尚硅谷大数据技术之面试题复习

（尚硅谷研究院）

版本：V1.6

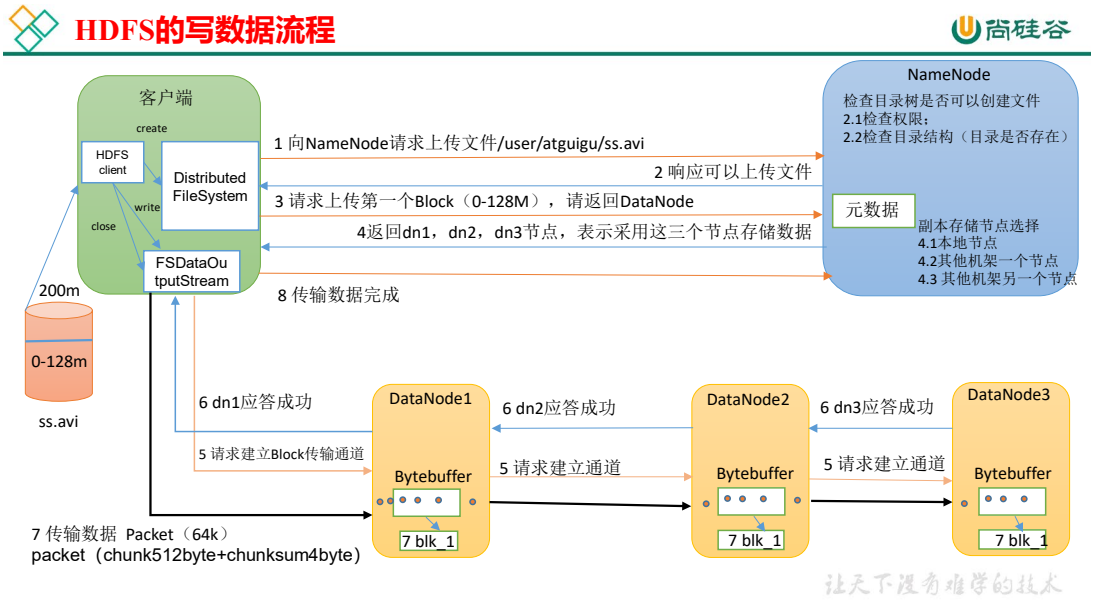
第 1 章 技术复习

1.1 第 1 次考试（准备 20 分钟，考试 30 分钟）

1）Linux 常用高级命令

```
top（查看内存）
jmap -heap（查看进程内存）
free -m（查看系统内存使用情况）
ps -ef|grep 进程（查看进程）
netstat -tunlp|grep 端口（查看端口占用）
df -h 查看磁盘存储
du -sh 路径*（查看路径下磁盘存储情况）
```

2）HDFS 读写流程



3）HDFS 小文件危害及解决办法

```
小文件影响：
    （1）存储层面，1 个文件块占用 NN150 字节内存
    （2）计算层面，每个小文件都起一个 MapTask（默认内存 1G），浪费资源。
解决方法：
```

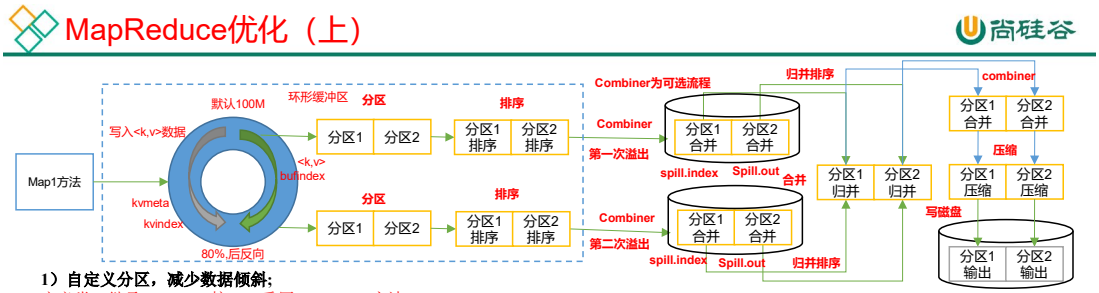
（1）对小文件采用 har 归档

（2）采用 CombineTextInputFormat

（3）有小文件的场景，开启 JVM 重用，通过设置 mapred-site.xml 文件中 mapreduce.job.jvm.numtasks 参数，通常 10-20 之间。

1.2 第 2 次考试（准备 15 分钟，考试 20 分钟）

1）Shuffle 及其优化



1）自定义分区，减少数据倾斜：
定义类，继承Partitioner接口，重写getPartition方法

2）减少溢写的次数
mapreduce.task.io.sort.mb
Shuffle的环形缓冲区大小，默认100m，可以提高到200m
mapreduce.map.sort.spill.percent
环形缓冲区溢出的阈值，默认80%，可以提高的90%

3）增加每次Merge合并次数
mapreduce.task.io.sort.factor默认10，可以提高到20

4）在不影响业务结果的前提条件下可以提前采用Combiner
`job.setCombinerClass(xxxReducer.class);`

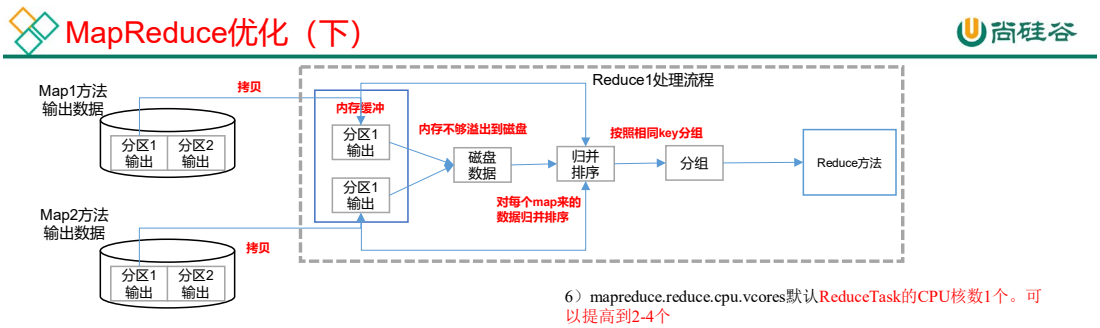
5）为了减少磁盘IO，可以采用Snappy或者LZO压缩
conf.setBoolean("mapreduce.map.output.compress", true);
conf.setClass("mapreduce.map.output.compress.codec",
SnappyCodec.class,CompressionCodec.class);

6）mapreduce.map.memory.mb 默认MapTask内存上限1024MB。
可以根据128m数据对应1G内存原则提高该内存。

7）mapreduce.map.java.opts：控制MapTask堆内存大小。（如果内存不够，报：java.lang.OutOfMemoryError）

8）mapreduce.map.cpu.vcores 默认MapTask的CPU核数1。计算密集型任务可以增加CPU核数

9）异常重试
mapreduce.map.maxattempts每个Map Task最大重试次数，一旦重试次数超过该值，则认为Map Task运行失败，默认值：4。根据机器性能适当提高。



1）mapreduce.reduce.shuffle.parallelcopies每个Reduce去Map中拉取数据的并行数，默认值是5。可以提高到10。

2）mapreduce.reduce.shuffle.input.buffer.percent
Buffer大小占Reduce可用内存的比例，默认值0.7。可以提高到0.8

3）mapreduce.reduce.shuffle.merge.percent Buffer中的数据达到多少比例开始写入磁盘，默认值0.66。可以提高到0.75

4）mapreduce.reduce.memory.mb 默认ReduceTask内存上限1024MB，根据128m数据对应1G内存原则，适当提高内存到4-6G

5）mapreduce.reduce.java.opts：控制ReduceTask堆内存大小。（如果内存不够，报：java.lang.OutOfMemoryError）

6）mapreduce.reduce.cpu.vcores默认ReduceTask的CPU核数1个。可以提高到2-4个

7）mapreduce.reduce.maxattempts每个Reduce Task最大重试次数，一旦重试次数超过该值，则认为Map Task运行失败，默认值：4。

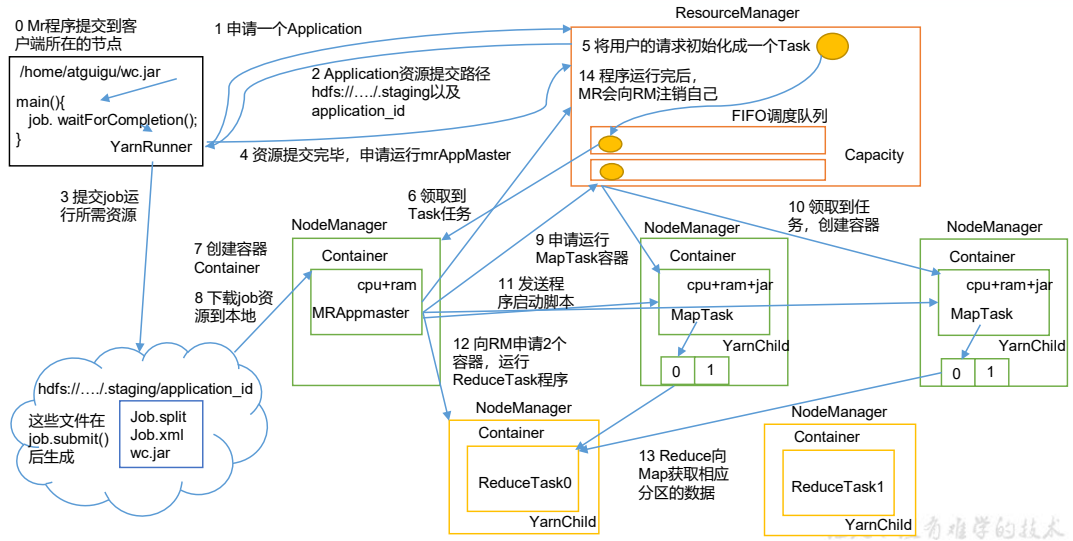
8）mapreduce.job.reduce.slowstart.completedmaps当MapTask完成的比例达到该值后才会为ReduceTask申请资源。默认是0.05。

9）`mapreduce.task.timeout`如果一个Task在一定时间内没有任何进入，即不会读取新的数据，也没有输出数据，则认为该Task处于Block状态，可能是卡住了，也许永远会卡住，为了防止因为用户程序永远Block住不退出，则强制设置了一个该超时时间（单位毫秒），默认是600000（10分钟）。如果你的程序对每条输入数据的处理时间过长，建议将该参数调大。

10）如果可以不用Reduce，尽可能不用

2）Yarn 工作机制

YARN工作机制



YARN生产环境核心参数

1）ResourceManager相关

yarn.resourcemanager.scheduler.class 配置调度器，**默认容量**

yarn.resourcemanager.scheduler.client.thread-count ResourceManager处理调度器请求的线程数量，**默认50**

2）NodeManager相关

yarn.nodemanager.resource.detect-hardware-capabilities 是否让yarn自己检测硬件进行配置，默认false

yarn.nodemanager.resource.count-logical-processors-as-cores 是否将虚拟核数当作CPU核数，默认false

yarn.nodemanager.resource.pcores-vcores-multiplier 虚拟核数和物理核数乘数，例如：4核8线程，该参数就应设为2，默认1.0

yarn.nodemanager.resource.memory-mb NodeManager使用内存，默认8G

yarn.nodemanager.resource.system-reserved-memory-mb NodeManager为系统保留多少内存

以上二个参数配置一个即可

yarn.nodemanager.resource.cpu-vcores NodeManager使用CPU核数，默认8个

yarn.nodemanager.pmem-check-enabled 是否开启物理内存检查限制container，默认打开

yarn.nodemanager.vmem-check-enabled 是否开启虚拟内存检查限制container，默认打开

yarn.nodemanager.vmem-pmem-ratio 虚拟内存物理内存比例，**默认2.1**

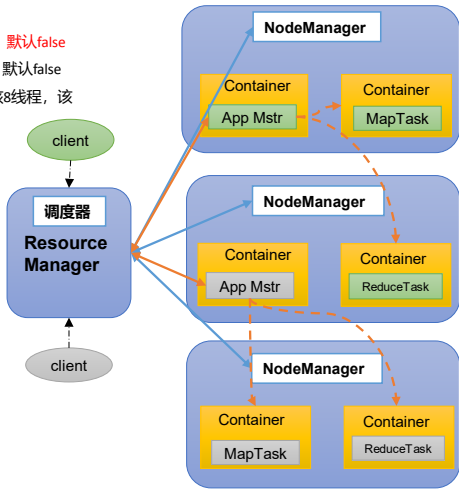
3）Container相关

yarn.scheduler.minimum-allocation-mb 容器最最小内存，默认1G

yarn.scheduler.maximum-allocation-mb 容器最最大内存，默认8G

yarn.scheduler.minimum-allocation-vcores 容器最小CPU核数，默认1个

yarn.scheduler.maximum-allocation-vcores 容器最大CPU核数，默认4个



3）Yarn 中各个调度器特点及生产环境中怎么选择

1）Hadoop 调度器重要分为三类：

FIFO、Capacity Scheduler（容量调度器）和 Fair Sceduler（公平调度器）。

Apache 默认的资源调度器是容量调度器；

CDH 默认的资源调度器是公平调度器。

2）区别：

FIFO 调度器：支持单队列 、先进先出。生产环境不会用。

容量调度器：支持多队列。队列资源分配，优先选择资源占用率最低的队列分配资源；作业资源分配，按照作业的优先级和提交时间顺序分配资源；容器资源分配，本地原则（同一节点/同一机架/不同节点不同机架）

公平调度器：支持多队列，保证每个任务公平享有队列资源。资源不够时可以按照缺额分配。

3）在生产环境下怎么选择？

大厂：如果对并发度要求比较高，选择公平，要求服务器性能必须 OK；

中小公司，集群服务器资源不太充裕选择容量。

4）在生产环境怎么创建队列？

（1）调度器默认就 1 个 default 队列，不能满足生产要求。

（2）按照框架：hive /spark/ flink 每个框架的任务放入指定的队列（企业用的不是特别多）

（3）按照部门：业务部门 1、业务部门 2

（4）按照业务模块：登录注册、购物车、下单

5）创建多队列的好处？

（1）因为担心员工不小心，写递归死循环代码，把所有资源全部耗尽。

（2）实现任务的降级使用，特殊时期保证重要的任务队列资源充足。

4）Zookeeper 非第一次选举机制

选举 Leader 的规则：

（1）EPOCH 大的直接胜出

（2）EPOCH 相同，事务 id 大的胜出

（3）事务 id 相同，服务器 id 大的胜出

5）Zookeeper 符合 CAP 法则中哪两个

C：一致性

A：可用性

P：分区容错性

Zookeeper 保证的是 CP，Zookeeper 在选举时不能一直对外提供服务，不满足可用性。

1.3 第 3 次考试（准备 20 分钟，考试 25 分钟）

1）解释一下零点漂移产生的原因及解决办法（Flume）

1）拦截器注意事项

（1）ETL 拦截器：主要是用来判断 json 是否完整。没有做复杂的清洗操作主要是防止过多的降低传输速率。

（2）时间戳拦截器：主要是解决零点漂移问题

2）自定义拦截器步骤

（1）实现 Interceptor

（2）重写四个方法

initialize 初始化

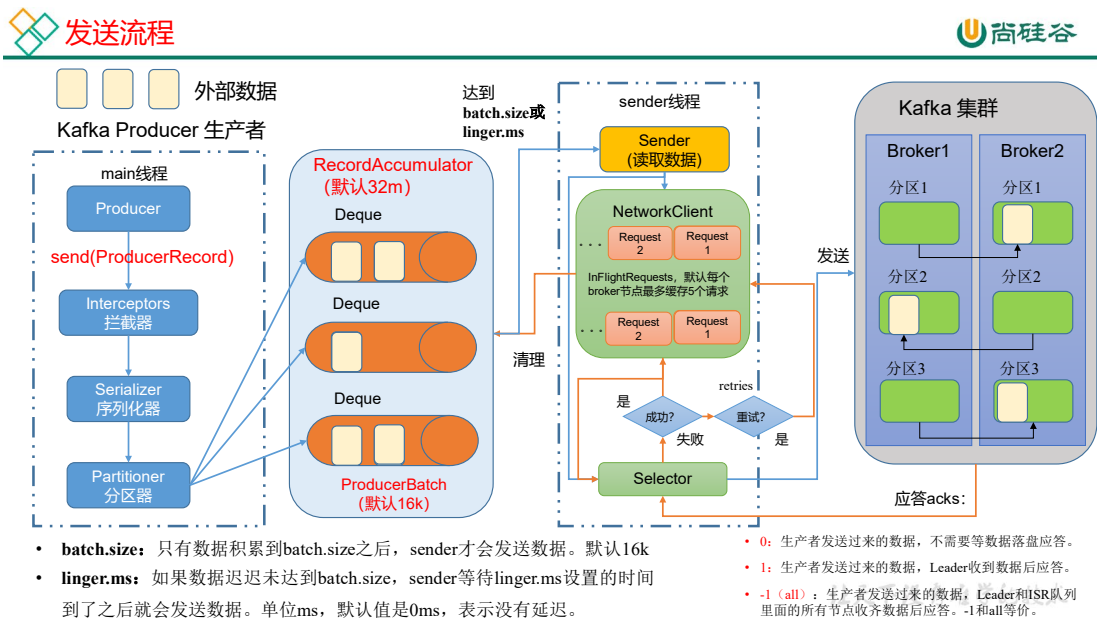
public Event intercept(Event event) 处理单个 Event

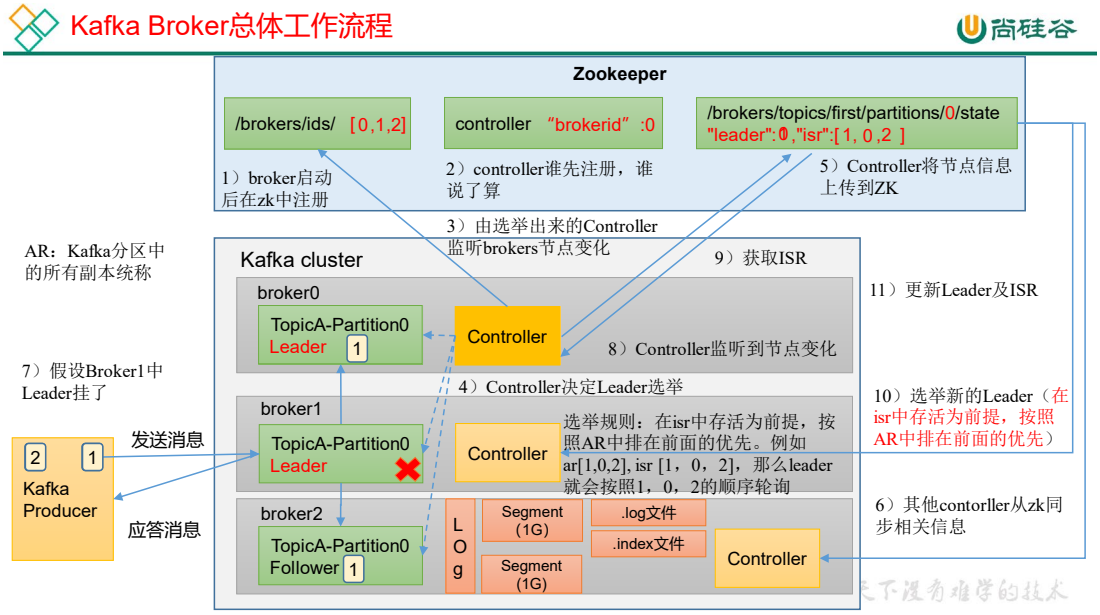
public List<Event> intercept(List<Event> events) 处理多个 Event

close 方法

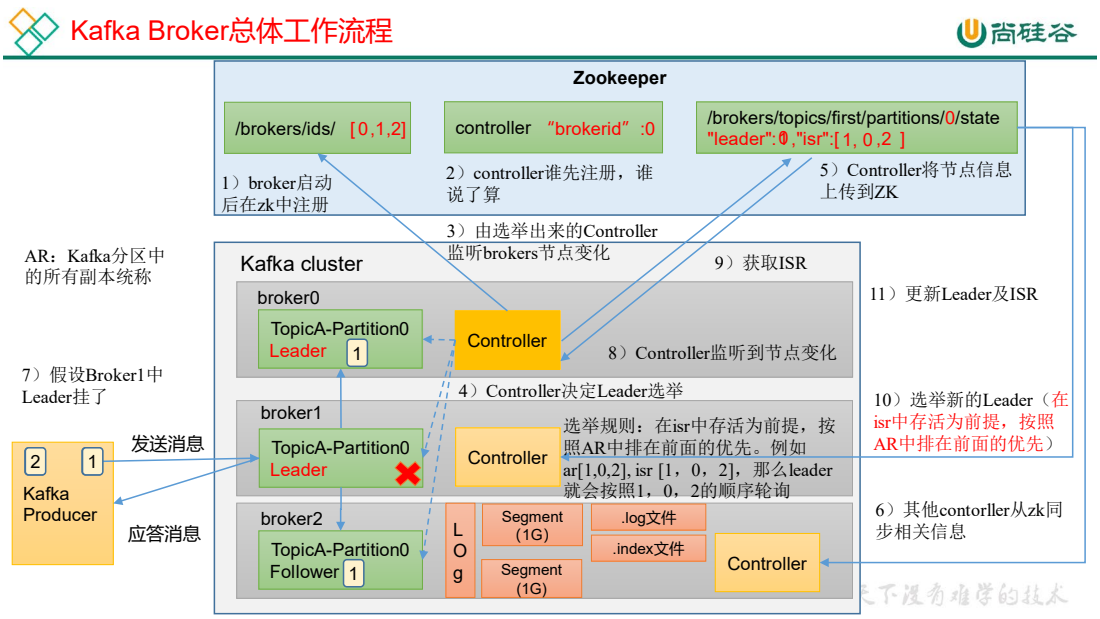
（3）静态内部类，实现 Interceptor.Builder

2）Kafka 生产者发消息流程

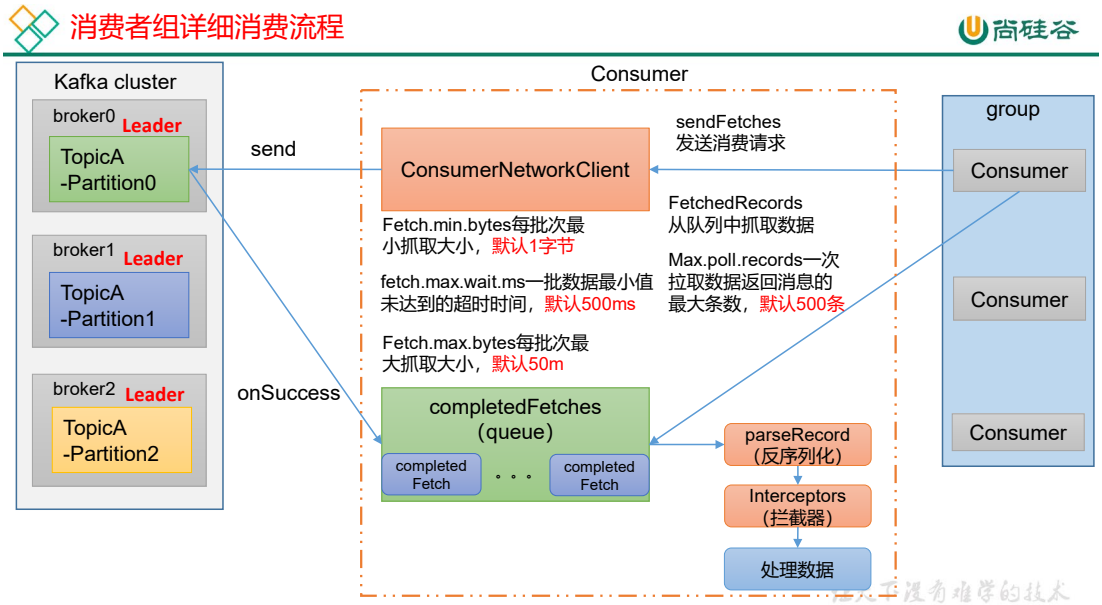




3）Kafka 的 Broker 工作流程



4）Kafka 的消费者组消费流程



1.4 第 4 次考试（准备 20 分钟，考试 25 分钟）

1）Kafka 挂了如何处理

（1）紧急处理方式：先尝试重新启动一下，如果能启动正常，那直接解决。

（2）分析影响：分析数据的一致性（丢数、重复、乱序）。如果数据出现问题，在日志服务器有 30 天备份

（3）定位问题：查看 Kafka 日志

（4）解决问题：如果因为资源问题，如 OOM、CPU 使用率过高等，考虑增加内存、增加 CPU、网络带宽。

2）Kafka 怎么保证数据不丢

1）producer 角度

acks=0，生产者发送过来数据就不管了，可靠性差，效率高；

acks=1，生产者发送过来数据 Leader 应答，可靠性中等，效率中等；

acks=-1，生产者发送过来数据 Leader 和 ISR 队列里面所有 Follwer 应答，可靠性高，效率低；

在生产环境中，acks=0 很少使用；acks=1，一般用于传输普通日志，允许丢个别数据；acks=-1，一般用于传输和钱相关的数据，对可靠性要求比较高的场景。

retries 参数控制当数据发送失败时候的重试次数，默认为 int 最大值，当发送失败时候会自动进行重试。生产中如果不希望数据丢失，retries 参数必须大于 0。即使 ack=-1，当 retries 为 0 时候数据也会丢失。

2）broker 角度

副本数大于等于 2

min.insync.replicas 大于等于 2

3）Kafka 数据重复如何处理

精确一次 = 幂等性 + 事务

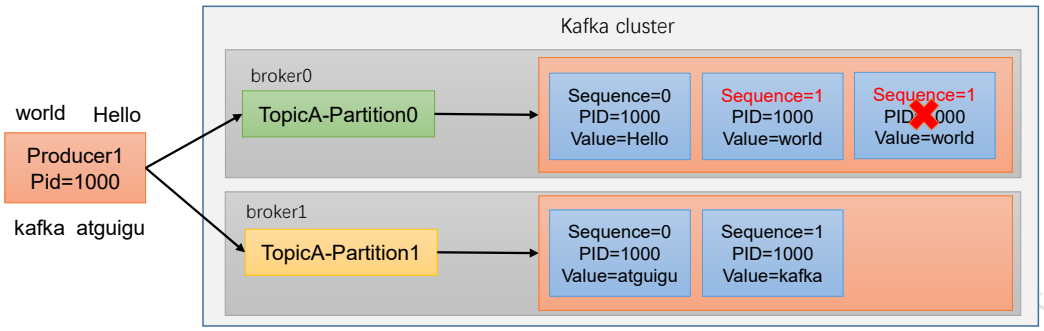
幂等性原理

幂等性就是指Producer不论向Broker发送多少次重复数据，Broker端都只会持久化一条，保证了不重复。

精确一次（Exactly Once）= 幂等性 + 至少一次（ ack=-1 + 分区副本数>=2 + ISR最小副本数量>=2）。

重复数据的判断标准：具有<PID, Partition, SeqNumber>相同主键的消息提交时，Broker只会持久化一条。其中PID是Kafka每次重启都会分配一个新的；Partition 表示分区号；Sequence Number是单调自增的。

所以幂等性只能保证的是在单分区单会话内不重复。



总结，如果要实现**全局的精准一次**，需要达到以下条件。

（1）生产者角度

acks 设置为-1 （acks=-1）。

幂等性（enable.idempotence = true） + 事务 。

（2）broker 服务端角度

分区副本大于等于 2 （--replication-factor 2）。

ISR 里应答的最小副本数量大于等于 2 （min.insync.replicas = 2）。

（3）消费者

事务 + 手动提交 offset （enable.auto.commit = false）。

消费者输出的目的地必须支持事务（MySQL、Kafka）。

4）Kafka 数据积压如何处理

（1）如果是 Kafka 消费能力不足，则可以考虑增加 Topic 的分区数，并且同时提升消费组的消费者

数量，消费者数 = 分区数。（两者缺一不可）

增加分区数通过 alter 命令完成

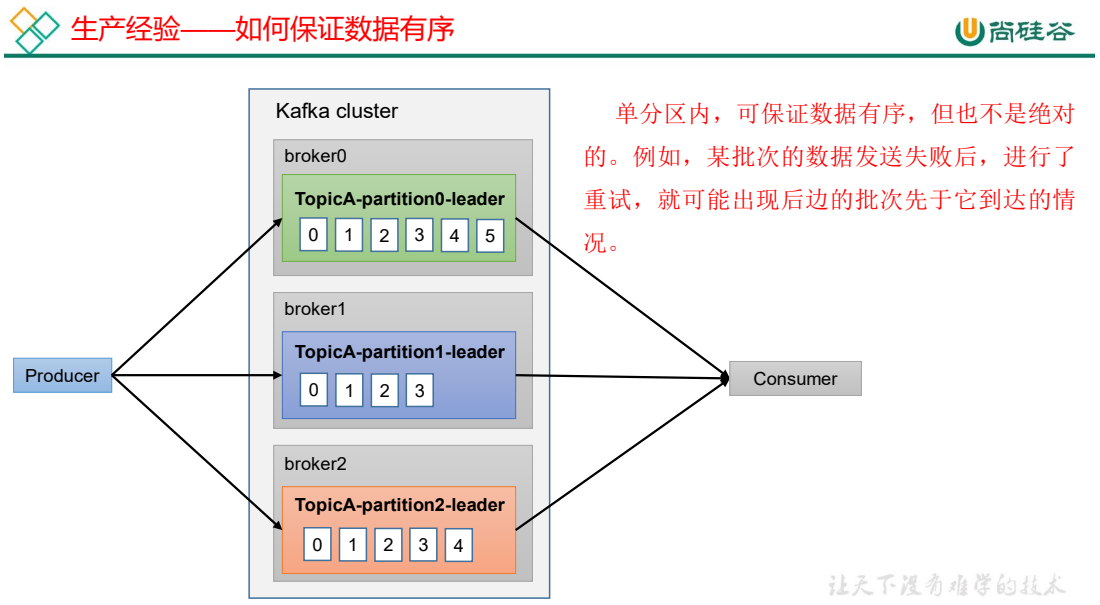
```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --alter --topic first --partitions 3
```

（2）如果是下游的数据处理不及时：提高每批次拉取的数量。批次拉取数据过少（拉取数据/处理时间 < 生产速度），使处理的数据小于生产的数据，也会造成数据积压。

参数名称	描述
fetch.max.bytes	默认 Default: 52428800（50 m）。消费者获取服务器端一批消息最大的字节数。如果服务器端一批次的数据大于该值（50m）仍然可以拉取回来这批数据，因此，这不是一个绝对最大值。一批次的大小受 message.max.bytes （broker config）or max.message.bytes （topic config）影响。
max.poll.records	一次 poll 拉取数据返回消息的最大条数，默认是 500 条

5）Kafka 如何保证数据有序 or 怎么解决乱序问题

（1）Kafka 最多只保证单分区内的消息是有序的，所以如果要保证业务全局严格有序，就要设置 Topic 为单分区。



（2）如何保证单分区内数据有序？

生产经验——如何保证单分区数据有序

方案一：

禁止重试，需设置以下参数

设置retries等于0

说明：数据出现乱序的根本原因是，失败重试，关闭重试，则可保证数据是有序的。但是这样做，可能会导致数据的丢失。

方案二：

启用幂等，需设置以下参数

设置enable.idempotence=true，启用幂等

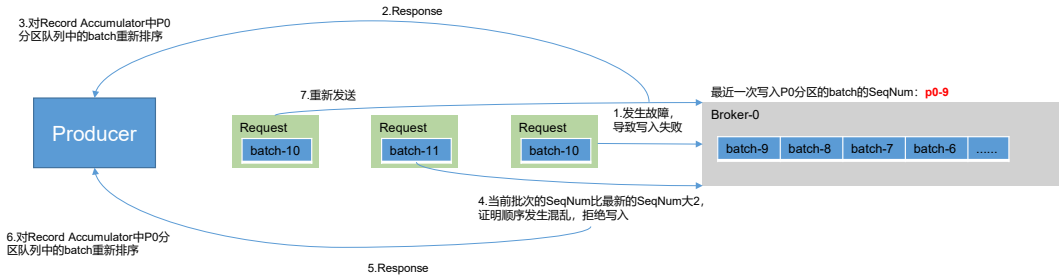
设置max.in.flight.requests.per.connection，1.0.X版本前，需设置为1，1.0.X之后，小于等于5

设置retries，保证其大于0

设置acks，保证其为all

注：幂等机制保证数据有序的原理如下：

生产经验——如何保证数据有序



1.5 第 5 次考试（准备 10 分钟，考试 25 分钟）

1）Kafka 怎么做到的高效读写

1）Kafka 本身是分布式集群，可以采用分区技术，并行度高

2）读数据采用稀疏索引，可以快速定位要消费的数据

3）顺序写磁盘

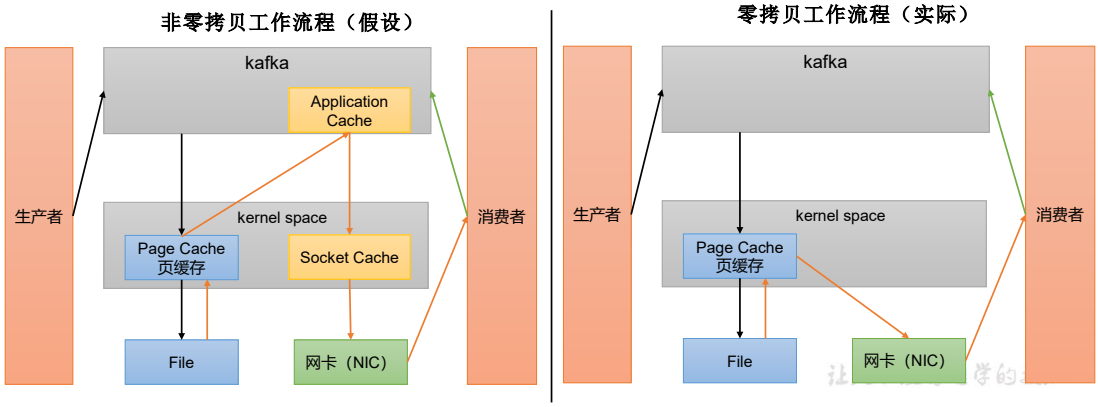
Kafka 的 producer 生产数据，要写入到 log 文件中，写的过程是一直追加到文件末端，为顺序写。官网有数据表明，同样的磁盘，顺序写能到 600M/s，而随机写只有 100K/s。这与磁盘的机械机构有关，顺序写之所以快，是因为其省去了大量磁头寻址的时间。

4）页缓存 + 零拷贝技术

页缓存 + 零拷贝技术

零拷贝：Kafka的数据加工处理操作交由Kafka生产者和Kafka消费者处理。Kafka Broker应用层不关心存储的数据，所以就不用走应用层，传输效率高。

PageCache页缓存：Kafka重度依赖底层操作系统提供的PageCache功能。当上层有写操作时，操作系统只是将数据写入PageCache。当读操作发生时，先从PageCache中查找，如果找不到，再去磁盘中读取。实际上PageCache是把尽可能多的空闲内存都当做了磁盘缓存来使用。



2）Kafka 如何提高吞吐量

1）提高生产者的吞吐量

（1）buffer.memory：发送消息的缓冲区大小，默认值是 32m，可以增加到 64m。

（2）batch.size：默认是 16k。如果 batch 设置太小，会导致频繁网络请求，吞吐量下降；如果 batch 太大，会导致一条消息需要等待很久才能被发送出去，增加网络延时。

（3）linger.ms，这个值默认是 0，意思就是消息必须立即被发送。一般设置一个 5-100 毫秒。如果 linger.ms 设置的太小，会导致频繁网络请求，吞吐量下降；如果 linger.ms 太长，会导致一条消息需要等待很久才能被发送出去，增加网络延时。

（4）compression.type：默认是 none，不压缩，但是也可以使用 lz4 压缩，效率还是不错的，压缩之后可以减小数据量，提升吞吐量，但是会加大 producer 端的 CPU 开销。

2）增加分区

3）提高消费者的吞吐量

（1）调整 fetch.max.bytes 大小，默认是 50m。

（2）调整 max.poll.records 大小，默认是 500 条。

3）消费策略（Range、RoundRobin、粘性）

生产端分区分配

Kafka 原则

在IDEA中全局查找（ctrl +n）ProducerRecord类，在类中可以看到如下构造方法：

```
public ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value, Iterable<Header> headers) {
    ... ...
}
public ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value) {
    ... ...
}
public ProducerRecord(String topic, Integer partition, K key, V value, Iterable<Header> headers) {
    ... ...
}
public ProducerRecord(String topic, Integer partition, K key, V value) {
    ... ...
}
public ProducerRecord(String topic, K key, V value) {
    ... ...
}
public ProducerRecord(String topic, V value) {
    ... ...
}
```

（1）指明partition的情况下，直接将指明的值作为partition值；例如partition=0，所有数据写入分区0

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

例如：key1的hash值=5， key2的hash值=6 ，topic的partition数=2，那么key1 对应的value1写入1号分区，key2对应的value2写入0号分区。

（3）既没有partition值又没有key值的情况下，Kafka采用Sticky Partition（黏性分区器），会随机选择一个分区，并尽可能一直使用该分区，待该分区的batch已满或者已完成，Kafka再随机一个分区进行使用（和上一次的分区不同）。

例如：第一次随机选择0号分区，等0号分区当前批次满了（默认16k）或者linger.ms设置的时间到， Kafka再随机一个分区进行使用（如果还是0会继续随机）。

消费端分区分配

分区分配策略之Range

Range 是对每个 topic 而言的。

首先对同一个 topic 里面的分区按照序号进行排序，并对消费者按照字母顺序进行排序。

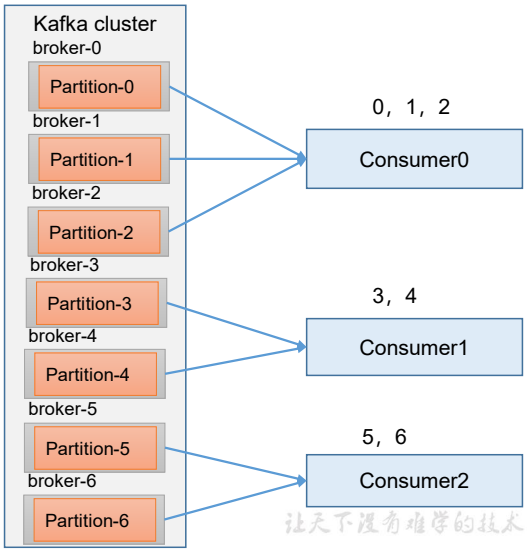
假如现在有 7 个分区，3 个消费者，排序后的分区将会是0,1,2,3,4,5,6；消费者排序完之后将会是C0,C1,C2。

通过 partitions数/consumer数 来决定每个消费者应该消费几个分区。如果除不尽，那么前面几个消费者将会多消费 1 个分区。

例如，7/3 = 2 余 1 ，除不尽，那么 消费者 C0 便会多消费 1 个分区。 8/3=2余2，除不尽，那么C0和C1分别多消费一个。

注意：如果只是针对 1 个 topic 而言，C0消费者多消费1个分区影响不是很大。但是如果有 N 多个 topic，那么针对每个 topic，消费者 C0都将多消费 1 个分区，topic越多，C0消费的分区会比其他消费者明显多消费 N 个分区。

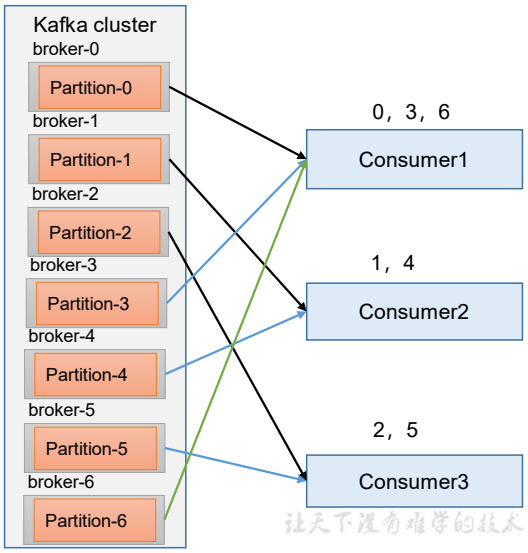
容易产生数据倾斜！



分区分配策略之RoundRobin

RoundRobin 针对集群中所有Topic而言。

RoundRobin 轮询分区策略，是把所有的 partition 和所有的 consumer 都列出来，然后按照 hashcode 进行排序，最后通过轮询算法来分配 partition 给到各个消费者。



1.6 第 6 次考试（准备 20 分钟，考试 25 分钟）

1）Hive 优化

1）MapJoin

用 MapJoin 把小表全部加载到内存在 map 端进行 join，避免 reducer 处理。

2）行列过滤

列处理：在 SELECT 中，只拿需要的列，如果有，尽量使用分区过滤，少用 SELECT *。

行处理：在分区剪裁中，当使用外关联时，如果将副表的过滤条件写在 Where 后面，那么就会先全表关联，之后再过滤。

3）列式存储

4）采用分区技术

5）合理设置 Map 数

mapred.min.split.size: 指的是数据的最小分割单元大小；min 的默认值是 1B

mapred.max.split.size: 指的是数据的最大分割单元大小；max 的默认值是 256MB

通过调整 max 可以起到调整 map 数的作用，减小 max 可以增加 map 数，增大 max 可以减少 map 数。

需要提醒的是，直接调整 mapred.map.tasks 这个参数是没有效果的。

6）合理设置 Reduce 数

Reduce 个数并不是越多越好

（1）过多的启动和初始化 Reduce 也会消耗时间和资源；

（2）另外，有多少个 Reduce，就会有多少个输出文件，如果生成了很多个小文件，那么如果这些小文件作为下一个任务的输入，则也会出现小文件过多的问题；

在设置 Reduce 个数的时候也需要考虑这两个原则：处理大数据量利用合适的 Reduce 数；使单个 Reduce 任务处理数据量大小要合适；

7）小文件如何产生的？

（1）动态分区插入数据，产生大量的小文件，从而导致 map 数量剧增；

（2）reduce 数量越多，小文件也越多（reduce 的个数和输出文件是对应的）；

（3）数据源本身就包含大量的小文件。

8）小文件解决方案

（1）在 Map 执行前合并小文件，减少 Map 数：CombineHiveInputFormat 具有对小文件进行合并的功能（系统默认的格式）。HiveInputFormat 没有对小文件合并功能。

（2）merge

```
// 输出合并小文件
SET hive.merge.mapfiles = true; -- 默认 true，在 map-only 任务结束时合并小文件
SET hive.merge.mapredfiles = true; -- 默认 false，在 map-reduce 任务结束时合并小文件
SET hive.merge.size.per.task = 268435456; -- 默认 256M
SET hive.merge.smallfiles.avgsize = 16777216; -- 当输出文件的平均大小小于 16m 该值时，启动一个独立的 map-reduce 任务进行文件 merge
```

（3）开启 JVM 重用

```
set mapreduce.job.jvm.numtasks=10
```

9）开启 map 端 combiner（不影响最终业务逻辑）

```
set hive.map.aggr=true;
```

10）压缩（选择快的）

设置 map 端输出、中间结果压缩。（不完全是解决数据倾斜的问题，但是减少了 IO 读写和网络传输，能提高很多效率）

```
set hive.exec.compress.intermediate=true --启用中间数据压缩
set mapreduce.map.output.compress=true --启用最终数据压缩
set mapreduce.map.outout.compress.codec=…; --设置压缩方式
```

11）采用 tez 引擎或者 spark 引擎

2）Hive 数据倾斜

（1）不同数据类型关联产生数据倾斜

情形：比如用户表中 user_id 字段为 int，log 表中 user_id 字段 string 类型。当按照 user_id 进行两个表的 Join 操作时。

解决方式：把数字类型转换成字符串类型

（2）控制空值分布

在生产环境经常会用大量空值数据进入到一个 reduce 中去，导致数据倾斜。

解决办法：自定义分区，将为空的 key 转变为字符串加随机数或纯随机数，将因空值而造成倾斜的数据分不到多个 Reducer。

（3）单表 group by

单 key

- 加随机数，双重聚合
- 配置参数，双重聚合 set hive.groupby.skewindata = true;
- 过滤出这个 Key 单独处理

多 key

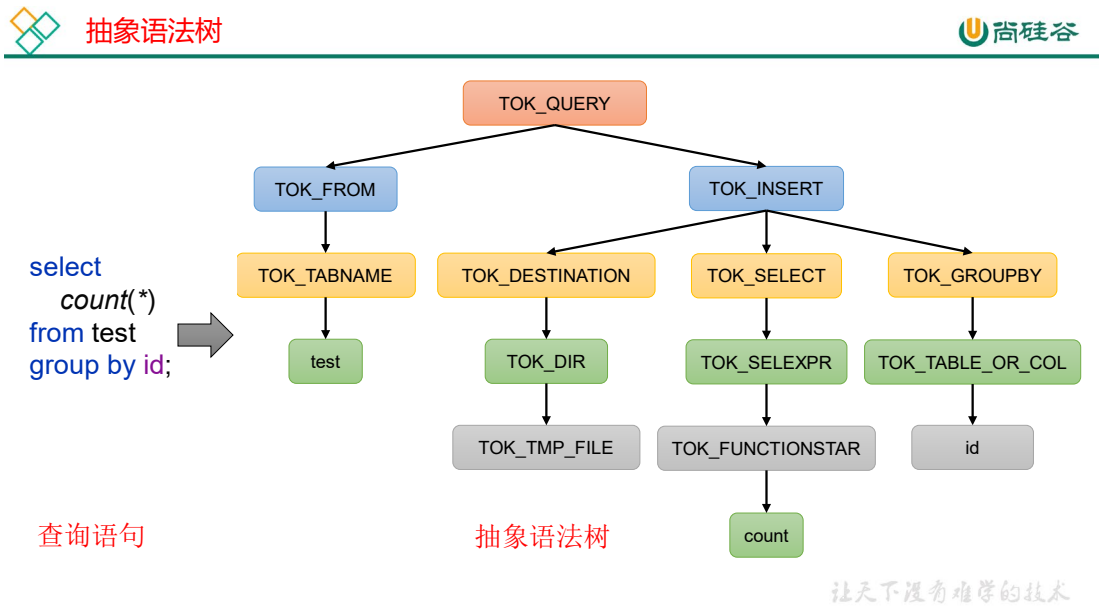
- 增加 Reducer 个数，一定程度上解决问题
- 自定义分区器
- 加随机数，双重聚合
- 配置参数，双重聚合 set hive.groupby.skewindata = true;

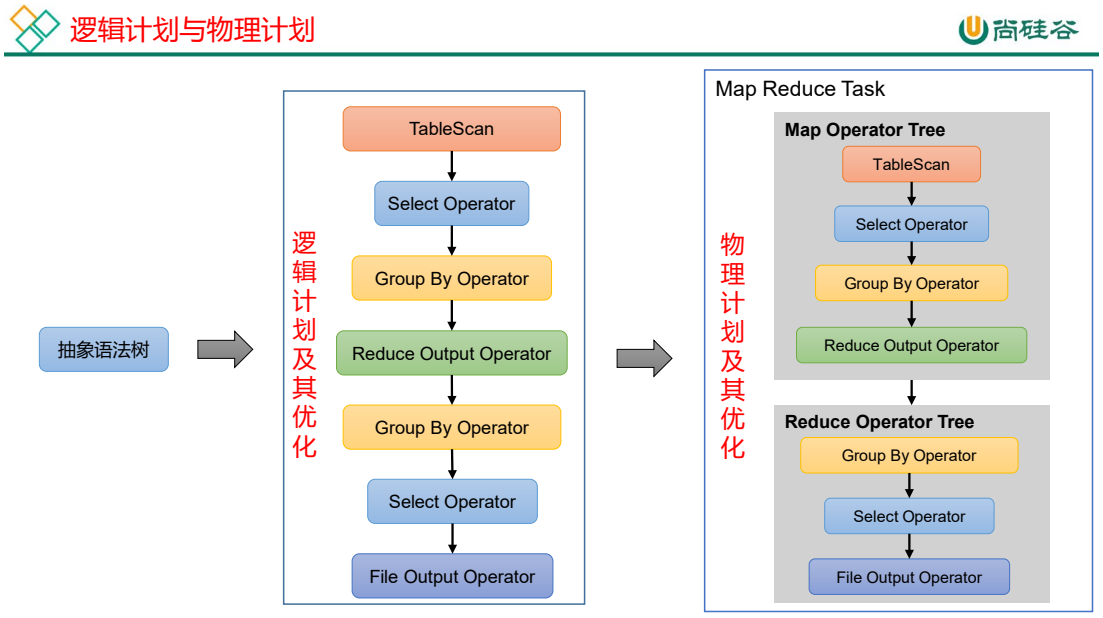
（4）JOIN ON 关联字段

大表 JOIN 小表　mapJoin　　避免了 Reducer

大表 JOIN 大表　A 表加随机数　B 表扩容　聚合

3）怎么将 Hive SQL 转换为可以执行的 MR





（1）解析器（SQLParser）：将 SQL 字符串转换成抽象语法树（AST）

（2）语义分析器（Semantic Analyzer）：将 AST 进一步抽象为 QueryBlock（可以理解

为一个子查询划分成一个 QueryBlock）

（2）逻辑计划生成器（Logical Plan Gen）：由 QueryBlock 生成逻辑计划

（3）逻辑优化器（Logical Optimizer）：对逻辑计划进行优化

（4）物理计划生成器（Physical Plan Gen）：根据优化后的逻辑计划生成物理计划

（5）物理优化器（Physical Optimizer）：对物理计划进行优化

（6）执行器（Execution）：执行该计划，得到查询结果并返回给客户端

1.7 第 7 次考试（准备 20 分钟，考试 25 分钟）

1）MaxWell 底层原理及为什么选择

MySQL 主从复制

支持断点续传。

全量同步。

自动根据库名和表名把数据发往 kafka 的对应主题。

2）MaxWell 怎么产生的重复数据，如何解决

同步历史数据时，bootstrap 引导服务需要依赖 Maxwell 进程。

同时 binlog 也在监听。

如果同步历史数据过程中，源表的数据发生改变：

=》 bootstrap 会同步到

=》 maxwell 监听 binlog 也会同步到

数据产生重复，只能保证至少一致性

解决： 根据 type 过滤，

bootstrap 同步的 type 为 bootstrap-insert

binlog 同步的为 insert

3）DataX 在使用过程中遇到哪些问题，怎么解决的

空置处理

1）MySQL（null） => hive （\N） 要求 hive 建表语句

解决该问题的方案有两个：

（1）修改 DataX HDFS Writer 的源码，增加自定义 null 值存储格式的逻辑，可参考 https://blog.csdn.net/u010834071/article/details/105506580。

（2）在 Hive 中建表时指定 null 值存储格式为空字符串（''），例如：

2）hive（\N） => mysql （null）

4）Spark 转换算子 10 个

1）单 Value

（1）map

（2）mapPartitions

（3）mapPartitionsWithIndex

（4）flatMap
（5）glom
（6）groupBy
（7）filter
（8）sample
（9）distinct
（10）coalesce
（11）repartition
（12）sortBy
（13）pipe
2）双 vlaue
（1）intersection
（2）union
（3）subtract
（4）zip
3）Key-Value
（1）partitionBy
（2）reduceByKey
（3）groupByKey
（4）aggregateByKey
（5）foldByKey
（6）combineByKey
（7）sortByKey
（8）mapValues

5）Spark 行动算子 5 个

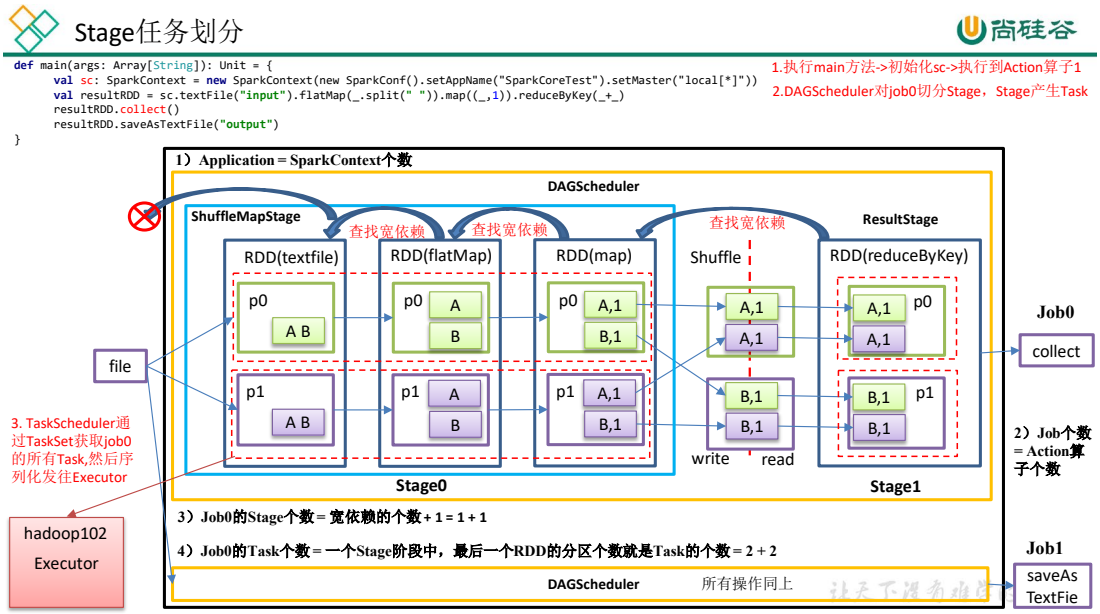
（1）reduce
（2）collect
（3）count
（4）first
（5）take
（6）takeOrdered
（7）aggregate
（8）fold
（9）countByKey
（10）save
（11）foreach

6）Spark 任务怎么切分的

（1）Application：初始化一个 SparkContext 即生成一个 Application；
（2）Job：一个 Action 算子就会生成一个 Job；

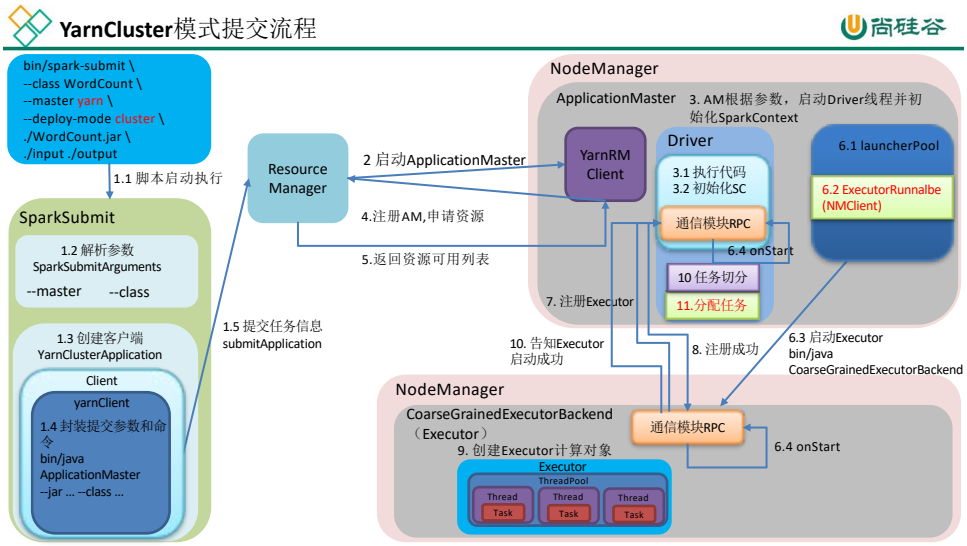
（3）Stage：Stage 等于宽依赖的个数加 1；

（4）Task：一个 Stage 阶段中，最后一个 RDD 的分区个数就是 Task 的个数。

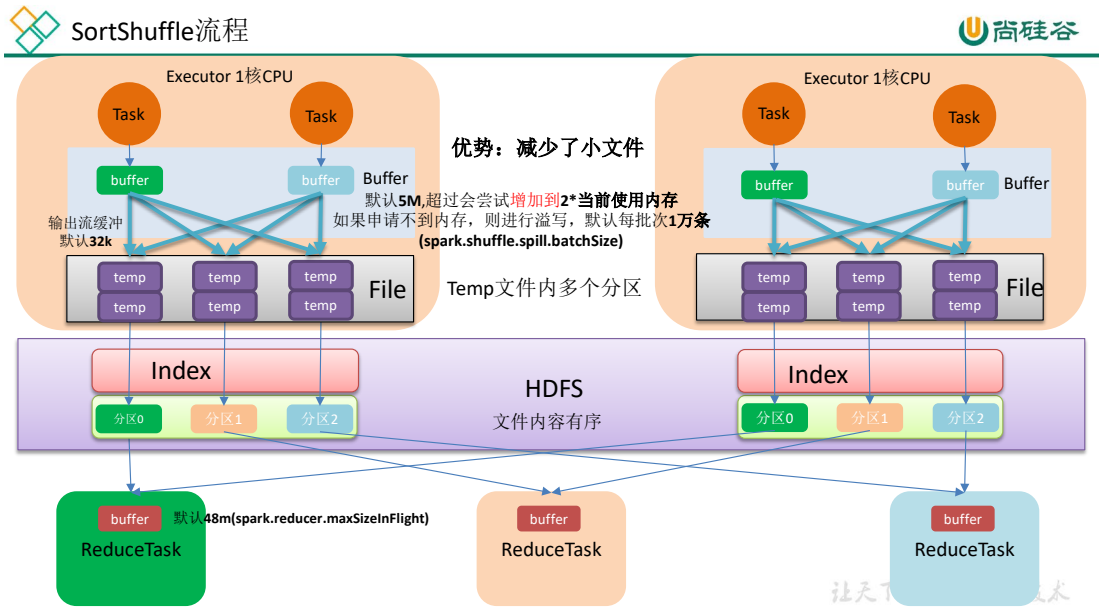


1.8 第 8 次考试（准备 20 分钟，考试 25 分钟）

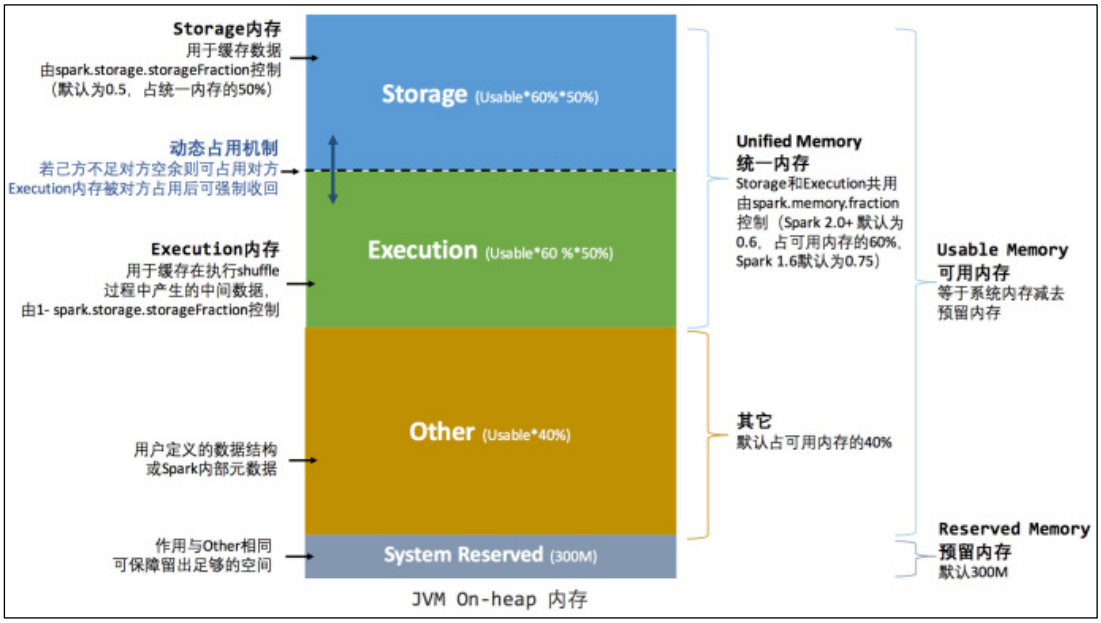
1）Spark 提交流程



2）SortShuffle 原理



3）统一内存模型



1.9 第 9 次考试（准备 20 分钟，考试 15 分钟）

1）Flink 的架构有哪些角色

Flink 程序在运行时主要有 TaskManager，JobManager，Client 三种角色。

JobManager 是集群的老大，负责接收 Flink Job，协调检查点，Failover 故障恢复等，同时管理 TaskManager。 包含：Dispatcher、ResourceManager、JobMaster。

TaskManager 是执行计算的节点，每个 TaskManager 负责管理其所在节点上的资源信息，如内存、磁盘、网络。内部划分 slot 隔离内存，不隔离 cpu。同一个 slot 共享组的不同算子的 subtask 可以共享 slot。

Client 是 Flink 程序提交的客户端，将 Flink Job 提交给 JobManager。

2）Flink 与 Spark Streaming 的区别

	Flink	Spark Streaming
计算模型	流计算	微批次
时间语义	三种	没有，处理时间
乱序	有	没有
窗口	多、灵活	少、不灵活（窗口长度必须是 批次的整数倍）
checkpoint	异步分界线快照	弱
状态	有，多	没有（updatestatebykey）
流式 sql	有	没有

3）介绍一下时间语义，谈谈你对 Watermark 的理解

1）时间语义

Event Time：是事件创建的时间。数据本身携带的时间。

Ingestion Time：是数据进入 Flink 的时间。

Processing Time：是每一个执行基于时间操作的算子的本地系统时间，与机器相关，默认的时间属性就是 Processing Time。

2）你对 Watermark 的认识

（1）Watermark 是一种衡量 Event Time 进展的机制，是一个逻辑时钟

（2）Watermark 是用于处理乱序事件的，而正确的处理乱序事件，通常用 Watermark 机制结合 window 来实现

（3）基于事件时间，用来触发窗口、定时器等

（4）watermark 主要属性就是时间戳，可以理解一个特殊的数据，插入到流里面

（5）watermark 是单调不减的

（6）数据流中的 Watermark 用于表示 timestamp 小于 Watermark 的数据，都已经到达了，如果后续还有 timestamp 小于 Watermark 的数据到达，称为迟到数据

4）窗口的分类、划分、生命周期

1）窗口分类：

Keyed Window 和 Non-keyed Window

基于时间：滚动、滑动、会话

基于数量：滚动、滑动

2）窗口的划分：如，基于事件时间的滚动窗口

start=按照数据的事件时间向下取窗口长度的整数倍

end=start+size

比如开了一个 10s 的滚动窗口，第一条数据是 857s，那么它属于[850s,860s)

3）窗口的创建：当属于某个窗口的第一个元素到达，Flink 就会创建一个窗口，并且放入单例集合

4）窗口的销毁：时间进展 >= 窗口最大时间戳 + 窗口允许延迟时间

（Flink 保证只删除基于时间的窗口，而不能删除其他类型的窗口，例如全局窗口）。

5）Flink 如何解决乱序问题

1）watermark 的乱序等待时间

2）使用窗口时，可以允许迟到

3）迟到特别久的，放到侧输出流处理

1.10 第 10 次考试（准备 15 分钟，考试 20 分钟）

1）介绍一下 Flink 的状态和状态后端

1） 算子状态：每个并行实例 各自维护状态

List

unionList

键控状态：经过 keyby 的，每个 key 各自维护状态

value、list、map

广播

2）状态后端

<1.13

	本地状态	checkpoint 存储
memory	TM 内存	JM 内存
fs	TM 内存	HDFS
rocksdb	TM 的 Rocksdb	HDFS

>=1.13

	本地状态
hashmap	TM 内存
rocksdb	TM 的 rocksdb

checkpoint 存储单独自己指定： JM 内存、hdfs

又短又小的，一般作业： hashmap ===》 快

又大又长的： rocksdb ===》 增量，比 hashmap 快、节省资源

我们怎么用： 基本都是 hashmap，个别状态大的用 rocksdb（3 个）

2）有没有状态比较大的作业，遇到过什么问题，怎么解决

开启 State 访问性能监控

Flink 1.13 中引入了 State 访问的性能监控，即 latency trackig state。此功能不局限于 State Backend 的类型，自定义实现的 State Backend 也可以复用此功能。

开启增量检查点和本地恢复

1）开启增量检查点

RocksDB 是目前唯一可用于支持有状态流处理应用程序增量检查点的状态后端，可以修改参数开启增量检查点：

state.backend.incremental: true #默认 false，改为 true。

或代码中指定

new EmbeddedRocksDBStateBackend(true)

2）开启本地恢复

当 Flink 任务失败时，可以基于本地的状态信息进行恢复任务，可能不需要从 hdfs 拉取数据。

本地恢复目前仅涵盖键控类型的状态后端（RocksDB），MemoryStateBackend 不支持本地恢复并忽略此选项。

```
state.backend.local-recovery: true
```

3）设置多目录

如果有多块磁盘，也可以考虑指定本地多目录

调整预定义选项

Flink 针对不同的设置为 RocksDB 提供了一些预定义的选项集合，其中包含了后续提到的一些参数，如果调整预定义选项后还达不到预期，再去调整后面的 block、writebuffer 等参数。

当前支持的预定义选项有 DEFAULT、SPINNING_DISK_OPTIMIZED、SPINNING_DISK_OPTIMIZED_HIGH_MEM 或 FLASH_SSD_OPTIMIZED。有条件上 SSD 的，可以指定为 FLASH_SSD_OPTIMIZED

```
state.backend.rocksdb.predefined-options : SPINNING_DISK_OPTIMIZED_HIGH_MEM
#设置为机械硬盘+内存模式
```

3）Flink 有没有遇到过反压，怎么发现、定位、分析、解决，效果如何

①反压的原因

短时间的负载高峰导致系统接收数据的速率远高于它处理数据的速率。许多日常问题都会导致反压，例如，垃圾回收停顿可能会导致流入的数据快速堆积，或遇到大促、秒杀活动导致流量陡增。

②反压的危害

会影响到 checkpoint 时长和 state 大小，甚至可能会导致资源耗尽甚至系统崩溃

checkpoint 是保证数据一致性的关键，checkpoint 时间变长有可能导致 checkpoint 超时失败，而 state 大小同样可能拖慢 checkpoint 甚至导致 OOM（使用 Heap-based StateBackend）或者物理内存使用超出容器资源（使用 RocksDBStateBackend）的稳定性问题。

③定位反压

利用 web ui 定位

定位到造成反压的节点，排查的时候，先把 operator chain 禁用，方便定位到具体算子。

Flink 现在在 UI 上通过颜色和数值来展示繁忙和反压的程度。

通过 WebUI 看具体算子的 BackPressure 显示 High。

分析瓶颈算子

如果处于反压状态，那么有两种可能性：

（1）该节点的发送速率跟不上它的产生数据速率。这一般会发生在一条输入多条输出的 Operator（比如 flatmap）。这种情况，该节点是反压的根源节点，它是从 Source Task 到 Sink Task 的第一个出现反压的节点。

（2）下游的节点接受速率较慢，通过反压机制限制了该节点的发送速率。这种情况，需要继续排查下游节点，一直找到第一个为 OK 的一般就是根源节点。

总体来看，如果我们找到第一个出现反压的节点，反压根源要么是就这个节点，要么是它紧接着的下游节点。

通常来讲，第二种情况更常见。如果无法确定，还需要结合 Metrics 进一步判断。

利用 Metrics 定位

可以根据指标分析反压

可以分析数据传输

④处理反压

反压可能是暂时的，可能是由于负载高峰、CheckPoint 或作业重启引起的数据积压而导致反压。如果反压是暂时的，应该忽略它。

查看是否数据倾斜

使用火焰图分析看顶层的哪个函数占据的宽度最大。只要有"平顶"（plateaus），就表示该函数可能存在性能问题。

分析 GC 日志

4）Flink 有没有遇到过数据倾斜，怎么发现、定位、分析、解决，效果如何

数据倾斜现象

相同 Task 的多个 Subtask 中，个别 Subtask 接收到的数据量明显大于其他 Subtask 接收到的数据量，通过 Flink Web UI 可以精确地看到每个 Subtask 处理了多少数据，即可判断出 Flink 任务是否存在数据倾斜。通常，数据倾斜也会引起反压。

数据倾斜解决

- LocalKeyBy

Flink 是实时流处理，如果 keyby 之后的聚合操作存在数据倾斜，且没有开窗口（没攒批）的情况下，简单的认为使用两阶段聚合，是不能解决问题的。在 keyBy 上游算子数据发送之前，首先在上游算子的本地对数据进行聚合后，再发送到下游，使下游接收到的数据量大大减少，从而使得 keyBy 之后的聚合操作不再是任务的瓶颈。类似 MapReduce 中 Combiner 的思想，但是这要求聚合操作必须是多条数据或者一批数据才能聚合，单条数据没有办法通过聚合来减少数据量。从 Flink LocalKeyBy 实现原理来讲，必然会存在一个积攒批次的过程，在上游算子中必须攒够一定的数据量，对这些数据聚合后再发送到下游。

- keyBy 之前发生数据倾斜

如果 keyBy 之前就存在数据倾斜，上游算子的某些实例可能处理的数据较多，某些实例可能处理的数据较少，产生该情况可能是因为数据源的数据本身就不均匀，例如由于某些原因 Kafka 的 topic 中某些 partition 的数据量较大，某些 partition 的数据量较少。对于不存在 keyBy 的 Flink 任务也会出现该情况。

这种情况，需要让 Flink 任务强制进行 shuffle。使用 shuffle、rebalance 或 rescale 算子即可将数据均匀分配，从而解决数据倾斜的问题。

- keyBy 后的窗口聚合操作存在数据倾斜

第一阶段聚合：key 拼接随机数前缀或后缀，进行 keyby、开窗、聚合

注意：聚合完不再是 WindowedStream，要获取 WindowEnd 作为窗口标记作为第二阶段分组依据，避免不同窗口的结果聚合到一起）

第二阶段聚合：按照原来的 key 及 windowEnd 作 keyby、聚合

1.11 第 11 次考试（准备 15 分钟，考试 25 分钟）

1）Flink 如何保证数据精准一次

source 端： 是 kafka，可重发

flink：开启 checkpoint，设置为精准一次（介绍 checkpoint 原理、barrier 对齐）

sink 端：幂等、事务（2pc），结合 kafka 介绍 2pc 过程

2）你是如何理解 Flink 的 Checkpoint

barriers 在数据流源处被注入并行数据流中。快照 n 的 barriers 被插入的位置（我们称之为 Sn）是快照所包含的数据在数据源中最大位置。

例如，在 Apache Kafka 中，此位置将是分区中最后一条记录的偏移量。将该位置 Sn 报告给 checkpoint 协调器（Flink 的 JobManager）。

然后 barriers 向下游流动。当一个中间操作算子从其所有输入流中收到快照 n 的 barriers 时，它会为快照 n 发出 barriers 进入其所有输出流中。

一旦 sink 操作算子（流式 DAG 的末端）从其所有输入流接收到 barriers n，它就向 checkpoint 协调器确认快照 n 完成。

在所有 sink 确认快照后，意味快照着已完成。一旦完成快照 n，job 将永远不再向数据源请求 Sn 之前的记录，因为此时这些记录（及其后续记录）将已经通过整个数据流拓扑，也即是已经被处理结束。

3）Task 的重启策略

（1）固定延迟重启策略：如果发生故障，系统会尝试重新启动作业 n 次，并在连续重启尝试之间等待 n 秒。

（2）固定延迟重启策略：尝试给定次数重新启动作业，如果超过最大尝试次数，则作业最终会失败。在两次连续重启尝试之间，重启策略等待一段固定的时间。

1.12 第 12 次考试（准备 15 分钟，考试 20 分钟）

1）Flink 有几种 Join ，详细说明特点

```
API: window join
        interval join
        connect + coGroup

SQL: regular join :  默认不清理状态，结合 ttl
               inner :    没有回撤数据
               left、right :  有回撤数据
        interval join :  where  between xxx and xxx
        lookup join （现查外部系统）
        temporal join : 时态表 join（保存多版本）
```

2）你对 keyby 算子的理解

（1）对指定的 key 调用自身的 hashCode 方法=》key.hashcode =》keyHash

（2）调用 murmruhash 算法，进行第二次 hash =》MathUtils.murmurHash(keyHash) % maxParallelism =》keygroupid

（3）计算出当前数据应该去往哪个下游分区：

keyGroupId * parallelism / maxParallelism

（4）键组 id * 下游算子并行度 / 最大并行度（默认 128）

3）你对 Interval Join 的理解

底层调用的是 keyby+connect ，处理逻辑：

（1）判断是否迟到（迟到就不处理了，直接 return）

（2）每条流都存了一个 Map 类型的状态（key 是时间戳，value 是 List 存数据）

（3）任一条流，来了一条数据，遍历对方的 map 状态，能匹配上就发往 join 方法

（4）使用定时器，超过有效时间范围，会删除对应 Map 中的数据（不是 clear，是 remove）

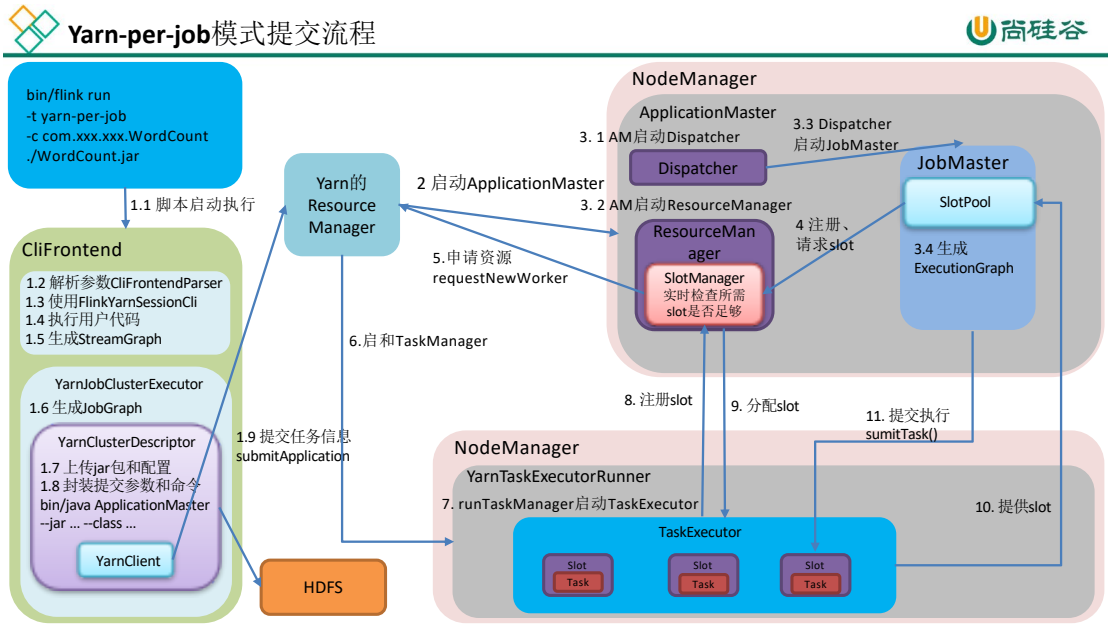
Interval join 不会处理 join 不上的数据，如果需要没 join 上的数据，可以用 coGroup+join 算子实现，或者直接使用 flinksql 里的 left join 或 right join 语法。

1.13 第 13 次考试（准备 20 分钟，考试 20 分钟）

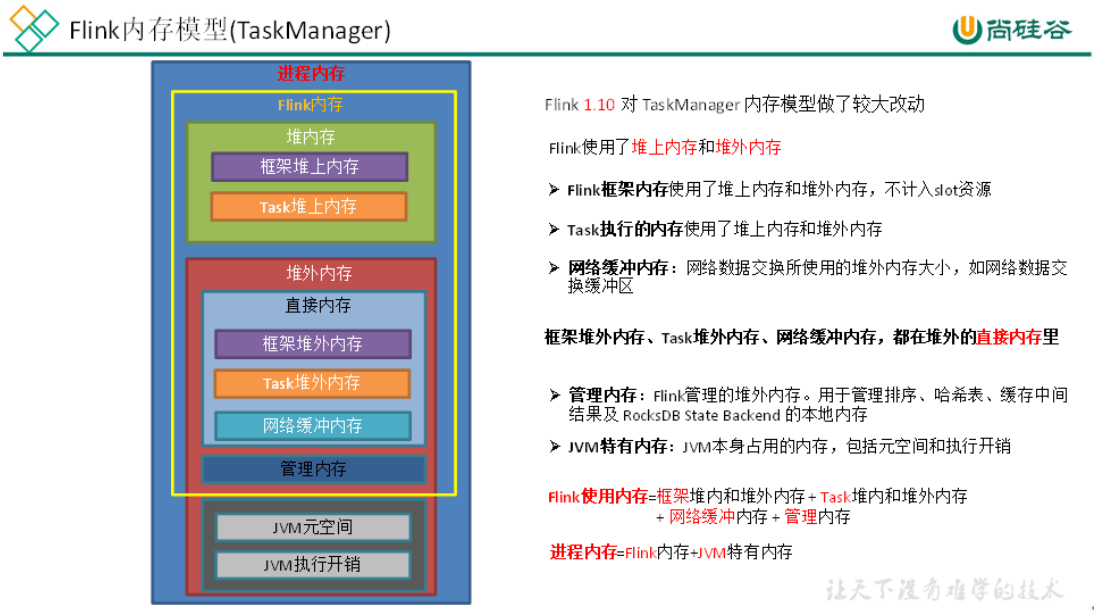
1）Flink 的提交参数如何设置，怎么考虑的

```
flink run -t yarn-per-job
-p    并行度 3，与 kafka 分区保持一致，效率最高
-D    JM 内存，2G，不需要处理数据，不用给太大
-D    TM 内存，4G， 按照 1cu 原则，观察过 webui 的内存 metrics
-D    每个 TM 的 slot 数 3，节省资源， 3 不浪费 slot
-D
-c 全类名
jar 包路径
```

2）Flink 的提交流程

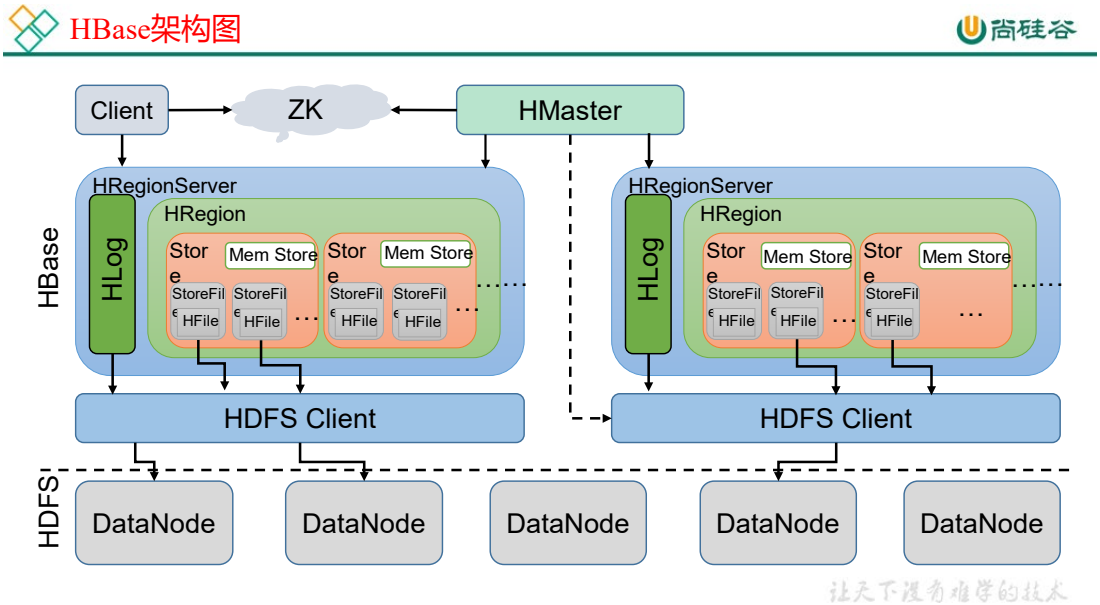


3）Flink 的内存模型

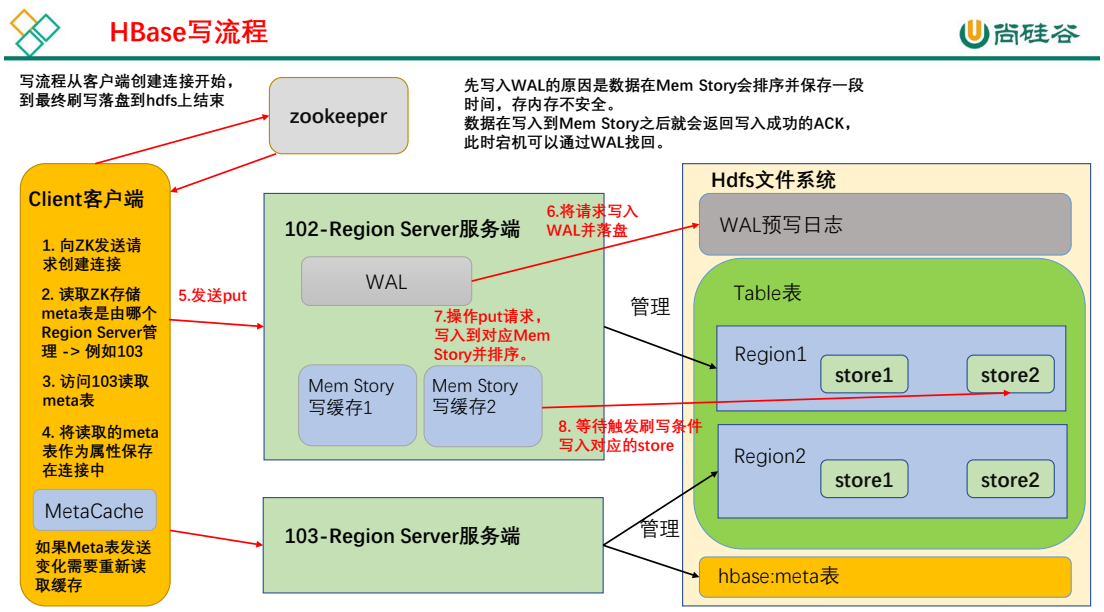


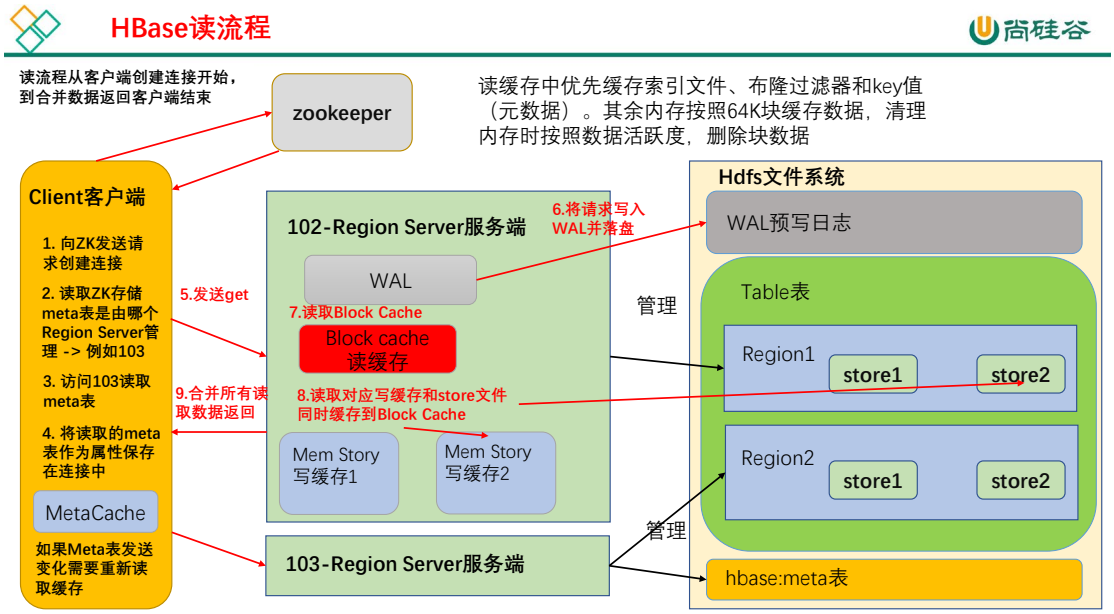
1.14 第 14 次考试（准备 20 分钟，考试 20 分钟）

1）HBase 存储结构



2）HBase 的读、写流程





3）HBase 热点问题如何解决

1）合理设计 RowKey

（1）rowkey 长度原则

（2）rowkey 散列原则

（3）rowkey 唯一原则

2）设置预分区

3）利用 Phoniex 的盐表

1.15 第 15 次考试（准备 20 分钟，考试 20 分钟）

1）Clickhouse 的优势

快： 提供了丰富的表引擎，每个表引擎 都做了尽可能的优化

为什么快：

（1）向量化

（2）列式

（3）尽可能使用本节点的 内存+cpu

（4）提供了 sql 化的语言

（5）支持自定义函数

（6）提供了丰富的表引擎，引擎都经过了优化

2）Flink 写入 Clickhouse 怎么保证一致性？

Clickhouse 没有事务，Flink 写入是至少一次语义。

利用 Clickhouse 的 ReplacingMergeTree 引擎会根据主键去重，但只能保证最终一致性。查询时加上 final 关键字可以保证查询结果的一致性。

3）Clickhouse 的常用引擎

（1）Log

（2）Special：Memory、Distributed

（3）MergeTree： replacingmergetree、summingmergetree

replicatedmergetree

（4）集成引擎： 外部系统映射，如 MySQL

1.16 第 16 次考试（准备 10 分钟，考试 20 分钟）

1）数仓项目建模准备（事实表 、维度表）

1、ER 模型（三范式 增删改查） 维度建模（尽可能减少 join 操作）

2、维度建模

星型模型（事实表周围一级维度） 减少 join 操作

雪花模型（事实表周围多级维度）

星座（多个事实表）

3、事实表

1）判断表是事实表

具有度量值，可以累加的值 （个数、件事、次数、金额）

2）事实表 数据量大 同步策略： 通常 增量 =》 特殊的 加购（周期快照事实表）

3）事务型事实表

原子操作

（1）选择业务过程

（2）声明粒度

（3）确定维度

（4）确定事实

不足

（1）连续性指标 库存 周期型快照事实表

（2）多事实表关联 加购 到支付 平均使用时长 累积型快照事实表

4）周期型快照事实表

5）累积型快照事实表

4、维度表

1）判断表是维度表

没有度量值　性别 身高、体重　年龄　姓名　都是描述信息

2）维度表　数据量小　同步策略：通常　全量　=》 特殊的　用户表 （拉链表）

3）用户表　拉链表

缓慢变化维场景

4）维度整合

减少 join 操作

（1）商品表 + 商品品类表 + 商品 SPU 表 + 一级分类+ 二级分类+ 三级分类=》 商品维度表

（2）省份表 + 地区表 =》 地区维度表

（3）活动信息表 + 活动规则表 =》 活动维度表

2）数仓项目建模（项目调研、数据域、业务矩阵、建模、指标体系建设等）

1）数仓建模（项目调研、数据域、业务矩阵、建模、指标体系建设等）

1、数据调研 （50%）

1）要原始的表　表的字段什么含义　要需求

快速熟悉业务　网站　小程序　安卓　ios

过滤掉　图片　权限　临时表　废弃的功能

2）和业务人员聊

3）和产品经理聊

需求：　原子指标　派生指标　衍生指标

派生指标=原子指标（业务过程 + 度量值+ 聚合逻辑） + 统计周期 + 业务限定 + 统计粒度

2、明确数据域

1）用户域　注册　登录

2）流量域：　启动、页面、动作、故障、曝光

3）交易域 ：　加购、下单 支付、物流、退单、退支付

4）工具域：　领取优惠卷、使用优惠卷下单、使用优惠卷支付

5）互动域：评论、点赞、收藏

3、构建业务矩阵

用户、商品（6 张整合）、活动（2 张整合）、优惠卷、地区（2 张表整合）、时间

1）用户域

注册　　√

登录

2）流量域

启动

页面

动作　　√

故障

曝光

3）交易域

加购

下单

支付

物流

退单

退支付

4）工具域

领取优惠卷

使用优惠卷下单

使用优惠卷支付

5）互动域

评论

点赞

收藏

4、维度建模

1）ODS 层

（1）保持数据原貌不做任何修改　备份

（2）采用压缩　减少磁盘空间

（3）创建分区表　防止后续全表扫描

2）DIM 层

（1）用户表　拉链表

缓慢变化维场景

（2）维度整合

减少 join 操作

商品表 + 商品品类表 + 商品 SPU 表 + 一级分类+ 二级分类 + 三级分类=》 商品维度表

省份表 + 地区表 =》 地区维度表

活动信息表 + 活动规则表 =》 活动维度表

（3）ETL　数据清洗

3）DWD 层

（1）事务型事实表

找的原子操作

①选择业务过程

选择感兴趣的　产品经理提出来的

过滤掉　图片　权限　临时表　废弃的功能

②声明粒度

粒度：一行信息代表什么含义　可以代表一次下单　一周下单

保持最小粒度

不做聚合操作

③确定维度

产品经理需要的　用户、商品、活动、优惠卷、地区、时间

④确定事实

确定事实表的度量值 可以累加的值 （个数、次数、件事、金额）

（2）周期快照事实表 （加购、连续）

①选择业务过程

②声明粒度 （按天、小时）

③确定维度

④确定事实

（3）累积型快照事实表（多事实表关联）

①选择业务过程

②声明粒度

③确定维度

④确定事实 （确定多张事实表的事实 ）

（4）ETL 数据清洗

5、指标体系建设

ads 层

日活、月活、周活、总用户、日新增、周新增、月新增、1 日-60 日留存、转化率（漏斗）、GMV（日、周、月）

7 天内连续三天（点赞、收藏、评价、登录） 统计同时在线人数

dws 层

派生指标 = 原子指标（业务过程 + 度量值 + 聚合逻辑） + 统计周期 + 业务限定 + 统计粒度

交易总额（下单 + 金额 + sum ） + 每天 + 手机 + 省份

统计 每天各个省份手机的交易总额

找公共的业务过程 + 统计周期 + 统计粒度 =》 建 dws 层宽表

第 2 章 项目复习

2.1 第 17 次考试（准备 30 分钟，考试 1 小时）

1）离线数仓项目串讲

2.2 第 18 次考试（准备 30 分钟，考试 1 小时）

1）实时数仓项目串讲