num.recovery.threads.per.data.dir=1
offsets.topic.replication.factor=1
transaction.state.log.replication.factor=1
transaction.state.log.min.isr=1
log.flush.interval.messages=10000
log.flush.interval.ms=1000
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
zookeeper.connect=node01:2181,node02:2181,node03:2181
zookeeper.connection.timeout.ms=6000
group.initial.rebalance.delay.ms=0
delete.topic.enable=true
host.name=node03
```

第四步：三台服务器的启动

三台服务器的启动

```
nohup bin/kafka-server-start.sh config/server.properties >/dev/null
2>&1 &
```

4.4 kafka 的原理

4.4.1 数据生产分发策略

第一种：如果指定了分区号，那么数据就会全部进入到指定的分区里面去

```
//producer.send(new ProducerRecord<String, String>("test",1,"1", "hello world"+i)); //ProducerRecord 使用两个形参，第一个形参是我们的 topic 主题，第二个参数就是我们需要发送的消息
```

如果给了分区号，也给了 key 值，那么优先使用指定的分区号

第二种：如果没有给定分区号，但是给了数据的 key，那么通过 key 的 hash 取值来决定数据到哪一个分区里面去

```
producer.send(new ProducerRecord<String, String>("test","101", "hello world"+i)); //ProducerRecord 使用两个形参，第一个形参是我们的 topic 主题，第二个参数就是我们需要发送的消息
```

如果没有指定分区号，给出了我们的 key，那么就会通过 key 的 hash 取值进行分区，实际工作当中，如果通过这种方式进行分区一定要注意，key 的值一定要变化

第三种：没有给定分区号，也没有给定 key 值，通过轮询的方式来决定数据去哪一个分区

//没有给定分区，也没有给数据的 key 值，那么就会使用轮循的方式实现分区

```
producer.send(new ProducerRecord<String, String>("test", "hello world"+i)); //ProducerRecord 使用两个形参，第一个形参是我们的 topic 主题，第二个参数就是我们需要发送的消息
```

第四种：自定义分区

```
public class MyOwnPartitioner implements Partitioner{

    /**
     * 这个方法决定了我们的数据的分区的方式，
     * @param topic
```

```

    * @param key
    * @param keyBytes
    * @param value
    * @param valueBytes
    * @param cluster
    * @return
    */
    public int partition(String topic, Object key, byte[] keyBytes, Object
value, byte[] valueBytes, Cluster cluster) {

        //返回值决定了数据去哪一个分区
        return 2;
    }

    public void close() {

    }

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

    }
}

```

```

props.put("partitioner.class","cn.itcast.kafkaStudy.MyOwnPartitioner");

```

4.4.2 数据存储机制

partition：分区概念，一个 topic 当中的消息，可以拆分成多个 partition 分区，存放在多个不同的服务器上，实现数据存放的横向扩展

repliation：副本，所有的 partition 都可以指定存放几个副本，做到数据的冗余，保证数据的安全

4.4.3 消息不丢失机制

生产者 producer：分为同步模式与异步模式，同步模式效率低，异步模式效率高
具体配置如下

kafka 的 ack 机制：在 kafka 发送数据的时候，每次发送消息都会有一个确认反馈机制，确保消息正常的能够被收到

如果是同步模式：ack 机制能够保证数据的不丢失，如果 ack 设置为 0，风险很大，一般不建议设置为 0

```
producer.type=sync
```

```
request.required.acks=1
```

如果是异步模式：通过 buffer 来进行控制数据的发送，有两个值来进行控制，时间阈值与消息的数量阈值，如果 buffer 满了数据还没有发送出去，如果设置的是立即清理模式，风险很大，一定要设置为阻塞模式

结论：producer 有丢数据的可能，但是可以通过配置保证消息的不丢失

```
producer.type=async
```

```
request.required.acks=1
```

```
queue.buffering.max.ms=5000
```

```
queue.buffering.max.messages=10000
```

```
queue.enqueue.timeout.ms = -1
```

```
batch.num.messages=200
```

消费者 consumer：通过 offset 来记录每次消费到了哪一条数据，

低速的消费模式：offset 记录在了本地磁盘文件

高速的消费模式：zookeeper 的节点上

服务器 broker：数据分区，备份保证数据的不丢失

4.4.4 消费者的负载均衡机制

kafka 集群的负载均衡：

一个 **partition** 只能被一个消费者消费，一个消费者可以消费多个 **partition**，所以如果设置的 **partition** 数量小于 **consumer** 的数量，就会导致空闲的 **consumer** 没有消费，所以 **partition** 的数量一定要大于或等于 **consumer** 的数量

另一方面：**partition** 的数量一定要大于 **broker** 的数量，这样 **leader partition** 就会均匀的分布在各个 **broker** 中，实现负载的均衡

4.4.5 kafka 的命令行的使用

创建 topic

```
./kafka-topics.sh --create --partitions 3 --replication-factor 2 --topic test
--zookeeper node01:2181,node02:2181,node03:2181
```

查看所有的 topic

```
./kafka-topics.sh --list --zookeeper
node01:2181,node02:2181,node03:2181
```

kafka 的消息发送

```
./kafka-console-producer.sh --broker-list
node01:9092,node02:9092,node03:9092 --topic test
```

kafka 消息的消费

```
./kafka-console-consumer.sh --bootstrap-server
node01:9092,node02:9092,node03:9092 --from-beginning --topic test
```

使用 zk 来连接集群

```
./kafka-console-consumer.sh --zookeeper
node01:2181,node02:2181,node03:2181 --from-beginning --topic test
```

4.5 kafka 的 API 使用

第一步：创建 maven 工程，导入 jar 包

```
<dependencies>

    <!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients -->
    <dependency>

        <groupId>org.apache.kafka</groupId>

        <artifactId>kafka-clients</artifactId>

        <version>1.0.0</version>

    </dependency>

</dependencies>
```

```
</dependency>
```

```
</dependencies>
```

第二步：kafka 的生产者 API

```
public class KafkaProducerStudy {  
    //通过 javaAPI 操作 kafka 的生产者，往 test 这个 topic 里面生产消息  
  
    public static void main(String[] args) {  
        Properties props = new Properties();  
        props.put("bootstrap.servers",  
"node01:9092,node02:9092,node03:9092");  
        props.put("acks", "all"); //kafka 的一个消息确认机制，确保消息的不丢失  
        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");  
  
        Producer<String, String> producer = new  
KafkaProducer<String, String>(props);  
        for (int i = 0; i < 100; i++){  
            producer.send(new ProducerRecord<String,  
String>("test", "hello world"+i)); //ProducerRecord 使用两个形参，第一个形参是我们的 topic 主题，第二个参数就是我们需要发送的消息  
        }  
        producer.close();  
    }  
}
```

第三步：kafka 的消费者的 API

自动管理 offset

```
public class KafkaProducerStudy {  
    //通过 javaAPI 操作 kafka 的生产者，往 test 这个 topic 里面生产消息  
  
    public static void main(String[] args) {  
        Properties props = new Properties();  
        props.put("bootstrap.servers",  
"node01:9092,node02:9092,node03:9092");  
        props.put("acks", "all"); //kafka 的一个消息确认机制，确保消息的不丢失  
        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");  
  
        Producer<String, String> producer = new  
KafkaProducer<String, String>(props);  
        for (int i = 0; i < 100; i++){  
            producer.send(new ProducerRecord<String,  
String>("test", "hello world"+i)); //ProducerRecord 使用两个形参，第一个形参是我们的 topic 主题，第二个参数就是我们需要发送的消息  
        }  
        producer.close();  
    }  
}
```

kafka 的手动管理 offset

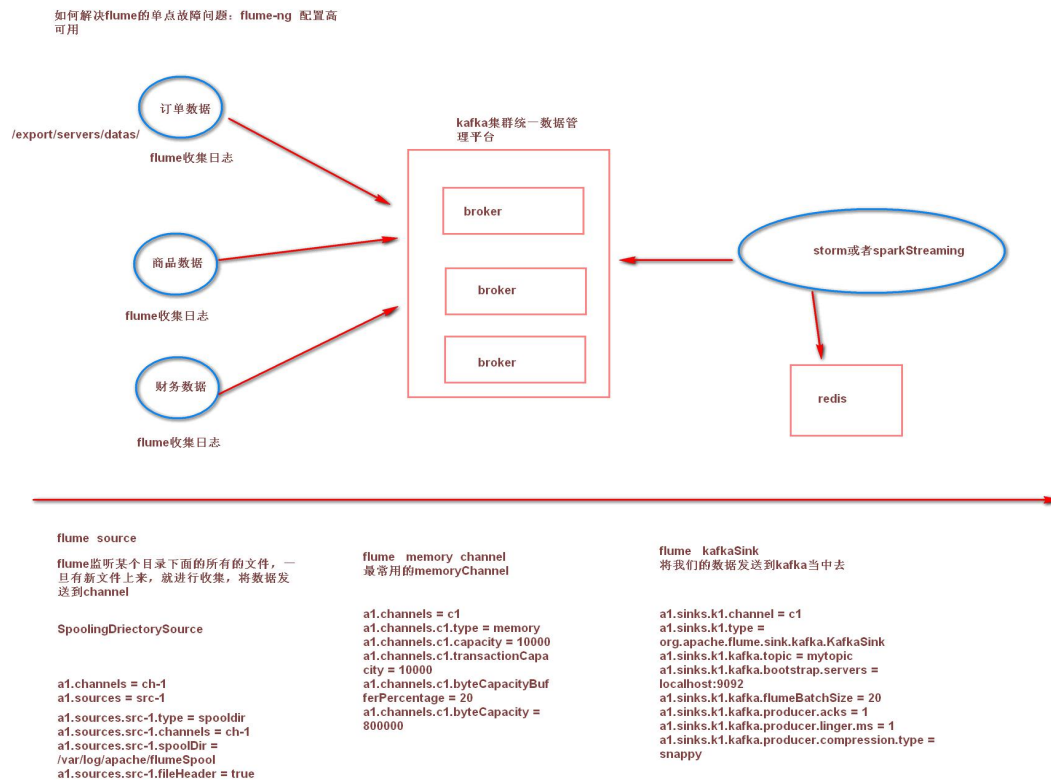

```

public static void main(String[] args) {
    Properties props = new Properties();
    props.put("bootstrap.servers",
"node01:9092,node02:9092,node03:9092");
    props.put("group.id", "test");
    props.put("enable.auto.commit", "false");//如果需要手动管理 offset,
一定要注意，这个配置要给 false
    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<String,String>(props);
    consumer.subscribe(Arrays.asList("test"));
    final int minBatchSize = 10;
    List<ConsumerRecord<String, String>> buffer = new ArrayList
<ConsumerRecord<String, String>>();
    while (true) {
        ConsumerRecords<String, String> records =
consumer.poll(100);//拉取数据
        for (ConsumerRecord<String, String> record : records) {
            buffer.add(record);
        }
        if (buffer.size() >= minBatchSize) {
            // insertIntoDb(buffer);实现自己的业务逻辑在这里
            consumer.commitSync();//一批次的提交我们的 offset
            buffer.clear();
        }
    }
}

```

4.6 kafka 与其他的整合使用

4.6.1 flume 与 kafka 的整合



实现 flume 监控某个目录下面的所有文件，然后将文件收集发送到 kafka 消息系统中

第一步: flume 下载地址

<http://archive.apache.org/dist/flume/1.8.0/apache-flume-1.8.0-bin.tar.gz>

第二步: 上传解压 flume

第三步: 配置 flume.conf

#为我们的 source channel sink 起名

```
a1.sources = r1
```

```
a1.channels = c1
```

```
a1.sinks = k1
```

#指定我们的 source 收集到的数据发送到哪个管道

```

a1.sources.r1.channels = c1
#指定我们的 source 数据收集策略
a1.sources.r1.type = spooldir
a1.sources.r1.spoolDir = /export/servers/flumedata
a1.sources.r1.deletePolicy = never
a1.sources.r1.fileSuffix = .COMPLETED
a1.sources.r1.ignorePattern = ^(.)*\\.tmp$
a1.sources.r1.inputCharset = GBK
#指定我们的 channel 为 memory,即表示所有的数据都装进 memory 当中
a1.channels.c1.type = memory
#指定我们的 sink 为 kafka sink, 并指定我们的 sink 从哪个 channel 当中读取数据
a1.sinks.k1.channel = c1
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic = test
a1.sinks.k1.kafka.bootstrap.servers =
node01:9092,node02:9092,node03:9092
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1

```

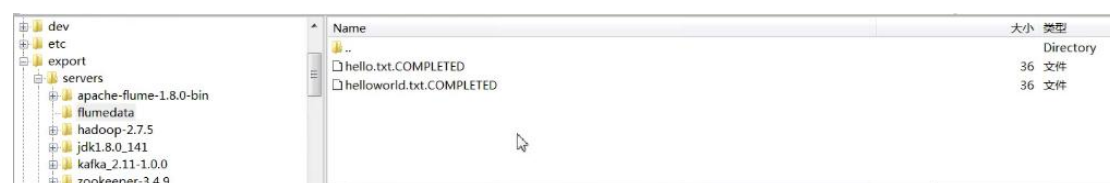
启动 flume

```

bin/flume-ng agent --conf conf --conf-file conf/flume.conf --name a1
-Dflume.root.logger=INFO,console

```

把文件放到 flumeData 文件夹下,刷新,文件会自动变为.COMPLETED 文件



然后在 kafka 中开启消费者,就会看到文件中的数据.

4.6.2 storm 与 kafka 集成

旧版本的 kafka 与 storm 之间相互集成

第一步: 导入 jar 包

```

<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-core</artifactId>
    <version>1.1.1</version>
</dependency>

<!-- use old kafka spout code -->
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-kafka</artifactId>
    <version>1.1.1</version>
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.10</artifactId>
    <version>0.8.2.1</version>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
        </exclusion>
    </exclusions>
</dependency>

```

第二步：代码实现

```

topologyBuilder.setSpout("WordCountFileSpout", new KafkaSpout(new SpoutConfig(new
ZkHosts("zk01:2181,zk02:2181,zk03:2181"),"test", "/test", "storm")), 1);

```

新版本的 kafka 与 storm1.1.1 集成

第一步：导入 jar 包

```
<!-- use new kafka spout code -->
    <dependency>
        <groupId>org.apache.storm</groupId>
        <artifactId>storm-kafka-client</artifactId>
        <version>1.1.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>0.10.0.0</version>
    </dependency>
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-core</artifactId>
    <version>1.1.1</version>
    <scope>provided</scope>
</dependency>
```

第二步：编写我们的主函数入口程序

```
public class KafkStormTopo {

    public static void main(String[] args) throws Exception {
        KafkaSpoutConfig.Builder<String, String> builder =
        KafkaSpoutConfig.builder("192.168.52.200:9092,192.168.52.201:9092,19
        2.168.52.202:9092","yun01");
        builder.setGroupId("test_storm_wc");
        KafkaSpoutConfig<String, String> kafkaSpoutConfig =
        builder.build();
        TopologyBuilder topologyBuilder = new TopologyBuilder();
        topologyBuilder.setSpout("WordCountFileSpout",new
        KafkaSpout<String,String>(kafkaSpoutConfig), 1);
        topologyBuilder.setBolt("readKafkaBolt", new
        KafkaBolt()).shuffleGrouping("WordCountFileSpout");
        Config config = new Config();
        if(args !=null && args.length > 0){
```

```

        config.setDebug(false);
        StormSubmitter submitter = new StormSubmitter();
        submitter.submitTopology("kafkaStromTopo", config,
        topologyBuilder.createTopology());
    }else{
        config.setDebug(true);
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("kafkaStromTopo", config,
        topologyBuilder.createTopology());
    }
}
}

```

第三步：开发我们的 kafkabolt 作为消息处理

```

public class KafkaBolt extends BaseBasicBolt {

    @Override
    public void execute(Tuple input, BasicOutputCollector collector) {
        System.out.println(input.getValues().get(4)+"消息接受 bolt");
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {

    }

}

```

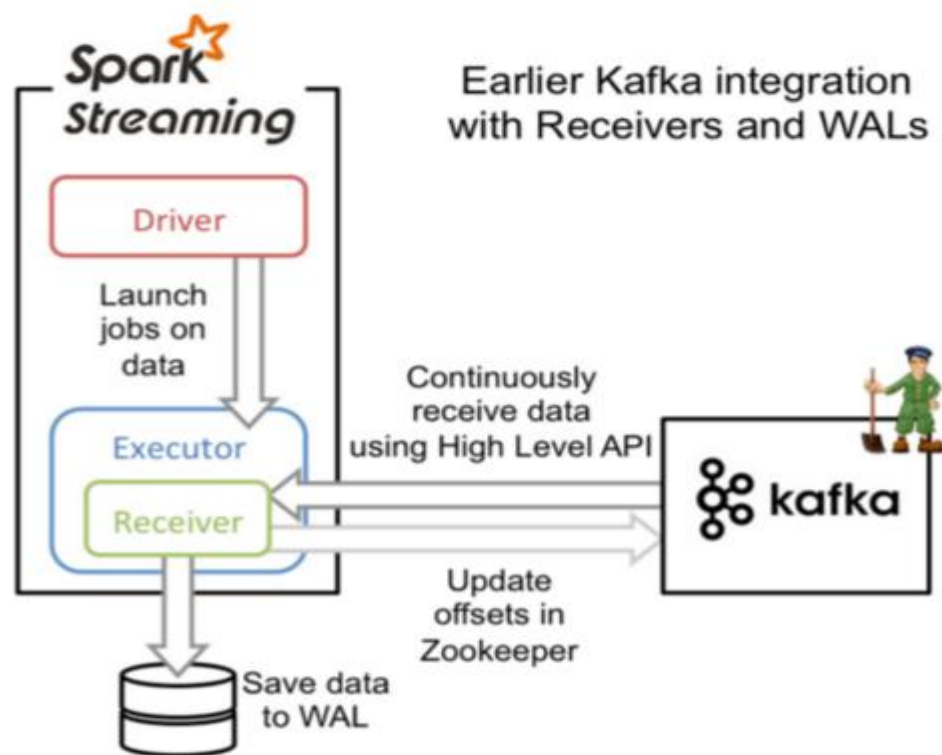
4.6.3 Spark Streaming 整合 kafka 实战

kafka 作为一个实时的分布式消息队列，实时的生产和消费消息，这里我们可以利用 SparkStreaming 实时地读取 kafka 中的数据，然后进行相关计算。

在 Spark1.3 版本后，KafkaUtils 里面提供了两个创建 dstream 的方法，一种为 KafkaUtils.createDstream，另一种为 KafkaUtils.createDirectStream。

KafkaUtils.createDstream 方式

KafkaUtils.createDstream(ssc, [zk], [group id], [per-topic,partitions]) 使用了 receivers 接收器来接收数据，利用的是 Kafka 高层次的消费者 api，对于所有的 receivers 接收到的数据将会保存在 Spark executors 中，然后通过 Spark Streaming 启动 job 来处理这些数据，默认会丢失，可启用 WAL 日志，它同步将接受到数据保存到分布式文件系统上比如 HDFS。所以数据在出错的情况下可以恢复出来。



A、创建一个 receiver 接收器来对 kafka 进行定时拉取数据，这里产生的 dstream 中 rdd 分区和 kafka 的 topic 分区不是一个概念，故如果增加特定主体分区数仅仅是增加一个 receiver 中消费 topic 的线程数，并没有增加 spark 的并行处理的数据量。

B、对于不同的 group 和 topic 可以使用多个 receivers 创建不同的 DStream

C、如果启用了 WAL(spark.streaming.receiver.writeAheadLog.enable=true)

同时需要设置存储级别(默认 StorageLevel.MEMORY_AND_DISK_SER_2),

KafkaUtils.createDstream 实战

(1) 添加 kafka 的 pom 依赖

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-streaming-kafka_0-8_2.11</artifactId>
  <version>2.0.2</version>
</dependency>
```

(2) 启动 zookeeper 集群

zkServer.sh start

(3) 启动 kafka 集群

kafka-server-start.sh /export/servers/kafka/config/server.properties

(4) 创建 topic

**kafka-topics.sh --create --zookeeper hdp-node-01:2181 --replication-factor 1
--partitions 3 --topic kafka_spark**

(5) 向 topic 中生产数据

通过 shell 命令向 topic 发送消息

kafka-console-producer.sh --broker-list hdp-node-01:9092 --topic kafka_spark

```
hadoop@hdp-node-01 conf1# kafka-console-producer.sh --broker-list hdp-node-01:9092 --topic kafka_spark
hadoop spark hive hadoop
spark sqoop flume kafka flume hive hadoop
```

向topic中发送数据 用空格隔开

(6) 编写 Spark Streaming 应用程序

```
package cn.itcast.dstream.kafka
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.dstream.{DStream, ReceiverInputDStream}
```



```

import org.apache.spark.streaming.kafka.KafkaUtils
import scala.collection.immutable

// todo: 利用 sparkStreaming 对接 kafka 实现单词计数----采用 receiver (高级 API)
object SparkStreamingKafka_Receiver {
  def main(args: Array[String]): Unit = {
    //1、创建 sparkConf
    val sparkConf: SparkConf = new SparkConf()
      .setAppName("SparkStreamingKafka_Receiver")
      .setMaster("local[4]")
      .set("spark.streaming.receiver.writeAheadLog.enable", "true") //开启 wal 预写
    日志, 保存数据源的可靠性
    //2、创建 sparkContext
    val sc = new SparkContext(sparkConf)
    sc.setLogLevel("WARN")
    //3、创建 StreamingContext
    val ssc = new StreamingContext(sc, Seconds(5))
    //设置 checkpoint
    ssc.checkpoint("./Kafka_Receiver")
    //4、定义 zk 地址
    val zkQuorum="node1:2181,node2:2181,node3:2181"
    //5、定义消费者组
    val groupId="spark_receiver1"
    //6、定义 topic 相关信息 Map[String, Int]
    // 这里的 value 并不是 topic 分区数, 它表示的 topic 中每一个分区被 N 个线程消费
    val topics=Map("spark_kafka" -> 2)
    //7、通过 KafkaUtils.createStream 对接 kafka
    //这个时候相当于同时开启 3 个 receiver 接受数据
    val receiverDstream: immutable.IndexedSeq[ReceiverInputDStream[(String,
String)]] = (1 to 3).map(x => {
      val stream: ReceiverInputDStream[(String, String)] =
      KafkaUtils.createStream(ssc, zkQuorum, groupId, topics)
      stream
    })
    //使用 ssc.union 方法合并所有的 receiver 中的数据
    val unionDStream: DStream[(String, String)] = ssc.union(receiverDstream)

    //8、获取 topic 中的数据
    val topicData: DStream[String] = unionDStream.map(_._2)
    //9、切分每一行, 每个单词计为 1
    val wordAndOne: DStream[(String, Int)] = topicData.flatMap(_.split("
")).map(_._1)
    //10、相同单词出现的次数累加
  }
}

```

```

    val result: DStream[(String, Int)] = wordAndOne.reduceByKey(_+_ )
    //11、打印输出
    result.print()
    //开启计算
    ssc.start()
    ssc.awaitTermination()
  }
}

```

(7) 运行代码,查看控制台结果数据

```

-----
Time: 1505116085000 ms
-----
(hive,1)
(spark,1)
(hadoop,2)
-----
Time: 1505116090000 ms
-----
(hive,2)
(spark,2)
(hadoop,3)
(sqoop,1)
(kafka,1)
(flume,2)

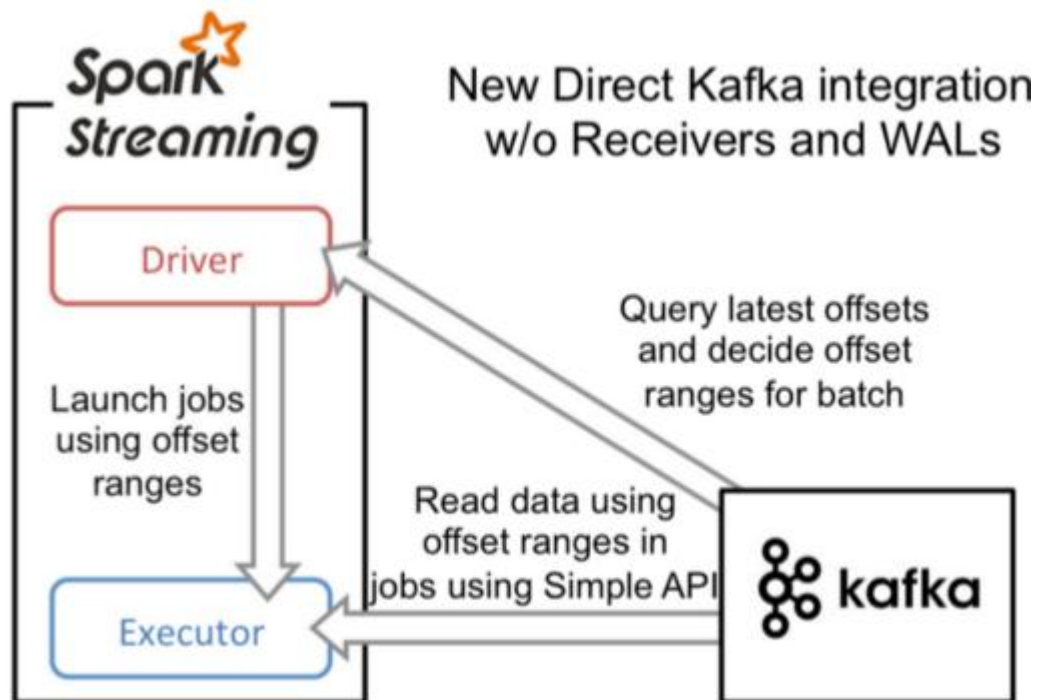
```

总结:

通过这种方式实现，刚开始的时候系统正常运行，没有发现问题，但是如果系统异常重新启动 sparkstreaming 程序后，发现程序会重复处理已经处理过的数据，这种基于 **receiver** 的方式，是使用 Kafka 的高级 API，topic 的 offset 偏移量在 ZooKeeper 中。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只被处理一次，可能会处理两次。**因为 Spark 和 ZooKeeper 之间可能是不同步的。官方现在也已经不推荐这种整合方式，我们使用官网推荐的第二种方式 kafkaUtils 的 createDirectStream() 方式。

KafkaUtils.createDirectStream 方式

这种方式不同于 Receiver 接收数据，它定期地从 kafka 的 topic 下对应的 partition 中查询最新的偏移量，再根据偏移量范围在每个 batch 里面处理数据，Spark 通过调用 kafka 简单的消费者 Api（低级 api）读取一定范围的数据。



相比基于 Receiver 方式有几个优点：

A、简化并行

不需要创建多个 kafka 输入流，然后 union 它们，sparkStreaming 将会创建和 kafka 分区数相同的 rdd 的分区数，而且会从 kafka 中并行读取数据，spark 中 RDD 的分区数和 kafka 中的 topic 分区数是一一对应的关系。

B、高效，

第一种实现数据的零丢失是将数据预先保存在 WAL 中，会复制一遍数据，会导致数据被拷贝两次，第一次是接受 kafka 中 topic 的数据，另一次是写到 WAL

中。而没有 receiver 的这种方式消除了这个问题。

C、恰好一次语义(Exactly-once-semantics)

Receiver 读取 kafka 数据是通过 kafka 高层次 api 把偏移量写入 zookeeper 中，虽然这种方法可以通过数据保存在 WAL 中保证数据不丢失，但是可能会因为 sparkStreaming 和 ZK 中保存的偏移量不一致而导致数据被消费了多次。EOS 通过实现 kafka 低层次 api，偏移量仅仅被 ssc 保存在 checkpoint 中，消除了 zk 和 ssc 偏移量不一致的问题。缺点是无法使用基于 zookeeper 的 kafka 监控工具。

KafkaUtils.createDirectStream 实战

```
package cn.itcast.dstream.kafka

import kafka.serializer.StringDecoder
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.dstream.{DStream, InputDStream}
import org.apache.spark.streaming.kafka.KafkaUtils

// todo: 利用 sparkStreaming 对接 kafka 实现单词计数----采用 Direct (低级 API)
object SparkStreamingKafka_Direct {
  def main(args: Array[String]): Unit = {
    //1、创建 sparkConf
    val sparkConf: SparkConf = new SparkConf()
      .setAppName("SparkStreamingKafka_Direct")
      .setMaster("local[2]")
    //2、创建 sparkContext
    val sc = new SparkContext(sparkConf)
    sc.setLogLevel("WARN")
    //3、创建 StreamingContext
    val ssc = new StreamingContext(sc, Seconds(5))
    ssc.checkpoint("./Kafka_Direct")
    //4、配置 kafka 相关参数
    val
    kafkaParams=Map("metadata.broker.list"->"node1:9092,node2:9092,node3:9092","group.
    id"->"Kafka_Direct")
```

```

//5、定义 topic
val topics=Set("spark01")
//6、通过 KafkaUtils.createDirectStream 接受 kafka 数据，这里采用是 kafka 低级
api 偏移量不受 zk 管理
val dstream: InputDStream[(String, String)] =
KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](ssc, kafka
Params, topics)
//7、获取 kafka 中 topic 中的数据
val topicData: DStream[String] = dstream.map(_._2)
//8、切分每一行，每个单词计为 1
val wordAndOne: DStream[(String, Int)] = topicData.flatMap(_.split("
")).map((_, 1))
//9、相同单词出现的次数累加
val result: DStream[(String, Int)] = wordAndOne.reduceByKey(_+_ )
//10、打印输出
result.print()
//开启计算
ssc.start()
ssc.awaitTermination()
}
}

```

(2) 查看对应的效果

向 topic 中添加数据

```

[root@hdp-node-01 conf]# kafka-console-producer.sh --broker-list hdp-node-01:9092 --topic kafka_spark
hadoop spark hive hadoop
spark sqoop flume kafka flume hive hadoop

```

查看控制台的输出：

```

17/09/11 16:40:30 WARN VerifiableProperties: Property serializer.class is not valid
-----
Time: 1505119230000 ms
-----
(hive,1)
(spark,1)
(hadoop,2)

17/09/11 16:40:35 WARN VerifiableProperties: Property serializer.class is not valid
-----
Time: 1505119235000 ms
-----
(hive,2)
(sqoop,1)
(kafka,1)
(spark,2)
(hadoop,3)
(flume,2)

```

4.6.4 kafka 与 Hbase 的集成使用

Kafka 接收数据,并消费到 Hbase 数据库

一、

1、生产者 产生数据

```
1 package kafakaTohbase;
2
3 import java.util.Properties;
4
5 import kafka.javaapi.producer.Producer;
6 import kafka.producer.KeyedMessage;
7 import kafka.producer.ProducerConfig;
8
9 public class KafkaProducer {
10
11     public static void main(String[] args) {
12         Properties props = new Properties();
13         props.put("zk.connect", KafkaProperties.zkConnect);
14         props.put("serializer.class", "kafka.serializer.StringEncoder");
15         props.put("metadata.broker.list", "hdjt01:9092,hdjt02:9092,hdjt03:9092");
16         ProducerConfig config = new ProducerConfig(props);
17         Producer<String, String> producer = new Producer<String, String>(config);
18         for (int i = 0; i < 10; i++){
19             producer.send(new KeyedMessage<String, String>("test5", "liu" + i));
20         }
21     }
22
23
24 }
```

注：props.put("serializer.class", "kafka.serializer.StringEncoder") 发送的数据是 String,

还可以是 二进制数组形式：

```
props.put("serializer.class", "kafka.serializer.DefaultEncoder");
props.put("key.serializer.class", "kafka.serializer.StringEncoder"); 如果没有这个，就代表 key 也是二进制形式。
```

生产者发送的都是 keyvalue 对

2、消费者

```
1 package kafakaTohbase;
2
3 import java.io.IOException;
4 import java.util.HashMap;
5 import java.util.List;
6 import java.util.Map;
7 import java.util.Properties;
8
9 import kafka.consumer.ConsumerConfig;
10 import kafka.consumer.ConsumerIterator;
11 import kafka.consumer.KafkaStream;
12 import kafka.javaapi.consumer.ConsumerConnector;
13 import kafka.javaapi.producer.Producer;
14 import kafka.producer.KeyedMessage;
15 import kafka.producer.ProducerConfig;
16
17 public class KafkaConsumer extends Thread{
18
19     private final ConsumerConnector consumer;
20     private final String topic;
21
22     public KafkaConsumer(String topic) {
23         consumer = kafka.consumer.Consumer
24             .createJavaConsumerConnector(createConsumerConfig());
25         this.topic = topic;
26     }
27
28     private static ConsumerConfig createConsumerConfig() {
29         Properties props = new Properties();
30         props.put("zookeeper.connect", KafkaProperties.zkConnect);
31         props.put("group.id", KafkaProperties.groupId1);
```

```

32         props.put("zookeeper.session.timeout.ms", "40000");           //zookeeper
33 与 region server 的连接超时时间
34         props.put("zookeeper.sync.time.ms", "200");
35         props.put("auto.commit.interval.ms", "1000");
36 <br>           //props.put("auto.offset.reset", "smallest");//可以读取旧数据,
37 默认不读取
38         return new ConsumerConfig(props);
39     }
40
41     @Override
42     public void run() {
43
44         Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
45         topicCountMap.put(topic, new Integer(1));
46         Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer
47             .createMessageStreams(topicCountMap);
48         KafkaStream<byte[], byte[]> stream = consumerMap.get(topic).get(0);
49         ConsumerIterator<byte[], byte[]> it = stream.iterator();
50         HBaseUtils hbase = new HBaseUtils();
51         while (it.hasNext()) { //相当于加了一把锁, 一直返回 true
52 //             System.out.println("3receive: " + it.next().message());
53             try {
54                 System.out.println("11111");
55                 hbase.put(new String(it.next().message()));
56             } catch (IOException e) {
57                 // TODO Auto-generated catch block
58                 e.printStackTrace();
59             }
60
61 //             try {
62 //                 sleep(300); // 每条消息延迟 300ms
63 //             } catch (InterruptedException e) {
64 //                 e.printStackTrace();
65 //             }
66         }
    }

```



```
}
```

连接 hbase, 配置信息

```
1 package kafakaTohbase;
2
3 import java.io.IOException;
4 import java.util.Random;
5
6 import org.apache.hadoop.conf.Configuration;
7 import org.apache.hadoop.hbase.HBaseConfiguration;
8 import org.apache.hadoop.hbase.client.HTable;
9 import org.apache.hadoop.hbase.client.Put;
10 import org.apache.hadoop.hbase.util.Bytes;
11
12 public class HBaseUtils {
13     public void put(String string) throws IOException {
14         //设置 HBase 数据库的连接配置参数
15         Configuration conf = HBaseConfiguration.create();
16         conf.set("hbase.zookeeper.quorum", "hdjt01:2181,hdjt02:2181,hdjt03:2
17 181"); // Zookeeper 的地址
18         // conf.set("hbase.zookeeper.property.clientPort", "42182");
19         Random random = new Random();
20         long a = random.nextInt(1000000000);
21         String tableName = "emp";
22         String rowkey = "rowkey"+a ;
23         String columnFamily = "basicinfo";
24         String column = "empname";
25         //String value = string;
26         HTable table=new HTable(conf, tableName);
27         Put put=new Put(Bytes.toBytes(rowkey));
28         put.add(Bytes.toBytes(columnFamily), Bytes.toBytes(column),
29 Bytes.toBytes(string));
30         table.put(put);//放入表
31         System.out.println("放入成功");
32         table.close();//释放资源
```

1	}
2	}
2	
2	
3	
2	
4	
2	
5	
2	
6	
2	
7	
2	
8	
2	
9	
3	
0	
3	
1	
3	
2	

测试消费者:

1	public class Kafkaceshi {
2	
3	public static void main(String[] args) {
4	// KafkaProducer a=new KafkaProducer ();
5	// a.producer();
6	KafkaConsumer consumerThread
7	= new KafkaConsumer(KafkaProperties.topic);
8	consumerThread.run();
9	
10	}
11	

	}
--	---

第 5 章 HBASE 数据库

5.1 数据库 OLAP、OLTP 的介绍和比较

数据处理大致可以分成两大类：**联机事务处理 OLTP**（on-line transaction processing）、**联机分析处理 OLAP**（On-Line Analytical Processing）。**OLTP** 是传统的关系型数据库的主要应用，主要是基本的、日常的事务处理，例如银行交易。**OLAP** 是数据仓库系统的主要应用，支持复杂的分析操作，侧重决策支持，并且提供直观易懂的查询结果。

OLTP 系统强调数据库内存效率，强调内存各种指标的命令率，强调绑定变量，强调并发操作；

OLAP 系统则强调数据分析，强调 **SQL** 执行市场，强调磁盘 I/O，强调分区等。

OLTP 与 OLAP 之间的比较：

	OLTP	OLAP
用户	操作人员,低层管理人员	决策人员,高级管理人员
功能	日常操作处理	分析决策
DB 设计	面向应用	面向主题
数据	当前的,最新的细节的,二维的分立的	历史的,聚集的,多维的集成的,统一的
存取	读/写数十条记录	读上百万条记录
工作单位	简单的事务	复杂的查询
用户数	上千个	上百个
DB 大小	100MB-GB	100GB-TB

5.2 Hbase 基础

什么是 OLTP

OLTP，也叫联机事务处理（Online Transaction Processing），表示事务性非常高的系统，一般都是高可用的在线系统，以小的事务以及小的查询为主，评估其系统的时候，一般看其每秒执行的 Transaction 以及 Execute SQL 的数量。在这样的系统中，单个数据库每秒处理的 Transaction 往往超过几百个，或者是几千个，Select 语句的执行量每秒几千甚至几万个。典型的 OLTP 系统有电子商务系统、银行、证券等，如美国 eBay 的业务数据库，就是很典型的 OLTP 数据库。

OLTP 系统最容易出现瓶颈的地方就是 CPU 与磁盘子系统。

(1) CPU 出现瓶颈常表现在逻辑读总量与计算性函数或者是过程上，逻辑读总量等于单个语句的逻辑读乘以执行次数，如果单个语句执行速度虽然很快，但是执行次数非常多，那么，也可能导致很大的逻辑读总量。设计的方法与优化的方法就是减少单个语句的逻辑读，或者是减少它们的执行次数。另外，一些计算型的函数，如自定义函数、decode 等的频繁使用，也会消耗大量的 CPU 时间，造成系统的负载升高，正确的设计方法或者是优化方法，需要尽量避免计算过程，如保存计算结果到统计表就是一个好的方法。

(2) 磁盘子系统在 OLTP 环境中，它的承载能力一般取决于它的 IOPS 处理能力。因为在 OLTP 环境中，磁盘物理读一般都是 db file sequential read，也就是单块读，但是这个读的次数非常频繁。如果频繁到磁盘子系统都不能承载其 IOPS 的时候，就会出现大的性能问题。

OLTP 比较常用的设计与优化方式为 Cache 技术与 B-tree 索引技术，Cache 决定了很多语句不需要从磁盘子系统获得数据，所以，Web cache 与 Oracle data buffer 对 OLTP 系统是很重要的。另外，在索引使用方面，语句越简单越好，这样执行计划也稳定，而且一定要使用绑定变量，减少语句解析，尽量减少表关联，尽量减少分布式事务，基本不使用分区技术、MV 技术、并行技术及位图索引。因为并发量很高，批量更新时要分批快速提交，以避免阻塞的发生。

OLTP 系统是一个数据块变化非常频繁，SQL 语句提交非常频繁的系统。对于数据块来说，应尽可能让数据块保存在内存当中，对于 SQL 来说，尽可能使用变量绑定技术来达到 SQL 重用，减少物理 I/O 和重复的 SQL 解析，从而极大的改善数据库的性能。

这里影响性能除了绑定变量，还有可能是热块 (hot block)。当一个块被多个用户同时读取时，Oracle 为了维护数据的一致性，需要使用 Latch 来串行化用户的操作。当一个用户获得了 latch 后，其他用户就只能等待，获取这个数据块的用户越多，等待就越明显。这就是热块的问题。这种热块可能是数据块，也可能是回滚段块。对于数据块来讲，通常是数据库的数据分布不均匀导致，如果是索引的数据块，可以考虑创建反向索引来达到重新分布数据的目的，对于回滚段数据块，可以适当多增加几个回滚段来避免这种争用。

什么是 OLAP

OLAP，也叫联机分析处理 (Online Analytical Processing) 系统，有的时候也叫 DSS 决策支持系统，就是我们说的数据仓库。在这样的系统中，语句的执行量不是考核标准，因为一条语句的执行时间可能会非常长，读取的数据也非常多。所以，在这样的系统中，考核的标准往往是磁盘子系统的吞吐量 (带宽)，如能达到多少 MB/s 的流量。

磁盘子系统的吞吐量则往往取决于磁盘的个数，这个时候，Cache 基本是没有效果的，数据库的读写类型基本上是 db file scattered read 与 direct path read/write。应尽量采用个数比较多的磁盘以及比较大的带宽，如 4Gb 的光纤接口。

在 OLAP 系统中，常使用分区技术、并行技术。

分区技术在 OLAP 系统中的重要性主要体现在数据库管理上，比如数据库加载，可以通过分区交换的方式实现，备份可以通过备份分区表空间实现，删除数据可以通过分区进行删除，至于分区在性能上的影响，它可以使得一些大表的扫描变得很快 (只扫描单个分区)。另外，如果分区结合并行的话，也可以使得整个表的扫描会变得很快。总之，分区主要的功能是管理上的方便性，它并不能绝对保证查询性能的提高，有时候分区会带来性能上的提高，有时候会降低。

并行技术除了与分区技术结合外，在 Oracle 10g 中，与 RAC 结合实现多节点的同时扫描，效果也非常不错，可把一个任务，如 select 的全表扫描，平均地分派到多个 RAC 的节点上去。

在 OLAP 系统中，不需要使用绑定（BIND）变量，因为整个系统的执行量很小，分析时间对于执行时间来说，可以忽略，而且可避免出现错误的执行计划。但是 OLAP 中可以大量使用位图索引，物化视图，对于大的事务，尽量寻求速度上的优化，没有必要像 OLTP 要求快速提交，甚至要刻意减慢执行的速度。

绑定变量真正的用途是在 OLTP 系统中，这个系统通常有这样的特点，用户并发数很大，用户的请求十分密集，并且这些请求的 SQL 大多数是可以重复使用的。

对于 OLAP 系统来说，绝大多数时候数据库上运行着的是报表作业，执行基本上是聚合类的 SQL 操作，比如 group by，这时候，把优化器模式设置为 all_rows 是恰当的。而对于一些分页操作比较多的网站类数据库，设置为 first_rows 会更好一些。但有时候对于 OLAP 系统，我们又有分页的情况下，我们可以考虑在每条 SQL 中用 hint。如：

```
Select /*+first_rows(10) */ a.* from table a;
```

分开设计与优化

在设计上要特别注意，如在高可用的 OLTP 环境中，不要盲目地把 OLAP 的技术拿过来用。

如分区技术，假设不是大范围地使用分区关键字，而采用其它的字段作为 where 条件，那么，如果是本地索引，将不得不扫描多个索引，而性能变得更为低下。如果是全局索引，又失去分区意义。

并行技术也是如此，一般在完成大型任务时才使用，如在实际生活中，翻译一本书，可以先安排多个人，每个人翻译不同的章节，这样可以提高翻译速度。如果只是翻译一页书，也去分配不同的人翻译不同的行，再组合起来，就没必要了，因为在分配工作的时间里，一个人或许早就翻译完了。

位图索引也是一样，如果用在 OLTP 环境中，很容易造成阻塞与死锁。但是，在 OLAP 环境中，可能会因为其特有的特性，提高 OLAP 的查询速度。MV 也是基本一样，包括触发器等，在 DML 频繁的 OLTP 系统上，很容易成为瓶颈，甚至是 Library Cache 等待，而在 OLAP 环境上，则可能会因为使用恰当而提高查询速度。

对于 OLAP 系统，在内存上可优化的余地很小，增加 CPU 处理速度和磁盘 I/O 速度是最直接的提高数据库性能的方法，当然这也意味着系统成本的增加。

比如我们要对几亿条或者几十亿条数据进行聚合处理，这种海量的数据，全部放在内存中操作是很难的，同时也没有必要，因为这些数据很少重用，缓存起来也没有实际意义，而且还会造成物理 I/O 相当大。所以这种系统的瓶颈往往是磁盘 I/O 上面的。

对于 OLAP 系统，SQL 的优化非常重要，因为它的数据库量很大，做全表扫描和索引对性能上来说差异是非常大的。

5.2.1 hbase 数据库介绍

简介

hbase 是基于 Google BigTable 模型开发的，典型的 key/value 系统。是建立在 hdfs 之上，提供高可靠性、高性能、列存储、可伸缩、实时读写 nosql 的数据库系统。主要用于海量结构化和半结构化数据存储。

它介于 nosql 和 RDBMS 之间，仅能通过主键(row key)和主键的 range 来检索数据，仅支持单行事务(可通过 hive 支持来实现多表 join 等复杂操作)。

Hbase 查询数据功能很简单，不支持 join 等复杂操作，不支持复杂的事务（行级的事务）

与 hadoop 一样，Hbase 目标主要依靠横向扩展，通过不断增加廉价的商用服务器，来增加计算和存储能力。

HBase 中的表的特点：

- ✧ 大：一个表可以有上十亿行，上百万列
- ✧ 无模式：每行都有一个可排序的主键和任意多的列，列可以根据需要动态的增加，同一张表中不同的行可以有截然不同的列；
- ✧ 面向列：面向列(族)的存储和权限控制，列(族)独立检索。
- ✧ 稀疏：对于为空(null)的列，并不占用存储空间，因此，表可以设计的非常稀疏。
- ✧ 数据多版本：每个单元中的数据可以有多个版本，默认情况下版本号自动分配，是单元格插入时的时间戳
- ✧ 数据类型单一：Hbase 中的数据都是字节数组 byte[]。

表结构逻辑视图

HBase 以表的形式存储数据。表有行和列组成。列划分为若干个列族(column family)

表名: t_user_info

rowkey	列族1: base_info	列族2: extra_info
000000001	name:zhangsan age:30	address: BJ
000000002	name:lisi age: 50 sex: male	address: SH hobbies: sing
	name: lisisi	

rowkey: hbase表的行索引

rowkey: 按照字典顺序排列

rowkey的这个特性对于hbase表的很重要。

cell: 可以锁定唯一的值
(rowkey+列族+列族下的列名称+值+时间戳)

每一个cell里面, 存放不同版本数据

按照数据插入的先后顺序确定的, 版本号是按照时间戳大小给定的

最小的版本数: 最少可以允许存放几个版本数据

最大的版本数: 最多可以允许存放几个版本数据

1、Row Key

与 nosql 数据库们一样, row key 是用来检索记录的主键。访问 hbase table 中的行, 只有三种方式:

- 1 通过单个 row key 访问
- 2 通过 row key 的 range
- 3 全表扫描

Row key 行键 (Row key) 可以是任意字符串 (最大长度是 64KB, 实际应用中长度一般为 10-100bytes), 在 hbase 内部, row key 保存为字节数组。

Hbase 会对表中的数据按照 rowkey 排序(字典顺序)

存储时, 数据按照 Row key 的字典序(byte order)排序存储。设计 key 时, 要充分排序存储这个特性, 将经常一起读取的行存储放到一起。(位置相关性)

注意:

字典序对 int 排序的结果是

1, 10, 100, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2, 20, 21, ..., 9, 91, 92, 93, 94, 95, 96, 97, 98, 99。要保持整形的自然序, 行键必须用 0 作左填充。

行的一次读写是原子操作 (不论一次读写多少列)。这个设计决策能够使用户很容易的理解程序在对同一个行进行并发更新操作时的行为。

2、列族

hbase 表中的每个列, 都归属与某个列族。列族是表的 schema 的一部分(而列不是), 必须在使用表之前定义。

列名都以列族作为前缀。例如 `courses:history` , `courses:math` 都属于 `courses` 这个列族。

访问控制、磁盘和内存的使用统计都是在列族层面进行的。

列族越多，在取一行数据时所参与 IO、搜寻的文件就越多，所以，如果没有必要，不要设置太多的列族。一般设置 2-3 个比较合理。

3、时间戳

HBase 中通过 `row` 和 `columns` 确定的为一个存储单元称为 `cell`。每个 `cell` 都保存着同一份数据的多个版本。版本通过时间戳来索引。时间戳的类型是 64 位整型。时间戳可以由 `hbase` (在数据写入时自动) 赋值，此时时间戳是精确到毫秒的当前系统时间。时间戳也可以由客户显式赋值。如果应用程序要避免数据版本冲突，就必须自己生成具有唯一性的时间戳。每个 `cell` 中，不同版本的数据按照时间倒序排序，即最新的数据排在最前面。

为了避免数据存在过多版本造成的管理（包括存储和索引）负担，`hbase` 提供了两种数据版本回收方式：

- ✧ 保存数据的最后 `n` 个版本
- ✧ 保存最近一段时间内的版本（设置数据的生命周期 `TTL`）。

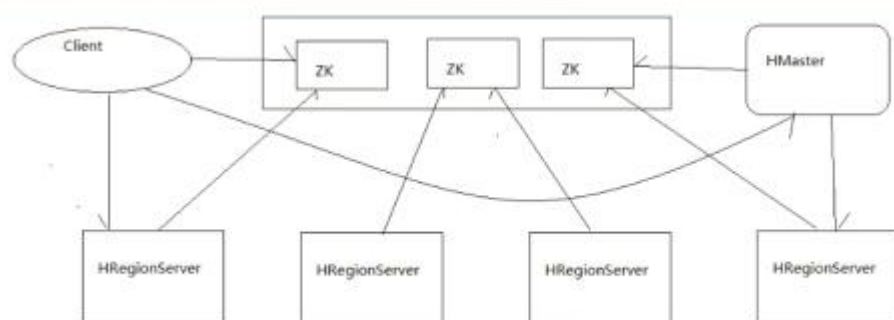
用户可以针对每个列族进行设置。

4、Cell

由 `{row key, column(=<family> + <label>), version}` 唯一确定的单元。

`cell` 中的数据是没有类型的，全部是字节码形式存储。

5.2.2 hbase 集群结构



Hbase 基本组件说明:

Client:

包含访问 Hbase 的接口，并维护 cache 来加快对 Hbase 的访问，比如 region 的位置信息。

HMaster:

是 hbase 集群的主节点，可以配置多个，用来实现 HA

为 RegionServer 分配 region

负责 RegionServer 的负载均衡

发现失效的 RegionServer 并重新分配其上的 region

RegionServer:

Regionserver 维护 region，处理对这些 region 的 IO 请求

Regionserver 负责切分在运行过程中变得过大的 region

Region:

分布式存储的最小单元。

Zookeeper 作用:

通过选举，保证任何时候，集群中只有一个活着的 HMaster，HMaster 与 RegionServers 启动时会向 ZooKeeper 注册

存贮所有 Region 的寻址入口

实时监控 Region server 的上线和下线信息。并实时通知给 HMaster

存储 HBase 的 schema 和 table 元数据

Zookeeper 的引入使得 HMaster 不再是单点故障

5.2.3 hbase 集群搭建 (简易步骤)

——先部署一个 zookeeper 集群，并同步所有服务器的时间

1. 配置 hbase 集群，要修改 3 个文件 hbase-env.sh, hbase-site.xml, regionservers, backup-masters 来指定备用的主节点

注意：要把 hadoop 的 hdfs-site.xml 和 core-site.xml 放到 hbase/conf 下

2. 启动：./zkServer.sh start, start-dfs.sh, start-hbase.sh

3. 访问：itcast01:16010

5.2.4 命令行演示

基本 shell 命令

进入 hbase 命令行

./hbase shell

Create 'tablename' , 'column family'

Put 'tablename','rowkey','column family:','value'

Put 'tablename','rowkey','column family:column','value'

Scan 'tablename'

scan 'user_test',{COLUMNS =>'info:username',LIMIT =>10, STARTROW =>

'test',STOPROW=>'test2'}

scan 'tablename',{COLUMNS => 'column family'}

Get 'tablename','rowkey'

Get 'tablename','rowkey','column family'

Get 'tablename','rowkey','column family:column'

delete 'tablename','rowkey','column family:column'

```
deleteall 'tablename','rowkey'
```

删除列族: disable 'tablename'

```
Alter 'tablename',{NAME=>'column family',METHOD=>'delete'}
```

删除表: disable 'tablename'

```
drop 'tablename'
```

5. 2. 5hbase 代码开发（基本，过滤器查询）

1.基本增删改查 java 实现 –例子

```
private Configuration conf = null;
@Before
public void init(){
    conf = HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum", "itcast01:2181,itcast02:2181,itcast03:2181");
}

@Test
public void testDrop() throws Exception{
    HBaseAdmin admin = new HBaseAdmin(conf);
    admin.disableTable("account");
    admin.deleteTable("account");
    admin.close();
}
```

2.过滤器查询（基于 scan）

引言：过滤器的类型很多，但是可以分为两大类——**比较过滤器**，**专用过滤器**
过滤器的作用是在服务端判断数据是否满足条件，然后只将满足条件的数据返回给客户端；

hbase 过滤器的**比较运算符**：

LESS < EQUAL= GREATER > NO_OP 排除所有 not <>

Hbase 过滤器的**比较器（指定比较机制）**：

BinaryComparator 按字节索引顺序比较指定字节数组，采用 Bytes.compareTo(byte[]) BinaryPrefixComparator 跟前面相同，只是比较左端的数据是否相同 NullComparator 判断给定的是否为空 BitComparator 按位比较 RegexStringComparator 提供一个正则的比较器，仅支持 EQUAL 和非 EQUAL SubstringComparator 判断提供的子串是否出现在 value 中。

Hbase 的过滤器分类

➤ 比较过滤器

1.1 行键过滤器 RowFilter -- 为例

1.2 列族过滤器 FamilyFilter

1.3 列过滤器 QualifierFilter

1.4 值过滤器 ValueFilter

<pre>Filter filter1 = new RowFilter(CompareOp.LESS_OR_EQUAL, new BinaryComparator(Bytes.toBytes("row-22"))); scan.setFilter(filter1);</pre>

➤ 专用过滤器

2.1 单列值过滤器 SingleColumnValueFilter -----会返回满足条件的整行

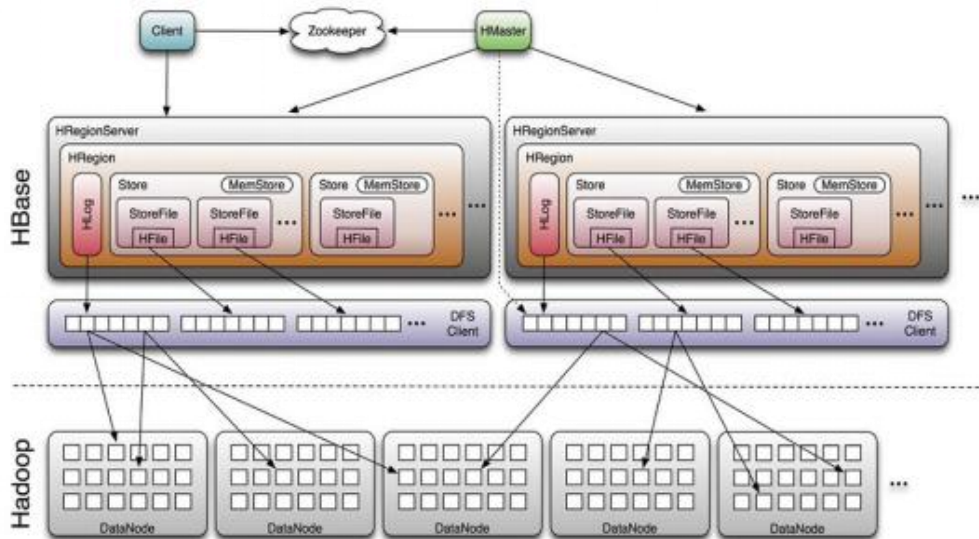
2.2 前缀过滤器 PrefixFilter-----针对行键

2.3 列前缀过滤器 ColumnPrefixFilter

<pre>SingleColumnValueFilter filter = new SingleColumnValueFilter(Bytes.toBytes("colfam1"), Bytes.toBytes("col-5"), CompareFilter.CompareOp.NOT_EQUAL, new SubstringComparator("val-5")); filter.setFilterIfMissing(true); //如果不设置为 true，则那些不包含指定 column 的行也会返回 scan.setFilter(filter1);</pre>

5. 2. 6hbase 内部原理

系统架构



Client

1 包含访问 hbase 的接口，**client** 维护着一些 **cache** 来加快对 **hbase** 的访问，比如 **region** 的位置信息。

Zookeeper

- 1 保证任何时候，集群中只有一个 **master**
- 2 存贮所有 **Region** 的寻址入口----**root** 表在哪台服务器上。
- 3 实时监控 **Region Server** 的状态，将 **Region server** 的上线和下线信息实时通知给 **Master**
- 4 存储 **Hbase** 的 **schema**,包括有哪些 **table**，每个 **table** 有哪些 **column family**

Master 职责

- 1 为 **Region server** 分配 **region**
- 2 负责 **region server** 的负载均衡
- 3 发现失效的 **region server** 并重新分配其上的 **region**
- 4 **HDFS** 上的垃圾文件回收

5 处理 schema 更新请求

Region Server 职责

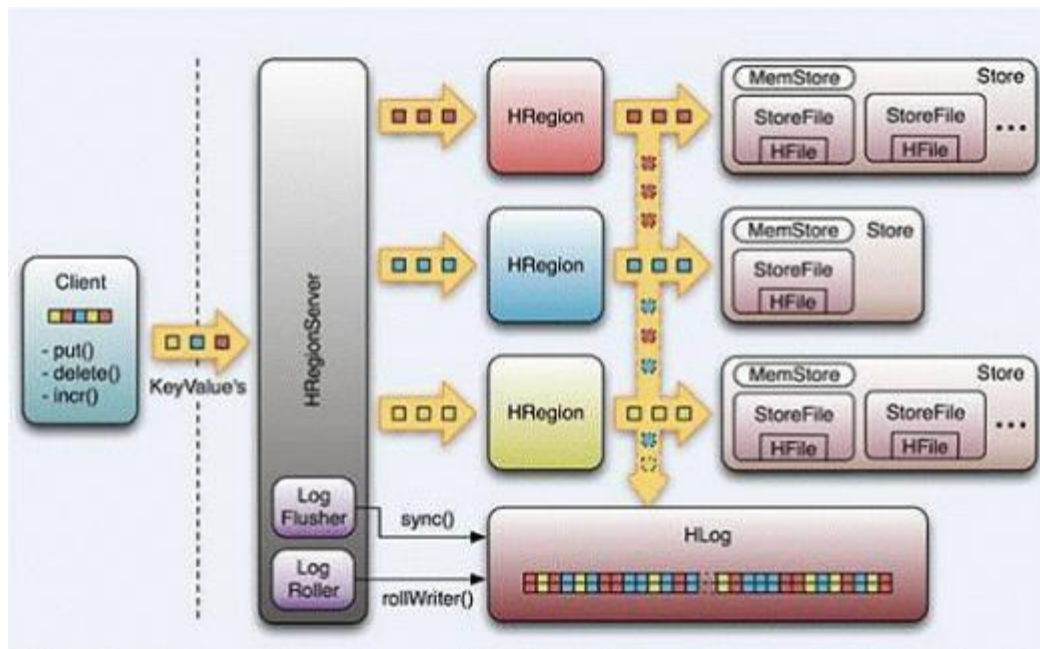
1 Region server 维护 Master 分配给它的 region，处理对这些 region 的 IO 请求

2 Region server 负责切分在运行过程中变得过大的 region

可以看到,client 访问 hbase 上数据的过程并不需要 master 参与(寻址访问 zookeeper 和 region server, 数据读写访问 region server), master 仅仅维护者 table 和 region 的元数据信息, 负载很低。

5. 2. 7物理存储

1、整体结构



1 Table 中的所有行都按照 row key 的字典序排列。

2 Table 在行的方向上分割为多个 Hregion。

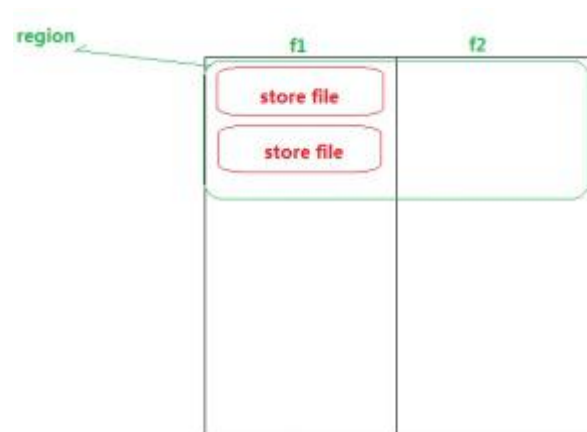
3 region 按大小分割的(默认 10G)，每个表一开始只有一个 region，随着数据不断插入表，region 不断增大, 当增大到一个阈值的时候, Hregion 就会等分会两个新的 Hregion。当 table 中的行不断增多，就会有越来越多的 Hregion。

4 Hregion 是 Hbase 中分布式存储和负载均衡的最小单元。最小单元就表示不同的 Hregion 可以分布在不同的 HRegion server 上。但一个 Hregion 是不会拆分到多个 regionserver 上的。

5 HRegion 虽然是负载均衡的最小单元，但并不是物理存储的最小单元。

事实上，HRegion 由一个或者多个 Store 组成，每个 store 保存一个 column family。

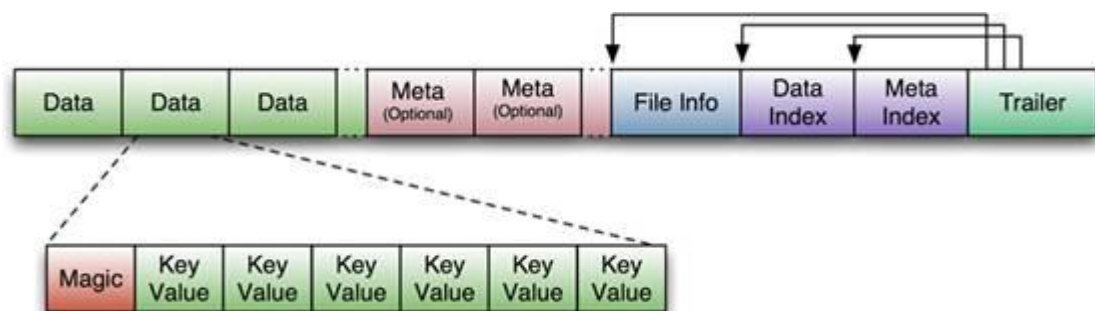
每个 Store 又由一个 memStore 和 0 至多个 StoreFile 组成。如上图



2、STORE FILE & HFILE 结构

StoreFile 以 HFile 格式保存在 HDFS 上。

附：HFile 的格式为：



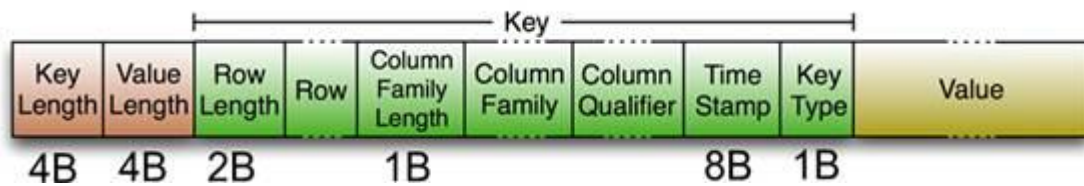
首先 HFile 文件是不定长的，长度固定的只有其中的两块：Trailer 和 FileInfo。正如图中所示的，Trailer 中有指针指向其他数据块的起始点。

File Info 中记录了文件的一些 Meta 信息，例如：AVG_KEY_LEN, AVG_VALUE_LEN, LAST_KEY, COMPARATOR, MAX_SEQ_ID_KEY 等。

Data Index 和 Meta Index 块记录了每个 Data 块和 Meta 块的起始点。

Data Block 是 HBase I/O 的基本单元，为了提高效率，HRegionServer 中有基于 LRU 的 Block Cache 机制。每个 Data 块的大小可以在创建一个 Table 的时候通过参数指定，大号的 Block 有利于顺序 Scan，小号 Block 利于随机查询。每个 Data 块除了开头的 Magic 以外就是一

个个 KeyValue 对拼接而成, Magic 内容就是一些随机数字, 目的是防止数据损坏。
HFile 里面的每个 KeyValue 对就是一个简单的 byte 数组。但是这个 byte 数组里面包含了很多项, 并且有固定的结构。我们来看看里面的具体结构:



开始是两个固定长度的数值, 分别表示 Key 的长度和 Value 的长度。紧接着是 Key, 开始是固定长度的数值, 表示 RowKey 的长度, 紧接着是 RowKey, 然后是固定长度的数值, 表示 Family 的长度, 然后是 Family, 接着是 Qualifier, 然后是两个固定长度的数值, 表示 Time Stamp 和 Key Type (Put/Delete)。Value 部分没有这么复杂的结构, 就是纯粹的二进制数据了。

HFile 分为六个部分:

Data Block 段-保存表中的数据, 这部分可以被压缩

Meta Block 段 (可选的)-保存用户自定义的 kv 对, 可以被压缩。

File Info 段-Hfile 的元信息, 不被压缩, 用户也可以在这一部分添加自己的元信息。

Data Block Index 段-Data Block 的索引。每条索引的 key 是被索引的 block 的第一条记录的 key。

Meta Block Index 段 (可选的)-Meta Block 的索引。

Trailer-这一段是定长的。保存了每一段的偏移量, 读取一个 HFile 时, 会首先 读取 Trailer, Trailer 保存了每个段的起始位置(段的 Magic Number 用来做安全 check), 然后, DataBlock Index 会被读取到内存中, 这样, 当检索某个 key 时, 不需要扫描整个 HFile, 而只需从内存中找到 key 所在的 block, 通过一次磁盘 io 将整个 block 读取到内存中, 再找到需要的 key。DataBlock Index 采用 LRU 机制淘汰。

HFile 的 Data Block, Meta Block 通常采用压缩方式存储, 压缩之后可以大大减少网络 IO 和磁盘 IO, 随之而来的开销当然是需要花费 cpu 进行压缩和解压缩。

目标 Hfile 的压缩支持两种方式: Gzip, Lzo。

3、Memstore 与 storefile

一个 region 由多个 store 组成, 每个 store 包含一个列族的所有数据。

Store 包括位于内存的 memstore 和位于硬盘的 storefile。

写操作先写入 memstore, 当 memstore 中的数据量达到某个阈值, Hregionserver 启动 flashcache 进程写入 storefile, 每次写入形成单独一个 storefile。

当 storefile 的个数超过一定阈值后(默认参数 `hbase.hstore.blockingStoreFiles=10`), 多个 storeFile 会进行合并, 当该 region 的所有 store 的 storefile 大小之和, 即所有 store 的大小超过 `hbase.hregion.max.filesize=10G` 时, 这个 region 会被拆分会把当前的 region

分割成两个，并由 Hmaster 分配给相应的 region 服务器，实现负载均衡。

客户端检索数据时，先在 memstore 找，找不到再找 storefile。

4、HLog(WAL log)

WAL 意为 Write ahead log(http://en.wikipedia.org/wiki/Write-ahead_logging)，该机制用于数据的容错和恢复，Hlog 记录数据的所有变更，一旦数据修改，就可以从 log 中进行恢复。

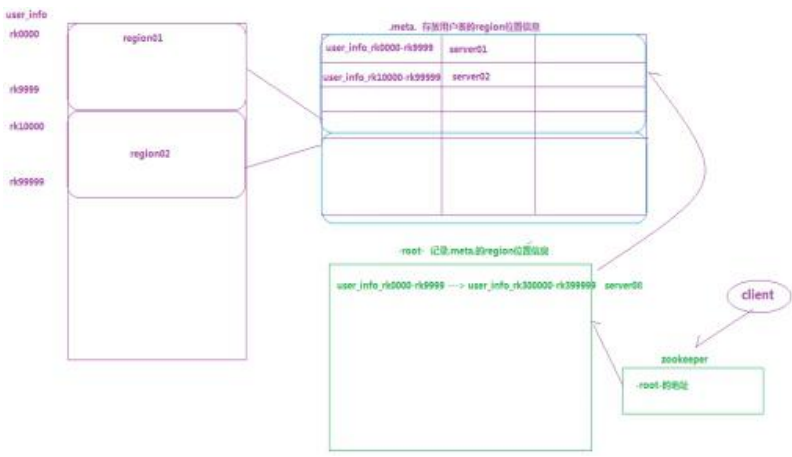
每个 HRegionServer 中都有一个 HLog 对象，HLog 是一个实现 Write Ahead Log 的类，在每次用户操作写入 MemStore 的同时，也会写一份数据到 HLog 文件中（HLog 文件格式见后续），HLog 文件定期会滚动出新的，并删除旧的文件（已持久化到 StoreFile 中的数据）。当 HRegionServer 意外终止后，HMaster 会通过 Zookeeper 感知到，HMaster 首先会处理遗留的 HLog 文件，将其中不同 Region 的 Log 数据进行拆分，分别放到相应 region 的目录下，然后再将失效的 region 重新分配，领取到这些 region 的 HRegionServer 在 Load Region 的过程中，会发现有历史 HLog 需要处理，因此会 Replay HLog 中的数据到 MemStore 中，然后 flush 到 StoreFiles，完成数据恢复。

HLog 文件就是一个普通的 Hadoop Sequence File：

- ✧ HLog Sequence File 的 Key 是 HLogKey 对象，HLogKey 中记录了写入数据的归属信息，除了 table 和 region 名字外，同时还包括 sequence number 和 timestamp，timestamp 是“写入时间”，sequence number 的起始值为 0，或者是最近一次存入文件系统中 sequence number。
- ✧ HLog Sequece File 的 Value 是 HBase 的 KeyValue 对象，即对应 HFile 中的 KeyValue，可参见上文描述。

5. 2. 8 寻址机制

1、寻址示意图

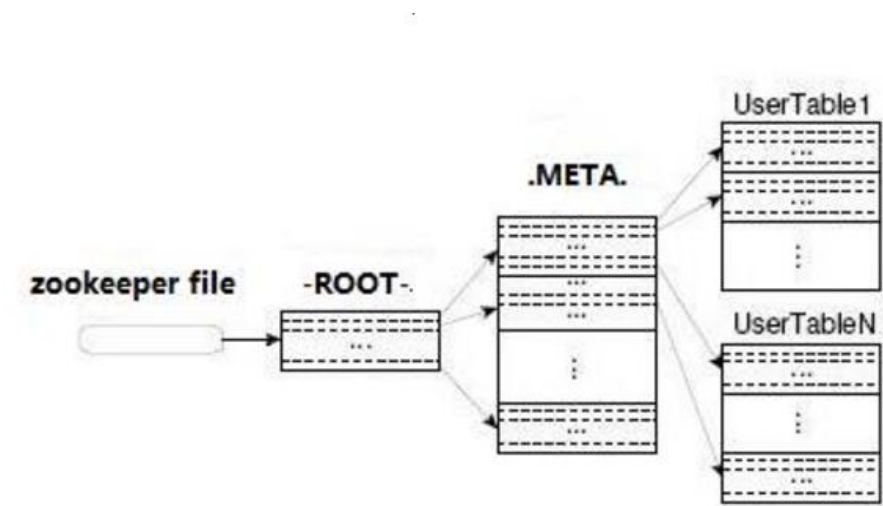


2、-ROOT-和.META.表结构

-ROOT-表结构 : rowkey info(regioninfo server serverstartcode) historiam

.META. 行记录结构 : rowkey info(regioninfo server serverstartcode) historiam

3、寻址流程



系统如何找到某个 row key (或者某个 row key range) 所在的 region

bigtable 使用三层类似 B+树的结构来保存 region 位置。

第一层: 保存 zookeeper 里面的文件, 它持有 root region 的位置。

第二层: root region 是 .META. 表的第一个 region 其中保存了 .META. 表其它 region 的位置。
通过 root region, 我们就可以访问 .META. 表的数据。

第三层: .META. 表它是一个特殊的表, 保存了 hbase 中所有数据表的 region 位置信息。

说明:

- (1) root region 永远不会被 split, 保证了最需要三次跳转, 就能定位到任意 region 。
- (2) .META. 表每行保存一个 region 的位置信息, row key 采用表名+表的最后一行编码而成。
- (3) 为了加快访问, .META. 表的全部 region 都保存在内存中。
- (4) client 会将查询过的位置信息保存缓存起来, 缓存不会主动失效, 因此如果 client 上的缓存全部失效, 则需要进行最多 6 次网络来回, 才能定位到正确的 region(其中三次用来发现缓存失效, 另外三次用来获取位置信息)。

5.2.9 读写过程

1、读请求过程:

- (1) 客户端通过 zookeeper 以及 root 表和 meta 表找到目标数据所在的 regionserver
- (2) 联系 regionserver 查询目标数据
- (3) regionserver 定位到目标数据所在的 region, 发出查询请求
- (4) region 先在 memstore 中查找, 命中则返回
- (5) 如果在 memstore 中找不到, 则在 storefile 中扫描 (可能会扫描到很多的 storefile----bloomfilter 布隆过滤器)

补充: 布隆过滤器参数类型有 2 种:

Row、row+col

2、写请求过程：

- (1) client 向 region server 提交写请求
- (2) region server 找到目标 region
- (3) region 检查数据是否与 schema 一致
- (4) 如果客户端没有指定版本，则获取当前系统时间作为数据版本
- (5) 将更新写入 WAL log
- (6) 将更新写入 Memstore
- (7) 判断 Memstore 的是否需要 flush 为 StoreFile 文件。

细节描述：

hbase 使用 MemStore 和 StoreFile 存储对表的更新。

数据在更新时首先写入 Log (WAL log) 和内存 (MemStore) 中，MemStore 中的数据是排序的，当 MemStore 累计到一定阈值时，就会创建一个新的 MemStore，并且将老的 MemStore 添加到 flush 队列，由单独的线程 flush 到磁盘上，成为一个 StoreFile。于此同时，系统会在 zookeeper 中记录一个 redo point，表示这个时刻之前的变更已经持久化了。

当系统出现意外时，可能导致内存 (MemStore) 中的数据丢失，此时使用 Log (WAL log) 来恢复 checkpoint 之后的数据。

StoreFile 是只读的，一旦创建后就不可以再修改。因此 Hbase 的更新其实是不断追加的操作。当一个 Store 中的 StoreFile 达到一定的阈值后，就会进行一次合并 (minor_compact, major_compact)，将对同一个 key 的修改合并到一起，形成一个大的 StoreFile，当 StoreFile 的大小达到一定阈值后，又会对 StoreFile 进行 split，等分为两个 StoreFile。

由于对表的更新是不断追加的，compact 时，需要访问 Store 中全部的 StoreFile 和

MemStore，将他们按 row key 进行合并，由于 StoreFile 和 MemStore 都是经过排序的，并且 StoreFile 带有内存中索引，合并的过程还是比较快。

5.2.10 Region 管理

(1) region 分配

任何时刻，一个 region 只能分配给一个 region server。master 记录了当前有哪些可用的 region server。以及当前哪些 region 分配给了哪些 region server，哪些 region 还没有分配。当需要分配的新的 region，并且有一个 region server 上有可用空间时，master 就给这个 region server 发送一个装载请求，把 region 分配给这个 region server。region server 得到请求后，就开始对此 region 提供服务。

(2) region server 上线

master 使用 zookeeper 来跟踪 region server 状态。当某个 region server 启动时，会首先在 zookeeper 上的 server 目录下建立代表自己的 znode。由于 master 订阅了 server 目录上的变更消息，当 server 目录下的文件出现新增或删除操作时，master 可以得到来自 zookeeper 的实时通知。因此一旦 region server 上线，master 能马上得到消息。

(3) region server 下线

当 region server 下线时，它和 zookeeper 的会话断开，zookeeper 而自动释放代表这台 server 的文件上的独占锁。master 就可以确定：

- 1 region server 和 zookeeper 之间的网络断开了。

- 2 region server 挂了。

无论哪种情况，region server 都无法继续为它的 region 提供服务了，此时 master 会删除 server 目录下代表这台 region server 的 znode 数据，并将这台 region server 的 region 分配给其它还活着的同志。

5.2.11 Master 工作机制

➤ master 上线

master 启动进行以下步骤：

- (1) 从 zookeeper 上获取唯一一个代表 active master 的锁，用来阻止其它 master 成为活着的 master。
- (2) 扫描 zookeeper 上的 server 父节点，获得当前可用的 region server 列表。
- (3) 和每个 region server 通信，获得当前已分配的 region 和 region server 的对应关系。
- (4) 扫描 META.region 的集合，计算得到当前还未分配的 region，将他们放入待分配 region 列表。

➤ master 下线

由于 master 只维护表和 region 的元数据，而不参与表数据 IO 的过程，master 下线仅导致所有元数据的修改被冻结（无法创建删除表，无法修改表的 schema，无法进行 region 的负载均衡，无法处理 region 上下线，无法进行 region 的合并，唯一例外的是 region 的 split 可以正常进行，因为只有 region server 参与），表的数据读写还可以正常进行。因此 master 下线短时间内对整个 hbase 集群没有影响。

5.2.12 HBase 容错性

Master 容错：Zookeeper 重新选择一个新的 Master

无 Master 过程中，数据读取仍照常进行；

无 Master 过程中，region 切分、负载均衡等无法进行；

RegionServer 容错：定时向 Zookeeper 汇报心跳，如果一旦时间内未出现心跳，Master 将该 RegionServer 上的 Region 重新分配到其他 RegionServer 上，失效服务器上“预写”日志由主服务器进行分割并派送给新的 RegionServer

Zookeeper 容错：Zookeeper 是一个可靠地服务，一般配置 3 或 5 个 Zookeeper 实例

5.3 Hbase 高级应用

5.3.1 建表高级属性

下面几个 shell 命令在 hbase 操作中可以起到很到的作用，且主要体现在建表的过程中，看下面几个 create 属性

1、BLOOMFILTER 默认是 NONE 是否使用布隆过滤及使用何种方式

布隆过滤可以每列族单独启用。

使用 `HColumnDescriptor.setBloomFilterType(NONE | ROW | ROWCOL)` 对列族单独启用布隆。

✧ Default = ROW 对行进行布隆过滤。

✧ 对 ROW，行键的哈希在每次插入行时将被添加到布隆。

✧ 对 ROWCOL，行键 + 列族 + 列族修饰的哈希将在每次插入行时添加到布隆

使用方法: `create 'table',{BLOOMFILTER =>'ROW'}`

启用布隆过滤可以节省读磁盘过程，可以有助于降低读取延迟

2、VERSIONS 默认是 1 这个参数的意思是数据保留 1 个版本，如果我们认为我们的数据没有这么大的必要保留这么多，随时都在更新，而老版本的数据对我们毫无价值，那将此参数设为 1 能节约 2/3 的空间

使用方法: `create 'table',{VERSIONS=>'2'}`

附：MIN_VERSIONS => '0'是说在 compact 操作执行之后，至少要保留的版本

3、COMPRESSION 默认值是 NONE 即不使用压缩

这个参数意思是该列族是否采用压缩，采用什么压缩算法

使用方法: `create 'table',{NAME=>'info',COMPRESSION=>'SNAPPY'}`

建议采用 SNAPPY 压缩算法

HBase 中，在 Snappy 发布之前（Google 2011 年对外发布 Snappy），采用的 LZO 算法，目标是达到尽可能快的压缩和解压速度，同时减少对 CPU 的消耗；

在 Snappy 发布之后，建议采用 Snappy 算法（参考《HBase: The Definitive Guide》），具体可以根据实际情况对 LZO 和 Snappy 做过更详细的对比测试后再做选择。

Algorithm	% remaining	Encoding	Decoding
GZIP	13.4%	21 MB/s	118 MB/s
LZO	20.5%	135 MB/s	410 MB/s
Zippy/Snappy	22.2%	172 MB/s	409 MB/s

如果建表之初没有压缩，后来想要加入压缩算法，可以通过 alter 修改 schema

4、alter

使用方法：如 修改压缩算法


```
disable 'table'
alter 'table',{NAME=>'info',COMPRESSION=>'snappy'}
enable 'table'
```

但是需要执行 `major_compact 'table'` 命令之后 才会做实际的操作。

5、TTL

默认是 2147483647 即:Integer.MAX_VALUE 值大概是 68 年

这个参数是说明该列族数据的存活时间, 单位是 s

这个参数可以根据具体的需求对数据设定存活时间, 超过存活时间的数据将在表中不再显示, 待下次 `major compact` 的时候再彻底删除数据.

注意的是 TTL 设定之后 `MIN_VERSIONS=>'0'` 这样设置之后, TTL 时间戳过期后, 将全部彻底删除该 family 下所有的数据, 如果 `MIN_VERSIONS` 不等于 0 那将保留最新的 `MIN_VERSIONS` 个版本的数据, 其它的全部删除, 比如 `MIN_VERSIONS=>'1'` 届时将保留一个最新版本的数据, 其它版本的数据将不再保存。

6、`describe 'table'` 这个命令查看了 create table 的各项参数或者是默认值。

7、`disable_all 'toplist.*'` `disable_all` 支持正则表达式, 并列出当前匹配的表:

```
toplist_a_total_1001
```

...

Disable the above 25 tables (y/n)? 并给出确认提示.

8、`drop_all` 这个命令和 `disable_all` 的使用方式是一样的

9、hbase 表预分区 ----手动分区

默认情况下, 在创建 HBase 表的时候会自动创建一个 region 分区, 当导入数据的时候, 所有的 HBase 客户端都向这一个 region 写数据, 直到这个 region 足够大了才进行切分。一种可以加快批量写入速度的方法是通过预先创建一些空的 regions, 这样当数据写入 HBase 时, 会按照 region 分区情况, 在集群内做数据的负载均衡。

命令方式:

```
create 't1', 'f1', {NUMREGIONS => 15, SPLITALGO => 'HexStringSplit'}
```

也可以使用 api 的方式:

```
bin/hbase org.apache.hadoop.hbase.util.RegionSplitter test_table HexStringSplit -c 10
-f info
```

参数:

HexStringSplit:split 方式 -c 是分 10 个 region -f 是 family

这样就可以将表预先分为 15 个区,减少数据达到 storefile 大小的时候自动分区的时间消耗,并且还有一个优势,就是合理设计 rowkey 能让各个 region 的并发请求平均分配(趋于均匀)使 IO 效率达到最高,但是预分区需要将 filesize 设置一个较大的值,设置哪个参数呢,hbase.hregion.max.filesize 这个值默认是 10G 也就是说单个 region 默认大小是 10G,

这个参数的默认值在 0.90 到 0.92 到 0.94.3 各版本的变化: 256M--1G--10G

但是如果 MapReduce Input 类型为 TableInputFormat 使用 hbase 作为输入的时候,就要注意了,每个 region 一个 map,如果数据小于 10G 那只会启用一个 map 造成很大的资源浪费,这时候可以考虑适当调小该参数的值,或者采用预分配 region 的方式,并将检测如果达到这个值,再手动分配 region。

5.3.2 HBase 的设计原则

HBase 是三维有序存储的,通过 rowkey(行键),column key(column family 和 qualifier)和 Timestamp(时间戳)这三个维度可以对 HBase 中的数据进行快速定位。

HBase 中 rowkey 可以唯一标识一行记录,在 HBase 查询的时候,有以下几种方式:

1. 通过 get 方式,指定 rowkey 获取唯一一条记录
2. 通过 scan 方式,设置 startRow 和 stopRow 参数进行范围匹配
3. 全表扫描,即直接扫描整张表中所有行记录

5.3.3 rowkey 长度原则

rowkey 是一个二进制码流,可以是任意字符串,最大长度 64kb,实际应用

中一般为 10-100bytes,以 byte[]形式保存,一般设计成定长。

建议越短越好,不要超过 16 个字节,原因如下:

- ❖ 数据的持久化文件 HFile 中是按照 KeyValue 存储的，如果 rowkey 过长，比如超过 100 字节，1000w 行数据，光 rowkey 就要占用 $100 \times 1000w = 10$ 亿个字节，将近 1G 数据，这样会极大影响 HFile 的存储效率；
- ❖ MemStore 将缓存部分数据到内存，如果 rowkey 字段过长，内存的有效利用率就会降低，系统不能缓存更多的数据，这样会降低检索效率。

1.rowkey 散列原则

如果 rowkey 按照时间戳的方式递增，不要将时间放在二进制码的前面，建议将 rowkey 的高位作为散列字段，由程序随机生成，低位放时间字段，这样将提高数据均衡分布在每个 RegionServer，以实现负载均衡的几率。如果没有散列字段，首字段直接是时间信息，所有的数据都会集中在一个 RegionServer 上，这样在数据检索的时候负载会集中在个别的 RegionServer 上，造成热点问题，会降低查询效率。

2.rowkey 唯一原则

必须在设计上保证其唯一性，rowkey 是按照字典顺序排序存储的，因此，设计 rowkey 的时候，要充分利用这个排序的特点，将经常读取的数据存储到一块，将最近可能会被访问的数据放到一块。

5.3.4 什么是热点

HBase 中的行是按照 rowkey 的字典顺序排序的，

热点发生在大量的 client 直接访问集群的一个或极少数个节点（访问可能是读，写或者其他操作）。大量访问会使热点 region 所在的单个机器超出自身承受能力，引起性能下降甚至 region 不可用，这也会影响同一个 RegionServer 上的其他 region，由于主机无法服务其他 region 的请求。

避免热点的方法

1) 加盐

在 rowkey 的前面增加随机数，具体就是给 rowkey 分配一个随机前缀以使得

它和之前的 rowkey 的开头不同。分配的前缀种类数量应该和你想使用数据分散到不同的 region 的数量一致。加盐之后的 rowkey 就会根据随机生成的前缀分散到各个 region 上，以避免热点。

2) 哈希

哈希会使同一行永远用一个前缀加盐。哈希也可以使负载分散到整个集群，但是读却是可以预测的。使用确定的哈希可以让客户端重构完整的 rowkey，可以使用 get 操作准确获取某一个行数据。

3) 反转

第三种防止热点的方法时反转固定长度或者数字格式的 rowkey。这样可以使得 rowkey 中经常改变的部分（最没有意义的部分）放在前面。这样可以有效的随机 rowkey，但是牺牲了 rowkey 的有序性。

反转 rowkey 的例子以手机号为 rowkey，可以将手机号反转后的字符串作为 rowkey，这样的就避免了以手机号那样比较固定开头导致热点问题

4) 时间戳反转

一个常见的数据处理问题是快速获取数据的最近版本，使用反转的时间戳作为 rowkey 的一部分对这个问题十分有用，可以用 Long.MaxValue - timestamp 追加到 key 的末尾，例如 [key][reverse_timestamp]，[key] 的最新值可以通过 scan [key] 获得[key]的第一条记录，因为 HBase 中 rowkey 是有序的，第一条记录是最后录入的数据。

列族尽可能越短越好，最好是一个字符，冗长的属性名虽然可读性好，但是

更短的属性名存储在 HBase 中会更好。

5.4 HBase 性能优化完全版

5.4.1 垃圾回收优化

Java 本身提供了垃圾回收机制，依靠 JRE 对程序行为的各种假设进行垃圾回收，但是 HBase 支持海量数据持续入库，非常占用内存，因此繁重的负载会迫使内存分配策略无法安全地依赖于 JRE 的判断：需要调整 JRE 的参数来调整垃圾回收策略。有关 java 内存回收机制的问题具体请参考：<http://my.oschina.net/sunnywu/blog/332870>。

(1) HBASE_OPTS 或者 HBASE_REGIONSERVER_OPT 变量来设置垃圾回收的选项，后面一般是用于配置 RegionServer 的，需要在每个子节点的 HBASE_OPTS 文件中进行配置。

1) 首先是设置新生代大小的参数，不能过小，过小则导致年轻代过快成为老生代，引起老生代产生内存随便。同样不能过大，过大导致所有的 JAVA 进程停止时间长。-XX:MaxNewSize=256m-XX:NewSize=256m 这两个可以合并成为-Xmn256m 这一个配置来完成。

2) 其次是设置垃圾回收策略：-XX:+UseParNewGC -XX:+UseConcMarkSweepGC 也叫收集器设置。

3) 设置 CMS 的值，占比多少时，开始并发标记和清扫检查。-XX:CMSInitiatingOccupancyFraction=70

4) 打印垃圾回收信息: `-verbose:gc -XX: +PrintGCDetails -XX:+PrintGCTimeStamps`

`-Xloggc:$HBASE_HOME/logs/gc-$(hostname)-hbase.log`

最终可以得到: `HBASE_REGIONSERVER_OPT="-Xmx8g -Xms8g -Xmn256m -XX:+UseParNewGC -XX:+UseConcMarkSweepGC \`

`-XX:CMSInitiatingOccupancyFraction=70 -verbose:gc \`

`-XX:+PrintGCDetails -XX:+PrintGCTimeStamps \`

`-Xloggc:$HBASE_HOME/logs/gc-$(hostname)-hbase.log`

(2) `hbase.hregion.memstore.mslab.enabled` 默认值: `true`, 这个是在 `hbase-site.xml` 中进行配置的值。

说明: 减少因内存碎片导致的 Full GC, 提高整体性能。

5.4.2 启用压缩, 详情自行搜索, 暂时未曾尝试, 后面持续更新。

5.4.3 优化 Region 拆分合并以及与拆分 Region

(1) `hbase.hregion.max.filesize` 默认为 256M (在 `hbase-site.xml` 中进行配置), 当 region 达到这个阈值时, 会自动拆分。可以把这个值设的无限大, 则可以关闭 HBase 自动管理拆分, 手动运行命令来进行 region 拆分, 这样可以在不同的 region 上交错运行, 分散 I/O 负载。

(2) 预拆分 region

用户可以在建表的时候就制定好预设定的 region，这样就可以避免后期 region 自动拆分造成 I/O 负载。

5.4.4 4.客户端入库调优

(1) 用户在编写程序入库时，HBase 的自动刷写是默认开启的，即用户每一次 put 都会提交到 HBase server 进行一次刷写，如果需要高速插入数据，则会造成 I/O 负载过重。在这里可以关闭自动刷写功能，`setAutoFlush(false)`。如此，put 实例会先写到一个缓存中，这个缓存的大小通过 `hbase.client.write.buffer` 这个值来设定缓存区，当缓存区被填满之后才会被送出。如果想要显示刷写数据，可以调用 `flushCommits()` 方法。

此处引申：采取这个方法要估算服务器端内存占用则可以：`hbase.client.write.buffer*hbase.regionserver.handler.count` 得出内存情况。

(2) 第二个方法，是关闭每次 put 上的 WAL (`writeToWAL(false)`) 这样可以刷写数据前，不需要预写日志，但是如果数据重要的话建议不要关闭。

(3) `hbase.client.scanner.caching`：默认为 1

这是设计客户端读取数据的配置调优，在 `hbase-site.xml` 中进行配置，代表 scanner 一次缓存多少数据（从服务器一次抓取多少数据来 scan）默认的太小，但是对于大文件，值不应太大。

(4) `hbase.regionserver.lease.period` 默认值：60000

说明：客户端租用 HRegion server 期限，即超时阈值。

调优：这个配合 `hbase.client.scanner.caching` 使用，如果内存够大，但是取出较多数据后计算过程较长，可能超过这个阈值，适当可设置较长的响应时间以防被认为宕机。

(5) 还有诸多实践，如设置过滤器，扫描缓存等，指定行扫描等多种客户端调优方案，需要在实践中慢慢挖掘。

5.4.5 HBase 配置文件

上面涉及到的调优内容或多或少在 HBase 配置文件中都有所涉及，因此，下面的配置不涵盖上面已有的配置。

(1) `zookeeper.session.timeout` (默认 3 分钟)

ZK 的超期参数，默认配置为 3 分钟，在生产环境上建议减小这个值在 1 分钟或更小。

设置原则：这个值越小，当 RS 故障时 Hmaster 获知越快，Hlog 分裂和 region 部署越快，集群恢复时间越短。但是，设置这个值得原则是留足够的时间进行 GC 回收，否则会导致频繁的 RS 宕机。一般就做默认即可

(2) `hbase.regionserver.handler.count` (默认 10)

对于大负载的 put (达到了 M 范围) 或是大范围的 Scan 操作，handler 数目不易过大，易造成 OOM。对于小负载的 put 或是 get, delete 等操作，handl

er 数要适当调大。根据上面的原则，要看我们的业务的情况来设置。（具体情况具体分析）。

(3) HBASE_HEAPSIZE (hbase-env.sh 中配置)

我的前两篇文章 MemstoreSize40% (默认) blockcache 20% (默认) 就是依据这个而成的，总体 HBase 内存配置。设到机器内存的 1/2 即可。

(4) 选择使用压缩算法，目前 HBase 默认支持的压缩算法包括 GZ, LZO 以及 snappy (hbase-site.xml 中配置)

(5) hbase.hregion.max.filesize 默认 256M

上面说过了，hbase 自动拆分 region 的阈值，可以设大或者无限大，无限大需要手动拆分 region，懒的人别这样。

(6) hbase.hregion.memstore.flush.size

单个 region 内所有的 memstore 大小总和超过指定值时，flush 该 region 的所有 memstore。

(7) hbase.hstore.blockingStoreFiles 默认值：7

说明：在 flush 时，当一个 region 中的 Store (ColumnFamily) 内有超过 7 个 storefile 时，则 block 所有的写请求进行 compaction，以减少 storefile 数量。

调优：block 写请求会严重影响当前 regionServer 的响应时间，但过多的 storefile 也会影响读性能。从实际应用来看，为了获取较平滑的响应时间，可将值设为无限大。如果能容忍响应时间出现较大的波峰波谷，那么默认或根据自身场景调整即可。

(8) hbase.hregion.memstore.block.multiplier 默认值：2

说明：当一个 region 里总的 memstore 占用内存大小超过 hbase.hregion.memstore.flush.size 两倍的大小时，block 该 region 的所有请求，进行 flush，释放内存。

虽然我们设置了 region 所占用的 memstores 总内存大小，比如 64M，但想象一下，在最后 63.9M 的时候，我 Put 了一个 200M 的数据，此时 memstore 的大小会瞬间暴涨到超过预期的 hbase.hregion.memstore.flush.size 的几倍。这个参数的作用是当 memstore 的大小增至超过 hbase.hregion.memstore.flush.size 2 倍时，block 所有请求，遏制风险进一步扩大。

调优：这个参数的默认值还是比较靠谱的。如果你预估你的正常应用场景（不包括异常）不会出现突发写或写的量可控，那么保持默认值即可。如果正常情况下，你的写请求量就会经常暴长到正常的几倍，那么你应该调大这个倍数并调整其他参数值，比如 hfile.block.cache.size 和 hbase.regionserver.global.memstore.upperLimit/lowerLimit，以预留更多内存，防止 HBase server OOM。

(9) hbase.regionserver.global.memstore.upperLimit：默认 40%

当 RegionServer 内所有 region 的 memstores 所占用内存总和达到 heap 的 40% 时，HBase 会强制 block 所有的更新并 flush 这些 region 以释放所有 memstore 占用的内存。

hbase.regionserver.global.memstore.lowerLimit: 默认 35%

同 upperLimit，只不过 lowerLimit 在所有 region 的 memstores 所占用内存达到 Heap 的 35% 时，不 flush 所有的 memstore。它会找一个 memstore 内存占用最大的 region，做个别 flush，此时写更新还是会被 block。lowerLimit 算是一个在所有 region 强制 flush 导致性能降低前的补救措施。在日志中，表现为 “** Flushtread woke up with memory above low water.”。

调优：这是一个 Heap 内存保护参数，默认值已经能适用大多数场景。

(10) hfile.block.cache.size: 默认 20%

这是涉及 hbase 读取文件的主要配置，BlockCache 主要提供给读使用。读请求先到 memstore 中查数据，查不到就到 blockcache 中查，再查不到就会到磁盘上读，并把读的结果放入 blockcache。由于 blockcache 是一个 LRU，因此 blockcache 达到上限($\text{heapsize} * \text{hfile.block.cache.size}$)后，会启动淘汰机制，淘汰掉最老的一批数据。对于注重读响应时间的系统，应该将 blockcache 设大些，比如设置 blockcache=0.4，memstore=0.39，这会加大缓存命中率。

(11) hbase.regionserver.hlog.blocksize 和 hbase.regionserver.maxlogs

之所以把这两个值放在一起，是因为 WAL 的最大值由 `hbase.regionserver.maxlogs*hbase.regionserver.hlog.blocksize` (2GB by default)决定。一旦达到这个值，Memstore flush 就会被触发。所以，当你增加 Memstore 的大小以及调整其他的 Memstore 的设置项时，你也需要去调整 HLog 的配置项。否则，WAL 的大小限制可能会首先被触发，因而，你将利用不到其他专门为 Memstore 而设计的优化。抛开这些不说，通过 WAL 限制来触发 Memstore 的 flush 并非最佳方式，这样做可能会一次 flush 很多 Region，尽管“写数据”是很好的分布于整个集群，进而很有可能会引发 flush “大风暴”。

提示：最好将 `hbase.regionserver.hlog.blocksize* hbase.regionserver.maxlogs` 设置为稍微大于 `hbase.regionserver.global.memstore.lowerLimit* HBASE_HEAPSIZE`。

5.4.6 HDFS 优化部分

HBase 是基于 hdfs 文件系统的一个数据库，其数据最终是写到 hdfs 中的，因此涉及 hdfs 调优的部分也是必不可少的。

(1) `dfs.replication.interval`:默认 3 秒

可以调高，避免 hdfs 频繁备份，从而提高吞吐率。

(2) `dfs.datanode.handler.count`:默认为 10

可以调高这个处理线程数，使得写数据更快

(3) `dfs.namenode.handler.count`: 默认为 8

(4) `dfs.datanode.socket.write.timeout`: 默认 480 秒, 并发写数据量大的时候可以调高一些, 否则会出现我另外一篇博客介绍的错误。

(5) `dfs.socket.timeout`: 最好也要调高, 默认的很小。

5.5 大数据 Hbase 面试题

hbase 的特点是什么

- (1) Hbase 一个分布式的基于列式存储的数据库, 基于 Hadoop 的 hdfs 存储, zookeeper 进行管理。
- (2) Hbase 适合存储半结构化或非结构化数据, 对于数据结构字段不够确定或者杂乱无章很难按一个概念去抽取的数据。
- (3) Hbase 为 null 的记录不会被存储。
- (4) 基于的表包含 rowkey, 时间戳, 和列族。新写入数据时, 时间戳更新, 同时可以查询到以前的版本。
- (5) hbase 是主从架构。hmaster 作为主节点, hregionserver 作为从节点

hbase 如何导入数据

使用 MapReduce Job 方式, 根据 Hbase API 编写 java 脚本, 将文本文件用文件流的方式截取, 然后存储到多个字符串数组中, 在 put 方法下, 通过对表中的列族进行 for 循环遍历列名, 用 if 判断列名后进行 for 循环调用 put.add 的方法对列族下每一个列进行设值, 每个列族下有几个了就赋值几次! 没有表先对先创建表。

hbase 的存储结构?

答: Hbase 中的每张表都通过行键(rowkey)按照一定的范围被分割成多个子表 (HRegion), 默认一个 HRegion 超过 256M 就要被分割成两个, 由 HRegionServer 管理, 管理哪些 HRegion

由 Hmaster 分配。HRegion 存取一个子表时，会创建一个 HRegion 对象，然后对表的每个列族（Column Family）创建一个 store 实例，每个 store 都会有 0 个或多个 StoreFile 与之对应，每个 StoreFile 都会对应一个 HFile，HFile 就是实际的存储文件，因此，一个 HRegion 还拥有有一个 MemStore 实例。

Hbase 和 hive 有什么区别

问题导读：

hive 与 hbase 的底层存储是什么？

hive 是产生的原因是什么？

hbase 是为了弥补 hadoop 的什么缺陷

答案：

共同点：

1.hbase 与 hive 都是架构在 hadoop 之上的。都是用 hadoop 作为底层存储

区别：

2.Hive 是建立在 Hadoop 之上为了减少 MapReducejobs 编写工作的批处理系统，

HBase 是为了支持弥补 Hadoop 对实时操作的缺陷的项目。

3.想象你在操作 RMDB 数据库，如果是全表扫描，就用 Hive+Hadoop,如果是索引访问，就用 HBase+Hadoop。

4.Hive query 就是 MapReduce jobs 可以从 5 分钟到数小时不止，

HBase 是非常高效的，肯定比 Hive 高效的多。

5.Hive 本身不存储和计算数据，它完全依赖于 HDFS 和 MapReduce，Hive 中的表纯逻辑。

6.hive 借用 hadoop 的 MapReduce 来完成一些 hive 中的命令的执行

7.hbase 是物理表，不是逻辑表，提供一个超大的内存 hash 表，搜索引擎通过它来存储索引，方便查询操作。

8.hbase 是列存储。

9.hdfs 作为底层存储，hdfs 是存放文件的系统，而 Hbase 负责组织文件。

10.hive 需要用到 hdfs 存储文件，需要用到 MapReduce 计算框架。

解释下 hbase 实时查询的原理

答：实时查询，可以认为是从内存中查询，一般响应时间在 1 秒内。HBase 的机制是数据先写入到内存中，当数据量达到一定的量（如 128M），再写入磁盘中，在内存中，是不进行数据的更新或合并操作的，只增加数据，这使得用户的写操作只要进入内存中就可以立即返回，保证了 HBase I/O 的高性能。

描述 Hbase 的 rowKey 的设计原则

问题导读：联系 **region** 和 **rowkey** 关系说明,设计可参考以下三个原则。

1 rowkey 长度原则

rowkey 是一个二进制码流，可以是任意字符串，最大长度 64kb，实际应用中一般为 10-100bytes，以 byte[] 形式保存，一般设计成定长。建议越短越好，不要超过 16 个字节，原因如下：

数据的持久化文件 HFile 中是按照 KeyValue 存储的，如果 rowkey 过长会极大影响 HFile 的存储效率

MemStore 将缓存部分数据到内存，如果 rowkey 字段过长，内存的有效利用率就会降低，系统不能缓存更多的数据，这样会降低检索效率

2 rowkey 散列原则

如果 rowkey 按照时间戳的方式递增，不要将时间放在二进制码的前面，**建议将 rowkey 的高位作为散列字段，由程序随机生成，低位放时间字段，这样将提高数据均衡分布在每个 RegionServer，以实现负载均衡的几率。**如果没有散列字段，首字段直接是时间信息，所有的数据都会集中在一个 RegionServer 上，这样在数据检索的时候负载会集中在个别的 RegionServer 上，造成热点问题，会降低查询效率。

3 rowkey 唯一原则

必须在设计上保证其唯一性，rowkey 是按照字典顺序排序存储的，因此，设计 rowkey 的时候，要充分利用这个排序的特点，**将经常读取的数据存储到一块，将最近可能会被访问的数据放到一块。**

列簇怎么创建比较好？ (<=2)

答：rowKey 最好要创建有规则的 rowKey，即最好是有序的。HBase 中一张表最好只创建一到两个列族比较好，因为 HBase 不能很好的处理多个列族。

描述 Hbase 中 scan 和 get 的功能以及实现的异同.

1. 按指定 RowKey 获取唯一一条记录，get 方法 (org.apache.hadoop.hbase.client.Get) Get 的方法处理分两种：设置了 ClosestRowBefore 和没有设置的 rowlock .主要是用来保证行的事务性，即每个 get 是以一个 row 来标记的. 一个 row 中可以有很多 family 和 column.

2.按指定的条件获取一批记录，scan 方法(org.apache.Hadoop.hbase.client.Scan)实现条件查询功能使用的就是 scan 方式.

1)scan 可以通过 **setCaching** 与 **setBatch** 方法提高速度(以空间换时间); 2)scan 可以通过 setStartRow 与 setEndRow 来限定范围([start, end]start 是闭区间, end 是开区间)。范围越小，性能越高。

3)scan 可以通过 setFilter 方法添加过滤器，这也是分页、多条件查询的基础。 3.全表扫描，即直接扫描整张表中所有行记录

请详细描述 Hbase 中一个 Cell 的结构

HBase 中通过 row 和 columns 确定的为一个存储单元称为 cell。

Cell：由 {row key, column(=<family> + <label>), version} 是唯一确定的单元 cell 中的数据是没有类型的，全部是字节码形式存储

请描述 Hbase 中 scan 对象的 setCache 和 setBatch 方法的使用。

<https://www.cnblogs.com/editice/archive/2013/04/22/3035728.html>

cache 是面向行的优化处理，batch 是面向列的优化处理

简述 HBASE 中 compact 用途是什么，什么时候触发，分为哪两种，有什么区别，有哪些相关配置参数？

在 hbase 中每当有 memstore 数据 flush 到磁盘之后，就形成一个 storefile，当 storefile 的数量达到一定程度后，就需要将 storefile 文件来进行 compaction 操作。

Compact 的作用：

1>.合并文件

2>.清除过期，多余版本的数据

3>.提高读写数据的效率

HBase 中实现了两种 compaction 的方式：minor and major. 这两种 compaction 方式的区别是：

1、Minor 操作只用来做部分文件的合并操作以及包括 minVersion=0 并且设置 ttl 的过期版本清理，不做任何删除数据、多版本数据的清理工作。

2、Major 操作是对 Region 下的 HStore 下的所有 StoreFile 执行合并操作，最终的结果是整理合并出一个文件。

简述 Hbase filter 的实现原理是什么？结合实际项目经验，写出几个使用 filter 的场景

HBase 为筛选数据提供了一组过滤器，通过这个过滤器可以在 HBase 中的数

据的多个维度（行，列，数据版本）上进行对数据的筛选操作，也就是说过滤器最终能够筛选的数据能够细化到具体的一个存储单元格上（由行键，列名，时间戳定位）。RowFilter、PrefixFilter。。。

hbase 的 filter 是通过 scan 设置的，所以是基于 scan 的查询结果进行过滤。

过滤器的类型很多，但是可以分为两大类——**比较过滤器，专用过滤器**
过滤器的作用是在服务端判断数据是否满足条件，然后只将满足条件的数据返回给客户端；
如在进行订单开发的时候，我们使用 rowkeyfilter 过滤出某个用户的所有订单

Hbase 内部是什么机制

在 HBase 中无论是增加新行还是修改已有的行，其内部流程都是相同的。HBase 接到命令后存下变化信息，或者写入失败抛出异常。默认情况下，执行写入时会写到两个地方：预写式日志（write-ahead log，也称 HLog）和 MemStore（见图 2-1）。HBase 的默认方式是把写入动作记录在这两个地方，以保证数据持久化。只有当这两个地方的变化信息都写入并确认后，才认为写动作完成。

MemStore 是内存里的写入缓冲区，HBase 中数据在永久写入硬盘之前在这里累积。当 MemStore 填满后，其中的数据会刷写到硬盘，生成一个 HFile。HFile 是 HBase 使用的底层存储格式。HFile 对应于列族，一个列族可以有多个 HFile，但一个 HFile 不能存储多个列族的数据。在集群的每个节点上，每个列族有一个 MemStore。

大型分布式系统中硬件故障很常见，HBase 也不例外。设想一下，如果 MemStore 还没有刷写，服务器就崩溃了，内存中没有写入硬盘的数据就会丢失。HBase 的应对办法是在写动作完成之前先写入 WAL。HBase 集群中每台服务器维护一个 WAL 来记录发生的变化。

WAL 是底层文件系统上的一个文件。直到 WAL 新记录成功写入后，写动作才被认为成功完成。这可以保证 HBase 和支撑它的文件系统满足持久性。大多数情况下，HBase 使用 Hadoop 分布式文件系统（HDFS）来作为底层文件系统。

如果 HBase 服务器宕机，没有从 MemStore 里刷写到 HFile 的数据将可以通过回放 WAL 来恢复。你不需要手工执行。Hbase 的内部机制中有恢复流程部分来处理。每台 HBase 服务器有一个 WAL，这台服务器上的所有表（和它们的列族）共享这个 WAL。

你可能想到，写入时跳过 WAL 应该会提升写性能。但我们不建议禁用 WAL，除非你愿意在出问题丢失数据。如果你想测试一下，如下代码可以禁用 WAL：注意：不写入 WAL 会在 RegionServer 故障时增加丢失数据的风险。关闭 WAL，出现故障时 HBase 可能无法恢复数据，没有刷写到硬盘的所有写入数据都会丢失。

HBase 宕机如何处理

答：宕机分为 HMaster 宕机和 HRegioner 宕机，如果是 HRegioner 宕机，HMaster 会将其所管理的 region 重新分布到其他活动的 RegionServer 上，由于数据和日志都持久在 HDFS 中，该操作不会导致数据丢失。所以数据的一致性和安全性是有保障的。

如果是 HMaster 宕机，HMaster 没有单点问题，HBase 中可以启动多个 HMaster，通过

Zookeeper 的 Master Election 机制保证总有一个 Master 运行。即 ZooKeeper 会保证总会有一个 HMaster 在对外提供服务。

补充 --- 知识点深度剖析(可选看):

1.导致 Hbase 挂掉的场景

<https://blog.csdn.net/zlfprogram/article/details/74066585> 2.深入理解 HBase 的 memestore、storeFile(HFile)

<https://blog.csdn.net/xiaoshunzi111/article/details/69844526>

3.HBase-7.hbase 查询多版本数据&过滤器原则&批量导入 Hbase&hbase 预分区

<https://blog.csdn.net/shenfuli/article/details/50589496>

补充: 布隆过滤可以每列族单独启用。

使用 `HColumnDescriptor.setBloomFilterType(NONE | ROW | ROWCOL)` 对列族

启用布隆过滤可以节省读磁盘过程, 可以有助于降低读取延迟

4. HBase 原理 - 数据读取流程解析+HBase 最佳实践 - 写性能优化策略

<http://hbasefly.com/2016/12/21/hbase-getorscan/>

<http://hbasefly.com/2016/12/10/hbase-parctice-write/>

5. HBase 原理 - 所有 Region 切分的细节都在这里了

<http://hbasefly.com/2017/08/27/hbase-split/>

6. HBase Compaction 的前生今世

<http://hbasefly.com/2016/07/13/hbase-compaction-1/>

第 6 章 spark

6.1 什么是 Spark

Spark 是基于内存计算大数据分析引擎, 提高了在大数据环境下数据处理的实时性。Spark 仅仅只涉及到数据的计算, 没有涉及到数据的存储。

6.1.1 Spark 的特点及相对于 MapReduce 的优势

MapReduce 存在的问题

1. MapReduce 框架局限性

1) 仅支持 Map 和 Reduce 两种操作

2) 处理效率低效。

a) Map 中间结果写磁盘，Reduce 写 HDFS，多个 MR 之间通过 HDFS 交换数据；任务调度和启动开销大；

b) 无法充分利用内存

c) Map 端和 Reduce 端均需要排序

3) 不适合迭代计算(如机器学习、图计算等)，交互式处理(数据挖掘) 和流式处理(点击日志分析)

2. MapReduce 编程不够灵活

1) 尝试 scala 函数式编程语言

Spark 的特点及优势

1. 高效(比 MapReduce 快 10~100 倍)

1) 内存计算引擎，提供 Cache 机制来支持需要反复迭代计算或者多次数据共享，减少数据读取的 IO 开销

2) DAG 引擎，减少多次计算之间中间结果写到 HDFS 的开销

3) 使用多线程池模型来减少 task 启动开销，shuffle 过程中避免 不必要的 sort 操作以及减少磁盘 IO 操作

2. 易用

1) 提供了丰富的 API，支持 Java，Scala，Python 和 R 四种语言

2) 代码量比 MapReduce 少 2~5 倍

3. 兼容性

可与 Hadoop 集成 读写 HDFS/Hbase/Cassandra 与 YARN 集成

4. 通用性

Spark 可以用于批处理、交互式查询（Spark SQL）、实时流处理（Spark Streaming）、机器学习（Spark MLlib）和图计算（GraphX）

6.1.2 Spark HA 高可用部署

*Spark HA 解决 Master 单点故障的两种方案:

- 1.基于文件系统的单点恢复(主要用于开发或测试环境)
- 2.基于 zookeeper 的 Standby Masters(用于生产模式)

*基于 zookeeper 的 Spark HA 高可用集群部署

(1)vim spark-env.sh

注释掉 export SPARK_MASTER_HOST=hdp-node-01

(2)在 spark-env.sh 添加 SPARK_DAEMON_JAVA_OPTS, 内容如下:

```
export          SPARK_DAEMON_JAVA_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER
-Dspark.deploy.zookeeper.url=hdp-node-01:2181,hdp-node-02:2181,hdp-node-03:2181
-Dspark.deploy.zookeeper.dir=/spark"
```

参数说明

`spark.deploy.recoveryMode:`

恢复模式（Master 重新启动的模式）有三种：(1)ZooKeeper (2) FileSystem

(3)NONE

`spark.deploy.zookeeper.url:` ZooKeeper 的 Server 地址

`spark.deploy.zookeeper.dir:` 保存集群元数据信息的文件、目录。

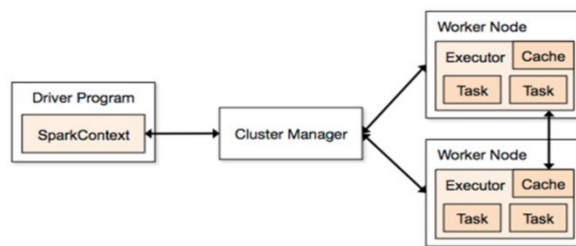
包括 Worker, Driver 和 Application。

注意:

在普通模式下启动 spark 集群, 只需要在主机上面执行 start-all.sh 就可以了。

在高可用模式下启动 spark 集群，先需要在任意一台节点上启动 start-all.sh 命令。然后在另外一台节点上单独启动 master。命令 start-master.sh。

6.1.3 Spark 的角色



Driver Program：运行main 函数并且新建 SparkContext 的程序(初始化工作)。

Application：基于 Spark 的应用程序，包含了 driver 程序和集群上的 executor(代码逻辑与运行资源)。

Cluster Manager：指的是在集群上获取资源的外部服务(提供外部运行资源，资源分配)。目前有三种类型

- (1) Standalone: spark 原生的资源管理，由 Master 负责资源的分配
- (2) Apache Mesos:与 hadoop MR 兼容性良好的一种资源调度框架
- (3) Hadoop Yarn: 主要是指 Yarn 中的 ResourceManager

Master：集群中老大，负责资源的分配和任务的调度

Worker Node：集群中任何可以运行 Application 代码的节点(小弟)，在 Standalone 模式中指的是通过 slaves 文件配置的 Worker 节点，在 Spark on Yarn 模式下就是 NodeManager 节点

Executor：是在一个 worker node 上为某应用启动的一个进程，该进程负责

运行行任务，并且负责将数据存在内存或者磁盘上。每个应用都有各自独立的 executor(运行当前任务所需要的资源)。

Task：被送到某个 executor 上的工作单元(线程)。

Cache：Spark 中设置的数据缓存

6.1.4 Spark 程序提交任务模式

普通模式提交任务(Master 节点已知):

```
bin/spark-submit \ (提交任务脚本)
--class org.apache.spark.examples.SparkPi \ (需要运行的主类)
--master spark://hdp-node-01:7077 \ (指定任务提交到哪个 Master)
--executor-memory 1G \ (指定 executor 内存大小)
--total-executor-cores 2 \ (总 executor 运行核数)
examples/jars/spark-examples_2.11-2.0.2.jar \ (程序 jar 包所在地址)
10 (main 方法所需参数)
```

高可用模式提交任务:

```
bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master spark://hdp-node-01:7077,hdp-node-02:7077,hdp-node-03:7077 \
(指定 Master 列表)
--executor-memory 1G \
--total-executor-cores 2 \
examples/jars/spark-examples_2.11-2.0.2.jar \
```

6.1.5 Spark-Shell

读取本地文件

1. 运行 spark-shell --master local[N](N 表线程数)
2. 编写 scala 代码

```
sc.textFile("file:///root///words.txt")  
  
.flatMap(_._split(" ")).map((_,1)).reduceByKey(_+_).collect
```

读取 HDFS 上数据

1. 整合 spark 和 HDFS, 修改配置文件 spark-env.sh

```
export HADOOP_CONF_DIR=/opt/bigdata/hadoop-2.6.4/etc/hadoop
```
2. 启动 hdfs, 然后重启 spark 集群
3. 向 hdfs 上传个文件
4. 在 spark shell 中用 scala 语言编写 spark 程序

指定具体的 master 地址

1. 执行启动命令:

```
spark-shell \  
  
--master spark://hdp-node-01:7077 \  
  
--executor-memory 1g \  
  
--total-executor-cores 2
```


若没指定 master 地址则默认本地模式

2. 编写 scala 代码

6.1.6 在 IDEA 中编写 WordCount 程序

- 1.创建 Maven 项目并配置 pom.xml
- 2.添加 src/main/scala 和 src/test/scala, 与 pom.xml 中的配置保持一致
- 3.新建一个 scala class, 类型为 Object,并编写 spark 程序
- 4.使用 Maven 打包上传到 Spark 集群中的某个节点上
- 5.启动 hdfs 和 Spark 集群
- 6.使用 spark-submit 命令提交 Spark 应用

6.2 Spark Streaming

6.2.1 Spark Streミング 的特性

易用、容错、易整合

6.2.2 Spark Streaming 对比 Storm

	SparkStreaming	Storm
开发语言	Scala	Clojure
编程模型	DStream	Spout/Bolt
实时性	准实时, 批处理	实时流处理

6.3 算子操作

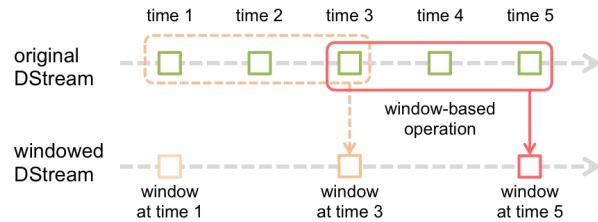
6.3.1 Transformations

常用的转换算子	
Transformation	Meaning

map (func)	对 DStream 中的各个元素进行 func 函数操作，然后返回一个新的 DStream
flatMap (func)	与 map 方法类似，只不过各个输入项可以被输出为零个或多个输出项
filter (func)	过滤出所有函数 func 返回值为 true 的 DStream 元素并返回一个新的 DStream
repartition (numPartitions)	增加或减少 DStream 中的分区数，从而改变 DStream 的并行度
union (otherStream)	将源 DStream 和输入参数为 otherDStream 的元素合并，并返回一个新的 DStream.
count ()	通过对 DStream 中的各个 RDD 中的元素进行计数，然后返回只有一个元素的 RDD 构成的 DStream
reduce (func)	对源 DStream 中的各个 RDD 中的元素利用 func 进行聚合操作，然后返回只有一个元素的 RDD 构成的新的 DStream.
countByValue ()	对于元素类型为 K 的 DStream, 返回一个元素为 (K, Long) 键值对形式的新的 DStream, Long 对应的值为源 DStream 中各个 RDD 的 key 出现的次数
reduceByKey (func, [numTasks])	利用 func 函数对源 DStream 中的 key 进行聚合操作，然后返回新的 (K, V) 对构成的 DStream
join (otherStream, [numTasks])	输入为 (K,V)、(K,W) 类型的 DStream，返回一个新的 (K, (V, W)) 类型的 DStream
cogroup (otherStream, [numTasks])	输入为 (K,V)、(K,W) 类型的 DStream，返回一个新的 (K, Seq[V], Seq[W]) 元组类型的 DStream
transform (func)	通过 RDD-to-RDD 函数作用于 DStream 中的各个 RDD，可以是任意的 RDD 操作，从而返回一个新的 RDD

特殊的转换算子（以下两个都需要设置 checkpoint）	
Transformation	Meaning
UpdateStateByKey （累加统计）	<p>根据 key 的之前状态值和 key 的新值，对 key 进行更新，返回一个新状态的 DStream</p> <pre>val ds4: DStream[(String, Int)] = ds3.updateStateByKey((newValues, runningCount) => { Option(newValues.sum + runningCount.getOrElse(0)) }))</pre> <p>需要注意这个算子的参数是一个函数，函数的第一个参数 newValues 为 ds3 的值，runningCount 为上一次的计算结果的值。</p>

Window Operations(开窗函数)



(1)红色的矩形就是一个窗口，窗口框住的是一段时间内的数据流。

(2)这里面每一个 time 都是时间单元，在官方的例子中，每隔 window size 是 3 time unit, 而且每隔 2 个单位时间，窗口会 slide 一次。

所以需要两个参数：

- a.窗口大小，一段时间内数据的容器。
- b.滑动间隔，每隔多久计算一次。

```
//构建 StreamingContext 对象，每个批处理的时间间隔
val scc = new StreamingContext(sc, Seconds(5))

...

val ds4: DStream[(String, Int)] =
ds3.reduceByKeyAndWindow((a: Int, b: Int) => {
  a + b
}, Seconds(5), Seconds(5))
```

注意
窗口大小和滑动间隔 一定要是设定的数据批处理间隔的整数倍，否则会报错

6.3.2 OutputOperations

Output Operations 可以将 DStream 的数据输出到外部的数据库或文件系统，当某个 Output Operations 被调用时（与 RDD 的 Action 相同），spark streaming 程序才会开始真正的计算过程

Output Operation	Meaning
print()	打印到控制台
saveAsTextFiles(prefix, [suffix])	保存流的内容为文本文件，文件名为 "prefix-TIME_IN_MS[.suffix]".
saveAsObjectFiles(prefix, [suffix])	保存流的内容为 SequenceFile，文件名为 "prefix-TIME_IN_MS[.suffix]".
saveAsHadoopFiles(prefix, [suffix])	保存流的内容为 hadoop 文件，文件名为 "prefix-TIME_IN_MS[.suffix]".
foreachRDD(func)	对 Dstream 里面的每个 RDD 执行 func

6.3.3 Spark Streaming 编程实战

开发流程：

- * 1、构建 `sparkContext` 对象
- * 2、构建 `StreamingContext` 对象
- * 3、创建输入流 `InputDStream`
- * 4、对 `DStream` 执行算子操作
- * 5、将执行结果保存或者输出

6.3.4 SparkStreaming 整合 flume

Spark Streaming 对接 FlumeNG 有两种方式：

(1)、FlumeNG 主动将消息 Push 推给 Spark Streaming

Spark 程序需要启动一个端口接受数据，所以 flume 的配置文件中需要配置 spark 程序所运行的 ip 和端口

(2)、Spark Streaming 主动从 flume 中 Poll 拉取数据。

Flume 需要启动一个端口来输出数据，所以 flume 配置文件中配置的是 flume 机器的主机名和端口，而在 spark 程序中需要绑定 flume 的 ip 和输出端口，这样 spark 程序才能主动拉取到数据

两种方式的启动顺序对比：

在 push 模式中,先启动 spark application,进入等待状态,等待 flume push 数据,此时启动 flume 进行数据的传递.

在 pull 模式中,spark application 会从配置的端口 pull 数据,此时若 flume 还未启动,spark application 会提示端口连接失败.所以需要先启动 flume 后启动 spark application

6.3.5 SparkStreaming 整合 Kafka

SparkStreaming 整合 Kafka 有两种方式：

(1)、Receiver 方式 (KafkaUtils.createStream (调用 kafka 高级 api))

这种方法使用 Receiver 来接收数据。Receiver 是使用 Kafka 高级消费者 API 实现的。与所有的接收者一样，通过 Receiver 从 Kafka 接收的数据存储在 Spark 执行程序 executor 中，然后由 Spark Streaming 启动的作业处理数据。但是，在默认配置下，这种方法可能会在失败时丢失数据。为了确保零数据丢失，您必须在 Spark Streaming（在 Spark 1.2 中引入）中额外启用 wal 预写日志（需要设置检查点 checkpoint<包含消息和 offset>保证了消息不丢失），同时保存所有接收到的 Kafka 数据写入 hdfs 的预写日志，以便所有数据都可以在失败时恢复。

每次消费完一条数据都需要在 zk 上更新 offset 的值，如果在更新 offset 过程中 zk 挂掉了，消费的消息偏移量并没有更新成功。下次程序在消费的时候就会出现重复消费的情况。

(2)、Direct 方式 (KafkaUtils.createDirectStream (调用 kafka 低级 api))

定期地从 kafka 的 topic 下对应的 partition 中查询最新的偏移量，再根据偏移量范围在每个 batch 里面处理数据，Spark 通过调用 kafka 简单的消费者 Api (低级 api)

读取一定范围的数据。

Direct 对比 Receiver 方式的优点：

1、简化并行

一个 rdd 分区对比 topic 上一个分区

2、高效

第一种实现数据的零丢失是将数据预先保存在 WAL 中，会复制一遍数据，会导致数据被拷贝两次，第一次是接受 kafka 中 topic 的数据，另一次是写到 WAL 中。而没有 receiver 的这种方式消除了这个问题

3、恰好一次

offset 仅被保存在 checkpoint 目录里，消除了 zk 与 ssc 偏移量不一致问题，**解决重复消费问题**，缺点是无法使用基于 zk 的 kafka 监控工具

6.3.6 Checkpoint

Spark Streaming 的检查点具有容错机制，有足够的信息能够支持故障恢复。

支持两种数据类型的检查点：**元数据检查点**和**数据检查点**。

(1) **元数据检查点**，在类似 HDFS 的容错存储上，保存 Streaming 计算信息。这种检查点用来**恢复运行 Streaming 应用程序失败的 Driver 进程**。

(2) **数据检查点**，在进行跨越多个批次合并数据的有状态操作时尤其重要。通过周期检查将转换 RDD 的中间状态进行可靠存储，借以切断无限增加的依赖。使用有状态的转换，如果 updateStateByKey 或者 reduceByKeyAndWindow 在应用程序中使用，那么需要提供检查点路径，对 RDD 进行周期性检查。

当程序因为**异常重启**时，如果**检查点路径存在**，则 **context** 将从检查点数据中**重建**。

如果检查点目录不存在，将会新建 context，并设置 DStream。

6.4 Spark SQL

6.4.1 几个知识点

1) Spark sql 的属性

① **易整合**: 可以通过 sql 开发对应的应用程序, 也可以使用 java/scala/python/R 编写的 API 来开发

② **统一的数据源访问**: 可以使用相同的方式来连接到不同的数据源

// 即: `sparkSession.read.文件格式(文件路径)`

③ **兼容 hive**: 可以使用 spark sql 来操作 hive sql

④ **标准的数据连接**: spark sql 可以使用标准的数据库连接(JDBC, ODBC)来操作关系型数据库

2) DataFrame 与 RDD 的优缺点

RDD 的优缺点:

优点:

(1) 编译时类型安全

编译时就能检查出类型错误

(2) 面向对象的编程风格

直接通过对象调用方法的形式来操作数据

缺点:

(1) 序列化和反序列化的性能开销

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。

(2) GC(垃圾回收)的性能开销

频繁的创建和销毁对象, 势必会增加 GC

DataFrame 通过引入 **schema** (即数据的结构信息)和 **off-heap** (不在堆里面的内存, 指的是除了不在堆的内存, 使用操作系统上的内存), 解决了 **RDD** 的缺点, Spark 通过 **schame** 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; 通过 **off-heap** 引入, 可以快速的操作数据, 避免大量的 **GC**。但是却丢了 **RDD** 的优点, **DataFrame** 不是**类型安全**的, API 也不是**面向对象风格**的

3) DataFrame、DataSet、RDD 的区别

1, 张三, 23	ID:String	Name:String	Age:int	value:String
2, 李四, 35	1	张三	23	1, 张三, 23
	2	李四	35	2, 李四, 35
RDD	DataFrame	DataSet		

DataSet 包含了 DataFrame 的功能, Spark2.0 中两者统一, DataFrame 表示为 DataSet[Row], 即 DataSet 的子集

① DataSet 可以在编译时检查类型 (即类型安全)

② 并且是面向对象的编程接口

// 也就是说, DataSet 弥补了 DataFrame 的缺点, 并且继承了它的优点

4) 读取数据源创建 DataFrame

5 ① 读取文本文件创建 DataFrame

```
sparkSession.read.text("路径 + txt 文件名")
```

6 ② 读取 json 文件创建 DataFrame

```
sparkSession.read.json("路径 + json 文件名")
```

7 ③ 读取 parquet 列式存储格式文件创建 DataFrame

```
sparkSession.read.parquet("路径 + parquet 文件名")
```

6.4.2 002 以编程方式执行 Spark SQL 查询

1) 编写 Spark SQL 程序实现 RDD 转换成 DataFrame

8 ① 通过反射推断 Schema

```
case class Person(id:Int,name:String,age:Int)

//3、读取数据文件
val dataRdd: RDD[Array[String]] = sc.textFile("d:\\person.txt").map(x=>x.split(" "))

//4、将rdd与样例类关联
val personRDD: RDD[Person] = dataRdd.map(x=> Person(x(0).toInt,x(1),x(2).toInt))
```



```
//5、将rdd转换成dataFrame
//手动导入隐式转换
import spark.implicits._
val personDF: DataFrame = personRDD.toDF
```

// 这种方式首先是定义样例类 Person, 然后通过将 rdd 与该样例类关联(通过样例类创建 schema, case class 的参数名称会被利用反射机制作为列名) 生成最终的 RDD, 最后该 RDD 调用 toDF 方法转换为 DataFrame. 有了 DataFrame 之后就可以调用各种方法来执行 spark sql 查询了

9 ② 通过 StructType 直接指定 Schema

```
//4、将rdd与Row类型关联
val rowRDD: RDD[Row] = data.map(x => Row(x(0).toInt, x(1), x(2).toInt))
//5、通过StructType指定schema
val schema: StructType = StructType(
    StructField("id", IntegerType, true) ::
    StructField("name", StringType, false) ::
    StructField("age", IntegerType, false) :: Nil)
//6、sparkSession的createDataFrame 获取dataFrame
val df: DataFrame = spark.createDataFrame(rowRDD, schema)
```

// 这种方式首先是将 rdd 与 row 类型关联得到 rowRDD, 然后定义 structType 类型数据用来指定 schema. 最后通过 sparkSession 调用 createDataFrame 方法(含 rowRDD 以及 schema 两个参数)来得到 DataFrame. 有了 DataFrame 之后就可以调用各种方法来执行 spark sql 查询了

2) 编写 Spark SQL 程序操作 HiveContext

```
//1、创建sparkSession
val spark: SparkSession = SparkSession.builder()
    .appName("HiveSupport")
    .master("local[2]")
    .enableHiveSupport() |
    .getOrCreate()
```

// 设置为 enableHiveSupport 的 sparkSession 可以直接操作 hql (即 sparkSession 调用 sql 方法, 里面可以直接写 hql 语句来操作)

6.4.3 JDBC 数据源

1) SparkSql 从 MySQL 中加载数据

```
//2、读取mysql表中的 数据
//定义url
val url="jdbc:mysql://192.168.200.100:3306/spark"
//定义表名
val tableName="iplocation"
//定义相关属性
val properties=new Properties
//设置用户名和密码
properties.setProperty("user","root")
properties.setProperty("password","123456")
val mysqlDF: DataFrame = spark.read.jdbc(url,tableName,properties)
```

// sparkSession 通过调用 read 和 jdbc 方法(包含 url, 表名, 用户名及密码这几个参数)来得到 DataFrame. 有了 DataFrame 之后就可以调用各种方法来执行 spark sql 查询了

2) SparkSql 将数据写入到 MySQL 中

```
//把dataFrame写入到mysql表中
val properties=new Properties
properties.setProperty("user","root")
properties.setProperty("password","123456")
sortedDF.write.jdbc("jdbc:mysql://192.168.200.100:3306/spark","student",properties)
```

// DataFrame 通过调用 write 和 jdbc 方法(包含 url, 表名, 用户名及密码这几个参数)就可以将数据直接写入到对应的 mysql 中

6.5 SparkRDD

6.5.1 了解 SparkRDD

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, 它代表一个不可变、可分区、里面的元素可并行计算的集合。

RDD 具有数据流模型的特点: 自动容错、位置感知性调度和可伸缩性。RDD 允许用户在执行多个查询时显式地将数据缓存在内存中, 后续的查询能够重用这些数据, 这极大地提升了查询速度。

Dataset: 一个数据集，用于存放数据的。

Distributed: RDD 中的数据是分布式存储的，可用于分布式计算。

Resilient: RDD 中的数据可以存储在内存中或者磁盘中。

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. A list of partitions : 一个分区 (Partition) 列表，数据集的基本组成单位。

对于 RDD 来说，每个分区都会被一个计算任务处理，并决定并行计算的粒度。用户可以在创建 RDD 时指定 RDD 的分区个数，如果没有指定，那么就会采用默认值。（比如：读取 HDFS 上数据文件产生的 RDD 分区数跟 block 的个数相等）

2.A function for computing each split : 一个计算每个分区的函数。

Spark 中 RDD 的计算是以分区为单位的，每个 RDD 都会实现 compute 函数以达到这个目的。

3. A list of dependencies on other RDDs: 一个 RDD 会依赖于其他多个 RDD, RDD 之间的依赖关系。

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

4.Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is

hash-partitioned): 一个 Partitioner, 即 RDD 的分区函数 (可选项)。

当前 Spark 中实现了两种类型的分区函数, 一个是基于哈希的 HashPartitioner, 另外一个则是基于范围的 RangePartitioner。只有对于 key-value 的 RDD, 才会有 Partitioner, 非 key-value 的 RDD 的 Partitioner 的值是 None。Partitioner 函数决定了 parent RDD Shuffle 输出时的分区数量。

5.Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file): 一个列表, 存储每个 Partition 的优先位置(可选项)。

对于一个 HDFS 文件来说, 这个列表保存的就是每个 Partition 所在的块的位置。按照“移动数据不如移动计算”的理念, Spark 在进行任务调度的时候, 会尽可能地将计算任务分配到其所要处理数据块的存储位置 (spark 进行任务分配的时候尽可能选择那些存有数据的 worker 节点来进行任务计算)。

3.为什么会产生 RDD?

1. 传统的 MapReduce 虽然具有自动容错、平衡负载和可拓展性的优点, 但是其最大缺点是采用非循环式的数据流模型, 使得在迭代计算中要进行大量的磁盘 IO 操作。RDD 正是解决这一缺点的抽象方法。

(2) RDD 是 Spark 提供的最重要的抽象的概念, 它是一种具有容错机制的特殊集合, 可以分布在集群的节点上, 以函数式编程来操作集合, 进行各种并行操作。可以把 RDD 的结果数据进行缓存, 方便进行多次重用, 避免重复计算。

4) RDD 在 Spark 中的地位及作用

1. 为什么会有 Spark?

因为传统的并行计算模型无法有效的解决迭代计算 (iterative) 和交互式计算 (interactive); 而 Spark 的使命便是解决这两个问题, 这也是他存在的价值和理由。

2. Spark 如何解决迭代计算?

其主要实现思想就是 RDD, 把所有计算的数据保存在分布式的内存中。迭代计算通常情况下都是对同一个数据集做反复的迭代计算, 数据在内存中将大大提升 IO 操作。这也是 Spark 涉及的核心: 内存计算。

3. Spark 如何实现交互式计算?

因为 Spark 是用 scala 语言实现的, Spark 和 scala 能够紧密的集成, 所以 Spark 可以完美的运用 scala 的解释器, 使得其中的 scala 可以向操作本地集合对象一样轻松操作分布式数据集。

4. Spark 和 RDD 的关系?

RDD 是一种具有容错性、基于内存计算的抽象方法, RDD 是 Spark Core 的底层核心, Spark 则是这个抽象方法的实现。

6.5.2 创建 RDD

1. 由一个已经存在的 Scala 集合创建。

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

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

```
val rdd2 = sc.textFile("/words.txt")
```

3. 已有的 RDD 经过算子转换生成新的 RDD

```
val rdd3=rdd2.flatMap(_.split(" "))
```

6.5.3 RDD 编程 API

1 RDD 的算子分类

Transformation (转换): 根据数据集创建一个新的数据集, 计算后返回一个新 RDD; 例如: 一个 rdd 进行 map 操作后生了一个新的 rdd。

Action (动作): 对 rdd 结果计算后返回一个数值 value 给驱动程序;

例如: collect 算子将数据集的所有元素收集完成返回给驱动程序。

2 Transformation

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

常用的 Transformation:

转换	含义
<code>map(func)</code>	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
<code>filter(func)</code>	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
<code>flatMap(func)</code>	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)
<code>mapPartitions(func)</code>	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 <code>Iterator[T] => Iterator[U]</code>

<code>mapPartitionsWithIndex(func)</code>	类似于 <code>mapPartitions</code> , 但 <code>func</code> 带有一个整数参数表示分片的索引值, 因此在类型为 <code>T</code> 的 RDD 上运行时, <code>func</code> 的函数类型必须是 <code>(Int, Iterator[T]) => Iterator[U]</code>
<code>union(otherDataset)</code>	对源 RDD 和参数 RDD 求并集后返回一个新的 RDD
<code>intersection(otherDataset)</code>	对源 RDD 和参数 RDD 求交集后返回一个新的 RDD
<code>distinct([numTasks])</code>	对源 RDD 进行去重后返回一个新的 RDD
<code>groupByKey([numTasks])</code>	在一个(K,V)的 RDD 上调用, 返回一个(K, Iterator[V])的 RDD
<code>reduceByKey(func, [numTasks])</code>	在一个(K,V)的 RDD 上调用, 返回一个(K,V)的 RDD, 使用指定的 <code>reduce</code> 函数, 将相同 <code>key</code> 的值聚合到一起, 与 <code>groupByKey</code> 类似, <code>reduce</code> 任务的个数可以通过第二个可选的参数来设置
<code>sortByKey([ascending], [numTasks])</code>	在一个(K,V)的 RDD 上调用, <code>K</code> 必须实现 <code>Ordered</code> 接口, 返回一个按照 <code>key</code> 进行排序的(K,V)的 RDD
<code>sortBy(func, [ascending], [numTasks])</code>	与 <code>sortByKey</code> 类似, 但是更灵活
<code>join(otherDataset, [numTasks])</code>	在类型为(K,V)和(K,W)的 RDD 上调用, 返回一个相同 <code>key</code> 对应的所有元素对在一起的(K,(V,W))的 RDD
<code>cogroup(otherDataset, [numTasks])</code>	在类型为 (K,V) 和 (K,W) 的 RDD 上调用, 返回一个 <code>(K,(Iterable<V>,Iterable<W>))</code> 类型的 RDD
<code>coalesce(numPartitions)</code>	减少 RDD 的分区数到指定值。
<code>repartition(numPartitions)</code>	重新给 RDD 分区
<code>repartitionAndSortWithinPartitions(partitioner)</code>	重新给 RDD 分区, 并且每个分区内以记录的 <code>key</code> 排序

3 Action

动作	含义
----	----

<code>reduce(func)</code>	reduce 将 RDD 中元素前两个传给输入函数，产生一个新的 return 值，新产生的 return 值与 RDD 中下一个元素（第三个元素）组成两个元素，再被传给输入函数，直到最后只有一个值为止。
<code>collect()</code>	在驱动程序中，以数组的形式返回数据集的所有元素
<code>count()</code>	返回 RDD 的元素个数
<code>first()</code>	返回 RDD 的第一个元素（类似于 take(1)）
<code>take(n)</code>	返回一个由数据集的前 n 个元素组成的数组
<code>takeOrdered(n, [ordering])</code>	返回自然顺序或者自定义顺序的前 n 个元素
<code>saveAsTextFile(path)</code>	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
<code>saveAsSequenceFile(path)</code>	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。
<code>saveAsObjectFile(path)</code>	将数据集的元素，以 Java 序列化的方式保存到指定的目录下
<code>countByKey()</code>	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。
<code>foreach(func)</code>	在数据集的每一个元素上，运行函数 func
<code>foreachPartition(func)</code>	在数据集的每一个分区上，运行函数 func

6.5.4 RDD 常用的算子操作

Spark Rdd 的所有算子操作，请见《sparkRDD 函数详解.docx》

启动 spark-shell 进行测试：

```
spark-shell --master spark://node1:7077
```

练习 1：map、filter

//通过并行化生成 rdd

```
val rdd1 = sc.parallelize(List(5, 6, 4, 7, 3, 8, 2, 9, 1, 10))
```

//对 rdd1 里的每一个元素乘 2 然后排序

```
val rdd2 = rdd1.map(_ * 2).sortBy(x => x, true)
```

//过滤出大于等于 5 的元素

```
val rdd3 = rdd2.filter(_ >= 5)
//将元素以数组的方式在客户端显示
rdd3.collect
```

练习 2: flatMap

```
val rdd1 = sc.parallelize(Array("a b c", "d e f", "h i j"))
//将 rdd1 里面的每一个元素先切分在压平
val rdd2 = rdd1.flatMap(_.split(" "))
rdd2.collect
```

练习 3: 交集、并集

```
val rdd1 = sc.parallelize(List(5, 6, 4, 3))
val rdd2 = sc.parallelize(List(1, 2, 3, 4))
//求并集
val rdd3 = rdd1.union(rdd2)
//求交集
val rdd4 = rdd1.intersection(rdd2)
//去重
rdd3.distinct.collect
rdd4.collect
```

练习 4: join、groupByKey

```
val rdd1 = sc.parallelize(List(("tom", 1), ("jerry", 3), ("kitty", 2)))
val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 1), ("shuke", 2)))
//求 join
val rdd3 = rdd1.join(rdd2)
rdd3.collect
//求并集
val rdd4 = rdd1 union rdd2
rdd4.collect
//按 key 进行分组
val rdd5=rdd4.groupByKey
rdd5.collect
```


练习 5: cogroup

```
val rdd1 = sc.parallelize(List(("tom", 1), ("tom", 2), ("jerry", 3), ("kitty", 2)))
val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 1), ("jim", 2)))
//cogroup
val rdd3 = rdd1.cogroup(rdd2)
//注意 cogroup 与 groupByKey 的区别
rdd3.collect
```

练习 6: reduce

```
val rdd1 = sc.parallelize(List(1, 2, 3, 4, 5))
//reduce 聚合
val rdd2 = rdd1.reduce(_ + _)
rdd2.collect
```

练习 7: reduceByKey、sortByKey

```
val rdd1 = sc.parallelize(List(("tom", 1), ("jerry", 3), ("kitty", 2), ("shuke", 1)))
val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 3), ("shuke", 2), ("kitty", 5)))
val rdd3 = rdd1.union(rdd2)
//按 key 进行聚合
val rdd4 = rdd3.reduceByKey(_ + _)
rdd4.collect
//按 value 的降序排序
val rdd5 = rdd4.map(t => (t._2, t._1)).sortByKey(false).map(t => (t._2, t._1))
rdd5.collect
```

练习 8: repartition、coalesce

```
val rdd1 = sc.parallelize(1 to 10,3)
//利用 repartition 改变 rdd1 分区数
//减少分区
rdd1.repartition(2).partitions.size
//增加分区
rdd1.repartition(4).partitions.size
//利用 coalesce 改变 rdd1 分区数
//减少分区
```

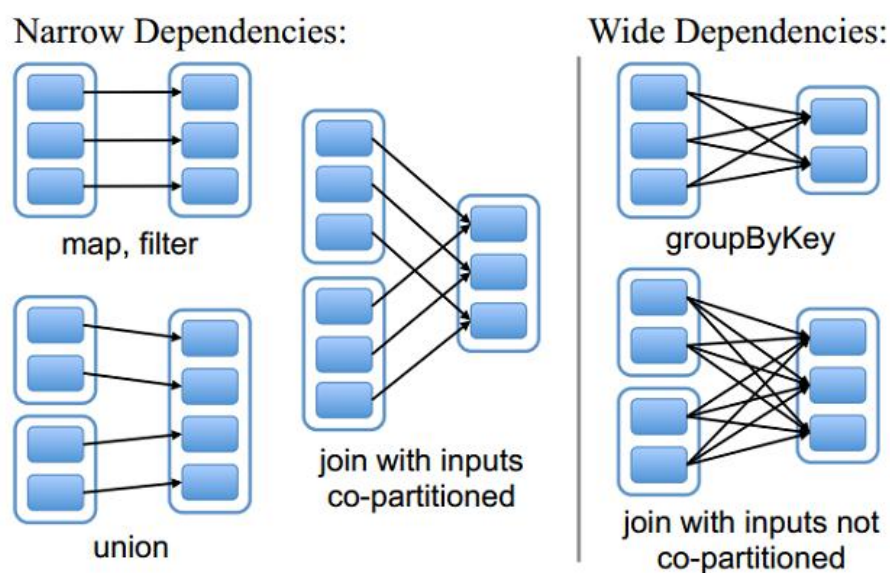
rdd1.coalesce(2).partitions.size

注意： repartition 可以增加和减少 rdd 中的分区数，coalesce 只能减少 rdd 分区数，增加 rdd 分区数不会生效。

6.5.5 RDD 的依赖关系

1 RDD 的依赖

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



2 窄依赖

窄依赖指的是每一个父 RDD 的 Partition 最多被子 RDD 的一个 Partition 使用

总结：窄依赖我们形象的比喻为独生子女

3 宽依赖

宽依赖指的是多个子 RDD 的 Partition 会依赖同一个父 RDD 的 Partition

总结：宽依赖我们形象的比喻为超生

4 Lineage（血统）

RDD 只支持粗粒度转换，即只记录单个块上执行的单个操作。将创建 RDD 的一系列 Lineage（即血统）记录下来，以便恢复丢失的分区。RDD 的 Lineage 会记录 RDD 的元数据信息和转换行为，当该 RDD 的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

6.5.6 RDD 的缓存

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

RDD 缓存方式

RDD 通过 `persist` 方法或 `cache` 方法可以将前面的计算结果缓存，但是并不是这两个方法被调用时立即缓存，而是触发后面的 action 时，该 RDD 将会被缓存在计算节点的内存中，并供后面重用。

```
/** 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。

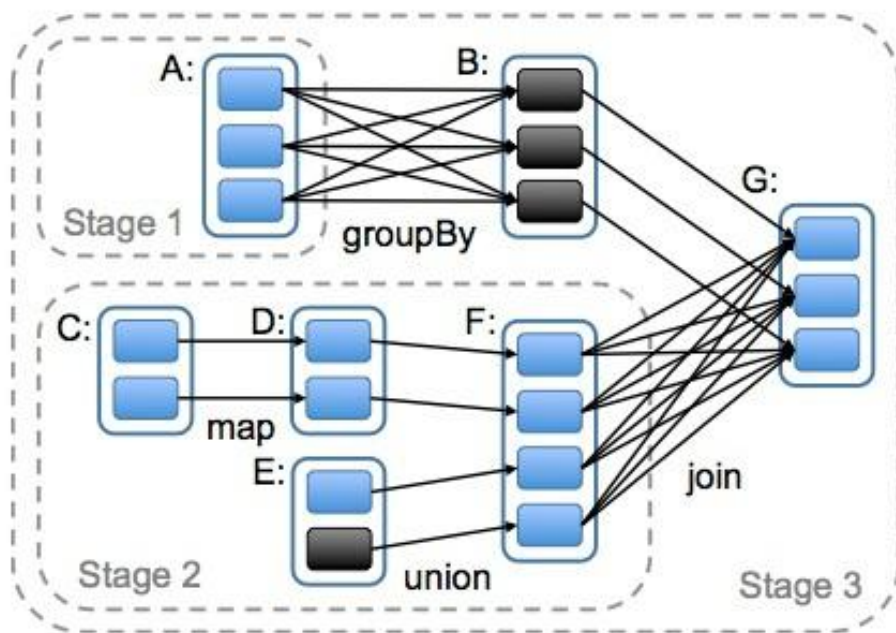
```
object StorageLevel {  
  val NONE = new StorageLevel(false, false, false, false)  
  val DISK_ONLY = new StorageLevel(true, false, false, false)  
  val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)  
  val MEMORY_ONLY = new StorageLevel(false, true, false, true)  
  val MEMORY_ONLY_2 = 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)
```

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

6.5.7 DAG 的生成

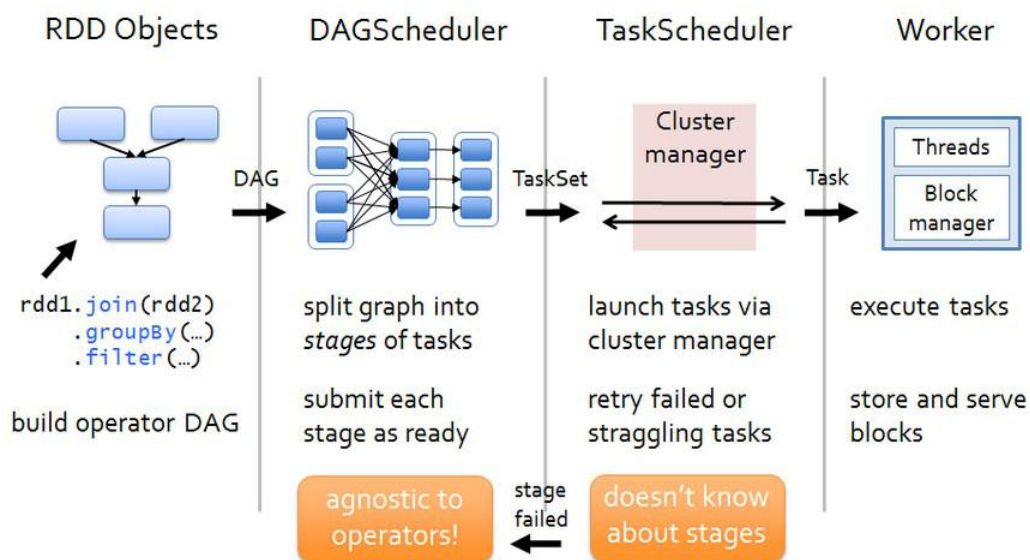
什么是 DAG

DAG(Directed Acyclic Graph)叫做有向无环图，原始的 RDD 通过一系列的转换就形成了 DAG, 根据 RDD 之间依赖关系的不同将 DAG 划分成不同的 Stage(调度阶段)。对于窄依赖，partition 的转换处理在一个 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。



6.5.8 Spark 任务调度

1 任务调度流程图



各个 RDD 之间存在着依赖关系，这些依赖关系就形成有向无环图 DAG，DAGScheduler 对这些依赖关系形成的 DAG 进行 Stage 划分，划分的规则很简单，从后往前回溯，遇到窄依赖加入本 stage，遇见宽依赖进行 Stage 切分。完成了

Stage 的划分。DAGScheduler 基于每个 Stage 生成 TaskSet, 并将 TaskSet 提交给 TaskScheduler。TaskScheduler 负责具体的 task 调度, 最后在 Worker 节点上启动 task。

2 DAGScheduler

1. DAGScheduler 对 DAG 有向无环图进行 Stage 划分。
2. 记录哪个 RDD 或者 Stage 输出被物化 (缓存), 通常在一个复杂的 shuffle 之后, 通常物化一下 (cache、persist), 方便之后的计算。
3. 重新提交 shuffle 输出丢失的 stage (stage 内部计算出错) 给 TaskScheduler
4. 将 Taskset 传给底层调度器
 - a. - spark-cluster TaskScheduler
 - b. - yarn-cluster YarnClusterScheduler
 - c. - yarn-client YarnClientClusterScheduler

3 TaskScheduler

- (1) 为每一个 TaskSet 构建一个 TaskSetManager 实例管理这个 TaskSet 的生命周期
- (2) 数据本地性决定每个 Task 最佳位置
- (3) 提交 taskset (一组 task) 到集群运行并监控
- (4) 推测执行, 碰到计算缓慢任务需要放到别的节点上重试
- (5) 重新提交 Shuffle 输出丢失的 Stage 给 DAGScheduler

6.6 RDD 容错机制之 checkpoint

6.6.1 checkpoint 是什么

(1)、Spark 在生产环境下经常会面临 transformation 的 RDD 非常多 (例如一个 Job 中包含 1 万个 RDD) 或者具体 transformation 的 RDD 本身计算特别复杂或者耗时 (例如计算时长超过 1 个小时), 这个时候就要考虑对计算结果数据持久化保存;

(2)、Spark 是擅长多步骤迭代的, 同时擅长基于 Job 的复用, 这个时候如果能够对曾经计算的过程产生的数据进行复用, 就可以极大的提升效率;

(3)、如果采用 persist 把数据放在内存中, 虽然是快速的, 但是也是最不可靠的; 如果把数据放在磁盘上, 也不是完全可靠的! 例如磁盘会损坏, 系统管理员可能清空磁盘。

(4)、Checkpoint 的产生就是为了相对而言更加可靠的持久化数据, 在 Checkpoint 的时候可以指定把数据放在本地, 并且是多副本的方式, 但是在生产环境下是放在 HDFS 上, 这就天然的借助了 HDFS 高容错、高可靠的特征来完成了最大化的可靠的持久化数据的方式;

假如进行一个 1 万个算子操作, 在 9000 个算子的时候 persist, 数据还是有可能丢失的, 但是如果 checkpoint, 数据丢失的概率几乎为 0。

6.6.2 checkpoint 原理机制

1.当 RDD 使用 cache 机制从内存中读取数据, 如果数据没有读到, 会使用 checkpoint 机制读取数据。此时如果没有 checkpoint 机制, 那么就需要找到

父 RDD 重新计算数据了，因此 checkpoint 是个很重要的容错机制。

checkpoint 就是对于一个 RDD chain（链）如果后面需要反复使用某些中间结果 RDD，可能因为一些故障导致该中间数据丢失，那么就可以针对该 RDD 启动 checkpoint 机制，使用 checkpoint 首先需要调用 `sparkContext` 的 `setCheckpoint` 方法，设置一个容错文件系统目录，比如 hdfs，然后对 RDD 调用 `checkpoint` 方法。之后在 RDD 所处的 job 运行结束后，会启动一个单独的 job 来将 checkpoint 过的数据写入之前设置的文件系统持久化，进行高可用。所以后面的计算在使用该 RDD 时，如果数据丢失了，但是还是可以从它的 checkpoint 中读取数据，不需要重新计算。

2.persist 或者 cache 与 checkpoint 的区别在于,前者持久化只是将数据保存在 BlockManager 中但是其 lineage 是不变的，但是后者 checkpoint 执行完后, rdd 已经没有依赖 RDD, 只有一个 checkpointRDD, checkpoint 之后, RDD 的 lineage 就改变了。persist 或者 cache 持久化的数据丢失的可能性更大, 因为可能磁盘或内存被清理, 但是 checkpoint 的数据通常保存到 hdfs 上, 放在了高容错文件系统。

问题：哪些 RDD 需要 cache?

会被重复使用的（但不能太大）。

问题：用户怎么设定哪些 RDD 要 cache?

因为用户只与 driver program 打交道，因此只能用 `rdd.cache()` 去 cache 用户能看到的 RDD。所谓能看到指的是调用 `transformation()` 后生成的 RDD，而某些在 `transformation()` 中 Spark 自己生成的 RDD 是不能被用户直接 cache 的，比如 `reduceByKey()` 中会生成的 `ShuffledRDD`、`MapPartitionsRDD` 是不能被用户直接 cache 的。

问题：哪些 RDD 需要 cache？

会被重复使用的（但不能太大）。

问题：用户怎么设定哪些 RDD 要 cache？

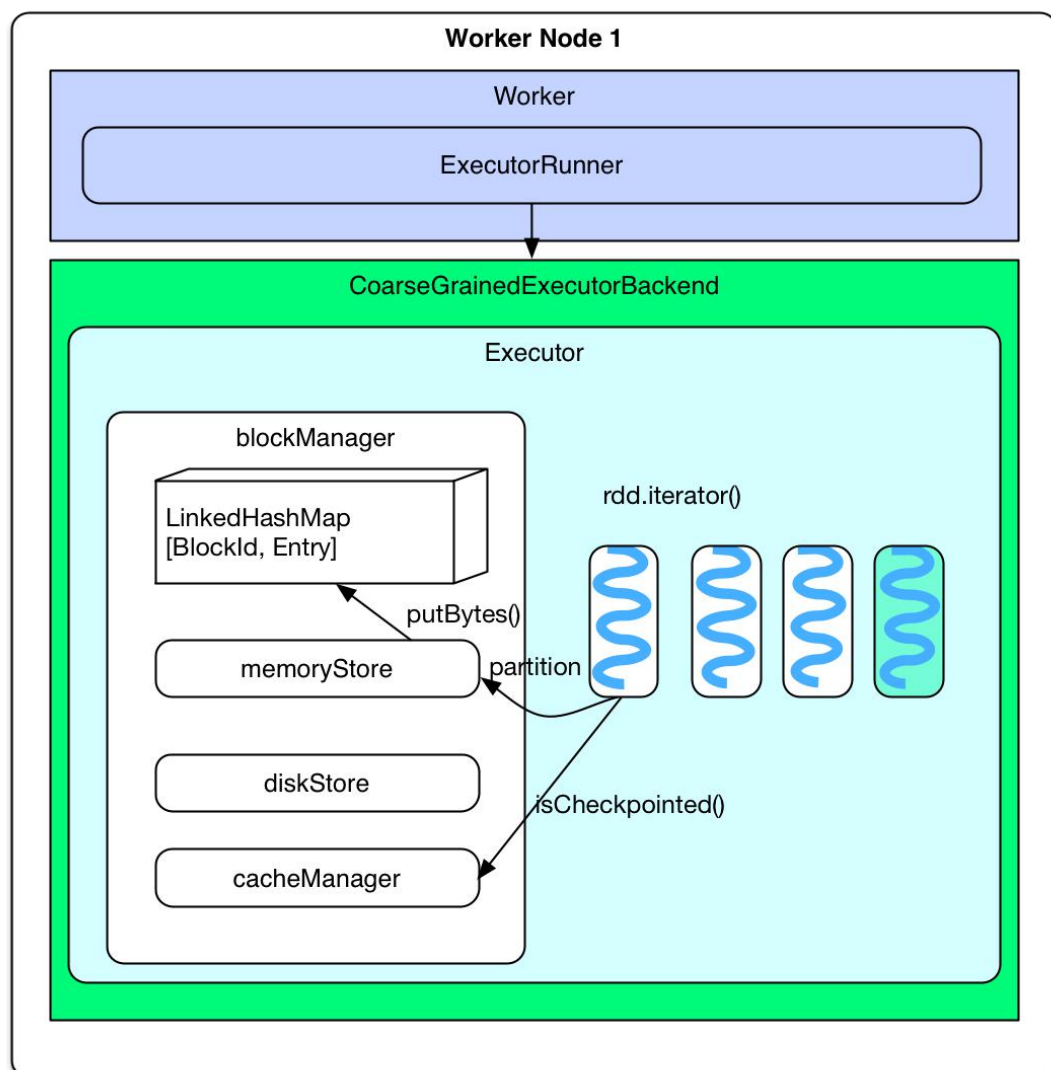
因为用户只与 driver program 打交道，因此只能用 `rdd.cache()` 去 cache 用户能看到的 RDD。所谓能看到指的是调用 `transformation()` 后生成的 RDD，而某些在 `transformation()` 中 Spark 自己生成的 RDD 是不能被用户直接 cache 的，比如 `reduceByKey()` 中会生成的 `ShuffledRDD`、`MapPartitionsRDD` 是不能被用户直接 cache 的。

问题：driver program 设定 `rdd.cache()` 后，系统怎么对 RDD 进行 cache？

先不看实现，自己来想象一下如何完成 cache：当 task 计算得到 RDD 的某个 partition 的第一个 record 后，就去判断该 RDD 是否要被 cache，如果要被 cache 的话，将这个 record 及后续计算的到的 records 直接丢给本地 `blockManager` 的 `memoryStore`，如果 `memoryStore` 存不下就交给 `diskStore` 存放到磁盘。

实际实现与设想的基本类似，区别在于：将要计算 RDD partition 的时候（而不是已经计算得到第一个 record 的时候）就去判断 partition 要不要被 cache。如果要被 cache 的话，先将 partition 计算出来，然后 cache 到内存。cache 只使用 memory，写磁盘的话那就叫 checkpoint 了。

调用 `rdd.cache()` 后，`rdd` 就变成 `persistRDD` 了，其 `StorageLevel` 为 `MEMORY_ONLY`。`persistRDD` 会告知 driver 说自己是需要被 persist 的。



如果用代码表示：

```
rdd.iterator()
=> SparkEnv.get.cacheManager.getOrCompute(thisRDD, split, context, storageLevel)
=> key = RDDBlockId(rdd.id, split.index)
=> blockManager.get(key)
=> computedValues = rdd.computeOrReadCheckpoint(split, context)
    if (isCheckpointed) firstParent[T].iterator(split, context)
    else compute(split, context)
=> elements = new ArrayBuffer[Any]
=> elements ++= computedValues
=> updatedBlocks = blockManager.put(key, elements, tellMaster = true)
```

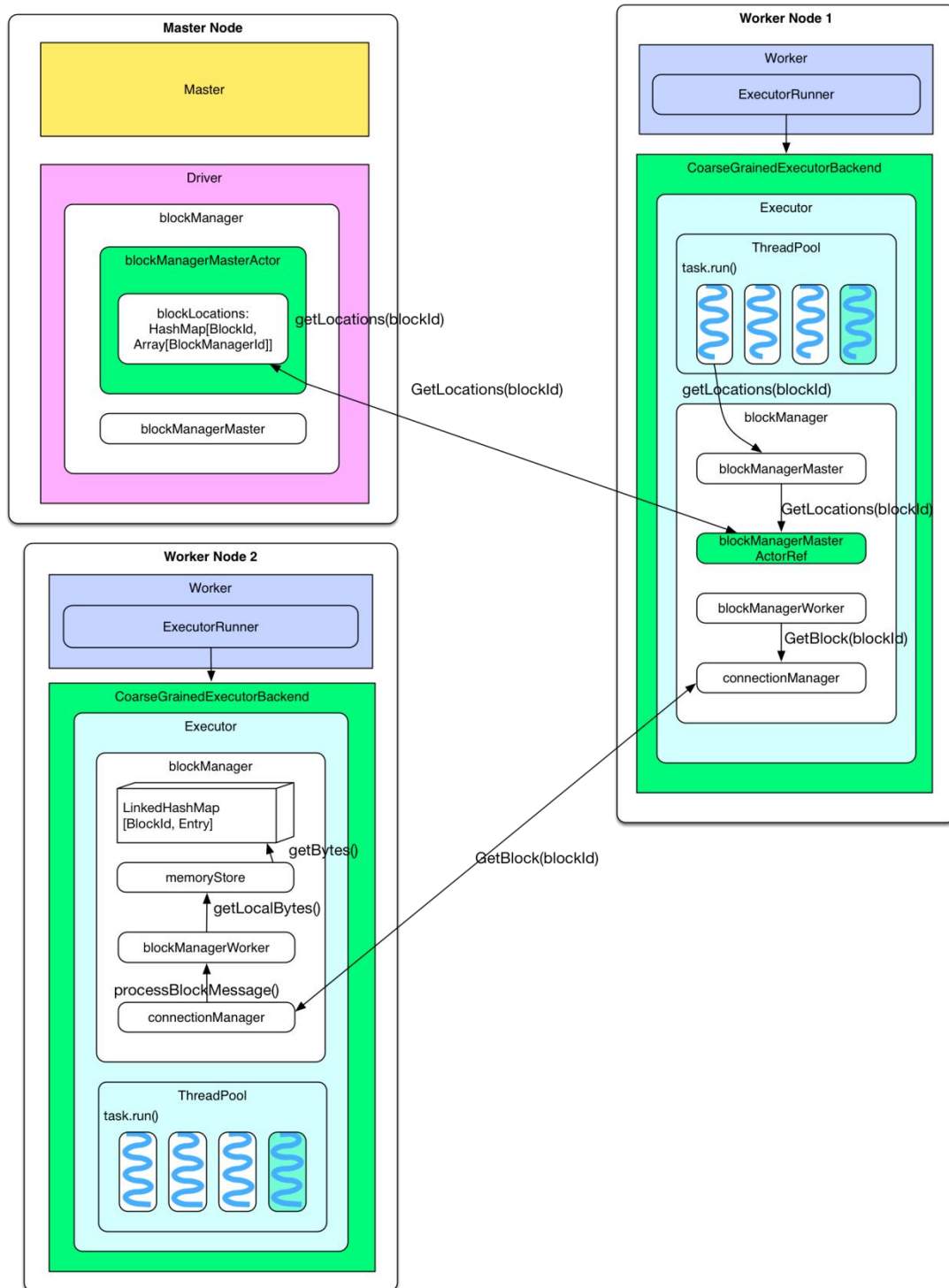
当 `rdd.iterator()` 被调用的时候，也就是要计算该 `rdd` 中某个 `partition` 的时候，会先去 `cacheManager` 那里领取一个 `blockId`，表明是要存哪个 `RDD` 的哪个 `partition`，这个 `blockId` 类型是 `RDDBlockId`（`memoryStore` 里面可能还存放有 `task` 的 `result` 等数据，因此 `blockId` 的类型是用来区分不同的数据）。然后去 `blockManager` 里面查看该 `partition` 是不是已经被 `checkpoint` 了，如果是，表明以前运行过该 `task`，那就不用计算该 `partition` 了，直接从 `checkpoint` 中读取该 `partition` 的所有 `records` 放到叫做 `elements` 的 `ArrayBuffer` 里面。如果没有被 `checkpoint` 过，先将 `partition` 计算出来，然后将其所有 `records` 放到 `elements` 里面。最后将 `elements` 交给 `blockManager` 进行 `cache`。

`blockManager` 将 `elements`（也就是 `partition`）存放到 `memoryStore` 管理的 `LinkedHashMap[BlockId, Entry]` 里面。如果 `partition` 大于 `memoryStore` 的存储极限（默认是 60% 的 `heap`），那么直接返回说存不下。如果剩余空间也许能放下，会先 `drop` 掉一些早先被 `cached` 的 `RDD` 的 `partition`，为新来的 `partition` 腾地方，如果腾出的地方够，就把新来的 `partition` 放到

LinkedHashMap 里面，腾不出就返回说存不下。注意 drop 的时候不会去 drop 与新来的 partition 同属于一个 RDD 的 partition。drop 的时候先 drop 最早被 cache 的 partition。（说好的 LRU 替换算法呢？）

问题：cached RDD 怎么被读取？

下次计算（一般是同一 application 的下一个 job 计算）时如果用到 cached RDD，task 会直接去 blockManager 的 memoryStore 中读取。具体地讲，当要计算某个 rdd 中的 partition 时候（通过调用 rdd.iterator()）会先去 blockManager 里面查找是否已经被 cache 了，如果 partition 被 cache 在本地，就直接使用 blockManager.getLocal() 去本地 memoryStore 里读取。如果该 partition 被其他节点上 blockManager cache 了，会通过 blockManager.getRemote() 去其他节点上读取，读取过程如下图。



获取 **cached partitions** 的存储位置: partition 被 cache 后所在节点上的 **blockManager** 会通知 driver 上的 **blockMangerMasterActor** 说某 rdd 的 partition 已经被我 cache 了, 这个信息会存储在 **blockMangerMasterActor** 的 **blockLocations: HashMap** 中。等到 task 执行需要 **cached rdd** 的时候, 会调用

blockManagerMaster 的 `getLocations(blockId)` 去询问某 partition 的存储位置, 这个询问信息会发到 driver 那里, driver 查询 `blockLocations` 获得位置信息并将信息送回。

读取其他节点上的 cached partition: task 得到 cached partition 的位置信息后, 将 `GetBlock(blockId)` 的请求通过 `connectionManager` 发送到目标节点。目标节点收到请求后从本地 `blockManager` 那里的 `memoryStore` 读取 cached partition, 最后发送回来。

6.6.3 Checkpoint 常见面试问题

问题：哪些 RDD 需要 checkpoint?

运算时间很长或运算量太大才能得到的 RDD, `computing chain` 过长或依赖其他 RDD 很多的 RDD。实际上, 将 `ShuffleMapTask` 的输出结果存放到本地磁盘也算是 checkpoint, 只不过这个 checkpoint 的主要目的是去 partition 输出数据。

问题：什么时候 checkpoint?

cache 机制是每计算出一个要 cache 的 partition 就直接将其 cache 到内存了。但 checkpoint 没有使用这种第一次计算得到就存储的方法, 而是等到 job 结束后另外启动专门的 job 去完成 checkpoint。也就是说需要 checkpoint 的 RDD 会被计算两次。因此, 在使用 `rdd.checkpoint()` 的时候, 建议加上 `rdd.cache()`, 这样第二次运行的 job 就不用再去计算该 rdd 了, 直接读取 cache 写磁盘。其实 Spark 提供了 `rdd.persist(StorageLevel.DISK_ONLY)` 这样的方法,

相当于 cache 到磁盘上，这样可以做到 rdd 第一次被计算得到时就存储到磁盘上，但这个 persist 和 checkpoint 有很多不同，之后会讨论。

问题：checkpoint 怎么实现？

RDD 需要经过 [Initialized --> marked for checkpointing --> checkpointing in progress --> checkpointed] 这几个阶段才能被 checkpoint。

Initialized: 首先 driver program 需要使用 rdd.checkpoint() 去设定哪些 rdd 需要 checkpoint，设定后，该 rdd 就接受 RDDCheckpointData 管理。用户还要设定 checkpoint 的存储路径，一般在 HDFS 上。

marked for checkpointing: 初始化后，RDDCheckpointData 会将 rdd 标记为 MarkedForCheckpoint。

checkpointing in progress: 每个 job 运行结束后会调用 finalRdd.doCheckpoint(), finalRdd 会顺着 computing chain 回溯扫描，碰到要 checkpoint 的 RDD 就将其标记为 CheckpointingInProgress, 然后将写磁盘 (比如写 HDFS) 需要的配置文件 (如 core-site.xml 等) broadcast 到其他 worker 节点上的 blockManager。完成以后，启动一个 job 来完成 checkpoint (使用 `rdd.context.runJob(rdd, CheckpointRDD.writeToFile(path.toString, broadcastedConf)))`)。

checkpointed: job 完成 checkpoint 后，将该 rdd 的 dependency 全部清掉，并设定该 rdd 状态为 checkpointed。然后，为该 rdd 强加一个依赖，设置该 rdd 的 parent rdd 为 CheckpointRDD，该 CheckpointRDD 负责以后读取在文件系统上的 checkpoint 文件，生成该 rdd 的 partition。

有意思的是我在 driver program 里 checkpoint 了两个 rdd, 结果只有一个 (下面的 result) 被 checkpoint 成功, pairs2 没有被 checkpoint, 也不知道是 bug 还是故意只 checkpoint 下游的 RDD:

```
val data1 = Array[(Int, Char)]((1, 'a'), (2, 'b'), (3, 'c'),  
    (4, 'd'), (5, 'e'), (3, 'f'), (2, 'g'), (1, 'h'))  
val pairs1 = sc.parallelize(data1, 3)  
  
val data2 = Array[(Int, Char)]((1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'))  
val pairs2 = sc.parallelize(data2, 2)  
  
pairs2.checkpoint  
  
val result = pairs1.join(pairs2)  
result.checkpoint
```

问题: 怎么读取 checkpoint 过的 RDD?

在 runJob() 的时候会先调用 finalRDD 的 partitions() 来确定最后会有多个 task。rdd.partitions() 会去检查 (通过 RDDCheckpointData 去检查, 因为它负责管理被 checkpoint 过的 rdd) 该 rdd 是否会 checkpoint 过了, 如果该 rdd 已经被 checkpoint 过了, 直接返回该 rdd 的 partitions 也就是 Array[Partition]。

当调用 rdd.iterator() 去计算该 rdd 的 partition 的时候, 会调用 computeOrReadCheckpoint(split: Partition) 去查看该 rdd 是否被 checkpoint 过了, 如果是, 就调用该 rdd 的 parent rdd 的 iterator() 也就是 CheckpointRDD.iterator(), CheckpointRDD 负责读取文件系统上的文件, 生成

该 rdd 的 partition。这就解释了为什么那么 tricky 地为 checkpointed rdd 添加一个 parent CheckpointRDD。

问题：cache 与 checkpoint 的区别？

关于这个问题，Tathagata Das 有一段回答：There is a significant difference between cache and checkpoint. Cache materializes the RDD and keeps it in memory and/or disk（其实只有 memory）。But the lineage（也就是 computing chain）of RDD（that is, seq of operations that generated the RDD）will be remembered, so that if there are node failures and parts of the cached RDDs are lost, they can be regenerated. However, **checkpoint saves the RDD to an HDFS file and actually forgets the lineage completely**. This allows long lineages to be truncated and the data to be saved reliably in HDFS (which is naturally fault tolerant by replication).

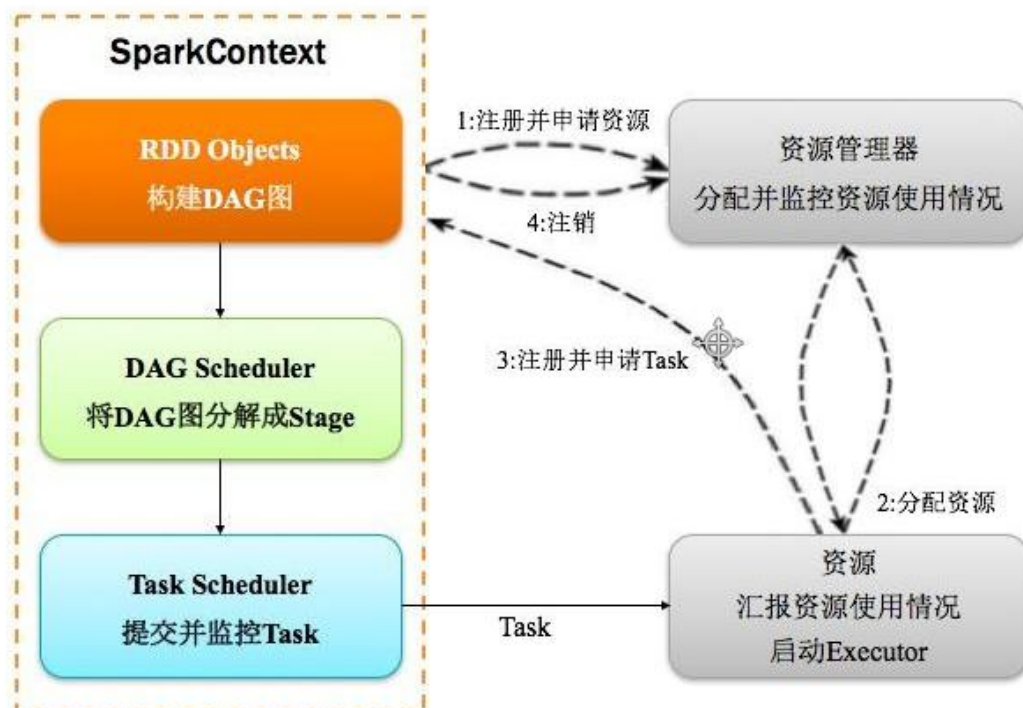
深入一点讨论，`rdd.persist(StorageLevel.DISK_ONLY)` 与 checkpoint 也有区别。

前者虽然可以将 RDD 的 partition 持久化到磁盘，但该 partition 由 blockManager 管理。一旦 driver program 执行结束，也就是 executor 所在进程 CoarseGrainedExecutorBackend stop，blockManager 也会 stop，被 cache 到磁盘上的 RDD 也会被清空（整个 blockManager 使用的 local 文件夹被删除）。而 checkpoint 将 RDD 持久化到 HDFS 或本地文件夹，如果不被手动 remove 掉（话说怎么 **remove checkpoint 过的 RDD?**），是一直存在的，也就是说可以被下一个 driver program 使用，而 cached RDD 不能被其他 driver program 使用。

6.7 Spark 运行架构

6.7.1 Spark 运行基本流程

Spark 运行基本流程参见下面示意图：



- 1) 构建 Spark Application 的运行环境 (启动 SparkContext) , SparkContext 向资源管理器 (可以是 Standalone、Mesos 或 YARN) 注册并申请运行 Executor 资源;
- 2) 资源管理器分配 Executor 资源并启动 Executor, Executor 运行情况将随着心跳发送到资源管理器上;
- 3) SparkContext 构建 DAG 图, 将 DAG 图分解成 Stage, 并把 Taskset 发送给 Task Scheduler。Executor 向 SparkContext 申请 Task, Task Scheduler 将 Task 发放给 Executor 运行同时 SparkContext 将应用程序代码发放给 Executor。
- 4) Task 在 Executor 上运行, 运行完毕释放所有资源。

6. 7. 2 Spark 运行架构特点

Spark 运行架构特点:

- ◆ 每个 Application 获取专属的 executor 进程, 该进程在 Application 期间一直驻留, 并以多线程方式运行 tasks。
- ◆ Spark 任务与资源管理器无关, 只要能够获取 executor 进程, 并能保持相互通信就可以了。
- ◆ 提交 SparkContext 的 Client 应该靠近 Worker 节点 (运行 Executor 的节点), 最好是在同一个 Rack 里, 因为 Spark 程序运行过程中 SparkContext 和 Executor 之间有大量的信息交换; 如果想在远程集群中运行, 最好使用 RPC 将 SparkContext 提交给集群, 不要远离 Worker 运行 SparkContext。
- ◆ Task 采用了数据本地性和推测执行的优化机制。

应用内调度

在指定的 Spark 应用内部（对应同一 SparkContext 实例），多个线程可能并发地提交 Spark 作业（job）。在本节中，作业（job）是指，由 Spark action 算子（如 `:collect`）触发的一系列计算任务的集合。Spark 调度器是完全线程安全的，并且能够支持 Spark 应用同时处理多个请求（比如：来自不同用户的查询）。

默认，Spark 应用内部使用 FIFO 调度策略。每个作业被划分为多个阶段（stage）（例如：`map` 阶段和 `reduce` 阶段），第一个作业在其启动后会优先获取所有的可用资源，然后是第二个作业再申请，再第三个……。如果前面的作业没有把集群资源占满，则后续的作业可以立即启动运行，否则，后提交的作业会有明显的延迟等待。

不过从 Spark 0.8 开始，Spark 也能支持各个作业间的公平（Fair）调度。公平调度时，Spark 以轮询的方式给每个作业分配资源，因此所有的作业获得的资源大体上是平均分配。这意味着，即使有大作业在运行，小的作业再提交也能立即获得计算资源而不是等待前面的作业结束，大大减少了延迟时间。这种模式特别适合于多用户配置。要启用公平调度器，只需设置一下 SparkContext 中

`spark.scheduler.mode` 属性为 FAIR 即可：

```
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.set("spark.scheduler.mode", "FAIR")
val sc = new SparkContext(conf)
```

公平调度资源池

公平调度器还可以支持将作业分组放入资源池（pool），然后给每个资源池配置不同的选项（如：权重）。这样你就可以给一些比较重要的作业创建一个“高优先级”资源池，或者你也可以把每个用户的作业分到一组，这样一来就是各个用户平均分享集群资源，而不是各个作业平分集群资源。Spark 公平调度的实现方式基本都是模仿 [Hadoop Fair Scheduler](#) 来实现的。

默认情况下，新提交的作业都会进入到默认资源池中，不过作业对应于哪个资源池，可以在提交作业的线程中用

`SparkContext.setLocalProperty` 设定 `spark.scheduler.pool` 属性。示例代码如下：

```
// Assuming sc is your SparkContext variable  
sc.setLocalProperty("spark.scheduler.pool", "pool1")
```

一旦设好了局部属性，所有该线程所提交的作业（即：在该线程中调用 action 算子，如：`RDD.save/count/collect` 等）都会使用这个资源池。这个设置是以线程为单位保存的，你很容易实现用同一线程来提交同一用户的所有作业到同一个资源池中。同样，如果需要清除资源池设置，只需在对应线程中调用如下代码：

```
sc.setLocalProperty("spark.scheduler.pool", null)
```

资源池默认行为

默认地，各个资源池之间平分整个集群的资源（包括 `default` 资源池），但在资源池内部，默认情况下，作业是 FIFO 顺序执行的。举例来说，如果你为每个用户创建了一个资源池，那么久意味着各个用户之

间共享整个集群的资源，但每个用户自己提交的作业是按顺序执行的，而不会出现后提交的作业抢占前面作业的资源。

配置资源池属性

资源池的属性需要通过配置文件来指定。每个资源池都支持以下 3 个属性：

- `schedulingMode`: 可以是 FIFO 或 FAIR，控制资源池内部的作业是如何调度的。
- `weight`: 控制资源池相对其他资源池，可以分配到资源的比例。默认所有资源池的 `weight` 都是 1。如果你将某个资源池的 `weight` 设为 2，那么该资源池中的资源将是其他池子的 2 倍。如果将 `weight` 设得很高，如 1000，可以实现资源池之间的调度优先级 – 也就是说，`weight=1000` 的资源池总能立即启动其对应的作业。
- `minShare`: 除了整体 `weight` 之外，每个资源池还能指定一个最小资源分配值（CPU 个数），管理员可能会需要这个设置。公平调度器总是会尝试优先满足所有活跃（active）资源池的最小资源分配值，然后再根据各个池子的 `weight` 来分配剩下的资源。因此，`minShare` 属性能够确保每个资源池都能至少获得一定量的集群资源。`minShare` 的默认值是 0。

资源池属性是一个 XML 文件，可以基于
conf/fairscheduler.xml.template 修改，然后在 [SparkConf](#) 的
spark.scheduler.allocation.file 属性指定文件路径：

```
conf.set("spark.scheduler.allocation.file", "/path/to/file")
```

面试题

1.请列出 spark 的调度器，简述调度原理

分别是 DAGScheduler、TaskScheduler。

DAGScheduler

1. DAGScheduler 对 DAG 有向无环图进行 Stage 划分。
2. 记录哪个 RDD 或者 Stage 输出被物化 (缓存)，通常在一个复杂的 shuffle 之后，通常物化一下(cache、persist)，方便之后的计算。
3. 重新提交 shuffle 输出丢失的 stage (stage 内部计算出错) 给

TaskScheduler

4. 将 Taskset 传给底层调度器
 - a. - spark-cluster TaskScheduler
 - b. - yarn-cluster YarnClusterScheduler
 - c. - yarn-client YarnClientClusterScheduler

TaskScheduler

为每一个 TaskSet 构建一个 TaskSetManager 实例管理这个 TaskSet 的生命周期

- (2) 数据本地性决定每个 Task 最佳位置
- (3) 提交 taskset (一组 task) 到集群运行并监控
- (4) 推测执行, 碰到计算缓慢任务需要放到别的节点上重试
- (5) 重新提交 Shuffle 输出丢失的 Stage 给 DAGScheduler

2. rdd 的特点

RDD 具有数据流模型的特点: 自动容错、位置感知性调度和可伸缩性。RDD 允许用户在执行多个查询时显式地将数据缓存在内存中, 后续的查询能够重用这些数据, 这极大地提升了查询速度。

Spark 优化指南完全版

-----spark 优化指南 -----

[Apache Spark 内存管理详解 \(转载\)](#)

[Spark 性能优化指南——基础篇 \(转载\)](#)

[Spark 性能优化指南——高级篇 \(转载\)](#)

[Spark 官方调优文档翻译 \(转载\)](#)

整套调优方案主要分为开发调优、资源调优、数据倾斜调优、shuffle 调优几个部分

6.8 开发调优

6.8.1 调优概述

Spark 性能优化的第一步, 就是要在开发 Spark 作业的过程中注意和应用一些性能优化的基本原则。开发调优, 就是要让大家了解以下一些 Spark 基本开发原则, 包括: RDD lineage 设计、算子的合理使用、特殊操作的优化等。在开发过程中, 时时刻刻都应该注意以上原则, 并将这些原则根据具体的业务以及实际的应用场景, 灵活地运用到自己的 Spark 作业中。

6.8.2 原则一: 避免创建重复的 RDD

通常来说, 我们在开发一个 Spark 作业时, 首先是基于某个数据源 (比如 Hive 表或 HDFS 文件) 创建一个初始的 RDD; 接着对这个 RDD 执行某个算子操作, 然后得到下一个 RDD; 以此类推, 循环往复, 直到计算出最终我们需要的结果。在这个过程中, 多个 RDD 会通过

不同的算子操作（比如 `map`、`reduce` 等）串起来，这个“RDD 串”，就是 RDD lineage，也就是“RDD 的血缘关系链”。

我们在开发过程中要注意：对于同一份数据，只应该创建一个 RDD，不能创建多个 RDD 来代表同一份数据。

一些 Spark 初学者在刚开始开发 Spark 作业时，或者是有经验的工程师在开发 RDD lineage 极其冗长的 Spark 作业时，可能会忘了自己之前对于某一份数据已经创建过一个 RDD 了，从而导致对于同一份数据，创建了多个 RDD。这就意味着，我们的 Spark 作业会进行多次重复计算来创建多个代表相同数据的 RDD，进而增加了作业的性能开销。

一个简单的例子



```
1 // 需要对名为“hello.txt”的HDFS文件进行一次map操作，再进行一次
  reduce操作。也就是说，需要对一份数据执行两次算子操作。 2 3 // 错误的
  做法：对于同一份数据执行多次算子操作时，创建多个RDD。 4 // 这里执
  行了两次textFile方法，针对同一个HDFS文件，创建了两个RDD出来，然后分
  别对每个RDD都执行了一个算子操作。 5 // 这种情况下，Spark需要从HDFS
  上两次加载hello.txt文件的内容，并创建两个单独的RDD；第二次加载HDFS
  文件以及创建RDD的性能开销，很明显是白白浪费掉的。 6 val rdd1 =
  sc.textFile("hdfs://192.168.0.1:9000/hello.txt") 7 rdd1.map(...) 8
  val rdd2 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt") 9
  rdd2.reduce(...) 10 11 // 正确的用法：对于一份数据执行多次算子操作时，
  只使用一个RDD。 12 // 这种写法很明显比上一种写法要好多了，因为我们
  对于同一份数据只创建了一个RDD，然后对这一个RDD执行了多次算子操作。13 //
  但是要注意到这里为止优化还没有结束，由于rdd1被执行了两次算子操作，第
  二次执行reduce操作的时候，还会再次从源头处重新计算一次rdd1的数据，因
  此还是会有重复计算的性能开销。 14 // 要彻底解决这个问题，必须结合“原
  则三：对多次使用的RDD进行持久化”，才能保证一个RDD被多次使用时只被
  计算一次。 15 val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")
  16 rdd1.map(...) 17 rdd1.reduce(...)
```



6.8.3 原则二：尽可能复用同一个 RDD

除了要避免在开发过程中对一份完全相同的数据创建多个 RDD 之外，在对不同的数据执行算子操作时还要尽可能地复用同一个 RDD。比如说，有一个 RDD 的数据格式是 `key-value` 类型的，另一个是单 `value` 类型的，这两个 RDD 的 `value` 数据是完全一样的。那么此时我们可以只使用 `key-value` 类型的那个 RDD，因为其中已经包含了另一个的数据。对于类似这种多个 RDD 的数据有重叠或者包含的情况，我们应该尽量复用同一个 RDD，这样可以尽可能地减少 RDD 的数量，从而尽可能减少算子执行的次数。

一个简单的例子



```
1 // 错误的做法。 2 3 // 有一个<Long, String>格式的 RDD, 即 rdd1。 4
// 接着由于业务需要, 对 rdd1 执行了一个 map 操作, 创建了一个 rdd2, 而 rdd2
// 中的数据仅仅是 rdd1 中的 value 值而已, 也就是说, rdd2 是 rdd1 的子集。 5
JavaPairRDD<Long, String> rdd1 = ... 6 JavaRDD<String> rdd2 =
rdd1.map(...) 7 8 // 分别对 rdd1 和 rdd2 执行了不同的算子操作。 9
rdd1.reduceByKey(...) 10 rdd2.map(...) 11 12 // 正确的做法。 13 14 //
// 上面这个 case 中, 其实 rdd1 和 rdd2 的区别无非就是数据格式不同而已, rdd2
// 的数据完全就是 rdd1 的子集而已, 却创建了两个 rdd, 并对两个 rdd 都执行了
// 一次算子操作。 15 // 此时会因为对 rdd1 执行 map 算子来创建 rdd2, 而多执
// 行一次算子操作, 进而增加性能开销。 16 17 // 其实在这种情况下完全可以
// 复用同一个 RDD。 18 // 我们可以使用 rdd1, 既做 reduceByKey 操作, 也做 map
// 操作。 19 // 在进行第二个 map 操作时, 只使用每个数据的 tuple._2, 也就
// 是 rdd1 中的 value 值, 即可。 20 JavaPairRDD<Long, String> rdd1 = ... 21
rdd1.reduceByKey(...) 22 rdd1.map(tuple._2...) 23 24 // 第二种方式相
// 较于第一种方式而言, 很明显减少了一次 rdd2 的计算开销。 25 // 但是到
// 这里为止, 优化还没有结束, 对 rdd1 我们还是执行了两次算子操作, rdd1 实
// 际上还是会被计算两次。 26 // 因此还需要配合“原则三: 对多次使用的 RDD
// 进行持久化”进行使用, 才能保证一个 RDD 被多次使用时只被计算一次。
```



6.8.4 原则三: 对多次使用的 RDD 进行持久化

当你在 Spark 代码中多次对一个 RDD 做了算子操作后, 恭喜, 你已经实现 Spark 作业第一步的优化了, 也就是尽可能复用 RDD。此时就该在这个基础之上, 进行第二步优化了, 也就是要保证对一个 RDD 执行多次算子操作时, 这个 RDD 本身仅仅被计算一次。

Spark 中对于一个 RDD 执行多次算子的默认原理是这样的: 每次你对一个 RDD 执行一个算子操作时, 都会重新从源头处计算一遍, 计算出那个 RDD 来, 然后再对这个 RDD 执行你的算子操作。这种方式的性能是很差的。

因此对于这种情况, 我们的建议是: 对多次使用的 RDD 进行持久化。此时 Spark 就会根据你的持久化策略, 将 RDD 中的数据保存到内存或者磁盘中。以后每次对这个 RDD 进行算子操作时, 都会直接从内存或磁盘中提取持久化的 RDD 数据, 然后执行算子, 而不会从源头处重新计算一遍这个 RDD, 再执行算子操作。

对多次使用的 **RDD** 进行持久化的代码示例



```

1 // 如果要对一个 RDD 进行持久化,只要对这个 RDD 调用 cache() 和 persist()
  即可。 2 3 // 正确的做法。 4 // cache() 方法表示: 使用非序列化的方式
  将 RDD 中的数据全部尝试持久化到内存中。 5 // 此时再对 rdd1 执行两次算
  子操作时,只有在第一次执行 map 算子时,才会将这个 rdd1 从源头处计算一次。
6 // 第二次执行 reduce 算子时,就会直接从内存中提取数据进行计算,不会重
  复计算一个 rdd。 7 val rdd1 =
sc.textFile("hdfs://192.168.0.1:9000/hello.txt").cache() 8
rdd1.map(...) 9 rdd1.reduce(...) 10 11 // persist() 方法表示: 手动选
  择持久化级别,并使用指定的方式进行持久化。 12 // 比如说,
StorageLevel.MEMORY_AND_DISK_SER 表示,内存充足时优先持久化到内存中,
  内存不充足时持久化到磁盘文件中。 13 // 而且其中的_SER 后缀表示,使用序
  列化的方式来保存 RDD 数据,此时 RDD 中的每个 partition 都会序列化成一个大的
  字节数组,然后再持久化到内存或磁盘中。 14 // 序列化的方式可以减少持久
  化的数据对内存/磁盘的占用量,进而避免内存被持久化数据占用过多,从而
  发生频繁 GC。 15 val rdd1 =
sc.textFile("hdfs://192.168.0.1:9000/hello.txt").persist(StorageLevel.
  MEMORY_AND_DISK_SER) 16 rdd1.map(...) 17 rdd1.reduce(...)

```



对于 `persist()` 方法而言,我们可以根据不同的业务场景选择不同的持久化级别。

Spark 的持久化级别

持久化级别	含义解释
MEMORY_ONLY	使用未序列化的 Java 对象格式,将数据保存在内存中。如果内存不够存放所有的数据,则数据可能就不会进行持久化。那么下次对这个 RDD 执行算子操作时,那些没有被持久化的数据,需要从源头处重新计算一遍。这是默认的持久化策略,使用 <code>cache()</code> 方法时,实际就是使用的这种持久化策略。
MEMORY_AND_DISK	使用未序列化的 Java 对象格式,优先尝试将数据保存在内存中。如果内存不够存放所有的数据,会将数据写入磁盘文件中,下次对这个 RDD 执行算子时,持久化在磁盘文件中的数据会被读取出来使用。
MEMORY_ONLY_SER	基本含义同 <code>MEMORY_ONLY</code> 。唯一的区别是,会将 RDD 中的数据进行序列化,RDD 的每个 <code>partition</code> 会被序列化成一个字节数组。这种方式更加节省内存,从而可以避免持久化的数据占用过多内存导致频繁 GC。
MEMORY_AND_DISK_SER	基本含义同 <code>MEMORY_AND_DISK</code> 。唯一的区别是,会将 RDD 中的数据进行序列化,RDD 的每个 <code>partition</code> 会被序列化成一个字节数组。这种方式更加节省内存,从而可以避免持久化的数据占用过多内存导致频繁 GC。

持久化级别	含义解释
ER	
DISK_ONLY	使用未序列化的 Java 对象格式，将数据全部写入磁盘文件中。
MEMORY_ONLY_2, MEMORY_AND_DISK_2, 等等.	对于上述任意一种持久化策略，如果加上后缀_2，代表的是将每个持久化的数据，都复制一份副本，并将副本保存到其他节点上。这种基于副本的持久化机制主要用于进行容错。假如某个节点挂掉，节点的内存或磁盘中的持久化数据丢失了，那么后续对 RDD 计算时还可以使用该数据在其他节点上的副本。如果没有副本的话，就只能将这些数据从源头处重新计算一遍了。

如何选择一种最合适的持久化策略

- 默认情况下，性能最高的当然是 **MEMORY_ONLY**，但前提是你的内存必须足够大，可以绰绰有余地存放下整个 **RDD** 的所有数据。因为不进行序列化与反序列化操作，就避免了这部分的性能开销；对这个 **RDD** 的后续算子操作，都是基于纯内存中的数据的操作，不需要从磁盘文件中读取数据，性能也很高；而且不需要复制一份数据副本，并远程传送到其他节点上。但是这里必须要注意的是，在实际的生产环境中，恐怕能够直接用这种策略的场景还是有限的，如果 **RDD** 中数据比较多时（比如几十亿），直接用这种持久化级别，会导致 **JVM** 的 **OOM** 内存溢出异常。
- 如果使用 **MEMORY_ONLY** 级别时发生了内存溢出，那么建议尝试使用 **MEMORY_ONLY_SER** 级别。该级别会将 **RDD** 数据序列化后再保存在内存中，此时每个 **partition** 仅仅是一个字节数组而已，大大减少了对对象数量，并降低了内存占用。这种级别比 **MEMORY_ONLY** 多出来的性能开销，主要就是序列化与反序列化的开销。但是后续算子可以基于纯内存进行操作，因此性能总体还是比较高的。此外，可能发生的问题同上，如果 **RDD** 中的数据量过多的话，还是可能会导致 **OOM** 内存溢出的异常。
- 如果纯内存的级别都无法使用，那么建议使用 **MEMORY_AND_DISK_SER** 策略，而不是 **MEMORY_AND_DISK** 策略。因为既然到了这一步，就说明 **RDD** 的数据量很大，内存无法完全放下。序列化后的数据比较少，可以节省内存和磁盘的空间开销。同时该策略会优先尽量尝试将数据缓存在内存中，内存缓存不下才会写入磁盘。
- 通常不建议使用 **DISK_ONLY** 和后缀为_2 的级别：因为完全基于磁盘文件进行数据的读写，会导致性能急剧降低，有时还不如重新计算一次所有 **RDD**。后缀为_2 的级别，必须将所有数据都复制一份副本，并发送到其他节点上，数据复制以及网络传输会导致较大的性能开销，除非是要求作业的高可用性，否则不建议使用。

6.8.5 原则四：尽量避免使用 shuffle 类算子

如果有可能的话，要尽量避免使用 shuffle 类算子。因为 Spark 作业运行过程中，最消耗性能的地方就是 shuffle 过程。shuffle 过程，简单来说，就是将分布在集群中多个节点上的同一个 key，拉取到同一个节点上，进行聚合或 join 等操作。比如 reduceByKey、join 等算子，都会触发 shuffle 操作。

shuffle 过程中，各个节点上的相同 key 都会先写入本地磁盘文件中，然后其他节点需要通过网络传输拉取各个节点上的磁盘文件中的相同 key。而且相同 key 都拉取到同一个节点进行聚合操作时，还有可能会因为一个节点上处理的 key 过多，导致内存不够存放，进而溢写到磁盘文件中。因此在 shuffle 过程中，可能会发生大量的磁盘文件读写的 IO 操作，以及数据的网络传输操作。磁盘 IO 和网络数据传输也是 shuffle 性能较差的主要原因。

因此在我们的开发过程中，能避免则尽可能避免使用 reduceByKey、join、distinct、repartition 等会进行 shuffle 的算子，尽量使用 map 类的非 shuffle 算子。这样的话，没有 shuffle 操作或者仅有较少 shuffle 操作的 Spark 作业，可以大大减少性能开销。

Broadcast 与 map 进行 join 代码示例



```
1 // 传统的 join 操作会导致 shuffle 操作。 2 // 因为两个 RDD 中，相同的
key 都需要通过网络拉取到一个节点上，由一个 task 进行 join 操作。 3 val
rdd3 = rdd1.join(rdd2) 4 5 // Broadcast+map 的 join 操作，不会导致
shuffle 操作。 6 // 使用 Broadcast 将一个数据量较小的 RDD 作为广播变量。
7 val rdd2Data = rdd2.collect() 8 val rdd2DataBroadcast =
sc.broadcast(rdd2Data) 9 10 // 在 rdd1.map 算子中，可以从
rdd2DataBroadcast 中，获取 rdd2 的所有数据。 11 // 然后进行遍历，如果发现
rdd2 中某条数据的 key 与 rdd1 的当前数据的 key 是相同的，那么就判定可以
进行 join。 12 // 此时就可以根据自己需要的方式，将 rdd1 当前数据与 rdd2
中可以连接的数据，拼接在一起（String 或 Tuple）。 13 val rdd3 =
rdd1.map(rdd2DataBroadcast...) 14 15 // 注意，以上操作，建议仅仅在 rdd2
的数据量比较少（比如几百 M，或者一两 G）的情况下使用。 16 // 因为每个
Executor 的内存中，都会驻留一份 rdd2 的全量数据。
```

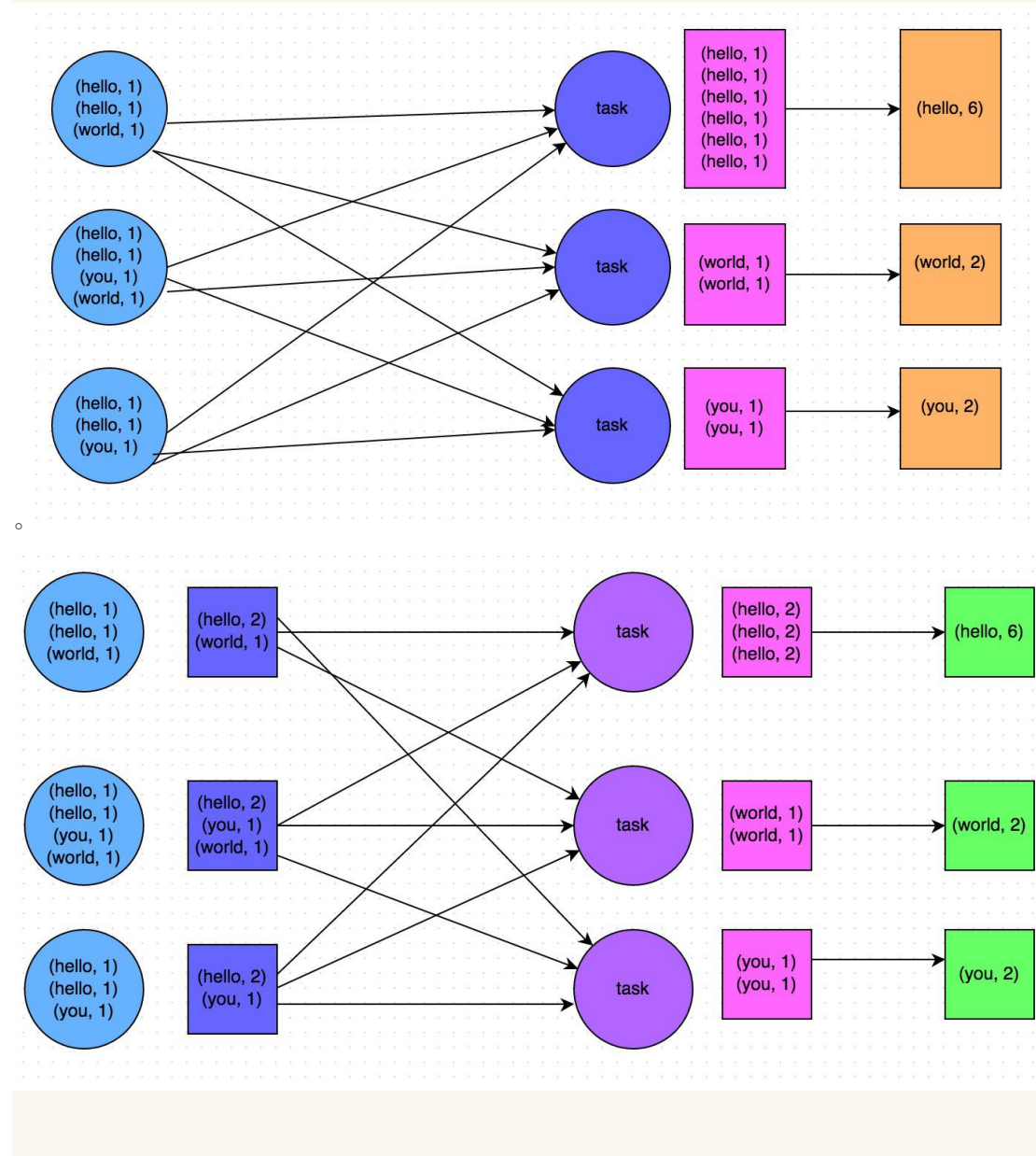


6.8.6 原则五：使用 map-side 预聚合的 shuffle 操作

如果因为业务需要，一定要使用 shuffle 操作，无法用 map 类的算子来替代，那么尽量使用可以 map-side 预聚合的算子。

所谓的 **map-side 预聚合**，说的是在每个节点本地对相同的 **key** 进行一次聚合操作，类似于 **MapReduce** 中的本地 **combiner**。**map-side** 预聚合之后，每个节点本地就只会有一条相同的 **key**，因为多条相同的 **key** 都被聚合起来了。其他节点在拉取所有节点上的相同 **key** 时，就会大大减少需要拉取的数据数量，从而也就减少了磁盘 **IO** 以及网络传输开销。通常来说，在可能的情况下，建议使用 **reduceByKey** 或者 **aggregateByKey** 算子来替代掉 **groupByKey** 算子。因为 **reduceByKey** 和 **aggregateByKey** 算子都会使用用户自定义的函数对每个节点本地的相同 **key** 进行预聚合。而 **groupByKey** 算子是不会进行预聚合的，全量的数据会在集群的各个节点之间分发和传输，性能相对来说比较差。

比如如下两幅图，就是典型的例子，分别基于 **reduceByKey** 和 **groupByKey** 进行单词计数。其中第一张图是 **groupByKey** 的原理图，可以看到，没有进行任何本地聚合时，所有数据都会在集群节点之间传输；第二张图是 **reduceByKey** 的原理图，可以看到，每个节点本地的相同 **key** 数据，都进行了预聚合，然后才传输到其他节点上进行全局聚合。



6.8.7 原则六：使用高性能的算子

除了 shuffle 相关的算子有优化原则之外，其他的算子也都有着相应的优化原则。

使用 reduceByKey/aggregateByKey 替代 groupByKey

详情见“原则五：使用 map-side 预聚合的 shuffle 操作”。

使用 mapPartitions 替代普通 map

mapPartitions 类的算子，一次函数调用会处理一个 partition 所有的数据，而不是一次函数调用处理一条，性能相对来说会高一些。但是有的时候，使用 mapPartitions 会出现 OOM（内存溢出）的问题。因为单次函数调用就要处理掉一个 partition 所有的数据，如果内存不够，垃圾回收时是无法回收掉太多对象的，很可能出现 OOM 异常。所以使用这类操作时要慎重！

使用 foreachPartitions 替代 foreach

原理类似于“使用 mapPartitions 替代 map”，也是一次函数调用处理一个 partition 的所有数据，而不是一次函数调用处理一条数据。在实践中发现，foreachPartitions 类的算子，对性能的提升还是很有帮助的。比如在 foreach 函数中，将 RDD 中所有数据写 MySQL，那么如果是普通的 foreach 算子，就会一条数据一条数据地写，每次函数调用可能就会创建一个数据库连接，此时就势必会频繁地创建和销毁数据库连接，性能是非常低下；但是如果用 foreachPartitions 算子一次性处理一个 partition 的数据，那么对于每个 partition，只要创建一个数据库连接即可，然后执行批量插入操作，此时性能是比较高的。实践中发现，对于 1 万条左右的数据量写 MySQL，性能可以提升 30% 以上。

使用 filter 之后进行 coalesce 操作

通常对一个 RDD 执行 filter 算子过滤掉 RDD 中较多数据后（比如 30% 以上的数据），建议使用 coalesce 算子，手动减少 RDD 的 partition 数量，将 RDD 中的数据压缩到更少的 partition 中去。因为 filter 之后，RDD 的每个 partition 中都会有很多数据被过滤掉，此时如果照常进行后续的计算，其实每个 task 处理的 partition 中的数据量并不是很多，有一点资源浪费，而且此时处理的 task 越多，可能速度反而越慢。因此用 coalesce 减少 partition 数量，将 RDD 中的数据压缩到更少的 partition 之后，只要使用更少的 task 即可处理完所有的 partition。在某些场景下，对于性能的提升会有一定的帮助。

使用 repartitionAndSortWithinPartitions 替代 repartition 与 sort 类操作

`repartitionAndSortWithinPartitions` 是 Spark 官网推荐的一个算子，官方建议，如果需要在 `repartition` 重分区之后，还要进行排序，建议直接使用 `repartitionAndSortWithinPartitions` 算子。因为该算子可以一边进行重分区的 `shuffle` 操作，一边进行排序。`shuffle` 与 `sort` 两个操作同时进行，比先 `shuffle` 再 `sort` 来说，性能可能是要高的。

6.8.8 原则七：广播大变量

有时在开发过程中，会遇到需要在算子函数中使用外部变量的场景（尤其是大变量，比如 100M 以上的大集合），那么此时就应该使用 Spark 的广播（Broadcast）功能来提升性能。

在算子函数中使用到外部变量时，默认情况下，Spark 会将该变量复制多个副本，通过网络传输到 task 中，此时每个 task 都有一个变量副本。如果变量本身比较大的话（比如 100M，甚至 1G），那么大量的变量副本在网络中传输的性能开销，以及在各个节点的 Executor 中占用过多内存导致的频繁 GC，都会极大地影响性能。

因此对于上述情况，如果使用的外部变量比较大，建议使用 Spark 的广播功能，对该变量进行广播。广播后的变量，会保证每个 Executor 的内存中，只驻留一份变量副本，而 Executor 中的 task 执行时共享该 Executor 中的那份变量副本。这样的话，可以大大减少变量副本的数量，从而减少网络传输的性能开销，并减少对 Executor 内存的占用开销，降低 GC 的频率。

广播大变量的代码示例



```
1 // 以下代码在算子函数中，使用了外部的变量。 2 // 此时没有做任何特殊操作，每个 task 都会有一份 list1 的副本。 3 val list1 = ... 4 rdd1.map(list1...) 5 6 // 以下代码将 list1 封装成了 Broadcast 类型的广播变量。 7 // 在算子函数中，使用广播变量时，首先会判断当前 task 所在 Executor 内存中，是否有变量副本。 8 // 如果有则直接使用；如果没有则从 Driver 或者其他 Executor 节点上远程拉取一份放到本地 Executor 内存中。 9 // 每个 Executor 内存中，就只会驻留一份广播变量副本。 10 val list1 = ... 11 val list1Broadcast = sc.broadcast(list1) 12 rdd1.map(list1Broadcast...)
```



6.8.9 原则八：使用 Kryo 优化序列化性能

在 Spark 中，主要有三个地方涉及到了序列化：

- 在算子函数中使用到外部变量时，该变量会被序列化后进行网络传输（见“原则七：广播大变量”中的讲解）。
- 将自定义的类型作为 RDD 的泛型类型时（比如 `JavaRDD`，`Student` 是自定义类型），所有自定义类型对象，都会进行序列化。因此这种情况下，也要求自定义的类必须实现 `Serializable` 接口。
- 使用可序列化的持久化策略时（比如 `MEMORY_ONLY_SER`），Spark 会将 RDD 中的每个 `partition` 都序列化成一个大的字节数组。

对于这三种出现序列化的地方，我们都可以通过使用 Kryo 序列化类库，来优化序列化和反序列化的性能。Spark 默认使用的是 Java 的序列化机制，也就是 `ObjectOutputStream/ObjectInputStream` API 来进行序列化和反序列化。但是 Spark 同时支持使用 Kryo 序列化库，Kryo 序列化类库的性能比 Java 序列化类库的性能要高很多。官方介绍，Kryo 序列化机制比 Java 序列化机制，性能高 10 倍左右。Spark 之所以默认没有使用 Kryo 作为序列化类库，是因为 Kryo 要求最好要注册所有需要进行序列化的自定义类型，因此对于开发者来说，这种方式比较麻烦。

以下是使用 Kryo 的代码示例，我们只要设置序列化类，再注册要序列化的自定义类型即可（比如算子函数中使用到的外部变量类型、作为 RDD 泛型类型的自定义类型等）：

```
1 // 创建 SparkConf 对象。 2 val conf = new
SparkConf().setMaster(...).setAppName(...) 3 // 设置序列化器为
KryoSerializer。 4 conf.set("spark.serializer",
"org.apache.spark.serializer.KryoSerializer") 5 // 注册要序列化的自定义类型。 6 conf.registerKryoClasses(Array(classOf[MyClass1],
classOf[MyClass2]))
```

6.8.10 原则九：优化数据结构

Java 中，有三种类型比较耗费内存：

- 对象，每个 Java 对象都有对象头、引用等额外的信息，因此比较占用内存空间。
- 字符串，每个字符串内部都有一个字符数组以及长度等额外信息。
- 集合类型，比如 `HashMap`、`LinkedList` 等，因为集合类型内部通常会使用一些内部类来封装集合元素，比如 `Map.Entry`。

因此 Spark 官方建议，在 Spark 编码实现中，特别是对于算子函数中的代码，尽量不要使用上述三种数据结构，尽量使用字符串替代对象，使用原始类型（比如 `Int`、`Long`）替代

字符串，使用数组替代集合类型，这样尽可能地减少内存占用，从而降低 GC 频率，提升性能。

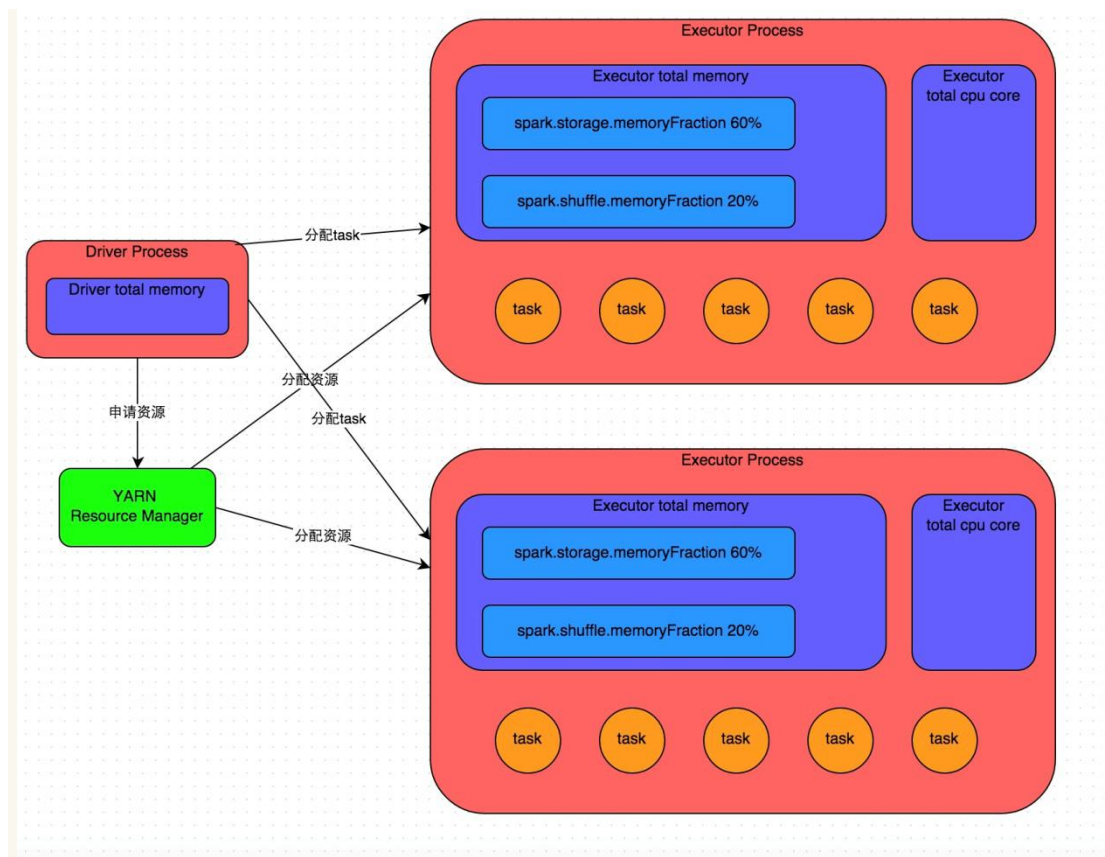
但是在笔者的编码实践中发现，要做到该原则其实并不容易。因为我们同时考虑到代码的可维护性，如果一个代码中，完全没有任何对象抽象，全部是字符串拼接的方式，那么对于后续的代码维护和修改，无疑是一场巨大的灾难。同理，如果所有操作都基于数组实现，而不使用 `HashMap`、`LinkedList` 等集合类型，那么对于我们的编码难度以及代码可维护性，也是一个极大的挑战。因此笔者建议，在可能以及合适的情况下，使用占用内存较少的数据结构，但是前提是要保证代码的可维护性。

6.9 资源调优

6.9.1 调优概述

在开发完 `Spark` 作业之后，就该为作业配置合适的资源了。`Spark` 的资源参数，基本都可以在 `spark-submit` 命令中作为参数设置。很多 `Spark` 初学者，通常不知道该设置哪些必要的参数，以及如何设置这些参数，最后就只能胡乱设置，甚至压根儿不设置。资源参数设置的不合理，可能会导致没有充分利用集群资源，作业运行会极其缓慢；或者设置的资源过大，队列没有足够的资源来提供，进而导致各种异常。总之，无论是哪种情况，都会导致 `Spark` 作业的运行效率低下，甚至根本无法运行。因此我们必须对 `Spark` 作业的资源使用原理有一个清晰的认识，并知道在 `Spark` 作业运行过程中，有哪些资源参数是可以设置的，以及如何设置合适的参数值。

Spark 作业基本运行原理



详细原理见上图。我们使用 `spark-submit` 提交一个 Spark 作业之后，这个作业就会启动一个对应的 Driver 进程。根据你使用的部署模式（`deploy-mode`）不同，Driver 进程可能在本地启动，也可能在集群中某个工作节点上启动。Driver 进程本身会根据我们设置的参数，占有一定数量的内存和 CPU core。而 Driver 进程要做的第一件事情，就是向集群管理器（可以是 Spark Standalone 集群，也可以是其他的资源管理集群，美团·大众点评使用的是 YARN 作为资源管理集群）申请运行 Spark 作业需要使用的资源，这里的资源指的就是 Executor 进程。YARN 集群管理器会根据我们为 Spark 作业设置的资源参数，在各个工作节点上，启动一定数量的 Executor 进程，每个 Executor 进程都占有一定数量的内存和 CPU core。

在申请到了作业执行所需的资源之后，Driver 进程就会开始调度和执行我们编写的作业代码了。Driver 进程会将我们编写的 Spark 作业代码分拆为多个 stage，每个 stage 执行一部分代码片段，并为每个 stage 创建一批 task，然后将这些 task 分配到各个 Executor 进程中执行。task 是最小的计算单元，负责执行一模一样的计算逻辑（也就是我们自己编写的某个代码片段），只是每个 task 处理的数据不同而已。一个 stage 的所有 task 都执行完毕之后，会在各个节点本地的磁盘文件中写入计算中间结果，然后 Driver 就会调度运行下一个 stage。下一个 stage 的 task 的输入数据就是上一个 stage 输出的中间结果。如此循环往复，直到将我们自己编写的代码逻辑全部执行完，并且计算完所有的数据，得到我们想要的结果为止。

Spark 是根据 shuffle 类算子来进行 stage 的划分。如果我们的代码中执行了某个 shuffle 类算子（比如 `reduceByKey`、`join` 等），那么就会在该算子处，划分出一个 stage 界限来。可以大致理解为，shuffle 算子执行之前的代码会被划分为一个 stage，shuffle 算子执行以及之后的代码会被划分为下一个 stage。因此一个 stage 刚开始执行的时候，它的每个 task 可能都会从上一个 stage 的 task 所在的节点，去通过网络传输拉取需要自己处理的所有 key，然后对拉取到的所有相同的 key 使用我们自己编写的算子函数执行聚合操作（比如 `reduceByKey()` 算子接收的函数）。这个过程就是 shuffle。

当我们在代码中执行了 `cache/persist` 等持久化操作时，根据我们选择的持久化级别的不同，每个 task 计算出来的数据也会保存到 Executor 进程的内存或者所在节点的磁盘文件中。

因此 Executor 的内存主要分为三块：第一块是让 task 执行我们自己编写的代码时使用，默认是占 Executor 总内存的 20%；第二块是让 task 通过 shuffle 过程拉取了上一个 stage 的 task 的输出后，进行聚合等操作时使用，默认也是占 Executor 总内存的 20%；第三块是让 RDD 持久化时使用，默认占 Executor 总内存的 60%。

task 的执行速度是跟每个 Executor 进程的 CPU core 数量有直接关系的。一个 CPU core 同一时间只能执行一个线程。而每个 Executor 进程上分配到的多个 task，都是以每个 task 一条线程的方式，多线程并发运行的。如果 CPU core 数量比较充足，而且分配到的 task 数量比较合理，那么通常来说，可以比较快速和高效地执行完这些 task 线程。

以上就是 Spark 作业的基本运行原理的说明，大家可以结合上图来理解。理解作业基本原理，是我们进行资源参数调优的基本前提。

6.9.2 资源参数调优

了解完了 Spark 作业运行的基本原理之后，对资源相关的参数就容易理解了。所谓的 Spark 资源参数调优，其实主要就是对 Spark 运行过程中各个使用资源的地方，通过调节各种参数，来优化资源使用的效率，从而提升 Spark 作业的执行性能。以下参数就是 Spark 中主要的资源参数，每个参数都对应着作业运行原理中的某个部分，我们同时也给出了一个调优的参考值。

num-executors

- 参数说明：该参数用于设置 Spark 作业总共要用多少个 Executor 进程来执行。Driver 在向 YARN 集群管理器申请资源时，YARN 集群管理器会尽可能按照你的设置来在集群的各个工作节点上，启动相应数量的 Executor 进程。这个参数非常之重要，如果不设置的话，默认只会给你启动少量的 Executor 进程，此时你的 Spark 作业的运行速度是非常慢的。
- 参数调优建议：每个 Spark 作业的运行一般设置 50~100 个左右的 Executor 进程比较合适，设置太少或太多的 Executor 进程都不好。设置的太少，无法充分利用集群资源；设置的太多的话，大部分队列可能无法给予充分的资源。

executor-memory

- 参数说明：该参数用于设置每个 **Executor** 进程的内存。**Executor** 内存的大小，很多时候直接决定了 **Spark** 作业的性能，而且跟常见的 **JVM OOM** 异常，也有直接的关联。
- 参数调优建议：每个 **Executor** 进程的内存设置 **4G~8G** 较为合适。但是这只是一个参考值，具体的设置还是得根据不同部门的资源队列来定。可以看看自己团队的资源队列的最大内存限制是多少，**num-executors** 乘以 **executor-memory**，是不能超过队列的最大内存量的。此外，如果你是跟团队里其他人共享这个资源队列，那么申请的内存量最好不要超过资源队列最大总内存的 **1/3~1/2**，避免你自己的 **Spark** 作业占用了队列所有的资源，导致别的同学的作业无法运行。

executor-cores

- 参数说明：该参数用于设置每个 **Executor** 进程的 **CPU core** 数量。这个参数决定了每个 **Executor** 进程并行执行 **task** 线程的能力。因为每个 **CPU core** 同一时间只能执行一个 **task** 线程，因此每个 **Executor** 进程的 **CPU core** 数量越多，越能够快速地执行完分配给自己的所有 **task** 线程。
- 参数调优建议：**Executor** 的 **CPU core** 数量设置为 **2~4** 个较为合适。同样得根据不同部门的资源队列来定，可以看看自己的资源队列的最大 **CPU core** 限制是多少，再依据设置的 **Executor** 数量，来决定每个 **Executor** 进程可以分配到几个 **CPU core**。同样建议，如果是跟他人共享这个队列，那么 **num-executors * executor-cores** 不要超过队列总 **CPU core** 的 **1/3~1/2** 左右比较合适，也是避免影响其他同学的作业运行。

driver-memory

- 参数说明：该参数用于设置 **Driver** 进程的内存。
- 参数调优建议：**Driver** 的内存通常来说不设置，或者设置 **1G** 左右应该就够了。唯一需要注意的一点是，如果需要使用 **collect** 算子将 **RDD** 的数据全部拉取到 **Driver** 上进行处理，那么必须确保 **Driver** 的内存足够大，否则会出现 **OOM** 内存溢出的问题。

spark.default.parallelism

- 参数说明：该参数用于设置每个 **stage** 的默认 **task** 数量。这个参数极为重要，如果不设置可能会直接影响你的 **Spark** 作业性能。
- 参数调优建议：**Spark** 作业的默认 **task** 数量为 **500~1000** 个较为合适。很多同学常犯的一个错误就是不去设置这个参数，那么此时就会导致 **Spark** 自己根据底层 **HDFS** 的 **block** 数量来设置 **task** 的数量，默认是一个 **HDFS block** 对应一个 **task**。通常来

说，Spark 默认设置的数量是偏少的（比如就几十个 task），如果 task 数量偏少的话，就会导致你前面设置好的 Executor 的参数都前功尽弃。试想一下，无论你的 Executor 进程有多少个，内存和 CPU 有多大，但是 task 只有 1 个或者 10 个，那么 90% 的 Executor 进程可能根本就没有 task 执行，也就是白白浪费了资源！因此 Spark 官网建议的设置原则是，设置该参数为 `num-executors * executor-cores` 的 2~3 倍较为合适，比如 Executor 的总 CPU core 数量为 300 个，那么设置 1000 个 task 是可以的，此时可以充分地利用 Spark 集群的资源。

spark.storage.memoryFraction

- 参数说明：该参数用于设置 RDD 持久化数据在 Executor 内存中能占的比例，默认是 0.6。也就是说，默认 Executor 60% 的内存，可以用来保存持久化的 RDD 数据。根据你选择的不同的持久化策略，如果内存不够时，可能数据就不会持久化，或者数据会写入磁盘。
- 参数调优建议：如果 Spark 作业中，有较多的 RDD 持久化操作，该参数的值可以适当提高一些，保证持久化的数据能够容纳在内存中。避免内存不够缓存所有的数据，导致数据只能写入磁盘中，降低了性能。但是如果 Spark 作业中的 shuffle 类操作比较多，而持久化操作比较少，那么这个参数的值适当降低一些比较合适。此外，如果发现作业由于频繁的 gc 导致运行缓慢（通过 spark web ui 可以观察到作业的 gc 耗时），意味着 task 执行用户代码的内存不够用，那么同样建议调低这个参数的值。


spark.shuffle.memoryFraction

- 参数说明：该参数用于设置 shuffle 过程中一个 task 拉取到上个 stage 的 task 的输出后，进行聚合操作时能够使用的 Executor 内存的比例，默认是 0.2。也就是说，Executor 默认只有 20% 的内存用来进行该操作。shuffle 操作在进行聚合时，如果发现使用的内存超出了这个 20% 的限制，那么多余的数据就会溢写到磁盘文件中，此时就会极大地降低性能。
- 参数调优建议：如果 Spark 作业中的 RDD 持久化操作较少，shuffle 操作较多时，建议降低持久化操作的内存占比，提高 shuffle 操作的内存占比比例，避免 shuffle 过程中数据过多时内存不够用，必须溢写到磁盘上，降低了性能。此外，如果发现作业由于频繁的 gc 导致运行缓慢，意味着 task 执行用户代码的内存不够用，那么同样建议调低这个参数的值。

资源参数的调优，没有一个固定的值，需要同学们根据自己的实际情况（包括 Spark 作业中的 shuffle 操作数量、RDD 持久化操作数量以及 spark web ui 中显示的作业 gc 情况），同时参考本篇文章中给出的原理以及调优建议，合理地设置上述参数。

6.9.3 资源参数参考示例

以下是一份 `spark-submit` 命令的示例，大家可以参考一下，并根据自己的实际情况进行调节：



```
1 ./bin/spark-submit \ 2  --master yarn-cluster \ 3  --num-executors  
100 \ 4  --executor-memory 6G \ 5  --executor-cores 4 \ 6  
--driver-memory 1G \ 7  --conf spark.default.parallelism=1000 \ 8  
--conf spark.storage.memoryFraction=0.5 \ 9  --conf  
spark.shuffle.memoryFraction=0.3 \
```



6.10 数据倾斜调优

6.10.1 调优概述

有的时候，我们可能会遇到大数据计算中一个最棘手的问题——数据倾斜，此时 **Spark** 作业的性能会比期望差很多。数据倾斜调优，就是使用各种技术方案解决不同类型的数据倾斜问题，以保证 **Spark** 作业的性能。

6.10.2 数据倾斜发生时的现象

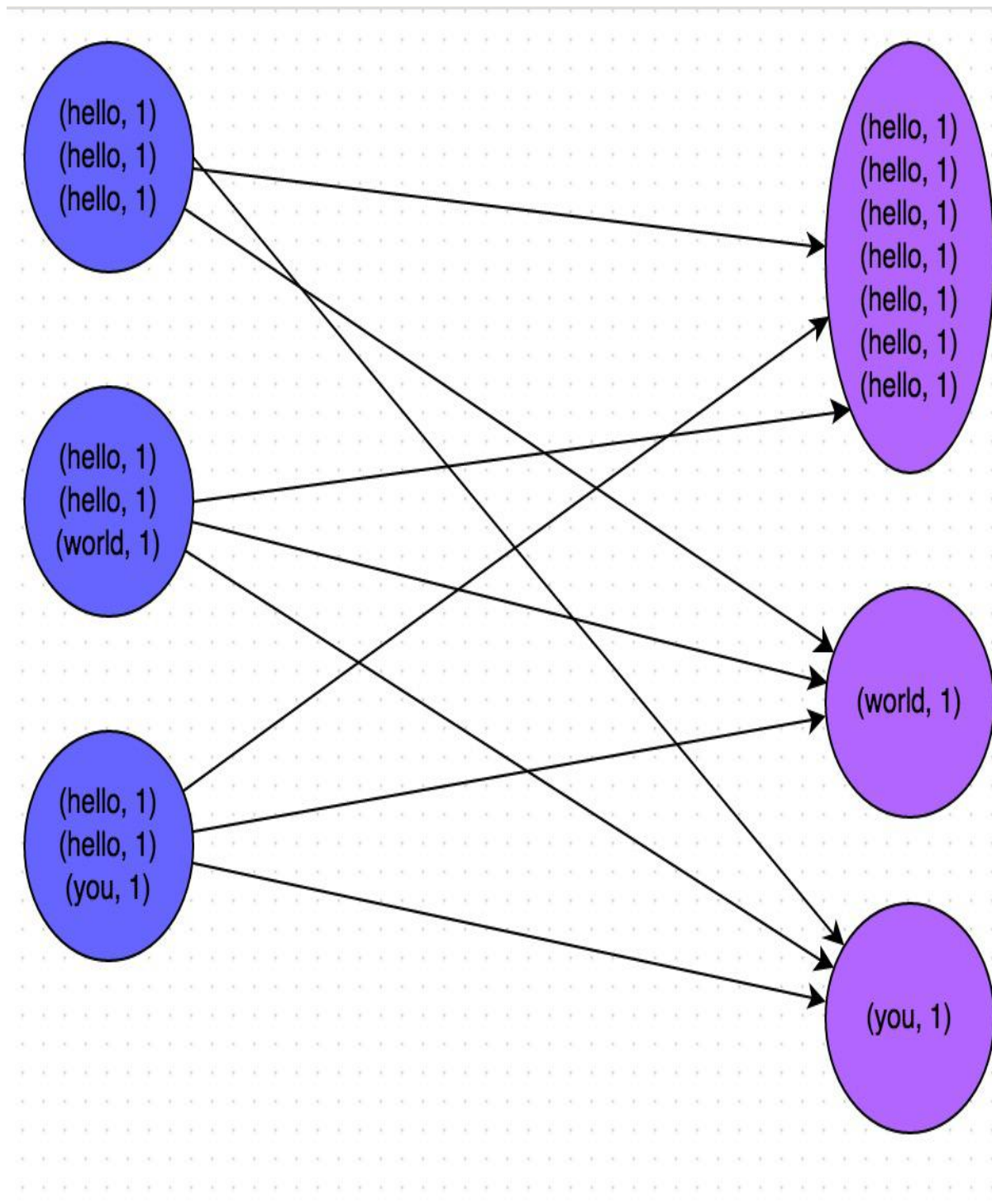
- 绝大多数 **task** 执行得都非常快，但个别 **task** 执行极慢。比如，总共有 1000 个 **task**，997 个 **task** 都在 1 分钟之内执行完了，但是剩余两三个 **task** 却要一两个小时。这种情况很常见。
- 原本能够正常执行的 **Spark** 作业，某天突然报出 **OOM**（内存溢出）异常，观察异常栈，是我们写的业务代码造成的。这种情况比较少见。

6.10.3 数据倾斜发生的原理

数据倾斜的原理很简单：在进行 **shuffle** 的时候，必须将各个节点上相同的 **key** 拉取到某个节点上的一个 **task** 来进行处理，比如按照 **key** 进行聚合或 **join** 等操作。此时如果某个 **key** 对应的数据量特别大的话，就会发生数据倾斜。比如大部分 **key** 对应 10 条数据，但是个别 **key** 却对应了 100 万条数据，那么大部分 **task** 可能就只会分配到 10 条数据，然后 1 秒钟就运行完了；但是个别 **task** 可能分配到了 100 万数据，要运行一两个小时。因此，整个 **Spark** 作业的运行进度是由运行时间最长的那个 **task** 决定的。

因此出现数据倾斜的时候，**Spark** 作业看起来会运行得非常缓慢，甚至可能因为某个 **task** 处理的数据量过大导致内存溢出。

下图就是一个很清晰的例子：**hello** 这个 **key**，在三个节点上对应了总共 7 条数据，这些数据都会被拉取到同一个 **task** 中进行处理；而 **world** 和 **you** 这两个 **key** 分别才对应 1 条数据，所以另外两个 **task** 只要分别处理 1 条数据即可。此时第一个 **task** 的运行时间可能是另外两个 **task** 的 7 倍，而整个 **stage** 的运行速度也由运行最慢的那个 **task** 所决定。



6.10.4 如何定位导致数据倾斜的代码

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。

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，那么本地是直接可以看到 `log` 的，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。

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。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。

85	154	0	SUCCESS	PROCESS_LOCAL	3 / rz-data-hdp-dn0912.rz.sankuai.com	2016/01/29 13:42:02	3.2 min	0.9 s	807.7 KB / 8691
86	155	0	SUCCESS	PROCESS_LOCAL	46 / rz-data-hdp-dn0890.rz.sankuai.com	2016/01/29 13:42:02	49 s	0.5 s	531.4 KB / 5309
87	156	0	SUCCESS	PROCESS_LOCAL	92 / rz-data-hdp-dn1275.rz.sankuai.com	2016/01/29 13:42:02	31 s	0.6 s	360.7 KB / 3696
88	157	0	SUCCESS	PROCESS_LOCAL	64 / rz-data-hdp-dn0121.rz.sankuai.com	2016/01/29 13:42:02	27 s	0.4 s	406.1 KB / 4104
89	158	0	SUCCESS	PROCESS_LOCAL	13 / rz-data-hdp-dn1184.rz.sankuai.com	2016/01/29 13:42:02	14 s	0.4 s	347.3 KB / 3561
90	159	0	SUCCESS	PROCESS_LOCAL	5 / rz-data-hdp-dn0912.rz.sankuai.com	2016/01/29 13:42:02	13 s	0.3 s	351.4 KB / 3622
91	160	0	RUNNING	PROCESS_LOCAL	90 / rz-data-hdp-dn0059.rz.sankuai.com	2016/01/29 13:42:02	3.9 min	0.8 s	1617.0 KB / 18545
92	161	0	SUCCESS	PROCESS_LOCAL	87 / rz-data-hdp-dn0879.rz.sankuai.com	2016/01/29 13:42:02	26 s	0.4 s	318.1 KB / 3081
93	162	0	SUCCESS	PROCESS_LOCAL	55 / rz-data-hdp-dn0875.rz.sankuai.com	2016/01/29 13:42:02	19 s	0.5 s	359.6 KB / 3574
94	163	0	RUNNING	PROCESS_LOCAL	82 / rz-data-hdp-dn0430.rz.sankuai.com	2016/01/29 13:42:02	3.9 min	0.8 s	2023.4 KB / 22812
95	164	0	SUCCESS	PROCESS_LOCAL	99 / rz-data-hdp-dn0817.rz.sankuai.com	2016/01/29 13:42:02	5 s	0.2 s	188.1 KB / 1426
96	165	0	SUCCESS	PROCESS_LOCAL	56 / rz-data-hdp-dn0875.rz.sankuai.com	2016/01/29 13:42:02	10 s	0.3 s	214.5 KB / 1683
97	166	0	SUCCESS	PROCESS_LOCAL	71 / rz-data-hdp-dn0576.rz.sankuai.com	2016/01/29 13:42:02	2.9 min	0.4 s	673.8 KB / 6932
98	167	0	SUCCESS	PROCESS_LOCAL	77 / rz-data-hdp-dn0242.rz.sankuai.com	2016/01/29 13:42:02	13 s	0.3 s	276.3 KB / 2349
99	168	0	RUNNING	PROCESS_LOCAL	58 / rz-data-hdp-dn0491.rz.sankuai.com	2016/01/29 13:42:02	3.9 min	1 s	1321.0 KB / 14508

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一个 `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 Spark 最基础的入门程序——单词计数来举例，如何用最简单的方法大致推算出一个 stage 对应的代码。如下示例，在整个代码中，只有一个 reduceByKey 是会发生 shuffle 的算子，因此就可以认为，以这个算子为界限，会划分出前后两个 stage。

- stage0，主要是执行从 textFile 到 map 操作，以及执行 shuffle write 操作。shuffle write 操作，我们可以简单理解为对 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，stage1 的各个 task 一开始运行，就会首先执行 shuffle read 操作。执行 shuffle read 操作的 task，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。



```
1 val conf = new SparkConf() 2 val sc = new SparkContext(conf) 3 4 val lines = sc.textFile("hdfs://...") 5 val words = lines.flatMap(_.split(" ")) 6 val pairs = words.map((_, 1)) 7 val wordCounts = pairs.reduceByKey(_ + _) 8 9 wordCounts.collect().foreach(println(_))
```



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由 reduceByKey 算子导致的数据倾斜问题。比如某个单词出现了 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。

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。

6.10.5 查看导致数据倾斜的 key 的数据分布情况

知道了数据倾斜发生在哪里之后，通常需要分析一下那个执行了 **shuffle** 操作并且导致了数据倾斜的 **RDD/Hive** 表，查看一下其中 **key** 的分布情况。这主要是为之后选择哪一种技术方案提供依据。针对不同的 **key** 分布与不同的 **shuffle** 算子组合起来的各种情况，可能需要选择不同的技术方案来解决。

此时根据你执行操作的情况不同，可以有很多种查看 **key** 分布的方式：

1. 如果是 **Spark SQL** 中的 **group by**、**join** 语句导致的数据倾斜，那么就查询一下 **SQL** 中使用的表的 **key** 分布情况。
2. 如果是对 **Spark RDD** 执行 **shuffle** 算子导致的数据倾斜，那么可以在 **Spark** 作业中加入查看 **key** 分布的代码，比如 **RDD.countByKey()**。然后对统计出来的各个 **key** 出现的次数，**collect/take** 到客户端打印一下，就可以看到 **key** 的分布情况。

举例来说，对于上面所说的单词计数程序，如果确定了是 **stage1** 的 **reduceByKey** 算子导致了数据倾斜，那么就应该看看进行 **reduceByKey** 操作的 **RDD** 中的 **key** 分布情况，在这个例子中指的是 **pairs RDD**。如下示例，我们可以先对 **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。

```
1 val sampledPairs = pairs.sample(false, 0.1) 2 val sampledWordCounts =
sampledPairs.countByKey() 3 sampledWordCounts.foreach(println(_))
```

6.10.6 数据倾斜的解决方案

解决方案一：使用 **Hive ETL** 预处理数据

方案适用场景：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。

方案实现思路：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。

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。

方案优点：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。

方案缺点：治标不治本，Hive ETL 中还是会发生数据倾斜。

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。

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。

解决方案二：过滤少数导致倾斜的 key

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。

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。

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。

方案优点：实现简单，而且效果也很好，可以完全规避掉数据倾斜。

方案缺点：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。

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。

解决方案三：提高 **shuffle** 操作的并行度

方案适用场景：如果我们必须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。

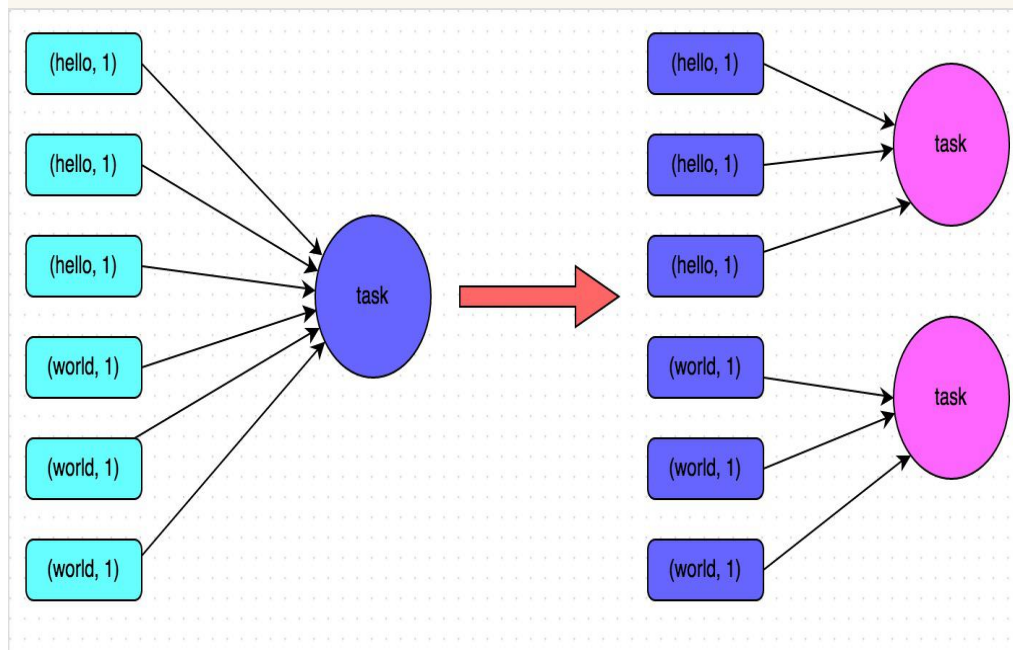
方案实现思路：在对 RDD 执行 **shuffle** 算子时，给 **shuffle** 算子传入一个参数，比如 **reduceByKey(1000)**，该参数就设置了这个 **shuffle** 算子执行时 **shuffle read task** 的数量。对于 Spark SQL 中的 **shuffle** 类语句，比如 **group by**、**join** 等，需要设置一个参数，即 **spark.sql.shuffle.partitions**，该参数代表了 **shuffle read task** 的并行度，该值默认是 200，对于很多场景来说都有点过小。

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。

方案优点：实现起来比较简单，可以有效缓解和减轻数据倾斜的影响。

方案缺点：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。

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。



解决方案四：两阶段聚合（局部聚合+全局聚合）

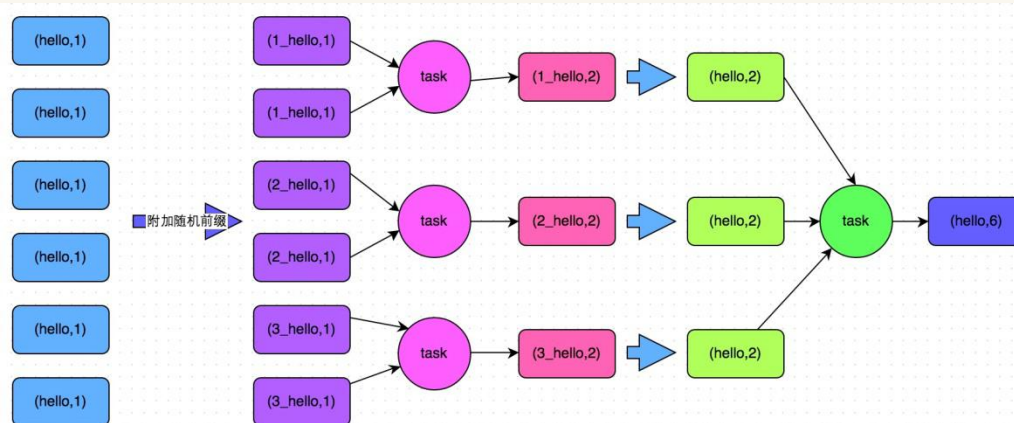
方案适用场景：对 RDD 执行 `reduceByKey` 等聚合类 shuffle 算子或者在 Spark SQL 中使用 `group by` 语句进行分组聚合时，比较适用这种方案。

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)`。

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。

方案优点：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。

方案缺点：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。



```
1 // 第一步，给 RDD 中的每个 key 都打上一个随机前缀。 2
JavaPairRDD<String, Long> randomPrefixRdd = rdd.mapToPair( 3
new PairFunction<Tuple2<Long, Long>, String, Long>() { 4
private static final long serialVersionUID = 1L; 5
@Override 6
public Tuple2<String, Long> call(Tuple2<Long, Long> tuple) 7
throws Exception { 8
Random random = new Random(); 9
int prefix =
random.nextInt(10); 10
return new Tuple2<String,
Long>(prefix + "_" + tuple._1, tuple._2); 11
} 12
});
13 14 // 第二步，对打上随机前缀的 key 进行局部聚合。 15
JavaPairRDD<String, Long> localAggrRdd =
randomPrefixRdd.reduceByKey( 16
new Function2<Long, Long,
Long>() { 17
private static final long serialVersionUID = 1L;
18
@Override 19
public Long call(Long v1, Long
v2) throws Exception { 20
return v1 + v2;
21
} 22
}); 23 24 // 第三步，去除 RDD 中每个 key
的随机前缀。 25 JavaPairRDD<Long, Long> removedRandomPrefixRdd =
```



```

localAggrRdd.mapToPair( 26          new
PairFunction<Tuple2<String,Long>, Long, Long>() { 27
private static final long serialVersionUID = 1L; 28
@Override 29          public Tuple2<Long, Long> call(Tuple2<String,
Long> tuple) 30          throws Exception { 31
long originalKey = Long.valueOf(tuple._1.split("_")[1]); 32
return new Tuple2<Long, Long>(originalKey, tuple._2); 33          }
34          }); 35 36 // 第四步,对去除了随机前缀的 RDD 进行全局聚合。 37
JavaPairRDD<Long, Long> globalAggrRdd =
removedRandomPrefixRdd.reduceByKey( 38          new Function2<Long,
Long, Long>() { 39          private static final long
serialVersionUID = 1L; 40          @Override 41          public
Long call(Long v1, Long v2) throws Exception { 42          return
v1 + v2; 43          } 44          });

```



解决方案五：将 reduce join 转为 map join

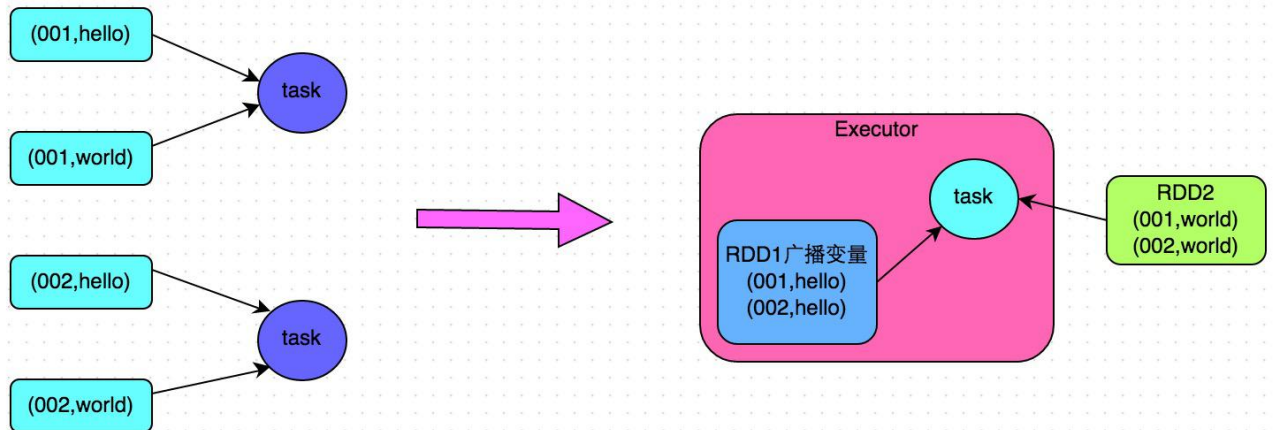
方案适用场景：在对 RDD 使用 join 类操作，或者是在 Spark SQL 中使用 join 语句时，而且 join 操作中的一个 RDD 或表的数据量比较小（比如几百 M 或者一两 G），比较适用此方案。

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；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。

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。

方案优点：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。

方案缺点：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。



```
1 // 首先将数据量比较小的 RDD 的数据，collect 到 Driver 中来。 2
List<Tuple2<Long, Row>> rdd1Data = rdd1.collect() 3 // 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。 4 // 可以尽可能节省内存空间，并且减少网络传输性能开销。 5
final Broadcast<List<Tuple2<Long, Row>>> rdd1DataBroadcast =
sc.broadcast(rdd1Data); 6 7 // 对另外一个 RDD 执行 map 类操作，而不再
是 join 类操作。 8 JavaPairRDD<String, Tuple2<String, Row>> joinedRdd =
rdd2.mapToPair( 9         new PairFunction<Tuple2<Long, String>,
String, Tuple2<String, Row>>() { 10             private static final
long serialVersionUID = 1L; 11             @Override 12
public Tuple2<String, Tuple2<String, Row>> call(Tuple2<Long, String>
tuple) 13                 throws Exception { 14                     //
在算子函数中，通过广播变量，获取到本地 Executor 中的 rdd1 数据。 15
List<Tuple2<Long, Row>> rdd1Data = rdd1DataBroadcast.value(); 16
// 可以将 rdd1 的数据转换为一个 Map，便于后面进行 join 操作。 17
Map<Long, Row> rdd1DataMap = new HashMap<Long, Row>(); 18
for(Tuple2<Long, Row> data : rdd1Data) { 19
    rdd1DataMap.put(data._1, data._2); 20
} 21
// 获取当前 RDD 数据的 key 以及 value。 22
String key = tuple._1; 23
String value = tuple._2; 24
// 从 rdd1 数据 Map 中，根据 key 获取到可以 join 到的数据。 25
Row rdd1Value = rdd1DataMap.get(key); 26
return new
Tuple2<String, String>(key, new Tuple2<String, Row>(value, rdd1Value));
```

```
27         } 28     }); 29 30 // 这里得提示一下。 31 // 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。 32 // 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。 33 // rdd2 中每条数据都可能会返回多条 join 后的数据。
```



解决方案六：采样倾斜 **key** 并分拆 **join** 操作

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。

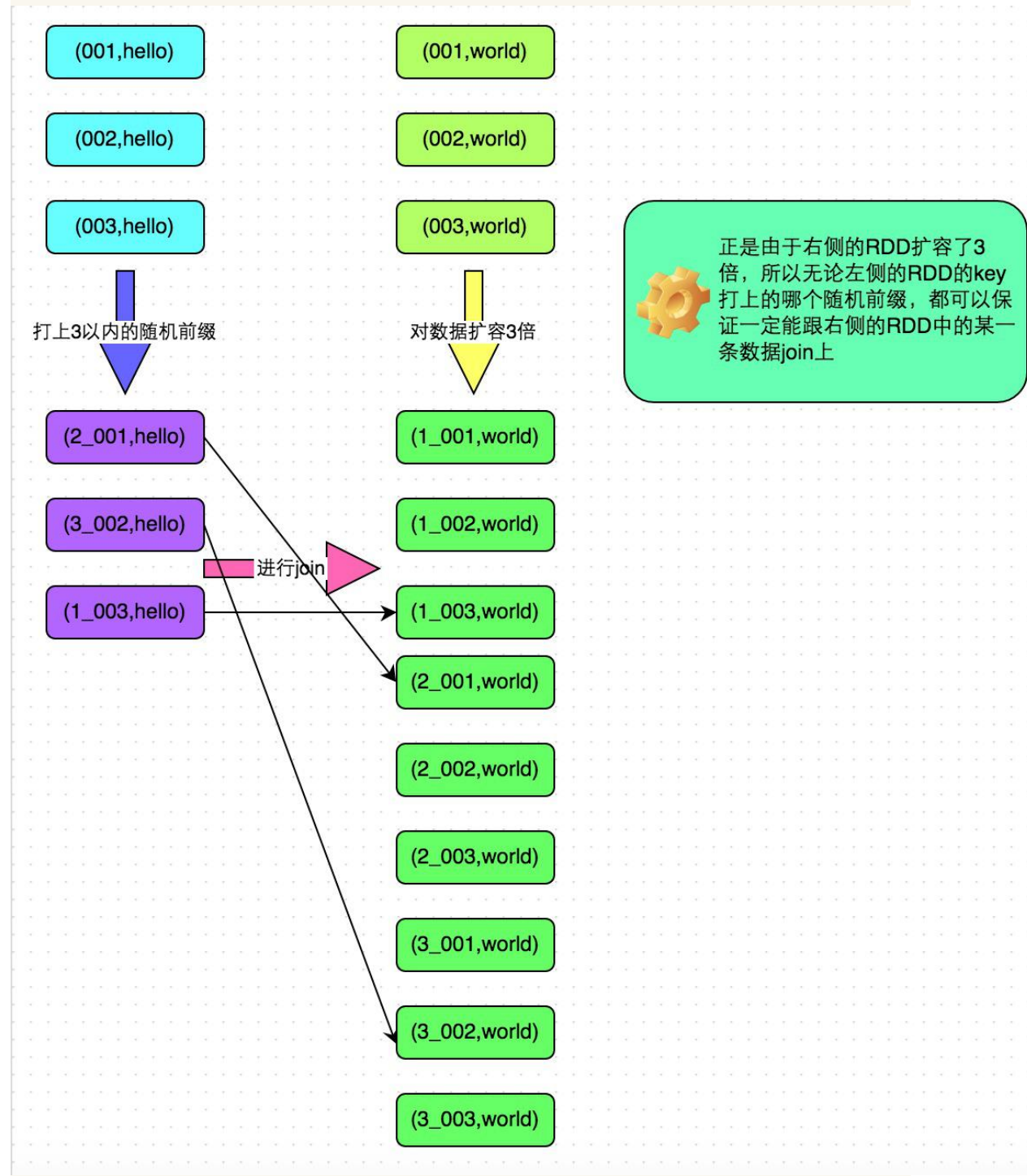
方案实现思路：

- 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新的 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。

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。

方案优点：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。

方案缺点：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。





```
1 // 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。 2 JavaPairRDD<Long, String> sampledRDD = rdd1.sample(false, 0.1);
3 4 // 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。
5 // 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。 6 // 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。 7 JavaPairRDD<Long, Long> mappedSampledRDD
= sampledRDD.mapToPair( 8 new
PairFunction<Tuple2<Long, String>, Long, Long>() { 9
private static final long serialVersionUID = 1L; 10
@Override 11 public Tuple2<Long, Long> call(Tuple2<Long,
String> tuple) 12 throws Exception { 13
return new Tuple2<Long, Long>(tuple._1, 1L); 14 }
15 }); 16 JavaPairRDD<Long, Long> countedSampledRDD =
mappedSampledRDD.reduceByKey( 17 new Function2<Long, Long,
Long>() { 18 private static final long serialVersionUID =
1L; 19 @Override 20 public Long call(Long v1,
Long v2) throws Exception { 21 return v1 + v2;
22 } 23 }); 24 JavaPairRDD<Long, Long>
reversedSampledRDD = countedSampledRDD.mapToPair( 25 new
PairFunction<Tuple2<Long, Long>, Long, Long>() { 26
private static final long serialVersionUID = 1L; 27
@Override 28 public Tuple2<Long, Long> call(Tuple2<Long,
Long> tuple) 29 throws Exception { 30
return new Tuple2<Long, Long>(tuple._2, tuple._1); 31 }
32 }); 33 final Long skewedUserid =
reversedSampledRDD.sortByKey(false).take(1).get(0)._2; 34 35 // 从
rdd1 中分拆出导致数据倾斜的 key, 形成独立的 RDD。 36 JavaPairRDD<Long,
String> skewedRDD = rdd1.filter( 37 new
Function<Tuple2<Long, String>, Boolean>() { 38 private
static final long serialVersionUID = 1L; 39 @Override 40
public Boolean call(Tuple2<Long, String> tuple) throws Exception { 41
return tuple._1.equals(skewedUserid); 42 }
43 }); 44 // 从 rdd1 中分拆出不导致数据倾斜的普通 key, 形成独
立的 RDD。 45 JavaPairRDD<Long, String> commonRDD = rdd1.filter( 46
new Function<Tuple2<Long, String>, Boolean>() { 47 private
static final long serialVersionUID = 1L; 48 @Override 49
public Boolean call(Tuple2<Long, String> tuple) throws Exception { 50
return !tuple._1.equals(skewedUserid); 51 }
52 }); 53 54 // rdd2, 就是那个所有 key 的分布相对较为均匀的
rdd。 55 // 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。 56 // 对
扩容的每条数据, 都打上 0~100 的前缀。 57 JavaPairRDD<String, Row>
```



```

skewedRdd2 = rdd2.filter( 58          new Function<Tuple2<Long, Row>,
Boolean>() { 59          private static final long serialVersionUID
= 1L; 60          @Override 61          public Boolean
call(Tuple2<Long, Row> tuple) throws Exception { 62
return tuple._1.equals(skewedUserId); 63          }
64          }).flatMapToPair(new PairFlatMapFunction<Tuple2<Long, Row>,
String, Row>() { 65          private static final long
serialVersionUID = 1L; 66          @Override 67
public Iterable<Tuple2<String, Row>> call( 68
Tuple2<Long, Row> tuple) throws Exception { 69          Random
random = new Random(); 70          List<Tuple2<String, Row>>
list = new ArrayList<Tuple2<String, Row>>(); 71
for(int i = 0; i < 100; i++) { 72          list.add(new
Tuple2<String, Row>(i + "_" + tuple._1, tuple._2));
73          } 74          return list;
75          } 76 77          }); 78 79 // 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。 80 // 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。 81 JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD1 =
skewedRDD.mapToPair( 82          new PairFunction<Tuple2<Long, String>,
String, String>() { 83          private static final long
serialVersionUID = 1L; 84          @Override 85
public Tuple2<String, String> call(Tuple2<Long, String> tuple) 86
throws Exception { 87          Random random = new Random(); 88
int prefix = random.nextInt(100); 89          return new
Tuple2<String, String>(prefix + "_" + tuple._1, tuple._2);
90          } 91          })
92          .join(skewedUserId2infoRDD) 93          .mapToPair(new
PairFunction<Tuple2<String, Tuple2<String, Row>>, Long, Tuple2<String,
Row>>() { 94          private static final long
serialVersionUID = 1L; 95          @Override 96
public Tuple2<Long, Tuple2<String, Row>> call( 97
Tuple2<String, Tuple2<String, Row>> tuple) 98
throws Exception { 99          long key =
Long.valueOf(tuple._1.split("_")[1]); 100
return new Tuple2<Long, Tuple2<String, Row>>(key, tuple._2);
101          } 102          }); 103 104 //
将 rdd1 中分拆出来的包含普通 key 的独立 rdd，直接与 rdd2 进行 join。 105
JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD2 =
commonRDD.join(rdd2); 106 107 // 将倾斜 key join 后的结果与普通 key join
后的结果，union 起来。 108 // 就是最终的 join 结果。 109 JavaPairRDD<Long,
Tuple2<String, Row>> joinedRDD = joinedRDD1.union(joinedRDD2);

```



解决方案七：使用随机前缀和扩容 RDD 进行 join

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。

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。

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。

方案优点：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。

方案缺点：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。

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。



```
1 // 首先将其中一个 key 分布相对较为均匀的 RDD 膨胀 100 倍。 2  
JavaPairRDD<String, Row> expandedRDD = rdd1.flatMapToPair( 3
```

```

new PairFlatMapFunction<Tuple2<Long, Row>, String, Row>() { 4
private static final long serialVersionUID = 1L; 5
@Override 6 public Iterable<Tuple2<String, Row>>
call(Tuple2<Long, Row> tuple) 7 throws Exception
{ 8 List<Tuple2<String, Row>> list = new
ArrayList<Tuple2<String, Row>>(); 9 for(int i = 0; i <
100; i++) { 10 list.add(new Tuple2<String, Row>(0 +
"_" + tuple._1, tuple._2)); 11 } 12
return list; 13 } 14 }); 15 16 // 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。 17
JavaPairRDD<String, String> mappedRDD = rdd2.mapToPair( 18 new
PairFunction<Tuple2<Long, String>, String, String>() { 19
private static final long serialVersionUID = 1L; 20
@Override 21 public Tuple2<String, String> call(Tuple2<Long,
String> tuple) 22 throws Exception { 23
Random random = new Random(); 24 int prefix =
random.nextInt(100); 25 return new Tuple2<String,
String>(prefix + "_" + tuple._1, tuple._2); 26 }
27 }); 28 29 // 将两个处理后的 RDD 进行 join 即可。 30
JavaPairRDD<String, Tuple2<String, Row>> joinedRDD =
mappedRDD.join(expandedRDD);

```



解决方案八：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和二，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。

6.11 shuffle 调优

6.11.1 调优概述

大多数 Spark 作业的性能主要就是消耗在了 shuffle 环节，因为该环节包含了大量的磁盘 IO、序列化、网络数据传输等操作。因此，如果能让作业的性能更上一层楼，就有必要对 shuffle 过程进行调优。但是也必须提醒大家的是，影响一个 Spark 作业性能的因素，主要还是代码开发、资源参数以及数据倾斜，shuffle 调优只能在整个 Spark 的性能调优中占到

一小部分而已。因此大家务必把握住调优的基本原则，千万不要舍本逐末。下面我们就给大家详细讲解 **shuffle** 的原理，以及相关参数的说明，同时给出各个参数的调优建议。

6.11.2 ShuffleManager 发展概述

在 **Spark** 的源码中，负责 **shuffle** 过程的执行、计算和处理的组件主要就是 **ShuffleManager**，也即 **shuffle** 管理器。而随着 **Spark** 的版本的发展，**ShuffleManager** 也在不断迭代，变得越来越先进。

在 **Spark 1.2** 以前，默认的 **shuffle** 计算引擎是 **HashShuffleManager**。该 **ShuffleManager** 而 **HashShuffleManager** 有着一个非常严重的弊端，就是会产生大量的中间磁盘文件，进而由大量的磁盘 **IO** 操作影响了性能。

因此在 **Spark 1.2** 以后的版本中，默认的 **ShuffleManager** 改成了 **SortShuffleManager**。**SortShuffleManager** 相较于 **HashShuffleManager** 来说，有了一定的改进。主要就在于，每个 **Task** 在进行 **shuffle** 操作时，虽然也会产生较多的临时磁盘文件，但是最后会将所有的临时文件合并（**merge**）成一个磁盘文件，因此每个 **Task** 就只有一个磁盘文件。在下一个 **stage** 的 **shuffle read task** 拉取自己的数据时，只要根据索引读取每个磁盘文件中的部分数据即可。

下面我们详细分析一下 **HashShuffleManager** 和 **SortShuffleManager** 的原理。

6.11.3 HashShuffleManager 运行原理

未经优化的 HashShuffleManager

下图说明了未经优化的 **HashShuffleManager** 的原理。这里我们先明确一个假设前提：每个 **Executor** 只有 1 个 **CPU core**，也就是说，无论这个 **Executor** 上分配多少个 **task** 线程，同一时间都只能执行一个 **task** 线程。

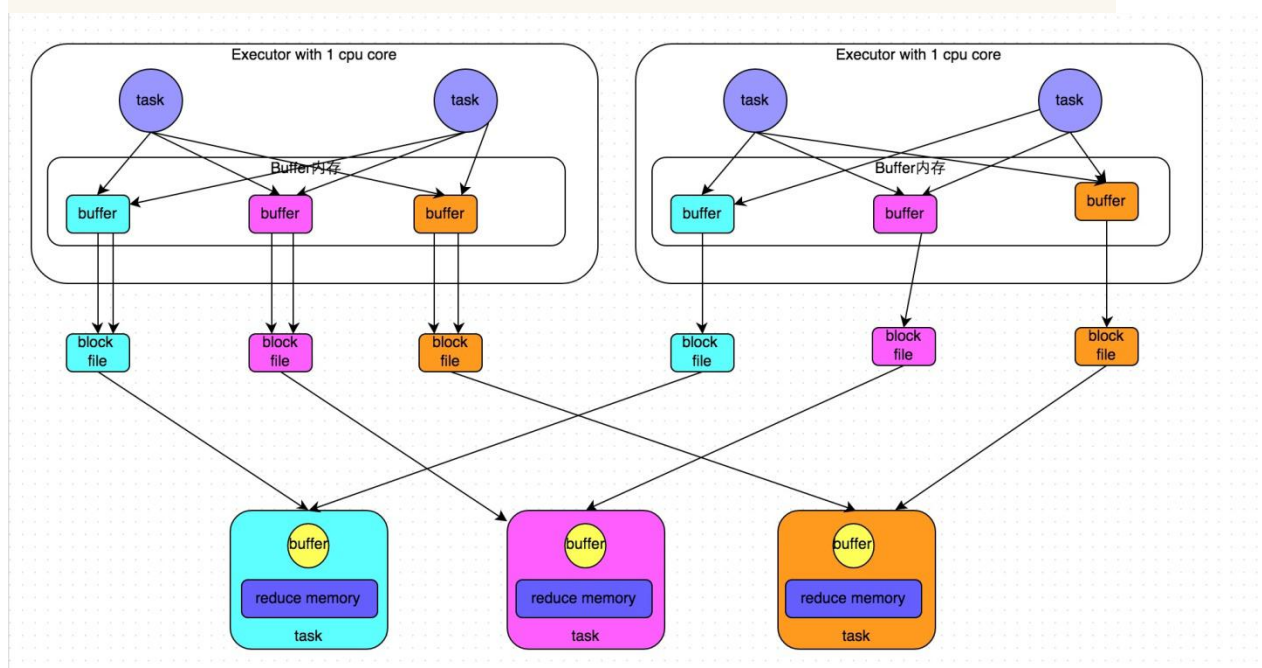
我们先从 **shuffle write** 开始说起。**shuffle write** 阶段，主要就是在一个 **stage** 结束计算之后，为了下一个 **stage** 可以执行 **shuffle** 类的算子（比如 **reduceByKey**），而将每个 **task** 处理的数据按 **key** 进行“分类”。所谓“分类”，就是对相同的 **key** 执行 **hash** 算法，从而将相同 **key** 都写入同一个磁盘文件中，而每一个磁盘文件都只属于下游 **stage** 的一个 **task**。在将数据写入磁盘之前，会先将数据写入内存缓冲中，当内存缓冲填满之后，才会溢写到磁盘文件中去。

那么每个执行 **shuffle write** 的 **task**，要为下一个 **stage** 创建多少个磁盘文件呢？很简单，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 100 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 5000

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。

接着我们来说 shuffle read。shuffle read，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，task 给下游 stage 的每个 task 都创建了一个磁盘文件，因此 shuffle read 的过程中，每个 task 只要从上游 stage 的所有 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拉取完，并得到最终的结果。



6.11.4 SortShuffleManager 运行原理

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。

普通运行机制

下图说明了普通的 SortShuffleManager 的原理。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。

bypass 运行机制

下图说明了 **bypass SortShuffleManager** 的原理。**bypass** 运行机制的触发条件如下：

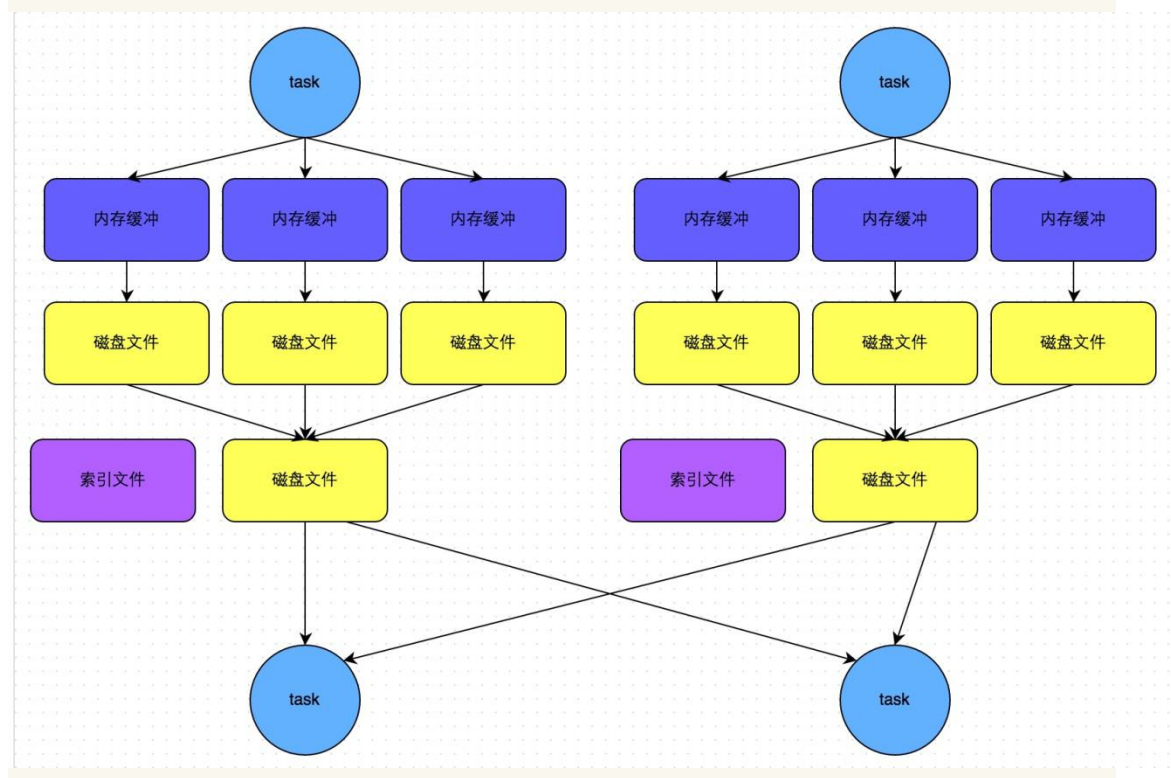
- **shuffle map task** 数量小于 **spark.shuffle.sort.bypassMergeThreshold** 参数的值。
- 不是聚合类的 **shuffle** 算子（比如 **reduceByKey**）。

此时 **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。



6.11.5 shuffle 相关参数调优

以下是 Shffule 过程中的一些主要参数，这里详细讲解了各个参数的功能、默认值以及基于实践经验给出的调优建议。

spark.shuffle.file.buffer

- 默认值：32k
- 参数说明：该参数用于设置 shuffle write task 的 BufferedOutputStream 的 buffer 缓冲大小。将数据写到磁盘文件之前，会先写入 buffer 缓冲中，待缓冲写满之后，才会溢写到磁盘。
- 调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如 64k），从而减少 shuffle write 过程中溢写磁盘文件的次数，也就可以减少磁盘 I/O 次数，进而提升性能。在实践中发现，合理调节该参数，性能会有 1%~5% 的提升。

spark.reducer.maxSizeInFlight

- 默认值：48m
- 参数说明：该参数用于设置 shuffle read task 的 buffer 缓冲大小，而这个 buffer 缓冲决定了每次能够拉取多少数据。
- 调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如 96m），从而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。在实践中发现，合理调节该参数，性能会有 1%~5% 的提升。

spark.shuffle.io.maxRetries

- 默认值：3
- 参数说明：shuffle read task 从 shuffle write task 所在节点拉取属于自己的数据时，如果因为网络异常导致拉取失败，是会自动进行重试的。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。
- 调优建议：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。

spark.shuffle.io.retryWait

- 默认值：5s
- 参数说明：具体解释同上，该参数代表了每次重试拉取数据的等待间隔，默认是 5s。
- 调优建议：建议加大间隔时长（比如 60s），以增加 shuffle 操作的稳定性。

spark.shuffle.memoryFraction

- 默认值：0.2
- 参数说明：该参数代表了 Executor 内存中，分配给 shuffle read task 进行聚合操作的内存比例，默认是 20%。
- 调优建议：在资源参数调优中讲解过这个参数。如果内存充足，而且很少使用持久化操作，建议调高这个比例，给 shuffle read 的聚合操作更多内存，以避免由于内存不足导致聚合过程中频繁读写磁盘。在实践中发现，合理调节该参数可以将性能提升 10% 左右。

spark.shuffle.manager

- 默认值: sort
- 参数说明: 该参数用于设置 ShuffleManager 的类型。Spark 1.5 以后, 有三个可选项: hash、sort 和 tungsten-sort。HashShuffleManager 是 Spark 1.2 以前的默认选项, 但是 Spark 1.2 以及之后的版本默认都是 SortShuffleManager 了。tungsten-sort 与 sort 类似, 但是使用了 tungsten 计划中的堆外内存管理机制, 内存使用效率更高。
- 调优建议: 由于 SortShuffleManager 默认会对数据进行排序, 因此如果你的业务逻辑中需要该排序机制的话, 则使用默认的 SortShuffleManager 就可以; 而如果你的业务逻辑不需要对数据进行排序, 那么建议参考后面的几个参数调优, 通过 bypass 机制或优化的 HashShuffleManager 来避免排序操作, 同时提供较好的磁盘读写性能。这里要注意的是, tungsten-sort 要慎用, 因为之前发现了一些相应的 bug。

spark.shuffle.sort.bypassMergeThreshold

- 默认值: 200
- 参数说明: 当 ShuffleManager 为 SortShuffleManager 时, 如果 shuffle read task 的数量小于这个阈值 (默认是 200), 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。
- 调优建议: 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。那么此时就会自动启用 bypass 机制, 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。

spark.shuffle consolidateFiles

- 默认值: false
- 参数说明: 如果使用 HashShuffleManager, 该参数有效。如果设置为 true, 那么就会开启 consolidate 机制, 会大幅度合并 shuffle write 的输出文件, 对于 shuffle read task 数量特别多的情况下, 这种方法可以极大地减少磁盘 IO 开销, 提升性能。
- 调优建议: 如果的确不需要 SortShuffleManager 的排序机制, 那么除了使用 bypass 机制, 还可以尝试将 spark.shuffle.manager 参数手动指定为 hash, 使用 HashShuffleManager, 同时开启 consolidate 机制。在实践中尝试过, 发现其性能比开启了 bypass 机制的 SortShuffleManager 要高出 10%~30%。

6.12 Spark 面试题汇总

!!!!Spark 面试题汇总大全!!!!:

<http://www.aboutyun.com/thread-24246-1-1.html>

1. spark 中的 RDD 是什么，有哪些特性

* RDD (Resilient Distributed Dataset) 叫做分布式数据集，是 Spark 中最基本的数据抽象，它代表一个不可变、可分区、里面的元素可并行计算的集合。

* 弹性表示

- * 1、RDD 中的数据可以存储在内存或者是磁盘
- * 2、RDD 中的分区是可以改变的

* 五大特性：

* A list of partitions

一个分区列表，RDD 中的数据都存在一个分区列表里面

* A function for computing each split

作用在每一个分区中的函数

* A list of dependencies on other RDDs

一个 RDD 依赖于其他多个 RDD，这个点很重要，RDD 的容错机制就是依据这个特性而来的

* Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)

可选的，针对于 kv 类型的 RDD 才具有这个特性，作用是决定了数据的来源以及数据处理后的去向

* Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)

可选项，数据本地性，数据位置最优

2. 概述一下 spark 中的常用算子区别 (map、mapPartitions、foreach、foreachPartition)

* map: 用于遍历 RDD,将函数 f 应用于每一个元素，返回新的 RDD(transformation 算子)。

* foreach:用于遍历 RDD,将函数 f 应用于每一个元素，无返回值(action 算子)。

* mapPartitions:用于遍历操作 RDD 中的每一个分区,返回生成一个新的 RDD(transformation 算子)。

* foreachPartition: 用于遍历操作 RDD 中的每一个分区。无返回值(action 算子)。

* 总结：一般使用 mapPartitions 或者 foreachPartition 算子比 map 和 foreach 更加高效，推荐使用。

3. 谈谈 spark 中的宽窄依赖

- * 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)。
- * 宽依赖：指的是多个子 RDD 的 Partition 会依赖同一个父 RDD 的 Partition
- * 窄依赖：指的是每一个父 RDD 的 Partition 最多被子 RDD 的一个 Partition 使用。

4. spark 中如何划分 stage

- * 1. Spark Application 中可以因为不同的 Action 触发众多的 job，一个 Application 中可以有很多的 job，每个 job 是由一个或者多个 Stage 构成的，后面的 Stage 依赖于前面的 Stage，也就是说只有前面依赖的 Stage 计算完毕后，后面的 Stage 才会运行。
- * 2. Stage 划分的依据就是宽依赖，何时产生宽依赖，例如 reduceByKey, groupByKey 的算子，会导致宽依赖的产生。
- * 3. 由 Action (例如 collect) 导致了 SparkContext.runJob 的执行，最终导致了 DAGScheduler 中的 submitJob 的执行，其核心是通过发送一个 case class JobSubmitted 对象给 eventProcessLoop。
eventProcessLoop 是 DAGSchedulerEventProcessLoop 的具体实例，而 DAGSchedulerEventProcessLoop 是 eventLoop 的子类，具体实现 EventLoop 的 onReceive 方法，onReceive 方法转过来回调 doOnReceive
- * 4. 在 doOnReceive 中通过模式匹配的方法把执行路由到
- * 5. 在 handleJobSubmitted 中首先创建 finalStage，创建 finalStage 时候会建立父 Stage 的依赖链条

* 总结：以来是从代码的逻辑层面上来展开说的，可以简单点说：写介绍什么是 RDD 中的宽窄依赖，然后在根据 DAG 有向无环图进行划分，从当前 job 的最后一个算子往前推，遇到宽依赖，那么当前在这个批次中的所有算子操作都划分成一个 stage，然后继续按照这种方式在继续往前推，如在遇到宽依赖，又划分成一个 stage，一直到最前面的一个算子。最后整个 job 会被划分成多个 stage，而 stage 之间又存在依赖关系，后面的 stage 依赖于前面的 stage。

5. spark-submit 的时候如何引入外部 jar 包

- * 在通过 spark-submit 提交任务时，可以通过添加配置参数来指定
 - * --driver-class-path 外部 jar 包
 - * --jars 外部 jar 包

6. spark 如何防止内存溢出

* driver 端的内存溢出

可以增大 driver 的内存参数: `spark.driver.memory` (default 1g)

这个参数用来设置 Driver 的内存。在 Spark 程序中, `SparkContext`, `DAGScheduler` 都是运行在 Driver 端的。对应 rdd 的 Stage 切分也是在 Driver 端运行, 需要调大 Driver 的内存。

map 过程产生大量对象导致内存溢出

* 可以在会产生大量对象的 map 操作之前调用 `repartition` 方法, 分区成更小的块传入 map。例如: `rdd.repartition(10000).map(x=>for(i <- 1 to 10000) yield i.toString)`。

面对这种问题注意, 不能使用 `rdd.coalesce` 方法, 这个方法只能减少分区, 不能增加分区, 不会有 shuffle 的过程。

数据不平衡导致内存溢出

* 数据不平衡除了有可能导致内存溢出外, 也有可能导致性能的问题, 解决方法和上面说的类似, 就是调用 `repartition` 重新分区。这里就不再赘述了。

shuffle 后内存溢出

* shuffle 内存溢出的情况可以说都是 shuffle 后, 单个文件过大导致的。通过 `spark.default.parallelism` 控制 (在 `spark-sql` 中用 `spark.sql.shuffle.partitions`), `spark.default.parallelism` 参数只对 `HashPartitioner` 有效,

standalone 模式下资源分配不均匀导致内存溢出

* 在 standalone 的模式下如果配置了 `--total-executor-cores` 和 `--executor-memory` 这两个参数, 同时配置 `--executor-cores` 或者 `spark.executor.cores` 参数, 确保 Executor 资源分配均匀。

7. spark 中 cache 和 persist 的区别

* `cache`: 缓存数据, 默认是缓存在内存中, 其本质还是调用 `persist`

* `persist`: 缓存数据, 有丰富的数据缓存策略。数据可以保存在内存也可以保存在磁盘中, 使用的时候指定对应的缓存级别就可以了。

8. 简要描述 Spark 分布式集群搭建的步骤

主要是引入了 zookeeper

9. spark 中的数据倾斜的现象、原因、后果

* (1)、数据倾斜的现象

* 多数 task 执行速度较快, 少数 task 执行时间非常长, 或者等待很长时间后提示你内存不足, 执行失败。

* (2)、数据倾斜的原因

- * 数据问题
 - * 1、key 本身分布不均衡（包括大量的 key 为空）
 - * 2、key 的设置不合理
- * spark 使用问题
 - * 1、shuffle 时的并发度不够
 - * 2、计算方式有误
- * (3)、数据倾斜的后果
 - * 1、spark 中的 stage 的执行时间受限于最后那个执行完成的 task,因此运行缓慢的任务会拖垮整个程序的运行速度（分布式程序运行的速度是由最慢的那个 task 决定的）。
 - * 2、过多的数据在同一个 task 中运行，将会把 executor 撑爆。

10. 如何解决 spark 中的数据倾斜问题

* 发现数据倾斜的时候，不要急于提高 executor 的资源，修改参数或是修改程序，首先要检查数据本身，是否存在异常数据。

* 1、数据问题造成的数据倾斜

* 找出异常的 key

* 如果任务长时间卡在最后最后 1 个(几个)任务，首先要对 key 进行抽样分析，判断是哪些 key 造成的。

选取 key，对数据进行抽样，统计出现的次数，根据出现次数大小排序取出前几个。

* 比 如 :

```
df.select("key").sample(false,0.1).(k=>(k,1)).reduceByKey(_+_).map(k=>(k._2,k._1)).sortByKey(false).take(10)
```

* 如果发现多数数据分布都较为平均，而个别数据比其他数据大上若干个数量级，则说明发生了数据倾斜。

* 经过分析，倾斜的数据主要有以下三种情况：

- * 1、null（空值）或是一些无意义的信息()之类的,大多是这个原因引起。
- * 2、无效数据，大量重复的测试数据或是对结果影响不大的有效数据。
- * 3、有效数据，业务导致的正常数据分布。

* 解决办法

* 第 1, 2 种情况，直接对数据进行过滤即可（因为该数据对当前业务不会产生影响）。

* 第 3 种情况则需要进行一些特殊操作，常见的有以下几种做法

* (1) 隔离执行，将异常的 key 过滤出来单独处理，最后与正常数据的处理结果进行 union 操作。

* (2) 对 key 先添加随机值，进行操作后，去掉随机值，再进行一次操作。

* (3) 使用 reduceByKey 代替 groupByKey(reduceByKey 用于对每个 key 对应的多个 value 进行 merge 操作,最重要的是它能够在本地先进行 merge 操作,并且 merge 操作可以通过函数自定义.)

* (4) 使用 map join。

* 案例

* 如果使用 reduceByKey 因为数据倾斜造成运行失败的问题。具体操作流程如

下:

- * (1) 将原始的 key 转化为 key + 随机值(例如 Random.nextInt)
- * (2) 对数据进行 reduceByKey(func)
- * (3) 将 key + 随机值 转成 key
- * (4) 再对数据进行 reduceByKey(func)

* 案例操作流程分析:

* 假设说有倾斜的 Key, 我们给所有的 Key 加上一个随机数, 然后进行 reduceByKey 操作; 此时同一个 Key 会有不同的随机数前缀, 在进行 reduceByKey 操作的时候原来的一个非常大的倾斜的 Key 就分而治之变成若干个更小的 Key, 不过此时结果和原来不一样, 怎么破? 进行 map 操作, 目的是把随机数前缀去掉, 然后再次进行 reduceByKey 操作。(当然, 如果你很无聊, 可以再次做随机数前缀), 这样我们就可以把原本倾斜的 Key 通过分而治之方案分散开来, 最后又进行了全局聚合

* 注意 1: 如果此时依旧存在问题, 建议筛选出倾斜的数据单独处理。最后将这份数据与正常的数据进行 union 即可。

* 注意 2: 单独处理异常数据时, 可以配合使用 Map Join 解决。

* 2、spark 使用不当造成的数据倾斜

* 提高 shuffle 并行度

* dataframe 和 sparkSql 可以设置 spark.sql.shuffle.partitions 参数控制 shuffle 的并发度, 默认为 200。

* rdd 操作可以设置 spark.default.parallelism 控制并发度, 默认参数由不同的 Cluster Manager 控制。

* 局限性: 只是让每个 task 执行更少的不同的 key。无法解决个别 key 特别大的情况造成的倾斜, 如果某些 key 的大小非常大, 即使一个 task 单独执行它, 也会受到数据倾斜的困扰。

* 使用 map join 代替 reduce join

* 在小表不是特别大(取决于你的 executor 大小)的情况下使用, 可以使程序避免 shuffle 的过程, 自然也就没有数据倾斜的困扰了。(详细见 <http://blog.csdn.net/lsshls/article/details/50834858>、<http://blog.csdn.net/lsshls/article/details/48694893>)

* 局限性: 因为是先将小数据发送到每个 executor 上, 所以数据量不能太大。

11. flume 整合 sparkStreaming 问题

* (1)、如何实现 sparkStreaming 读取 flume 中的数据

* 可以这样说:

* 前期经过技术调研, 查看官网相关资料, 发现 sparkStreaming 整合 flume 有 2 种模式, 一种是拉模式, 一种是推模式, 然后在简单的聊聊这 2 种模式的特点, 以及如何部署实现, 需要做哪些事情, 最后对比两种模式的特点, 选择那种模式更好。

* 推模式: Flume 将数据 Push 推给 Spark Streaming

* 拉模式: Spark Streaming 从 flume 中 Poll 拉取数据

* (2)、在实际开发的时候是如何保证数据不丢失的

* 可以这样说:

* flume 那边采用的 channel 是将数据落地到磁盘中, 保证数据源端安全性 (可以在补充一下, flume 在这里的 channel 可以设置为 memory 内存中, 提高数据接收处理的效率, 但是由于数据在内存中, 安全机制保证不了, 故选择 channel 为磁盘存储。整个流程运行有一点的延迟性)

* sparkStreaming 通过拉模式整合的时候, 使用了 FlumeUtils 这样一个类, 该类是需要依赖一个额外的 jar 包 (spark-streaming-flume_2.10)

* 要想保证数据不丢失, 数据的准确性, 可以在构建 StreamingContext 的时候, 利用 StreamingContext.getOrCreate (checkpoint, creatingFunc: () => StreamingContext) 来创建一个 StreamingContext, 使用 StreamingContext.getOrCreate 来创建 StreamingContext 对象, 传入的第一个参数是 checkpoint 的存放目录, 第二参数是生成 StreamingContext 对象的用户自定义函数。如果 checkpoint 的存放目录存在, 则从这个目录中生成 StreamingContext 对象; 如果不存在, 才会调用第二个函数来生成新的 StreamingContext 对象。在 creatingFunc 函数中, 除了生成一个新的 StreamingContext 操作, 还需要完成各种操作, 然后调用 ssc.checkpoint(checkpointDirectory)来初始化 checkpoint 功能, 最后再返回 StreamingContext 对象。

这样, 在 StreamingContext.getOrCreate 之后, 就可以直接调用 start()函数来启动 (或者是从中断点继续运行) 流式应用了。如果有其他在启动或继续运行都要做的工作, 可以在 start()调用前执行。

---- 流式计算中使用 checkpoint 的作用----

* 保存元数据, 包括流式应用的配置、流式没崩溃之前定义的各种操作、未完成所有操作的 batch。元数据被存储到容忍失败的存储系统上, 如 HDFS。这种 checkpoint 主要针对 driver 失败后的修复。

* 保存流式数据, 也是存储到容忍失败的存储系统上, 如 HDFS。这种 checkpoint 主要针对 window operation、有状态的操作。无论是 driver 失败了, 还是 worker 失败了, 这种 checkpoint 都能够快速恢复, 而不需要将很长的历史数据都重新计算一遍 (以便得到当前的状态)。

* 设置流式数据 checkpoint 的周期

* 对于一个需要做 checkpoint 的 DStream 结构, 可以通过调用 DStream.checkpoint(checkpointInterval)来设置 checkpoint 的周期, 经验上一般将这个 checkpoint 周期设置成 batch 周期的 5 至 10 倍。

* 使用 write ahead logs 功能

* 这是一个可选功能, 建议加上。这个功能将使得输入数据写入之前配置的 checkpoint 目录。这样有状态的数据可以从上一个 checkpoint 开始计算。开启的方法是把 spark.streaming.receiver.writeAheadLogs.enable 这个 property 设置为 true。另外, 由于输入 RDD 的默认 StorageLevel 是 MEMORY_AND_DISK_2, 即数据会在两台 worker 上做 replication。实际上, Spark Streaming 模式下, 任何从网络输入数据的 Receiver (如 kafka、flume、socket) 都会在这两台机器上做数据备份。如果开启了 write ahead logs 的功能, 建议把 StorageLevel 改成 MEMORY_AND_DISK_SER。修改的方法是, 在创建 RDD 时由参数传入。

* 使用以上的 checkpoint 机制, 确实可以保证数据 0 丢失。但是一个前提条件是, 数据发送端必须要有缓存功能, 这样才能保证在 spark 应用重启期间, 数据发送端不会因为 spark streaming 服务不可用而把数据丢弃。而 flume 具备这种特性, 同样 kafka 也具备。

* (3)Spark Streaming 的数据可靠性

* 有了 checkpoint 机制、write ahead log 机制、Receiver 缓存机器、可靠的 Receiver (即数据接收并备份成功后会发送 ack), 可以保证无论是 worker 失效还是 driver 失效, 都是数据 0 丢失。原因是: 如果没有 Receiver 服务的 worker 失效了, RDD 数据可以依赖血统来重新计算; 如果 Receiver 所在 worker 失败了, 由于 Receiver 是可靠的, 并有 write ahead log 机制, 则收到的数据可以保证不丢; 如果 driver 失败了, 可以从 checkpoint 中恢复数据重新构建。

12. kafka 整合 sparkStreaming 问题

* (1)、如何实现 sparkStreaming 读取 kafka 中的数据

* 可以这样说: 在 kafka0.10 版本之前有二种方式与 sparkStreaming 整合, 一种是基于 receiver, 一种是 direct, 然后分别阐述这 2 种方式分别是什么

* receiver: 是采用了 kafka 高级 api, 利用 receiver 接收器来接受 kafka topic 中的数据, 从 kafka 接收来的数据会存储在 spark 的 executor 中, 之后 spark streaming 提交的 job 会处理这些数据, kafka 中 topic 的偏移量是保存在 zk 中的。

* 基本使用: `val kafkaStream = KafkaUtils.createStream(streamingContext, [ZK quorum], [consumer group id], [per-topic number of Kafka partitions to consume])`

* 还有几个需要注意的点:

* 在 Receiver 的方式中, Spark 中的 partition 和 kafka 中的 partition 并不是相关的, 所以如果我们加大每个 topic 的 partition 数量, 仅仅是增加线程来处理由单一 Receiver 消费的主题。但是这并没有增加 Spark 在处理数据上的并行度。

* 对于不同的 Group 和 topic 我们可以使用多个 Receiver 创建不同的 Dstream 来并行接收数据, 之后可以利用 union 来统一成一个 Dstream。

* 在默认配置下, 这种方式可能会因为底层的失败而丢失数据。因为 receiver 一直在接收数据, 在其已经通知 zookeeper 数据接收完成但是还没有处理的时候, executor 突然挂掉(或是 driver 挂掉通知 executor 关闭), 缓存在其中的数据就会丢失。如果希望做到高可靠, 让数据零丢失, 如果我们启用了 Write Ahead Logs(`spark.streaming.receiver.writeAheadLog.enable=true`) 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。复制到文件系统如 HDFS, 那么 storage level 需要设置成 `StorageLevel.MEMORY_AND_DISK_SER`, 也就是 `KafkaUtils.createStream(..., StorageLevel.MEMORY_AND_DISK_SER)`

* direct: 在 spark1.3 之后, 引入了 Direct 方式。不同于 Receiver 的方式, Direct 方式没有 receiver 这一层, 其会周期性的获取 Kafka 中每个 topic 的每个 partition 中的最新 offsets, 之后根据设定的 `maxRatePerPartition` 来处理每个 batch。(设置 `spark.streaming.kafka.maxRatePerPartition=10000`。限制每秒钟从 topic 的每个 partition 最多消费的消息条数)。

* (2) 对比这 2 中方式的优缺点:

* 采用 receiver 方式: 这种方式可以保证数据不丢失, 但是无法保证数据只被处理一次, WAL 实现的是 At-least-once 语义 (至少被处理一次), 如果在写入到外部存储的数据还没有将 offset 更新到 zookeeper 就挂掉, 这些数据将会被反复消费。同时, 降低了程序的吞吐量。

* 采用 direct 方式: 相比 Receiver 模式而言能够确保机制更加健壮。区别于使用 Receiver

来被动接收数据, Direct 模式会周期性地主动查询 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:

* 1、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.

* 2、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.

* 3、一次且仅一次的事务机制

* 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. 不过需要自己完成将 offset 写入 zk 的过程, 在官方文档中都有相应介绍.

*简单代码实例:

```
* messages.foreachRDD(rdd=>{  
    val message = rdd.map(_._2)//对数据进行一些操作  
    message.map(method)//更新 zk 上的 offset (自己实现)  
    updateZKOffsets(rdd)  
})
```

* sparkStreaming 程序自己消费完成后, 自己主动去更新 zk 上面的偏移量. 也可以将 zk 中的偏移量保存在 mysql 或者 redis 数据库中, 下次重启的时候, 直接读取 mysql 或者 redis 中的偏移量, 获取到上次消费的偏移量, 接着读取数据。

Spark: 拥有 Hadoop MapReduce 所具有的优点; 但不同于 MapReduce 的是 Job 中间输出结果可以保存在内存中, 从而不再需要读写 HDFS, 因此 Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 MapReduce 的算法。

数据过于繁杂, 并且需要让计算通过迭代, 并在内存中, 极大地提高效率的场景

Storm: 一个分布式实时计算系统, Storm 是一个任务并行连续计算引擎。 Storm 本身并不典型在 Hadoop 集群上运行, 它使用 Apache ZooKeeper 的和自己的主/从工作进程, 协调拓扑, 主机和工作者状态, 保证信息的语义。无论如何, Storm 必定还是可以从 HDFS 文件消费或者从文件写入到 HDFS。

Hive：基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供简单的 sql 查询功能，可以将 sql 语句转换为 MapReduce 任务进行运行。
应用场景：十分适合数据仓库的统计分析。

Hbase:

应用场景： 数据量太大，以至于传统 RDBMS 无法胜任、
联机业务功能开发、
离线数据分析（数据仓库），

6.12.1 ----简答题 ---- 网上资料 ---

1. Spark master 使用 zookeeper 进行 HA 的，有哪些元数据保存在 Zookeeper？

答: spark 通过这个参数 `spark.deploy.zookeeper.dir` 指定 master 元数据在 zookeeper 中保存的位置，包括 Worker, Driver 和 Application 以及 Executors。standby 节点要从 zk 中，获得元数据信息，恢复集群运行状态，才能对外继续提供服务，作业提交资源申请等，在恢复前是不能接受请求的。另外，Master 切换需要注意 2 点：在 Master 切换的过程中，所有的已经在运行的程序皆正常运行！因为 Spark Application 在运行前就已经通过 Cluster Manager 获得了计算资源，所以在运行时 Job 本身的调度和处理和 Master 是没有任何关系的！在 Master 的切换过程中唯一的影响是不能提交新的 Job：一方面不能够提交新的应用程序给集群，因为只有 Active Master 才能接受新的程序的提交请求；另外一方面，已经运行的程序中也不能够因为 Action 操作触发新的 Job 的提交请求；

2. Spark master HA 主从切换过程不会影响集群已有的作业运行？

程序在运行之前，已经申请过资源了，driver 和 Executors 通讯，不需要和 master 进行通讯的。

3. Apache Spark 有哪些常见的稳定版本

答：常见的大的稳定版本有 Spark 1.3, Spark 1.6, Spark

4. driver 的功能是什么？

答：一个 Spark 作业运行时包括一个 Driver 进程，也是作业的主进程，具有 main 函数，并且有 SparkContext 的实例，是程序的人口点；2) 功能：负责向集群申请资源，向 master 注册信息，负责了作业的调度，负责作业的解析、生成 Stage 并调度 Task 到 Executor 上。包括 DAGScheduler, TaskScheduler。

5. spark 的有几种部署模式，每种模式特点？

1) 本地模式

Spark 不一定非要跑在 hadoop 集群，可以在本地，起多个线程的方式来指定。将 Spark 应用以多线程的方式直接运行在本地，一般都是为了方便调试，本地模式分三类

- local: 只启动一个 executor
- local[k]: 启动 k 个 executor
- local: 启动跟 cpu 数目相同的 executor

2) standalone 模式

分布式部署集群，自带完整的服务，资源管理和任务监控是 Spark 自己监控，这个模式也是其他模式的基础，

3) Spark on yarn 模式

分布式部署集群，资源和任务监控交给 yarn 管理，但是目前仅支持粗粒度资源分配方式，包含 cluster 和 client 运行模式，cluster 适合生产，driver 运行在集群子节点，具有容错功能，client 适合调试，driver 运行在客户端

4) Spark On Mesos 模式。官方推荐这种模式（当然，原因之一是血缘关系）。正是由于 Spark 开发之初就考虑到支持 Mesos，因此，目前而言，Spark 运行在 Mesos 上会比运行在 YARN 上更加灵活，更加自然。用户可选择两种调度模式之一运行自己的应用程序：

6. Spark 技术栈有哪些组件，每个组件都有什么功能，适合什么应用场景？

答：1) Sparkcore: 是其它组件的基础，spark 的内核，主要包含：有向循环图、RDD、Lingage、Cache、broadcast 等，并封装了底层通讯框架，是 Spark 的基础。

2) SparkStreaming 是一个对实时数据流进行高通量、容错处理的流式处理系统，可以对多种数据源（如 Kdfka、Flume、Twitter、Zero 和 TCP 套接字）进行类似 Map、Reduce 和 Join 等复杂操作，将流式计算分解成一系列短小的批处理作业。

3) Spark sql: Shark 是 SparkSQL 的前身，Spark SQL 的一个重要特点是其能够统一处理关系表和 RDD，使得开发人员可以轻松地使用 SQL 命令进行外部查询，同时进行更复杂的数据分析

4) BlinkDB : 是一个用于在海量数据上运行交互式 SQL 查询的大规模并行查询引擎，它允许用户通过权衡数据精度来提升查询响应时间，其数据的精度被控制在允许的误差范围内。

5) MLBase 是 Spark 生态圈的一部分专注于机器学习，让机器学习的门槛更低，让一些可能并不了解机器学习的用户也能方便地使用 MLbase。MLBase 分为四部分：MLlib、MLI、ML

Optimizer 和 MLRuntime。

6) GraphX 是 Spark 中用于图和图并行计算

7. Spark 中 Work 的主要工作是什么？

答：主要功能：管理当前节点内存，CPU 的使用状况，接收 master 分配过来的资源指令，通过 ExecutorRunner 启动程序分配任务，worker 就类似于包工头，管理分配新进程，做计算的服务，相当于 process 服务。需要注意的是：1) worker 会不会汇报当前信息给 master，worker 心跳给 master 主要只有 workid，它不会发送资源信息以心跳的方式给 master，master 分配的时候就知道 work，只有出现故障的时候才会发送资源。2) worker 不会运行代码，具体运行的是 Executor 是可以运行具体 application 写的业务逻辑代码，操作代码的节点，它不会运行程序的代码的。

8. Spark 为什么比 mapreduce 快？

答：1) 基于内存计算，减少低效的磁盘交互；2) 高效的调度算法，基于 DAG；3)容错机制 Lineage，精华部分就是 DAG 和 Lingae

9. 简单说一下 hadoop 和 spark 的 shuffle 相同和差异？

答：1) 从 high-level 的角度来看，两者并没有大的差别。都是将 mapper (Spark 里是 ShuffleMapTask) 的输出进行 partition，不同的 partition 送到不同的 reducer (Spark 里 reducer 可能是下一个 stage 里的 ShuffleMapTask，也可能是 ResultTask)。Reducer 以内存作缓冲区，边 shuffle 边 aggregate 数据，等到数据 aggregate 好以后进行 reduce() (Spark 里可能是后续的一系列操作)。

2) 从 low-level 的角度来看，两者差别不小。Hadoop MapReduce 是 sort-based，进入 combine() 和 reduce() 的 records 必须先 sort。这样的好处在于 combine/reduce() 可以处理大规模的数据，因为其输入数据可以通过外排得到 (mapper 对每段数据先做排序，reducer 的 shuffle 对排好序的每段数据做归并)。目前的 Spark 默认选择的是 hash-based，通常使用 HashMap 来对 shuffle 来的数据进行 aggregate，不会对数据进行提前排序。如果用户需要经过排序的数据，那么需要自己调用类似 sortByKey() 的操作；如果你是 Spark 1.1 的用户，可以将 spark.shuffle.manager 设置为 sort，则会对数据进行排序。在 Spark 1.2 中，sort 将作为默认的 Shuffle 实现。

3) 从实现角度来看，两者也有不少差别。Hadoop MapReduce 将处理流程划分出明显的几个阶段：map(), spill, merge, shuffle, sort, reduce() 等。每个阶段各司其职，可以按照程式的编程思想来逐一实现每个阶段的功能。在 Spark 中，没有这样功能明确的阶段，只有不同的 stage 和一系列的 transformation()，所以 spill, merge, aggregate 等操作需要蕴含在 transformation() 中。

如果我们将 map 端划分数据、持久化数据的过程称为 shuffle write，而将 reducer 读入数据、aggregate 数据的过程称为 shuffle read。那么在 Spark 中，问题就变为怎么在 job 的逻辑或者物理执行图中加入 shuffle write 和 shuffle read 的处理逻辑？以及两个处理逻辑应该怎么高效实现？

Shuffle write 由于不要求数据有序, shuffle write 的任务很简单: 将数据 partition 好, 并持久化。之所以要持久化, 一方面是要减少内存存储空间压力, 另一方面也是为了 fault-tolerance。

10. Mapreduce 和 Spark 的都是并行计算, 那么他们有什么相同和区别

答: 两者都是用 mr 模型来进行并行计算:

1)hadoop 的一个作业称为 job, job 里面分为 map task 和 reduce task, 每个 task 都是在自己的进程中运行的, 当 task 结束时, 进程也会结束。

2)spark 用户提交的任务成为 application, 一个 application 对应一个 sparkcontext, app 中存在多个 job, 每触发一次 action 操作就会产生一个 job。这些 job 可以并行或串行执行, 每个 job 中有多个 stage, stage 是 shuffle 过程中 DAGScheduler 通过 RDD 之间的依赖关系划分 job 而来的, 每个 stage 里面有多个 task, 组成 taskset 有 TaskScheduler 分发到各个 executor 中执行, executor 的生命周期是和 app 一样的, 即使没有 job 运行也是存在的, 所以 task 可以快速启动读取内存进行计算。

3)hadoop 的 job 只有 map 和 reduce 操作, 表达能力比较欠缺而且在 mr 过程中会重复的读写 hdfs, 造成大量的 io 操作, 多个 job 需要自己管理关系。

spark 的迭代计算都是在内存中进行的, API 中提供了大量的 RDD 操作如 join, groupby 等, 而且通过 DAG 图可以实现良好的容错。

11. RDD 机制?

答: rdd 分布式弹性数据集, 简单的理解成一种数据结构, 是 spark 框架上的通用货币。

所有算子都是基于 rdd 来执行的, 不同的场景会有不同的 rdd 实现类, 但是都可以进行互相转换。

rdd 执行过程中会形成 dag 图, 然后形成 lineage 保证容错性等。从物理的角度来看 rdd 存储的是 block 和 node 之间的映射。

12. spark 工作机制?

答: 用户在 client 端提交作业后, 会由 Driver 运行 main 方法并创建 sparkcontext 上下文。执行 add 算子形成 dag 图输入 dagscheduler, 按照 add 之间的依赖关系划分 stage 输入 task scheduler。task scheduler 会将 stage 划分为 taskset 分发到各个节点 executor 中执行。

13. spark 的优化怎么做?

答: spark 调优比较复杂, 但是大体可以分为三个方面来进行, 1) 平台层面的调优: 防止不必要的 jar 包分发, 提高数据的本地性, 选择高效的存储格式如 parquet, 2) 应用程序层面的调优: 过滤操作符的优化降低过多小任务, 降低单条记录的资源开销, 处理数据倾斜,

复用 RDD 进行缓存，作业并行化执行等等，3) JVM 层面的调优：设置合适的资源量，设置合理的 JVM，启用高效的序列化方法如 kyro，增大 off head 内存等等

14. spark-submit 的时候如何引入外部 jar 包

方法一：spark-submit -jars

根据 spark 官网，在提交任务的时候指定-jars，用逗号分开。这样做的缺点是每次都要指定 jar 包，如果 jar 包少的话可以这么做，但是如果多的话会很麻烦。

命令：spark-submit --master yarn-client --jars ***.jar,***.jar

方法二：extraClassPath

提交时在 spark-default 中设定参数，将所有需要的 jar 包考到一个文件里，然后在参数中指定该目录就可以了，较上一个方便很多：

spark.executor.extraClassPath=/home/hadoop/wzq_workspace/lib/*

spark.driver.extraClassPath=/home/hadoop/wzq_workspace/lib/*

需要注意的是,你要在所有可能运行 spark 任务的机器上保证该目录存在，并且将 jar 包考到所有机器上。这样做的好处是提交代码的时候不用再写一长串 jar 了，缺点是要把所有的 jar 包都拷一遍。

15. cache 和 persist 的区别

答：1) cache 和 persist 都是用于将一个 RDD 进行缓存的，这样在之后使用的过程中就不需要重新计算了，可以大大节省程序运行时间；

2) cache 只有一个默认的缓存级别 MEMORY_ONLY，cache 调用了 persist，而 persist 可以根据情况设置其它的缓存级别；

3) executor 执行的时候，默认 60%做 cache，40%做 task 操作，persist 最根本的函数，最底层的函数

16. Spark 使用 parquet 文件存储格式能带来哪些好处？

1) 如果说 HDFS 是大数据时代分布式文件系统首选标准，那么 parquet 则是整个大数据时代文件存储格式实时首选标准

2) 速度更快：从使用 spark sql 操作普通文件 CSV 和 parquet 文件速度对比上看，绝大多数情况会比使用 csv 等普通文件速度提升 10 倍左右，在一些普通文件系统无法在 spark 上成功运行的情况下，使用 parquet 很多时候可以成功运行

3) parquet 的压缩技术非常稳定出色，在 spark sql 中对压缩技术的处理可能无法正常的完成工作例如会导致 losttask，lost executor 但是此时如果使用 parquet 就可以正常的完成

4) 极大的减少磁盘 I/O,通常情况下能够减少 75%的存储空间，由此可以极大的减少 spark sql 处理数据的时候的数据输入内容，尤其是在 spark1.6x 中有个下推过滤器在一些情况下可以极大的减少磁盘的 IO 和内存的占用，（下推过滤器）

5) spark 1.6x parquet 方式极大的提升了扫描的吞吐量，极大提高了数据的查找速度 spark1.6 和 spark1.5x 相比而言，提升了大约 1 倍的速度，在 spark1.6X 中，操作 parquet 时候 cpu 也进行了极大的优化，有效的降低了 cpu

6) 采用 parquet 可以极大的优化 spark 的调度和执行。我们测试 spark 如果用 parquet 可以有效的减少 stage 的执行消耗，同时可以优化执行路径

17. Spark 如何自定义 partitioner 分区器？

答：1) spark 默认实现了 HashPartitioner 和 RangePartitioner 两种分区策略，我们也可以自己扩展分区策略，自定义分区器的时候继承 org.apache.spark.Partitioner 类，实现类中的三个方法

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

def getPartition(key: Any): Int: 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；

equals(): 这个是 Java 标准的判断相等的函数，之所以要求用户实现这个函数是因为 Spark 内部会比较两个 RDD 的分区是否一样。

6.12.2 -----Spark on Yarn 面试篇

1.描述 Yarn 执行一个任务的过程？

1) 客户端 client 向 ResourceManager 提交 Application，ResourceManager 接受 Application 并根据集群资源状况选取一个 node 来启动 Application 的任务调度器 driver (ApplicationMaster)

2) ResourceManager 找到那个 node，命令其该 node 上的 nodeManager 来启动一个新的 JVM 进程运行程序的 driver (ApplicationMaster) 部分，driver (ApplicationMaster) 启动时会首先向 ResourceManager 注册，说明由自己来负责当前程序的运行

3) driver (ApplicationMaster) 开始下载相关 jar 包等各种资源，基于下载的 jar 等信息决定向 ResourceManager 申请具体的资源内容。

4) ResourceManager 接受到 driver (ApplicationMaster) 提出的申请后，会最大化的满足资源分配请求，并发送资源的元数据信息给 driver (ApplicationMaster)；

5) driver (ApplicationMaster) 收到发过来的资源元数据信息后会根据元数据信息发指令给具体

机器上的 NodeManager，让其启动具体的 container。

6) NodeManager 收到 driver 发来的指令，启动 container，container 启动后必须向 driver (ApplicationMaster) 注册。

7) driver (ApplicationMaster) 收到 container 的注册，开始进行任务的调度和计算，直到任务完成。

补充：如果 ResourceManager 第一次没有能够满足 driver (ApplicationMaster) 的资源请求，后续发现有空闲的资源，会主动向 driver (ApplicationMaster) 发送可用资源的元数据信息

以提供更多的资源用于当前程序的运行。

2.Yarn 中的 container 是由谁负责销毁的，在 Hadoop Mapreduce 中 container 可以复用么？

答：ApplicationMaster 负责销毁，在 Hadoop Mapreduce 不可以复用，在 spark on yarn 程序 container 可以复用

3. 提交任务时，如何指定 Spark Application 的运行模式？

- 1) cluster 模式：./spark-submit --class xx.xx.xx --master yarn --deploy-mode cluster xx.jar
- 2) client 模式：./spark-submit --class xx.xx.xx --master yarn --deploy-mode client xx.jar

4. 不启动 Spark 集群 Master 和 work 服务，可不可以运行 Spark 程序？

答：可以，只要资源管理器第三方管理就可以，如由 yarn 管理，spark 集群不启动也可以使用 spark；spark 集群启动的是 work 和 master，这个其实就是资源管理框架，yarn 中的 resourceManager 相当于 master，NodeManager 相当于 worker，做计算是 Executor，和 spark 集群的 work 和 manager 可以没关系，归根接底还是 JVM 的运行，只要所在的 JVM 上安装了 spark 就可以。

5. Spark 中的 4040 端口由什么功能？

答：收集 Spark 作业运行的信息

6. spark on yarn Cluster 模式下，ApplicationMaster 和 driver 是在同一个进程么？

答：是，driver 位于 ApplicationMaster 进程中。该进程负责申请资源，还负责监控程序、资源的动态情况。

7. 如何使用命令查看 application 运行的日志信息

答：yarn logs -applicationId <app ID>

8. Spark on Yarn 模式有哪些优点？

- 1)与其他计算框架共享集群资源(eg.Spark 框架与 MapReduce 框架同时运行, 如果不用 Yarn

进行资源分配, MapReduce 分到的内存资源会很少, 效率低下); 资源按需分配, 进而提高集群资源利用等。

2)相较于 Spark 自带的 Standalone 模式, Yarn 的资源分配更加细致

3)Application 部署简化, 例如 Spark, Storm 等多种框架的应用由客户端提交后, 由 Yarn 负责资源的管理和调度, 利用 Container 作为资源隔离的单位, 以它为单位去使用内存,cpu 等。

4)Yarn 通过队列的方式, 管理同时运行在 Yarn 集群中的多个服务, 可根据不同类型的应用程序负载情况, 调整对应的资源使用量, 实现资源弹性管理。

9. 谈谈你对 container 的理解?

1) Container 作为资源分配和调度的基本单位, 其中封装了的资源如内存, CPU, 磁盘, 网络带宽等。目前 yarn 仅仅封装内存和 CPU

2)Container 由 ApplicationMaster 向 ResourceManager 申请的, 由 ResourceManager 中的资源调度器异步分配给 ApplicationMaster

3) Container 的运行是由 ApplicationMaster 向资源所在的 NodeManager 发起的, Container 运行时需提供内部执行的任务命令。

10. 运行在 yarn 中 Application 有几种类型的 container?

1) 运行 ApplicationMaster 的 Container: 这是由 ResourceManager (向内部的资源调度器) 申请和启动的, 用户提交应用程序时, 可指定唯一的 ApplicationMaster 所需的资源;

2) 运行各类任务的 Container: 这是由 ApplicationMaster 向 ResourceManager 申请的, 并由 ApplicationMaster 与 NodeManager 通信以启动之。

11. Spark on Yarn 架构是怎么样? (要会画哦, 这个图)

12. Executor 启动时, 资源通过哪几个参数指定?

1)num-executors 是 executor 的数量

2)executor-memory 是每个 executor 使用的内存

3)executor-cores 是每个 executor 分配的 CPU

13. 一个 task 的 map 数量由谁来决定?

一般情况下, 在输入源是文件的时候, 一个 task 的 map 数量由 splitSize 来决定的, 那么 splitSize 是由以下几个来决定的

$goalSize = totalSize / mapred.map.tasks$

$inSize = \max \{mapred.min.split.size, minSplitSize\}$

$splitSize = \max (minSize, \min(goalSize, dfs.block.size))$

一个 task 的 reduce 数量, 由 partition 决定。

14. reduce 后输出的数据量有多大？

并不是想知道确切的数据量有多大这个，而是想问你，MR 的执行机制，开发完程序，有没有认真评估程序运行效率

1) 用于处理 redcue 任务的资源情况，如果是 MRV1 的话，分了多少资源给 map，多少个 reduce

如果是 MRV2 的话，可以提一下，集群有分了多少内存、CPU 给 yarn 做计算。

2) 结合实际应用场景回答，输入数据有多大，大约多少条记录，做了哪些逻辑操作，输出的时候有多少条记录，执行了多久，reduce 执行时候的数据有没有倾斜等

3) 再提一下，针对 mapReduce 做了哪几点优化，速度提升了多久，列举 1,2 个优化点就可以

15. 你的项目提交到 job 的时候数据量有多大？

答：1) 回答出数据是什么格式，有没有采用什么压缩，采用了压缩的话，压缩比大概是多少；2) 文件大概多大：大概起了多少个 map，起了多少个 reduce，map 阶段读取了多少数据，reduce 阶段读取了多少数据，程序大约执行了多久，3) 集群什么规模，集群有多少节点，多少内存，多少 CPU 核数等。把这些点回答进去，而不是给个数字了事。

16. 你们提交的 job 任务大概有多少个？这些 job 执行完大概用多长时间？

还是考察你开发完程序有没有认真观察过程序的运行，有没有评估程序运行的效率

17. 你们业务数据量多大？有多少行数据？

这个也是看你们有没有实际的经验,对于没有实战的同学，请把回答的侧重点放在 MR 的运行机制上面，

MR 运行效率方面，以及如何优化 MR 程序（看别人的优化 demo，然后在虚拟机上拿 demo 做一下测试）。

18. 如何杀死一个正在运行的 job

杀死一个 job

MRV1: Hadoop job kill jobid

YARN: yarn application -kill applicationId

19. 列出你所知道的调度器，说明其工作原理

- a) Fifo scheduler 默认的调度器 先进先出
- b) Capacity scheduler 计算能力调度器 选择占用内存小、优先级高的
- c) Fair scheduler 公平调度器 所有 job 占用相同资源

6.12.3 -----spark sql 面试篇

Join 常见分类以及基本实现机制

当前 SparkSQL 支持三种 Join 算法 – shuffle hash join、broadcast hash join 以及 sort merge join。其中前两者归根到底都属于 hash join，只不过在 hash join 之前需要先 shuffle 还是先 broadcast

<http://hbasefly.com/2017/03/19/sparksql-basic-join/>

6.12.4 -----选择题 ---

1. Spark 的四大组件下面哪个不是 (D)

- A. Spark Streaming
- B. Mlib
- C. Graphx
- D. Spark R

2. 下面哪个端口不是 spark 自带服务的端口 (C)

- A. 8080
- B. 4040
- C. 8090
- D. 18080

备注：8080：spark 集群 web ui 端口，4040：sparkjob 监控端口，18080：jobhistory 端口

3. spark 1.4 版本的最大变化 (B)

- A. spark sql Release 版本
- B. 引入 Spark R
- C. DataFrame
- D. 支持动态资源分配

4. Spark Job 默认的调度模式 (A)

- A. FIFO
- B. FAIR
- C. 无
- D. 运行时指定

5. 哪个不是本地模式运行的条件 (D)

- A. spark.localExecution.enabled=true
- B. 显式指定本地运行
- C. finalStage 无父 Stage
- D. partition 默认值

- 6.下面哪个不是 RDD 的特点 (C)
A. 可分区 B 可序列化 C 可修改 D 可持久化
7. 关于广播变量, 下面哪个是错误的 (D)
A 任何函数调用 B 是只读的
C 存储在各个节点 D 存储在磁盘或 HDFS
8. 关于累加器, 下面哪个是错误的 (D)
A 支持加法 B 支持数值类型
C 可并行 D 不支持自定义类型
- 9.Spark 支持的分布式部署方式中哪个是错误的 (D)
A standalone B spark on mesos
C spark on YARN D Spark on local
- 10.Stage 的 Task 的数量由什么决定 (A)
A Partition B Job C Stage D TaskScheduler
- 11.下面哪个操作是窄依赖 (B)
A join B filter
C group D sort
- 12.下面哪个操作肯定是宽依赖 (C)
A map B flatMap
C reduceByKey D sample
- 13.spark 的 master 和 worker 通过什么方式进行通信的? (D)
A http B nio C netty D Akka
- 14 默认的存储级别 (A)
A MEMORY_ONLY B MEMORY_ONLY_SER
C MEMORY_AND_DISK D MEMORY_AND_DISK_SER
- 15 spark.deploy.recoveryMode 不支持那种 (D)
A.ZooKeeper B. FileSystem
D NONE D Hadoop
- 16.下列哪个不是 RDD 的缓存方法 (C)
A persist() B Cache()
C Memory()
- 17.Task 运行在下来哪里个选项中 Executor 上的工作单元 (C)
A Driver program B. spark master
C.worker node D Cluster manager

18.hive 的元数据存储在 derby 和 MySQL 中有什么区别 (B)

A.没区别 B.多会话

C.支持网络环境 D 数据库的区别

19.DataFrame 和 RDD 最大的区别 (B)

A.科学统计支持 B.多了 schema

C.存储方式不一样 D.外部数据源支持

20.Master 的 ElectedLeader 事件后做了哪些操作 (D)

A. 通知 driver B.通知 worker

C.注册 application D.直接 ALIVE

6.12.5 补充资料: (spark 集群 standalone + spark on yarn)

!!推荐!! [Spark 入门实战系列--4.Spark 运行架构](#)

Spark 在不同集群中的运行架构

Spark 注重建立良好的生态系统, 它不仅支持多种外部文件存储系统, 提供了多种多样的集群运行模式。部署在单台机器上时, 既可以用本地 (Local) 模式运行, 也可以使用伪分布式模式来运行; 当以分布式集群部署的时候, 可以根据自己集群的实际情况选择 Standalone 模式 (Spark 自带的模式)、YARN-Client 模式或者 YARN-Cluster 模式。Spark 的各种运行模式虽然在启动方式、运行位置、调度策略上各有不同, 但它们的目的一致, 就是在合适的位置安全可靠的根据用户的配置和 Job 的需要运行和管理 Task。

Spark on Standalone 运行过程

Standalone 模式是 Spark 实现的资源调度框架, 其主要的节点有 Client 节点、Master 节点和 Worker 节点。其中 Driver 既可以运行在 Master 节点上, 也可以运行在本地 Client 端。当用 spark-shell 交互式工具提交 Spark 的 Job 时, Driver 在 Master 节点上运行; 当使用 spark-submit 工具提交 Job 或者在 Eclipse、IDEA 等开发平台上使用 "new SparkConf.setManager("spark://master:7077")" 方式运行 Spark 任务时, Driver 是运行在本地 Client 端上的。

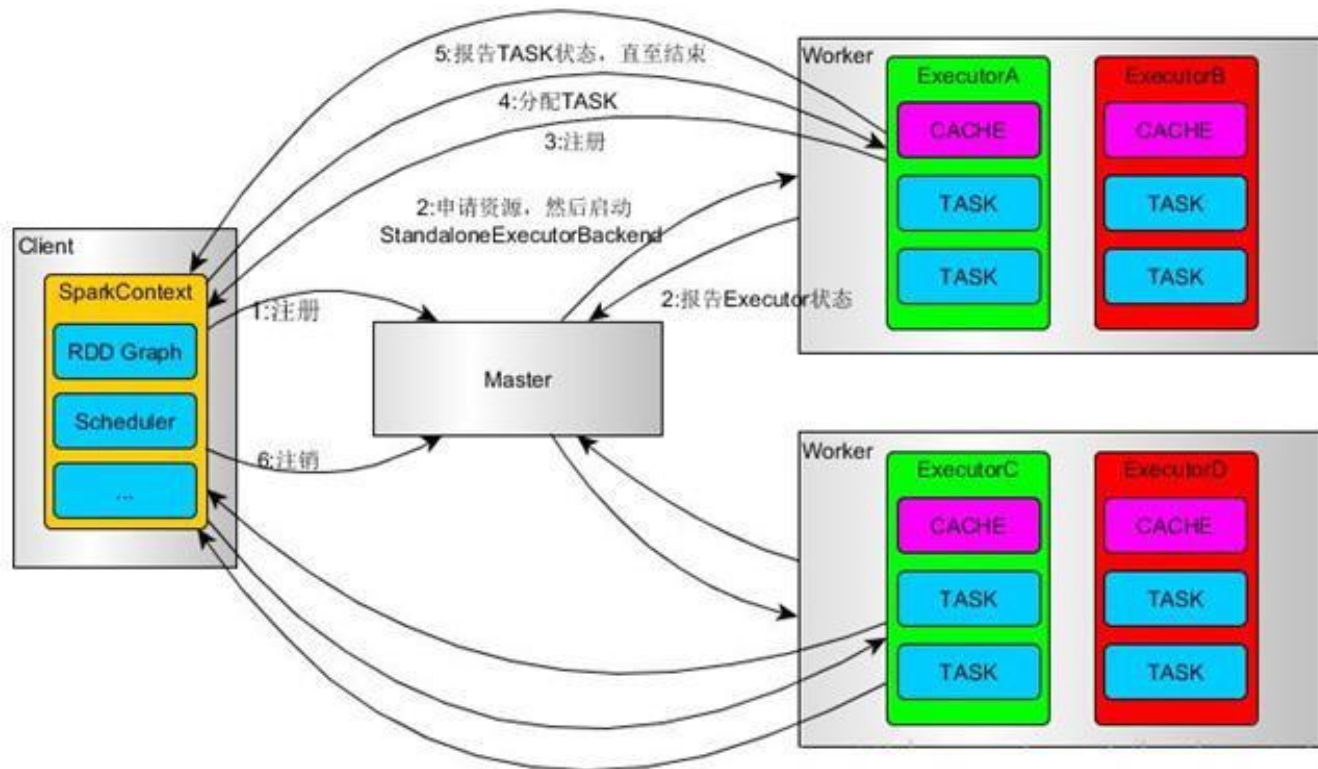
其运行过程如下:

- 1.SparkContext 连接到 Master, 向 Master 注册并申请资源 (CPU Core 和 Memory);
- 2.Master 根据 SparkContext 的资源申请要求和 Worker 心跳周期内报告的信息决定在哪个 Worker 上分配资源, 然后在该 Worker 上获取资源, 然后启动 StandaloneExecutorBackend;
- 3.StandaloneExecutorBackend 向 SparkContext 注册;
- 4.SparkContext 将 Application 代码发送给 StandaloneExecutorBackend; 并且 SparkContext

解析 Application 代码, 构建 DAG 图, 并提交给 DAG Scheduler 分解成 Stage (当碰到 Action 操作时, 就会催生 Job; 每个 Job 中含有 1 个或多个 Stage, Stage 一般在获取外部数据和 shuffle 之前产生), 然后以 Stage (或者称为 TaskSet) 提交给 Task Scheduler, Task Scheduler 负责将 Task 分配到相应的 Worker, 最后提交给 StandaloneExecutorBackend 执行;

5.StandaloneExecutorBackend 会建立 Executor 线程池, 开始执行 Task, 并向 SparkContext 报告, 直至 Task 完成。

6.所有 Task 完成后, SparkContext 向 Master 注销, 释放资源。



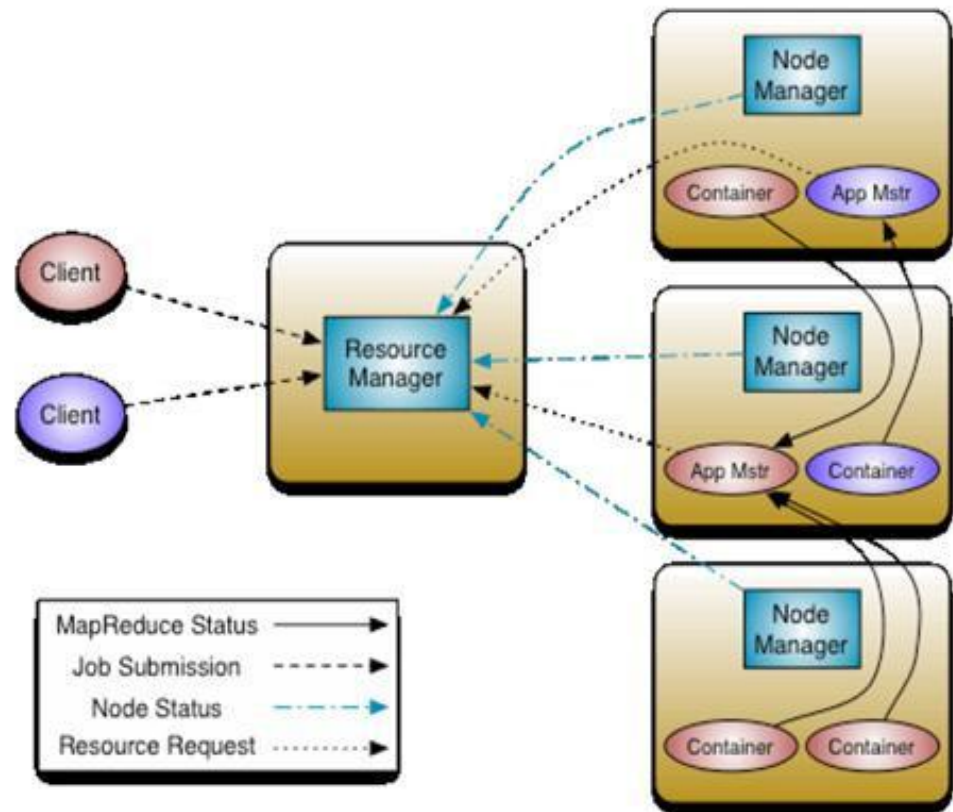
Spark on YARN 运行过程

YARN 是一种统一资源管理机制, 在其上面可以运行多套计算框架。目前的大数据技术世界, 大多数公司除了使用 Spark 来进行数据计算, 由于历史原因或者单方面业务处理的性能考虑而使用着其他的计算框架, 比如 MapReduce、Storm 等计算框架。Spark 基于此种情况开发了 Spark on YARN 的运行模式, 由于借助了 YARN 良好的弹性资源管理机制, 不仅部署 Application 更加方便, 而且用户在 YARN 集群中运行的服务和 Application 的资源也完全隔离, 更具实践应用价值的是 YARN 可以通过队列的方式, 管理同时运行在集群中的多个服务。Spark on YARN 模式根据 Driver 在集群中的位置分为两种模式: 一种是 YARN-Client 模式, 另一种是 YARN-Cluster (或称为 YARN-Standalone 模式)。

2.2.1 YARN 框架流程

任何框架与 YARN 的结合, 都必须遵循 YARN 的开发模式。在分析 Spark on YARN 的实现细节之前, 有必要先分析一下 YARN 框架的一些基本原理。

Yarn 框架的基本运行流程图为:

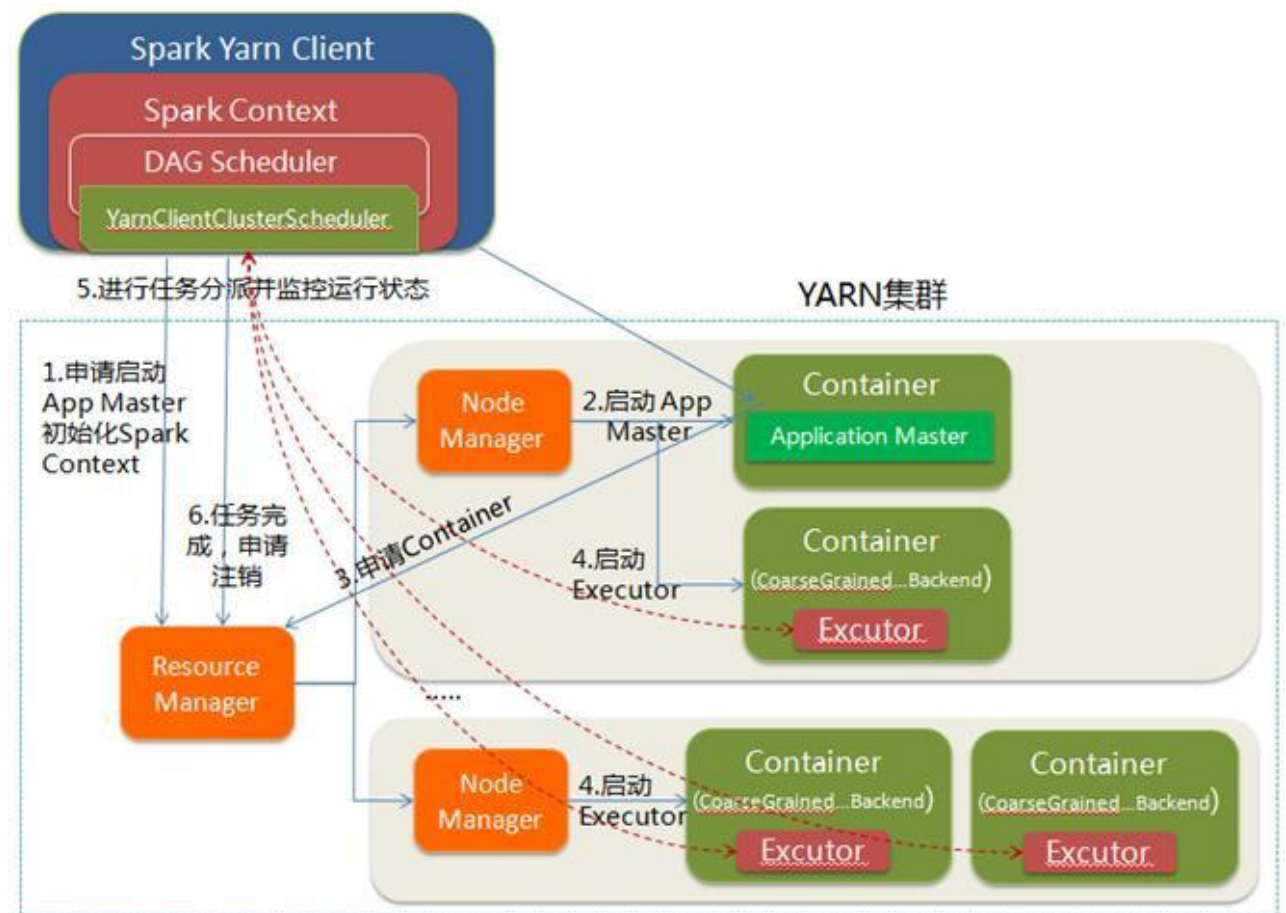


其中，ResourceManager 负责将集群的资源分配给各个应用使用，而资源分配和调度的基本单位是 Container，其中封装了机器资源，如内存、CPU、磁盘和网络等，每个任务会被分配一个 Container，该任务只能在该 Container 中执行，并使用该 Container 封装的资源。NodeManager 是一个个的计算节点，主要负责启动 Application 所需的 Container，监控资源（内存、CPU、磁盘和网络等）的使用情况并将之汇报给 ResourceManager。ResourceManager 与 NodeManagers 共同组成整个数据计算框架，ApplicationMaster 与具体的 Application 相关，主要负责同 ResourceManager 协商以获取合适的 Container，并跟踪这些 Container 的状态和监控其进度。

2.2.2 YARN-Client

Yarn-Client 模式中，Driver 在客户端本地运行，这种模式可以使得 Spark Application 和客户端进行交互，因为 Driver 在客户端，所以可以通过 webUI 访问 Driver 的状态，默认是 <http://hadoop1:4040> 访问，而 YARN 通过 <http://hadoop1:8088> 访问。

YARN-client 的工作流程分为以下几个步骤：

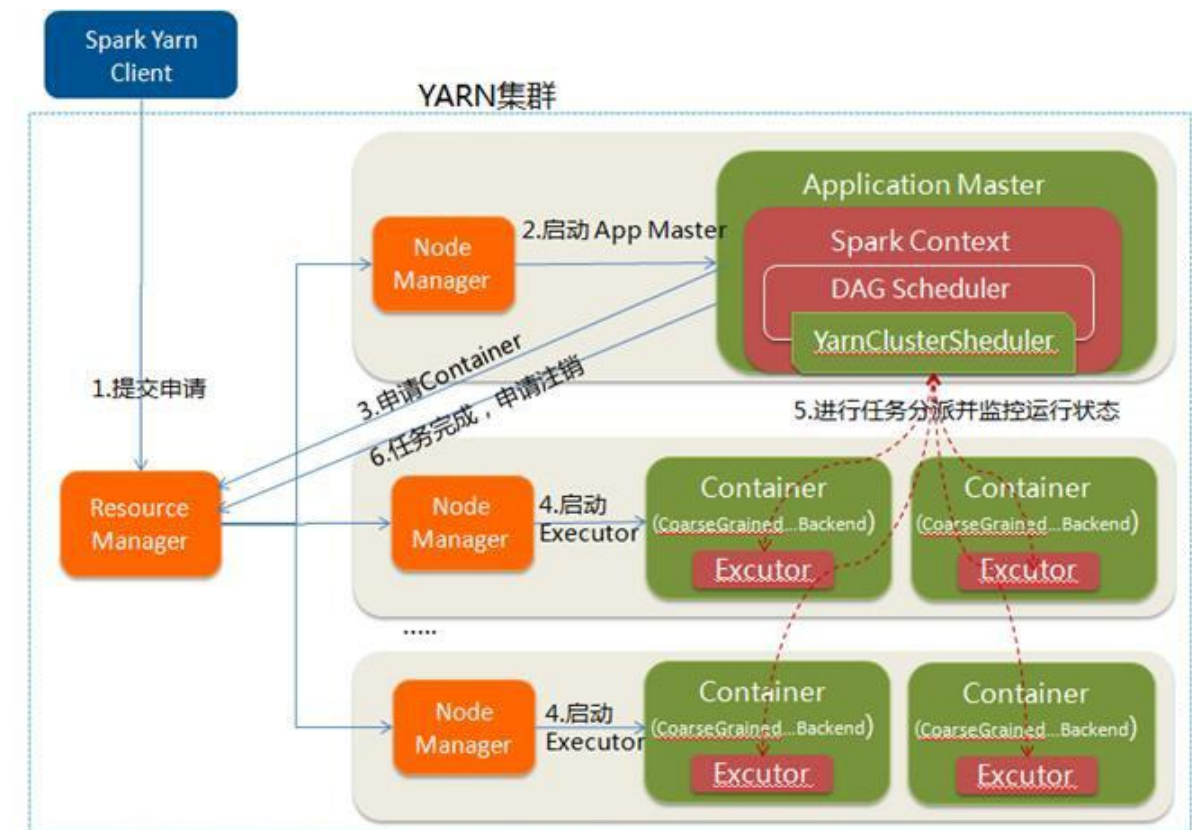


1. Spark Yarn Client 向 YARN 的 ResourceManager 申请启动 Application Master。同时在 SparkContext 初始化中将创建 DAGScheduler 和 TASKScheduler 等，由于我们选择的是 Yarn-Client 模式，程序会选择 YarnClientClusterScheduler 和 YarnClientSchedulerBackend；
2. ResourceManager 收到请求后，在集群中选择一个 NodeManager，为该应用程序分配第一个 Container，要求它在这个 Container 中启动应用程序的 ApplicationMaster，与 YARN-Cluster 区别的是在该 ApplicationMaster 不运行 SparkContext，只与 SparkContext 进行联系进行资源的分派；
3. Client 中的 SparkContext 初始化完毕后，与 ApplicationMaster 建立通讯，向 ResourceManager 注册，根据任务信息向 ResourceManager 申请资源 (Container)；
4. 一旦 ApplicationMaster 申请到资源 (也就是 Container) 后，便与对应的 NodeManager 通信，要求它在获得的 Container 中启动启动 CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend 启动后会向 Client 中的 SparkContext 注册并申请 Task；
5. Client 中的 SparkContext 分配 Task 给 CoarseGrainedExecutorBackend 执行，CoarseGrainedExecutorBackend 运行 Task 并向 Driver 汇报运行的状态和进度，以让 Client 随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务；
6. 应用程序运行完成后，Client 的 SparkContext 向 ResourceManager 申请注销并关闭自己。

2.2.3 YARN-Cluster

在 YARN-Cluster 模式中，当用户向 YARN 中提交一个应用程序后，YARN 将分两个阶段运行该应用程序：第一个阶段是把 Spark 的 Driver 作为一个 ApplicationMaster 在 YARN 集群中先启动；第二个阶段是由 ApplicationMaster 创建应用程序，然后为它向 ResourceManager

申请资源，并启动 Executor 来运行 Task，同时监控它的整个运行过程，直到运行完成。
YARN-cluster 的工作流程分为以下几个步骤：



1. Spark Yarn Client 向 YARN 中提交应用程序，包括 ApplicationMaster 程序、启动 ApplicationMaster 的命令、需要在 Executor 中运行的程序等；
2. ResourceManager 收到请求后，在集群中选择一个 NodeManager，为该应用程序分配第一个 Container，要求它在这个 Container 中启动应用程序的 ApplicationMaster，其中 ApplicationMaster 进行 SparkContext 等的初始化；
3. ApplicationMaster 向 ResourceManager 注册，这样用户可以直接通过 ResourceManager 查看应用程序的运行状态，然后它将采用轮询的方式通过 RPC 协议为各个任务申请资源，并监控它们的运行状态直到运行结束；
4. 一旦 ApplicationMaster 申请到资源（也就是 Container）后，便与对应的 NodeManager 通信，要求它在获得的 Container 中启动启动 CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend 启动后会向 ApplicationMaster 中的 SparkContext 注册并申请 Task。这一点和 Standalone 模式一样，只不过 SparkContext 在 Spark Application 中初始化时，使用 CoarseGrainedSchedulerBackend 配合 YarnClusterScheduler 进行任务的调度，其中 YarnClusterScheduler 只是对 TaskSchedulerImpl 的一个简单包装，增加了对 Executor 的等待逻辑等；
5. ApplicationMaster 中的 SparkContext 分配 Task 给 CoarseGrainedExecutorBackend 执行，CoarseGrainedExecutorBackend 运行 Task 并向 ApplicationMaster 汇报运行的状态和进度，以让 ApplicationMaster 随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务；
6. 应用程序运行完成后，ApplicationMaster 向 ResourceManager 申请注销并关闭自己。

2.2.4 YARN-Client 与 YARN-Cluster 区别

理解 YARN-Client 和 YARN-Cluster 深层次的区别之前先清楚一个概念: Application Master。在 YARN 中, 每个 Application 实例都有一个 ApplicationMaster 进程, 它是 Application 启动的第一个容器。它负责和 ResourceManager 打交道并请求资源, 获取资源之后告诉 NodeManager 为其启动 Container。从深层次的含义讲 YARN-Cluster 和 YARN-Client 模式的区别其实就是 ApplicationMaster 进程的区别。

YARN-Cluster 模式下, Driver 运行在 AM(Application Master)中, 它负责向 YARN 申请资源, 并监督作业的运行状况。当用户提交了作业之后, 就可以关掉 Client, 作业会继续在 YARN 上运行, 因而 YARN-Cluster 模式不适合运行交互类型的作业;

YARN-Client 模式下, Application Master 仅仅向 YARN 请求 Executor, Client 会和请求的 Container 通信来调度他们工作, 也就是说 Client 不能离开。

