# 一、Hadoop

## 1.Hadoop常用端口号

| | hadoop2.x | hadoop3.x |
|---|---|---|
| 访问 HDFS 端口 | 50070 | 9870 |
| 访问 MR 执行情况端口 | 8088 | 8088 |
| 历史服务器 | 19888 | 19888 |
| 客户端访问集群端口 | 9000 | 8020 |

## 2.Hadoop常用命令

**HDFS常用命令:**

```
#创建指定路径文件夹
hdfs dfs -mkdir /user/zhaojuanjuan/syuchen_files
#查看指定目录下的文件
hdfs dfs -ls /path
#查看指定目录下每一个文件夹的大小
hadoop fs -cat /xxxx/xxx.gz | gzip -d
#查看gz文件内容前几行
hadoop fs -cat /xxxx/xxx.gz | gzip -d | head -10
#移动文件
hdfs dfs -mv 源文件路径 目标路径
#删除hdfs文件夹
hdfs dfs -rm -r /path
#直接删除，不走回收站
hdfs dfs -rm -r -f -skipTrash /path
#上传文件
hdfs dfs -put /本地路径 /路径
#正则匹配OriginalFilePath目录下的文件（夹），批量上传到hdfs的targetFilePath目录下。其中-E表示告诉grep后面是一个正则表达式
ls /OriginalFilePath | grep -E "2020-09.*" | xargs -i hdfs dfs -put /OriginalFilePath/{} /targetFilePath
#下载文件到本地
hdfs dfs -get /hdfs路径 /本地路径
#检查当前是否处于安全模式
hdfs dfsadmin -safemode get
#离开安全模式。一般都是因为空间满了就自动进入安全模式了，此时读写数据会报错
hdfs dfsadmin -safemode leave
```

**yarn常用命令:**

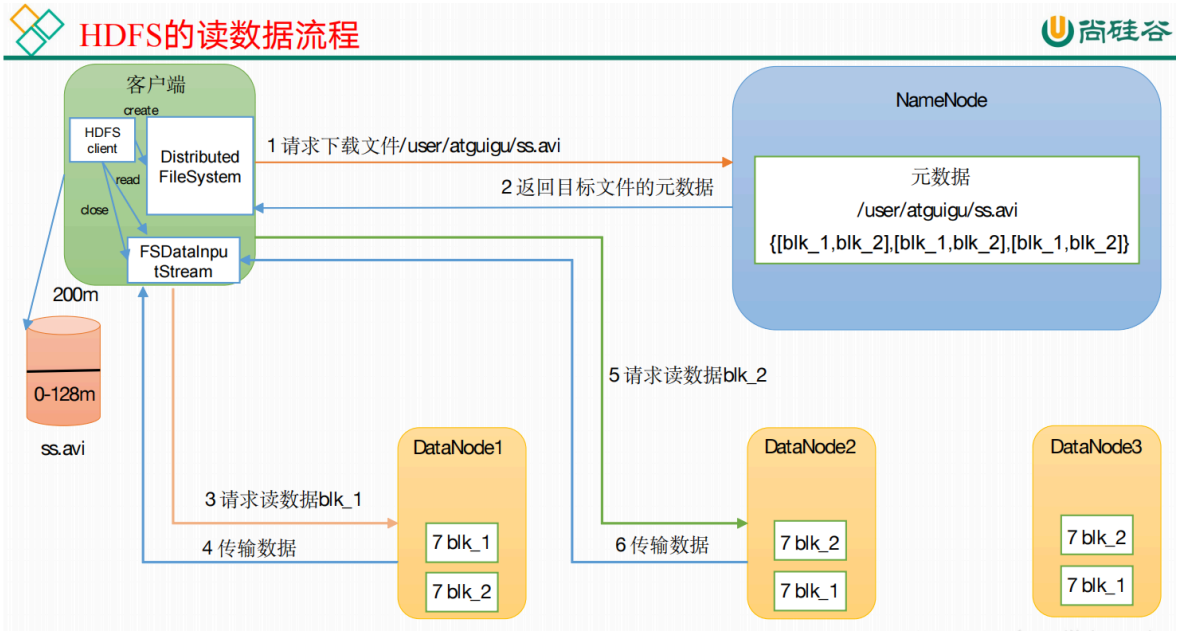
```
#查看yarn集群中正在运行的应用，可以看到各个应用的执行状态和进度（progress可能不准确，一直为10%）
yarn application -list
#根据应用id强行终止指定应用
yarn application -kill appid
#查看所有处于running状态的节点
yarn node -list
#查看所有节点
yarn node -list -all
```

## 3.Hadoop3.x相对于Hadoop2.x的新特性

1、最低要求Java版本从7增加到Java8
2、支持HDFS的纠删码（一种持久存储数据的方法，利用计算来关联存储文件，节省大量空间。用于存储较冷，访问频率较低的数据）
3、shell脚本重写兼容
4、MapReduce任务本地优化，增加了对 map 输出收集器的本地执行的支持，对于 shuffle 密集型工作，这可以使性能提高30%或更多
5、支持两个以上的NameNode
6、多个服务的默认端口被更改
7、支持Microsoft Azure数据湖和阿里云对象存储系统文件系统连接器
8、数据内节点平衡器
9、基于HDFS路由器的联合，简化了现有HDFS客户端对联合群集的访问
10、YARN资源类型通用化

# 1、HDFS基础

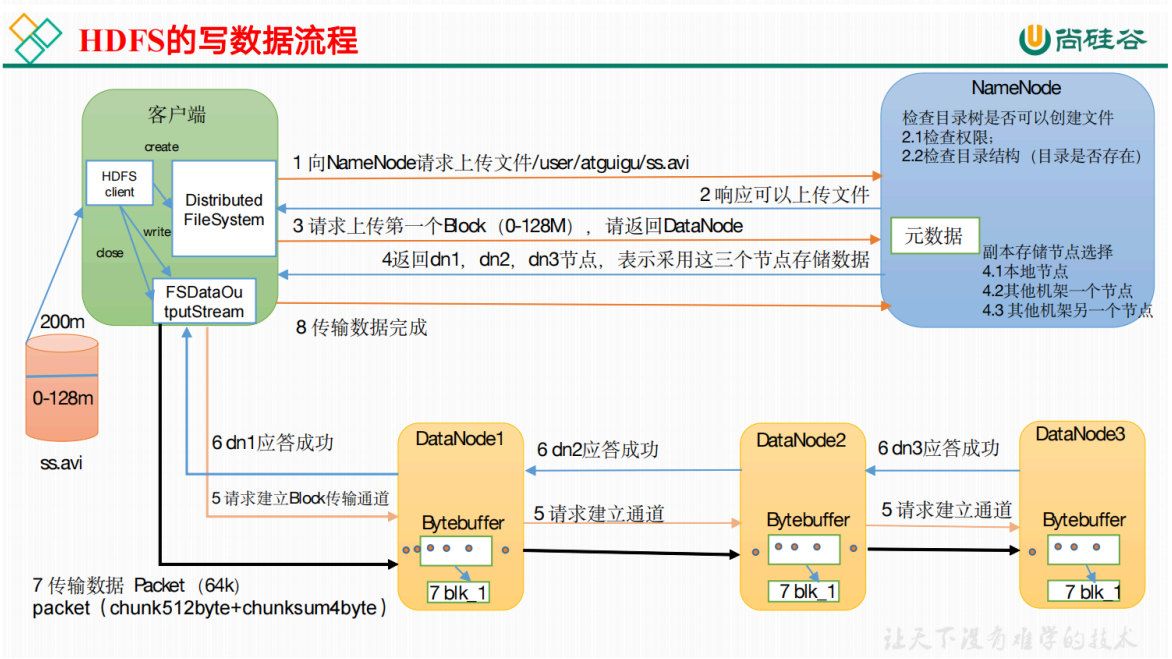
## 1.HDFS读流程和写流程




**详细步骤：**

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 DataNode 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 DataNode 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；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；
5、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；
6、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。
7、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；
8、最终读取来所有的 block 会合并成一个完整的最终文件。




**详细步骤：**

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 DataNode 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 DataNode 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；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；
5、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；
6、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。
7、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；
8、最终读取来所有的 block 会合并成一个完整的最终文件。

## 2.HDFS小文件处理

存储层面：HDFS 设计初衷是为了存储大文件，通常文件块大小为 128MB 或 256MB。每个文件（或块）的元数据会存储在 NameNode 的内存中，大约每个文件或块需要 150字节的内存。如果存在大量小文件，NameNode 内存中存储的元数据数量会急剧增加，从而造成内存占用过高，甚至引发垃圾回收频繁、NameNode 重启变慢等问题

> 128G 能存储多少文件块?
>
> 128 g*1024m*1024kb*1024byte/150 字节 = 9.1 亿文件块

计算层面：每个小文件都会起到一个 MapTask，1 个 MapTask 默认内存 1G。浪费资源。

**如何解决：**

（1）采用 HAR（Hadoop Archive） 归档方式，将多个小文件打包成一个归档文件，从而减少 NameNode 维护的文件数量
（2）采用 CombineTextInputFormat，将多个小文件合并成一个 Split，从而减少 Map 任务数量，降低任务调度和启动的开销
（3）自己写一个 MR 程序将产生的小文件合并成一个大文件。如果是 Hive 或者 Spark有 merge 功能自动帮助我们合并。
（4）有小文件场景开启 JVM 重用；如果没有小文件，不要开启 JVM 重用，因为会一直占用使用到的 Task 卡槽，直到任务完成才释放。JVM 重用可以使得 JVM 实例在同一个 job 中重新使用 N 次，N 的值可以在 Hadoop 的mapred-site.xml 文件中进行配置`mapreduce.job.jvm.numtasks`参数。通常在 10-20 之间。

## 3.HDFS NameNode内存

HDFS 中的 NameNode 主要负责管理整个文件系统的元数据，这部分元数据主要包括两个方面：

**Namespace（目录树）**：存储文件与目录的名称、权限、所属用户、修改时间、以及目录树结构等信息。

**BlocksMap（块映射）**：记录每个数据块及其副本所在的 DataNode 列表，保证能够快速定位文件数据在集群中的物理位置。

> 1）Hadoop2.x 系列，配置 NameNode 默认 2000m
>
> 2）Hadoop3.x 系列，配置 NameNode 内存是动态分配的
>
> NameNode 内存最小值 1G，每增加 100 万个文件 block，增加 1G 内存。

NameNode内存优化措施：

- 合理规划目录结构和合并小文件，减少 INodes 数量；
- 调整 JVM 参数、增加堆内存、优化 GC 策略；
- 部署 HDFS Federation联盟，将单个 NameNode 的压力分散到多个 NameNode 上。

## 4.HDFS的数据压缩算法

Hadoop中常用的压缩算法有**bzip2、gzip、lzo、snappy**，其中lzo、snappy需要操作系统安装native库才可以支持。

# 2、YARN基础

**集群架构和工作原理：**

YARN 是 Hadoop 的资源管理与作业调度框架，其核心组件包括 ResourceManager、NodeManager 和 ApplicationMaster。RM 负责全局资源分配，NM 负责节点资源监控和 Container 启动（负责该节点内所有容器的生命周期的管理），AM 则针对每个应用负责任务调度和容错管理。Container是Yarn中的资源抽象，一个应用程序会分配一个Container，**这个应用程序只能使用这个Container中描述的资源**

- ResourceManager（RM）：

  `ResourceManager` 通常在独立的机器上以后台进程的形式运行，它是整个集群资源的主要协调者和管理者。`ResourceManager` 负责给用户提交的所有应用程序分配资源，它根据应用程序优先级、队列容量、ACLs、数据位置等信息，做出决策，然后以共享的、安全的、多租户的方式制定分配策略，调度集群资源。

- NodeManager（NM）：

  `NodeManager` 是 YARN 集群中的每个具体节点的管理者。主要负责该节点内所有容器的生命周期的管理，监视资源和跟踪节点健康。具体如下：

  - 启动时向 `ResourceManager` 注册并定时发送心跳消息，等待 `ResourceManager` 的指令；
  - 维护 `Container` 的生命周期，监控 `Container` 的资源使用情况；
  - 管理任务运行时的相关依赖，根据 `ApplicationMaster` 的需要，在启动 `Container` 之前将需要的程序及其依赖拷贝到本地。

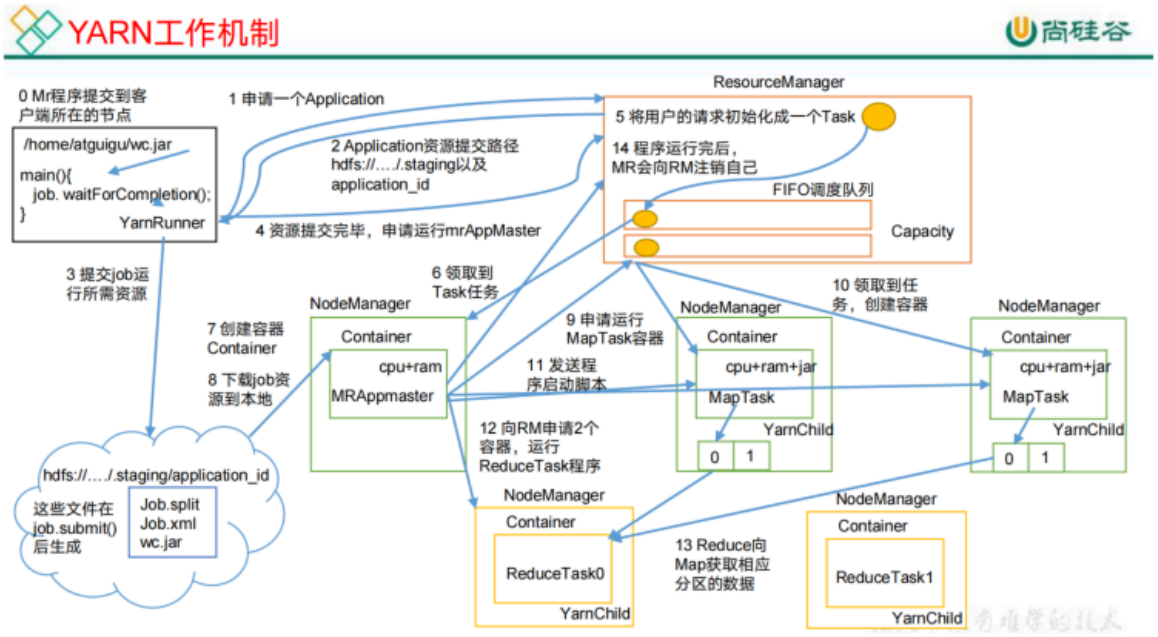
- ApplicationMaster（AM）：

  在用户提交一个应用程序时，YARN 会启动一个轻量级的进程 `ApplicationMaster`。`ApplicationMaster` 负责协调来自 `ResourceManager` 的资源，并通过 `NodeManager` 监视容器内资源的使用情况，同时还负责任务的监控与容错。具体如下：

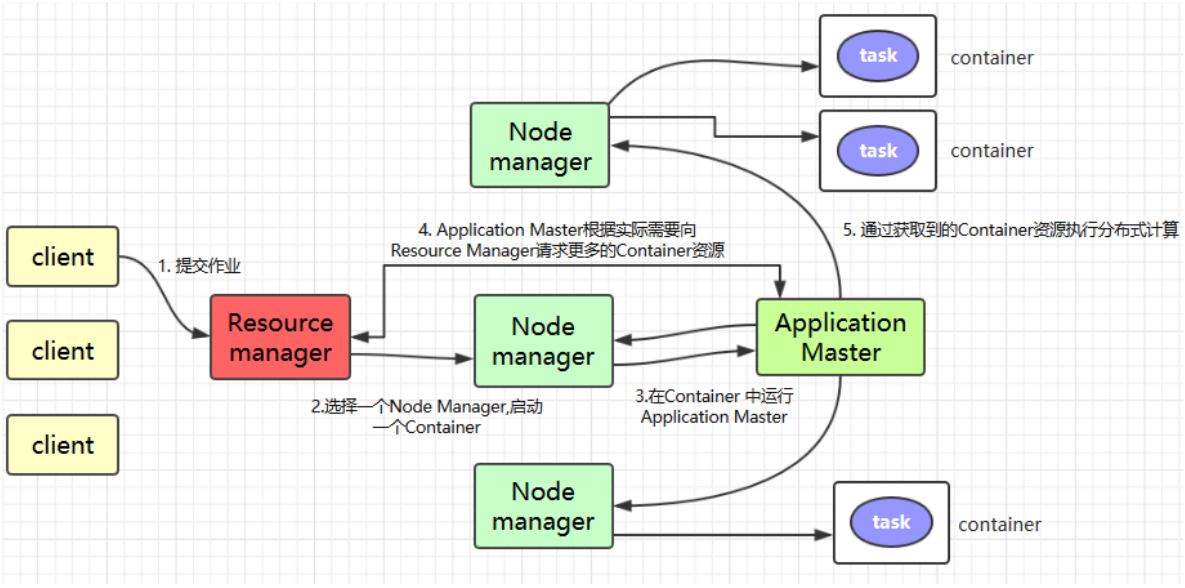
  - 根据应用的运行状态来决定动态计算资源需求；
  - 向 `ResourceManager` 申请资源，监控申请的资源的使用情况；
  - 跟踪任务状态和进度，报告资源的使用情况和应用的进度信息；
  - 负责任务的容错。

- Container：

  `Container` 是 YARN 中的资源抽象，它封装了某个节点上的多维度资源，如内存、CPU、磁盘、网络等。当 AM 向 RM 申请资源时，RM 为 AM 返回的资源是用 `Container` 表示的。YARN 会为每个任务分配一个 `Container`，该任务只能使用该 `Container` 中描述的资源。`ApplicationMaster` 可在 `Container` 内运行任何类型的任务。例如，`MapReduce ApplicationMaster` 请求一个容器来启动 map 或 reduce 任务，而 `Giraph ApplicationMaster` 请求一个容器来运行 Giraph 任务。




# 1.Yarn工作原理简述




1. `Client` 提交作业到 YARN 上；
2. `Resource Manager` 选择一个 `Node Manager`，启动一个 `Container` 并运行 `Application Master` 实例；
3. `Application Master` 根据实际需要向 `Resource Manager` 请求更多的 `Container` 资源（如果作业很小, 应用管理器会选择在其自己的 JVM 中运行任务）；
4. `Application Master` 通过获取到的 `Container` 资源执行分布式计算。

# 2.Yarn工作原理详述

### 1. 作业提交

client 调用 job.waitForCompletion 方法，向整个集群提交 MapReduce 作业 (第 1 步) 。新的作业 ID(应用 ID) 由资源管理器分配 (第 2 步)。作业的 client 核实作业的输出, 计算输入的 split, 将作业的资源 (包括 Jar 包，配置文件, split 信息) 拷贝给 HDFS(第 3 步)。 最后, 通过调用资源管理器的 submitApplication() 来提交作业 (第 4 步)。

### 2. 作业初始化

当资源管理器RM收到 submitApplciation() 的请求时, 就将该请求发给调度器 (scheduler), 调度器分配 container, 然后资源管理器在该 container 内启动应用管理器进程, 由节点管理器监控 (第 5 步)。

MapReduce 作业的应用管理器是一个主类为 MRAppMaster 的 Java 应用，其通过创造一些 bookkeeping 对象来监控作业的进度, 得到任务的进度和完成报告 (第 6 步)。然后其通过分布式文件系统得到由客户端计算好的输入 split(第 7 步)，然后为每个输入 split 创建一个 map 任务, 根据 mapreduce.job.reduces 创建 reduce 任务对象。

**3. 任务分配**

如果作业很小, 应用管理器会选择在其自己的 JVM 中运行任务。

如果不是小作业, 那么应用管理器向资源管理器请求 container 来运行所有的 map 和 reduce 任务 (第 8 步)。这些请求是通过心跳来传输的, 包括每个 map 任务的数据位置，比如存放输入 split 的主机名和机架 (rack)，调度器利用这些信息来调度任务，尽量将任务分配给存储数据的节点, 或者分配给和存放输入 split 的节点相同机架的节点。

**4. 任务运行**

当一个任务由资源管理器的调度器分配给一个 container 后，应用管理器通过联系节点管理器来启动 container(第 9 步)。任务由一个主类为 YarnChild 的 Java 应用执行， 在运行任务之前首先本地化任务需要的资源，比如作业配置，JAR 文件, 以及分布式缓存的所有文件 (第 10 步。 最后, 运行 map 或 reduce 任务 (第 11 步)。

YarnChild 运行在一个专用的 JVM 中, 但是 YARN 不支持 JVM 重用。

**5. 进度和状态更新**

YARN 中的任务将其进度和状态 (包括 counter) 返回给应用管理器, 客户端每秒 (通 mapreduce.client.progressmonitor.pollinterval 设置) 向应用管理器请求进度更新, 展示给用户。

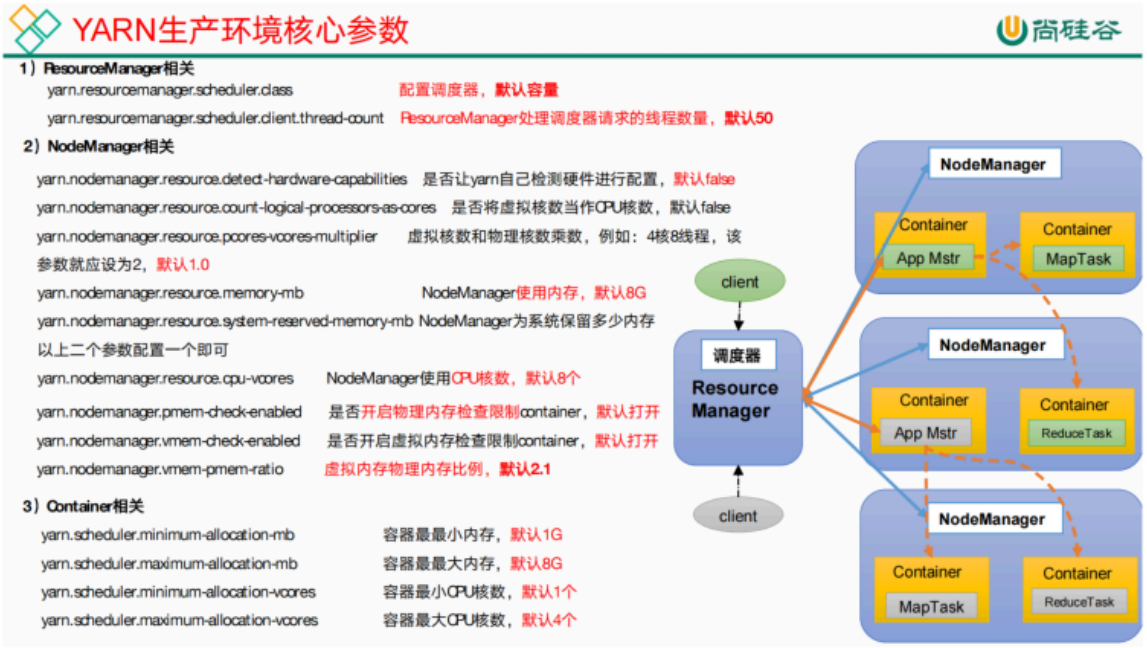
**6. 作业完成**

除了向应用管理器请求作业进度外, 客户端每 5 分钟都会通过调用 waitForCompletion() 来检查作业是否完成，时间间隔可以通过 mapreduce.client.completion.pollinterval 来设置。作业完成之后, 应用管理器和 container 会清理工作状态， OutputCommiter 的作业清理方法也会被调用。作业的信息会被作业历史服务器存储以备之后用户核查。

# 3. YARN 的运行流程（简要）

1. 用户提交作业到 RM。
2. RM 为作业启动一个 AM。
3. AM 向 RM 申请 Container 资源。
4. RM 分配 Container。
5. NM 启动 Container 并运行任务。
6. 任务执行完毕，AM 向 RM 汇报状态。
7. 作业完成，AM 退出。

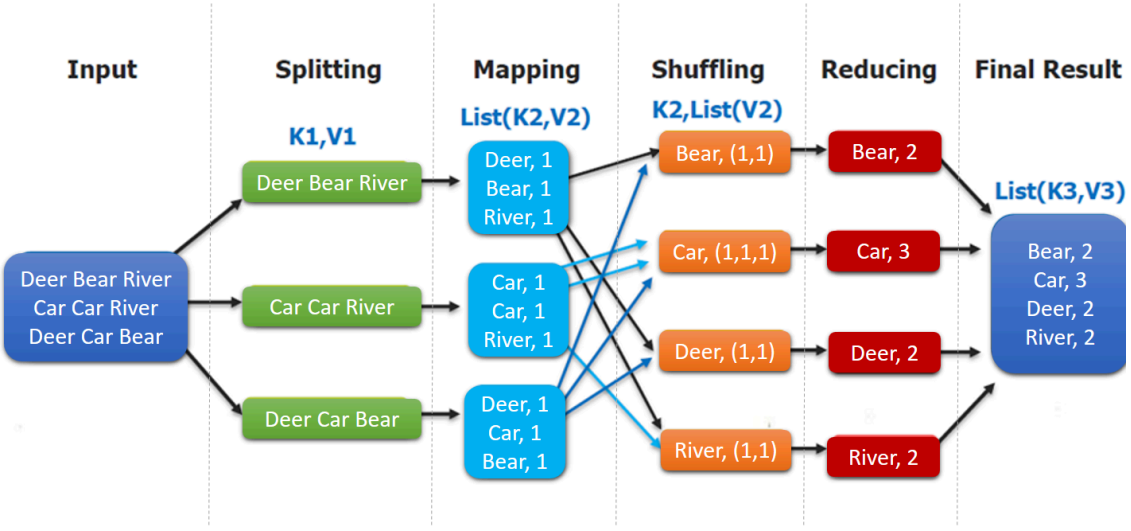
## 4.Yarn生产环境核心参数




# 3、MapReduce基础

Hadoop MapReduce 是一个分布式计算框架，用于编写批处理应用程序。编写好的程序可以提交到 Hadoop 集群上用于并行处理大规模的数据集。

MapReduce 作业通过将输入的数据集拆分为独立的块，这些块由 `map` 以并行的方式处理，框架对 `map` 的输出进行排序，然后输入到 `reduce` 中。MapReduce 框架专门用于 `<key，value>` 键值对处理，它将作业的输入视为一组 `<key，value>` 对，并生成一组 `<key，value>` 对作为输出。输入和输出的 `key` 和 `value` 都必须实现Writable 接口

> 这里以词频统计为例进行说明，MapReduce 处理的流程如下：
>
> 
> 
>
> 1. **input** : 读取文本文件；
> 2. **splitting** : 将文件按照行进行拆分，此时得到的 `K1` 行数，`V1` 表示对应行的文本内容；
> 3. **mapping** : 并行将每一行按照空格进行拆分，拆分得到的 `List(K2,V2)`，其中 `K2` 代表每一个单词，由于是做词频统计，所以 `V2` 的值为 1，代表出现 1 次；

> 4. **shuffling**：由于 `Mapping` 操作可能是在不同的机器上并行处理的，所以需要通过 `shuffling` 将相同 `key` 值的数据分发到同一个节点上去合并，这样才能统计出最终的结果，此时得到 `K2` 为每一个单词，`List(V2)` 为可迭代集合，`V2` 就是 Mapping 中的 V2；
> 5. **Reducing** : 这里的案例是统计单词出现的总次数，所以 `Reducing` 对 `List(V2)` 进行归约求和操作，最终输出。
>
> MapReduce 编程模型中 `splitting` 和 `shuffing` 操作都是由框架实现的，需要我们自己编程实现的只有 `mapping` 和 `reducing`，这也就是 MapReduce 这个称呼的来源。

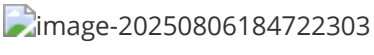
# 1.基本原理

MapReduce 是一种分布式计算模型，主要思想是“先分后合”（分而治之）：

- **Map 阶段**：将大规模数据分割成若干小块，由多个 Mapper 并行处理，每个 Mapper 依据自定义的 Map 函数将输入数据转换为中间键值对。
- **Shuffle 阶段**：将所有 Mapper 输出的中间数据按照键进行分组、排序，并将相同 Key 的数据传递给同一 Reducer。
- **Reduce 阶段**：Reducer 对同一 Key 对应的所有值进行聚合计算，生成最终输出结果。

这种模型的优势在于利用大量计算资源进行并行处理，同时具有较好的扩展性和容错性，但对实时计算和小数据处理不够友好。

# 2.工作流程概述



**输入分片（Input Split）**

- 在任务开始前，将大文件按预设的块大小（通常 64MB 或 128MB）切分成若干逻辑片段，每个片段交由一个 Map Task 处理。

**Map 阶段**

- 每个 Map Task 读取对应输入分片，通过 RecordReader 按行或其它单位解析数据；
- 调用用户自定义的 Map 函数，把每条记录转换成一对中间键值对；
- 将生成的键值对先存入内存中的环形缓冲区，并在缓冲区达到一定阈值时进行本地排序、溢写到磁盘；
- 如果配置了 Combiner，还会在 Map 端做局部预聚合，减少网络传输数据量。

**Shuffle 阶段**

- Reduce Task 通过网络从各个 Map Task 拉取数据；
- 数据在 Reduce 端进行合并、归并排序，保证同一 Key 的所有数据连续排列。

**Reduce 阶段**

- Reduce Task 针对每个唯一 Key 调用 Reduce 函数，对对应的值集合进行计算（例如累加、聚合等），生成最终输出；
- 最终结果写入 HDFS 或其它存储系统。

# 3.MapReduce的核心概念

### 2.1 Combiner

- Combiner 是一种可选的“局部 Reducer”，在 Map Task 内部对中间数据进行预聚合，减少数据传输量；
- 注意：Combiner 的执行不是强制的，且其逻辑必须满足交换律和结合律，确保不会影响最终计算结果。

### 2.2 分区器（Partitioner）

- 负责将 Mapper 输出的键值对分配到不同的 Reducer，通常采用默认的哈希分区方式（即对Key的哈希值取模）；
- 自定义分区器可以根据业务需要调整数据分布，防止部分 Reducer 处理数据过多（数据倾斜）。

### 2.3 Shuffle 过程

- Shuffle 是 Map 和 Reduce 两个阶段之间的桥梁，它负责对中间数据进行分区、排序和合并，确保 Reducer 接收到的数据按 Key 分组排列；
- 该过程往往是性能瓶颈之一，优化方法包括压缩传输数据、调整缓冲区大小等。

### 2.4 数据倾斜

- 数据倾斜指某些 Key 出现频率异常高，导致对应 Reducer 任务处理的数据远多于其他任务，从而拖慢整个作业进度；
- 解决方法包括：
  - 自定义分区器，把高频 Key 分散到多个 Reducer 上；
  - 使用 Combiner 在 Map 端预聚合；
  - 合理设置 Reducer 数量，或对高频数据做特殊处理。

# 4.MapReduce进程

一个完整的 MapReduce 程序在分布式运行时有三类实例进程：

（1）**MrAppMaster**：负责整个程序的过程调度及状态协调。

（2）**MapTask**：负责 Map 阶段的整个数据处理流程。

（3）**ReduceTask**：负责 Reduce 阶段的整个数据处理流程。

# 4、HDFS相关面试题

## 1. HDFS 的基本架构是什么?

**解析：**
HDFS（Hadoop Distributed File System）是一个分布式文件系统，主要由以下几个关键组件构成：

- **NameNode**：主节点，负责管理文件系统的元数据（如目录结构、文件与数据块的映射等）。
- **DataNode**：从节点，负责存储实际数据块，并定期向 NameNode 发送心跳信息。
- **Secondary NameNode**（或 Standby/Backup NameNode，在 HA 场景下则有 ZKFC 等组件）：用于辅助 NameNode 进行元数据的检查点（Checkpoint），并在 NameNode 重启时加快恢复过程。

- **Client**：客户端通过 RPC 与 NameNode 交互，获取数据块信息，再直接与 DataNode 通信完成数据的读写。

## 2. HDFS 的默认块大小是多少？如何配置?

**解析：**

- 默认块大小在早期版本通常为 64MB，Hadoop 2.x 及之后的版本通常默认设置为 128MB，部分集群会根据业务需求设置为 256MB 或更大。
- 块大小通过配置参数 `dfs.blocksize` 来设置，可以在 `hdfs-site.xml` 中进行调整。

## 3. HDFS 如何实现数据的高容错性?

**解析：**

- **副本机制**：HDFS 默认将每个数据块复制 3 份（通过参数 `dfs.replication` 配置），将副本分布在不同的 DataNode 上，其中通常在同一机架上存两份，另一份存放在不同机架上，以应对单点故障。
- **心跳机制和 BlockReport**：DataNode 定期向 NameNode 发送心跳及块报告，确保 NameNode 能够及时发现失效节点，并根据策略重新复制数据块。

## 4. HDFS 中数据块的副本机制是如何保证数据可靠性的?

**解析：**

- 每个数据块在 HDFS 中默认有 3 个副本。NameNode 根据心跳和 BlockReport 信息来监控副本状态；如果某个副本丢失，会触发复制任务，在其他 DataNode 上重新复制该数据块。
- 副本分布算法考虑了网络拓扑，尽可能保证副本分布在不同机架上，从而提高容错能力。
- 即多副本机制、定期块扫描、NameNode元数据管理（协调修复）

## 5. HDFS 的 NameNode 如何实现高可用性（HA）？

**解析：**

- 通过**HDFS Federation** 或**双 NameNode 模式**实现高可用。
- 在 HA 模式下，通常会部署一个主 NameNode和一个备用 NameNode（Standby NameNode），通过 Zookeeper 进行选举和故障切换；当主 NameNode 出现故障时，备用 NameNode 自动接管服务，保证整个集群的元数据管理不中断。

## 6.HDFS 与传统文件系统相比有哪些优缺点?

**优点：**

- 针对大文件设计，具备高吞吐量；
- 具备高容错性，通过副本机制保证数据安全；
- 适合流式数据访问，支持分布式计算框架。

**缺点：**

- 不适合大量小文件存储，会带来 NameNode 内存压力；
- 不支持随机写入，仅支持写一次读多次；
- 元数据存储在内存中，扩展性受到一定限制。

## 7. HDFS 如何保证数据的可扩展性?

**解析:**

- **数据块分布**：大文件会被分割成多个块分布存储在众多 DataNode 上；
- **集群扩展**：通过增加 DataNode 数量扩展存储容量；
- **Federation**：引入 HDFS Federation 后，可以部署多个 NameNode，每个 NameNode 管理部分命名空间，从而突破单点扩展的限制；
- **纠删码**：在 Hadoop 3.x 中，纠删码技术（Erasure Coding）可以在保证数据可靠性的前提下减少存储冗余。

## 8. HDFS中谈谈你对其中性能瓶颈的认识，以及可能的优化方向。

**解析:**

- **性能瓶颈**可能出现在 NameNode 内存瓶颈、网络传输、以及大量小文件的管理上。
- 优化方向包括:
  - 合理规划文件大小，避免大量小文件；
  - 优化 NameNode 内存配置和 JVM 参数；
  - 使用 HDFS Federation 或纠删码等技术改善扩展性和存储效率；
  - 优化数据本地性，减少跨机架数据传输。

## 9. HDFS其他常见问题

- **HDFS 中的 BlockReport 有什么作用?**
  DataNode 定期发送 BlockReport，向 NameNode 汇报本地存储的所有数据块信息，以便 NameNode 更新 BlocksMap 并监控数据副本状态。
- **如何监控 HDFS 的健康状态?**
  通过 HDFS Web UI、JMX、以及第三方监控工具（如 Cloudera Manager、Hortonworks Ambari 等）监控 NameNode 内存、GC 日志、心跳延迟以及网络 I/O 等指标。
- **NameNode 的重启时间为何随着元数据规模增加而延长?**
  重启时 NameNode 需要加载 FsImage 和回放 editlog，元数据越多，加载和回放时间就越长，因此重启时间随数据规模呈线性增长。

## 10.HDFS读取文件时，其中一个块突然损坏怎么办?

> 在HDFS中，文件被分割成多个块（Block），每个块默认有3个副本存储在不同的DataNode上

如果客户端在读取某个块时发现该块损坏（例如校验和不匹配或无法读取），HDFS会采取以下步骤:

1. **校验和验证**:
   - HDFS为每个块存储了**校验和（Checksum）**，客户端读取数据时会验证校验和。
   - 如果校验和不匹配，说明数据损坏，客户端会标记该块为损坏。
2. **尝试读取其他副本**:
   - 客户端会从NameNode获取该块的其他副本位置，并尝试从其他DataNode读取数据。
   - 由于默认有3个副本，客户端可以快速切换到其他副本继续读取。
3. **报告损坏块**:
   - 客户端会向NameNode报告损坏的块及其所在的DataNode。

    - NameNode会标记该块为损坏，并触发数据修复机制。
4. **数据修复**：
    - NameNode会检查该块的副本数量是否满足配置要求（默认3个）。
    - 如果副本数量不足，NameNode会选择一个健康的DataNode，从其他副本复制数据，生成新的副本。

# 11.HDFS上传文件时，其中一个DataNode突然挂掉怎么办?

客户端上传文件时与DataNode建立pipeline管道，管道的正方向是客户端向DataNode发送的数据包，管道反向是DataNode向客户端发送ack确认，也就是正确接收到数据包之后发送一个已确认接收到的应答。

当DataNode突然挂掉了，客户端接收不到这个DataNode发送的ack确认，客户端会通知NameNode，NameNode检查该块的副本与规定的不符，NameNode会通知DataNode去复制副本，并将挂掉的DataNode作下线处理，不再让它参与文件上传与下载。

# 12.HDFS NameNode启动过程?

NameNode数据存储在内存和本地磁盘，本地磁盘数据存储在**fsimage镜像文件和edits编辑日志文件**。

- 首次启动NameNode：

  1、**格式化文件系统，为了生成fsimage镜像文件**
  2、启动NameNode：
  　1）读取fsimage文件，将文件内容加载进内存；
  　2）等待DataNade注册与发送block report；
  3、启动DataNode：
  　1）向NameNode注册
  　2）发送block report
  　3）检查fsimage中记录的块的数量和block report中的块的总数是否相同
  4、对文件系统进行操作（创建目录，上传文件，删除文件等）：
  　此时内存中已经有文件系统改变的信息，但是磁盘中没有文件系统改变的信息，此时会将这些改变信息写入edits文件中，edits文件中存储的是文件系统元数据改变的信息。

- 第二次启动NameNode：

  1、读取fsimage和edits文件；
  2、将fsimage和edits文件合并成新的fsimage文件；
  3、创建新的edits文件，内容开始为空；
  4、启动DataNode。

# 13.Hadoop脑裂原因及解决办法?

**1）出现脑裂的原因**

Leader 出现故障，系统开始改朝换代，当 Follower 完成全部工作并且成为 Leader 后，原 Leader 又复活了（它的故障可能是暂时断开或系统暂时变慢，不能及时响应，但其NameNode 进程还在），并且由于某种原因它对应的 ZKFC 并没有把它设置为 Standby，所以原 Leader 还认为自己是 Leader，客户端向它发出的请求仍会响应，于是脑裂就发生了。

**2）Hadoop通常不会出现脑裂。**

如果出现脑裂，意味着多个 Namenode 数据不一致，此时只能选择保留其中一个的数据。

例如：现在有三台 Namenode，分别为 nn1、nn2、nn3，出现脑裂，想要保留 nn1 的数据

步骤为：
（1）关闭 nn2 和 nn3
（2）在 nn2 和 nn3 节点重新执行数据同步命令：hdfs namenode -bootstrapStandby
（3）重新启动 nn2 和 nn3

## 14.你是如何解决NameNode宕机?

未启用高可用HA时：
1、通过监控工具（Zabbix、Prometheus）或日志排查错误原因
2、重启NameNode进程，从FsImage（文件系统镜像）和EditLog（操作日志）中恢复元数据，定期备份FsImage和EditLog，定期合并其中数据
3、通过命令验证集群状态（hdfs dfsadmin -report），确保所有DataNode重新注册，数据块完整

启用了高可用HA时：
NameNode宕机会自动切换至Standby节点，服务中断时间较短。

## 15.HDFS特点?

**高容错**：由于 HDFS 采用数据的多副本方案，所以部分硬件的损坏不会导致全部数据的丢失。

**高吞吐量**：HDFS 设计的重点是支持高吞吐量的数据访问，而不是低延迟的数据访问。

**大文件支持**：HDFS 适合于大文件的存储，文档的大小应该是是 GB 到 TB 级别的。

**简单一致性模型**：HDFS 更适合于一次写入多次读取 (write-once-read-many) 的访问模型。支持将内容追加到文件末尾，但不支持数据的随机访问，不能从文件任意位置新增数据。

**跨平台移植性**：HDFS 具有良好的跨平台移植性，这使得其他大数据计算框架都将其作为数据持久化存储的首选方案。

# 5、Yarn相关面试题

## 1.Yarn任务提交流程?

当jobclient向YARN提交一个应用程序后，YARN将分两个阶段运行这个应用程序：一是启动ApplicationMaster;第二个阶段是由ApplicationMaster创建应用程序，为它申请资源，监控运行直到结束。具体步骤如下:

1. 用户向YARN提交一个应用程序，并指定ApplicationMaster程序、启动ApplicationMaster的命令、用户程序。
2. RM为这个应用程序分配第一个Container，并与之对应的NM通讯，要求它在这个Container中启动应用程序ApplicationMaster。
3. ApplicationMaster向RM注册，然后拆分为内部各个子任务，为各个内部任务申请资源，并监控这些任务的运行，直到结束。
4. AM采用轮询的方式向RM申请和领取资源。
5. RM为AM分配资源，以Container形式返回。
6. AM申请到资源后，便与之对应的NM通讯，要求NM启动任务。
7. NodeManager为任务设置好运行环境，将任务启动命令写到一个脚本中，并通过运行这个脚本启动任务。
8. 各个任务向AM汇报自己的状态和进度，以便当任务失败时可以重启任务。
9. 应用程序完成后，ApplicationMaster向ResourceManager注销并关闭自己。

## 2.Yarn资源调度的三种模型?

在Yarn中有三种调度器可以选择：FIFO Scheduler （先进先出），Capacity Scheduler（容量），Fair Scheduler（公平）。

Apache版本的hadoop默认使用的是Capacity Scheduler调度方式。CDH版本的默认使用的是Fair Scheduler调度方式

**FIFO 调度器：**支持单队列 、先进先出。生产环境不会

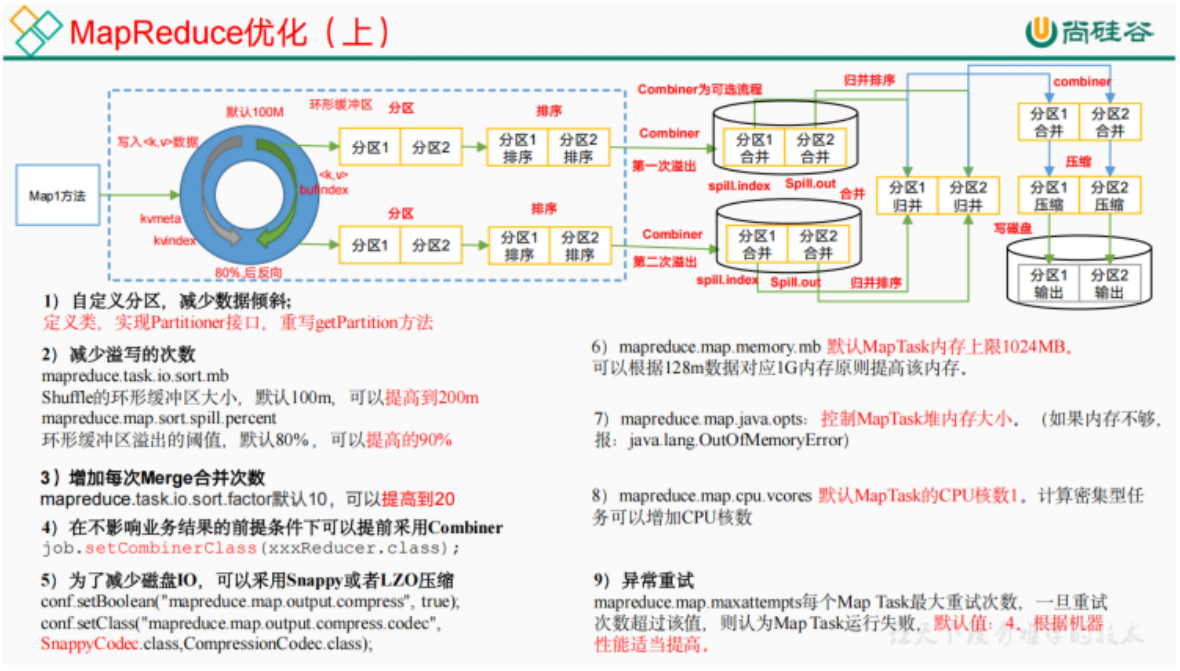
**容量调度器：**支持多队列。队列资源分配，优先选择资源占用率最低的队列分配资源；作业资源分配，按照作业的优先级和提交时间顺序分配资源；容器资源分配，本地原则（同一节点/同一机架/不同节点不同机架）。

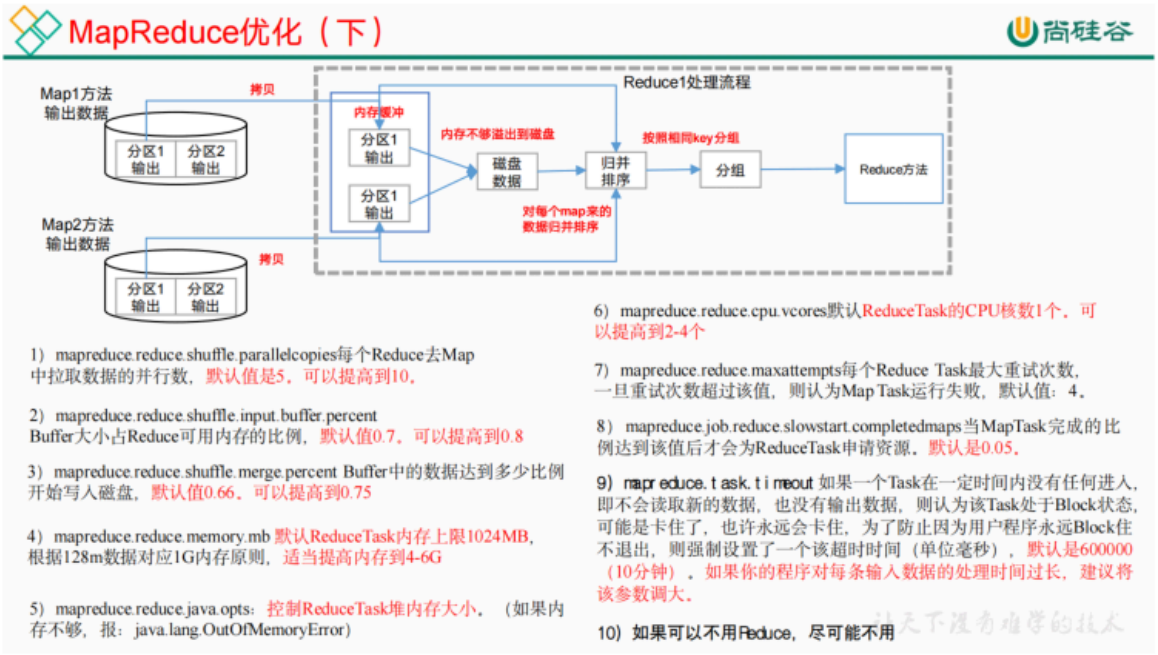
**公平调度器：**支持多队列，保证每个任务公平享有队列资源。资源不够时可以按照缺额分配。

> 大厂对并发度要求高，选择公平；小厂资源不充沛，选择容量

# 6、MapReduce相关面试题

## 1.MapReduce优化?






## 2.Map工作机制?

**简单概述：**

文件inputFile通过split被切割为多个split文件，通过Record按行读取内容给map（自己写的处理逻辑的方法），数据被map处理完之后交给OutputCollect收集器，对其结果key进行分区（默认使用的hashPartitioner），然后写入buffer，**每个map task 都有一个内存缓冲区**（环形缓冲区），存放着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溢写到磁盘，当整个map task 结束后再对磁盘中这个maptask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的拉取。

**详细概述：**

1. 读取数据组件 InputFormat (默认 TextInputFormat) 会通过 getSplits 方法对输入目录中的文件进行逻辑切片规划得到 block，有多少个 block就对应启动多少个 MapTask。
2. 将输入文件切分为 block 之后，由 RecordReader 对象 (默认是LineRecordReader) 进行读取，以 \n 作为分隔符, 读取一行数据, 返回 <key，value>， Key 表示每行首字符偏移值，Value 表示这一行文本内容。
3. 读取 block 返回 <key,value>, 进入用户自己继承的 Mapper 类中，执行用户重写的 map 函数，RecordReader 读取一行这里调用一次。
4. Mapper 逻辑结束之后，将 Mapper 的每条结果通过 context.write 进行collect数据收集。在 collect 中，会先对其进行分区处理，默认使用 HashPartitioner。
5. **接下来，会将数据写入内存，内存中这片区域叫做环形缓冲区(默认100M)，缓冲区的作用是 批量收集 Mapper 结果，减少磁盘 IO 的影响。我们的 Key/Value 对以及 Partition 的结果都会被写入缓冲区。当然，写入之前，Key 与 Value 值都会被序列化成字节数组。**
6. 当环形缓冲区的数据达到溢写比列(默认0.8)，也就是80M时，溢写线程启动，**需要对这 80MB 空间内的 Key 做排序 (Sort)。**排序是 MapReduce 模型默认的行为，这里的排序也是对序列化的字节做的排序。
7. 合并溢写文件，每次溢写会在磁盘上生成一个临时文件 (写之前判断是否有 Combiner)，如果 Mapper 的输出结果真的很大，有多次这样的溢写发生，磁盘上相应的就会有多个临时文件存在。当整个数据处理结束之后开始对磁盘中的临时文件进行 Merge 合并，因为最终的文件只有一个写入磁盘，并且为这个文件提供了一个索引文件，以记录每个reduce对应数据的偏移量。

## 3.Reduce工作机制?

**简单概述：**

Reduce 大致分为 copy、sort、reduce 三个阶段，重点在前两个阶段。

copy 阶段包含一个 eventFetcher 来获取已完成的 map 列表，由 Fetcher 线程去 copy 数据，在此过程中会启动两个 merge 线程，分别为 inMemoryMerger 和 onDiskMerger，分别将内存中的数据 merge 到磁盘和将磁盘中的数据进行 merge。待数据 copy 完成之后，copy 阶段就完成了。

开始进行 sort 阶段，sort 阶段主要是执行 finalMerge 操作，纯粹的 sort 阶段，完成之后就是 reduce 阶段，调用用户定义的 reduce 函数进行处理。

**详细描述：**

1. **Copy阶段**：简单地拉取数据。Reduce进程启动一些数据copy线程(Fetcher)，通过HTTP方式请求maptask获取属于自己的文件（map task 的分区会标识每个map task属于哪个reduce task ，默认reduce task的标识从0开始）。
2. **Merge阶段**：在远程拷贝数据的同时，ReduceTask启动了两个后台线程对内存和磁盘上的文件进行合并，以防止内存使用过多或磁盘上文件过多。

   merge有三种形式：内存到内存；内存到磁盘；磁盘到磁盘。默认情况下第一种形式不启用。当内存中的数据量到达一定阈值，就直接启动内存到磁盘的merge。与map端类似，这也是溢写的过程，这个过程中如果你设置有Combiner，也是会启用的，然后在磁盘中生成了众多的溢写文件。内存到磁盘的merge方式一直在运行，直到没有map端的数据时才结束，然后启动第三种磁盘到磁盘的merge方式生成最终的文件。
3. **合并排序**：把分散的数据合并成一个大的数据后，还会再对合并后的数据排序。
4. **对排序后的键值对调用reduce方法**：键相等的键值对调用一次reduce方法，每次调用会产生零个或者多个键值对，最后把这些输出的键值对写入到HDFS文件中。

## 4.Shuffle阶段?

shuffle阶段分为四个步骤：依次为：分区，排序，规约，分组，其中前三个步骤在map阶段完成，最后一个步骤在reduce阶段完成。

shuffle 是 Mapreduce 的核心，它分布在 Mapreduce 的 map 阶段和 reduce 阶段。一般把从 Map 产生输出开始到 Reduce 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阀值的时候，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. **MapTask阶段的Merge**：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阀值的时候，就会将数据写到磁盘之上。
5. **ReduceTask阶段的Merge**：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：`mapreduce.task.io.sort.mb` 默认100M

## 5.Shuffle压缩机制?

在shuffle阶段，可以看到数据通过大量的拷贝，从map阶段输出的数据，都要通过网络拷贝，发送到reduce阶段，这一过程中，涉及到大量的网络IO，如果数据能够进行压缩，那么数据的发送量就会少得多。

hadoop当中支持的压缩算法：
gzip、bzip2、LZO、LZ4、**Snappy**，这几种压缩算法综合压缩和解压缩的速率，谷歌的Snappy是最优的，一般都选择Snappy压缩。谷歌出品，必属精品。

## 6.MapTask并行度决定机制?

> 1G 的数据，启动 8 个 MapTask，可以提高集群的并发处理能力。那么 1K 的数据，也启动 8 个 MapTask，会提高集群性能吗？MapTask 并行任务是否越多越好呢？哪些因素影响了 MapTask 并行度

**数据块：**Block 是 HDFS 物理上把数据分成一块一块。数据块是 HDFS 存储数据单位。

**数据切片：**数据切片只是在逻辑上对输入进行分片，并不会在磁盘上将其切分成片进行存储。数据切片是 MapReduce 程序计算输入数据的单位，一个切片会对应启动一个 MapTask。

- **核心机制**：
  MapTask 的并行度首先由 **InputSplit 的数量**决定，每个 InputSplit 会生成一个独立的 MapTask。
  - 分片规则：
    - 文本文件：默认按 `dfs.blocksize`（如128MB）物理切分，但可通过 `mapreduce.input.fileinputformat.split.minsize/maxsize` 调整逻辑分片大小。
    - 非文本格式：由对应的 `InputFormat` 实现类控制（如 `DBInputFormat` 按SQL查询结果分片）。
  - **示例**：
    一个300MB的文件（块大小128MB）默认生成3个分片（128+128+44MB），即启动3个MapTask。
- **特殊场景**：
  - **小文件问题**：大量小文件会导致分片过多（每个文件至少1个分片），需通过 `CombineFileInputFormat` 合并。
  - **压缩文件**：不可切分的压缩格式（如GZIP）会强制整个文件作为1个分片，降低并行度。

## 6.MapTask都有哪些切片机制?

| 切片机制 | 使用场景 | 特点 |
| --- | --- | --- |
| 默认切片（TextInputFormat） | 普通大文件 | 按 block 大小切分 |
| CombineFileInputFormat | 小文件多 | 合并多个小文件 |
| NLineInputFormat | 行数均匀处理 | 每 N 行一片 |
| KeyValueTextInputFormat | 键值对文件 | 一行一对，分隔符控制 |

| 切片机制 | 使用场景 | 特点 |
| --- | --- | --- |
| SequenceFileInputFormat | 二进制 sequence 文件 | 按 block 分割 |
| 自定义 InputFormat | 特殊格式输入 | 完全自定义逻辑 |

# 二、HBase

## 1、HBase基础

Hbase 的表具有以下特点：

- 容量大：一个表可以有数十亿行，上百万列；
- 面向列：数据是按照列存储，每一列都单独存放，数据即索引，在查询时可以只访问指定列的数据，有效地降低了系统的 I/O 负担；
- 稀疏性：空 (null) 列并不占用存储空间，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，按照时间戳排序，新的数据在最上面；
- 存储类型：所有数据的底层存储格式都是字节数组 (byte[])。
- 不支持复杂的事务，只支持行级事务，即单行数据的读写都是原子性的；

Hadoop 可以通过 HDFS 来存储结构化、半结构甚至非结构化的数据，它是传统数据库的补充，是海量数据存储的最佳方法，它针对大文件的存储，批量访问和流式访问都做了优化，同时也通过多副本解决了容灾问题。但是 Hadoop 的缺陷在于它只能执行批处理，并且只能以顺序方式访问数据，这意味着即使是最简单的工作，也必须搜索整个数据集，无法实现对数据的随机访问。实现数据的随机访问是传统的关系型数据库所擅长的，但它们却不能用于海量数据的存储。在这种情况下，必须有一种新的方案来解决海量数据存储和随机访问的问题，HBase 就是其中之一 (HBase，Cassandra，couchDB，Dynamo 和MongoDB 都能存储海量数据并支持随机访问)。

### 1.HBase Table

Hbase是一个面向列族的数据库管理系统，表 schema 仅定义列族，表具有多个列族，每个列族可以包含任意数量的列，列由多个单元格（cell ）组成，单元格可以存储多个版本的数据，多个版本数据以时间戳进行区分。

### 2.Phoenix

Phoenix 是 HBase 的开源 SQL 中间层，它允许你使用标准 JDBC 的方式来操作 HBase 上的数据。在 Phoenix 之前，如果你要访问 HBase，只能调用它的 Java API，但相比于使用一行 SQL 就能实现数据查询，HBase 的 API 还是过于复杂。 Phoenix 的理念是 we put sql SQL back in NOSQL ，即你可以使用标准的 SQL 就能完成对 HBase 上数据的操作。同时这也意味着你可以通过集成 Spring Data JPA 或 Mybatis 等常用的持久层框架来操作HBase。

其次 Phoenix 的性能表现也非常优异， Phoenix 查询引擎会将 SQL 查询转换为一个或多个 HBase Scan，通过并行执行来生成标准的 JDBC 结果集。它通过直接使用 HBase API 以及协处理器和自定义过滤器，可以为小型数据查询提供毫秒级的性能，为千万行数据的查询提供秒级的性能。同时 Phoenix 还拥有二级索引等 HBase 不具备的特性，因为以上的优点，所以 Phoenix 成为了 HBase 最优秀的 SQL 中间层。

## 3.基本概念

- **Row Key（行键）**：是用来检索记录的主键。想要访问 HBase Table 中的数据，只有以下三种方式：

  1、通过指定的 Row Key 进行访问；
  2、通过 Row Key 的 range 进行访问，即访问指定范围内的行；
  3、进行全表扫描。

  > Row Key 可以是任意字符串，存储时数据按照 Row Key 的字典序进行排序。这里需要注意以下两点：
  >
  > 因为字典序对 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。如果你使用整型的字符串作为行键，那么为了保持整型的自然序，行键必须用 0 作左填充。
  >
  > 行的一次读写操作时原子性的 (不论一次读写多少列)。

- **Column Family（列族）**：

  HBase 表中的每个列，都归属于某个列族。列族是表的 Schema 的一部分，所以列族需要在创建表时进行定义。列族的所有列都以列族名作为前缀，例如 courses:history ， courses:math 都属于 courses 这个列族。

- **Column Qualifier（列限定符）**：

  列限定符，你可以理解为是具体的列名，例如 courses:history ， courses:math 都属于 courses这个列族，它们的列限定符分别是 history 和 math 。需要注意的是列限定符不是表 Schema 的一部分，你可以在插入数据的过程中动态创建列。

- **Column（列）**：

  HBase 中的列由列族和列限定符组成，它们由 : (冒号) 进行分隔，即一个完整的列名应该表述为 列族名 ： 列限定符 。

- **Cell（行）**：

  Cell 是行，列族和列限定符的组合，并包含值和时间戳。你可以等价理解为关系型数据库中由指定行和指定列确定的一个单元格，但不同的是 HBase 中的一个单元格是由多个版本的数据组成的，每个版本的数据用时间戳进行区分

- **Timestamp（时间戳）**：

  HBase 中通过 row key 和 column 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，也可以由客户显式指定。每个 Cell 中，不同版本的数据按照时间戳倒序排列，即最新的数据排在最前面。

## 4.存储结构

- **Regions**：

  HBase Table 中的所有行按照 Row Key 的字典序排列。HBase Tables 通过行键的范围 (row key range) 被水平切分成多个 Region , 一个 Region 包含了在 start key 和 end key 之间的所有行。

  > 每个表一开始只有一个 Region ， 随着数据不断增加， Region 会不断增大，当增大到一个阀值的时候， Region 就会等分为两个新的 Region 。当 Table 中的行不断增多，就会有越来越多的 Region 。

Region 是 HBase 中**分布式存储和负载均衡的最小单元。**这意味着不同的 Region 可以分布在不同的 Region　Server 上。但一个 Region 是不会拆分到多个 Server 上的。

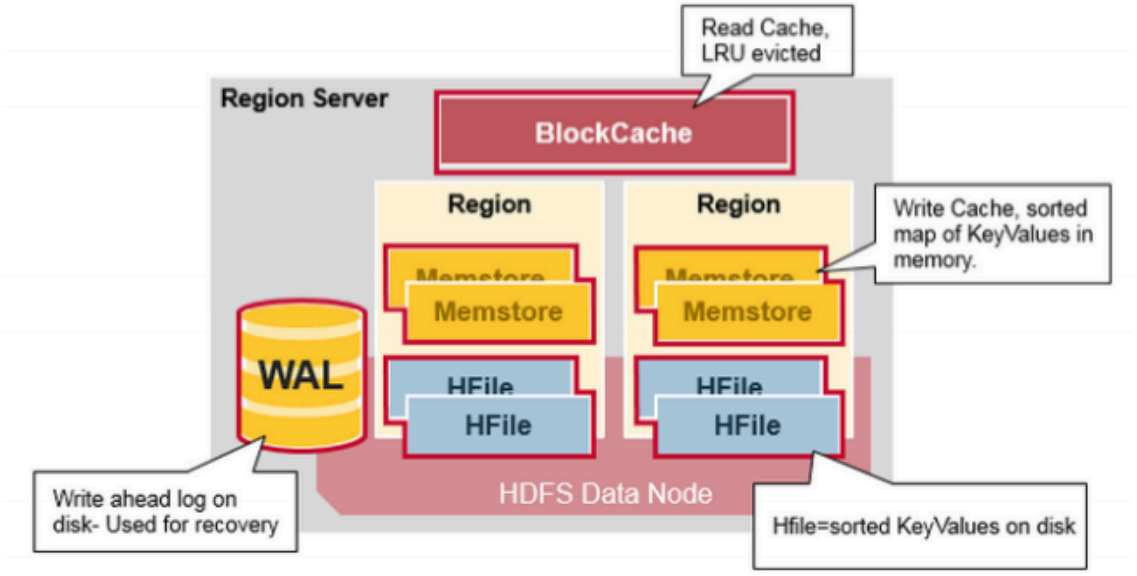
- **Region Server**：Region Server 运行在 HDFS 的 DataNode 上。它具有以下组件：

  1、**WAL(Write Ahead Log，预写日志)**：用于存储尚未进持久化存储的数据记录，以便在发生故障时进行恢复。

  2、**BlockCache**：读缓存。它将频繁读取的数据存储在内存中，如果存储不足，它将按照 最近最少使用原则 清除多余的数据。

  3、**MemStore**：写缓存。它存储尚未写入磁盘的新数据，并会在数据写入磁盘之前对其进行排序。每个 Region 上的每个列族都有一个 MemStore。

  4、**HFile** ： 将行数据按照 Key\Values 的形式存储在文件系统上。




> Region Server 存取一个子表时，会创建一个 Region 对象，然后对表的每个列族创建一个 Store 实例，每个 Store 会有 0 个或多个 StoreFile 与之对应，每个 StoreFile 则对应一个 HFile ， HFile 就是实际存储在 HDFS 上的文件。

## 5.系统架构

```
+-------------------+       +-------------------+
|   HBase Master    | <---->|     ZooKeeper     |
| - Region 分配     |       | - Master 选举     |
| - 元数据管理      |       | - RegionServer 注册|
| - 负载均衡        |       | - Meta 表位置管理  |
+--------^----------+       +---------^---------+
         |                            |
         v                            |
+-------------------+                 |
| RegionServer 1    |-----------------+
| - 多个 Region     |
| - WAL / MemStore  |
| - BlockCache      |
+-------------------+
| RegionServer 2    |
| ...               |
+-------------------+
```

HBase 系统遵循 Master/Salve 架构，由三种不同类型的组件组成：

**Zookeeper**

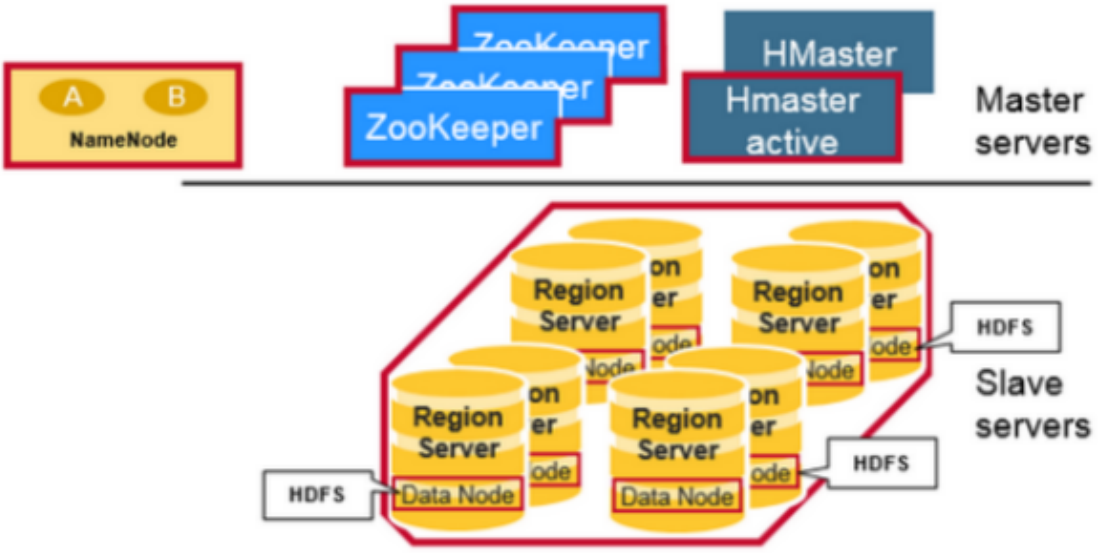
1. 保证任何时候，集群中只有一个 Master；
2. 存贮所有 Region 的寻址入口；
3. 实时监控 Region Server 的状态，将 Region Server 的上线和下线信息实时通知给 Master；
4. 存储 HBase 的 Schema，包括有哪些 Table，每个 Table 有哪些 Column Family 等信息。

**Master**

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. GFS 上的垃圾文件回收；
5. 处理 Schema 的更新请求。

**Region Server**

1. Region Server 负责维护 Master 分配给它的 Region ，并处理发送到 Region 上的 IO 请求；
2. Region Server 负责切分在运行过程中变得过大的 Region。




HBase 使用 ZooKeeper 作为分布式协调服务来维护集群中的服务器状态。 Zookeeper 负责维护可用服务列表，并提供服务故障通知等服务：

- 每个 Region Server 都会在 ZooKeeper 上创建一个临时节点，Master 通过 Zookeeper 的Watcher 机制对节点进行监控，从而可以发现新加入的 Region Server 或故障退出的 Region Server；
- 所有 Masters 会竞争性地在 Zookeeper 上创建同一个临时节点，由于 Zookeeper 只能有一个同名节点，所以必然只有一个 Master 能够创建成功，此时该 Master 就是主 Master，主 Master 会定期向 Zookeeper 发送心跳。备用 Masters 则通过 Watcher 机制对主 HMaster 所在节点进行监听；
- 如果主 Master 未能定时发送心跳，则其持有的 Zookeeper 会话会过期，相应的临时节点也会被删除，这会触发定义在该节点上的 Watcher 事件，使得备用的 Master Servers 得到通知。所有备用的 Master Servers 在接到通知后，会再次去竞争性地创建临时节点，完成主 Master 的选举。

## 6.集群环境

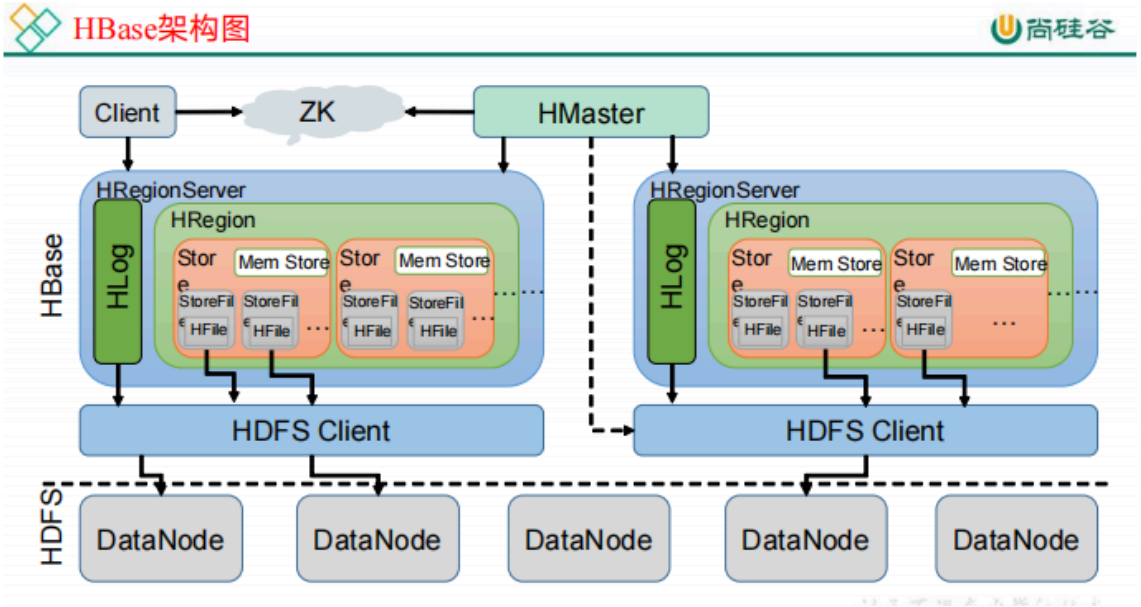
# 2、HBase常用命令

```
#1、基本命令-------------------------------------------------------------------------
--------
#打开HBase Shell
hbase shell
#查看服务器状态
status
#查看版本信息
version

#2、关于表的操作---------------------------------------------------------------------
---------
#查看所有表
list
#创建表
# 创建一张名为Student的表,包含基本信息（baseInfo）、学校信息（schoolInfo）两个列族
create 'Student','baseInfo','schoolInfo'
#查看表的基本信息
describe 'Student'
# 禁用表
disable 'Student'
# 检查表是否被禁用
is_disabled 'Student'
# 启用表
enable 'Student'
# 检查表是否被启用
is_enabled 'Student'
#检查表是否存在
exists 'Student'
#删除表
# 删除表前需要先禁用表
disable 'Student'
# 删除表
drop 'Student

#3、增删改---------------------------------------------------------------------------
---------
#添加列族
alter 'Student', 'teacherInfo'
#删除列族
alter 'Student', {NAME => 'teacherInfo', METHOD => 'delete'}
#更改列族存储版本的限制（默认情况下，列族只存储一个版本的数据，如果需要存储多个版本的数据，则需要修改列族的属性。修改后可通过 desc 命令查看。）
alter 'Student',{NAME=>'baseInfo',VERSIONS=>3}
#插入数据
put 'Student', 'rowkey1','baseInfo:name','tom'
put 'Student', 'rowkey1','baseInfo:birthday','1990-01-09'
put 'Student', 'rowkey1','baseInfo:age','29'
put 'Student', 'rowkey1','schoolInfo:name','Havard'
put 'Student', 'rowkey1','schoolInfo:localtion','Boston
put 'Student', 'rowkey2','baseInfo:name','jack'
put 'Student', 'rowkey2','baseInfo:birthday','1998-08-22'
```

```
put 'Student', 'rowkey2','baseInfo:age','21'
put 'Student', 'rowkey2','schoolInfo:name','yale'
put 'Student', 'rowkey2','schoolInfo:localtion','New Haven'
#获取指定行、指定行中的列族，列的信息
# 获取指定行中所有列的数据信息
get 'Student','rowkey3'
# 获取指定行中指定列族下所有列的数据信息
get 'Student','rowkey3','baseInfo'
# 获取指定行中指定列的数据信息
get 'Student','rowkey3','baseInfo:name'
#删除指定行、指定行中的列
# 删除指定行
delete 'Student','rowkey3'
# 删除指定行中指定列的数据
delete 'Student','rowkey3','baseInfo:name'

#4、查询------------------------------------------------------------------------
--------
（hbase 中访问数据有两种基本的方式：按指定 rowkey 获取数据：get 方法；按指定条件获取数据：
scan 方法）
#Get查询
# 获取指定行中所有列的数据信息
get 'Student','rowkey3'
# 获取指定行中指定列族下所有列的数据信息
get 'Student','rowkey3','baseInfo'
# 获取指定行中指定列的数据信息
get 'Student','rowkey
#查询整表的数据
scan 'Student'
#查询指定列簇的数据
scan 'Student', {COLUMN=>'baseInfo'}
#条件scan查询
# 查询指定列的数据
scan 'Student', {COLUMNS=> 'baseInfo:birthday'}
#条件查询
scan 'Student', FILTER=>"ValueFilter(=,'binary:24')"
```

# 3、HBase相关面试题

## 1.HBase的存储结构?




**架构角色：**

**1）Master**

实现类为 HMaster，负责监控集群中所有的 RegionServer 实例。主要作用如下：

（1）管理元数据表格 hbase:meta，接收用户对表格创建修改删除的命令并执行

（2）监控 region 是否需要进行负载均衡，故障转移和 region 的拆分。

通过启动多个后台线程监控实现上述功能：

①LoadBalancer 负载均衡器：周期性监控 region 分布在 regionServer 上面是否均衡，由参数 hbase.balancer.period 控制周期时间，默认 5 分钟。

②CatalogJanitor 元数据管理器：定期检查和清理 HBase：meta 中的数据。meta 表内容在进阶中介绍。

③MasterProcWAL Master 预写日志处理器：把 Master 需要执行的任务记录到预写日志 WAL 中，如果 Master 宕机，让 backupMaster读取日志继续干。

**2）Region Server**

Region Server 实现类为 HRegionServer，主要作用如下:

（1）负责数据 cell 的处理，例如写入数据 put，查询数据 get 等

（2）拆分合并 Region 的实际执行者，有 Master 监控，有 regionServer 执行。

**3）Zookeeper**

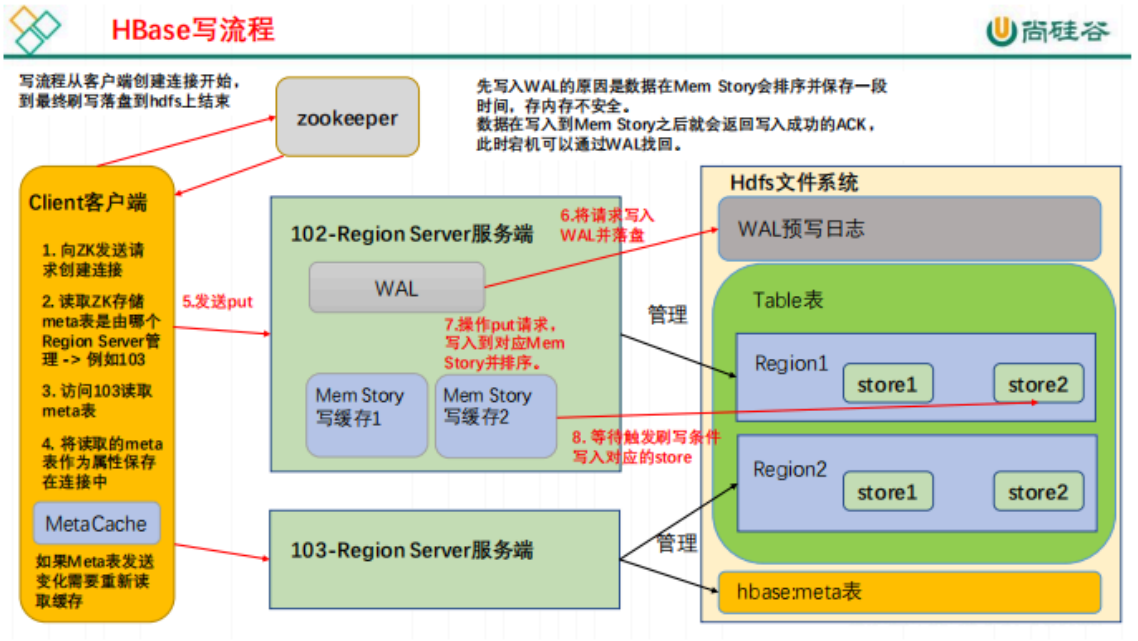
HBase 通过 Zookeeper 来做 Master 的高可用、记录 RegionServer 的部署信息、并且存储有 meta 表的位置信息。

HBase 对于数据的读写操作时直接访问 Zookeeper 的，在 2.3 版本推出 Master Registry模式，客户端可以直接访问 Master。使用此功能，会加大对 Master 的压力，减轻对 Zookeeper的压力。

**4）HDFS**

HDFS 为 HBase 提供最终的底层数据存储服务，同时为 HBase 提供高容错的支持。

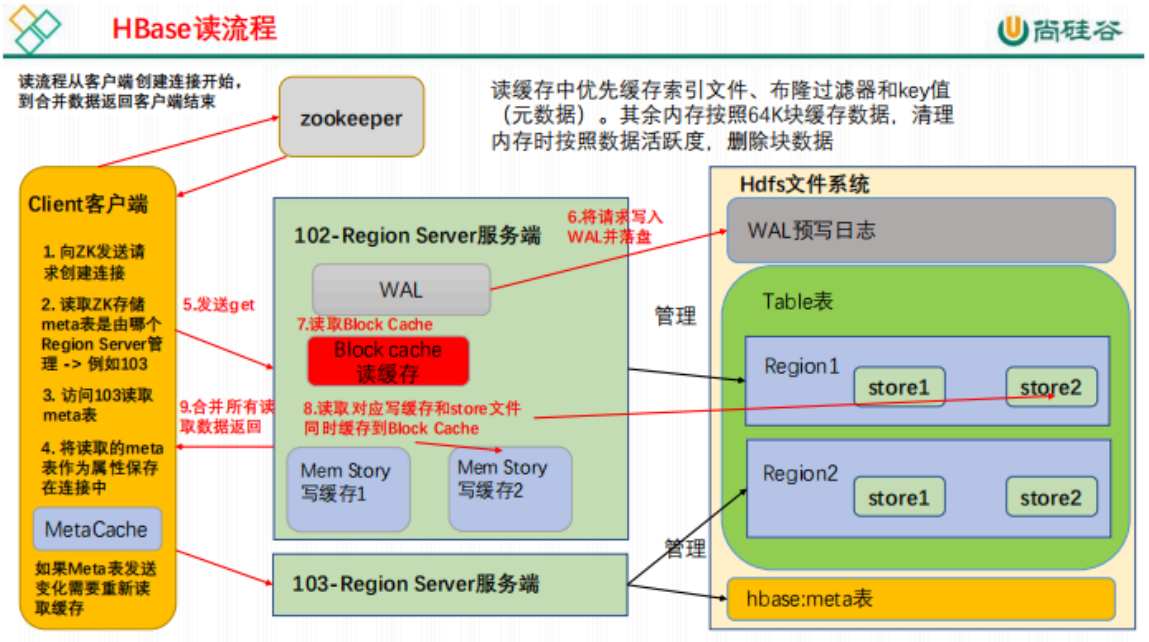
## 2.HBase的写流程?




写流程顺序正如 API 编写顺序，首先创建 HBase 的重量级连接

（1）读取本地缓存中的 Meta 表信息；（第一次启动客户端为空）
（2）向 ZK 发起读取 Meta 表所在位置的请求；
（3）ZK 正常返回 Meta 表所在位置；
（4）向 Meta 表所在位置的 RegionServer 发起请求读取 Meta 表信息；
（5）读取到 Meta 表信息并将其缓存在本地；
（6）向待写入表发起写数据请求；
（7）先写 WAL预写日志，再写 MemStore写缓存（排好序后再刷盘）,并向客户端返回写入数据成功

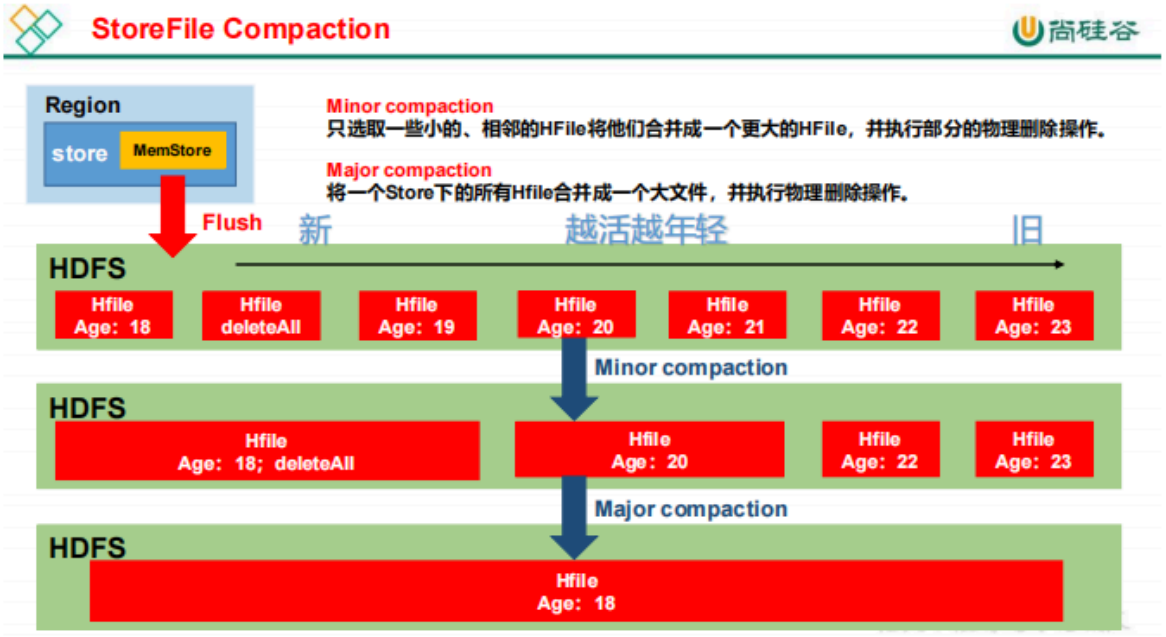
## 3.HBase的读流程




（1）读取本地缓存中的 Meta 表信息；（第一次启动客户端为空）
（2）向 ZK 发起读取 Meta 表所在位置的请求；
（3）ZK 正常返回 Meta 表所在位置；
（4）向 Meta 表所在位置的 RegionServer 发起请求读取 Meta 表信息；

（5）读取到 Meta 表信息并将其缓存在本地；
（6）MemStore、StoreFile、BlockCache
同时构建 MemStore 与 StoreFile 的扫描器，
MemStore：正常读
StoreFile：
根据索引确定待读取文件；
再根据 BlockCache 确定读取文件；
（7）合并多个位置读取到的数据,给用户返回最大版本的数据，如果最大版本数据为删除标记，则不给不返回任何数据

## 4.HBase的合并?




由于 memstore 每次刷写都会生成一个新的 HFile，文件过多读取不方便，所以会进行文件的合并，清理掉过期和删除的数据，会进行 StoreFile Compaction。

Compaction 分为两种，分别是 Minor Compaction 和 Major Compaction。

Minor Compaction会将临近的若干个较小的 HFile 合并成一个较大的 HFile，并清理掉部分过期和删除的数据，有系统使用一组参数自动控制；

Major Compaction 会将一个 Store 下的所有的 HFile 合并成一个大 HFile，并且**会**清理掉所有过期和删除的数据，由参数 hbase.hregion.majorcompaction控制，默认 7 天。

## 5.HBase的rowkey设计原则

HBase中，表会被划分为1...n个Region，被托管在RegionServer中。Region二个重要的属性：StartKey与EndKey表示这个Region维护的rowKey范围，当我们要读/写数据时，如果rowKey落在某个start-end key范围内，那么就会定位到目标region并且读/写到相关的数据。

**1、rowkey长度原则**

Rowkey是一个二进制码流，Rowkey的长度被很多开发者建议说设计在10~100个字节，不过建议是越短越好，不要超过16个字节。

原因如下：

1）数据的持久化文件HFile中是按照Key-Value 存储的，如果Rowkey过长比如100个字节，1000万列数据光Rowkey就要占用100*1000 万=10亿个字节，将近1G数据，这会极大影响 HFile的存储效率；

2）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会降低，系统将无法缓存更多的数据，这会降低检索效率。因此Rowkey的字节长度越短越好；

3）目前操作系统是都是64位系统，内存8字节对齐。控制在16个字节，8字节的整数倍利用操作系统的最佳特性。

**2、rowkey散列原则**

如果rowkey是按时间戳的方式递增，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，由程序循环生成，低位放时间字段，将会提高数据均衡分布在每个Regionserver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将产生所有新数据都在一个RegionServer上堆积的热点现象，这样在做数据检索的时候负载将会集中在个别 RegionServer，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。

# 6.HBase的rowkey如何设计?

**期望：**通过对rowkey的设计，使用户数据能够分散到多个region中

1、预分区，创建表时指定分区
2、按照rowkey设计三原则合理设计
3、写入时反转id
4、增加随机数
5、哈希
6、时间戳反转
7、尽量减少行和列的大小

# 7.HBase高可用？容灾和备份?

## 1）高可用

HBase 高可用的核心思想：

- **Master 高可用**（控制面）： 多Master部署，通过zk选举Active Master
- **RegionServer 高可用**（数据面）： 多RegionServer部署，依赖HDFS多副本保障数据不丢失
- **ZooKeeper 高可用**（协调面）： 部署奇数个zk节点
- **HDFS 高可用**（存储面，NameNode HA）： NameNode HA，DataNode多副本

## 2）容灾和备份

Hbase 常用的三种简单的容灾备份方案，即**CopyTable**、**Export/Import**（数据导出到HDFS）、**Snapshot**。

# 8.HBase优化?

1、rowkey设计优化
2、参数优化
    1）Zookeeper会话超时时间
    2）设置RPC监听数量
    3）手动控制Major Compaction
    4）优化HStore文件大小
    5）优化HBase客户端缓存

6）指定scan.next扫描HBase所获取的行数
7）适当调整BlockCache占用RegionServer堆内存的比例
8）适当调整MemStore占用RegionServer堆内存的比例

## 9.HBase使用经验法则

官方给出了权威的使用法则：

（1）Region 大小控制 10-50G
（2）cell 大小不超过 10M（性能对应小于 100K 的值有优化），如果使用 mob（Medium sized Objects 一种特殊用法）则不超过 50M。
（3）1 张表有 1 到 3 个列族，不要设计太多。最好就 1 个，如果使用多个尽量保证不会同时读取多个列族。
（4）1 到 2 个列族的表格，设计 50-100 个 Region。
（5）列族名称要尽量短，不要去模仿 RDBMS（关系型数据库）具有准确的名称和描述。
（6）如果 RowKey 设计时间在最前面，会导致有大量的旧数据存储在不活跃的 Region中，使用的时候，仅仅会操作少数的活动 Region，此时建议增加更多的 Region 个数。
（7）如果只有一个列族用于写入数据，分配内存资源的时候可以做出调整，即写缓存不会占用太多的内存。

# 三、Spark

Spark是一种基于内存的快速、通用、可扩展的大数据分析计算引擎，基于Scala语言进行开发，提供了Scala、Java、Python语言的API，依赖安装的Scala环境，Spark适合离线批处理。

# 1、Spark Core

## 1.Spark特点

1、使用先进的 DAG 调度程序，查询优化器和物理执行引擎，可以通过内存来高效处理数据流，以实现性能上的保证；
2、多语言支持，目前支持的有 Java，Scala，Python 和 R；
3、提供了 80 多个高级 API，可以轻松地构建应用程序；
4、支持批处理，流处理和复杂的业务分析；
5、丰富的类库支持：包括 SQL，MLlib，GraphX 和 Spark Streaming 等库，并且可以将它们无缝地进行组合； 6、丰富的部署模式：支持本地模式和自带的集群模式，也支持在 Hadoop，Mesos，Kubernetes 上运行；
7、多数据源支持：支持访问 HDFS，Alluxio，Cassandra，HBase，Hive 以及数百个其他数据源中的数据。

**1）快：**与Hadoop的MapReduce相比，Spark基于内存的运算要快100倍以上，基于硬盘的运算也要快10倍以上。Spark实现了高效的DAG执行引擎，可以通过基于内存来高效处理数据流。计算的中间结果是存在于内存中的。

**2）易用：**Spark支持Java、Python和Scala的API，还支持超过80种高级算法，使用户可以快速构建不同的应用。而且Spark支持交互式的Python和Scala的Shell，可以非常方便地在这些Shell中使用Spark集群来验证解决问题的方法。

**3）通用：**Spark提供了统一的解决方案。Spark可以用于，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）。这些不同类型的处理都可以在同一个应用中无缝使用。减少了开发和维护的人力成本和部署平台的物力成本。

**4）兼容性：**Spark可以非常方便地与其他的开源产品进行融合。比如，Spark可以使用Hadoop的YARN和Apache Mesos作为它的资源管理和调度器，并且可以处理所有Hadoop支持的数据，包括HDFS、HBase等。这对于已经部署Hadoop集群的用户特别重要，因为不需要做任何数据迁移就可以使用Spark的强大处理能力。

## 2.Spark运行模式

部署Spark集群大体上分为两种模式：**单机模式与集群模式**

（1）**Local模式**：在本地部署单个Spark服务。它采用单节点多线程方式运行，不用部署，开箱即用，适合日常

测试开发

（2）**Standalone模式**：Spark自带的任务调度模式。（国内不常用）

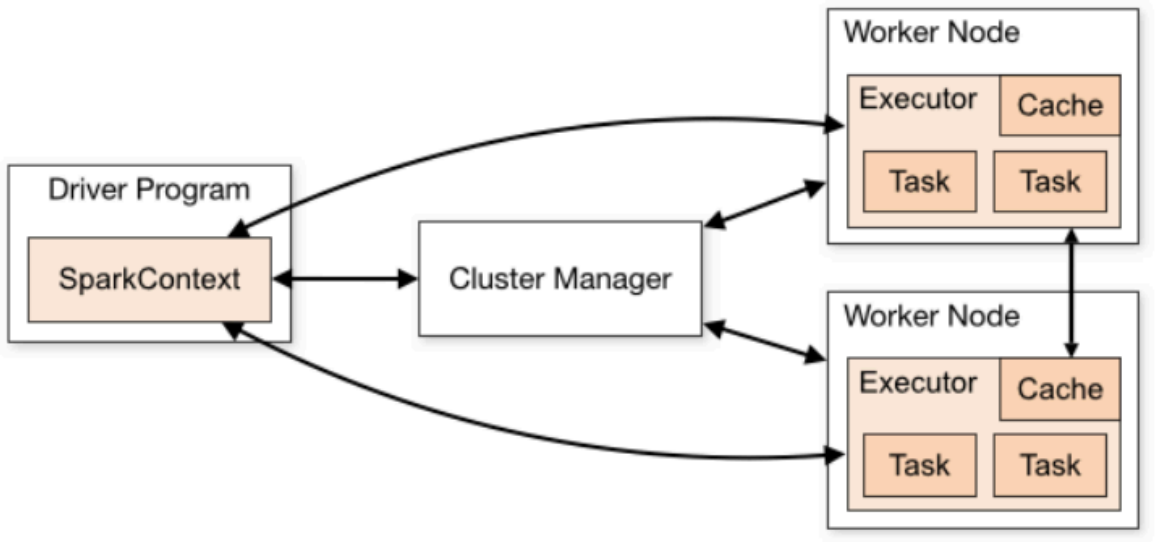
（3）**YARN模式**：Spark使用Hadoop的YARN组件进行资源与任务调度。（国内最常用）

（4）**Mesos模式**：Spark使用Mesos平台进行资源与任务的调度。（国内很少用）

> Spark查看当前Spark-shell运行任务情况端口号：4040
> Spark历史服务器端口号：18080

## 3.集群架构

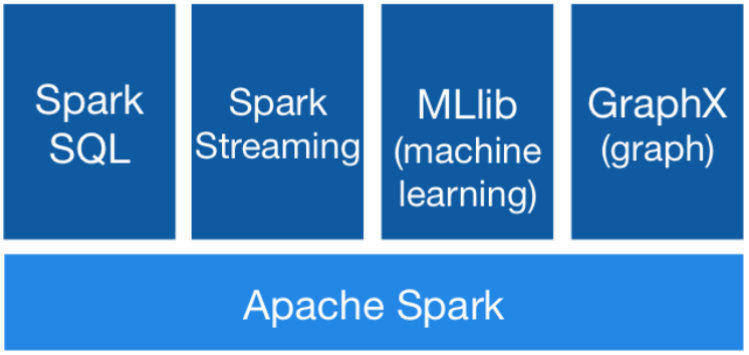
| Term | Meaning |
| --- | --- |
| Application | Spark 应用程序，由集群上的一个 Driver 节点和多个 Executor 节点组成。 |
| Driver program | 主应用程序，该进程运行应用的 main() 方法并且创建 SparkContext |
| Cluster manager | 集群资源管理器（例如，Standlone Manager，Mesos，YARN） |
| Worker node | 执行计算任务的工作节点 |
| Executor | 位于工作节点上的应用进程，负责执行计算任务并且将输出数据保存到内存或磁盘中 |
| Task | 被发送到 Executor 中的工作单元 |




**执行过程**:

1. 用户程序创建 SparkContext 后，它会连接到集群资源管理器，集群资源管理器会为用户程序分配计算资源，并启动 Executor；
2. Driver 将计算程序划分为不同的执行阶段和多个 Task，之后将 Task 发送给 Executor；
3. Executor 负责执行 Task，并将执行状态汇报给 Driver，同时也会将当前节点资源的使用情况汇报给集群资源管理器。

## 4.核心组件




- **Spark SQL**

  Spark SQL 主要用于结构化数据的处理。其具有以下特点：

  1、能够将 SQL 查询与 Spark 程序无缝混合，允许您使用 SQL 或 DataFrame API 对结构化数据进行查询；
  2、支持多种数据源，包括 Hive，Avro，Parquet，ORC，JSON 和 JDBC；
  3、支持 HiveQL 语法以及用户自定义函数 (UDF)，允许你访问现有的 Hive 仓库；
  4、支持标准的 JDBC 和 ODBC 连接；
  5、支持优化器，列式存储和代码生成等特性，以提高查询效率。

- **Spark Streaming**

  Spark Streaming 主要用于快速构建可扩展，高吞吐量，高容错的流处理程序。支持从 HDFS，Flume，Kafka，Twitter 和 ZeroMQ 读取数据，并进行处理。


Spark Streaming 的本质是微批处理，它将数据流进行极小粒度的拆分，拆分为多个批处理，从而达到接近于流处理的效果。

- **MLib**

  MLlib 是 Spark 的机器学习库。其设计目标是使得机器学习变得简单且可扩展。它提供了以下工具：

  **常见的机器学习算法**：如分类，回归，聚类和协同过滤；

  **特征化**：特征提取，转换，降维和选择；

  **管道**：用于构建，评估和调整 ML 管道的工具；

  **持久性**：保存和加载算法，模型，管道数据；

  **实用工具**：线性代数，统计，数据处理等。

- **Graphx**

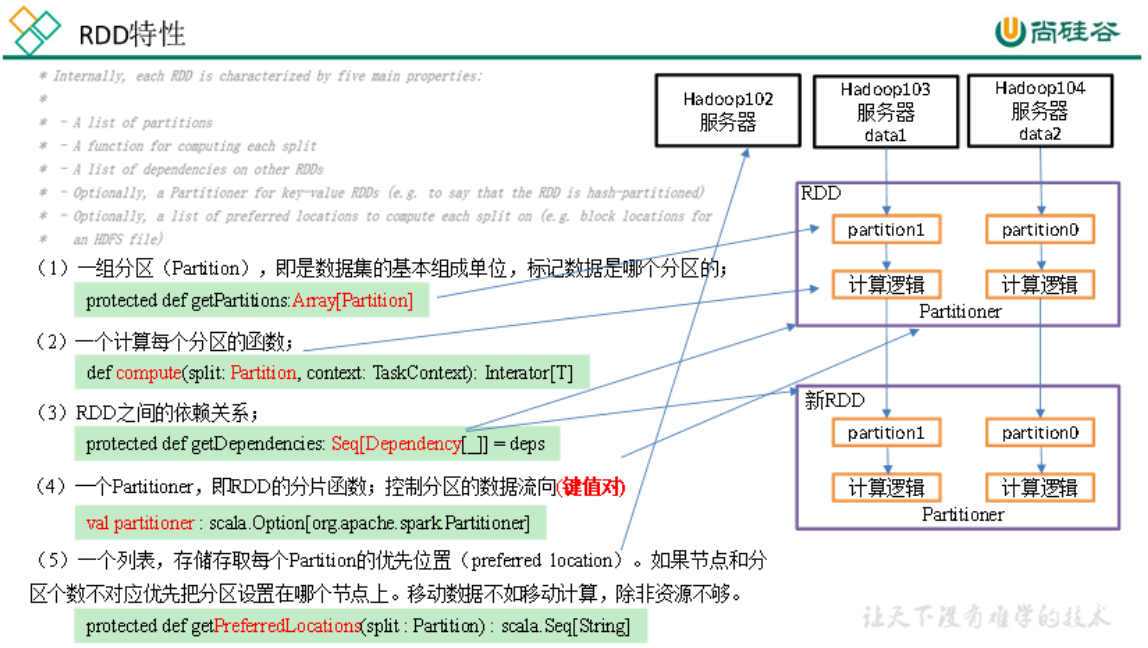
GraphX 是 Spark 中用于图形计算和图形并行计算的新组件。在高层次上，GraphX 通过引入一个新的图形抽象来扩展 RDD(一种具有附加到每个顶点和边缘的属性的定向多重图形)。为了支持图计算，GraphX 提供了一组基本运算符（如： subgraph，joinVertices 和 aggregateMessages）以及优化后的 Pregel API。此外，GraphX 还包括越来越多的图形算法和构建器，以简化图形分析任务。

## 5.弹性数据集RDDs

RDD 全称为 Resilient Distributed Datasets（弹性分布式数据集），是 Spark 最基本的数据抽象，它是只读的、分区记录的集合，支持并行操作，可以由外部数据集或其他 RDD 转换而来，是 Spark 的核心数据结构，可以看作是：

- **分布式**：数据被切分成多个分区（Partition），分布在集群节点上并行处理
- **不可变**：RDD 一旦创建不可修改，只能通过转换（Transformation）生成新的 RDD
- **可容错**：通过记录生成 RDD 的操作链（Lineage）来容错，分区丢失时可重算

它具有以下特性：




1、一个 RDD 由一个或者多个分区（Partitions）组成。对于 RDD 来说，每个分区会被一个计算任务所处理，用户可以在创建 RDD 时指定其分区个数，如果没有指定，则默认采用程序所分配到的CPU 的核心数；

2、RDD 拥有一个用于计算分区的函数 compute；

3、RDD 会保存彼此间的依赖关系，RDD 的每次转换都会生成一个新的依赖关系，这种 RDD 之间的依赖关系就像流水线一样。在部分分区数据丢失后，可以通过这种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、Key-Value 型的 RDD 还拥有 Partitioner(分区器)，用于决定数据被存储在哪个分区中，目前Spark 中支持 HashPartitioner(按照哈希分区) 和 RangeParationer(按照范围进行分区)；

5、一个优先位置列表 (可选)，用于存储每个分区的优先位置 (prefered location)。对于一个 HDFS 文件来说，这个列表保存的就是每个分区所在的块的位置，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，会尽可能的将计算任务分配到其所要处理数据块的存储位置

> RDD[T]抽象类的部分相关代码如下：

```
// 由子类实现以计算给定分区
def compute(split: Partition, context: TaskContext): Iterator[T]
// 获取所有分区
protected def getPartitions: Array[Partition]
// 获取所有依赖关系
protected def getDependencies: Seq[Dependency[_]] = deps
// 获取优先位置列表
protected def getPreferredLocations(split: Partition): Seq[String] = Nil
// 分区器 由子类重写以指定它们的分区方式
@transient val partitioner: Option[Partitioner] = None
```

## 1）操作RDD

RDD 支持两种类型的操作：*transformations*（转换，从现有数据集创建新数据集）和 *actions*（在数据集上运行计算后将值返回到驱动程序）。RDD 中的所有转换操作都是惰性的，它们只是记住这些转换操作，但不会立即执行，只有遇到 *action* 操作后才会真正的进行计算，这类似于函数式编程中的惰性求值

```scala
val list = List(1, 2, 3)
// map 是一个 transformations 操作，而 foreach 是一个 actions 操作
sc.parallelize(list).map(_ * 10).foreach(println)
// 输出： 10 20 30
```

### 2）缓存RDD持久化

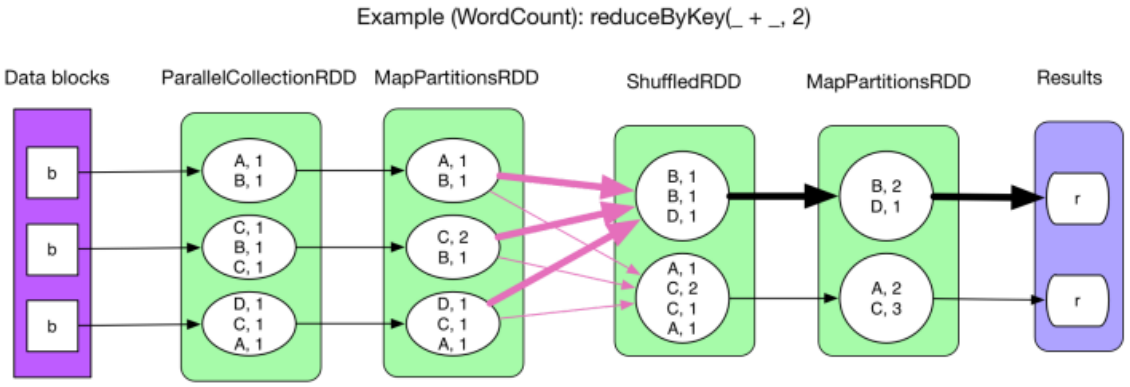
Spark 速度非常快的一个原因是 RDD 支持缓存。成功缓存后，如果之后的操作使用到了该数据集，则直接从缓存中获取。虽然缓存也有丢失的风险，但是由于 RDD 之间的依赖关系，如果某个分区的缓存数据丢失，只需要重新计算该分区即可。Spark 支持多种缓存级别：

| Storage Level | Meaning |
| --- | --- |
| MEMORY_ONLY | 默认的缓存级别，将 RDD 以反序列化的 Java 对象的形式存储在 JVM中。如果内存空间不够，则部分分区数据将不再缓存。 |
| MEMORY_AND_DISK | 将 RDD 以反序列化的 Java 对象的形式存储 JVM 中。如果内存空间不够，将未缓存的分区数据存储到磁盘，在需要使用这些分区时从磁盘读取。 |
| MEMORY_ONLY_SER | 将 RDD 以序列化的 Java 对象的形式进行存储（每个分区为一个 byte数组）。这种方式比反序列化对象节省存储空间，但在读取时会增加CPU 的计算负担。仅支持 Java 和 Scala 。 |
| MEMORY_AND_DISK_SER | 类似于 MEMORY_ONLY_SER ，但是溢出的分区数据会存储到磁盘，而不是在用到它们时重新计算。仅支持 Java 和 Scala。 |
| DISK_ONLY | 只在磁盘上缓存 RDD |
| MEMORY_ONLY_2 , MEMORY_AND_DISK_2 ,etc | 与上面的对应级别功能相同，但是会为每个分区在集群中的两个节点上建立副本。 |
| OFF_HEAP | 与 MEMORY_ONLY_SER 类似，但将数据存储在堆外内存中。这需要启 |

> 启动堆外内存需要配置两个参数：
>
> **spark.memory.offHeap.enabled** ： 是否开启堆外内存，默认值为 false，需要设置为true；
>
> **spark.memory.offHeap.size** : 堆外内存空间的大小，默认值为 0，需要设置为正值。

### 3）理解shuffle

在 Spark 中，一个任务对应一个分区，通常不会跨分区操作数据。但如果遇到 reduceByKey 等操作，Spark 必须从所有分区读取数据，并查找所有键的所有值，然后汇总在一起以计算每个键的最终结果，这称为 Shuffle 。




**Shuffle的影响：**

Shuffle 是一项昂贵的操作，因为它通常会跨节点操作数据，这会涉及磁盘 I/O，网络 I/O，和数据序列化。某些 Shuffle 操作还会消耗大量的堆内存，因为它们使用堆内存来临时存储需要网络传输的数据。Shuffle 还会在磁盘上生成大量中间文件，从 Spark 1.3 开始，这些文件将被保留，直到相应的 RDD 不再使用并进行垃圾回收，这样做是为了避免在计算时重复创建 Shuffle 文件。如果应用程序长期保留对这些 RDD 的引用，则垃圾回收可能在很长一段时间后才会发生，这意味着长时间运行的 Spark 作业可能会占用大量磁盘空间，通常可以使用 spark.local.dir 参数来指定这些临时文件的存储目录。

**导致Shuffle的操作：**

由于 Shuffle 操作对性能的影响比较大，所以需要特别注意使用，以下操作都会导致 Shuffle：

1、**涉及到重新分区操作**： 如 repartition 和 coalesce ；
2、**所有涉及到 ByKey 的操作**：如 groupByKey 和 reduceByKey ，但 countByKey 除外；
3、**联结操作**：如 cogroup 和 join 。

## 4）宽依赖和窄依赖

RDD 和它的父 RDD(s) 之间的依赖关系分为两种不同的类型：

- **窄依赖 (narrow dependency)**：父 RDDs 的一个分区最多被子 RDDs 一个分区所依赖；
- **宽依赖 (wide dependency)**：父 RDDs 的一个分区可以被子 RDDs 的多个子分区所依赖。

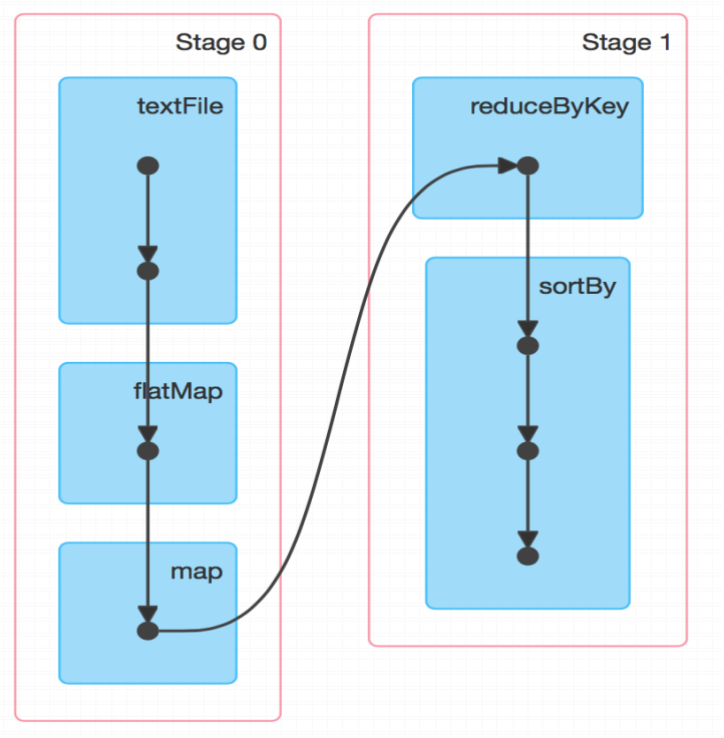
区分这两种依赖是非常有用的：

- 首先，窄依赖允许在一个集群节点上以流水线的方式（pipeline）对父分区数据进行计算，例如先执行 map 操作，然后执行 filter 操作。而宽依赖则需要计算好所有父分区的数据，然后再在节点之间进行 Shuffle，这与 MapReduce 类似。窄依赖能够更有效地进行数据恢复，因为只需重新对丢失分区的父分区进行计算，且不同节点之间
- 可以并行计算；而对于宽依赖而言，如果数据丢失，则需要对所有父分区数据进行计算并再次 Shuffle。

## 5）DAG的生成

RDD(s) 及其之间的依赖关系组成了 DAG(有向无环图)，DAG 定义了这些 RDD(s) 之间的 Lineage(血统) 关系，通过血统关系，如果一个 RDD 的部分或者全部计算结果丢失了，也可以重新进行计算。那么 Spark 是如何根据 DAG 来生成计算任务呢？主要是根据依赖关系的不同将 DAG 划分为不同的计算阶段 (Stage)：

- 对于窄依赖，由于分区的依赖关系是确定的，其转换操作可以在同一个线程执行，所以可以划分到同一个执行阶段；
- 对于宽依赖，由于 Shuffle 的存在，只能在父 RDD(s) 被 Shuffle 处理完成后，才能开始接下来的计算，因此遇到宽依赖就需要重新划分阶段。




## 6.Transformation和Action常用算子

### 1）spark 常用的 Transformation 算子如下表：

| Transformation 算子 | Meaning（含义） |
| --- | --- |
| **map**(*func*) | 对原 RDD 中每个元素运用 *func* 函数，并生成新的 RDD |
| **filter**(*func*) | 对原 RDD 中每个元素使用*func* 函数进行过滤，并生成新的 RDD |
| **flatMap**(*func*) | 与 map 类似，但是每一个输入的 item被映射成 0 个或多个输出的 items（*func* 返回类型需要为 Seq ）。 |
| **mapPartitions**(*func*) | 与 map 类似，但函数单独在 RDD 的每个分区上运行， *func*函数的类型为 Iterator => Iterator，其中 T 是RDD 的类型，即 RDD[T] |
| **mapPartitionsWithIndex**(*func*) | 与 mapPartitions 类似，但 *func* 类型为 (Int, Iterator) => Iterator，其中第一个参数为分区索引 |
| **sample**(*withReplacement*, *fraction*, *seed*) | 数据采样，有三个可选参数：设置是否放回（withReplacement）、采样的百分比（*fraction*）、随机数生成器的种子（seed）； |
| **union**(*otherDataset*) | 合并两个 RDD |
| **intersection**(*otherDataset*) | 求两个 RDD 的交集 |

| Transformation 算子 | Meaning（含义） |
| --- | --- |
| **distinct**([*numTasks*])) | 去重 |
| **groupByKey**([*numTasks*]) | 按照 key 值进行分区，即在一个 (K, V) 对的 dataset 上调用时，返回一个 (K,Iterable) **Note:** 如果分组是为了在每一个 key 上执行聚合操作（例如，sum 或average），此时使用 reduceByKey 或 aggregateByKey 性能会更好**Note:** 默认情况下，并行度取决于父RDD 的分区数。可以传入 numTasks 参数进行修改。 |
| **reduceByKey**(*func*, [*numTasks*]) | 按照 key 值进行分组，并对分组后的数据执行归约操作。 |
| **aggregateByKey**(*zeroValue*,*numPartitions*) (*seqOp*,*combOp*, [*numTasks*]) | 当调用（K，V）对的数据集时，返回（K，U）对的数据集，其中使用给定的组合函数和 zeroValue 聚合每个键的值。与 groupByKey 类似，reduce 任务的数量可通过第二个参数进行配置。 |
| **sortByKey**([*ascending*], [*numTasks*]) | 按照 key 进行排序，其中的 key 需要实现 Ordered 特质，即可比较 |
| **join**(*otherDataset*, [*numTasks*]) | 在一个 (K, V) 和 (K, W) 类型的 dataset上调用时，返回一个 (K, (V, W)) pairs 的 dataset，等价于内连接操作。如果想要执行外连接，可以使用leftOuterJoin , rightOuterJoin 和fullOuterJoin 等算子。 |
| **cogroup**(*otherDataset*, [*numTasks*]) | 在一个 (K, V) 对的 dataset 上调用时，返回一个 (K, (Iterable, Iterable))tuples 的 dataset。 |
| **cartesian**(*otherDataset*) | 在一个 T 和 U 类型的 dataset 上调用时，返回一个 (T, U) 类型的 dataset（即笛卡尔积）。 |
| **coalesce**(*numPartitions*) | 将 RDD 中的分区数减少为 numPartitions。 |
| **repartition**(*numPartitions*) | 随机重新调整 RDD 中的数据以创建更多或更少的分区，并在它们之间进行平 |
| **repartitionAndSortWithinPartitions**(*partitioner*) | 根据给定的 partitioner（分区器）对 RDD 进行重新分区，并对分区中的数据按照 key 值进行排序。这比调用 repartition 然后再 sorting（排序）效率更高，因为它可以将排序过程推送到 shuffle 操作所在的机器。 |

- map

```
val list = List(1,2,3)
sc.parallelize(list).map(_ * 10).foreach(println)
// 输出结果： 10 20 30 （这里为了节省篇幅去掉了换行,后文亦同）
```

- filter

```
val list = List(3, 6, 9, 10, 12, 21)
sc.parallelize(list).filter(_ >= 10).foreach(println)
// 输出： 10 12 21
```

- flatMap：flatMap(func) 与 map 类似，但每一个输入的 item 会被映射成 0 个或多个输出的 items（*func* 返回类型需要为 Seq ）。

```
val list = List(List(1, 2), List(3), List(), List(4, 5))
sc.parallelize(list).flatMap(_.toList).map(_ * 10).foreach(println)
// 输出结果 ： 10 20 30 40 50
```

flatMap 这个算子在日志分析中使用概率非常高，这里进行一下演示：拆分输入的每行数据为单个单词，并赋值为 1，代表出现一次，之后按照单词分组并统计其出现总次数，代码如下：

```
val lines = List("spark flume spark",
                 "hadoop flume hive")
sc.parallelize(lines).flatMap(line => line.split(" ")).
map(word=>(word,1)).reduceByKey(_+_).foreach(println)
// 输出:
(spark,2)
(hive,1)
(hadoop,1)
(flume,2)
```

- mapPartitions

与 map 类似，但函数单独在 RDD 的每个分区上运行， *func*函数的类型为 Iterator => Iterator (其中 T 是 RDD 的类型)，即输入和输出都必须是可迭代类型。

```
val list = List(1, 2, 3, 4, 5, 6)
sc.parallelize(list, 3).mapPartitions(iterator => {
  val buffer = new ListBuffer[Int]
  while (iterator.hasNext) {
    buffer.append(iterator.next() * 100)
  }
  buffer.toIterator
}).foreach(println)
//输出结果
100 200 300 400 500 600
```

- mapPartitionsWithIndex

与 mapPartitions 类似，但 *func* 类型为 (Int, Iterator) => Iterator ，其中第一个参数为分区索引。

```
val list = List(1, 2, 3, 4, 5, 6)
```

```
sc.parallelize(list, 3).mapPartitionsWithIndex((index, iterator) => {
  val buffer = new ListBuffer[String]
  while (iterator.hasNext) {
    buffer.append(index + "分区:" + iterator.next() * 100)
  }
  buffer.toIterator
}).foreach(println)
//输出
0 分区:100
0 分区:200
1 分区:300
1 分区:400
2 分区:500
2 分区:600
```

- sample

  数据采样。有三个可选参数：设置是否放回 (withReplacement)、采样的百分比 (fraction)、随机数生成器的种子 (seed)：

```
val list = List(1, 2, 3, 4, 5, 6)
sc.parallelize(list).sample(withReplacement = false, fraction =
0.5).foreach(println)
```

- union：合并两个RDD

```
val list1 = List(1, 2, 3)
val list2 = List(4, 5, 6)
sc.parallelize(list1).union(sc.parallelize(list2)).foreach(println)
// 输出: 1 2 3 4 5 6
```

- intersection：求两个RDD的交集

```
val list1 = List(1, 2, 3, 4, 5)
val list2 = List(4, 5, 6)
sc.parallelize(list1).intersection(sc.parallelize(list2)).foreach(println)
// 输出: 4 5
```

- distinct：去重

```
val list = List(1, 2, 2, 4, 4)
sc.parallelize(list).distinct().foreach(println)
// 输出: 4 1 2
```

- groupByKey：按照键进行分组

```
val list = List(("hadoop", 2), ("spark", 3), ("spark", 5), ("storm", 6),
("hadoop", 2))
sc.parallelize(list).groupByKey().map(x => (x._1,
x._2.toList)).foreach(println)
//输出:
(spark,List(3, 5))
(hadoop,List(2, 2))
(storm,List(6))
```

- reduceKey：按照键进行归约操作

```
val list = List(("hadoop", 2), ("spark", 3), ("spark", 5), ("storm", 6),
("hadoop", 2))
sc.parallelize(list).reduceByKey(_ + _).foreach(println)
//输出
(spark,8)
(hadoop,4)
(storm,6)
```

- sort & sortByKey：按照键进行排序

```
val list01 = List((100, "hadoop"), (90, "spark"), (120, "storm"))
sc.parallelize(list01).sortByKey(ascending = false).foreach(println)
// 输出
(120,storm)
(90,spark)
(100,hadoop)
```

按照指定元素进行排序：

```
val list02 = List(("hadoop",100), ("spark",90), ("storm",120))
sc.parallelize(list02).sortBy(x=>x._2,ascending=false).foreach(println)
// 输出
(storm,120)
(hadoop,100)
(spark,90)
```

- join：在一个 (K, V) 和 (K, W) 类型的 Dataset 上调用时，返回一个 (K, (V, W)) 的 Dataset，等价于内连接操作。如果想要执行外连接，可以使用 leftOuterJoin , rightOuterJoin 和 fullOuterJoin 等算子。

```
val list01 = List((1, "student01"), (2, "student02"), (3, "student03"))
val list02 = List((1, "teacher01"), (2, "teacher02"), (3, "teacher03"))
sc.parallelize(list01).join(sc.parallelize(list02)).foreach(println)
// 输出
(1,(student01,teacher01))
(3,(student03,teacher03))
(2,(student02,teacher02))
```

- cogroup：在一个 (K, V) 对的 Dataset 上调用时，返回多个类型为 (K, (Iterable, Iterable)) 的元组所组成的Dataset。

```
val list01 = List((1, "a"),(1, "a"), (2, "b"), (3, "e"))
val list02 = List((1, "A"), (2, "B"), (3, "E"))
val list03 = List((1, "[ab]"), (2, "[bB]"), (3, "eE"),(3, "eE"))
sc.parallelize(list01).cogroup(sc.parallelize(list02),sc.parallelize(list03))
.fo
reach(println)
// 输出： 同一个 RDD 中的元素先按照 key 进行分组，然后再对不同 RDD 中的元素按照 key 进行
分
组
(1,(CompactBuffer(a, a),CompactBuffer(A),CompactBuffer([ab])))
(3,(CompactBuffer(e),CompactBuffer(E),CompactBuffer(eE, eE)))
(2,(CompactBuffer(b),CompactBuffer(B),CompactBuffer([bB])))
```

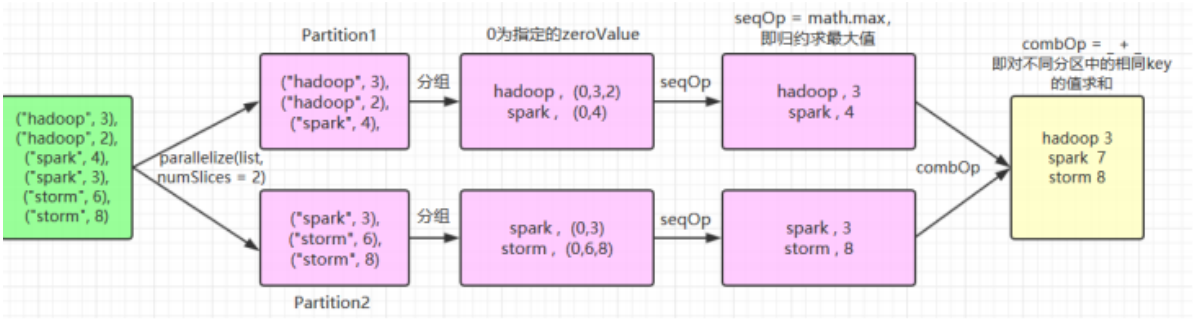
- cartesian：计算笛卡尔积

```
val list1 = List("A", "B", "C")
val list2 = List(1, 2, 3)
sc.parallelize(list1).cartesian(sc.parallelize(list2)).foreach(println)
//输出笛卡尔积
(A,1)
(A,2)
(A,3)
(B,1)
(B,2)
(B,3)
(C,1)
(C,2)
(C,3)
```

- aggregateByKey：当调用（K，V）对的数据集时，返回（K，U）对的数据集，其中使用给定的组合函数和zeroValue 聚合每个键的值。与 groupByKey 类似，reduce 任务的数量可通过第二个参数 numPartitions 进行配置。示例如下

```
// 为了清晰，以下所有参数均使用具名传参
val list = List(("hadoop", 3), ("hadoop", 2), ("spark", 4), ("spark", 3),
("storm", 6), ("storm", 8))
sc.parallelize(list,numSlices = 2).aggregateByKey(zeroValue = 0,numPartitions
=
3)(
      seqOp = math.max(_, _),
      combOp = _ + _
   ).collect.foreach(println)
//输出结果：
(hadoop,3)
(storm,8)
(spark,7)
```

这里使用了 numSlices = 2 指定 aggregateByKey 父操作 parallelize 的分区数量为 2，其执行流程如下




基于同样的执行流程，如果 numSlices = 1 ，则意味着只有输入一个分区，则其最后一步 combOp 相当于是无效的，执行结果为：同样的，如果每个单词对一个分区，即 numSlices = 6 ，此时相当于求和操作，执行结果为：aggregateByKey(zeroValue = 0,numPartitions = 3) 的第二个参数 numPartitions 决定的是输出 RDD 的分区数量

## 2）Spark 常用的 Action 算子如下：

| *Action （动作）* | Meaning（含义） |
| --- | --- |
| **reduce**(*func*) | 使用函数*func*执行归约操作 |
| **collect**() | 以一个 array 数组的形式返回 dataset 的所有元素，适用于小结果集。 |
| **count**() | 返回 dataset 中元素的个数。 |
| **first**() | 返回 dataset 中的第一个元素，等价于 take(1)。 |
| **take**(*n*) | 将数据集中的前 *n* 个元素作为一个 array 数组返回。 |
| **takeSample**(*withReplacement*,*num*, [*seed*]) | 对一个 dataset 进行随机抽样 |
| **takeOrdered**(*n*, *[ordering]*) | 按自然顺序（natural order）或自定义比较器（customcomparator）排序后返回前 *n* 个元素。只适用于小结果集，因为所有数据都会被加载到驱动程序的内存中进行排序。 |
| **saveAsTextFile**(*path*) | 将 dataset 中的元素以文本文件的形式写入本地文件系统、HDFS 或其它 Hadoop 支持的文件系统中。Spark 将对每个元素调用 toString 方法，将元素转换为文本文件中的一行记录。 |
| **saveAsSequenceFile**(*path*) | 将 dataset 中的元素以 Hadoop SequenceFile 的形式写入到本地文件系统、HDFS 或其它 Hadoop 支持的文件系统中。该操作要求 RDD 中的元素需要实现 Hadoop 的 Writable 接口。对于 Scala 语言而言，它可以将 Spark 中的基本数据类 |
| **saveAsObjectFile**(*path*) | 使用 Java 序列化后存储，可以使用 SparkContext.objectFile() 进行加载。(目前仅支持 Java and Scala) |
| **countByKey**() | 计算每个键出现的次数 |

| *Action（动作）* | Meaning（含义） |
| --- | --- |
| **foreach**(*func*) | 遍历 RDD 中每个元素，并对其执行*fun*函数 |

- reduce：使用函数func执行归约操作

```
val list = List(1, 2, 3, 4, 5)
sc.parallelize(list).reduce((x, y) => x + y)
sc.parallelize(list).reduce(_ + _)
// 输出 15
```

- takeOrdered：按自然顺序（natural order）或自定义比较器（custom comparator）排序后返回前 *n* 个元素。需要注意的是 takeOrdered 使用隐式参数进行隐式转换，以下为其源码。所以在使用自定义排序时，需要继承 Ordering[T] 实现自定义比较器，然后将其作为隐式参数引入。

```
// 继承 Ordering[T],实现自定义比较器，按照 value 值的长度进行排序
class CustomOrdering extends Ordering[(Int, String)] {
    override def compare(x: (Int, String), y: (Int, String)): Int
    = if (x._2.length > y._2.length) 1 else -1
}
val list = List((1, "hadoop"), (1, "storm"), (1, "azkaban"), (1, "hive"))
// 引入隐式默认值
implicit val implicitOrdering = new CustomOrdering
sc.parallelize(list).takeOrdered(5)
// 输出： Array((1,hive), (1,storm), (1,hadoop), (1,azkaban)
```

- countByKey：计算每个键出现的次数

```
val list = List(("hadoop", 10), ("hadoop", 10), ("storm", 3), ("storm", 3),
("azkaban", 1))
sc.parallelize(list).countByKey()
// 输出： Map(hadoop -> 2, storm -> 2, azkaban -> 1)
```

- saveAsTextFile：将 dataset 中的元素以文本文件的形式写入本地文件系统、HDFS 或其它 Hadoop 支持的文件系统中。Spark 将对每个元素调用 toString 方法，将元素转换为文本文件中的一行记录。

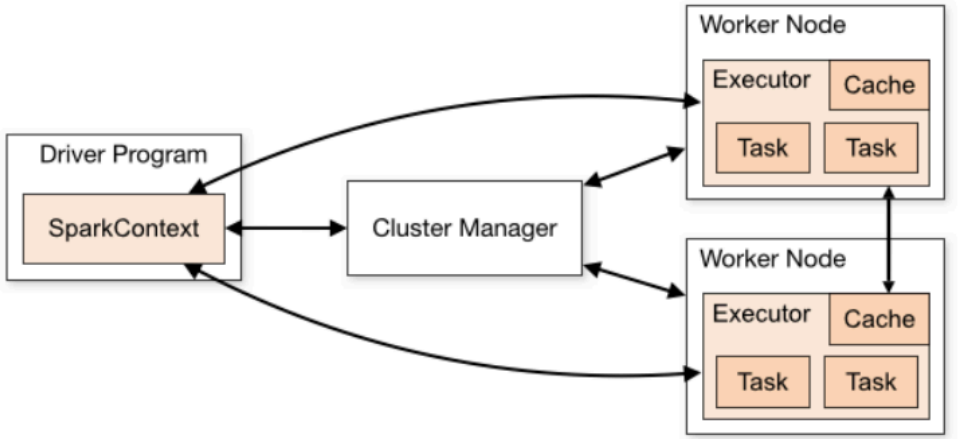
```
val list = List(("hadoop", 10), ("hadoop", 10), ("storm", 3), ("storm", 3),
("azkaban", 1))
sc.parallelize(list).saveAsTextFile("/usr/file/temp")
```

## 7.Spark累加器和广播变量

在 Spark 中，提供了两种类型的共享变量：累加器 (accumulator) 与广播变量 (broadcast variable)：

**累加器**：用来对Driver的信息进行聚合，主要用于累计计数等场景；

**广播变量**：主要用于在节点间高效分发大对象。不把副本变量分发到每个 Task 中，而是将其分发到每个 Executor，Executor 中的所有 Task 共享一个副本变量。




-

> **RDD 流程（结合这张图）**
>
> 1. Driver Program（运行SparkContext）提交 Job → SparkContext → Cluster Manager 请求资源
> 2. Cluster Manager （（可以是 Standalone、YARN、K8s））分配多个 Worker Node，启动 Executor
> 3. Executor 执行 Task（计算 RDD 分区），第一次计算的结果放进 Cache（或磁盘）
> 4. 同一 Executor 下的后续 Task 如果访问到已缓存的分区，**直接从 Cache 读取**，避免重复计算。
> 5. 如果分区丢失（节点故障、Cache 被清理），Driver 根据血缘（lineage）重新调度 Task 计算该分区。

## 8.基于Zookeeper搭建Spark高可用集群

搭建过程略，**提交作业：**

```
# 以client模式提交到yarn
spark-submit \
--class org.apache.spark.examples.SparkPi \
--master yarn \
--deploy-mode client \
--executor-memory 1G \
--num-executors 10 \
/usr/app/spark-2.4.0-bin-hadoop2.6/examples/jars/spark-examples_2.11-2.4.0.jar \
100

# 以client模式提交到standalone集群
spark-submit \
--class org.apache.spark.examples.SparkPi \
--master spark://hadoop001:7077 \
--executor-memory 2G \
--total-executor-cores 10 \
/usr/app/spark-2.4.0-bin-hadoop2.6/examples/jars/spark-examples_2.11-2.4.0.jar \
100

# 以cluster模式提交到standalone集群
spark-submit \
--class org.apache.spark.examples.SparkPi \
--master spark://207.184.161.138:7077 \
```

```
--deploy-mode cluster \
--supervise \  # 配置此参数代表开启监督，如果主应用程序异常退出，则自动重启 Driver
--executor-memory 2G \
--total-executor-cores 10 \
/usr/app/spark-2.4.0-bin-hadoop2.6/examples/jars/spark-examples_2.11-2.4.0.jar \
100
```

> deploy-mode 有 cluster 和 client 两个可选参数，默认为 client 。这里以 Spark On Yarn 模式进行说明：
>
> 1、在 cluster 模式下，Spark Drvier 在应用程序的 Master 进程内运行，该进程由群集上的 YARN 管理，提交作业的客户端可以在启动应用程序后关闭；
> 2、在 client 模式下，Spark Drvier 在提交作业的客户端进程中运行，Master 进程仅用于从 YARN 请求资源。

# 2、Spark SQL

Spark SQL 是 Spark 中的一个子模块，主要用于操作结构化数据。它具有以下特点：
1、能够将 SQL 查询与 Spark 程序无缝混合，允许您使用 SQL 或 DataFrame API 对结构化数据进行查询；
2、支持多种开发语言；
3、支持多达上百种的外部数据源，包括 Hive，Avro，Parquet，ORC，JSON 和 JDBC 等；
4、支持 HiveQL 语法以及 Hive SerDes 和 UDF，允许你访问现有的 Hive 仓库；
5、支持标准的 JDBC 和 ODBC 连接；
6、支持优化器，列式存储和代码生成等特性；
7、支持扩展并能保证容错。

## 1.DataFrame和DataSet

### 1）DataFrame

为了支持结构化数据的处理，Spark SQL 提供了新的数据结构 DataFrame。DataFrame 是一个由具名列组成的数据集。它在概念上等同于关系数据库中的表或 R/Python 语言中的 data frame 。 由于Spark SQL 支持多种语言的开发，所以每种语言都定义了 DataFrame 的抽象，例如Java中Dataset[T]

> DataFrame 和 RDDs 最主要的区别在于一个面向的是结构化数据，一个面向的是非结构化数据；
> DataFrame 内部的有明确 Scheme 结构，即列名、列字段类型都是已知的。

### 2）DataSet

Dataset 也是分布式的数据集合，在 Spark 1.6 版本被引入，它集成了 RDD 和 DataFrame 的优点，具备强类型的特点，同时支持 Lambda 函数，但只能在 Scala 和 Java 语言中使用。在 Spark 2.0 后，为了方便开发者，Spark 将 DataFrame 和 Dataset 的 API 融合到一起，提供了结构化的 API(Structured API)，即用户可以通过一套标准的 API 就能完成对两者的操作。

# 3、Spark Streaming

# 4、Spark相关面试题

## 1.Spark运行模式?

（1）Local：运行在一台机器上。测试用。

（2）Standalone：是 Spark 自身的一个调度系统。 对集群性能要求非常高时用。国内很少使用。

（3）Yarn：采用 Hadoop 的资源调度器。 国内大量使用。

Yarn-client 模式：Driver 运行在 Client 上（不在 AM 里）

Yarn-cluster 模式：Driver 在 AM 上

（4）Mesos：国内很少使用。

（5）K8S：趋势，但是目前不成熟，需要的配置信息太多。

## 2.Spark常用端口号?

（1）4040 spark-shell 任务端口

（2）7077 内部通讯端口。类比 Hadoop 的 8020/9000

（3）8080 查看任务执行情况端口。 类比 Hadoop 的 8088

（4）18080 历史服务器。类比 Hadoop 的 19888

注意：由于 Spark 只负责计算，所有并没有 Hadoop 中存储数据的端口 9870/50070。

## 3.RDD五大特性?

参考：弹性数据集RDDs

## 4.RDD弹性体现在哪里?

主要表现为存储弹性、计算弹性、任务（Task、Stage）弹性、数据位置弹性，具体如下：

（1）自动进行内存和磁盘切换

（2）基于 lineage 的高效容错

（3）Task 如果失败会特定次数的重试

（4）Stage 如果失败会自动进行特定次数的重试，而且只会只计算失败的分片

（5）Checkpoint【每次对 RDD 操作都会产生新的 RDD，如果链条比较长，计算比较笨重，就把数据放在硬盘中】和 persist 【内存或磁盘中对数据进行复用】(检查点、持久化)

（6）数据调度弹性：DAG Task 和资源管理无关

（7）数据分片的高度弹性 repartion

## 5.Spark转换算子（8个）？

**1）单 Value**

（1）map

（2）mapPartitions

（3）mapPartitionsWithIndex

（4）flatMap

（5）groupBy

（6）filter

（7）distinct

（8）coalesce

（9）repartition

（10）sortBy

**2）双vlaue**

（1）intersection

（2）union

（3）subtract

（4）zip

**3）Key-Value**

（1）partitionBy

（2）reduceByKey

（3）groupByKey

（4）sortByKey

（5）mapValues

（6）join

## 6.Spark的行动算子（5个）？

（1）reduce

（2）collect

（3）count

（4）first

（5）take

（6）save

（7）foreach

## 7.map 和 mapPartitions 区别?

（1）map：每次处理一条数据

（2）mapPartitions：每次处理一个分区数据

## 8.Repartition 和 Coalesce 区别?

**1）关系：**

两者都是用来改变 RDD 的 partition 数量的，repartition 底层调用的就是 coalesce 方法：

coalesce（numPartitions, shuffle = true）。

**2）区别：**

repartition 一定会发生 Shuffle（重新分配到新的分区上面），coalesce 根据传入的参数来判断是否发生 Shuffle（默认不Shuffle）。

一般情况下增大 rdd 的 partition 数量使用 repartition，减少 partition

**使用建议：**

> **增加分区** → 必须用 `repartition()`（或 `coalesce(..., shuffle = true)`）
>
> **减少分区且要均衡** → `repartition()` 或 `coalesce(..., shuffle = true)`
>
> **减少分区且不关心均衡** → `coalesce(..., shuffle = false)`（性能好）

## 9.reduceByKey 与 groupByKey 的区别?

reduceByKey：具有预聚合操作。

groupByKey：没有预聚合。

在不影响业务逻辑的前提下，优先采用 reduceByKey。

## 10.Spark中的血缘?

> 在 Spark 里，“血缘”（**Lineage**）指的是 **RDD 或 DataFrame 从最初数据源一步步转化而来的依赖关系链**。它本质上是 Spark 用来实现 **容错机制** 的核心元数据结构，形式上以有向无环图RAG形式存储，每个节点是一个 RDD，每条边是一个 Transformation。

**1）为什么需要血缘**

Hadoop MapReduce 的容错依赖数据落盘（HDFS），而 Spark 走的是**内存计算**，不可能把每个中间结果都落到磁盘，否则性能优势就没了。
Spark 用血缘替代了这种中间落盘的方式：

- 当一个分区丢失（比如节点挂了），Spark 会根据血缘关系图，找到丢失分区的父 RDD 及其操作链，重新执行计算。
- 这样既节省了存储，又保证了计算的可恢复性。

**2）血缘的两种依赖**

Spark 中的依赖分为两类，这也是血缘的重要特征（触发时机 = 遇到 Action 操作）：

| 依赖类型 | 特点 | 举例 | 影响 |
| --- | --- | --- | --- |
| **窄依赖**（Narrow Dependency） | 子 RDD 的每个分区只依赖父 RDD 的**一个或少量分区** | `map`、`filter`、`union`（部分情况） | 容错恢复时只需重算很少分区 |
| **宽依赖**（Wide Dependency） | 子 RDD 的每个分区依赖父 RDD 的**多个分区**（需要 shuffle） | `reduceByKey`、`groupByKey`、`join` | 容错恢复成本高，需要重算整个 Stage |

## 11.Spark任务的划分?

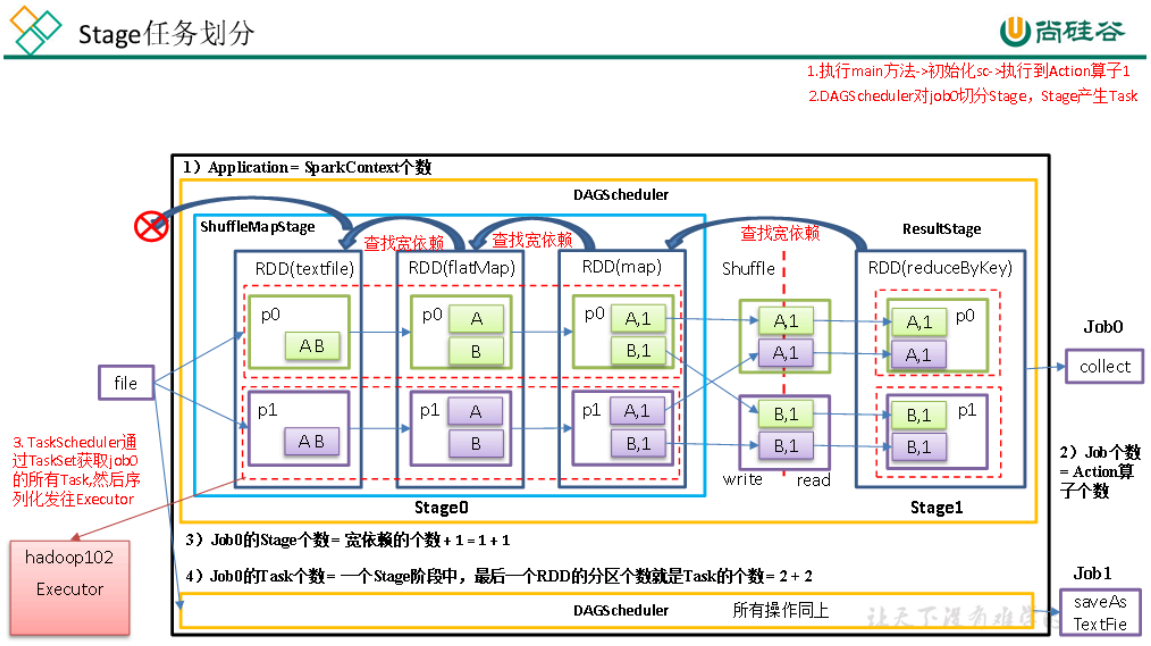
> RDD任务切分中间分为：Application、Job、Stage和Task

（1）Application：初始化一个 SparkContext 即生成一个 Application；

（2）Job：一个 Action 算子就会生成一个 Job；

（3）Stage：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。




> **核心口诀：**
>
> **窄依赖 → 同 Stage；宽依赖 → 切 Stage**
> **Stage 内 Task 数 = 最后 RDD 的分区数**

## 12.RDD中缓存和检查点的区别?

> RDD持久化（Persistence）是 Spark 中为了 **避免重复计算、提升性能** 而将中间结果缓存到内存或磁盘的机制

（1）Cache缓存只是将数据保存起来，不切断血缘依赖。Checkpoint检查点切断血缘依赖。
（2）Cache缓存的数据通常存储在磁盘、内存等地方，可靠性低。Checkpoint的数据通常存储在HDFS等容错、高可用的文件系统，可靠性高。
（3）建议对checkpoint()的RDD使用Cache缓存，这样checkpoint的job只需从Cache缓存中读取数据即可，否则需要再从头计算一次RDD。
（4）如果使用完了缓存，可以通过unpersist()方法释放缓存。

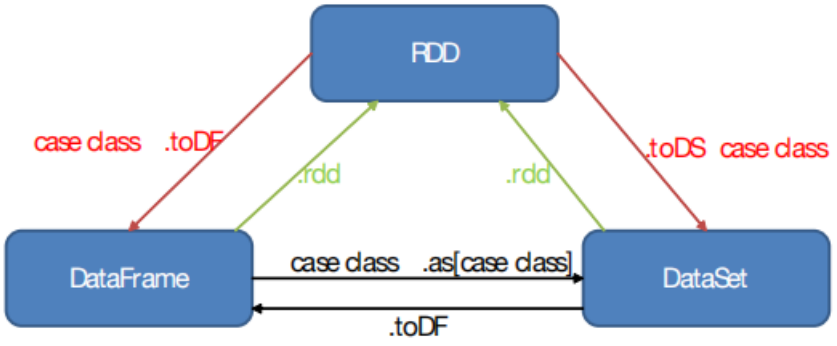
### 1）为什么需要持久化?

1、Spark 的 RDD 是 **惰性计算**（lazy evaluation），每次 Action 都会沿着血缘（lineage）回溯重新计算。
2、如果某个 RDD 被多次使用（被多个 Action 或后续 RDD 引用），不持久化就会重复计算，浪费 CPU 和 I/O。
3、持久化后，RDD 的数据会保留在内存 / 磁盘中，后续直接复用。

### 2）存储级别

## 13.SparkSQL中的RDD、DataFrame、DataSet三者的区别和转换?

> RDD（Spark1.0）=》Dataframe（Spark1.3）=》Dataset（Spark1.6）
>
> 如果同样的数据都给到这三个数据结构，他们分别计算之后，都会给出相同的结果。不同的是他们的执行效率和执行方式。




> DataFrame 和 DataSet 的区别：前者是 row 类型
>
> RDD（面向对象） 和 DataFrame（面向SQL）及 DataSet（两者兼顾） 的区别：前者没有字段和表信息

1、RDDs 适合非结构化数据的处理，而 DataFrame & DataSet 更适合结构化数据和半结构化的处理；

2、DataFrame & DataSet 可以通过统一的 Structured API 进行访问，而 RDDs 则更适合函数式编程的场景；

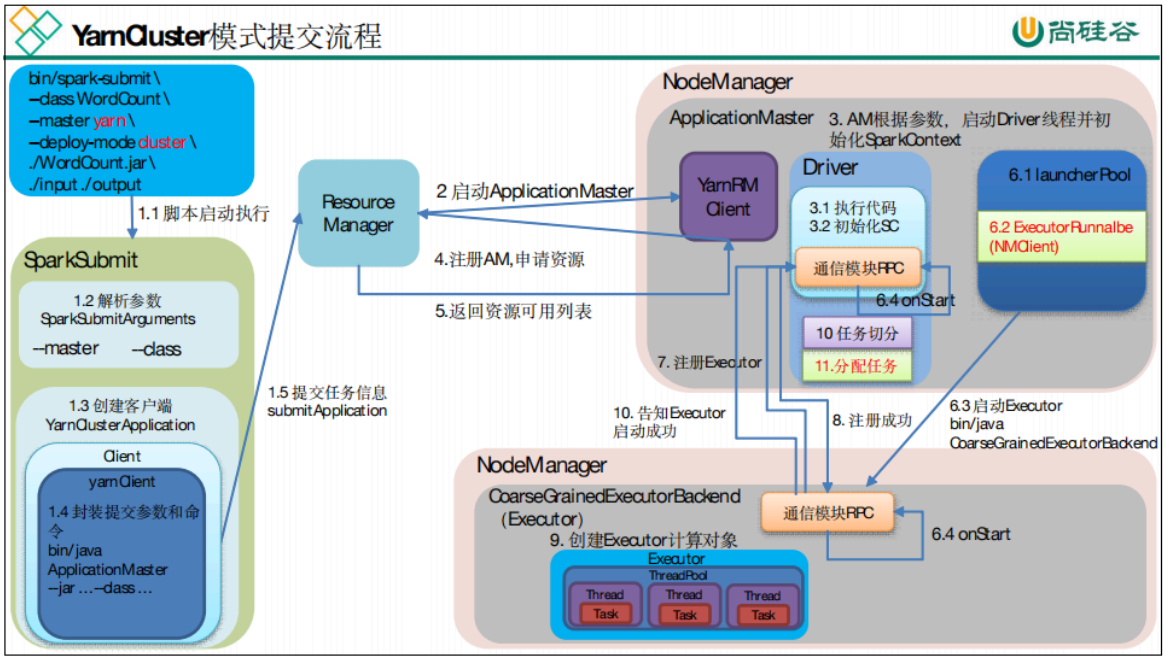
3、相比于 DataFrame 而言，DataSet 是强类型的 (Typed)，有着更为严格的静态类型检查（强类型安全）；DataSets、DataFrames、SQL 的底层都依赖了 RDDs API，并对外提供结构化的访问接口。

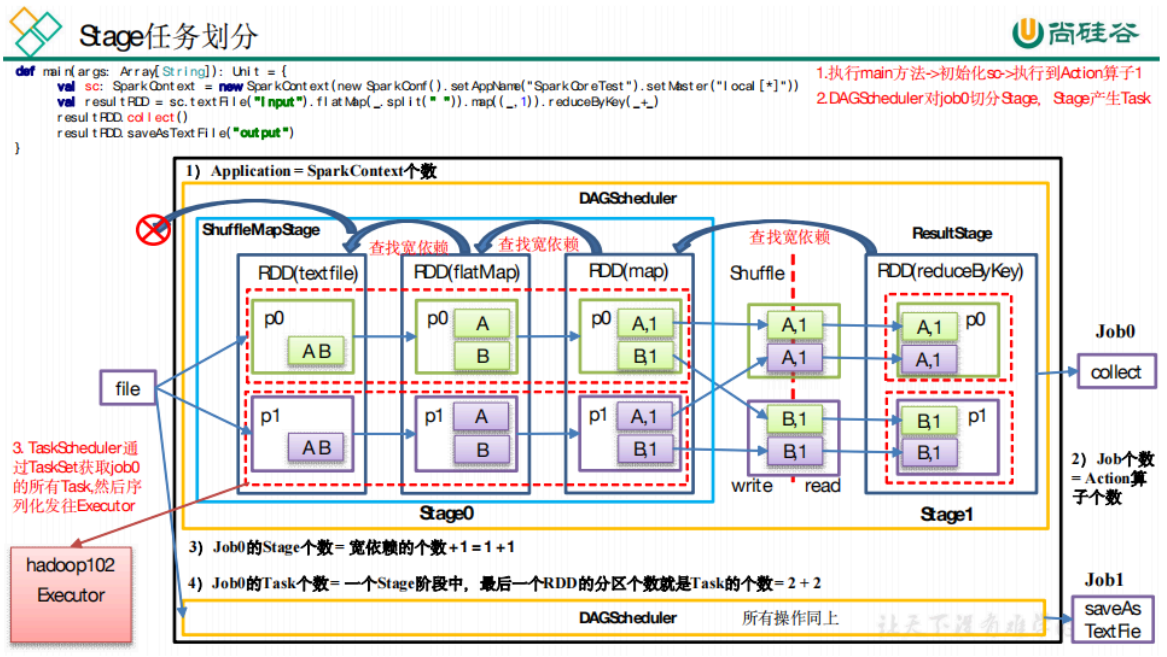
## 14.Hive on Spark 和 Spark on Hive区别?

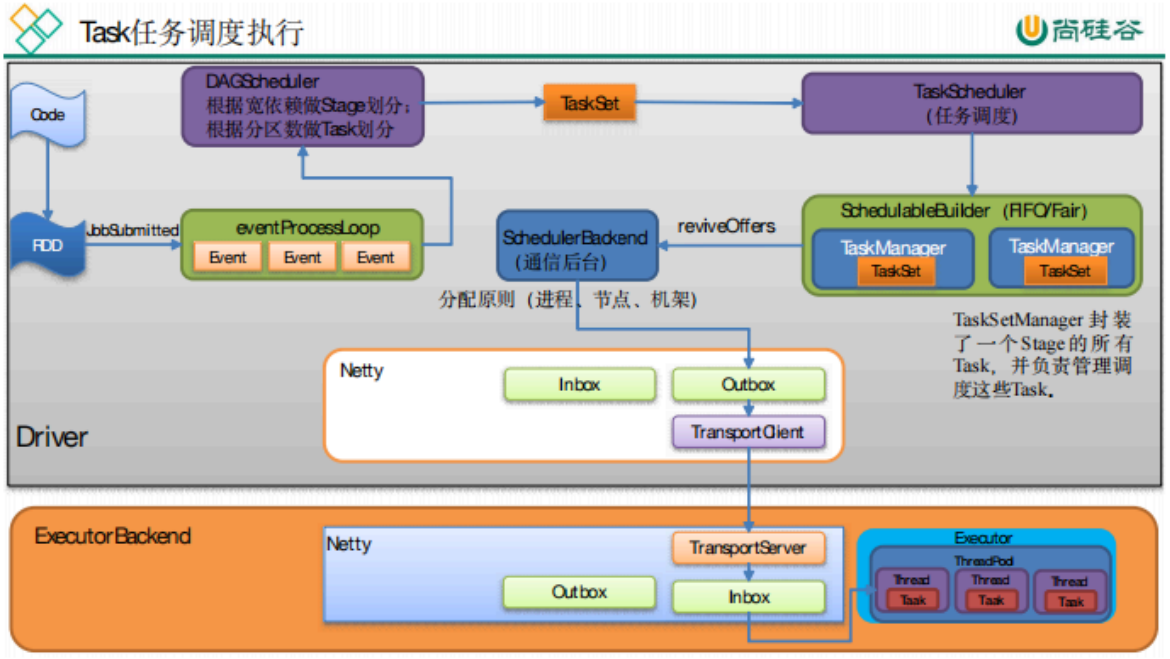
| | 元数据 | 执行引擎 | 语法 | 生态 |
|---|---|---|---|---|
| **Hive on Spark** | MySQL | rdd | HQL | 更加完善 |
| **Spark on Hive (Spark SQL )** | MySQL | df ds | Spark SQL | 有欠缺（权限管理、元数据管理） |
| **内置 Hive** | derby | | | |
| **外置 Hive** | MySQL | | | |

# 15.Spark内核源码?

## 1）提交流程






**Spark on Yarn Cluster 模式提交流程**（`spark-submit` → Yarn Client 申请 AM → AM 启动 Driver → 向 RM 申请 Executor → 划分 Stage & Task → Task 分发到 Executor → 执行完毕释放资源。）

**1. 提交任务**

- 用户通过 `spark-submit` 提交任务：

```
bin/spark-submit \
  --master yarn \
  --deploy-mode cluster \
  --class xxx.Main \
  xxx.jar
```

- `SparkSubmit` 解析参数，生成 `SparkSubmitArguments`，并创建 `YarnClusterApplication`。

**2. 与 Yarn ResourceManager 交互**

1. **Client 启动**（运行在提交端）：
   - 生成 Application 申请（`YarnClient`）。
   - 通过 **ResourceManager** 注册 Application，并申请启动 **ApplicationMaster**（AM）。
2. **ResourceManager 分配 AM 容器**，并通知对应的 NodeManager。

**3. ApplicationMaster 启动**

- NodeManager 在分配的容器里启动 **ApplicationMaster**（包含 Driver）。
- Driver 初始化 SparkContext，准备执行用户代码。
- AM 向 ResourceManager 申请 Executor 资源。

**4. 启动 Executor**

1. **ResourceManager 分配 Executor 容器。**

2. 对应 NodeManager 启动 **CoarseGrainedExecutorBackend**（长期存活，执行多个 Task）。
3. Executor 注册到 Driver。

**5. DAG 划分 Stage**

- Driver 中的 **DAGScheduler** 根据 RDD 依赖关系划分 **Stage**（窄依赖同一个 Stage，宽依赖切分 Stage）。
- 每个 Stage 中按分区数生成多个 **Task**。

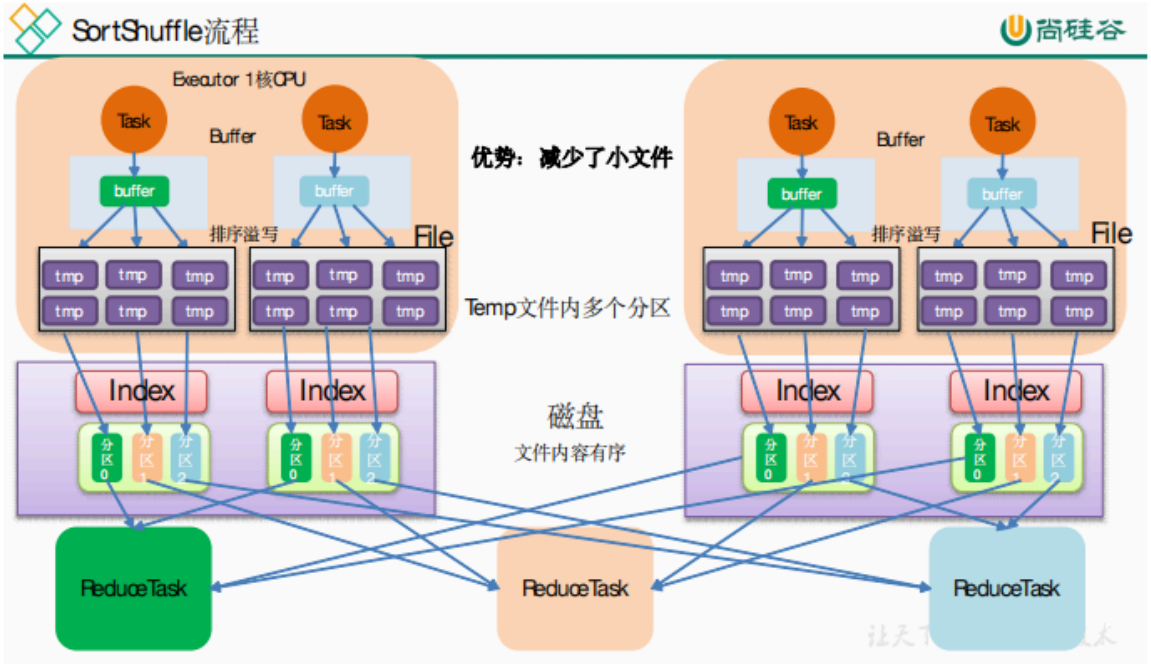
**6. Task 调度执行**

1. **TaskScheduler** 将 Task 集合（TaskSet）发送给 Executor。
2. **Executor** 执行 Task（线程池并发执行），必要时进行数据 Shuffle。
3. 执行结果返回给 Driver。

**7. 任务完成**

- 所有 Stage 完成后，Driver 向 AM 报告完成。
- ApplicationMaster 向 ResourceManager 注销。
- Yarn 释放资源。

## 2）Shuffle流程




Shuffle 是 Spark 宽依赖操作时的跨节点数据重分发过程，分 **Shuffle Write**（Map 端分区+排序+写磁盘+汇报 MapStatus）和 **Shuffle Read**（Reduce 端拉取数据+反序列化+合并）两个阶段。性能瓶颈在于磁盘 IO、网络 IO 和数据倾斜，可通过调节参数、压缩、合并小文件等优化。

**Step 1：Map 端写（Shuffle Write）**

1. **上游 Stage（Map 任务）执行完后**，每个 Task 会把结果按下游分区规则（Hash 分区、Range 分区等）拆分成多个 **分区块**（Partition）。
2. 在内存中用 **SortShuffleWriter** 对数据按分区 ID 排序，并按 Key 排序（如果需要）。
3. 排序完成后，数据会被写入磁盘（一个分区一个文件或合并文件，带索引）。
4. **生成 MapStatus**（记录每个分区的大小、位置），并汇报给 Driver。

> **优化点**：
>
> - **BypassMergeSortShuffle**（小分区场景跳过排序）
> - **Shuffle 文件合并**（减少小文件）
> - **内存缓冲大小（spark.shuffle.file.buffer）**

**Step 2：Reduce 端读（Shuffle Read）**

1. 下游 Stage 的 Task 启动后，Driver 会告诉它需要从哪些节点、哪些文件里拉取数据。
2. **BlockManager** 负责通过网络（Netty）从多个 Map 端节点拉取对应分区的数据。
3. 拉取的数据会先放入内存缓冲，不够时溢写磁盘。
4. 数据会被反序列化，并按需要进行合并（Aggregation）或排序。
5. Reduce Task 开始计算（如 `reduceByKey` 聚合）。

> **优化点**：
>
> - **spark.reducer.maxSizeInFlight**（每次拉取数据量）
> - **spark.shuffle.io.maxRetries / retryWait**（网络容错）
> - **spark.memory.fraction**（调整执行和存储内存比例）

# 16.Spark为什么比MR快?

**1、内存&硬盘**

（1）MR 在 Map 阶段会在溢写阶段将中间结果频繁的写入磁盘，在 Reduce 阶段再从磁盘拉取数据。频繁的磁盘 IO 消耗大量时间。
（2）Spark 不需要将计算的中间结果写入磁盘。这得益于 Spark 的 RDD，在各个 RDD的分区中，各自处理自己的中间结果即可。在迭代计算时，这一优势更为明显。

**2、Spark RAG任务划分减少了不必要的 Shuffle**

（1）对 MR 来说，每一个 Job 的结果都会落地到磁盘。后续依赖于次 Job 结果的 Job，会从磁盘中读取数据再进行计算。
（2）对于 Spark 来说，每一个 Job 的结果都可以保存到内存中，供后续 Job 使用。配合 Spark 的缓存机制，大大的减少了不必要的 Shuffle

**3）资源申请粒度：进程&线程**

开启和调度进程的代价一般情况下大于线程的代价。

（1）MR 任务以进程的方式运行在 Yarn 集群中。N 个 MapTask 就要申请 N 个进程
（2）Spark 的任务是以线程的方式运行在进程中。N 个 MapTask 就要申请 N 个线程。

# 17.Spark Shuffle和Hadoop Shuffle区别?

（1）Hadoop 不用等所有的 MapTask 都结束后开启 ReduceTask；Spark 必须等到父 Stage都完成，才能去 Fetch 数据。
（2）Hadoop 的 Shuffle 是必须排序的，那么不管是 Map 的输出，还是 Reduce 的输出，都是分区内有序的，而 Spark 不要求这一点。

## 18.Spark提交作业参数?

**在提交任务时的几个重要参数**

executor-cores —— 每个 executor 使用的内核数，默认为 1，官方建议 2-5 个，我们企业是 4 个
num-executors —— 启动 executors 的数量，默认为 2
executor-memory —— executor 内存大小，默认 1G
driver-cores —— driver 使用内核数，默认为 1
driver-memory —— driver 内存大小，默认 512M

## 19.Spark任务使用什么进行提交，JavaEE界面还是脚本?

Shell 脚本，或者编写Java封装代码；海豚调度器（DolphinScheduler）可以通过页面提交 Spark 任务。

## 20.Spark操作数据库时，如何减少Spark运行中的数据库连接数?

> `foreach` → **每条记录开一个连接，开销大、连接数爆炸**
> `foreachPartition` → **每个分区共用一个连接，性能好、连接可控、易批量操作**

**限制分区数** + **批量操作** + **连接池复用** + **foreachPartition替换foreach** 等等

## 21.Spark数据倾斜?

**出现原因：**

Spark 数据倾斜（Data Skew）是指在 **shuffle 阶段**，某些 key 的数据量特别大，导致某些 task 数据处理远远多于其他 task，从而造成整个 job 运行变慢甚至 OOM。

**常见症状：**

1、Spark Web UI 上，某几个 task 执行时间极长，Stage 一直卡在最后几个 task
2、Driver 或 Executor 报 OOM
3、Shuffle 文件极大，GC 时间长

**优化：**

1、减少shuffle数据量，避免使用groupByKey，改用reduceByKey、aggregateByKey，在map端先做聚合
2、打散热点key（随机加盐）
3、调整合理分区数/增加并行度
4、广播小表，可以用 `broadcast` 广播到每个 Executor
5、预聚合/数据预处理，在数据进入 Spark 前，在 Hive / 数据源侧提前做汇总
6、倾斜 Join 特殊处理，把热点数据过滤出来，先对非热点数据正常 join，再对热点 key 用加盐法

# 四、Hive

## 1、Hive基础

Hive 是一个构建在 Hadoop 之上的数据仓库，它可以将结构化的数据文件映射成表，并提供类 SQL 查询功能，用于查询的 SQL 语句会被转化为 MapReduce 作业，然后提交到 Hadoop 上运行。

**特点**：

1. 简单、容易上手 (提供了类似 sql 的查询语言 hql)，使得精通 sql 但是不了解 Java 编程的人也能很好地进行大数据分析；

2. 灵活性高，可以自定义用户函数 (UDF) 和存储格式；
3. 为超大的数据集设计的计算和存储能力，集群扩展容易;
4. 统一的元数据管理，可与 presto / impala / sparksql 等共享数据；
5. 执行延迟高，不适合做数据的实时处理，但适合做海量数据的离线处理。

**本质：**

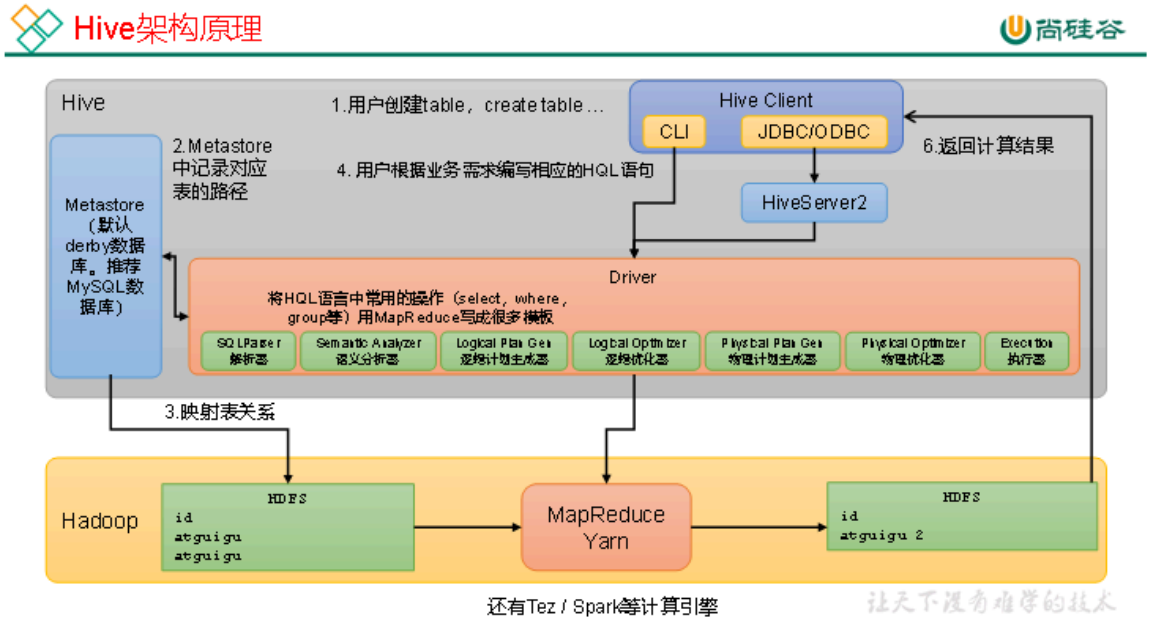
Hive是一个Hadoop客户端，用于将HQL（Hive SQL）转化成MapReduce程序。

（1）Hive中每张表的数据存储在HDFS

（2）Hive分析数据底层的实现是MapReduce（也可配置为Spark或者Tez）

（3）执行程序运行在Yarn上

# 1.Hive架构原理




**1）用户接口Client：**CLI（command-line interface）、JDBC/ODBC

**2）元数据Metastore：**元数据包括：数据库（默认是default）、表名、表的拥有者、列/分区字段、表的类型（是否是外部表）、表的数据所在目录等。默认存储在自带的derby数据库中，由于derby数据库只支持单客户端访问，生产环境中为了多人开发，推荐使用MySQL存储Metastore

**3）驱动器Driver：**将Hive语言sql转为底层计算引擎的执行结果

> Hive中的表在Hadoop中是目录；Hive中的数据在Hadoop中是文件。

## 1）HQL执行流程

Hive 在执行一条 HQL 的时候，会经过以下步骤：

1. 语法解析（SQLParser）：Antlr 定义 SQL 的语法规则，完成 SQL 词法，语法解析，将 SQL 字符串转化为抽象语法树AST Tree；
2. 语义解析（Semantic Analyzer）：遍历 AST Tree，将AST进一步抽象出查询的基本组成单元 QueryBlock；
3. 生成逻辑执行计划（Logical Plan Gen）：遍历 QueryBlock，翻译为执行操作树 OperatorTree；
4. 优化逻辑执行计划（Logical Optimizer）：逻辑层优化器进行 OperatorTree 变换，合并不必要的 ReduceSinkOperator，减少 shuffle 数据量，对逻辑计划进行优化；

5. 生成物理执行计划（Physical Plan Gen）：遍历 OperatorTree，翻译为 MapReduce 任务，根据优化后的逻辑计划生成物理计划；
6. 优化物理执行计划（Physical Optimizer）：物理层优化器进行 MapReduce 任务的变换，生成最终的执行计划。
7. 执行计划（Execution）：执行该计划，得到查询结果并返回给客户端。

# 2、Hive相关面试题

## 1.Hive的架构?

参考Hive的架构

## 2.HQL转换为MR流程?

参考HQL执行流程

## 3.Hive和数据库比较

Hive 和数据库除了拥有类似的查询语言，再无类似之处。

**1）数据存储位置**

Hive 存储在 HDFS 。数据库将数据保存在块设备或者本地文件系统中。

**2）数据更新**

Hive 中不建议对数据的改写。而数据库中的数据通常是需要经常进行修改的。

**3）执行延迟**

Hive 执行延迟较高。数据库的执行延迟较低。当然，这个是有条件的，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。

**4）数据规模**

Hive 支持很大规模的数据计算；数据库可以支持的数据规模较小。

## 4.内部表和外部表?

| 特性 | 内部表（Managed） | 外部表（External） |
| --- | --- | --- |
| 数据存储 | Hive 仓库目录 | 用户指定的 HDFS 路径 |
| 删除表时 | 元数据 + 数据一并删除 | 仅删除元数据，保留数据 |
| 数据管理权 | Hive 全面管理 | 用户/外部系统管理 |
| 适用场景 | 临时表、测试表、ETL 中间表 | 共享数据、持久数据 |

元数据、原始数据

**1）删除数据时**

内部表：元数据、原始数据，全删除

外部表：元数据 只删除

**2）在公司生产环境下，什么时候创建内部表，什么时候创建外部表?**

在公司中绝大多数场景都是外部表。

自己使用的临时表，才会创建内部表；

## 5.系统函数

**1）数值函数**

（1）round：四舍五入；（2）ceil：向上取整；（3）floor：向下取整

**2）字符串函数**

（1）substring：截取字符串；（2）replace：替换；（3）regexp_replace：正则替换

（4）regexp：正则匹配；（5）repeat：重复字符串；（6）split：字符串切割

（7）nvl：替换 null 值；（8）concat：拼接字符串；

（9）concat_ws：以指定分隔符拼接字符串或者字符串数组；

（10）get_json_object：解析 JSON 字符串

**3）日期函数**

（1）unix_timestamp：返回当前或指定时间的时间戳

（2）from_unixtime：转化 UNIX 时间戳（从 1970-01-01 00:00:00 UTC 到指定时间的秒数）到当前时区的时间格式

（3）current_date：当前日期

（4）current_timestamp：当前的日期加时间，并且精确的毫秒

（5）month：获取日期中的月；（6）day：获取日期中的日

（7）datediff：两个日期相差的天数（结束日期减去开始日期的天数）

（8）date_add：日期加天数；（9）date_sub：日期减天数

（10）date_format：将标准日期解析成指定格式字符串

**4）流程控制函数**

（1）case when：条件判断函数

（2）if：条件判断，类似于 Java 中三元运算符

**5）集合函数**

（1）array：声明 array 集合

（2）map：创建 map 集合

（3）named_struct：声明 struct 的属性和值

（4）size：集合中元素的个数

（5）map_keys：返回 map 中的 key

（6）map_values：返回 map 中的 value

（7）array_contains：判断 array 中是否包含某个元素

（8）sort_array：将 array 中的元素排序

**6）聚合函数**

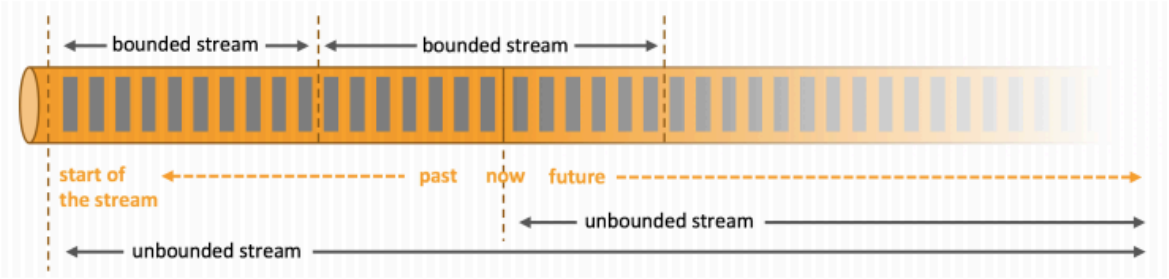
（1）collect_list：收集并形成 list 集合，结果不去重

（2）collect_set：收集并形成 set 集合，结果去重

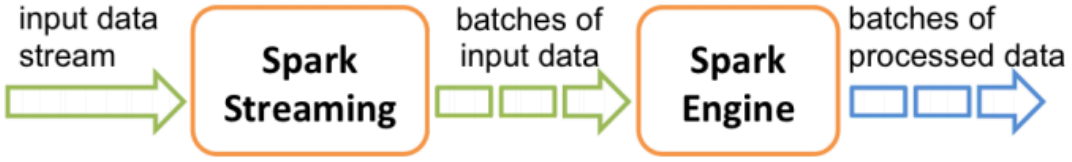
# 五、Flink

Flink 是一个分布式的流处理框架，它能够对有界和无界的数据流进行高效的处理。Flink 的核心是流处理，当然它也能支持批处理，Flink 将批处理看成是流处理的一种特殊情况，即数据流是有明确界限的。这和 Spark Streaming 的思想是完全相反的，Spark Streaming 的核心是批处理，它将流处理看成是批处理的一种特殊情况，即把数据流进行极小粒度的拆分，拆分为多个微批处理。

Flink 有界数据流和无界数据流：




Spark Streaming 数据流的拆分：

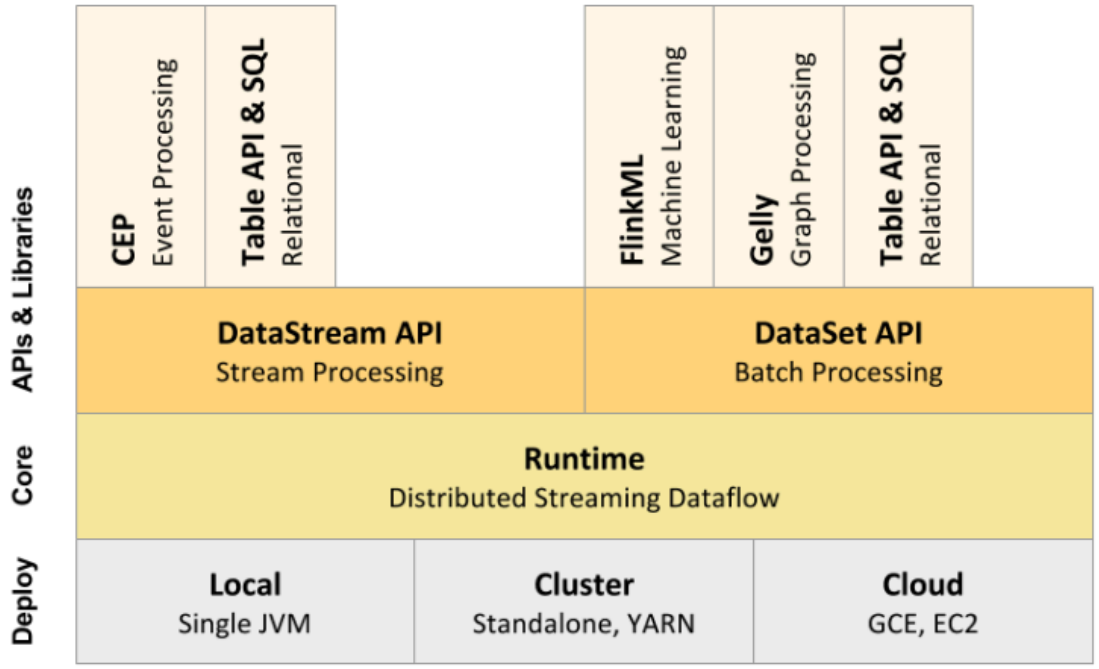



> Flink以流处理为根本，基本数据类型是数据流。以及事件Event序列
> Spark以批处理为根本，采用RDD模型，将DAG划分为不同的stage

# 1、Flink基础

## 1.核心架构

Flink 采用分层的架构设计，从而保证各层在功能和职责上的清晰。如下图所示，由上而下分别是 API & Libraries 层、Runtime 核心层以及物理部署层：




- **API & Libraries 层**：这一层主要提供了编程 API 和 顶层类库：

1、编程 API : 用于进行流处理的 DataStream API 和用于进行批处理的 DataSet API；
2、顶层类库：包括用于复杂事件处理的 CEP 库；用于结构化数据查询的 SQL & Table 库，以及基于批处理的机器学习库 FlinkML 和 图形处理库 Gelly。

- **Runtime 核心层**

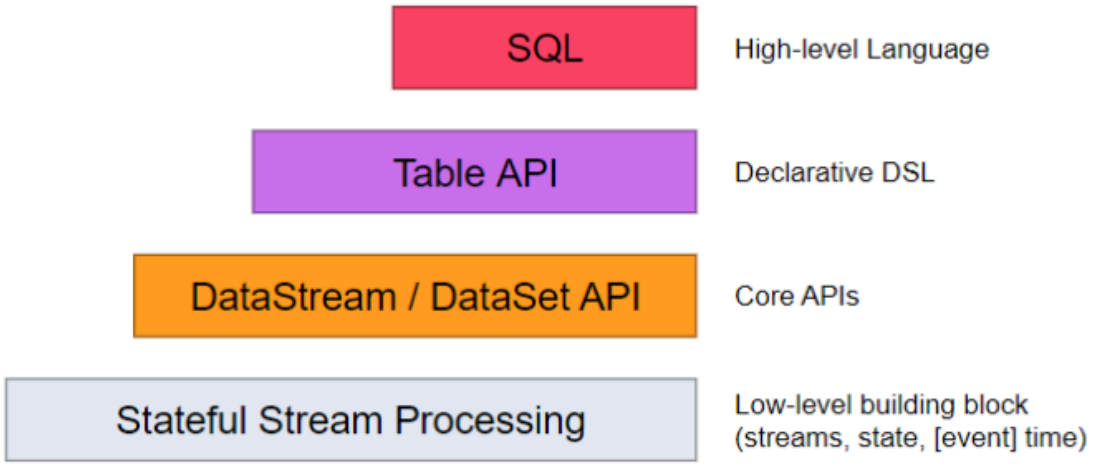
这一层是 Flink 分布式计算框架的核心实现层，包括作业转换，任务调度，资源分配，任务执行等功能，基于这一层的实现，可以在流式引擎下同时运行流处理程序和批处理程序。

- **物理部署层**

Flink 的物理部署层，用于支持在不同平台上部署运行 Flink 应用。

## 2.Flink分层API

在上面介绍的 API & Libraries 这一层，Flink 又进行了更为具体的划分。具体如下：




按照如上的层次结构，API 的一致性由下至上依次递增，接口的表现能力由下至上依次递减，各层的核心功能如下：

**1、SQL & Table API**

SQL & Table API 同时适用于批处理和流处理，这意味着你可以对有界数据流和无界数据流以相同的语义进行查询，并产生相同的结果。除了基本查询外， 它还支持自定义的标量函数，聚合函数以及表值函数，可以满足多样化的查询需求。

**2、DataStream & DataSet API**

DataStream & DataSet API 是 Flink 数据处理的核心 API，支持使用 Java 语言或 Scala 语言进行调用，提供了数据读取，数据转换和数据输出等一系列常用操作的封装。

**3、Stateful Stream Processing**

Stateful Stream Processing 是最低级别的抽象，它通过 Process Function 函数内嵌到 DataStream API 中。 Process Function 是 Flink 提供的最底层 API，具有最大的灵活性，允许开发者对于时间和状态进行细粒度的控制

# 3.核心组件

Flink 核心架构的第二层是 Runtime 层（提交和执行任务）， 该层采用标准的 Master - Slave 结构， 其中，Master 部分又包含了三个核心组件：Dispatcher、ResourceManager 和 JobManager，而 Slave 则主要是 TaskManager 进程。它们的功能分别如下：

**JobManagers** (也称为 *masters*)： JobManagers 接收由 Dispatcher 传递过来的执行程序，该执行程序包含了作业图 (JobGraph)，逻辑数据流图 (logical dataflow graph) 及其所有的 classes 文件以及第三方类库 (libraries) 等等 。紧接着 JobManagers 会将 JobGraph 转换为执行图(ExecutionGraph)，然后向 ResourceManager 申请资源来执行该任务，一旦申请到资源，就将执行图分发给对应的 TaskManagers 。因此每个作业 (Job) 至少有一个 JobManager；高可用部署下可以有多个 JobManagers，其中一个作为 *leader*，其余的则处于 *standby* 状态。

**TaskManagers** (也称为 *workers*) : TaskManagers 负责实际的子任务 (subtasks) 的执行，每个 TaskManagers 都拥有一定数量的 slots。Slot 是一组固定大小的资源的合集 (如计算能力，存储空间)。TaskManagers 启动后，会将其所拥有的 slots 注册到 ResourceManager 上，由ResourceManager 进行统一管理。

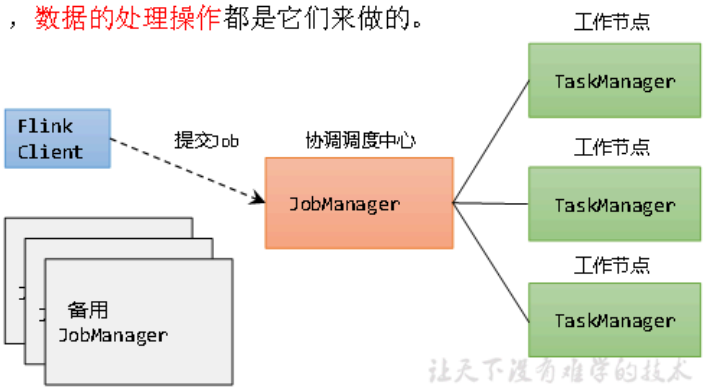
**Dispatcher**：负责接收客户端提交的执行程序，并传递给 JobManager 。除此之外，它还提供了一个 WEB UI 界面，用于监控作业的执行情况。

**ResourceManager**： 负责管理 slots 并协调集群资源。ResourceManager 接收来自JobManager 的资源请求，并将存在空闲 slots 的 TaskManagers 分配给 JobManager 执行任务。Flink 基于不同的部署平台，如 YARN , Mesos，K8s 等提供了不同的资源管理器，当TaskManagers 没有足够的 slots 来执行任务时，它会向第三方平台发起会话来请求额外的资源。

Flink提交作业和执行任务，需要几个关键组件：

- 客户端（Client）：代码由客户端获取并做转换，之后提交给JobManger
- JobManager就是Flink集群里的“管事人”，对作业进行中央调度管理；而它获取到要执行的作业后，会进一步处理转换，然后分发任务给众多的TaskManager。
- TaskManager，就是真正“干活的人”，数据的处理操作都是它们来做的。

注意：Flink是一个非常灵活的处理框架，它支持多种不同的部署场景，还可以和不同的资源管理平台方便地集成。所以接下来我们会先做一个简单的介绍，让大家有一个初步的认识，之后再展开讲述不同情形下的Flink部署。




# 2、Flink部署模式

## 1.部署模式

主要有以下三种：会话模式（Session Mode）、单作业模式（Per-Job Mode）、应用模式（Application Mode）

> 区别主要在于：集群的生命周期以及资源的分配方式；以及应用的main方法到底在哪里执行——客户端（Client）还是JobManager

- **会话模式**

  会话模式最符合常规思维，需要先启动一个集群保持一个会话，这个会话通过客户端提交作业，集群启动时所有资源都已经确定，所以所有提交的作业会竞争集群中资源。

- **单作业模式**

  会话模式因为资源共享会导致很多问题，所以为了更好隔离资源，可以考虑为每个提交的作业启动一个集群，作业完成后，集群就会关闭，左右资源也会释放，单作业模式一般借助一些资源管理框架来启动集群，比如YARN、k8s。

- **应用模式**

  前面两种模式，应用代码都是在客户端上执行，然后由客户端提交给JobManager，但是这种方式客户端需要占用大量网络带宽，由此可以不要客户端，直接把应用提交到JobManager上运行，需要为每一个提交的应用单独启动一个JobManager，生命周期同应用结束而结束。

## 2.YARN运行模式

YARN上部署的过程是：客户端把Flink应用提交给Yarn的ResourceManager，Yarn的ResourceManager会向Yarn的NodeManager申请容器。在这些容器上，Flink会部署JobManager和TaskManager的实例，从而启动集群。Flink会根据运行在JobManger上的作业所需要的Slot数量动态分配TaskManager资源。

## 3.Flink如何跟YARN关联起来的?

Flink 内部有一个 `flink-yarn` 模块（打包时的依赖），它实现了和 YARN 的通信逻辑。大致过程如下：

1. **提交作业**
   你执行命令：

   ```
   ./bin/flink run-application -t yarn-application myjob.jar
   ```

2. **Flink 客户端向 YARN 申请资源**
   - 客户端会调用 YARN 的 **ResourceManager API**，请求一个 ApplicationMaster 容器。
   - 这个 ApplicationMaster 在 Flink 里就是 **JobManager**。
3. **启动 JobManager (ApplicationMaster)**
   - YARN 在某个 NodeManager 节点里启动 JobManager。
   - JobManager 会向 YARN 继续申请 TaskManager 容器。
4. **启动 TaskManagers**
   - YARN 根据 JobManager 的请求，分配若干容器。
   - 每个容器里启动一个 TaskManager 进程。
   - TaskManagers 注册到 JobManager。
5. **任务调度和执行**
   - JobManager 拆分 DAG，分配 task 到各个 TaskManager。
   - Flink 程序开始流式/批处理计算。
6. **资源回收**
   - 作业执行完毕，JobManager 通知 YARN 释放容器。
   - YARN 收回资源。

### 1）Flink 在 YARN 上有两种常见模式：

- **YARN Session 模式**
  先起一个 Session（`yarn-session.sh`），启动一个长期存在的 Flink 集群（JobManager + TaskManagers）。
  后续的 `flink run` 作业可以复用这个集群。
- **YARN Per-Job 单作业模式**
  每提交一个 Flink 作业，YARN 就单独启动一个 JobManager + TaskManagers 集群，作业结束后资源释放。
  （对应 `-t yarn-per-job` 或 `yarn-cluster` 模式）

### 2）关联配置

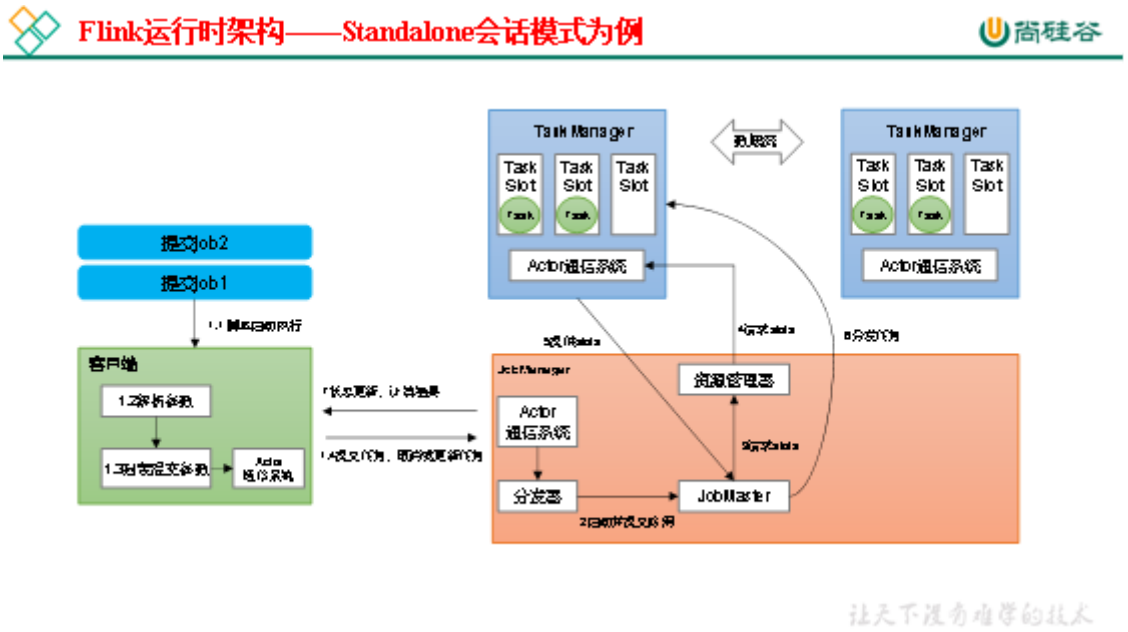
要让 Flink 和 YARN 正常关联，需要：

- `flink-conf.yaml` 里正确配置 **HDFS 和 YARN 的配置文件目录**：

  ```
  yarn.application-attempts: 2
  yarn.ship-files: /etc/hadoop/conf
  ```

- `HADOOP_CONF_DIR` 或 `YARN_CONF_DIR` 环境变量指向 Hadoop 的配置目录。Flink 启动时会读取这些配置，找到 YARN 的 ResourceManager。

# 3、Flink运行时架构




作业管理器JobManager：包含JobMaster、ResourceManager、Dispatcher
任务管理器TaskManager：具体工作进程

## 1.核心概念

### 1）并行度（Parallelism）

在Flink执行过程中，每一个算子（operator）可以包含一个或多个子任务（subtask），这些子任务在不同的线程、不同的物理机或不同的容器中完全独立地执行。一个特定算子的子任务（subtask）的个数被称之为其并行度（parallelism），包含并行子任务的数据流，就是并行数据流。

**并行度设置：**

1、代码中设置，只针对当前算子有效：`stream.map(word -> Tuple2.of(word, 1L)).setParallelism(2);`

2、提交应用时设置

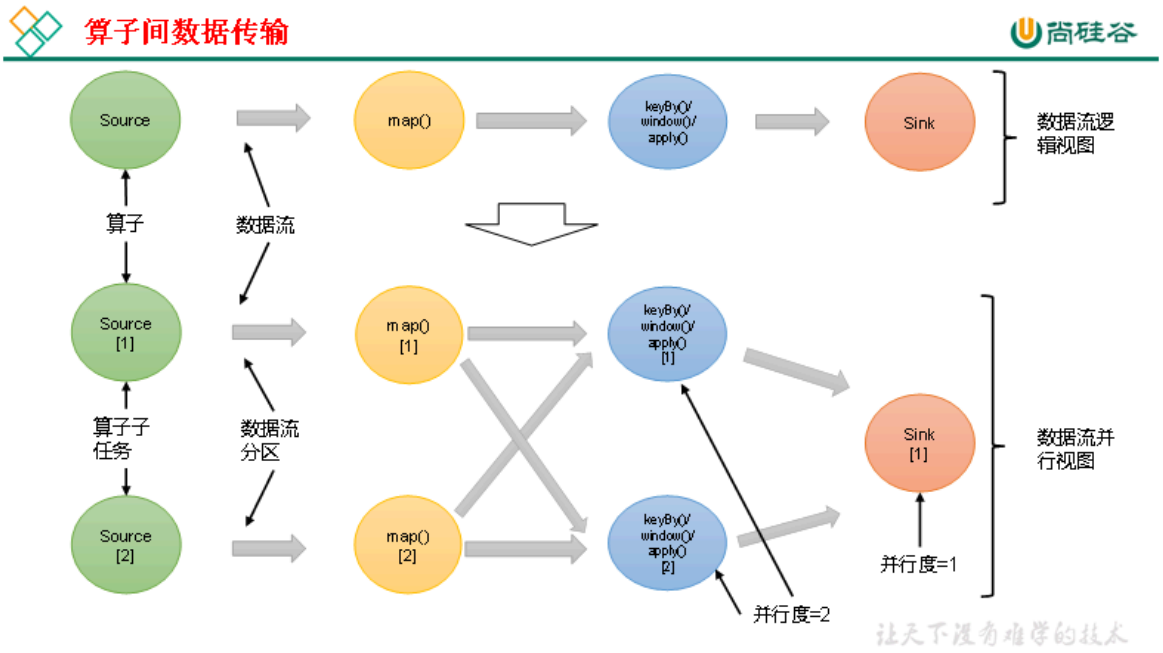
```
#-p参数来指定当前应用程序执行的并行度
bin/flink run -p 2 -c com.atguigu.wc.SocketStreamWordCount
./FlinkTutorial-1.0-SNAPSHOT.jar
```

3、配置文件中设置：flink-conf.yaml

```
parallelism.default: 2
```

> 在开发环境中，没有配置文件，默认并行度就是当前机器的cpu核心数

## 2）算子链（Operator Chain）

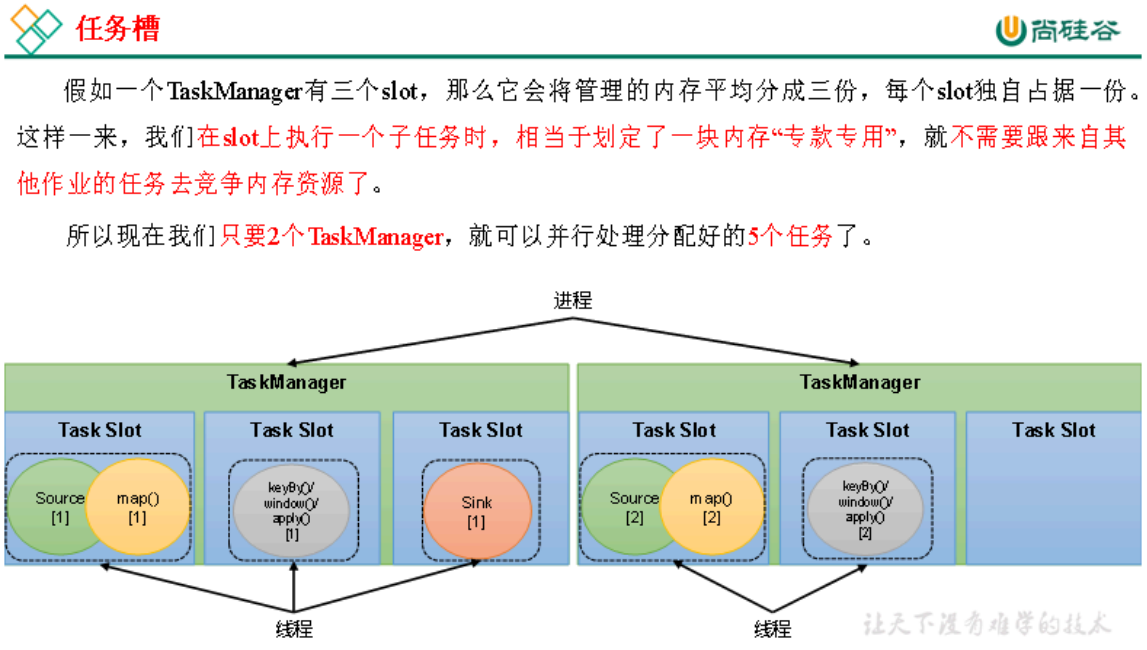



- 一对一 模式：map、filter、flatMap
- 重分区 模式：keyBy、window

> **算子链优化（合并）的本质：**
>
> 1. **减少线程切换**：从5线程 → 2线程（前4个Subtask可共享线程）
> 2. **避免序列化开销**：链内算子间直接内存传输
> 3. **提升吞吐量**：降低网络缓冲区交换频率

## 3）任务槽（Task Slots）

Flink中每一个TaskManager都是一个JVM进程，它可以启动多个独立的线程，来并行执行多个子任务（subtask）。为了控制并发量，我们需要在TaskManager上对每个任务运行所占用的资源做出明确的划分，这就是所谓的任务槽（task slots）。




> **三者之间的关系：**

> **并行度**：决定有多少个 SubTask。
> **算子链**：决定 SubTask 是拆成多个 Task 还是合并在一个 Task 里。
> **任务槽**：决定这些 Task 怎么分布到集群里。

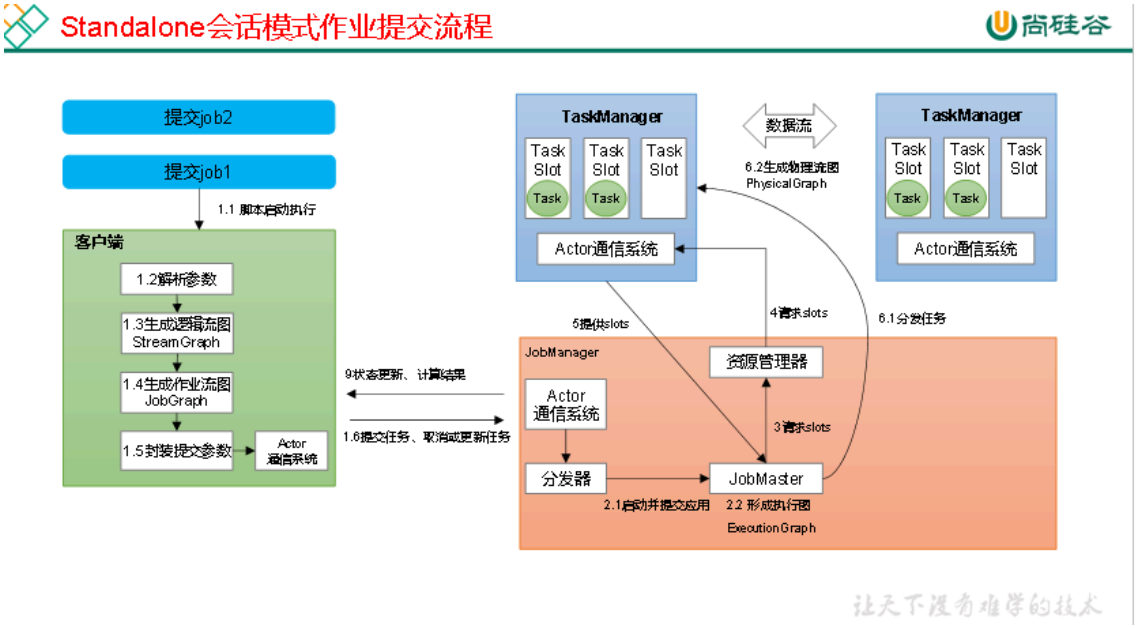
举个例子：

```
env.setParallelism(4);

source -> map -> filter -> keyBy -> sum
```

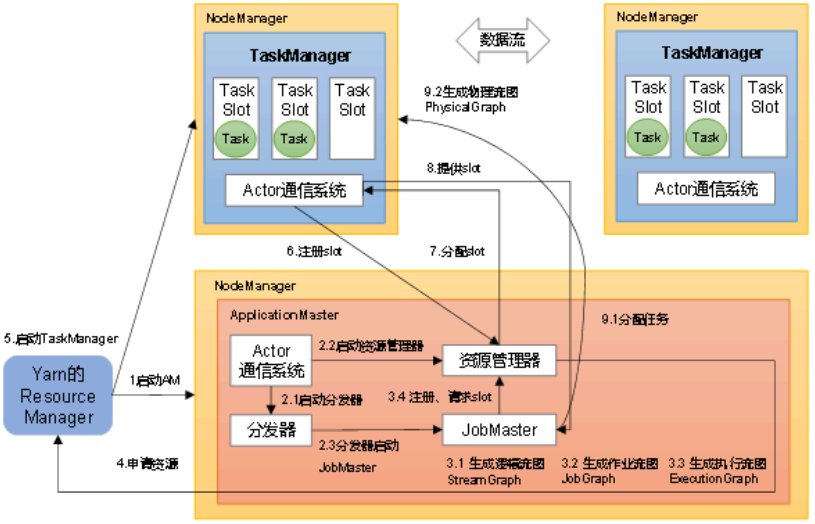
1. `source -> map -> filter` 可以 chain 在一起 → 形成 4 个 SubTask（并行度 4）。
2. `keyBy` 会触发 shuffle，所以 `sum` 不能和前面 chain 在一起，也会形成 4 个 SubTask。
3. 总共 8 个 SubTask。
4. 如果每个 TaskManager 有 2 个 Slot，集群里有 2 个 TaskManager → 共 4 个 Slot，显然不够（需要8个Slot） → Flink 会报错或者等待资源。

## 2.作业提交流程

- **Standalone会话模式**




- **Yarn应用模式**






# 4、DataStream API

## 1.执行环境

**创建执行环境：**

1）getExecutionEnvironment：根据上下文自动返回执行环境
2）createLocalEnvironment：返回本地执行环境
3）createRemoteEnvironment：返回集群执行环境

**执行模式：**

DataStream API执行模式包括：流执行模式、批执行模式和自动模式

## 2.源算子

- 从集合中读取：`DataStreamSource<Integer> ds = env.fromCollection(data);`
- 从文件读取：`env.fromSource(fileSource,WatermarkStrategy.noWatermarks(),"file").print();`
- 从Socket读取：`DataStream<String> stream = env.socketTextStream("localhost", 7777);`
- 从Kafka读取：`env.fromSource(kafkaSource, WatermarkStrategy.noWatermarks(), "kafka-source");`
- 从数据生成器读取

**Flink支持的数据类型：**

1、Java基本类型和包装类
2、数组类
3、复合数据类型：Java元组类型（TUPLE）、Scala样例类和元组、行类型（ROW）、POJO
4、辅助类型：Option、Either、List、Map等
5、泛型类型：由Kryo序列化

## 3.转换算子

基本转换算子：map、filter、flatMap
聚合算子：keyBy、sum/min/max/minBy/maxBy、reduce
用户自定义函数UDF：函数类、匿名函数lambda、富函数类

## 4.物理分区算子

1、随机分区（shuffle）
2、轮询分区（round-robin）
3、重缩放分区（recale）
4、广播（broadcast）
5、全局分区（global）
6、自定义分区（custom）：通过自定义分区器partitionCustom()方法

## 5.分流

所谓“分流”，就是将一条数据流拆分成完全独立的两条、甚至多条流。也就是基于一个DataStream，定义一些筛选条件，将符合条件的数据拣选出来放到对应的流里。

调用上下文ctx的.output()方法，就可以输出任意类型的数据了。而侧输出流的标记和提取，都离不开一个“输出标签”（OutputTag），指定了侧输出流的id和类型

## 6.基本合流操作

在实际应用中，我们经常会遇到来源不同的多条流，需要将它们的数据进行联合处理。所以Flink中合流的操作会更加普遍，对应的API也更加丰富。

1、联合（union）：直接将多条流合在一起，受限于数据类型不能改变
2、连接（connect）：连接允许操作流的数据类型不同，可以看作形式上的“统一”，最后做一个同处理即可
CoMapFunction()、CoProcessFunction()、

## 7.输出算子

- 连接到外部系统：`stream.sinkTo(…);`
- 输出到文件：`FileSink支持行编码（Row-encoded）和批量编码（Bulk-encoded）格式`
- 输出到kafka：`KafkaSink<String> kafkaSink = KafkaSink.<String>builder()`
- 输出到mysql（jdbc）：`SinkFunction<WaterSensor> jdbcSink = JdbcSink.sink(...)`
- 自定义Sink输出：`stream.addSink(new MySinkFunction<String>());`

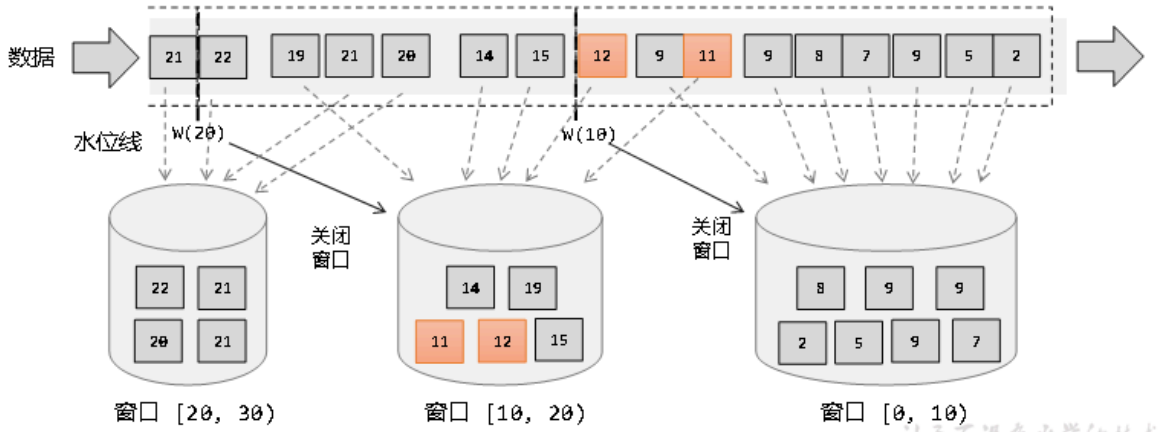
# 5、Flink中的时间和窗口

## 1.窗口

在批处理统计中，我们可以等待一批数据都到齐后，统一处理。但是在实时处理统计中，我们是来一条就得处理一条，那么我们怎么统计最近一段时间内的数据呢？引入“窗口”。

所谓的“窗口”，一般就是划定的一段时间范围，也就是“时间窗”；对在这范围内的数据进行处理，就是所谓的窗口计算。所以窗口和时间往往是分不开的。接下来我们就深入了解一下Flink中的时间语义和窗口的应用

正确理解：在Flink中，窗口其实并不是一个“框”，应该把窗口理解成一个“桶”。在Flink中，窗口可以把流切割成有限大小的多个“存储桶”（bucket）；每个数据都会分发到对应的桶中，当到达窗口结束时间时，就对每个桶中收集的数据进行计算处理。



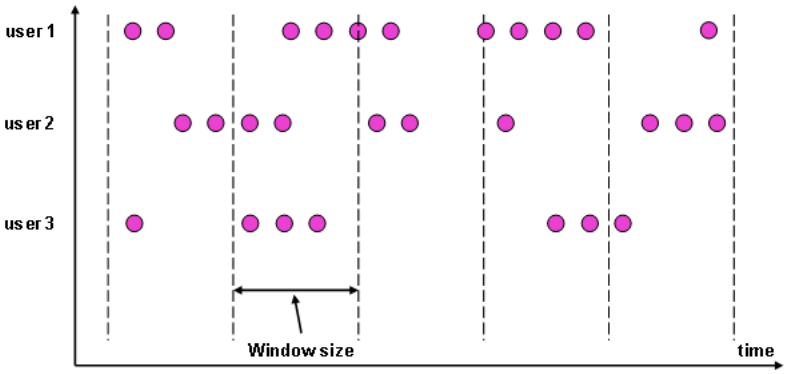

## 1）窗口的分类

- 按照驱动类型
  - 时间窗口：以时间点来定义窗口的开始start和结束end
  - 计数窗口：基于元素的个数来截取数据，“人齐发车”
- 按照窗口分类数据的规则
  - 滚动窗口（Tumbling Window）

    滚动窗口有固定的大小，是一种对数据进行“均匀切片”的划分方式。窗口之间没有重叠，也不会有间隔，是“首尾相接”的状态。这是最简单的窗口形式，每个数据都会被分配到一个窗口，而且只会属于一个窗口。

    滚动窗口可以基于时间定义，也可以基于数据个数定义；需要的参数只有一个，就是窗口的大小（window size）。

    比如我们可以定义一个长度为1小时的滚动时间窗口，那么每个小时就会进行一次统计；或者定义一个长度为10的滚动计数窗口，就会每10个数进行一次统计。

    滚动窗口应用非常广泛，它可以对每个时间段做聚合统计，很多BI分析指标都可以用它来实现。
  - 滑动窗口（Sliding Window）

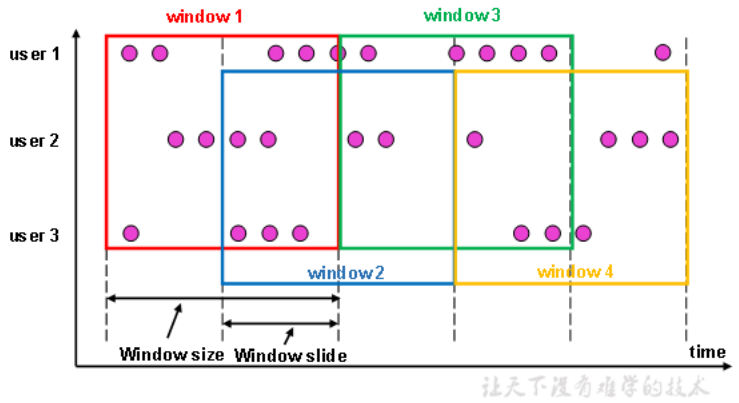
滑动窗口的大小也是固定的。但是窗口之间并不是首尾相接的，而是可以“错开”一定的位置。

定义滑动窗口的参数有两个：除去窗口大小（window size）之外，还有一个“滑动步长”（window slide），它其实就代表了窗口计算的频率。窗口在结束时间触发计算输出结果，那么滑动步长就代表了计算频率。

当滑动步长小于窗口大小时，滑动窗口就会出现重叠，这时数据也可能会被同时分配到多个窗口中。而具体的个数，就由窗口大小和滑动步长的比值（size/slide）来决定。

滚动窗口也可以看作是一种特殊的滑动窗口——窗口大小等于滑动步长（size = slide）。

滑动窗口适合计算结果更新频率非常高的场景



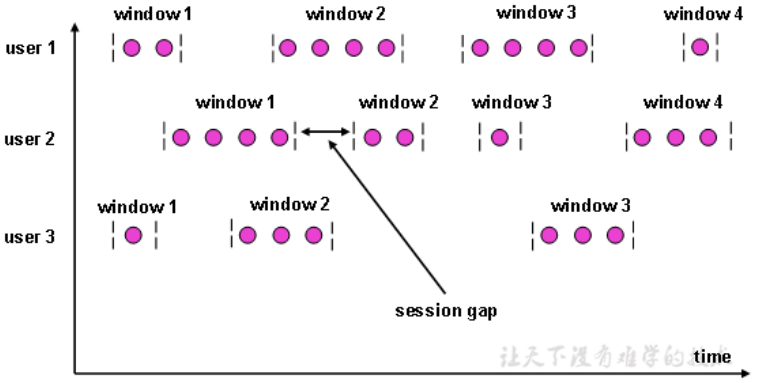

- 会话窗口（Session Window）

会话窗口，是基于“会话”（session）来来对数据进行分组的。会话窗口只能基于时间来定义。

会话窗口中，最重要的参数就是会话的超时时间，也就是两个会话窗口之间的最小距离。如果相邻两个数据到来的时间间隔（Gap）小于指定的大小（size），那说明还在保持会话，它们就属于同一个窗口；如果gap大于size，那么新来的数据就应该属于新的会话窗口，而前一个窗口就应该关闭了。

会话窗口的长度不固定，起始和结束时间也是不确定的，各个分区之间窗口没有任何关联。会话窗口之间一定是不会重叠的，而且会留有至少为size的间隔（session gap）。

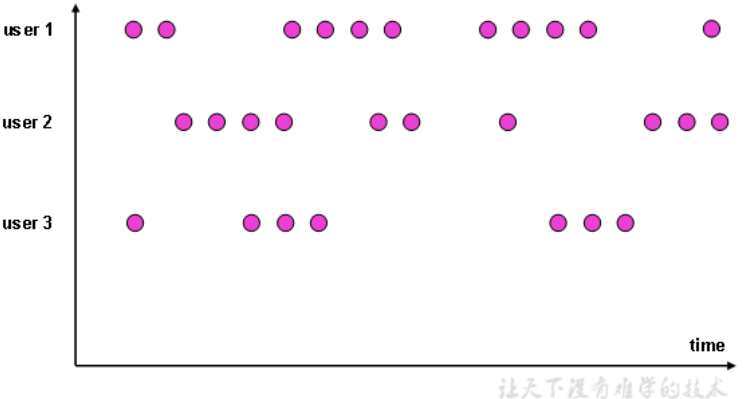
在一些类似保持会话的场景下，可以使用会话窗口来进行数据的处理统计。




- 全局窗口（Global Window）

“全局窗口”，这种窗口全局有效，会把相同key的所有数据都分配到同一个窗口中。这种窗口没有结束的时候，默认是不会做触发计算的。如果希望它能对数据进行计算处理，还需要自定义“触发器”（Trigger）。

全局窗口没有结束的时间点，所以一般在希望做更加灵活的窗口处理时自定义使用。Flink中的计数窗口（Count Window），底层就是用全局窗口实现的。




## 2）窗口API概览

1、按键分区（Keyed）和非按键分区（Non-Keyed）

2、代码中窗口API的调用：主要两个部分，窗口分配器和窗口函数

```
stream.keyBy(<key selector>)
       .window(<window assigner>)
       .aggregate(<window function>)
```

### 3）窗口分配器

构建窗口算子的第一步，它的作用就是定义数据应该被“分配”到哪个窗口。所以可以说，窗口分配器其实就是在指定窗口的类型。

窗口按照驱动类型可以分成时间窗口和计数窗口，而按照具体的分配规则，又有滚动窗口、滑动窗口、会话窗口、全局窗口四种。除去需要自定义的全局窗口外，其他常用的类型Flink中都给出了内置的分配器实现，我们可以方便地调用实现各种需求。

- 时间窗口
  - 滚动处理时间窗口

```
// 创建了一个长度为5秒的滚动窗口
stream.keyBy(...)
      .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
      .aggregate(...)
```

  - 滑动处理时间窗口

```
// 创建了一个长度为10秒、滑动步长为5秒的滑动窗口
stream.keyBy(...)
      .window(SlidingProcessingTimeWindows.of(Time.seconds(10),
Time.seconds(5)))
      .aggregate(...)
```

  - 处理时间会话窗口

```
// 创建了静态会话超时时间为10秒的会话窗口
stream.keyBy(...)
      .window(ProcessingTimeSessionWindows.withGap(Time.seconds(10)))
      .aggregate(...)
```

  - 滚动事件时间窗口

```
// 开5秒滚动事件时间窗口,间隔5秒滚动一次
stream.keyBy(...)
      .window(TumblingEventTimeWindows.of(Time.seconds(5)))
      .aggregate(...)
```

  - 滑动事件时间窗口

```
stream.keyBy(...)
      .window(SlidingEventTimeWindows.of(Time.seconds(10),
Time.seconds(5)))
      .aggregate(...)
```

  - 事件时间会话窗口

```
stream.keyBy(...)
      .window(EventTimeSessionWindows.withGap(Time.seconds(10)))
      .aggregate(...)
```

- 计数窗口
  - 滚动计数窗口

```
// 定义了一个长度为10的滚动计数窗口
stream.keyBy(...)
        .countWindow(10)
```

  - 滑动计数窗口

```
// 定义了一个长度为10、滑动步长为3的滑动计数窗口。每个窗口统计10个数据，每隔3个数据就统计输出一次结果。
stream.keyBy(...)
        .countWindow(10, 3)
```

  - 全局窗口

```
// 使用全局窗口，必须自行定义触发器才能实现窗口计算，否则起不到任何作用
stream.keyBy(...)
        .window(GlobalWindows.create());
```

### 4）窗口函数

窗口函数定义了要对窗口中收集的数据做的计算操作，根据处理的方式可以分为两类：增量聚合函数和全窗口函数。

- 增量聚合函数（ReduceFunction / AggregateFunction）

> 窗口将数据收集起来，最基本的处理操作当然就是进行聚合。我们可以每来一个数据就在之前结果上聚合一次，这就是“增量聚合”。

| 特性 | ReduceFunction | AggregateFunction |
| --- | --- | --- |
| 输入 / 输出类型 | 必须相同 | 可以不同 |
| 中间状态 | 和输出相同 | 可以自定义累加器 |
| 复杂度 | 简单 | 更灵活、功能强大 |
| 适用场景 | sum / min / max / count | 平均值、TopN、复杂指标 |

- 全窗口函数（full window functions）

> 有些场景下，我们要做的计算必须基于全部的数据才有效，这时做增量聚合就没什么意义了；另外，输出的结果有可能要包含上下文中的一些信息（比如窗口的起始时间），这是增量聚合函数做不到的。
>
> 所以，我们还需要有更丰富的窗口计算方式。窗口操作中的另一大类就是全窗口函数。与增量聚合函数不同，全窗口函数需要先收集窗口中的数据，并在内部缓存起来，等到窗口要输出结果的时候再取出数据进行计算。

| 特性 | WindowFunction | ProcessWindowFunction |
| --- | --- | --- |
| 是否缓存整个窗口数据 | 是 | 是 |

| 特性 | WindowFunction | ProcessWindowFunction |
| --- | --- | --- |
| 是否能访问窗口上下文 | 否 | ✅ 可以（时间、水位线、状态） |
| 性能 | 较低 | 较低（除非配合 reduce/aggregate） |
| 典型场景 | 简单窗口聚合 | 需要窗口元数据、迟到数据处理、配合增量聚合 |

### 5）其他API

> 对于一个窗口算子而言，窗口分配器和窗口函数是必不可少的。除此之外，Flink还提供了其他一些可选的API，让我们可以更加灵活地控制窗口行为

1、触发器（Trigger）：用来控制窗口什么时候触发
2、移除器（Evictor）：定义移除某些数据的逻辑

## 2、时间语义

从Flink1.12版本开始，Flink已经将事件时间作为默认的时间语义

## 3、水位线

> **水位线 = Flink 的时钟**，用来驱动窗口触发。

Flink 里的 **事件时间 EventTime** 表示的是数据本身的时间戳，而不是系统处理时间。

**Watermark（水位线）** 用来标记“事件时间已经推进到某个点”，意味着 **小于等于这个水位线的事件，Flink 认为都到齐了**，可以触发对应窗口计算。水位线传递

### 1）水位线传递

水位线在 Flink 程序中和数据 **一起向下游流动。**

📌 关键点：

- **水位线和数据流一样，也是流里的一种特殊元素**（和数据记录 interleave 交替出现）。
- 当上游算子产生了水位线，它会被发送到下游。
- 下游算子会根据 **所有输入分区的水位线** 取 **最小值**，作为自己的当前水位线。

## 4、基于时间的合流-双流联结Join

> **什么是双流join?**
>
> - 有两个输入流：**流 A** 和 **流 B**。
> - 想要按照某个 key（如用户 id、订单号）进行匹配。
> - 但是是 **基于时间**的，也就是只在一定时间范围内的事件才会匹配成功。
>
> 👉 例如：订单流和支付流，**要求支付必须在下单后的 30 分钟内**才算成功。

Flink提供的Join API有两类：

- **窗口联结Window Join**

  基于窗口的 Join。

  - 两个流先按 key 分组 → 落入相同窗口的数据 → 做笛卡尔匹配。

- 要求必须有 **窗口**，比如 `TumblingWindow`、`SlidingWindow`。

👉 适合 “只关心窗口范围内” 的匹配。

- **间隔联结Internal Join**

  基于事件时间区间的 Join。

  - 必须在 **KeyedStream** 上调用。
  - 可以定义时间上下界（例如 -5 秒到 +10 秒）。
  - 更加灵活，不依赖窗口。

  👉 适合 “A 事件发生前后的一段时间里，是否有 B 事件发生”。

| 特点 | Window Join | Interval Join |
| --- | --- | --- |
| 是否依赖窗口 | ✅ 必须有窗口 | ❌ 不需要窗口 |
| 匹配范围 | 窗口范围内 | 指定时间区间 |
| 灵活性 | 中等 | 更灵活 |
| 使用场景 | 双流数据按窗口统计 | 双流事件匹配（对账、风控） |

# 6、处理函数ProcessFunction

> - Flink 的 **DataStream API** 已经提供了很多算子：`map`、`flatMap`、`reduce`、`window`...
> - 但是这些算子往往只能完成 **常见模式**，不够灵活。
> - **Process Function** 是一种更底层的 API，它能让你：
>   1. 访问事件（event）本身
>   2. 访问时间戳（event time / processing time）
>   3. 使用 **定时器（timer）**
>   4. 使用 **状态（state）**
>
> 👉 所以 `ProcessFunction` 是 Flink 的“万能接口”，用来做复杂业务逻辑。

Flink 提供了几类 `ProcessFunction`，常见的有：

| 函数 | 作用 | 使用场景 |
| --- | --- | --- |
| **ProcessFunction** | 基础处理函数 | 一般用于 DataStream 基础处理，按事件处理、注册定时器 |
| **KeyedProcessFunction** | 针对 keyBy 之后的流 | 支持 **键控状态** 和 **定时器**，最常用 |
| **ProcessWindowFunction** | 针对窗口 | 可以访问窗口信息（时间范围、窗口上下文） |
| **CoProcessFunction** | 双流处理函数 | 两个流合并时，可以分别处理来自两边的元素 |
| **BroadcastProcessFunction** | 广播流处理 | 配置流（规则流）+ 业务流，常见于风控规则 |

| 函数 | 作用 | 使用场景 |
| --- | --- | --- |
| **KeyedBroadcastProcessFunction** | 广播流 + keyBy 流 | 广播动态规则，keyBy 数据流 |

# 7、状态管理

Flink的状态有两种：托管状态（Managed State）和原始状态（Raw State）

托管状态分为两类：算子状态和按键分区状态。

## 1. 算子状态（Operator State）

- **范围**：绑定到某个算子实例（SubTask），**整个算子实例共享一份状态。**
- **特点**：不能按 key 拆分，也就是说某个 subtask 内所有数据都更新这份状态。
- **典型场景**：
  - Source 维护 Kafka offset（记录消费到哪里了）。
  - 批量 sink：先缓存在状态里，再定时写出。

常见数据结构：

- `ListState<T>`（列表状态）：一组值
- `UnionListState<T>`（联合列表状态）：checkpoint 恢复时合并所有子任务的状态
- `BroadcastState<K,V>`（广播状态）：广播状态（常见于流与流的广播 join）

## 2. 键控状态（Keyed State）

- **范围**：必须在 `keyBy()` 之后使用。
- **特点**：状态是 **按 key 隔离存储**，不同 key 的数据有自己的状态副本。
  - 在 WordCount 里，每个单词都有自己的 count。
- **状态是托管在 KeyedStateBackend 中**（HashMap / RocksDB），Flink 帮你存储和恢复。

常见数据结构：

- `ValueState<T>`（值状态）：保存单个值（最常用）
- `ListState<T>`（列表状态）：保存一个列表
- `MapState<K,V>`（Map状态）：保存一个映射
- `ReducingState<T>`（归约状态）：聚合状态（连续 reduce、归约状态）
- `AggregatingState<IN, OUT>`（聚合状态）：聚合状态（带不同输入输出类型）

| 类型 | 作用范围 | 是否按 key 隔离 | 典型用途 |
| --- | --- | --- | --- |
| 算子状态 | 算子 subtask 级别 | ❌ 否 | Source offset、广播流配置、批量缓存 |
| 键控状态 | keyBy 后的 key 范围 | ✅ 是 | WordCount、用户维度指标、会话窗口统计 |

## 3. 状态的存储方式（State Backend）

Flink 提供了不同的 **状态后端（State Backend）**，决定状态存在什么地方：

1. **HashMapStateBackend** 哈希表状态后端（默认，小状态）
   - 状态存在 **TaskManager （内存）里**
   - Checkpoint 时写到 JobManager 的存储
   - 适合小规模状态
2. **EmbeddedRocksDBStateBackend** 内嵌RocksDB状态后端（大状态）
   - 状态存在 **RocksDB（本地磁盘）**
   - Checkpoint 时写到远端（HDFS / S3）
   - 适合超大状态、几十 GB / TB

## 4. 状态一致性保障

Flink 通过 **Checkpoint** 和 **Savepoint** 保证状态一致性。

- **Checkpoint**：周期性快照，用于容错恢复（Exactly-once）。
- **Savepoint**：手动触发的快照，主要用于程序升级、迁移。

# 8、容错机制

Flink 容错机制 =
**checkpoint（自动） + savepoint（手动） + 状态后端（存状态） + barrier（划分一致性停留点、快照屏障） + 两阶段提交 sink（端到端一致性）**

## 1.Flink 容错的目标

- **不中断大规模流处理作业**：即使算子、TaskManager、JobManager 挂掉，作业也能恢复。
- **保证数据一致性**：即使失败重启，结果和“没出错”一样。
- 提供 **至少一次（At-Least-Once）** 和 **精确一次（Exactly-Once）** 两种语义（默认追求 Exactly-Once）。

## 2.容错机制的核心：检查点（Checkpoint）

- Flink 定期为 **算子的状态** 生成快照（snapshot）。
- 这个快照被保存到 **持久化存储**（例如 HDFS、S3、RocksDB backend）。
- 作业失败时，可以 **从最近一次成功的 checkpoint 恢复**，继续消费数据，避免丢失或重复。

👉 这个机制叫做 **分布式快照**，实现基于 **Chandy-Lamport 一致性快照算法**。

## 3.Checkpoint原理（轻量级分布式快照）

1、**触发 checkpoint**：JobManager 定期触发 checkpoint，给所有 Source 插入一个“barrier（水位线样的特殊标记）”。
2、**barrier 传递**：barrier 会沿着数据流向下游算子传递，划分数据流为不同的 epoch。
3、**状态快照**：当某个算子收到所有输入流的 barrier，就对自己的状态做一次快照，并将 barrier 继续往下游传。
4、**快照完成**：当所有算子完成快照，整个 checkpoint 成功。

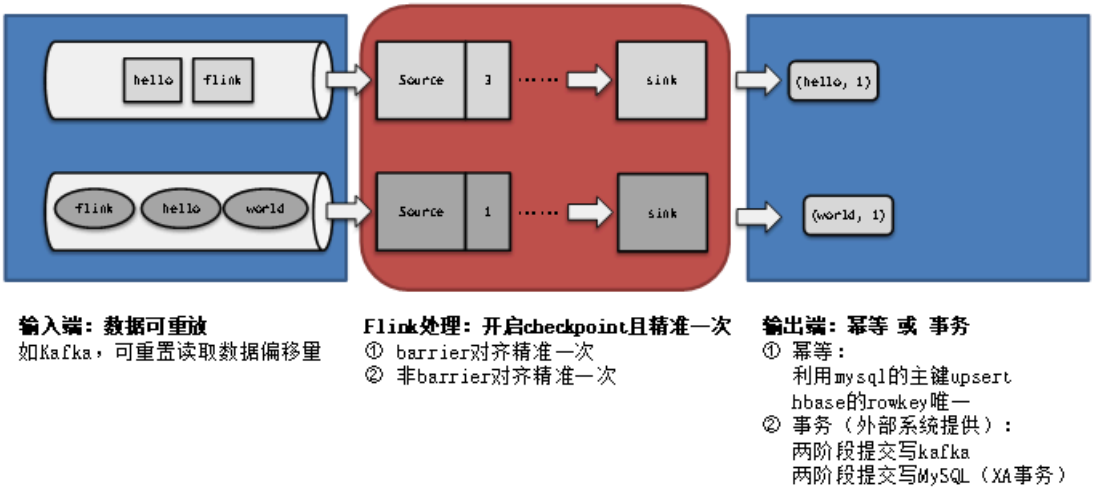
## 4.保存点（Savepoint）

Savepoint = **人工触发**的 checkpoint。

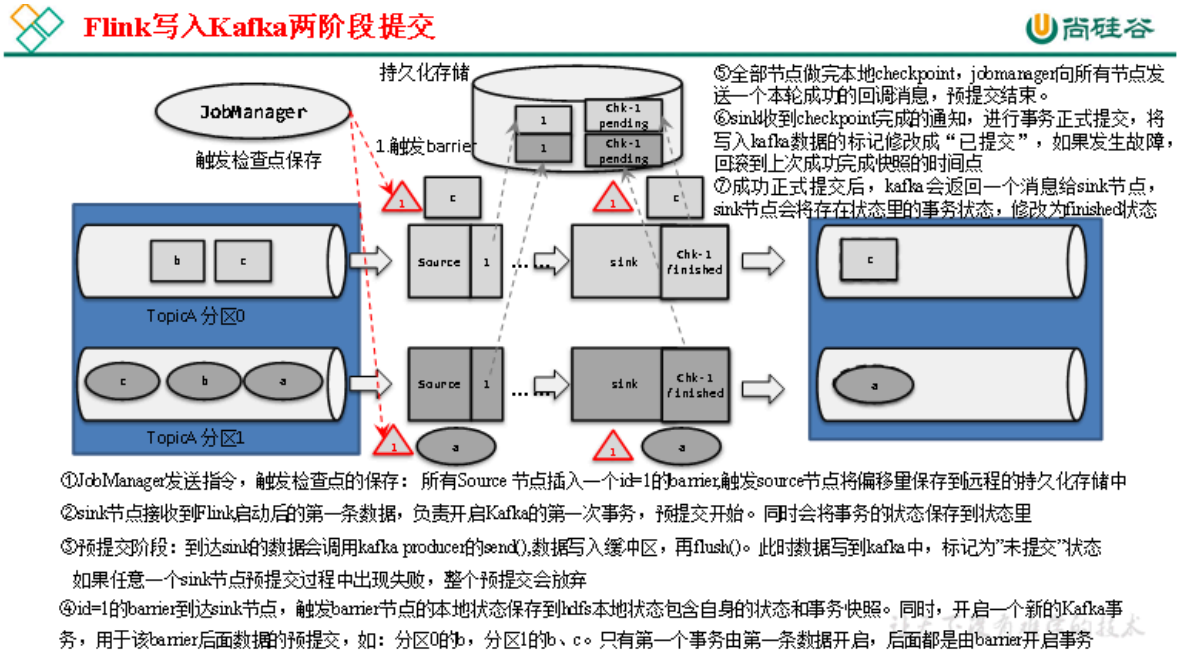
通常用于 **升级作业、迁移集群、停止并恢复。**

格式与 checkpoint 类似，但更稳定、可长期保存。

## 5.端到端精确一次Exactly-Once




## 6.Flink写入Kafka




既然是端到端的exactly-once，我们依然可以从三个组件的角度来进行分析：

**（1）Flink内部**：Flink内部可以通过检查点机制保证状态和处理结果的exactly-once语义。

**（2）输入端**：输入数据源端的Kafka可以对数据进行持久化保存，并可以重置偏移量（offset）。所以我们可以在Source任务（FlinkKafkaConsumer）中将当前读取的偏移量保存为算子状态，写入到检查点中；当发生故障时，从检查点中读取恢复状态，并由连接器FlinkKafkaConsumer向Kafka重新提交偏移量，就可以重新消费数据、保证结果的一致性了。

**（3）输出端**：输出端保证exactly-once的最佳实现，当然就是两阶段提交（2PC）。作为与Flink天生一对的Kafka，自然需要用最强有力的一致性保证来证明自己。

也就是说，我们写入Kafka的过程实际上是一个两段式的提交：处理完毕得到结果，写入Kafka时是基于事务的“预提交”；等到检查点保存完毕，才会提交事务进行“正式提交”。如果中间出现故障，事务进行回滚，预提交就会被放弃；恢复状态之后，也只能恢复所有已经确认提交的操作。

# 9、Flink SQL

## 1. Flink SQL 是什么

- Flink SQL 是 Flink 的 **流批一体 SQL 引擎**。
- 支持对 **无限流（Streaming）** 和 **有限数据（Batch）** 使用同一套 SQL 来计算。
- 内核是 **动态表（Dynamic Table）** 概念，把流抽象成一张随时间变化的表。
- SQL 最终会转换为 Flink **DataStream 作业**执行。

## 2. Flink SQL 的核心概念

1. **表（Table）**
   - 可以来自外部系统（Kafka、MySQL、Hive、Elasticsearch、文件系统等）。
   - 使用 **DDL** 定义：`CREATE TABLE ... WITH (connector=...)`。
2. **动态表（Dynamic Table）**
   - 流 → 表：无限流抽象为“随时间不断更新”的表。
   - 表 → 流：查询结果会转换成 **Changelog Stream**（Insert/Update/Delete）。
3. **时间语义**
   - **Processing Time**：基于算子本地系统时间。
   - **Event Time**：基于事件自身时间戳，需要 **Watermark（水位线）**。
4. **执行模式**
   - **批模式**（Bounded）：一次性处理全量数据。
   - **流模式**（Unbounded）：实时处理无限数据流。

## 3. Flink SQL 的执行流程

1. **解析 SQL** → 生成 AST（抽象语法树）
2. **优化**（Calcite 负责）→ 逻辑计划、物理计划优化
3. **转换** → 转换成 Flink Table API 调用
4. **生成执行计划** → 最终下发到 Flink Runtime 执行（TaskManager 并行执行）

## 4. 常见语法示例

① 定义表（Kafka 源）

```
CREATE TABLE user_events (
    user_id STRING,
    event_time TIMESTAMP(3),
    page_id STRING,
    WATERMARK FOR event_time AS event_time - INTERVAL '5' SECOND
) WITH (
    'connector' = 'kafka',
    'topic' = 'user_topic',
    'properties.bootstrap.servers' = 'hadoop102:9092',
    'format' = 'json'
);
```

② 简单查询

```
SELECT user_id, COUNT(*) AS cnt
FROM user_events
GROUP BY user_id;
```

③ 窗口聚合

```
SELECT
  TUMBLE_START(event_time, INTERVAL '10' MINUTE) AS win_start,
  user_id,
  COUNT(*) AS cnt
FROM user_events
GROUP BY TUMBLE(event_time, INTERVAL '10' MINUTE), user_id;
```

④ 双流 Join（基于时间窗口）

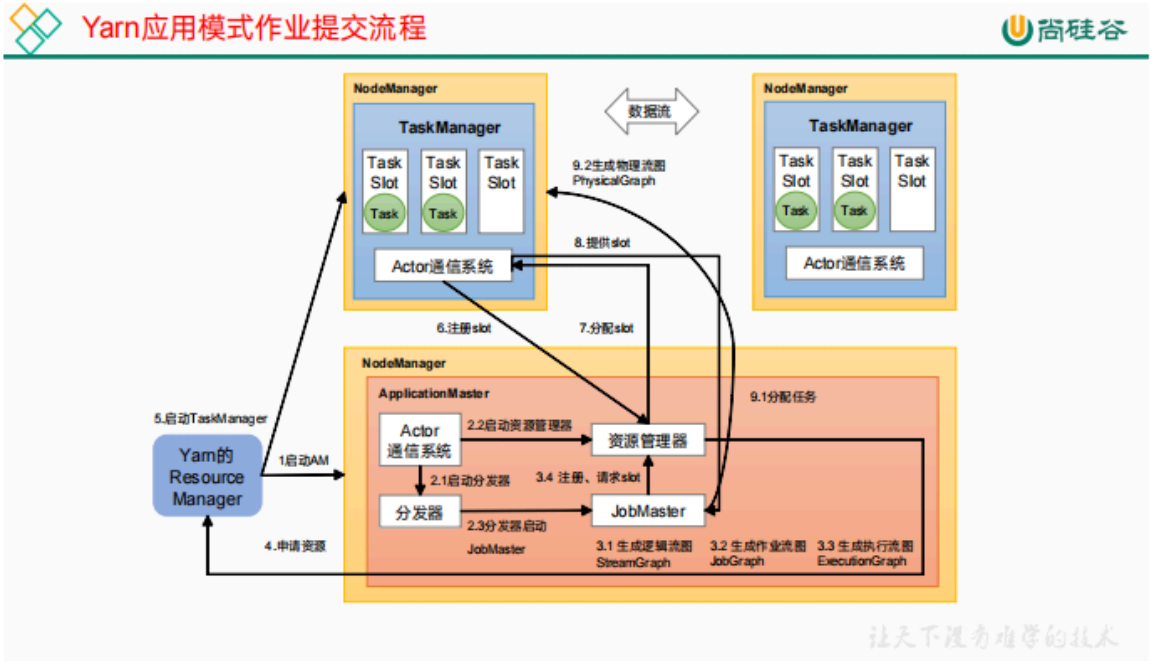
```
SELECT o.order_id, o.user_id, p.pay_amount
FROM orders o
JOIN payments p
ON o.order_id = p.order_id
AND o.order_time BETWEEN p.pay_time - INTERVAL '10' MINUTE
                     AND p.pay_time + INTERVAL '10' MINUTE;
```

## 🛠️ 5. 应用场景

- **实时数仓**：Kafka → Flink SQL → Hudi/Iceberg/Hive
- **实时报表**：Kafka → Flink SQL → ClickHouse/Elasticsearch
- **实时风控 / 监控**：基于窗口聚合、Join、Pattern 匹配
- **实时 ETL**：数据清洗、维表 Join

# 10、Flink相关面试题

## 1.Flink基础架构组成




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

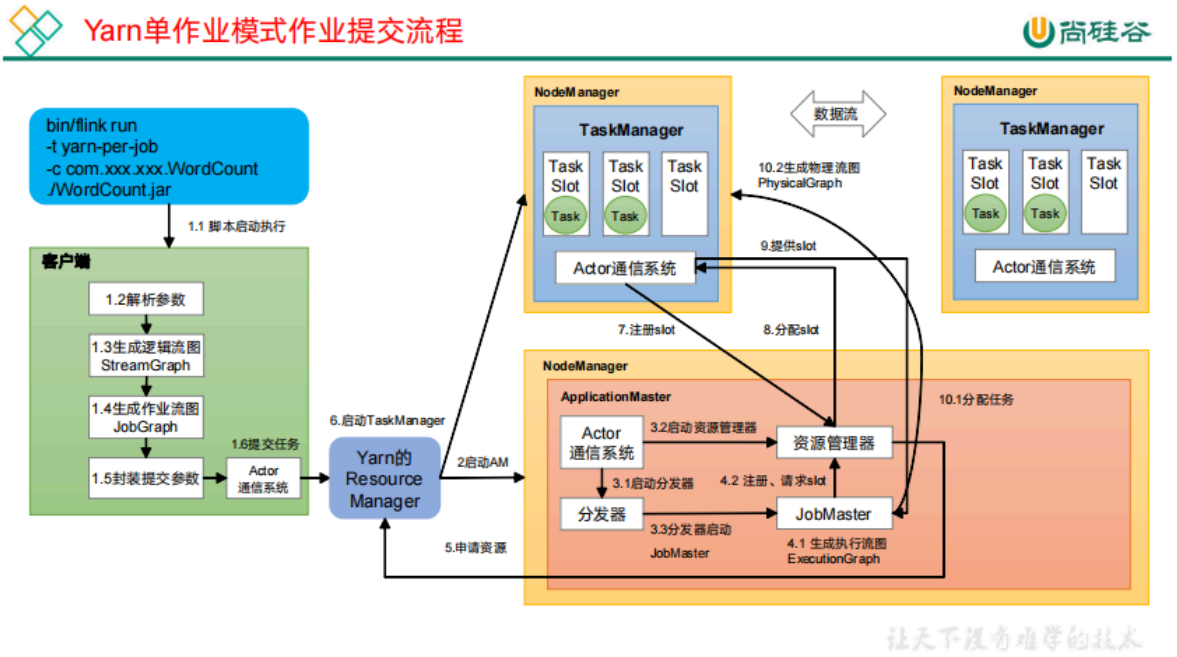
- JobManager 是集群的老大，负责接收 Flink Job，协调检查点，Failover 故障恢复等，同时管理 TaskManager。 包含：Dispatcher、ResourceManager、JobMaster。
- TaskManager 是执行计算的节点，每个 TaskManager 负责管理其所在节点上的资源信息，如内存、磁盘、网络。内部划分 slot 隔离内存，不隔离 cpu。同一个 slot 共享组的不同算子的 subtask 可以共享 slot。
- Client 是 Flink 程序提交的客户端，将 Flink Job 提交给 JobManager。

## 2.Flink和Spark Streaming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.Flink提交作业流程和核心概念?

**1）Flink提交流程（Yarn-Per-Job）**




**2）算子链路**

Flink 自动做的优化，要求 One-to-one，并行度相同。代码 disableOperatorChaining()禁用算子链。

**3）Graph生成与传递**

| | 在哪里生成 | 传递给谁 | 做了什么事 |
|---|---|---|---|
| **逻辑流图 StreamGraph** | Client | Client | 最初的 DAG 图 |
| **作业流图 JobGraph** | Client | JobManager | 算子链路优化 |
| **执行流图 ExecutionGraph** | JobManager | JobManager | 并行度的细化 |
| **物理流图** | | | |

**4）Task和Subtask的区别**

Subtask：算子的一个并行实例。

Task：Subtask 运行起来之后，就叫 Task

**5）并行度和Slot的关系**

Slot 是静态的概念，是指 TaskMangaer 具有的并发执行能力。
并行度是动态的概念，指程序运行时实际使用的并发能力。
设置合适的并行度能提高运算效率，太多太少都不合适。

**6）Slot共享组了解吗，如何独享Slot插槽**

默认共享组时 default，同一共享组的 task 可以共享 Slot。

通过 slotSharingGroup()设置共享组。

## 4.Flink部署模式?

1）**Local**：本地模式，Flink 作业在单个 JVM 进程中运行，适用于测试阶段

2）**Standalone**：Flink 作业在一个专门的 Flink 集群上运行，独立模式不依赖于其他集群管理器（Yarn 或者 Kubernetes）|

3）**Yarn**：

Per-job：独享资源，代码解析在 Client

Application：独享资源，代码解析在 JobMaster

Session：共享资源，一套集群多个 job

4）**K8s**：支持云原生，未来的趋势

5）**Mesos**：国外使用，仅作了解

## 5.Flink任务并行度优先级设置？资源一般如何配置?

设置并行度有多种方式，优先级：算子 > 全局 Env > 提交命令行 > 配置文件

**1）并行度根据任务设置**

（1）常规任务：Source，Transform，Sink 算子都与 Kafka 分区保持一致
（2）计算偏大任务：Source，Sink 算子与 Kafka 分区保持一致，Transform 算子可设置成 2 的 n 次方，64，128...

**2）资源设置**：通用经验 1CU = 1CPU + 4G 内存

Taskmanager 的 Slot 数：1 拖 1（独享资源）、1 拖 N（节省资源，减少网络传输）
TaskManager 的内存数：4~8G
TaskManager 的 CPU：Flink 默认一个 Slot 分配一个 CPU
JobManager 的内存：2~4G
JobManager 的 CPU：默认是 1

**3）资源是否足够**

资源设置，然后压测，看每个并行度处理上限，是否会出现反压

例如：每个并行度处理 5000/s，开始出现反压，比如我们设置三个并行度，我们程序处理上限 15000/s

## 6.Flink三种时间语义?

**事件时间 Event Time**：是事件创建的时间。数据本身携带的时间。
**进入时间 Ingestion Time**：是数据进入 Flink 的时间。
**处理时间 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。

## 7.你对watermark水位线的认识?

水位线是 Flink 流处理中保证结果正确性的核心机制，它往往会跟窗口一起配合，完成对乱序数据的正确处理。

 水位线是插入到**数据流中的一个标记**，可以认为是一个**特殊的数据**

 水位线主要的内容是一个**时间戳**，用来**表示当前事件时间的进展**

 水位线是**基于数据的时间戳生成的**

 水位线的时间戳必须**单调递增**，以确保任务的事件时间时钟一直向前推进

 水位线可以通过**设置延迟**，来保证正确**处理乱序**数据

 一个水位线 Watermark(t)，表示在当前流中事件时间已经达到了时间戳 t，**这代表 t之前的所有数据都到齐了，之后流中不会出现时间戳 t' ≤ t 的数据**

## 8.watermark多并行度的传递、生成原理?

**1）分类：**

间歇性：来一条数据，更新一次 Watermark。

周期性：固定周期更新 Watermark。

官方提供的 API 是基于周期的，默认 200ms，因为间歇性会给系统带来压力。

**2）生成原理**

Watermark = 当前最大事件时间 - 乱序时间 - 1ms

**3）传递**

Watermark 是一条携带时间戳的特殊数据，从代码指定生成的位置，插入到流里面。

一对多：**广播。**
多对一：**取最小。**
多对多：拆分来看，其实就是上面两种的结合。

## 9.Flink怎么处理乱序和迟到数据?

在 Apache Flink 中，迟到时间（lateness）和乱序时间（out-of-orderness）是两个与处理时间和事件时间相关的概念。它们在流处理过程中，尤其是在处理不按事件时间排序的数据时非常重要。

**（1）迟到时间 lateness：**：迟到时间可以影响窗口，在窗口计算完成后，仍然可以接收迟到的数据迟到时间是指事件到达流处理系统的延迟时间，即事件的实际接收时间与其事件时间的差值。在某些场景下，由于网络延迟、系统故障等原因，事件可能会延迟到达。为了处理这些迟到的事件，Flink 提供了一种机制，允许在窗口计算完成后仍然接受迟到的数据。设置迟到时间后，Flink 会在窗口关闭之后再等待一段时间，以便接收并处理这些迟到的事件。

设置迟到时间的方法如下：

在定义窗口时，使用 `allowedLateness` 方法设置迟到时间。例如，设置迟到时间为 10 分钟：

```
DataStream<T> input = ...;
input
    .keyBy(<key selector>)
    .window(<window assigner>)
    .allowedLateness(Time.minutes(10))
    .<window function>;
```

**（2）乱序时间 out-of-orderness**

乱序时间是通过影响水印来影响数据的摄入，它表示的是数据的混乱程度。

乱序时间是指事件在流中不按照事件时间的顺序到达。在某些场景下，由于网络延迟或数据源的特性，事件可能会乱序到达。

Flink 提供了处理乱序事件的方法，即水位线（watermark）。

水位线是一种表示事件时间进展的机制，它告诉系统当前处理到哪个事件时间。当水位线到达某个值时，说明所有时间戳小于该值的事件都已经处理完成。为了处理乱序事件，可以为水位线设置一个固定的延迟。

设置乱序时间的方法如下：

```
//在定义数据源时，使用`assignTimestampsAndWatermarks`方法设置水位线策略。例如，设置水位线延迟为 5 秒：
DataStream<T> input = env.addSource(<source>);
input
    .assignTimestampsAndWatermarks(
    WatermarkStrategy
    .<T>forBoundedOutOfOrderness(Duration.ofSeconds(5))
    .withTimestampAssigner(<timestamp assigner>))
    .<other operations>;
```

## 10.说说Flink的窗口?

> 分类、生命周期、触发、划分

**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4个相关重要组件：**

assigner（分配器）：如何**将元素分配给窗口。**
function（计算函数）：为窗口定义的计算。其实是一个计算函数，**完成窗口内容的计算。**
triger（触发器）：在什么条件下**触发窗口的计算。**

> 可以使用自定义触发器，解决事件时间，没有数据到达，窗口不触发计算问题，还可以使用持续性触发器，实现一个窗口多次触发输出结果，详细看连接
>
> **问题展示：**
>
> https://www.bilibili.com/video/BV1Gv4y1H7F8/?spm_id_from=333.999.0.0&vd_source=891aa1a363111d4914eb12ace2e039af
>
> **问题解决：**
>
> https://www.bilibili.com/video/BV1mM411N7uP/?spm_id_from=333.999.0.0&vd_source=891aa1a363111d4914eb12ace2e039af

evictor（退出器）：定义从窗口中**移除数据。**

**3）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)

**4）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

**5）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）。

**6）窗口为什么左闭右开：**属于窗口的最大时间戳 = end - 1ms

**7）窗口什么时候触发：**如基于事件时间的窗口 watermark >= end - 1ms

## 11.Flink的keyby怎么实现的分区？分区、分组的区别？

在 Flink 里，**keyBy 不是分组（group），而是分区（partition）操作。**

**分区：**分区（Partitioning）是将数据流划分为多个子集，这些子集可以在不同的任务实例上进行处理，以实现数据的并行处理。

数据具体去往哪个分区，是通过指定的 key 值先进行一次 hash 再进行一次murmurHash，通过上述计算得到的值再与并行度进行相应的计算得到。

**分组：**分组（Grouping）是将具有相同键值的数据元素归类到一起，以便进行后续操作（如聚合、窗口计算等）。key 值相同的数据将进入同一个分组中。

注意：数据如果具有相同的 key 将一定去往同一个分组和分区，但是同一分区中的数据不一定属于同一组。

> 分区是物理层面（shuffle、rebalance、rescalse、broadcast...），分组是逻辑层面，在 SQL 或 Table API 里你写 `group by userId`，就是逻辑分组，底层还是通过 **keyBy 分区** 实现。

## 12.Flink的Internal Join实现原理？Join不上的怎么办？

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。

## 13.介绍一下Flink的状态编程、状态机制？

（1）算子状态：作用范围是算子，算子的多个并行实例各自维护一个状态

（2）键控状态：每个分组维护一个状态

（3）状态后端：两件事=》本地状态存哪里、checkpoint 存哪里

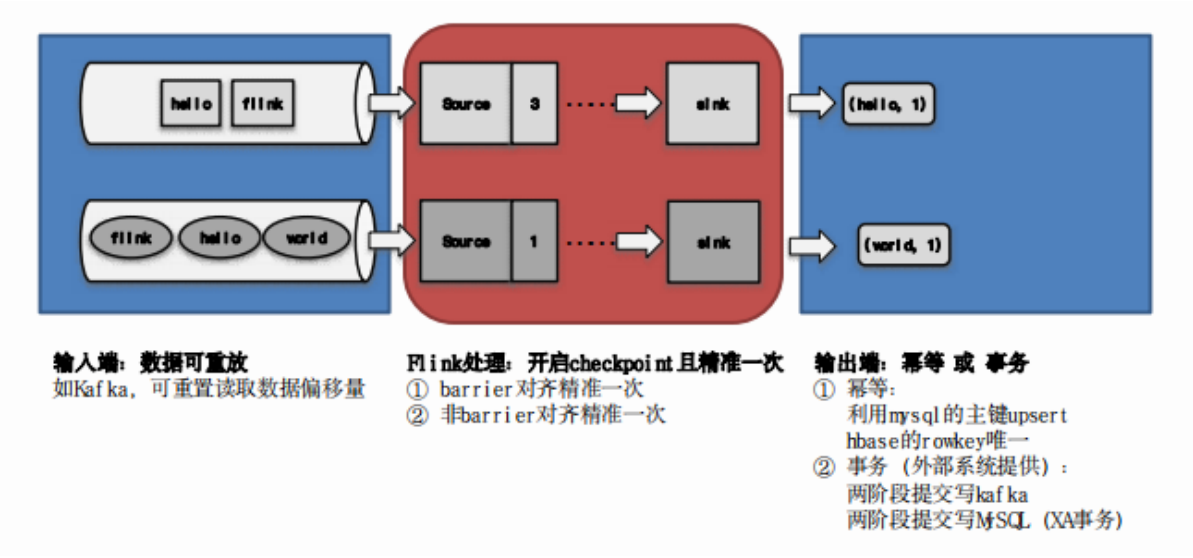
**本地状态存哪**（TaskManager 内存 / 磁盘）

- **HashMapStateBackend**（原 MemoryStateBackend） → 存内存，快，适合小状态。
- **EmbeddedRocksDBStateBackend** → 存 RocksDB（落盘），适合大状态。

**Checkpoint 存哪**（远端存储，容错用）

- HDFS / S3 / OSS / DFS 等。
- JobManager 只存元数据，真正的状态快照存在远端。

# 14.Flink如何实现端到端一致性?




## 1️⃣ Flink 一致性语义等级

Flink 在保证数据一致性上，分为三种语义：

- **At-most-once**：最多一次（可能丢数据）；
- **At-least-once**：至少一次（可能重复）；
- **Exactly-once**：精确一次（不丢、不重、不乱序）。

👉 面试常问的就是：**Flink 如何实现端到端 Exactly-once?**

## 2️⃣ 三个环节要保证一致

要做到端到端一致性，必须**输入、处理、输出**三步都保证一致：

1. **输入端（Source）**
   - Kafka Source 支持 checkpoint 对齐，即 **只在 checkpoint 成功后才提交 offset。**
   - 这样即使 Flink 崩溃，从 checkpoint 恢复时，也能重新从上次一致的 offset 消费。
2. **处理中（Flink Job 内）**
   - Flink 通过 **Checkpoint + Barrier 对齐**，把算子状态快照保存到状态后端（HDFS/S3 等）。
   - 失败时，从最近一次 checkpoint 恢复，保证处理逻辑 exactly-once。
3. **输出端（Sink）**
   - 普通 Sink（如直接写 MySQL insert）只能保证 **at-least-once**，因为失败重试会产生重复。
   - 为了做到 **exactly-once**，Flink 提供了 **两阶段提交（Two-Phase Commit）Sink**，典型如：
     - Kafka Sink
     - JDBC Sink (upsert)
     - HBase Sink

## 3️⃣ 两阶段提交协议（Two-Phase Commit）

Flink 的 **两阶段提交 Sink** 流程：

**① Pre-commit 阶段**

- 每个 checkpoint 触发时，Sink 将本次批次数据写入临时事务中，但不提交。

② **Commit 阶段**

- 当 Flink 的 checkpoint 完成（JobManager 通知所有算子 checkpoint 成功），Sink 才真正提交事务。
- 如果 checkpoint 失败，事务回滚，保证不会有脏数据写入。

👉 这样就实现了 **和 checkpoint 强一致性挂钩**，端到端达到 **Exactly-once**。

### 4️⃣ 举个例子

比如 **Kafka → Flink → MySQL** 流程：

1. Source (Kafka)：Flink 只在 checkpoint 成功后，提交 offset → 避免数据丢失。
2. Flink 内部：算子状态在 checkpoint 中保存，失败可恢复。
3. Sink (MySQL Upsert Sink)：用两阶段提交（先写临时表/事务，checkpoint 成功再提交），保证数据不会重复。

## 15.Flink分布式快照的原理是什么？

1. **JobManager 触发快照**
   - JobManager 向 Source 发送 **Checkpoint Barrier**。
2. **Barrier 沿着数据流传播**
   - Barrier 随着数据流向下游算子流动，每个算子在收到所有输入通道的 Barrier 后才进行快照。
3. **算子状态保存**
   - 当算子收到 Barrier 时：
     1. 暂停处理 Barrier 之前的事件
     2. 异步将当前状态写入 **状态后端（State Backend）**
   - 后续事件继续处理，不阻塞流。
4. **Barrier 合并与全局完成**
   - 下游算子需要从 **所有输入通道** 收到 Barrier 才认为该算子快照完成
   - 当所有算子完成快照后，JobManager 记录 **全局快照完成**。

> 核心点：Barrier 是 Flink 保证 **一致性的标记**，它划分了数据流中的事件前后界限。

## 16.Checkpoint 的参数怎么设置的？

（1）间隔：兼顾性能和延迟，一般任务设置分钟级（1~5min），要求延迟低的设置秒级

（2）Task 重启策略（Failover）：

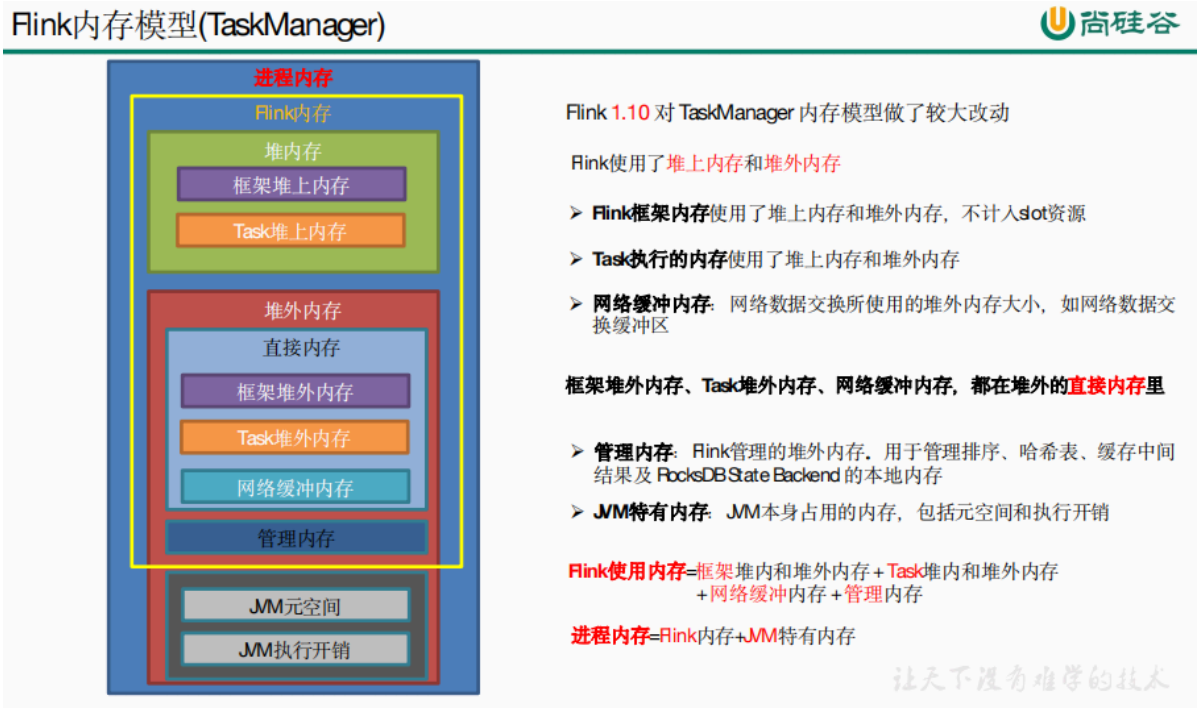
固定延迟重启策略：重试几次、每次间隔多久。

失败率重启策略：重试次数、重试区间、重试间隔。

无重启策略：一般在开发测试时使用。

Fallback 重启策略：默认固定延迟重启策略。

## 17.Flink内存模型?




## 18.Flink常见的维表Join方案?

> 事实表存"事件/指标"，维表存"背景信息/属性"；通过 **外键关联**，事实表可以丰富事件信息，便于分析；这就是 **事实表 + 维表 Join** 的根本原因

**示意图：**

```
css                                                    Copy  Edit

[用户维表]      [商品维表]
   U01  Alice       P01  手机
   U02  Bob         P03  耳机

      \             /
       \           /
        [事实表]
    订单ID  用户ID  商品ID  数量  金额
    10001   U01     P01     2     200
    10002   U02     P03     1     120
```

（1）预加载：open()方法，查询维表，存储下来 ==》 定时查询

（2）热存储：存在外部系统 Redis、HBase 等

（3）广播维表

（4）Lookup Join：外部存储，connector 创建，SQL 用法

## 19.FlinkCDC 锁表问题?

（1）FlinkCDC 1.x 同步历史数据会锁表

设置参数不加锁，但只能保证至少一次。

（2）2.x 实现了无锁算法，同步历史数据的时候不会锁表

2.x 在全量同步阶段可以多并行子任务同步，在增量阶段只能单并行子任务同步

# 六、ClickHouse

## 1、ClickHouse基础

## 2、ClickHouse高级

### 1. 执行计划查看 (EXPLAIN)

#### 基本语法

```
EXPLAIN [AST | SYNTAX | PLAN | PIPELINE] [setting = value, ...]
SELECT ... [FORMAT ...]
```

#### 主要类型

- **PLAN**: 查看执行计划（默认）
- **AST**: 查看语法树
- **SYNTAX**: 查看语法优化
- **PIPELINE**: 查看管道执行计划

#### 实用示例

```
-- 查看执行计划
EXPLAIN SELECT database,table,count(1) cnt
FROM system.parts
WHERE database in ('datasets','system')
GROUP BY database,table;

-- 查看语法优化
EXPLAIN SYNTAX SELECT number = 1 ? 'hello' : 'world' FROM numbers(10);
```

### 2. 建表优化最佳实践

#### 2.1 数据类型选择

- **时间字段**: 优先使用 `DateTime` 而非字符串或长整型
- **空值处理**: 避免使用 `Nullable` 类型，使用默认值或业务无意义值替代

```
-- 推荐写法
CREATE TABLE t_good(
    id UInt32,
    create_time DateTime DEFAULT now(),
    status UInt8 DEFAULT 0  -- 用0表示空值
) ENGINE = MergeTree()
ORDER BY id;
```

### 2.2 分区和索引设计

- **分区粒度**: 按天分区，也可以指定为 Tuple()，单表亿级数据控制在10-30个分区
- **必须指定索引列**: 通过 `ORDER BY` 指定，遵循"高基数列在前、查询频率高的在前"

```
CREATE TABLE hits_example(
    CounterID UInt32,
    EventDate Date,
    UserID UInt64
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(EventDate)
ORDER BY (CounterID, EventDate, intHash32(UserID));
```

### 2.3 写入优化

- 避免单条或小批量操作
- 每秒2-3次写入，每次2-5万条数据，使用 WAL 预写日志，提高写入性能
- 写入前对数据进行排序

## 3. SQL优化规则

### 3.1 COUNT优化

```
-- 自动优化为读取系统表统计
SELECT count() FROM table_name;  -- 推荐
SELECT count(*) FROM table_name;  -- 推荐
SELECT count(column) FROM table_name;  -- 不会优化
```

### 3.2 谓词下推

> 当 group by 有 having 子句，但是没有 with cube、with rollup 或者 with totals 修饰的时候，having 过滤会下推到 where 提前过滤

ClickHouse会自动将WHERE条件下推到子查询中：

```
-- 原始查询
SELECT * FROM (SELECT UserID FROM visits) WHERE UserID = '123';

-- 自动优化后
SELECT * FROM (SELECT UserID FROM visits WHERE UserID = '123') WHERE UserID =
'123';
```

### 3.3 聚合优化

```
-- 聚合计算外推
SELECT sum(UserID * 2) FROM visits;  -- 优化为: SELECT sum(UserID) * 2

-- 聚合函数消除（GROUP BY键上的min/max/any会被消除）
SELECT max(UserID), sum(amount) FROM visits GROUP BY UserID;
-- 优化为: SELECT UserID, sum(amount) FROM visits GROUP BY UserID;
```

其他优化规则：消除子查询重复字段、删除重复的order by key、删除重复的limit by key、删除重复的USING Key、标量替换、三元运算优化

# 4. 查询优化技巧

## 4.1 单表查询优化

**PREWHERE替代WHERE**（prewhere 只支持MergeTree 族系列引擎的表）

```
#关闭 where 自动转 prewhere(默认情况下， where 条件会自动优化成 prewhere)
set optimize_move_to_prewhere=0;
-- 手动指定PREWHERE（当查询列多于筛选列时效果显著）
SELECT * FROM hits_v1 PREWHERE UserID='123456789';
```

**数据采样**：可极大提升数据分析的性能（采样修饰符只有在 MergeTree engine 表中才有效）

```
SELECT Title, count(*) FROM hits_v1
SAMPLE 0.1  -- 采样10%数据
WHERE CounterID = 57
GROUP BY Title;
```

**列裁剪与分区裁剪**：分区裁剪就是只读取需要的分区，在过滤条件中指定。

```
-- 避免 SELECT *，明确指定需要的列
SELECT UserID, EventDate, URL FROM hits_v1
WHERE EventDate = '2014-03-23';
```

**order by 结合where、limit**

**避免构建虚拟列**

**uniqCombined（近似去重）替代distinct**

**使用物化视图**

**其他注意事项**：查询熔断、关闭虚拟内存、配置join_user_nulls、批量写入时先排序、关注cpu

## 4.2 多表查询优化

用IN代替JOIN

```
-- 推荐：只需要左表数据时使用IN
SELECT * FROM hits WHERE CounterID IN (SELECT CounterID FROM visits);

-- 避免：不必要的JOIN
SELECT a.* FROM hits a LEFT JOIN visits b ON a.CounterID = b.CounterID;
```

大小表JOIN原则：小表在右（因为不管什么join都是拿着右表做为基础比对）

```
-- 小表在右侧（会被加载到内存）
SELECT * FROM big_table a LEFT JOIN small_table b ON a.id = b.id;
```

分布式表使用GLOBAL：右表只会在接收查询请求的那个节点查询一次

```
-- 分布式表JOIN必须加GLOBAL
SELECT * FROM dist_table1 a
GLOBAL LEFT JOIN dist_table2 b ON a.id = b.id;
```

使用字典表、提前过滤

# 5. 数据一致性（重点）

> 查询 CK 手册发现，即便对数据一致性支持最好的 Mergetree，也只是保证**最终一致性**：ReplacingMergeTree适用于在后台清楚重复的数据以节省空间，但是不保证没有重复的数据出现。

## 5.1 手动OPTIMIZE

```
-- 在写入数据后，立刻执行 OPTIMIZE 强制触发新写入分区的合并动作
OPTIMIZE TABLE test_table FINAL;
```

## 5.2 Group By去重查询

```
-- 使用argMax获取最新数据
SELECT
    user_id,
    argMax(score, create_time) AS score,
    argMax(deleted, create_time) AS deleted
FROM test_table
GROUP BY user_id
HAVING deleted = 0;
```

## 5.3 FINAL查询

> 这样在查询的过程中将会执行 Merge 的特殊逻辑（例如数据去重，预聚合等）

```
-- 查询时执行去重逻辑（新版本支持多线程）
SELECT * FROM visits_v1 FINAL WHERE StartDate = '2014-03-17';
```

# 6. 物化视图

> ClickHouse 的物化视图是一种查询结果的持久化，物化视图不会随着基础表的变化而变化，所以它也称为快照（snapshot）

物化视图和普通视图的区别：**普通视图不保存数据，保存的仅仅是查询语句**，查询的时候还是从原表读取数据，可以将普通视图理解为是个子查询。**物化视图则是把查询的结果根据相应的引擎存入到了磁盘或内存中**，对数据重新进行了组织，你可以理解物化视图是完全的一张新表。

### 基本语法

```
CREATE MATERIALIZED VIEW view_name
ENGINE = SummingMergeTree
PARTITION BY toYYYYMM(EventDate)
ORDER BY (EventDate, UserID)
AS SELECT
    UserID,
    EventDate,
    count() AS cnt,
    sum(Income) AS total_income
FROM source_table
GROUP BY UserID, EventDate;
```

### 关键特点

- **优点**: 查询速度快，预计算结果
- **缺点**: 增加写入负担，不适合去重分析
- **数据更新**: 源表写入时自动更新
- **POPULATE**: 不推荐使用，创建时不包含历史数据

## 7. MaterializeMySQL引擎

ClickHouse 20.8.2.3 版本新增加了 MaterializeMySQL 的 database 引擎，该 database 能映射到 MySQL 中的某个 database，并自动在 ClickHouse 中创建对应的ReplacingMergeTree。ClickHouse 服务做为 MySQL 副本，读取 Binlog 并执行 DDL 和 DML 请求，实现了基于 MySQL Binlog 机制的业务数据库实时同步功能。

### 使用细则

**（1）DDL 查询**

MySQL DDL 查询被转换成相应的 ClickHouse DDL 查询（ALTER, CREATE, DROP, RENAME）。如果 ClickHouse 不能解析某些 DDL 查询，该查询将被忽略。

**（2）数据复制**

MaterializeMySQL 不支持直接插入、删除和更新查询，而是将 DDL 语句进行相应转换：

MySQL INSERT 查询被转换为 INSERT with _sign=1。

MySQL DELETE 查询被转换为 INSERT with _sign=-1。

MySQL UPDATE 查询被转换成 INSERT with _sign=1 和 INSERT with _sign=-1。

**（3）SELECT 查询**

如果在 SELECT 查询中没有指定_version，则使用 FINAL 修饰符，返回_version 的最大值对应的数据，即最新版本的数据。

如果在 SELECT 查询中没有指定_sign，则默认使用 WHERE _sign=1，即返回未删除状态（_sign=1)的数据。

**（4）索引转换**

ClickHouse 数据库表会自动将 MySQL 主键和索引子句转换为 ORDER BY 元组。

### 配置MySQL

```
# my.cnf配置，开启binlog和gtid模式
server-id=1
log-bin=mysql-bin
binlog_format=ROW
gtid-mode=on
enforce-gtid-consistency=1
```

### 创建同步

```
-- 开启clickhouse物化引擎
SET allow_experimental_database_materialize_mysql=1;

-- 创建复制管道，同步数据库
CREATE DATABASE ck_mysql ENGINE = MaterializeMySQL(
    'mysql_host:3306',
    'mysql_db',
    'username',
    'password'
);
```

### 数据转换规则

- MySQL INSERT → ClickHouse INSERT (_sign=1)
- MySQL DELETE → ClickHouse INSERT (_sign=-1)
- MySQL UPDATE → ClickHouse INSERT (*sign=-1) + INSERT (*sign=1)

# 3、ClickHouse运维

# 4、ClickHouse相关面试题

## 1.介绍一下ClickHouse

ClickHouse是俄罗斯的Yandex于2016年开源的列式存储数据库（DBMS），使用C++语言编写，主要用于在线分析处理查询（OLAP），能够使用SQL查询实时生成分析数据报告。

ClickHouse专注于快速的数据插入和复杂的分析查询，适用于大数据分析场景。

ClickHouse主要特点（优点）如下：

**1、列式存储**

ClickHouse使用列式存储，将相同类型的数据存储在一起，提供了更高的压缩比和更快的查询速度，常用于聚合和分析操作。

列式储存的好处：

- 对于列的聚合，计数，求和等统计操作原因优于行式存储。
- 由于某一列的数据类型都是相同的，针对于数据存储更容易进行数据压缩，每一列选择更优的数据压缩算法，大大提高了数据的压缩比重。
- 由于数据压缩比更好，一方面节省了磁盘空间，另一方面对于cache也有了更大的发挥空间。

**2、兼容SQL**

几乎覆盖了标准SQL的大部分语法，包括 DDL和 DML，以及配套的各种函数，用户管理及权限管理，数据的备份与恢复。

**3、多样化引擎**

ClickHouse和MySQL类似，把表级的存储引擎插件化，根据表的不同需求可以设定不同的存储引擎。目前包括合并树、日志、接口和其他四大类20多种引擎。

**4、高吞吐写入能力**

ClickHouse采用类LSM Tree的结构，数据写入后定期在后台Compaction。通过类LSM tree的结构，ClickHouse在数据导入时全部是顺序append写，写入后数据段不可更改，在后台compaction时也是多个段merge sort后顺序写回磁盘。顺序写的特性，充分利用了磁盘的吞吐能力，即便在HDD上也有着优异的写入性能。

官方公开benchmark测试显示能够达到50MB-200MB/s的写入吞吐能力，按照每行100Byte估算，大约相当于50W-200W条/s的写入速度。

**5、数据分区与线程级并行**

ClickHouse将数据划分为多个partition，每个partition再进一步划分为多个index granularity，然后通过多个CPU核心分别处理其中的一部分来实现并行数据处理。在这种设计下，单条Query就能利用整机所有CPU。极致的并行处理能力，极大的降低了查询延时。

所以，ClickHouse即使对于大量数据的查询也能够化整为零平行处理。但是有一个弊端就是对于单条查询使用多cpu，就不利于同时并发多条查询。所以对于高qps的查询业务，ClickHouse并不是强项。

# 2.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）**向量化执行**：在查询时采用批量处理（vectorized execution），充分利用 CPU 的 SIMD 指令集，减少函数调用开销。
（2）**列式存储**：查询时只读取需要的列，大幅减少 IO 和内存消耗；列数据类型一致，便于高效压缩
（3）**极致利用单机资源（CPU + 内存）**，不依赖 Hadoop、Spark 这类外部分布式计算框架，减少了中间层的性能损耗。
（4）提供了类 sql 化的语言
（5）支持自定义函数：内置了丰富的函数库，还支持用户自定义函数（UDF），增强了扩展性和灵活性
（6）提供了丰富的表引擎，引擎都经过了优化

# 3.ClickHouse的引擎?

ClickHouse 提供了多种表引擎，不同引擎适用于不同场景，大致分为以下几类：

1. **Log 系列**
    - **TinyLog / StripeLog / Log**：轻量级存储方式，适合小数据量、测试或一次性查询。
    - 特点：写入快，但功能有限（无索引、无分区、无副本）。
2. **Special 特殊引擎**
    - **Memory**：数据常驻内存，读写速度最快，但重启会丢失，常用于临时计算或缓存。
    - **Distributed**：逻辑表引擎，用于分布式查询，把请求分发到多个节点的分片上执行，再聚合结果。
3. **MergeTree 系列（核心引擎）**

- **MergeTree**：最常用，支持大数据量存储，具备主键索引、分区、并行合并等能力。
- **ReplacingMergeTree**：支持基于主键的去重（保留最新版本的数据）。
- **SummingMergeTree**：自动对相同主键的数据进行聚合求和。
- **ReplicatedMergeTree**：支持多副本、数据自动同步，保证高可用。
  👉 **面试必答：大规模生产环境几乎都用 MergeTree 系列。**

4. **集成引擎**
   - 用于和外部系统打通，例如：
     - **MySQL**：把 ClickHouse 表映射到 MySQL 表上，支持查询外部 MySQL 数据。
     - **Kafka**：从 Kafka 流式消费数据。
     - **ODBC / JDBC / HDFS / S3**：访问外部系统或存储。

## 4.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。

1. **事务支持**
   - ClickHouse **不支持事务**，因此无法像传统数据库那样实现严格的"精确一次（Exactly-Once）"写入。
   - Flink Sink 写入 ClickHouse 通常只能保证 **至少一次（At-Least-Once）** 语义，可能会有重复数据。
2. **如何处理重复数据**
   - **ReplacingMergeTree** 引擎：
     - 基于主键自动去重，保留最新版本的数据。
     - 能够实现 **最终一致性**，但在数据合并前，短时间内仍可能看到重复。
   - 查询时加上 `FINAL` 关键字：
     - 强制在查询阶段对重复行进行去重，保证结果一致性。
     - 缺点是会增加查询开销，不建议在高并发实时查询场景下频繁使用。
3. **实践中的一致性方案**
   - **幂等写入**：
     - 在 Flink 写入前生成唯一标识（如业务主键 + 批次号），写入 ClickHouse 时通过主键覆盖，避免重复。
   - **批量写入**：
     - Flink 可以先将数据写入缓存或临时表，再周期性批量写入 ClickHouse，减少重复可能性。
   - **物化视图 / 分区合并**：
     - 借助 ClickHouse 的物化视图或者后台合并机制，进一步清理重复数据。

## 5.Clickhouse 存储多少数据？几张表？

10 几张宽表，每天平均 10 来 G，存储一年。

需要磁盘 10G * 365 天 * 2 副本/0.7 = 约 11T

## 6.ClickHouse用的是本地表还是分布式表？

**1）我们用的本地表，2个副本**

**2）分布式表写入存在的问题：**

假如现有一个 2 分片的集群，使用 clickhouse 插入分布式表。

（1）资源消耗问题：在分片 2 的数据写入临时目录中会产生写放大现象，会大量消耗分片节点 1 的 CPU 和磁盘等资源。

（2）数据准确性和一致性问题：在写入分片 2 的时候，节点 1 或节点 2 的不正常都会导致数据问题。（节点 1 挂了数据丢失、节点 2 挂了或者节点 2 表删了节点 1 会无限制重试，占用资源）。

（3）part 过多问题：每个节点每秒收到一个 Insert Query，N 个节点，分发 N-1 次，一共就是每秒生成 Nx（N-1）个 part 目录。集群 shard 数越多，分发产生的小文件也会越多（如果写本地表就会相对集中些），最后会导致写入到 MergeTree 的 Part 的数会特别多，最后会拖垮整个文件的系统

## 7.ClickHouse的物化视图？

一种查询结果的持久化，记录了查询语句和对应的查询结果。

优点：查询速度快，要是把物化视图这些规则全部写好，它比原数据查询快了很多，总的行数少了，因为都预计算好了。

缺点：它的本质是一个流式数据的使用场景，是累加式的技术，所以要用历史数据做去重、去核这样的分析，在物化视图里面是不太好用的。在某些场景的使用也是有限的。而且如果一张表加了好多物化视图，在写这张表的时候，就会消耗很多机器的资源，比如数据带宽占满、存储一下子增加了很多。

## 8.ClickHouse的优化？

**1）内存优化**

max_memory_usage：单个查询的内存上限，128G 内存的服务器==》设为 100G

max_bytes_before_external_group_by：设为一半，超过该值后启用外部存储（磁盘）做 `group by`，避免内存溢出，一般设为内存的一半，如 50G。

max_bytes_before_external_sort：设为一半，超过该值后启用外部排序，同样设为内存的一半，如 50G。

**优化点**：减少 OOM，同时保障大查询能落盘执行。

**2）CPU**

**max_concurrent_queries**：控制并发查询数，默认 100，可根据 CPU 核心数和业务压力调整到 300/s。

**max_threads**：设置单个查询最大线程数，通常设置为 CPU 核数，避免过度上下文切换。

**3）存储**

SSD 更快

**4）物化视图**

**5）写入时攒批**，避免写入过快导致 too many parts

**6）查询调优**

避免频繁使用 `FINAL`（性能开销大）。

合理使用 `LIMIT`、`PREWHERE`，减少扫描量。

使用 **分区裁剪 + min-max 索引 + bloom filter 索引** 加速过滤。

# 9.ClickHouse新特性Projection?

Projection 意指一组列的组合，可以按照与原表不同的排序存储，并且支持聚合函数的查询。
ClickHouse Projection 可以看做是一种更加智能的物化视图，它有如下特点：

**1）part-level 存储**

相比普通物化视图是一张独立的表，Projection 物化的数据就保存在原表的分区目录中，支持明细数据的普通 Projection 和 预聚合 Projection。

**2）无感使用，自动命中**

可以对一张 MergeTree 创建多个 Projection ，当执行 Select 语句的时候，能根据查询范围，自动匹配最优的 Projection 提供查询加速。如果没有命中 Projection，就直接查询底表。

**3）数据同源、同生共死**

因为物化的数据保存在原表的分区，所以数据的更新、合并都是同源的，也就不会出现不一致的情况了

# 10.Cilckhouse的索引、底层存储?

## 1）索引机制

（1）**一级索引：稀疏索引（Primary Index）**

- 基于 `ORDER BY` 定义的列构建。
- 默认索引粒度是 **8192 行**，即每 8192 行存储一个索引点。
- 用来快速定位数据块，缩小扫描范围，但并非逐行索引。

（2）**二级索引：跳数索引（Skip Index）**

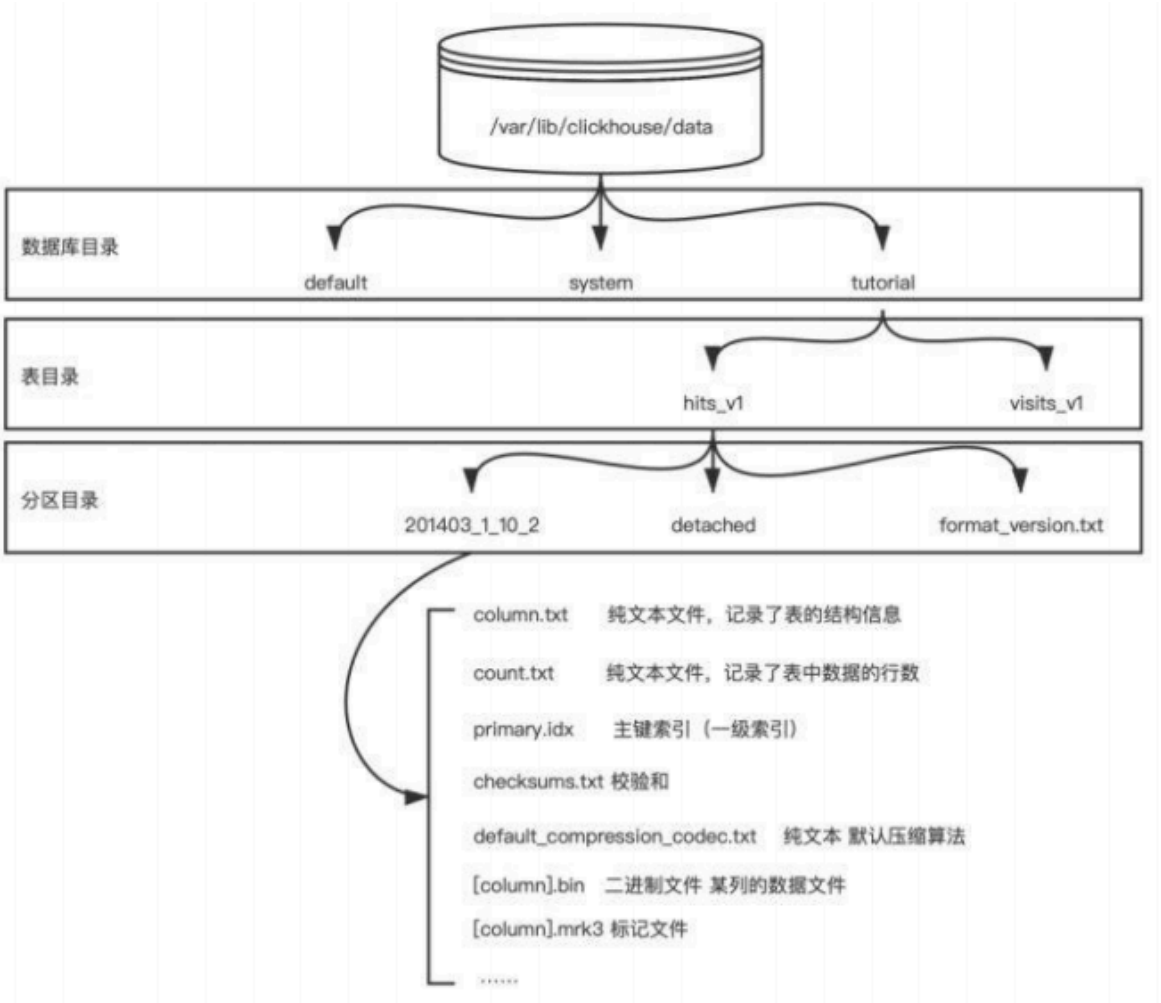
- 用于辅助过滤，加速查询。
- 常见类型：
  - **minmax**：记录某列在数据块的最小值、最大值，快速跳过不符合范围的数据。
  - **set**：记录某列在数据块里的 distinct 值集合，适合基数较低的字段。
  - **bloom filter**：基于概率的索引，适合模糊匹配（如 `LIKE`、`IN`）。
- 可以和一级索引结合，大幅减少扫描数据量。

👉 **总结**：ClickHouse 的索引不是像传统数据库那样的 B+Tree，而是"稀疏 + 跳跃"索引，更适合大规模数据分析。

## 2）底层存储

- **存储目录**：
  - 默认在 `/var/lib/clickhouse/data/`，每个数据库是一个文件夹，每张表也会有独立的子目录。
- **文件组织（以 MergeTree 为例）**：
  - 每个数据分区（partition）对应一个子目录。
  - 分区目录下是多个 **part（数据块）**，每个 part 里包含多种文件：
    - `*.bin`：存储实际列数据，经过压缩。
    - `*.mrk2`：对应的 mark 文件，记录索引点的偏移位置。
    - `primary.idx`：主键稀疏索引文件。
    - `checksums.txt`：校验文件。
    - `count.txt`：行数信息。
  - 当后台合并（merge）执行时，小的 part 会合并成大的 part，提高查询性能。

👉 **总结**：ClickHouse 的底层存储是 **列式存储**，按分区和 part 管理，结合稀疏索引和 mark 文件实现快速定位和扫描。



| 202103_1_10_2 | 目录 | 分区目录，由分区+LSM 生成的 |
| --- | --- | --- |
| detached | 目录 | 通过 DETACH 语句卸载后的表分区存放位置 |
| format_version.txt | 文本文件 | 纯文本，记录存储的格式 |

分区目录命名 = 分区 ID_最小数据块编号_最大数据块编号_层级构成．数据块编号从 1 开始自增，新创建的数据块最大和最小编号相同，当发生合并时会将其修改为合并的数据块编号．同时每次合并都会将层级增加 1．

# 七、Prometheus+Grafana

## 1、Prometheus 基础知识

### 1.1 Prometheus 简介

- **起源**：受启发于 Google 的 Borgmon 监控系统
- 发展历程：
  - 2012年：由前Google工程师在Soundcloud开始研发
  - 2015年：发布早期版本
  - 2016年5月：加入CNCF基金会（继Kubernetes后第二个）
  - 2016年6月：发布1.0版本
  - 2017年底：发布2.0版本（全新存储层）

### 1.2 Prometheus 核心特点

#### 1.2.1 易于管理

- **单一二进制**：核心只有一个二进制文件，无第三方依赖，唯一需要的就是本地磁盘
- **Pull模型**：基于拉取模式的架构，可在任何环境搭建
- **服务发现**：支持动态服务发现能力动态管理监控目标。

#### 1.2.2 监控内部状态

- 提供丰富的Client库，用户可以轻松的在应用程序中添加对 Prometheus 的支持
- 支持应用程序内部真实运行状态监控

#### 1.2.3 强大的数据模型

所有采集的监控数据均以指标(metric)的形式保存在**内置的时间序列数据库**当中(**TSDB**)。所有的样本除了基本的指标名称以外，还包含一组用于描述该样本特征的标签

```
http_request_status{code='200',content_path='/api/path',environment='production'} =>
[value1@timestamp1,value2@timestamp2...]
```

- **指标名称**：http_request_status
- **标签维度**：{code='200',content_path='/api/path',environment='production'}
- **时间序列值**：[value1@timestamp1,value2@timestamp2...]

### 1.2.4 PromQL查询语言

- 内置强大的数据查询语言
- 支持数据查询、聚合、可视化、告警

### 1.2.5 高性能

- 单实例可处理：
  - 数以百万的监控指标
  - 每秒数十万的数据点

### 1.2.6 可扩展性

- 支持联邦集群
- 功能分区(sharding) + 联邦集群(federation)

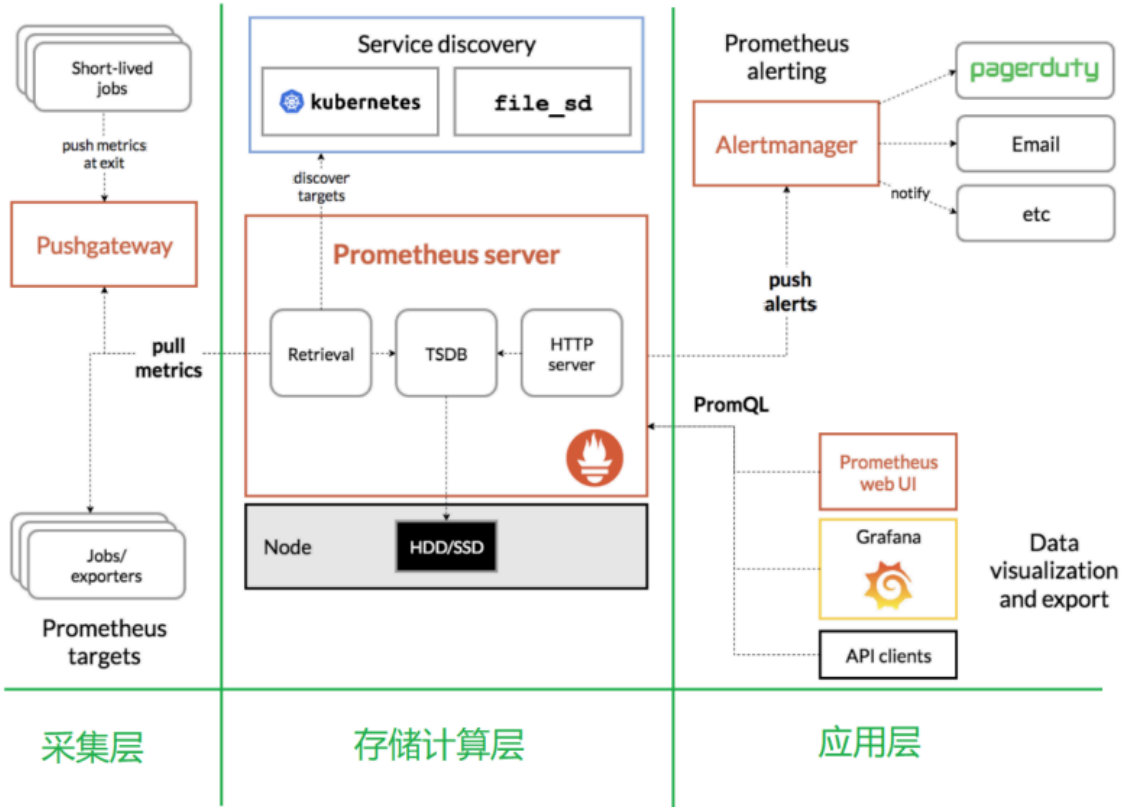
### 1.2.7 易于集成

- 支持多种语言SDK：Java、JMX、Python、Go、Ruby、.Net、Node.js
- 第三方集成：Graphite、Statsd、MySQL、PostgreSQL等120+项

### 1.2.8 可视化支持

- 内置Prometheus UI
- 支持Grafana
- 提供API开发自定义UI

## 2、Prometheus 架构与组件



## 2.1 生态圈组件

- **Prometheus Server**：主服务器，负责收集和存储时间序列数据
- **Client Libraries**：应用程序代码插桩，将监控指标嵌入到被监控应用程序中
- **Pushgateway**：推送网关，支持短期作业
- **Exporter**：数据摄取组件（如HAProxy、StatsD、Graphite）
- **Alertmanager**：告警处理组件

## 2.2 三层架构

### 2.2.1 存储计算层

- **Prometheus Server**：包含存储引擎和计算引擎
- **Retrieval**：取数组件，从Pushgateway或Exporter拉取数据
- **Service Discovery**：动态发现监控目标
- **TSDB**：时间序列数据库，核心存储与查询
- **HTTP Server**：对外提供HTTP服务

### 2.2.2 采集层

- **短作业**：通过API直接推送给Pushgateway
- **长作业**：Retrieval直接从Job或Exporter拉取

### 2.2.3 应用层

- **AlertManager**：告警管理（邮件、电话、短信等）
- **数据可视化**：Prometheus WebUI、Grafana等

# 3、Prometheus 安装部署

> Prometheus 基于 Golang 编写，编译后的软件包，不依赖于任何的第三方依赖。只需要下载对应平台的二进制包，解压并且添加基本的配置即可正常启动 Prometheus Server。

## 3.1 Prometheus Server 安装

### 3.1.1 安装步骤

```
# 1. 解压安装包
tar -zxvf prometheus-2.29.1.linux-amd64.tar.gz -C /opt/module
mv prometheus-2.29.1.linux-amd64 prometheus-2.29.1

# 2. 修改配置文件 prometheus.yml
vim prometheus.yml
```

### 3.1.2 配置文件详解

```
# 全局配置
global:
  scrape_interval: 15s         # 拉取数据时间间隔
  evaluation_interval: 15s     # 规则验证时间间隔
```

```
# 规则配置文件
rule_files:
  # - "first_rules.yml"

# 采集目标配置
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['hadoop202:9090']

  - job_name: 'pushgateway'
    static_configs:
      - targets: ['hadoop202:9091']
    labels:
      instance: pushgateway

  - job_name: 'node_exporter'
    static_configs:
      - targets: ['hadoop202:9100', 'hadoop203:9100', 'hadoop204:9100']
```

### 3.1.3 启动命令

```
nohup ./prometheus --config.file=prometheus.yml > ./prometheus.log 2>&1 &
```

## 3.2 Pushgateway 安装

> PushGateway 就是一个中转组件，通过配置 Flink on YARN 作业将 metric 推到PushGateway，Prometheus 再从 PushGateway 拉取就可以了

```
# 解压
tar -zxvf pushgateway-1.4.1.linux-amd64.tar.gz -C /opt/module
mv pushgateway-1.4.1.linux-amd64 pushgateway-1.4.1

# 启动
nohup ./pushgateway --web.listen-address :9091 > ./pushgateway.log 2>&1 &
```

## 3.3 Node Exporter 安装

> 实际的监控样本数据的收集则是由 Exporter 完成，如主机的 CPU 使用率, 内存，磁盘等信息

```
# 解压
tar -zxvf node_exporter-1.2.2.linux-amd64.tar.gz -C /opt/module
mv node_exporter-1.2.2.linux-amd64 node_exporter-1.2.2

# 启动
./node_exporter

# 设为系统服务
sudo vim /usr/lib/systemd/system/node_exporter.service
```

**Service配置文件**：

```
[Unit]
Description=node_export
Documentation=https://github.com/prometheus/node_exporter
After=network.target

[Service]
Type=simple
User=atguigu
ExecStart=/opt/module/node_exporter-1.2.2/node_exporter
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

# 4、PromQL 查询语言

> Prometheus 通过指标名称（metrics name）以及对应的一组标签（labelset）唯一定义一条时间序列。指标名称反映了监控样本的基本标识，而 label 则在这个基本特征上为采集到的数据提供了多种特征维度。用户可以基于这些特征维度过滤，聚合，统计从而产生新的计算后的一条时间序列。PromQL 是 Prometheus **内置的数据查询语言**，其提供对时间序列数据丰富的查询，聚合以及逻辑运算能力的支持。并且被广泛应用在 Prometheus的日常应用当中，包括对数据查询、可视化、告警处理当中。可以这么说，PromQL 是Prometheus 所有应用场景的基础，理解和掌握 PromQL 是 Prometheus 入门的第一课。

## 4.1 基本概念

- **指标名称**：metrics name
- **标签集**：labelset
- **时间序列**：指标名称 + 标签集唯一标识

## 4.2 基本用法

### 4.2.1 查询时间序列

```
# 查询所有时间序列
prometheus_http_requests_total

# 等同于
prometheus_http_requests_total{}

# 标签匹配
prometheus_http_requests_total{instance="localhost:9090"}

# 标签排除
prometheus_http_requests_total{instance!="localhost:9090"}

# 正则匹配
prometheus_http_requests_total{environment=~"staging|testing|development"}

# 正则排除
prometheus_http_requests_total{environment!~"staging|testing|development"}
```

### 4.2.2 范围查询

```
# 瞬时向量（最新值）
prometheus_http_requests_total{}

# 区间向量（过去5分钟）
prometheus_http_requests_total{}[5m]
```

**时间单位**：

- s：秒
- m：分钟
- h：小时
- d：天
- w：周
- y：年

### 4.2.3 时间位移操作

```
# 5分钟前的瞬时数据
prometheus_http_requests_total{} offset 5m

# 昨天一天的区间数据
prometheus_http_requests_total{}[1d] offset 1d
```

### 4.2.4 聚合操作

```
# 查询系统所有http请求总量
sum(prometheus_http_requests_total)

# 按照mode计算主机CPU平均使用时间
avg(node_cpu_seconds_total) by (mode)

# 按主机查询CPU使用率
sum(sum(irate(node_cpu_seconds_total{mode!='idle'}[5m])) /
sum(irate(node_cpu_seconds_total[5m]))) by (instance)
```

## 4.3 PromQL 操作符

### 4.3.1 数学运算符

- `+` 加法
- `-` 减法
- `*` 乘法
- `/` 除法
- `%` 求余
- `^` 幂运算

### 4.3.2 布尔运算符

- `==` 相等
- `!=` 不相等
- `>` 大于
- `<` 小于
- `>=` 大于等于
- `<=` 小于等于

**bool修饰符**：

```
# 返回布尔值而非过滤
prometheus_http_requests_total > bool 1000
```

### 4.3.3 集合运算符

- `and` 并且
- `or` 或者
- `unless` 排除

### 4.3.4 操作符优先级

1. `^`
2. `*`, `/`, `%`
3. `+`, `-`
4. `==`, `!=`, `<=`, `<`, `>=`, `>`
5. `and`, `unless`
6. `or`

## 4.4 聚合操作符

- `sum` 求和
- `min` 最小值
- `max` 最大值
- `avg` 平均值
- `stddev` 标准差
- `stdvar` 标准差异
- `count` 计数
- `count_values` 对value计数
- `bottomk` 后n条时序
- `topk` 前n条时序
- `quantile` 分布统计

**语法**：

```
<aggr-op>([parameter,] <vector expression>) [without|by (<label list>)]
```

**示例：**

```
# 移除instance标签
sum(prometheus_http_requests_total) without (instance)

# 只保留指定标签
sum(prometheus_http_requests_total) by (code,handler,job,method)

# 前5位时序数据
topk(5, prometheus_http_requests_total)

# 中位数
quantile(0.5, prometheus_http_requests_total)
```

# 5、Prometheus 与 Flink 集成

## 5.1 集成步骤

### 5.1.1 拷贝jar包

```
cp flink-metrics-prometheus-1.12.0.jar /opt/module/flink-prometheus/lib/
```

### 5.1.2 修改Flink配置

在 `flink-conf.yaml` 中添加：

```
##### 与Prometheus集成配置 #####
metrics.reporter.promgateway.class:
org.apache.flink.metrics.prometheus.PrometheusPushGatewayReporter
metrics.reporter.promgateway.host: hadoop202
metrics.reporter.promgateway.port: 9091
metrics.reporter.promgateway.jobName: flink-metrics-ppg
metrics.reporter.promgateway.randomJobNameSuffix: true
metrics.reporter.promgateway.deleteOnShutdown: false
metrics.reporter.promgateway.interval: 30 SECONDS
```

### 5.1.3 提交任务

```
bin/flink run -t yarn-per-job -c
com.atguigu.flink.chapter02.Flink03_WordCount_UnboundStream ./flink-base-1.0-
SNAPSHOT-jar-with-dependencies.jar
```

## 5.2 集成原理

- Flink通过Push模式将metrics推送到Pushgateway
- Prometheus从Pushgateway拉取Flink指标数据
- 解决了Flink on YARN作业动态发现的问题

# 6、Grafana 可视化

## 6.1 Grafana 安装

```
# 解压
tar -zxvf grafana-enterprise-8.1.2.linux-amd64.tar.gz -C /opt/module/

# 启动
nohup ./bin/grafana-server web > ./grafana.log 2>&1 &
```

## 6.2 添加数据源

1. 访问：http://hadoop202:3000
2. 默认用户名密码：admin/admin
3. 配置 → Data Sources → Add data source
4. 选择Prometheus，配置URL：http://hadoop202:9090

## 6.3 创建仪表盘

### 6.3.1 手动创建

1. 点击"+"→ Dashboard
2. Add an empty panel
3. 配置查询指标和可视化选项

### 6.3.2 导入模板

1. 访问：https://grafana.com/dashboards
2. 搜索Flink或Node Exporter模板
3. 下载JSON文件
4. Grafana中Import → Upload JSON file

## 6.4 常用模板

- **Flink模板**：搜索"Flink"，选择下载量高的模板
- **Node Exporter模板**：搜索"Node Exporter"，选择中文版本

# 7.、监控告警配置

## 7.1 任务失败监控

### 7.1.1 监控原理

基于 `flink_jobmanager_job_uptime` 指标：

- 正常情况：按配置频率递增
- 任务失败：数值不再变化

### 7.1.2 查询表达式

```
((flink_jobmanager_job_uptime)-(flink_jobmanager_job_uptime offset 30s))/100
```

- 正常值：非0
- 失败值：0

## 7.2 网络延时/重启监控

### 7.2.1 查询表达式

```
((flink_jobmanager_job_uptime offset 30s)-(flink_jobmanager_job_uptime))/1000
```

- 正常值：-30
- 延时：突然小于-30
- 重启：突然大于0（uptime清零重新计数）

## 7.3 重启次数监控

### 7.3.1 基于指标

`flink_jobmanager_job_numRestarts`：记录重启次数

### 7.3.2 差值监控

```
flink_jobmanager_job_numRestarts - (flink_jobmanager_job_numRestarts offset 30s)
```

- 等于1：刚重启一次
- 等于0：无重启

## 7.4 告警配置步骤

1. 创建监控面板
2. 设置告警规则（Alert Rules）
3. 配置通知渠道（Notification channels）
4. 设置告警阈值和条件

# 8、Prometheus相关面试题

## 1.Prometheus 的核心监控模型（拉取 vs 推取）及其优缺点?

Prometheus 主要采用 Pull（拉取）模型，由 Server 主动从配置的 Exporter（如 node_exporter, mysqld_exporter）或支持 Prometheus 协议的应用程序（如 Kubernetes API Server）抓取（scrape）指标数据。优点：中心化管理目标、避免客户端推送失败导致的数据丢失、更容易知道哪些目标不可达、天然支持动态服务发现。缺点：需要目标暴露 HTTP 端点、实时性可能略低于推模式（取决于抓取间隔）、需要处理网络可达性问题。

## 2.Prometheus 的数据是如何存储的？什么是 TSDB？

Prometheus 使用内置的自研时序数据库 TSDB（Time Series Database） 存储数据。它针对时间序列数据（指标名 + 标签集 + 时间戳 + 值）进行了高度优化，支持高效写入和按时间范围查询。数据按时间分块存储（Block），并定期压缩合并旧数据。最新数据在内存中的 Head Block 中，定期写入持久化块。

## 3.解释一下 Prometheus 数据模型中的“指标（Metric）”、“标签（Label）”和“样本（Sample）”？

Metric: 监控指标的名称（如 http_requests_total）。

Label: 用于区分同一指标的不同维度（键值对，如 method="GET", status="200", instance="10.0.0.1:9100"）。标签是 Prometheus 强大灵活性的核心。

Sample: 一个具体的数值点，包含：一个浮点数值（Value）和一个精确到毫秒的时间戳（Timestamp）。一个时间序列由其指标名称和一组唯一的标签集完全定义。

## 4.Alertmanager 的主要职责是什么？

Alertmanager 负责处理来自 Prometheus Server 发送的告警通知（Alert）。它的核心职责包括：

**分组（Grouping）**：将同类型告警（如来自不同实例的同一故障）合并为一个通知，避免告警风暴。

抑制（Inhibition）：当某个更高级别告警触发时，抑制相关的低级别告警（如网络故障时抑制所有依赖该网络的服务器告警）。

**静默（Silencing）**：临时屏蔽特定时间段或匹配特定条件的告警（如计划维护期间）。

**路由（Routing）**：根据告警标签将告警通知分发给不同的接收者（Receiver），如邮件给运维组、PagerDuty给值班人员、Slack给相关团队。

**去重（Deduplication）**：减少重复告警通知。

## 5.Prometheus Server配置文件的关键配置？

关键配置文件 prometheus.yml：

global：全局配置（抓取间隔、超时等）。
scrape_configs：定义抓取目标（job_name, static_configs 静态目标列表，或 file_sd_configs/kubernetes_sd_configs/consul_sd_configs 等动态服务发现配置）。
rule_files：指定告警规则文件（.rules）的路径。
alerting：配置 Alertmanager 实例地址（告诉 Prometheus 把告警发到哪里）。

## 6.Prometheus 监控大量目标时可能会遇到哪些性能问题？如何优化？

**问题：**抓取间隔过短、目标过多、指标基数过大（特别是高基数标签）、查询复杂数据量大导致查询慢或内存溢出、存储 I/O 压力大。

**优化：**

- 抓取端： 适当增加抓取间隔（scrape_interval），使用服务发现动态管理目标，避免不必要的抓取。确保 Exporter 高效。
- 指标端： 严格控制标签值，避免使用高基数的标签（如用户 ID、Trace ID 直接作为标签）。优化应用程序和 Exporter 暴露的指标。

- 存储/查询端： 使用 Recording Rules 预计算常用且耗时的查询结果。优化 PromQL 查询（避免 .* 匹配过多时间序列，合理使用聚合）。增加 Prometheus Server 资源（CPU、内存、高性能磁盘/SSD）。考虑分片（Sharding）或联邦（Federation）部署。
- 架构： 对于超大规模，考虑远程存储（Thanos, Cortex, Mimir）和查询前端（如 Thanos Query）分离读写和存储。

## 7.Alertmanager 的“分组（Grouping）”是如何工作的？请举例说明其好处？

Alertmanager 根据告警的标签（通常配置在路由的 group_by 字段中，如 ['alertname', 'cluster', 'service']）将具有相同标签值的告警分组到一起。当组内第一个告警触发时，会等待 group_wait 时间（如 30s），将这段时间内到达的同组告警合并，然后发送一个包含所有这些告警信息的通知。好处：避免同一时间段内同一服务的多个实例或同一类型故障产生大量独立告警通知（告警风暴），显著减少通知数量，使通知更清晰（例如：“10 台服务器上的 CPU 使用率过高” 而不是 10 封单独的邮件）。

## 8.Alertmanager 的“抑制（Inhibition）”规则如何配置？应用场景是什么？

在 alertmanager.yml 中配置 inhibit_rules。示例：

inhibit_rules:

```
inhibit_rules:
  - source_match: # 源告警（更严重/高级别的告警）
      severity: 'critical'
    target_match: # 目标告警（将被抑制的告警）
      severity: 'warning'
    equal: ['cluster', 'alertname'] # 当这些标签值相同时，目标告警被抑制
```

**应用场景：** 当发生更严重的故障时，抑制由其引起的、不那么重要或冗余的告警。例如：当整个集群网络故障（severity: critical）时，抑制该集群内所有服务器宕机（severity: warning）和所有服务不可用（severity: warning）的告警，因为根本原因是网络，收到网络告警就足够了。

## 9.如何临时静默（Silence）一个告警？

主要通过 Alertmanager 的 Web UI 操作：

1、访问 Alertmanager UI (http://alertmanager:9093)。
2、点击 "Silences" -> "New Silence"。
3、通过匹配器（Matcher）指定要静默的告警（如 alertname=HighCpuUsage, instance=10.0.0.1:9100）。
4、设置静默的开始时间和持续时间。
5、添加注释（为什么静默）。
6、点击 "Create"。创建后，匹配到的告警在静默期内将不会触发通知。

## 10.如果 Grafana 无法从 Prometheus 获取数据，你会如何排查？

1、检查 Grafana 数据源配置： URL 是否正确？端口是否正确？是否有认证问题（如果有设置）？点击 "Save & Test" 返回什么错误？
2、检查 Prometheus 状态： 访问 Prometheus Web UI (http://prometheus:9090) 是否正常？检查 /targets 页面，Grafana 查询的数据源（如 node job）是否 UP 且数据正常？直接在 Prometheus UI 的 Graph 页面尝试执行 Grafana 中相同的查询，是否成功？是否有报错？

3、检查网络连通性： 从 Grafana 服务器能否 telnet/nc 到 Prometheus 的端口（默认 9090）？是否有防火墙/安全组规则阻止?
4、检查 Prometheus 日志： 查看 Prometheus 服务日志，是否有抓取失败、查询错误或其他异常?
5、检查 Grafana 日志： 查看 Grafana 服务日志，是否有查询错误信息?
6、检查时间范围： 确认 Grafana 仪表盘或面板选择的时间范围内 Prometheus 确实有数据。
7、检查查询语法： 在 Prometheus UI 中验证 Grafana 面板使用的 PromQL 是否正确？是否有拼写错误或无效的标签匹配?

# 八、InfluxDB

在InfluxDB 3中，一个表通常对应一种类型的时间序列数据（例如，`cpu` 表存放CPU指标，`temperature` 表存放温度读数）

学习参考文档：https://blog.csdn.net/weixin_42331508/article/details/148041926

对于InfluxDB 3.x，官方推荐使用 `influxdb3-java` 这个新的、轻量级的、社区维护的客户端库

> 重要特性：标签集合与顺序的不可变性
> InfluxDB 3的一个核心设计是，当数据首次写入一个新表时，该表中出现的标签键及其顺序（InfluxDB内部决定的顺序）就被固定下来了。之后，你不能再为这个表添加新的标签键。这意味着在设计初期，必须仔细规划好一个表需要哪些核心的、用于索引和分组的维度作为标签。如果后续确实需要新的索引维度，可能需要重新设计表结构或创建新表。

# 1、InfluxDB基础

## 第1章 认识 InfluxDB

### 1.1 InfluxDB 的使用场景

**定义**: InfluxDB 是一种时序数据库，主要用于监控场景，包括运维和 IOT（物联网）领域。

**典型应用场景**:

- 服务器 CPU 使用情况监控（每10秒写入一条数据）
- 查询过去30秒CPU平均使用情况
- 配置报警规则（查询结果>阈值时触发报警）
- IOT 设备监控（机械设备轴承震动频率、农田湿度温度等）

### 1.2 为什么不用关系型数据库

#### 1.2.1 写入性能

- **关系型数据库问题**: 采用B+树结构，写入时可能触发叶裂变，产生磁盘随机读写，降低写入速度
- **时序数据库优势**: 采用LSM Tree变种，顺序写磁盘，单点每秒数十万写入能力

#### 1.2.2 数据价值（冷热数据差别明显）

- **热数据**: 近期数据（如最近10分钟），经常查询，存储在内存中
- **冷数据**: 历史数据（10分钟前），应该被压缩存储到磁盘以节省空间

#### 1.2.3 时间不可倒流，数据只写不改

- 时序数据描述实体在不同时间的状态
- 历史时刻的数据不会因为当前操作而改变
- 主要是插入操作，很少有更新需求

## 1.3 TICK 技术栈的演进

**数据应用的4个步骤**

1. **数据采集**
2. **存储**
3. **查询（包括聚合操作）**
4. **报警**

**1.X 的 TICK 技术栈**

- **T**: Telegraf - 数据采集组件，收集&发送数据到 InfluxDB
- **I**: InfluxDB - 存储数据&发送数据到 Chronograf
- **C**: Chronograf - 总的用户界面，起到总的管理功能
- **K**: Kapacitor - 后台处理报警信息

**2.X 的进一步融合**

ICK 功能全部融入 InfluxDB，仅需安装 InfluxDB 就能得到管理页面、定时任务和报警功能。

## 1.4 InfluxDB 版本比较与选型

#### 1.4.1 版本特性比较

**底层引擎**: 1.x 与 2.x 原理相差不大，但概念有转变（db/rp → org/bucket）

**查询语言**:

- **1.x**: InfluxQL（近似SQL风格）
- **2.x**: FLUX 查询语言（函数与管道符，更符合时序数据特性）

**集成性**: 2.x 有更好的管理页面和集成性

#### 1.4.2 选型考虑因素

**市场现状**: 1.X 使用量更多，但2.X 更便利

**集群功能**:

- 从0.11版本开始闭源集群功能
- 免费只能使用单节点版
- 企业版才有集群功能
- 开源替代方案:
    - InfluxDB Cluster（对应1.8.10）
    - InfluxDB Proxy（对应1.2-1.8和2.3）

**FLUX语言支持**:

- 自1.7和2.0开始推出

- 作为独立项目运作，目标成为通用标准
- 2.X 对FLUX支持更好

**产品概况**:

- InfluxDB 1.8: 仍在小版本更新，主要修复BUG
- InfluxDB 2.4: 较新版本，仍在增加新特性
- InfluxDB 企业版 1.9: 需购买，有集群功能
- InfluxDB Cloud: 云服务版本

**本课程选择**: InfluxDB 2.4

# 第2章 安装部署 InfluxDB

## 2.1 下载安装

**两种安装方式**:

1. 包管理工具安装（apt、yum）
2. 直接下载二进制程序压缩包（推荐）

**下载命令**:

bash

```
wget https://dl.influxdata.com/influxdb/releases/influxdb2-2.4.0-linux-amd64.tar.gz
```

**解压**:

bash

```
tar -zxvf influxdb2-2.4.0-linux-amd64.tar.gz -C /opt/module
```

**启动服务**:

bash

```
./influxd
```

## 2.2 进行初始化配置

**访问地址**: http://hadoop102:8086

**初始化步骤**:

1. 点击 GET STARTED
2. 填写组织名称、用户名称、用户密码
3. 点击 CONTINUE 完成初始化

# 第3章 InfluxDB 入门（借助 Web UI）

## 3.1 数据源相关

### 3.1.1 Load Data（加载数据）

**支持的数据格式**:

- CSV
- 带Flux注释的CSV
- InfluxDB行协议

**功能**:

1. 上传数据文件
2. 提供各种编程语言的连接库代码模板
3. 配置Telegraf输入插件

### 3.1.2 管理存储桶（Buckets）

**Bucket概念**: 类似关系型数据库中的database

**主要操作**:

1. **创建Bucket**: 指定名称和数据过期时间
2. **调整设置**: 修改过期时间和名称（不建议重命名）
3. **设置Label**: 为Bucket添加标签，这个功能一般很少用
4. **添加数据**: 导入数据或创建抓取任务（被抓取的数据在格式上必须符合 prometheus 数）

### 3.1.3 示例1：创建Bucket并从文件导入数据

**步骤**:

1. 创建名为example01的Bucket，删除策略保留默认的 NEVER
2. 进入Line Protocol数据上传页面
3. 手动输入数据:

```
// 数据格式叫做 InfluxDB 行协议
people,name=tony age=12
people,name=xiaohong age=13
people,name=xiaobai age=14
people,name=xiaohei age=15
people,name=xiaohua age=12
```

4. 指定时间精度并写入数据

**重要特点**: InfluxDB是无模式数据库，无需预先创建measurement或指定字段类型

### 3.1.4 管理Telegraf数据源

**Telegraf**: InfluxDB生态中的数据采集组件，可自动采集各种时序数据

**功能**:

1. 创建Telegraf配置文件

2. 管理配置文件接口
3. 修改配置文件

#### 3.1.5 示例2：使用Telegraf收集数据

**步骤概览**:

1. 下载并解压Telegraf
2. 创建新的Bucket（example02）
3. 在Web UI创建Telegraf配置
4. 设置环境变量INFLUX_TOKEN
5. 启动Telegraf
6. 验证数据采集结果

**启停脚本示例**:

bash

```
#!/bin/bash
# 完整的启停脚本包含start、stop、status功能
```

#### 3.1.6 管理抓取任务

**抓取任务**: InfluxDB定期访问URL，将数据入库

**特点**:

- 轮询间隔固定为10秒
- 目标URL必须暴露Prometheus数据格式
- InfluxDB自身暴露监控接口: http://localhost:8086/metrics

#### 3.1.7 示例3：让InfluxDB主动拉取数据

**步骤**:

1. 创建存储桶example03
2. 创建抓取任务example03_scraper
3. 设置目标路径
4. 验证抓取结果

#### 3.1.8 管理API Token

**API Token作用**:

- InfluxDB权限管理主要体现在API Tokens上
- 客户端将token放在HTTP请求头中
- 服务端根据token判断权限（读写、删除、创建等）

**Token管理**:

1. 查看token权限
2. 修改token名称
3. 临时关停或删除token

4. 创建不同权限的token

**Token类型模板**:

- Read/Write API Token: 仅读写存储桶
- All Access API Token: 所有权限

## 3.2 查询工具

### 3.2.1 前言

需要掌握FLUX语言和时序数据库的数据模型

### 3.2.2 了解Data Explorer

**功能**: 探索数据，作为FLUX语言的IDE

**页面结构**:

- **上半部分**: 数据预览区
- **下半部分**: 查询编辑区

**查询编辑区**:

1. **查询构造器**: 通过点击方式完成查询，自动生成FLUX语句
2. **FLUX脚本编辑器**: 手动编写FLUX脚本

**数据预览区**:

- 默认显示折线图
- 支持散点图、饼图、原始数据等展示方式

**其他功能**:

1. 导出数据为CSV
2. 保存为仪表盘单元
3. 创建定时任务
4. 定义全局变量

### 3.2.3 示例4：使用查询构造器

**查询步骤**:

1. 选择FROM存储桶（test_init）
2. 设置Filter（选择go_goroutines测量）
3. 注意查询时间范围（默认1h）
4. 设置窗口聚合选项（默认10s，平均值）
5. 提交查询

**原理**: 查询构造器生成FLUX脚本，可以切换到脚本编辑器查看

#### 3.2.4 了解Notebook

**Notebook**: 模仿Jupyter Notebook，用于开发、文档编写、运行代码

**主要用途**:

- 执行FLUX代码、可视化数据
- 创建报警或计划任务
- 数据降采样或清洗
- 生成Runbooks
- 将数据回写到存储桶

**与Data Explorer区别**:

- Data Explorer: 一锤子买卖
- Notebook: 将数据展示拆分为多个步骤

> 一个 NoteBook 工作流就是多个 Cell 按照先后顺序组合起来的执行流程。这些 Cell 中间随时可以插入别的 Cell，而且 Cell 和 Cell 还可以调换顺序。

**Cell类型**:

- **数据源Cell**: 查询构造器、FLUX脚本
- **可视化Cell**: Table、图表、笔记
- **行为Cell**: 报警、定时任务

#### 3.2.5 示例5：使用Notebook

**工作流范式**: 查询数据 → 展示数据 → 进一步处理 → 任务设置/报警

**Notebook控件**:

1. 时区转换（Local按钮）
2. 仅显示可视化（Presentation按钮）
3. 删除、复制、运行按钮

# 第4章 FLUX 语法

> **没完全过时**：Flux 仍然可用，特别是在 InfluxDB 2.x 环境里（比如已有 Flux 脚本、Dashboards、复杂数据处理场景）。
>
> **逐渐边缘化**：在 **InfluxDB 3.x 以后**，SQL 是主推，Flux 不再是核心发展方向。
>
> **建议**：
>
> - 新项目 → 优先考虑 **SQL**，因为生态和兼容性更强。
> - 老项目 → 如果已有大量 Flux 逻辑，可以继续用，但要考虑未来迁移路径。

## 4.1 认识FLUX语言

**定义**: 函数式的数据脚本语言，将查询、处理、分析和操作数据统一为一种语法

**设计理念**: 类似水处理过程

1. 从源头抽取数据
2. 在管道上进行系列处理

3. 输送到目的地

**目标**: 成为像SQL一样的通用标准，不仅仅是InfluxDB的特定语言

## 4.2 最简示例

**FLUX查询的操作流程**:

1. 从数据源查询指定数量的数据
2. 根据时间或字段筛选数据
3. 将数据进行处理或聚合
4. 返回最终结果

**三个示例对比**:

```
// 示例1：从InfluxDB查询
from(bucket: "example-bucket")
|> range(start: -1d)
|> filter(fn: (r) => r._measurement == "example-measurement")
|> mean()
|> yield(name: "_results")

// 示例2：从CSV文件查询
import "csv"
csv.from(file: "path/to/example/data.csv")
|> range(start: -1d)
|> filter(fn: (r) => r._measurement == "example-measurement")
|> mean()
|> yield(name: "_results")

// 示例3：从PostgreSQL查询
import "sql"
sql.from(
    driverName: "postgres",
    dataSourceName: "postgresql://user:password@localhost",
    query: "SELECT * FROM TestTable",
)
|> filter(fn: (r) => r.UserID == "123ABC456DEF")
|> mean(column: "purchase_total")
|> yield(name: "_results")
```

**关键函数说明**:

- `from()`: 指定数据源
- `|>`: 管道转发符
- `range()`, `filter()`: 数据过滤
- `mean()`: 计算平均值
- `yield()`: 返回结果

## 4.3 铭记FLUX是查询语言

**重要原则**: FLUX脚本必须返回表流才能成功执行

**单值转表流示例**:

```
import "array"
x = 1
array.from(rows: [{"value":x}])
```

## 4.4 版本兼容性

**重要提醒**: InfluxDB版本与FLUX语言版本有绑定关系

**参考文档**: https://docs.influxdata.com/flux/v0.x/influxdb-versions/

## 4.5 FLUX基本语法

### 4.5.1 注释

```
// 这是一行注释
```

### 4.5.2 变量与赋值

```
s = "foo"     // string
i = 1         // integer
f = 2.0       // float
```

### 4.5.3 基本表达式

```
1 + 1                                 // Returns 2
10 * 3                                // Returns 30
(12.0 + 18.0) / (2.0 ^ 2.0) + (240.0 % 55.0)  // Returns 27.5
"John " + "Doe " + "is here!"    // Returns John Doe is here!
```

### 4.5.4 谓词表达式

**比较运算符**:

```
"John" == "John"          // Returns true
41 < 30                   // Returns false
"John" == "John" and 41 < 30     // Returns false
"John" == "John" or 41 < 30      // Returns true
```

**正则表达式**:

```
"abcdefg" =~ "abc|bcd"    // Returns true
"abcdefg" !~ "abc|bcd"    // Returns false
```

**逻辑运算符**:

```
a = true
b = false
x = a and b      // Returns false
y = a or b       // Returns true
z = not a        // Returns false
```

#### 4.5.5 控制语句

**条件子句**（类似三元表达式）：

```
x = 0
y = if x == 0 then "hello" else "world"
```

# 第5章 FLUX中的数据类型

## 5.1 10个基本数据类型

### 5.1.1 Boolean（布尔型）

**类型转换**:

```
bool(v: "true")       // Returns true
bool(v: 0.0)          // Returns false
bool(v: 0)            // Returns false
bool(v: uint(v: 1))   // Returns true
```

### 5.1.2 Bytes（字节序列）

**创建bytes**:

```
bytes(v:"hello")      // Returns [104 101 108 108 111]
```

**十六进制转bytes**:

```
import "contrib/bonitoo-io/hex"
hex.bytes(v: "FF5733")  // Returns [255 87 51]
```

### 5.1.3 Duration（持续时间）

**时间单位**:

- ns: 纳秒, us: 微秒, ms: 毫秒, s: 秒
- m: 分钟, h: 小时, d: 天, w: 周
- mo: 日历月, y: 日历年

**示例**:

```
1ns          // 1纳秒
1h30m        // 1小时30分钟
3d12h4m25s   // 3天12小时4分钟25秒
```

**类型转换**:

```
duration(v: "1h30m")        // Returns 1h30m
duration(v: 1000000)        // Returns 1ms
```

**算术运算**:

```
duration(v: int(v: 6h4m) + int(v: 22h32s))     // Returns 1d4h4m32s
```

**时间运算**:

```
import "date"
date.add(d: 1w, to: 2021-01-01T00:00:00Z)       // 加一周
date.sub(d: 1w, from: 2021-01-01T00:00:00Z)     // 减一周
```

### 5.1.4 Regular Expression（正则表达式）

**定义**:

```
/\w/                        // 正则表达式字面量
```

**逻辑判断**:

```
"abc" =~ /\w/               // Returns true
"foo" !~ /^f/               // Returns false
"FOO" =~ /(?i)foo/          // Returns true (忽略大小写)
```

**字符串转正则**:

```
import "regexp"
regexp.compile(v: "^-[a-z0-9]{7}")
```

**字符串替换**:

```
import "regexp"
regexp.replaceAllString(r: /a(x*)b/, v: "-ab-axxb-", t: "T")  // Returns "-T-T-"
```

### 5.1.5 String（字符串）

**定义**:

```
"abc"
"string with double \" quote"
"string with backslash \\"
"日本語"
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e"     // 十六进制编码
```

**类型转换**:

```
string(v: 42)               // Returns "42"
```

#### 5.1.6 Time（时间点）

**RFC3339格式**:

```
2020-01-01
2020-01-01T00:00:00Z
2020-01-01T00:00:00.000Z
```

**时间操作**:

```
import "date"
x = 2020-01-01T19:22:31Z
date.hour(t:x)          // Returns 19
```

#### 5.1.7 Float（浮点数）

**定义**:

```
0.0
123.4
-123.456
```

**科学计数法**:

```
float(v: "1.23456e+78")     // Returns 1.23456e+78
```

**无限和NaN**:

```
float(v: "+Inf")            // Returns +Inf
float(v: "NaN")             // Returns NaN
```

**类型转换**:

```
float(v: "1.23")            // 1.23
float(v: true)              // Returns 1.0
float(v: 123)               // Returns 123.0
```

#### 5.1.8 Integer（整数）

**64位有符号整数**:

- 最小值: -9223372036854775808
- 最大值: 9223372036854775807

**类型转换**:

```
int(v: "123")               // 123
int(v: true)                // Returns 1
int(v: 1d3h24m)             // Returns 98640000000000 (纳秒数)
int(v: 2021-01-01T00:00:00Z) // Returns 1609459200000000000 (Unix时间戳)
int(v: 12.54)               // Returns 12 (截断)
```

**十六进制转换**:

```
import "contrib/bonitoo-io/hex"
hex.int(v: "e240")          // Returns 123456
```

#### 5.1.9 UIntegers（无符号整数）

**类型转换**:

```
uint(v: "123")              // 123
uint(v: true)               // Returns 1
uint(v: -54321)             // Returns 18446744073709497295 (整数环绕)
```

#### 5.1.10 Null（空值）

**创建null值**:

```
import "internal/debug"
debug.null(type: "string")  // 返回null字符串
debug.null(type: "int")     // 返回null整数
```

**判断是否为null**:

```
import "internal/debug"
x = debug.null(type: "string")
y = exists x                // Returns false
```

#### 5.1.11 正则表达式类型

```
regex = /^foo/
"foo" =~ regex              // Returns true
"bar" =~ regex              // Returns false
```

#### 5.1.12 display函数

```
x = bytes(v: "foo")
display(v: x)               // Returns "0x666f6f"
```

## 5.2 FLUX类型不代表InfluxDB类型

**重要提醒**: Duration和正则表达式等类型不能存储到InfluxDB中，只是FLUX语言特有的类型

## 5.3 四个复合类型

#### 5.3.1 Record（记录）

**定义**:

```
{foo: "bar", baz: 123.4, quz: -2}
{"Company Name": "ACME", "Street Address": "123 Main St.", id: 1123445}
```

**取值方式**:

```
c = {name: "John Doe", address: "123 Main St.", id: 1123445}
c.name                          // 点表示法
c["Company Name"]               // 中括号方式
```

**嵌套record**:

```
customer = {
    name: "John Doe",
    address: {
        street: "123 Main St.",
        city: "Pleasantville",
        state: "New York"
    }
}
customer.address.street     // 链式调用
customer["address"]["city"]
customer["address"].state
```

**扩展record**:

```
c = {name: "John Doe", id: 1123445}
{c with name: "Xiao Ming", pet: "Spot"}  // 覆盖并添加
```

**获取所有keys**:

```
import "experimental"
c = {name: "John Doe", id: 1123445}
experimental.objectKeys(o: c)    // Returns [name, id]
```

### 5.3.2 Array（数组）

**定义**:

```
["1st", "2nd", "3rd"]
[1.23, 4.56, 7.89]
[10, 25, -15]
```

**取值**:

```
arr = ["one", "two", "three"]
arr[0]                          // Returns "one"
arr[2]                          // Returns "three"
```

**检查元素**:

```
names = ["John", "Jane", "Joe", "Sam"]
contains(value: "Joe", set: names)  // Returns true
```

### 5.3.3 Dictionary（字典）

**定义**:

```
[0: "Sun", 1: "Mon", 2: "Tue"]
["red": "#FF0000", "green": "#00FF00", "blue": "#0000FF"]
```

**取值**:

```
import "dict"
positions = [
    "Manager": "Jane Doe",
    "Asst. Manager": "Jack Smith",
    "Clerk": "John Doe",
]
dict.get(dict: positions, key: "Manager", default: "Unknown position")
```

**其他操作**:

```
// 从列表创建字典
list = [{key: "k1", value: "v1"}, {key: "k2", value: "v2"}]
dict.fromList(pairs: list)

// 插入键值对
dict.insert(dict: exampleDict, key: "k3", value: "v3")

// 移除键值对
dict.remove(dict: exampleDict, key: "k2")
```

### 5.3.4 Function（函数）

**定义**:

```
square = (n) => n * n
square(n:3)                    // Returns 9
```

**默认参数**:

```
chengfa = (a, b=100) => a * b
chengfa(a:3)                   // Returns 300
```

## 5.4 函数包

**导入方式**:

```
import "array"
import "math"
import "influxdata/influxdb/sample"
```

**universe包**: 默认加载，函数可直接使用 **其他包**: 需要import后使用

# 第6章 如何使用FLUX语言的文档

## 6.1 如何查看函数文档

**文档地址**: https://docs.influxdata.com/flux/v0.x/

**使用方法**:

1. 点击Standard library查看所有函数包
2. 点击包名查看包内所有函数
3. 点击具体函数查看详细说明和使用示例

## 6.2 避免使用实验中的函数

**experimental包**: 实验性质，函数可能在未来版本中变化或被废弃

**建议**: 生产环境避免使用experimental包中的函数

## 6.3 查看函数可用版本

每个函数文档标题下方都标记了适用的FLUX版本范围，注意版本兼容性。

# 第7章 FLUX查询InfluxDB

## 7.1 前言

建议跟随视频课学习本章内容

## 7.2 FLUX查询InfluxDB的语法

**必须以from -> range开头**:

flux

```
from(bucket: "test_init")
|> range(start: -1h)
```

**重要**: range必须紧跟在from后面，否则会报错

## 7.3 表、表流以及序列

**概念统一**:

- **序列**: InfluxDB的数据管理方式
- **表**: 关系型数据库的数据结构
- **表流**: FLUX的统一概念，多张表的集合

**关系**:

- 一个序列对应一张表
- 表流是全表，子表是按*field*、*tag set*和measurement分组后的结果
- 聚合函数只在子表中进行聚合
- 表中的一行对应序列中的一个数据点

### 7.4 filter维度过滤

使用filter函数对数据按_measurement、标签集和字段进行过滤：

flux

```
|> filter(fn: (r) => r._measurement == "cpu")
|> filter(fn: (r) => r.host == "server01")
|> filter(fn: (r) => r._field == "usage_idle")
```

### 7.5 类型转换函数与下划线字段

**下划线字段的约定**:

- 以下划线开头的字段代表FLUX函数的依赖约定
- 很多函数依赖_value字段才能正常运行
- 程序员应遵守约定，不要随意修改下划线开头的字段

**类型转换示例**:

flux

```
|> toInt()    // 需要_value字段存在
```

### 7.6 map函数

**作用**: 遍历表流中的每一条数据

**示例**:

flux

```
import "array"
array.from(rows: [{"name":"tony"},{"name":"jack"}])
|> map(fn: (r)=> {
    return if r["name"] == "tony" then {"_name": "tony 不是jack"} else
{"_name":"jack 不是tony"}
})
```

**参数要求**: fn参数必须是单个record输入、record输出的函数

### 7.7 自定义管道函数

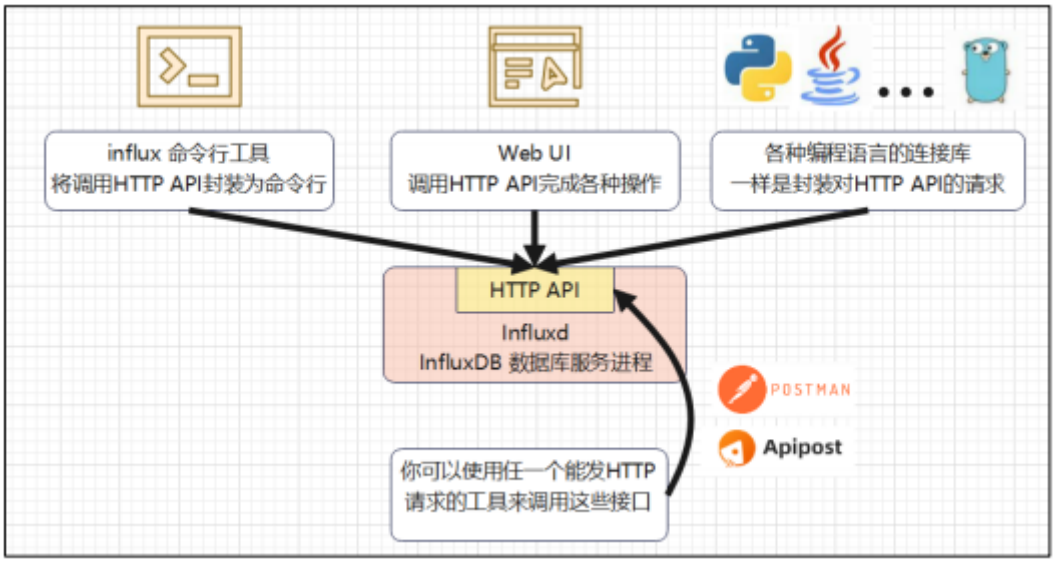
### 7.8 在文档中区分管道函数和普通函数

如果没有<-tables: stream[A]，那么它就是一个普通函数

### 7.9 window和aggregateWindow函数

### 7.10 yield和join

## 第8章 InfluxDB交互方式




## HTTP API基础

- InfluxDB启动后提供一套HTTP API
- 外部程序**仅能**通过HTTP API与InfluxDB通信
- influx命令行、Web UI、各编程语言客户端库内部都是对HTTP API的封装
- 所有客户端交互都需要**API TOKEN**

# 第9章 InfluxDB HTTP API详解

## 9.1 Token准备

- 使用账号密码登录Web UI选择或创建API TOKEN
- 课程使用"tony's Token"(具有全部权限)
- **生产环境应谨慎使用**，防止Token被劫持
- 每次HTTP请求都需要在请求头携带token

## 9.2 接口测试工具

- **推荐工具**: ApiPost 6 (国产软件，对标Postman)
- **官网**: https://www.apipost.cn/
- **配置公用header**: 为目录添加Authorization header，所有接口自动携带

## 9.3 接口授权方式

### 9.3.1 Token授权方式

- **成功标志**: 访问 `/api/v2/authorizations` 返回200状态码和token信息
- **失败表现**: 401状态码，返回"没有授权"信息

### 9.3.2 登录授权方式

**创建登录会话**:

- POST请求到登录接口
- 使用Basic auth认证方式
- 输入用户名密码(课程: tony/11111111)

**Basic auth认证原理**:

1. 用户名:密码 → `tony:11111111`
2. Base64编码 → `dG9ueToxMTExMTExMQ==`
3. 添加前缀 → `Basic dG9ueToxMTExMTExMQ==`
4. 放入Authorization请求头

**Cookie机制**:

- 服务端返回set-cookie响应头
- 浏览器/Session自动携带cookie
- InfluxDB根据cookie判断权限

**安全问题**:

- Base64不是加密算法，容易被解码
- 不应将Web UI暴露在公网
- 生产环境应使用token而非登录方式
- Cookie有过期时间(约30分钟)

## 9.4 HTTPS配置

### 9.4.1 使用OpenSSL生成证书

```
openssl req -x509 -nodes -newkey rsa:2048 \
-keyout /opt/module/influxdb2_linux_amd64/selfsigned.key \
-out /opt/module/influxdb2_linux_amd64/selfsigned.crt \
-days 60
```

**参数说明**:

- `req -x509`: 生成自签名证书格式
- `-newkey rsa:2048`: 生成2048位RSA密钥
- `-keyout`: 指定密钥文件名
- `-out`: 证书保存路径
- `-days`: 证书有效期(天)

### 9.4.2 启动HTTPS服务

```
./influxd \
--tls-cert="/opt/module/influxdb2_linux_amd64/selfsigned.crt" \
--tls-key="/opt/module/influxdb2_linux_amd64/selfsigned.key"
```

#### 9.4.3 验证HTTPS

- HTTP请求返回400状态码
- 提示"向HTTPS服务器发送了HTTP请求"
- 改用HTTPS协议可正常访问

#### 9.4.4 更新现有配置

- 更新Telegraf配置中的URL为HTTPS
- 更新Scrapers中的URL为HTTPS

### 9.5 其他生产安全考虑

#### 9.5.1 IP白名单

- **参考**: https://docs.influxdata.com/influxdb/v2.4/security/enable-hardening/
- **作用**: 限制FLUX脚本可以访问的地址
- **原因**: FLUX语言具有发送网络请求的能力

#### 9.5.2 机密管理

- **参考**: https://docs.influxdata.com/influxdb/v2.4/security/secrets/
- **用途**: 避免在FLUX脚本中硬编码用户名密码
- **实现**: 将敏感信息以键值对形式存储在InfluxDB中

**示例**:

```
// 不安全的方式
sql.from(
    driverName: "postgres",
    dataSourceName: "postgresql://tony:11111111@localhost",
    query:"SELECT * FROM example-table",
)

// 安全的方式
import "influxdata/influxdb/secrets"
username = secrets.get(key: "POSTGRES_USERNAME")
password = secrets.get(key: "POSTGRES_PASSWORD")
sql.from(
    driverName: "postgres",
    dataSourceName: "mysql://${username}:${password}@localhost",
    query:"SELECT * FROM example-table",
)
```

#### 9.5.3 Token管理类型

- **操作者Token**: 跨组织管理权限，对所有组织和资源有完全读写权限
- **全权限Token**: 对单个组织中所有资源的完全读写权限
- **读/写Token**: 对组织中特定存储桶的读写权限

#### 9.5.4 禁用开发功能

- `/metrics`: 监控InfluxDB运行指标
- `Web UI`: 图形界面交互
- `/debug/pprof`: Go程序运行时指标

### 9.6 API文档使用

#### 9.6.1 查看方式

- **本地**: `http(s)://localhost:8086/docs`
- **在线**: https://docs.influxdata.com/influxdb/v2.4/api/

#### 9.6.2 OpenAPI支持

- 访问 `http://localhost:8086/doc` 下载OpenAPI文档
- 支持Postman/ApiPost自动生成接口测试

## 第10章 influx命令行工具

### 10.1 安装

- **重要**: InfluxDB 2.1版本后，influx命令行工具需单独安装
- **下载**: 从GitHub获取发行版
- **项目地址**: https://github.com/influxdata/influx-cli

### 10.2 配置influx-cli

#### 10.2.1 创建配置

```
./influx config create --config-name influx.conf \
--host-url http://localhost:8086 \
--org atguigu \
--token
ZA8uWTSRFflhKhFvNW4TcZwwvd2NHFW1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAWg4XMv4tb
9TP3w== \
--active
```

#### 10.2.2 配置文件位置

- 配置保存在 `~/.influxdbv2/configs` 文件中
- 支持多配置管理

#### 10.2.3 配置操作

- **更新配置**: `./influx config update --config-name influx.conf`
- **切换配置**: `influx config influx.conf`
- **删除配置**: `./influx config remove influx2.conf`

### 10.3 主要命令功能

```
命令    功能      说明
apply 应用      InfluxDB模板
auth   认证管理 API Token
backup备份      备份数据(仅InfluxDB OSS)
bucket桶        管理存储桶
config配置      管理配置文件
delete删除      删除数据点
export导出      导出资源为模板
org    组织      组织管理
ping   检查      健康状态检查
query   查询     执行FLUX查询
restore恢复     从备份恢复数据
task    任务     任务管理
write   写向     InfluxDB写数据
```

# 第11章 Java操作InfluxDB

## 11.1 项目设置

### Maven依赖

```
<dependencies>
    <dependency>
        <groupId>com.influxdb</groupId>
        <artifactId>influxdb-client-java</artifactId>
        <version>6.5.0</version>
    </dependency>
</dependencies>
```

## 11.2 客户端连接

### 创建客户端对象

```
// 健康检查(无需token)
InfluxDBClient influxDBClient =
InfluxDBClientFactory.create("http://localhost:8086");

// 完整连接(需要token、org、bucket)
InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org,
bucket);
```

## 11.3 健康状态检查

```
System.out.println(influxDBClient.ping()); // 返回boolean
```

## 11.4 数据查询

### 11.4.1 query方法

```
List<FluxTable> query = queryApi.query("from(bucket:\"test_init\") |> range(start:-2m)");
```

- 返回 `List<FluxTable>` 对象
- 对应FLUX查询语言中的表流概念
- 可以访问groupKey、数据记录等

### 11.4.2 queryRaw方法

```
String result = queryApi.queryRaw("from(bucket:\"test_init\") |> range(start:-2m)");
```

- 返回原始CSV格式字符串
- 直接返回HTTP API的原始响应

## 11.5 数据写入

### 同步写入 vs 异步写入

- **同步写入**: 立即发送请求，线程阻塞等待完成
- **异步写入**: 数据放入缓冲区，攒批发送

### 11.5.1 三种写入方式

#### 1. Point对象写入

```
Point point = Point.measurement("temperature")
    .addTag("location", "west")
    .addField("value", 55D)
    .time(Instant.now(), WritePrecision.MS);
writeApiBlocking.writePoint(point);
```

#### 2. 行协议写入

```
writeApiBlocking.writeRecord(WritePrecision.NS, "temperature,location=west value=60.0");
```

#### 3. POJO类写入

```
// @Measuremet 注解必须加到类上，表示这个类对应 InfluxDB 中的哪个测量名称。
@Measurement(name = "temperature")
private static class Temperature {
    @Column(tag = true)
    String location;

    @Column
    Double value;

    @Column(timestamp = true)
```

```
    Instant time;
}

Temperature temperature = new Temperature();
temperature.location = "west";
temperature.value = 40D;
temperature.time = Instant.now();
writeApiBlocking.writeMeasurement(WritePrecision.NS, temperature);
```

#### 11.5.2 异步写入配置

- **缓冲区大小**: writeApi 里有一个缓冲区，这个缓冲区的大小默认是 10000 条数据
- **批次大小**: 虽然有缓冲区但是 writeApi 写出数据并不是一次把整个缓冲区都写出去，而是按照批次（默认是 1000 条）的单位来写
- 刷写方式:
  - 手动触发: `writeApi.flush()`
  - 关闭客户端: `influxDBClient.close()`

### 11.6 V1 API兼容

```
InfluxDBClient influxDBClient = InfluxDBClientFactory.createV1(/*参数*/);
```

## 第12章 InfluxDB模板

### 12.1 模板概念

- **定义**: YAML格式配置文件
- **内容**: 完整的仪表盘、Telegraf配置、报警配置
- **目标**: 开箱即用
- **官方仓库**: https://github.com/influxdata/community-templates

### 12.2 模板使用示例

#### 12.2.1 安装Docker监控模板

```
influx apply -f https://raw.githubusercontent.com/influxdata/community-templates/master/docker/docker.yml
```

#### 12.2.2 模板创建的资源

- **存储桶**: docker
- **Telegraf配置**: Docker Monitor
- **仪表盘**: Docker仪表盘
- 报警规则:
  - 容器CPU使用率持续15分钟超过80%
  - 容器硬盘使用率超过80%
  - 容器内存使用率持续15分钟超过80%
  - 容器非正常退出

#### 12.2.3 运行Telegraf

```
#!/bin/bash
export INFLUX_TOKEN=your_token
export INFLUX_HOST=http://localhost:8086/
export INFLUX_ORG=atguigu
/opt/module/telegraf-1.23.4/usr/bin/telegraf --config
http://localhost:8086/api/v2/telegrafs/09edf888eeeb6000
```

#### 12.2.4 删除模板实例

```
# Web UI删除
# 或使用influx-cli
influx stacks remove -o atguigu --stack-id=09ee20c80d692000
```

### 12.3 模板的不足

#### 12.3.1 FLUX兼容性问题

- FLUX语言版本变化快，向前兼容性不佳
- 不同InfluxDB版本对应不同FLUX版本
- 社区模板维护不及时

#### 12.3.2 生态对比

- Grafana社区更活跃，模板更丰富
- 支持InfluxDB作为数据源的Grafana模板有1117个
- 建议使用InfluxDB + Grafana方案

## 第13章 定时任务

### 13.1 定时任务概念

- **定义**: 定时执行的FLUX脚本
- **功能**: 查询数据 → 修改/聚合 → 写回InfluxDB或执行其他操作

### 13.2 任务创建示例

#### 13.2.1 需求

- 每30秒调度一次
- 查询最近30秒的第一条数据
- 将数据转为JSON
- 通过HTTP发送给外部服务

#### 13.2.2 FLUX脚本

```
import "json"
import "http"

option task = { name: "example_task", every: 30s, offset:0m }

from(bucket: "test_init")
```

```
|> range(start:-30s)
|> filter(fn: (r) => r["_measurement"] == "go_goroutines")
|> first(column: "_value")
|> map(
    fn: (r) => {
        status_code = http.post(url:"http://localhost:8080/", data:
json.encode(v:r))
        return {r with status_code:status_code}
    }
)
```

#### 13.2.3 创建任务

```
./influx task create --org atguigu -f /opt/module/examples/example_task.flux
```

### 13.3 数据迟到问题

- **问题**: 网络延迟导致数据晚于预期时间到达
- **解决方案**: 使用offset参数延迟任务执行时间
- **示例**: `offset: 5s` 将任务执行时间推迟5秒

### 13.4 Cron表达式支持

```
option task = {
    cron: "0 * * * *",  // 每小时执行一次
}
```

## 第14章 InfluxDB仪表盘

### 14.1 仪表盘基础

- **Cell**: 仪表盘中的单个图形组件
- **数据源**: 每个Cell都是一个FLUX查询语句

### 14.2 仪表盘控件

#### 基础控件

- **手动刷新**: 重新执行查询
- **自动刷新**: 定时自动刷新
- **时区切换**: Local/UTC时区显示
- **查询范围**: 设定查询时间范围
- **添加Cell/Note**: 添加图形或Markdown说明

#### 高级功能

- **变量显示**: 支持动态变量控制
- **注解功能**: Shift+鼠标左键添加参考线
- **全屏模式**: 全屏显示仪表盘
- **夜间模式**: 深色主题

## 14.3 动态仪表盘示例

### 14.3.1 创建查询变量

```
import "influxdata/influxdb/schema"
schema.tagValues(bucket: "example02", tag:"cpu")
```

### 14.3.2 创建CSV变量

- **类型**: CSV
- **值**: cpu0, cpu1, cpu2, cpu3, cpu1|cpu2
- **正则支持**: cpu1|cpu2 表示cpu1或cpu2

### 14.3.3 在FLUX中使用变量

```
basedata = from(bucket: "example02")
|> range(start: v.timeRangeStart, stop: v.timeRangeStop)
|> filter(fn: (r) => r["_measurement"] == "cpu")
|> filter(fn: (r) => r["_field"] == "usage_user")
|> filter(fn: (r) => regexp.matchRegexpString(r:regexp.compile(v:v.cpuxxx),
v:r["cpu"]))

basedata |> aggregateWindow(every: 1m, fn: median, createEmpty: false) |>
yield(name: "median")
basedata |> aggregateWindow(every: 1m, fn: max, createEmpty: false) |>
yield(name: "max")
basedata |> aggregateWindow(every: 1m, fn: min, createEmpty: false) |>
yield(name: "min")
```

# 第15章 InfluxDB服务进程参数

## 15.1 influxd命令

```
命令功能说明
downgrade降级将元数据格式降级匹配旧版本
help帮助打印帮助信息
inspect检查检查磁盘数据库数据
recovery恢复恢复操作权限，管理token、组织、用户
run运行运行influxd服务(默认)
upgrade升级从1.x升级到2.4
version版本打印当前版本
```

## 15.2 重要命令详解

### 15.2.1 inspect命令

```
# 查看数据存储情况
./influxd inspect report-tsm

# 导出存储桶数据为行协议
./influxd inspect export-tsm
```

#### 15.2.2 recovery命令

```
# 为用户创建操作者token
./influxd recovery auth create-operator --username tony --org atguigu

# 列出所有token
./influxd recovery auth list
```

### 15.3 配置方式

#### 15.3.1 命令行参数

```
./influxd --http-bind-address=:8088
```

#### 15.3.2 环境变量

```
export INFLUXD_HTTP_BIND_ADDRESS=:8089
./influxd
```

#### 15.3.3 配置文件

**支持格式**: config.json, config.toml, config.yaml

**示例config.json**:

```
{
    "http-bind-address": ":9090"
}
```

### 15.4 常用配置项

- **bolt-path**: BoltDB文件路径
- **engine-path**: InfluxDB数据文件路径
- **sqlite-path**: SQLite路径
- **flux-log-enabled**: 是否开启FLUX日志(默认false)
- **log-level**: 日志级别(debug/info/error，默认info)

# 2、InfluxDB相关面试题

## 1. 什么是 InfluxDB，它主要解决什么问题？

InfluxDB 是一款 **开源时间序列数据库（TSDB）**，专门用于存储和分析带有时间戳的数据，比如 IoT 传感器数据、应用监控指标、日志数据等。
优势：高写入性能、时序查询优化、支持 downsampling、内置函数支持聚合和窗口分析。
常见应用场景：**DevOps 监控（metrics/logs/traces）、物联网（IoT）、金融交易流数据。**

## 2. InfluxDB 的数据模型是什么样的？

核心概念：

- **measurement**：类似“表名”的概念，表示一类时间序列。
- **tag（标签）**：索引字段（string 类型），用于过滤、分组；高基数会影响内存与索引。

- **field（字段）**：非索引的度量值（float、integer、boolean、string），写入量通常大。
- **timestamp**：纳秒级时间戳，决定时序点的时间维度。
- **series**：由 `measurement + 完整 tag set` 唯一确定的一条时间序列。
  **要点：**标签用于高效过滤，字段用于存储度量；“不该做 tag 的别做 tag”，避免高基数爆炸。

数据点结构Line Protocol：

```
measurement,tag1=v1,tag2=v2 field1=val1,field2=val2 timestamp
```

**举例：**

```
weather,city=beijing,device=d1 temperature=26.5,pm25=40i,ok=true,desc="sunny" 1730544000000000000
```

- measurement：`weather`
- tags：`city=beijing, device=d1`（仅 string）
- fields：`temperature`(float), `pm25`(integer, 后缀 i), `ok`(boolean), `desc`(string, 引号)
- timestamp：纳秒（可配秒、毫秒、微秒、纳秒）
  **要点：相同 measurement+tagset+timestamp 的重复写入会覆盖**该点的字段值。

## 3. InfluxDB 和传统关系型数据库（MySQL、PostgreSQL）的区别?

- 优化目标不同：
  - RDBMS → 事务、关系建模、通用查询
  - InfluxDB → 高吞吐写入、时间序列分析
- 存储结构：
  - RDBMS → B+ 树索引
  - InfluxDB → TSM（Time-Structured Merge Tree），适合时间序列写入
- 查询语言：
  - MySQL → SQL
  - InfluxDB → InfluxQL（类似 SQL）/Flux/SQL(3.x)

## 4. InfluxDB 的存储引擎（TSM Tree）是怎么工作的?

- 数据先写入 **WAL（Write Ahead Log）**，保证崩溃恢复
- 后台刷入内存中的 **TSM 文件**（Time-Structured Merge Tree）
- TSM 文件不可变，通过 **LSM 类似的 compaction** 机制合并，减少存储碎片
- 这种结构特别适合 append-only 的时序写入场景

### 4.1 TSM/TSI 是什么？写入与查询流程简述?

**TSM（Time-Structured Merge Tree）**：列式、时间分段、可压缩的存储格式；WAL 先写后刷盘 -> 写成 TSM 文件 -> 后台 **compaction** 合并。

**TSI（Time Series Index）**：把索引从内存转为**磁盘可持久化索引**，降低内存占用，支持高基数。
**流程：**写入 -> WAL -> 缓存 -> 刷盘为 TSM -> compaction；查询先命中索引（TSI）定位 series，再扫描对应 TSM block。

## 5. InfluxDB 如何实现高效查询?

- **Tag 自动索引** → 类似 inverted index，加速按标签过滤
- **时间戳排序存储** → 范围查询快
- **Retention Policy（保留策略）** → 自动过期数据，减少扫描
- **Continuous Query/Task** → 预聚合减少实时计算压力

## 6. InfluxQL 和 Flux 有什么区别?

- **InfluxQL**：类 SQL，支持 SELECT、GROUP BY、WHERE，偏向查询和聚合
- **Flux**：函数式脚本语言（管道），支持 join、跨库查询、数据转换，更灵活
- 现状：在 InfluxDB 3.x 里，官方主推 **SQL**，Flux 地位边缘化。**选择建议：**简单查询/兼容老系统用 InfluxQL；复杂 ETL、跨 bucket 计算、告警工作流用 Flux。

## 7. InfluxDB 3.x 为何要引入 SQL?

- 标准化 → 开发者更容易上手
- 基于 Apache Arrow + DataFusion → 性能好、可与其他系统互操作
- 降低学习成本，SQL 社区生态成熟

## 8. 如何写一条插入数据的 InfluxDB Line Protocol?

```
cpu,host=server01,region=uswest value=0.64 1672531200000000000
```

解释：

- measurement：cpu
- tags：host=server01, region=uswest
- field：value=0.64
- timestamp：1672531200000000000（纳秒级）

## 9. InfluxQL常用查询示例?

**查询最近1小时的评价cpu使用率"**

```
SELECT MEAN(usage)
FROM cpu
WHERE time > now() - 1h
GROUP BY time(1m), host
```

含义：

- 取最近 1 小时
- 按 1 分钟时间窗口聚合
- 分主机计算平均值

**最近 15 分钟平均值（按 1m 分桶）：**

```
SELECT mean("cpu") FROM "sys"."autogen"."cpu_load"
WHERE time >= now() - 15m AND "host"='h1'
GROUP BY time(1m), "host" fill(none);
```

**TopN：**

```
SELECT top("latency", 5) FROM "api" WHERE time >= now() - 1h GROUP BY "endpoint";
```

**百分位：**

```
SELECT percentile("value", 95) FROM "rt" WHERE time >= now() - 24h;
```

## 10. InfluxDB 如何处理跨库或跨表查询？

- InfluxQL → 不支持跨库 JOIN
- Flux → 可以用 `join()` 结合多个数据源
- 3.x SQL → 支持多 measurement 查询，但跨库 JOIN 仍有限制

## 11. 如何优化 InfluxDB 的写入性能？

- 批量写入（batch write）而不是单点写入
- 使用 UDP/HTTP batch API
- 合理设计 Tag（避免高基数 tag）
- 调整 shard duration（过大或过小都影响性能）

## 12. 什么是高基数（High Cardinality时间线膨胀）问题？如何避免？

- **定义**：series 数量过多导致大量索引（由 measurement+tagset 组合决定）。高基数会导致内存、索引（TSI）与查询开销激增。
- **评估**：`SHOW SERIES CARDINALITY`（v1），v2 可用 API/监控指标；同时关注 `series`、`max-values-per-tag` 等指标。
- **控制**：
  1. 把变化频繁/唯一值（如 UUID、时间戳、随机数）放到 **field** 而非 tag。
  2. 规范化 tag 值（枚举/聚合）。
  3. 拆分 measurement，降低 tag 组合。
  4. 设置写入前端（Telegraf/应用）做**采样/聚合**。

## 13. InfluxDB 的 Retention Policy 是什么？

- 数据保留策略，定义数据存储多久自动删除
- 例：

```
CREATE RETENTION POLICY "one_week" ON "metrics" DURATION 7d REPLICATION 1 DEFAULT
```

- 这样 metrics 数据只保留 7 天，过期自动清理

> **RP**(Retention Policy ) 决定数据保留时长与默认写入对象；数据到期后自动淘汰。
>
> **Shard Group**(分片组) 将某段时间的数据分配到固定的物理分片，提升并行和压缩。

## 14. 什么是 Continuous Query（CQ）？

- 类似物化视图，定时执行查询并写回数据库
- 常用于预聚合，如计算每分钟平均值存储，避免查询时实时计算

## 15. 如何设计 InfluxDB 的 Tag 和 Field?

- 作为 **tag** 的条件：用于频繁过滤/分组、值域较小且稳定（如 `host`, `region`, `model`）。
- 作为 **field** 的条件：数值型测量、连续变化大、无需作为索引过滤（如温度、电流、延迟）。
  **口诀：“常用于过滤且低基数 -> tag；高频数值测量 -> field”。**
- 经验法则：tag 用于 **WHERE/GROUP BY**，field 用于 **SELECT**

### 16. InfluxDB 1.x、2.x、3.x 的区别?

- **1.x**：TSM 引擎，InfluxQL
- **2.x**：引入 Flux、Task、Dashboard，API 改进
- **3.x**：基于 Apache Arrow，支持 SQL，存算分离，更云原生

## 17. InfluxDB 如何实现高可用?

- 1.x：企业版支持集群（开源版单机）
- 2.x：开源版依旧单机，HA 要依赖外部复制（如 Kapacitor + Telegraf）
- 3.x：云服务或存算分离架构，可水平扩展

## 18. 如何在 Kubernetes 中部署 InfluxDB?

- 使用官方 Helm Chart 或 StatefulSet
- 配合 PVC（持久化卷）存储数据
- 配置 HPA/资源限制保障稳定性
- 通常还会结合 Telegraf（采集器）和 Grafana（展示）

## 19. InfluxDB 在监控体系中一般怎么用?

- **采集**：Telegraf 收集 metrics/logs/traces
- **存储**：InfluxDB 存储时序数据
- **分析**：Flux/SQL/任务
- **可视化**：Grafana 或 Chronograf

## 20. 如果要在 InfluxDB 上做告警，怎么实现?

- 方式一：Kapacitor（Influx 官方流处理器）
- 方式二：InfluxDB Task（Flux 脚本执行逻辑）
- 方式三：Grafana Alerting（外部实现告警逻辑）

## 21. 写入性能优化的 8 条硬核建议

1. **批量写**（batch/每批几千条，按时间有序）。
2. **时间递增**写入，减少乱序与后期重写。
3. 控制 **tag 数量** 与 **tag 值基数**。
4. 合理 **shard duration**（写多查少可略长；查多写多用默认或更细）。
5. **并行连接**适度（避免过多 TCP 连接与小批量）。
6. 关闭不必要的 **正则匹配查询**（耗时）。
7. **硬件**：充足的 IOPS、SSD、独立数据盘。
8. 调参：`cache-max-memory-size`、`compact-full-write-cold-duration`、`max-values-per-tag` 等与版本相关的服务参数按压测调优。

## 22. 查询性能优化的 7 条经验

1. 优先用 **tag filter** 缩小扫描范围。
2. 合理 **time range**，避免全库全时段。
3. 使用 **aggregateWindow**（Flux）或 `GROUP BY time()`（InfluxQL）做分桶聚合。
4. 避免在 where 中对字段做函数变换（不走索引）。
5. 提前 **downsample** 与 **rollup**，避免直接扫原始高频数据。
6. 只取必要字段，减少 I/O。
7. 合理并发与分页，防止单次超大结果集。

## 23. Schema 设计清单（能背会用就很能打）

- 仅把**稳定、低基数、需过滤/分组**的维度放入 **tag**。
- 把连续测量值放入 **field**，避免把 ID、UUID、时间戳当 tag。
- 控制 measurement 数量；同类数据共用 measurement，用 tag 区分维度。
- 设定合适 **RP保留策略 + shard duration**；为分析面另建 **downsample** bucket/measurement。
- 写入按时间递增、批量化；尽量减少乱序、回写。
- 预估 series 数量，做 **基数预算** 与压测。
- 给查询上**时间范围**与**精确 tag 过滤**，避免模糊/正则滥用。

# 九、Doris

## 1、Doris基础

> doris的默认引擎是olap

# 第1章 Doris简介

## 1.1 Doris概述

- **背景**：由百度大数据部研发（原名百度Palo，2018年贡献给Apache社区后更名为Doris）
- **应用规模**：百度内部200+产品线使用，部署1000+台机器，单一业务最大可达上百TB
- **定位**：现代化MPP（Massively Parallel Processing，大规模并行处理）分析型数据库产品
- 特点：
  - 亚秒级响应时间
  - 支持实时数据分析
  - 分布式架构简洁，易于运维
  - 支持10PB以上超大数据集
  - 满足多种数据分析需求（固定历史报表、实时数据分析、交互式数据分析、探索式数据分析）

## 1.2 Doris架构

> Doris 的架构很简洁，只设 FE(Frontend)、BE(Backend)两种角色、两个进程，不依赖于外部组件，方便部署和运维，FE、BE 都可线性扩展

**架构组件**

1. FE（Frontend）
   - 功能：存储维护集群元数据、接收解析查询请求、规划查询计划、调度查询执行、返回查询结果
   - 三种角色：
     - **Leader和Follower**：实现元数据高可用，保证单节点宕机时元数据实时在线恢复
     - **Observer**：扩展查询节点，元数据备份，只参与读取不参与写入
2. BE（Backend）
   - 功能：负责物理数据的存储和计算
   - 执行FE生成的物理计划，分布式执行查询
   - 数据可靠性保证：支持多副本（2副本或3副本），副本数可动态调整
3. MySQL Client
   - 借助MySQL协议，支持任意MySQL的ODBC/JDBC以及MySQL客户端直接访问
4. Broker
   - 独立的无状态进程
   - 封装文件系统接口
   - 提供读取远端存储系统文件的能力（HDFS、S3、BOS等）

# 第2章 编译与安装

## 2.1 Docker环境安装

```
# 查看内核版本（要求高于3.10）
uname -r

# 更新yum包
sudo yum update -y

# 卸载旧版本
sudo yum remove docker docker-common docker-selinux docker-engine

# 安装依赖
sudo yum install -y yum-utils device-mapper-persistent-data lvm2

# 设置yum源
sudo yum-config-manager --add-repo
https://download.docker.com/linux/centos/docker-ce.repo

# 安装Docker
sudo yum install docker-ce -y

# 启动并设置开机自启
sudo systemctl start docker
sudo systemctl enable docker
```

## 2.2 Docker开发镜像编译

1. **下载源码**

```
wget https://dist.apache.org/repos/dist/dev/incubator/doris/0.15/0.15.0-
rc04/apache-doris-0.15.0-incubating-src.tar.gz
tar -zxvf apache-doris-0.15.0-incubating-src.tar.gz -C /opt/software
```

1. **下载Docker镜像**

```
docker pull apache/incubator-doris:build-env-for-0.15.0
```

1. **挂载本地目录运行镜像**

```
docker run -it \
-v /opt/software/.m2:/root/.m2 \
-v /opt/software/apache-doris-0.15.0-incubating-src/:/root/apache-doris-0.15.0-
incubating-src/ \
apache/incubator-doris:build-env-for-0.15.0
```

1. **编译Doris**

```
# 切换JDK版本
alternatives --set java java-1.8.0-openjdk.x86_64
alternatives --set javac java-1.8.0-openjdk.x86_64
export JAVA_HOME=/usr/lib/jvm/java-1.8.0

# 编译
sh build.sh --clean --be --fe --ui
```

## 2.3 安装要求

### 2.3.1 软硬件需求

**Linux操作系统要求**:

- CentOS 7.1及以上
- Ubuntu 16.04及以上

**软件需求**:

- Java 1.8及以上
- GCC 4.8.2及以上

**开发测试环境**:

```
模块CPU内存磁盘网络实例数量
Frontend 8核+8GBSSD/SATA 10GB+千兆网卡1
Backend 8核+16GBSSD/SATA 50GB+千兆网卡1-3
```

**生产环境**:

```
模块CPU内存磁盘网络实例数量
Frontend 16核+64GBSSD/SATA 100GB+万兆网卡1-5
Backend 16核+64GBSSD/SATA 100GB+万兆网卡10-100
```

### 2.3.2 默认端口

```
实例端口名称默认端口通讯方向说明
BE be_port 9060 FE→BE BE上thrift server端口
BE webserver_port 8040 BE↔FE BE上http server端口
BE heartbeat_service_port 9050 FE→BEBE心跳服务端口
BE brpc_port 8060 FE↔BE,BE↔BE BE间通信端口
FE http_port 8030 FE↔FE,用户↔FE FE的http端口
FE rpc_port 9020 BE→FE,FE↔FE FE的thrift端口
FE query_port 9030 用户↔FE FE的mysql端口
FE edit_log_port 9010 FE↔FE FE间bdbje通信端口
Br okerbroker_ipc_port 8000 FE→Broker,BE→Broker Broker的thrift端口
```

## 2.4 集群部署

### 2.4.1 部署规划

```
主机1: FE(LEADER) + BE + BROKER
主机2: FE(FOLLOWER) + BE + BROKER
主机3: FE(OBSERVER) + BE + BROKER
```

### 2.4.2 部署FE节点

1. **创建元数据目录**

```
mkdir /opt/module/apache-doris-0.15.0/doris-meta
```

1. **修改配置文件**

```
vim /opt/module/apache-doris-0.15.0/fe/conf/fe.conf

# 指定元数据路径
meta_dir = /opt/module/apache-doris-0.15.0/doris-meta
# 修改绑定IP
priority_networks = 192.168.8.101/24
```

1. **启动FE**

```
/opt/module/apache-doris-0.15.0/fe/bin/start_fe.sh --daemon
```

### 2.4.3 配置BE节点

1. **创建数据存放目录**

```
mkdir /opt/module/apache-doris-0.15.0/doris-storage1
mkdir /opt/module/apache-doris-0.15.0/doris-storage2
```

1. **修改配置文件**

```
vim /opt/module/apache-doris-0.15.0/be/conf/be.conf

# 指定数据存放路径
storage_root_path = /opt/module/apache-doris-0.15.0/doris-
storage1;/opt/module/apache-doris-0.15.0/doris-storage2
# 修改绑定IP
priority_networks = 192.168.8.101/24
```

#### 2.4.4 在FE中添加BE节点

```
-- 连接FE
mysql -h hadoop1 -P 9030 -uroot

-- 添加BE节点
ALTER SYSTEM ADD BACKEND "hadoop1:9050";
ALTER SYSTEM ADD BACKEND "hadoop2:9050";
ALTER SYSTEM ADD BACKEND "hadoop3:9050";

-- 查看BE状态
SHOW PROC '/backends';
```

#### 2.4.5 启动BE

```
/opt/module/apache-doris-0.15.0/be/bin/start_be.sh --daemon
```

#### 2.4.6 部署Broker（可选）

```
# 启动Broker
/opt/module/apache-doris-0.15.0/apache_hdfs_broker/bin/start_broker.sh --daemon

# 添加Broker
ALTER SYSTEM ADD BROKER broker_name "hadoop1:8000","hadoop2:8000","hadoop3:8000";

# 查看Broker状态
SHOW PROC "/brokers";
```

### 2.5 扩容和缩容

#### 2.5.1 FE扩容

```
-- 添加Follower
ALTER SYSTEM ADD FOLLOWER "hadoop2:9010";

-- 添加Observer
ALTER SYSTEM ADD OBSERVER "hadoop3:9010";

-- 删除FE节点
ALTER SYSTEM DROP FOLLOWER[OBSERVER] "fe_host:edit_log_port";
```

启动新增的FE节点：

```
# 第一次启动需要指定helper
/opt/module/apache-doris-0.15.0/fe/bin/start_fe.sh --helper hadoop1:9010 --daemon
```

#### 2.5.2 BE扩容和缩容

```
-- 添加BE
ALTER SYSTEM ADD BACKEND "be_host:be_heartbeat_service_port";

-- 推荐的删除方式（安全删除）
ALTER SYSTEM DECOMMISSION BACKEND "be_host:be_heartbeat_service_port";

-- 取消删除操作
CANCEL DECOMMISSION BACKEND "be_host:be_heartbeat_service_port";
```

# 第3章 数据表的创建

> 在 Doris 中，数据都以关系表（Table）的形式进行逻辑上的描述。

## 3.1 创建用户和数据库

```
-- 创建用户
mysql -h hadoop1 -P 9030 -uroot -p
CREATE USER 'test' IDENTIFIED BY 'test';

-- 创建数据库
CREATE DATABASE test_db;

-- 用户授权
GRANT ALL ON test_db TO test;
```

## 3.2 基本概念

#### 3.2.1 Row & Column

- **Row**：用户的一行数据
- Column：描述行数据中不同的字段
  - 默认数据模型：分为排序列和非排序列
  - 聚合模型：分为Key列（维度列）和Value列（指标列）

#### 3.2.2 Partition & Tablet

- Partition（分区）：
  - 数据划分的第一层
  - 通常按用户指定的分区列进行范围划分
  - 逻辑上最小的管理单元
  - 数据导入与删除可针对单个Partition进行
- Tablet（分片）：
  - 每个分区内数据按Hash方式分桶
  - 数据划分的最小逻辑单元
  - 数据移动、复制等操作的最小物理存储单元
  - Tablet之间数据无交集，独立存储

## 3.3 建表示例

### 3.3.1 建表语法

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [database.]table_name
(column_definition1[, column_definition2, ...]
[, index_definition1[, index_definition12,]])
[ENGINE = [olap|mysql|broker|hive]]
[key_desc]
[COMMENT "table comment"]
[partition_desc]
[distribution_desc]
[rollup_index]
[PROPERTIES ("key"="value", ...)]
[BROKER PROPERTIES ("key"="value", ...)];
```

### 3.3.2 字段类型

**数值类型**：

- TINYINT (1字节): -2^7+1 ~ 2^7-1
- SMALLINT (2字节): -2^15+1 ~ 2^15-1
- INT (4字节): -2^31+1 ~ 2^31-1
- BIGINT (8字节): -2^63+1 ~ 2^63-1
- LARGEINT (16字节): -2^127+1 ~ 2^127-1
- FLOAT (4字节): 支持科学计数法
- DOUBLE (12字节): 支持科学计数法
- DECIMAL[(precision, scale)] (16字节): 精确小数类型

**日期时间类型**：

- DATE (3字节): 0000-01-01 ~ 9999-12-31
- DATETIME (8字节): 0000-01-01 00:00:00 ~ 9999-12-31 23:59:59

**字符串类型**：

- CHAR[(length)]: 定长字符串，1-255
- VARCHAR[(length)]: 变长字符串，1-65533
- STRING: 变长字符串，最大2GB-4

**其他类型**：

- BOOLEAN: 0代表false，1代表true
- HLL: HyperLogLog列类型
- BITMAP: 整型集合类型

### 3.3.3 建表示例

**Range分区表**（通常为时间列，以方便管理新旧数据，不可添加范围重叠的分区）：

```
CREATE TABLE IF NOT EXISTS example_db.example_range_tbl
(
    `user_id` LARGEINT NOT NULL COMMENT "用户id",
```

```
    `date` DATE NOT NULL COMMENT "数据灌入日期时间",
    `timestamp` DATETIME NOT NULL COMMENT "数据灌入的时间戳",
    `city` VARCHAR(20) COMMENT "用户所在城市",
    `age` SMALLINT COMMENT "用户年龄",
    `sex` TINYINT COMMENT "用户性别",
    `last_visit_date` DATETIME REPLACE DEFAULT "1970-01-01 00:00:00" COMMENT "用户最后一次访问时间",
    `cost` BIGINT SUM DEFAULT "0" COMMENT "用户总消费",
    `max_dwell_time` INT MAX DEFAULT "0" COMMENT "用户最大停留时间",
    `min_dwell_time` INT MIN DEFAULT "99999" COMMENT "用户最小停留时间"
)
ENGINE=olap
AGGREGATE KEY(`user_id`, `date`, `timestamp`, `city`, `age`, `sex`)
PARTITION BY RANGE(`date`)
(
    PARTITION `p201701` VALUES LESS THAN ("2017-02-01"),
    PARTITION `p201702` VALUES LESS THAN ("2017-03-01"),
    PARTITION `p201703` VALUES LESS THAN ("2017-04-01")
)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 16
PROPERTIES
(
    "replication_num" = "3",
    "storage_medium" = "SSD",
    "storage_cooldown_time" = "2018-01-01 12:00:00"
);
```

**List分区表**（分区列支持各种数据类型，分区值为枚举值，不可添加范围重叠的分区）：

```
CREATE TABLE IF NOT EXISTS example_db.example_list_tbl
(
    -- 列定义同上
)
ENGINE=olap
AGGREGATE KEY(`user_id`, `date`, `timestamp`, `city`, `age`, `sex`)
PARTITION BY LIST(`city`)
(
    PARTITION `p_cn` VALUES IN ("Beijing", "Shanghai", "Hong Kong"),
    PARTITION `p_usa` VALUES IN ("New York", "San Francisco"),
    PARTITION `p_jp` VALUES IN ("Tokyo")
)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 16
PROPERTIES("replication_num" = "3");
```

## 3.4 数据划分

### 3.4.1 列定义建议

- Key列必须在所有Value列之前
- 尽量选择整型类型（计算和查找效率高）
- 不同长度的整型选择够用即可
- VARCHAR和STRING类型长度够用即可
- 所有列总字节长度不超过100KB

#### 3.4.2 分区与分桶

**分区（Partition）：**

- 支持Range和List两种划分方式
- 分区列必须为KEY列，不论分区列是什么类型，在写分区值时，都需要加**双引号**
- 分区数量理论上无上限
- 不使用分区时，系统自动生成一个全值范围的隐藏分区

**分桶（Bucket/Tablet）：**

- 仅支持Hash划分方式
- 分桶列可以是多列，必须为Key列
- 分桶数量理论上无上限
- 分桶列选择需在查询吞吐和查询并发之间权衡

#### 3.4.3 PROPERTIES重要参数

- **replication_num**：副本数量，默认3，建议保持奇数
- **storage_medium**：存储介质（SSD或HDD）
- **storage_cooldown_time**：数据从SSD迁移到HDD的时间

### 3.5 数据模型

#### 3.5.1 Aggregate模型（聚合模型）

**特点**：（没有设置 AggregationType 的称为 Key，设置了 AggregationType 的称为 Value）

- 列分为Key列（维度列）和Value列（指标列）
- Key列相同的行会聚合成一行
- Value列按指定的聚合方式聚合

**聚合方式**：

- SUM：求和
- REPLACE：替换
- REPLACE_IF_NOT_NULL：遇null值不更新
- MAX：保留最大值
- MIN：保留最小值

**聚合发生的三个阶段**：

1. 每批次数据导入的ETL阶段
2. BE进行数据Compaction阶段
3. 数据查询阶段

**示例**：

> 可以看到，用户 10000 只剩下了一行聚合后的数据。而其余用户的数据和原始数据保持一致。经过聚合，Doris 中最终只会存储聚合后的数据。换句话说，即明细数据会丢失，用户不能够再查询到聚合前的明细数据了

```
CREATE TABLE IF NOT EXISTS test_db.example_site_visit
(
    `user_id` LARGEINT NOT NULL COMMENT "用户id",
    `date` DATE NOT NULL COMMENT "数据灌入日期时间",
    `city` VARCHAR(20) COMMENT "用户所在城市",
    `age` SMALLINT COMMENT "用户年龄",
    `sex` TINYINT COMMENT "用户性别",
    `last_visit_date` DATETIME REPLACE DEFAULT "1970-01-01 00:00:00",
    `cost` BIGINT SUM DEFAULT "0",
    `max_dwell_time` INT MAX DEFAULT "0",
    `min_dwell_time` INT MIN DEFAULT "99999"
)
AGGREGATE KEY(`user_id`, `date`, `city`, `age`, `sex`)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 10;
```

#### 3.5.2 Uniq模型

**特点**：（保证Key的唯一性）

- 保证Key的唯一性（Primary Key约束）
- 本质上是聚合模型的特例
- 所有Value列都是REPLACE类型

**示例**：

```
CREATE TABLE IF NOT EXISTS test_db.user
(
    `user_id` LARGEINT NOT NULL COMMENT "用户id",
    `username` VARCHAR(50) NOT NULL COMMENT "用户昵称",
    `city` VARCHAR(20) COMMENT "用户所在城市",
    `age` SMALLINT COMMENT "用户年龄",
    `sex` TINYINT COMMENT "用户性别",
    `phone` LARGEINT COMMENT "用户电话",
    `address` VARCHAR(500) COMMENT "用户地址",
    `register_time` DATETIME COMMENT "用户注册时间"
)
UNIQUE KEY(`user_id`, `username`)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 10;
```

#### 3.5.3 Duplicate模型

**特点**：（在某些多维分析场景下，数据既没有主键，也没有聚合需求，数据完全按照导入文件中的数据进行存储）

- 数据完全按照导入文件存储，不做任何聚合
- 即使两行数据完全相同也都会保留
- DUPLICATE KEY仅用于指明数据排序列

**示例**：

```
CREATE TABLE IF NOT EXISTS test_db.example_log
(
    `timestamp` DATETIME NOT NULL COMMENT "日志时间",
    `type` INT NOT NULL COMMENT "日志类型",
    `error_code` INT COMMENT "错误码",
    `error_msg` VARCHAR(1024) COMMENT "错误详细信息",
    `op_id` BIGINT COMMENT "负责人id",
    `op_time` DATETIME COMMENT "处理时间"
)
DUPLICATE KEY(`timestamp`, `type`)
DISTRIBUTED BY HASH(`timestamp`) BUCKETS 10;
```

#### 3.5.4 数据模型选择建议

> 因为数据模型在建表时就已经确定，且**无法修改**。所以，选择一个合适的数据模型**非常重要**。

- **Aggregate模型**：适合固定模式的报表类查询，可通过预聚合减少扫描数据量，但对count(*)查询不友好
- **Uniq模型**：需要唯一主键约束的场景，无法利用ROLLUP等预聚合优势
- **Duplicate模型**：适合任意维度的Ad-hoc查询，不受聚合模型约束，可发挥列存优势

### 3.6 动态分区

#### 3.6.1 原理

- FE启动后台线程，根据用户指定规则自动创建或删除分区
- 实现表级别的分区生命周期管理（TTL）
- 仅支持Range分区

#### 3.6.2 使用方式

**建表时指定**：

```
CREATE TABLE student_dynamic_partition1
(
    id int,
    time date,
    name varchar(50),
    age int
)
DUPLICATE KEY(id,time)
PARTITION BY RANGE(time)()
DISTRIBUTED BY HASH(id) BUCKETS 10
PROPERTIES(
    "dynamic_partition.enable" = "true",
    "dynamic_partition.time_unit" = "DAY",
    "dynamic_partition.start" = "-7",
    "dynamic_partition.end" = "3",
    "dynamic_partition.prefix" = "p",
    "dynamic_partition.buckets" = "10",
    "replication_num" = "1"
);
```

**运行时修改**：

```
ALTER TABLE tbl1 SET
(
    "dynamic_partition.enable" = "true",
    "dynamic_partition.time_unit" = "DAY",
    "dynamic_partition.start" = "-7",
    "dynamic_partition.end" = "3"
);
```

#### 3.6.3 动态分区参数

**主要参数**：

- `dynamic_partition.enable`：是否开启，默认true
- `dynamic_partition.time_unit`：调度单位（HOUR/DAY/WEEK/MONTH）
- `dynamic_partition.start`：起始偏移（负数）
- `dynamic_partition.end`：结束偏移（正数）
- `dynamic_partition.prefix`：分区名前缀
- `dynamic_partition.buckets`：分桶数量

**历史分区参数**：

- `dynamic_partition.create_history_partition`：是否创建历史分区
- `dynamic_partition.history_partition_num`：历史分区数量
- `dynamic_partition.hot_partition_num`：热分区数量

> Doris 动态分区会根据 **start/end** 生成区间范围内的分区，但最终受 **create_history_partition** 和 **history_partition_num** 约束。如果只允许保留 1 个历史分区，那么只有距离今天最近的那一个（p20210519）会被保留，其余历史分区会被自动清理。

## 3.7 Rollup

#### 3.7.1 基本概念

- Base表：用户通过建表语句创建的原始表
- ROLLUP表：基于Base表产生，物理上独立存储
- 作用：获得更粗粒度的聚合数据（提高某些查询的查询效率）

#### 3.7.2 创建和管理ROLLUP

**创建ROLLUP**：

```
ALTER TABLE example_site_visit2 ADD ROLLUP
rollup_cost_userid(user_id,cost);
```

**查看ROLLUP**：

```
DESC example_site_visit2 ALL;
```

**查看是否命中ROLLUP**：

```
EXPLAIN SELECT user_id, sum(cost) FROM example_site_visit2 GROUP BY user_id;
```

### 3.7.3 前缀索引

**原理**：

- Doris数据存储在类似SSTable的有序数据结构中
- 将一行数据的前36个字节作为前缀索引
- 遇到VARCHAR类型时直接截断

**ROLLUP调整前缀索引**：

```
-- 创建调整列顺序的ROLLUP
ALTER TABLE table_name ADD ROLLUP rollup_name(age, user_id, message, ...);
```

## 3.8 物化视图

### 3.8.1 概念和优势

**概念**：

- 查询结果预先存储起来的特殊表
- 满足既能对明细数据任意维度分析，又能快速对固定维度查询

**优势**：

- 对重复使用相同子查询结果的查询性能大幅提升
- Doris自动维护数据一致性
- 查询时自动匹配最优物化视图

### 3.8.2 创建物化视图

```
-- 创建物化视图
CREATE MATERIALIZED VIEW store_amt AS
SELECT store_id, sum(sale_amt)
FROM sales_records
GROUP BY store_id;

-- 查看创建状态
SHOW ALTER TABLE MATERIALIZED VIEW FROM test_db;

-- 查看所有物化视图
DESC sales_records ALL;

-- 验证是否匹配物化视图
EXPLAIN SELECT store_id, sum(sale_amt) FROM sales_records GROUP BY store_id;

-- 删除物化视图
DROP MATERIALIZED VIEW 物化视图名 ON Base表名;
```

### 3.8.3 使用限制

- 聚合函数参数不支持表达式，仅支持单列
- 单表过多物化视图影响导入效率（建议不超过10个）
- 相同列不同聚合函数不能在同一物化视图中
- 对Unique Key模型只能改变列顺序，不能聚合

## 3.9 修改表

### 3.9.1 Rename操作

```
-- 修改表名
ALTER TABLE table1 RENAME table2;

-- 修改ROLLUP名
ALTER TABLE example_table RENAME ROLLUP rollup1 rollup2;

-- 修改分区名
ALTER TABLE example_table RENAME PARTITION p1 p2;
```

### 3.9.2 Partition操作

```
-- 增加分区
ALTER TABLE example_db.my_table
ADD PARTITION p1 VALUES LESS THAN ("2014-01-01");

-- 修改分区副本数
ALTER TABLE example_db.my_table
MODIFY PARTITION p1 SET("replication_num"="1");

-- 删除分区
ALTER TABLE example_db.my_table
DROP PARTITION p1;
```

### 3.9.3 Schema Change（表结构变更）

```
-- 增加列
ALTER TABLE table1 ADD COLUMN uv BIGINT SUM DEFAULT '0' AFTER pv;

-- 查看作业进度
SHOW ALTER TABLE COLUMN;

-- 取消作业
CANCEL ALTER TABLE ROLLUP FROM table1;
```

## 3.10 删除数据

### 3.10.1 条件删除（DELETE FROM）

```
DELETE FROM table_name [PARTITION partition_name]
WHERE column_name1 op { value | value_list }
[ AND column_name2 op { value | value_list } ...];

-- 示例
DELETE FROM student_kafka WHERE id=1;
```

**注意事项**：

- 只能针对Partition级别删除
- WHERE条件只能针对Key列

- 谓词之间只能用AND连接
- 是同步命令
- 不会立即释放磁盘空间

#### 3.10.2 删除分区（DROP PARTITION）

```
ALTER TABLE table_name DROP PARTITION partition_name;
```

- 轻量级操作
- 同步命令，立即生效
- 推荐的数据删除方式

# 第4章 数据导入和导出

## 4.1 数据导入

### 4.1.1 Broker Load

**适用场景**：

- 源数据在Broker可访问的存储系统中（如HDFS）
- 数据量在几十到百GB级别

**基本语法**：

```
LOAD LABEL db_name.label_name
(
    DATA INFILE ('file_path', ...)
    INTO TABLE tbl_name
    [PARTITION (p1, p2)]
    [COLUMNS TERMINATED BY separator]
    [(col1, ...)]
    [SET (k1=f1(xx), k2=f2(xx))]
    [WHERE predicate]
)
WITH BROKER broker_name
(
    "key1"="value1",
    ...
)
PROPERTIES
(
    "timeout" = "3600",
    "max_filter_ratio" = "0.1"
);
```

**示例**：

```
-- CSV文件导入
LOAD LABEL test_db.student_result
(
    DATA INFILE("hdfs://my_cluster/student.csv")
    INTO TABLE `student_result`
```

```
    COLUMNS TERMINATED BY ","
    FORMAT AS "csv"
    (id, name, age, score)
)
WITH BROKER broker_name
(
    "dfs.nameservices" = "my_cluster",
    "dfs.ha.namenodes.my_cluster" = "nn1,nn2,nn3",
    "dfs.namenode.rpc-address.my_cluster.nn1" = "hadoop1:8020",
    "dfs.namenode.rpc-address.my_cluster.nn2" = "hadoop2:8020",
    "dfs.namenode.rpc-address.my_cluster.nn3" = "hadoop3:8020",
    "dfs.client.failover.proxy.provider" =
"org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider"
)
PROPERTIES
(
    "timeout" = "3600"
);
```

**查看导入状态**：

```
SHOW LOAD ORDER BY createtime DESC LIMIT 1\G
```

**取消导入**：

```
CANCEL LOAD [FROM db_name] WHERE LABEL="load_label";
```

#### 4.1.2 Stream Load

**适用场景**：

- 导入本地文件或数据流
- 同步导入方式
- 支持CSV和JSON格式

**基本语法**：

```
curl --location-trusted -u user:passwd [-H "key:value"...] -T data.file -XPUT \
http://fe_host:http_port/api/{db}/{table}/_stream_load
```

**示例**：

```
curl --location-trusted -u root -H "label:123" \
-H"column_separator:," -T student.csv -X PUT \
http://hadoop1:8030/api/test_db/student_result/_stream_load
```

**重要参数**：

- `label`：导入任务标识
- `column_separator`：列分隔符，默认\t
- `max_filter_ratio`：最大容忍率
- `columns`：列映射和转换

- `where`：过滤条件
- `partition`：指定分区
- `two_phase_commit`：两阶段提交

#### 4.1.3 Routine Load

**适用场景**：

- 从Kafka持续导入数据
- 支持CSV和JSON格式
- 常驻任务，不间断读取

**基本语法**：

```
CREATE ROUTINE LOAD [db.]job_name ON tbl_name
[merge_type]
[load_properties]
[job_properties]
FROM KAFKA
(
    "kafka_broker_list"= "host1:port1,host2:port2",
    "kafka_topic" = "topic_name",
    "property.group.id"="consumer_group",
    "property.kafka_default_offsets" = "OFFSET_BEGINNING"
);
```

**示例**：

```
CREATE ROUTINE LOAD test_db.kafka_test ON student_kafka
COLUMNS TERMINATED BY ",",
COLUMNS(id, name, age)
PROPERTIES
(
    "desired_concurrent_number"="3",
    "strict_mode" = "false"
)
FROM KAFKA
(
    "kafka_broker_list"= "hadoop1:9092,hadoop2:9092,hadoop3:9092",
    "kafka_topic" = "test_doris1",
    "property.group.id"="test_doris_group",
    "property.kafka_default_offsets" = "OFFSET_BEGINNING",
    "property.enable.auto.commit"="false"
);
```

**管理命令**：

```
-- 查看状态
SHOW ROUTINE LOAD;

-- 暂停任务
PAUSE ROUTINE LOAD job_name;

-- 恢复任务
RESUME ROUTINE LOAD job_name;

-- 停止任务
STOP ROUTINE LOAD job_name;
```

#### 4.1.4 Binlog Load

**适用场景**：

- MySQL增量同步（CDC功能）
- 支持INSERT/UPDATE/DELETE
- 需要依赖Canal

**配置步骤**：

1. **MySQL配置**：

```
[mysqld]
log-bin = mysql-bin
binlog-format=ROW
binlog-do-db=test
gtid-mode=on
enforce-gtid-consistency=1
```

1. **创建同步作业**：

```
CREATE SYNC test_db.job1
(
    FROM test.tbl1 INTO binlog_test
)
FROM BINLOG
(
    "type" = "canal",
    "canal.server.ip" = "hadoop1",
    "canal.server.port" = "11111",
    "canal.destination" = "doris-load",
    "canal.username" = "canal",
    "canal.password" = "canal"
);
```

1. **管理命令**：

```
-- 查看状态
SHOW SYNC JOB;

-- 停止任务
STOP SYNC JOB job_name;

-- 暂停任务
PAUSE SYNC JOB job_name;

-- 恢复任务
RESUME SYNC JOB job_name;
```

#### 4.1.5 Insert Into

**基本语法**：

```
-- 从查询插入
INSERT INTO tbl SELECT ...;

-- 插入值
INSERT INTO tbl (col1, col2, ...) VALUES (1, 2, ...), (1,3, ...);

-- 带标签的插入
INSERT INTO tbl2 WITH LABEL label1 SELECT * FROM tbl3;
```

**查看最近一次插入结果**：

```
SHOW LAST INSERT\G
```

### 4.2 数据导出

#### 4.2.1 Export导出

**基本语法**：

```
EXPORT TABLE db1.tbl1
PARTITION (p1,p2)
[WHERE [expr]]
TO "hdfs://host/path/to/export/"
PROPERTIES
(
    "label" = "mylabel",
    "column_separator"=",",
    "columns" = "col1,col2",
    "timeout" = "3600"
)
WITH BROKER "broker_name"
(
    "username" = "user",
    "password" = "passwd"
);
```

**示例**：

```
EXPORT TABLE example_site_visit2
TO "hdfs://mycluster/doris-export"
PROPERTIES
(
    "label" = "mylabel",
    "column_separator"="|",
    "timeout" = "3600"
)
WITH BROKER "broker_name"
(
    "dfs.nameservices"="mycluster",
    "dfs.ha.namenodes.mycluster"="nn1,nn2,nn3",
    "dfs.namenode.rpc-address.mycluster.nn1"= "hadoop1:8020",
    "dfs.namenode.rpc-address.mycluster.nn2"= "hadoop2:8020",
    "dfs.namenode.rpc-address.mycluster.nn3"="hadoop3:8020",

    "dfs.client.failover.proxy.provider.mycluster"="org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider"
);
```

**查看导出状态**：

```
SHOW EXPORT;
```

#### 4.2.2 查询结果导出

**基本语法**：

```
query_stmt
INTO OUTFILE "file_path"
[FORMAT AS CSV|PARQUET]
[PROPERTIES (...)]
```

**示例**：

```
-- 导出到HDFS
SELECT * FROM example_site_visit
INTO OUTFILE "hdfs://hadoop1:8020/doris-out/result_"
FORMAT AS CSV
PROPERTIES
(
    "broker.name" = "broker_name",
    "column_separator" = ",",
    "line_delimiter" = "\n",
    "max_file_size" = "100MB"
);

-- 直接使用HDFS协议
SELECT * FROM example_site_visit
INTO OUTFILE "hdfs://doris-out/hdfs_"
FORMAT AS CSV
PROPERTIES
(
    "hdfs.fs.defaultFS" = "hdfs://hadoop1:8020",
```

```
    "hdfs.hdfs_user" = "atguigu",
    "column_separator" = ","
);
```

# 第5章 查询

## 5.1 查询设置

### 5.1.1 增大内存

```
-- 查看当前设置
SHOW VARIABLES LIKE "%mem_limit%";

-- 设置为8GB（单位：byte）
SET exec_mem_limit = 8589934592;

-- 全局生效
SET GLOBAL exec_mem_limit = 8589934592;
```

### 5.1.2 修改超时时间

```
-- 查看当前超时设置
SHOW VARIABLES LIKE "%query_timeout%";

-- 设置为60秒
SET query_timeout = 60;

-- 全局生效
SET GLOBAL query_timeout = 60;
```

### 5.1.3 查询高可用

**方式一：代码层重试**

- 应用层自行配置多个FE地址
- 发现连接失败自动重试其他连接

**方式二：JDBC Connector**

```
jdbc:mysql://[host1][:port1],[host2][:port2],[host3][:port3].../[database]
```

**方式三：ProxySQL**

- 配置ProxySQL作为代理层
- 实现自动负载均衡和故障转移

## 5.2 简单查询

```
-- 基本查询
SELECT * FROM example_site_visit LIMIT 3;
SELECT * FROM example_site_visit ORDER BY user_id;

-- Join查询
SELECT SUM(example_site_visit.cost)
FROM example_site_visit
JOIN example_site_visit2
WHERE example_site_visit.user_id = example_site_visit2.user_id;

-- 子查询
SELECT SUM(cost) FROM example_site_visit2
WHERE user_id IN (SELECT user_id FROM example_site_visit WHERE user_id > 10003);
```

## 5.3 Join查询

### 5.3.1 Broadcast Join

默认实现方式，将小表广播到大表所在节点：

```
-- 默认使用
EXPLAIN SELECT SUM(example_site_visit.cost)
FROM example_site_visit
JOIN example_site_visit2
WHERE example_site_visit.city = example_site_visit2.city;

-- 显式指定
EXPLAIN SELECT SUM(example_site_visit.cost)
FROM example_site_visit
JOIN [broadcast] example_site_visit2
WHERE example_site_visit.city = example_site_visit2.city;
```

### 5.3.2 Shuffle Join

> 即将小表和大表都按照 Join 的 key 进行 Hash，然后进行分布式的 Join。这个对内存的消耗就会分摊到集群的所有计算节点上

小表无法放入内存时使用：

```
SELECT SUM(example_site_visit.cost)
FROM example_site_visit
JOIN [shuffle] example_site_visit2
WHERE example_site_visit.city = example_site_visit2.city;
```

### 5.3.3 Colocation Join

> Colocation Join 是在 Doris0.9 版本引入的功能，旨在为 Join 查询提供本性优化，来减少数据在节点上的传输耗时，加速查询

**使用要求**：

- 分桶列类型和数量完全一致
- 桶数一致

- 副本数一致

**创建Colocation表**：

```
CREATE TABLE `tbl1` (
    `k1` date NOT NULL,
    `k2` int(11) NOT NULL,
    `v1` int(11) SUM NOT NULL
) ENGINE=OLAP
AGGREGATE KEY(`k1`, `k2`)
DISTRIBUTED BY HASH(`k2`) BUCKETS 8
PROPERTIES ("colocate_with" = "group1");

CREATE TABLE `tbl2` (
    `k1` datetime NOT NULL,
    `k2` int(11) NOT NULL,
    `v1` double SUM NOT NULL
) ENGINE=OLAP
AGGREGATE KEY(`k1`, `k2`)
DISTRIBUTED BY HASH(`k2`) BUCKETS 8
PROPERTIES ("colocate_with" = "group1");
```

**管理Colocation Group**：

```
-- 查看Group
SHOW PROC '/colocation_group';

-- 修改表的Group
ALTER TABLE tbl SET ("colocate_with" = "group2");

-- 删除Colocation属性
ALTER TABLE tbl SET ("colocate_with" = "");
```

### 5.3.4 Bucket Shuffle Join

**开启方式**：

```
SET enable_bucket_shuffle_join = true;
```

**原理**：

- 根据表的数据分布信息减少网络传输
- BE会根据数据分布将右表数据发送到对应的左表节点

**使用条件**：

- Join条件为等值条件
- 左表的分桶列为Join条件
- 左表分桶列类型与右表Join列类型一致

### 5.3.5 Runtime Filter

**概念**：

- 运行时动态生成过滤条件
- 减少扫描数据量和网络传输

**配置**：

```
-- 设置RuntimeFilter类型
SET runtime_filter_type="BLOOM_FILTER,IN,MIN_MAX";

-- 其他重要参数
SET runtime_filter_wait_time_ms=1000;   -- 等待时间
SET runtime_filters_max_num=10;         -- 最大数量
SET runtime_filter_max_in_num=1024;     -- IN谓词最大值数量
```

**查看效果**：

```
-- 开启profile
SET enable_profile=true;

-- 执行查询
SELECT t1 FROM test JOIN test2 WHERE test.t1 = test2.t2;

-- 查看profile
-- 访问 http://fe_host:8030/QueryProfile/
```

## 5.4 SQL函数

```
-- 查看所有函数
SHOW BUILTIN FUNCTIONS IN test_db;

-- 查看函数详情
SHOW FULL BUILTIN FUNCTIONS IN test_db LIKE 'year';
```

# 第6章 集成其他系统

## 6.1 Spark读写Doris

### 6.1.1 SQL方式

```
sparkSession.sql(
  """
    |CREATE TEMPORARY VIEW spark_doris
    |USING doris
    |OPTIONS(
    |  "table.identifier"="test_db.table1",
    |  "fenodes"="hadoop1:8030",
    |  "user"="test",
    |  "password"="test"
    |);
  """.stripMargin)
```

```
// 读取数据
sparkSession.sql("select * from spark_doris").show()

// 写入数据
sparkSession.sql("insert into spark_doris values(99,99,'haha',5)")
```

#### 6.1.2 DataFrame方式

```
// 读取数据
val dorisSparkDF = sparkSession.read.format("doris")
  .option("doris.table.identifier", "test_db.table1")
  .option("doris.fenodes", "hadoop1:8030")
  .option("user", "test")
  .option("password", "test")
  .load()

// 写入数据
mockDataDF.write.format("doris")
  .option("doris.table.identifier", "test_db.table1")
  .option("doris.fenodes", "hadoop1:8030")
  .option("user", "test")
  .option("password", "test")
  .save()
```

#### 6.1.3 RDD方式

```
import org.apache.doris.spark._

val dorisSparkRDD = sc.dorisRDD(
  tableIdentifier = Some("test_db.table1"),
  cfg = Some(Map(
    "doris.fenodes" -> "hadoop1:8030",
    "doris.request.auth.user" -> "test",
    "doris.request.auth.password" -> "test"
  ))
)
```

### 6.2 Flink Doris Connector

#### 6.2.1 SQL方式

```
tableEnv.executeSql("CREATE TABLE flink_doris (\n" +
    "  siteid INT,\n" +
    "  citycode SMALLINT,\n" +
    "  username STRING,\n" +
    "  pv BIGINT\n" +
    ") WITH (\n" +
    "  'connector' = 'doris',\n" +
    "  'fenodes' = 'hadoop1:8030',\n" +
    "  'table.identifier' = 'test_db.table1',\n" +
    "  'username' = 'test',\n" +
    "  'password' = 'test'\n" +
    ")");
```

```
// 读取数据
tableEnv.executeSql("select * from flink_doris").print();

// 写入数据
tableEnv.executeSql("insert into flink_doris values(22,'wuyanzu',3)");
```

#### 6.2.2 DataStream方式

**Source**：

```
Properties properties = new Properties();
properties.put("fenodes","hadoop1:8030");
properties.put("username","test");
properties.put("password","test");
properties.put("table.identifier","test_db.table1");

env.addSource(new DorisSourceFunction(
    new DorisStreamOptions(properties),
    new SimpleListDeserializationSchema()
)).print();
```

**Sink**：

```
source.addSink(
    DorisSink.sink(
        fields,
        types,
        DorisReadOptions.builder().build(),
        DorisExecutionOptions.builder()
            .setBatchSize(3)
            .setBatchIntervalMs(0L)
            .setMaxRetries(3)
            .build(),
        DorisOptions.builder()
            .setFenodes("hadoop1:8030")
            .setTableIdentifier("test_db.table1")
            .setUsername("test")
            .setPassword("test")
            .build()
    ));
```

### 6.3 DataX doriswriter

**配置示例**：

```
{
  "job": {
    "content": [
      {
        "reader": {
          "name": "mysqlreader",
          "parameter": {
            "column": ["id", "ts", "vc"],
            "connection": [{
```

```
                "jdbcUrl": ["jdbc:mysql://hadoop1:3306/test"],
                "table": ["sensor"]
            }],
            "username": "root",
            "password": "000000"
        }
    },
    "writer": {
      "name": "doriswriter",
      "parameter": {
        "feLoadUrl": ["hadoop1:8030", "hadoop2:8030", "hadoop3:8030"],
        "beLoadUrl": ["hadoop1:8040", "hadoop2:8040", "hadoop3:8040"],
        "jdbcUrl": "jdbc:mysql://hadoop1:9030/",
        "database": "test_db",
        "table": "sensor",
        "column": ["id", "ts", "vc"],
        "username": "test",
        "password": "test",
        "maxBatchRows": 500000,
        "maxBatchByteSize": 104857600,
        "labelPrefix": "my_prefix"
        }
    }
  }
 ]
}
}
```

## 6.4 ODBC外部表

### 6.4.1 创建ODBC外表

**方式一：直接创建**

```
CREATE EXTERNAL TABLE `baseall_oracle` (
    `k1` decimal(9, 3) NOT NULL,
    `k2` char(10) NOT NULL,
    `k3` datetime NOT NULL,
    `k5` varchar(20) NOT NULL,
    `k6` double NOT NULL
) ENGINE=ODBC
PROPERTIES (
    "host" = "192.168.0.1",
    "port" = "8086",
    "user" = "test",
    "password" = "test",
    "database" = "test",
    "table" = "baseall",
    "driver" = "Oracle 19 ODBC driver",
    "odbc_type" = "oracle"
);
```

**方式二：通过Resource创建（推荐）**

```
-- 创建Resource
```

```
CREATE EXTERNAL RESOURCE `oracle_odbc`
PROPERTIES (
    "type" = "odbc_catalog",
    "host" = "192.168.0.1",
    "port" = "8086",
    "user" = "test",
    "password" = "test",
    "database" = "test",
    "odbc_type" = "oracle",
    "driver" = "Oracle 19 ODBC driver"
);

-- 基于Resource创建外表
CREATE EXTERNAL TABLE `baseall_oracle` (
    `k1` decimal(9, 3) NOT NULL,
    `k2` char(10) NOT NULL,
    `k3` datetime NOT NULL,
    `k5` varchar(20) NOT NULL,
    `k6` double NOT NULL
) ENGINE=ODBC
PROPERTIES (
    "odbc_catalog_resource" = "oracle_odbc",
    "database" = "test",
    "table" = "baseall"
);
```

## 6.5 Doris On ES

（1）创建 ES 外表后，FE 会请求建表指定的主机，获取所有节点的 HTTP 端口信息以及 index 的 shard 分布信息等，如果请求失败会顺序遍历 host 列表直至成功或完全失败

（2）查询时会根据 FE 得到的一些节点信息和 index 的元数据信息，生成查询计划并发给对应的 BE 节点

（3）BE 节点会根据就近原则即优先请求本地部署的 ES 节点，BE 通过 HTTP Scroll 方式流式的从 ES index 的每个分片中并发的从 source 或 docvalue 中获取数据

（4）Doris 计算完结果后，返回给用户

### 6.5.1 创建ES外表

```
CREATE EXTERNAL TABLE `es_test` (
    `k1` bigint(20),
    `k2` datetime,
    `k3` varchar(20),
    `k4` varchar(100),
    `k5` float
) ENGINE=ELASTICSEARCH
PROPERTIES (
    "hosts" = "http://hadoop1:9200,http://hadoop2:9200,http://hadoop3:9200",
    "index" = "test",
    "type" = "doc",
    "user" = "",
    "password" = ""
);
```

#### 6.5.2 重要参数

- `enable_docvalue_scan`：启用列式扫描优化查询速度
- `enable_keyword_sniff`：探测keyword类型字段
- `nodes_discovery`：开启节点自动发现
- `http_ssl_enabled`：启用HTTPS访问模式

#### 6.5.3 查询用法

**基本查询**：

```
SELECT * FROM es_table WHERE k1 > 1000 AND k3 ='term' OR k4 LIKE 'fu*z_';
```

**使用esquery函数**：

```
-- match_phrase查询
SELECT * FROM es_table WHERE esquery(k4, '{
    "match_phrase": {
        "k4": "doris on es"
    }
}');

-- bool查询
SELECT * FROM es_table WHERE esquery(k4, '{
    "bool": {
        "must": [
            {"terms": {"k1": [11, 12]}},
            {"terms": {"k2": [100]}}
        ]
    }
}');
```

# 第7章 监控和报警

## 7.1 Prometheus配置

1. **配置prometheus.yml**：

```
scrape_configs:
  - job_name: 'prometheus_doris'
    static_configs:
    - targets: ['hadoop1:8030','hadoop2:8030','hadoop3:8030']
      labels:
        group: fe
    - targets: ['hadoop1:8040','hadoop2:8040','hadoop3:8040']
      labels:
        group: be
```

1. **启动Prometheus**：

```
nohup ./prometheus --web.listen-address="0.0.0.0:8181" &
```

## 7.2 Grafana配置

1. **配置defaults.ini**：

```
http_addr = hadoop1
http_port = 8182
```

1. **启动Grafana**：

```
nohup /opt/module/grafana-7.5.2/bin/grafana-server &
```

1. **配置数据源和Dashboard**

- 访问 http://hadoop1:8182
- 添加Prometheus数据源
- 导入Doris监控模板

# 第8章 优化

## 8.1 查看QueryProfile

### 8.1.1 使用方式

```
-- 开启profile
SET enable_profile=true;

-- 执行查询
SELECT t1 FROM test JOIN test2 WHERE test.t1 = test2.t2;

-- 查看profile
-- 访问 http://hadoop1:8030/QueryProfile/
```

### 8.1.2 重要指标说明

**Fragment级别**：

- AverageThreadTokens：执行使用的线程数
- PeakMemoryUsage：内存使用峰值
- RowsProduced：处理的行数

**OLAP_SCAN_NODE**：

- BytesRead：读取的数据量
- RowsRead：从存储引擎返回的行数
- RowsReturned：返回给上层节点的行数
- TabletCount：涉及的Tablet数量

**EXCHANGE_NODE**：

- BytesReceived：网络接收的数据量
- DataArrivalWaitTime：等待数据的总时间
- DeserializeRowBatchTimer：反序列化耗时

## 8.2 Join Reorder

**开启方式：**

```
SET enable_cost_based_join_reorder=true;
```

**优化原理：**

1. 大表与小表尽量先Join
2. 有条件的Join表往前放
3. Hash Join优先级高于Nest Loop Join

## 8.3 Join优化原则

1. 选择同类型或简单类型的列进行Join
2. 尽量选择Key列进行Join
3. 大表之间的Join尽量使用Co-location
4. 合理使用Runtime Filter
5. 保证左表为大表，右表为小表

## 8.4 导入导出性能优化

### 8.4.1 FE配置优化

```
# 导入超时配置
max_load_timeout_second=259200
min_load_timeout_second=1

# 等待队列配置
desired_max_waiting_jobs=100

# 每个数据库最大运行导入数
max_running_txn_num_per_db=100

# Broker Load配置
min_bytes_per_broker_scanner=67108864
max_bytes_per_broker_scanner=3221225472
max_broker_concurrency=10

# Stream Load配置
stream_load_default_timeout_second=600
```

### 8.4.2 BE配置优化

```
# 写入速度限制
push_write_mbytes_per_sec=10

# 内存表大小
write_buffer_size=104857600

# RPC超时配置
tablet_writer_rpc_timeout_sec=600
streaming_load_rpc_max_alive_time_sec=600

# 导入内存限制
load_process_max_memory_limit_bytes=107374182400
load_process_max_memory_limit_percent=80
```

### 8.5 Bitmap索引

```
-- 创建Bitmap索引
CREATE INDEX table_bitmap ON table1 (siteid) USING BITMAP
COMMENT 'table1_bitmap_index';

-- 查看索引
SHOW INDEX FROM test_db.table1;

-- 删除索引
DROP INDEX IF EXISTS table_bitmap ON test_db.table1;
```

### 8.6 BloomFilter索引

```
-- 建表时指定
CREATE TABLE sale_detail_bloom (
    -- 列定义
) PROPERTIES (
    "bloom_filter_columns"="saler_id,category_id"
);

-- 修改BloomFilter索引
ALTER TABLE sale_detail_bloom
SET ("bloom_filter_columns" = "customer_id,sku_id");

-- 删除BloomFilter索引
ALTER TABLE sale_detail_bloom
SET ("bloom_filter_columns" = "");
```

## 8.7 合理设置分桶分区数

**建议原则**：

1. Tablet总数 = 分区数 × 分桶数
2. Tablet数量略多于集群磁盘数量
3. 单个Tablet数据量建议1G-10G
4. 数据量与数量原则冲突时，优先考虑数据量原则

**参考示例：**

- 500MB：4-8个分片
- 5GB：8-16个分片
- 50GB：32个分片
- 500GB：分区，每分区16-32个分片
- 5TB：分区，每分区16-32个分片

# 第9章 数据备份及恢复

## 9.1 备份

### 9.1.1 创建远端仓库

```
CREATE REPOSITORY `hdfs_ods_dw_backup`
WITH BROKER `broker_name`
ON LOCATION "hdfs://hadoop1:8020/tmp/doris_backup"
PROPERTIES (
    "username" = "",
    "password" = ""
);
```

### 9.1.2 执行备份

```
BACKUP SNAPSHOT test_db.backup1
TO hdfs_ods_dw_backup
ON (table1);
```

### 9.1.3 查看备份

```
-- 查看备份任务
SHOW BACKUP [FROM db_name];

-- 查看远端仓库镜像
SHOW SNAPSHOT ON hdfs_ods_dw_backup;

-- 查看特定备份详情
SHOW SNAPSHOT ON hdfs_ods_dw_backup
WHERE SNAPSHOT = "backup1" AND TIMESTAMP = "2021-05-05-15-34-26";
```

## 9.2 恢复

### 9.2.1 执行恢复

```
RESTORE SNAPSHOT test_db.backup1
FROM `hdfs_ods_dw_backup`
ON (
    table1 AS table_restore
)
PROPERTIES (
    "backup_timestamp"="2022-04-01-16-45-19"
);
```

#### 9.2.2 管理恢复任务

```
-- 查看恢复任务
SHOW RESTORE [FROM db_name];

-- 取消恢复任务
CANCEL RESTORE FROM db_name;
```

### 9.3 删除远端仓库

```
DROP REPOSITORY `hdfs_ods_dw_backup`;
```

# 第10章 1.0新特性

## 10.1 向量化执行引擎

**开启方式**：

```
SET enable_vectorized_engine = true;
SET batch_size = 4096;
```

**优势**：

- 列式内存布局，提高Cache亲和度
- 批量类型判断，减少虚函数开销
- 支持编译器内联和SIMD优化

**注意事项**：

- NULL值会导致性能劣化
- Float/Double计算可能有精度误差
- 不支持UDF和UDAF
- String/Text类型最大支持1MB

## 10.2 Hive外表

**创建Hive外表**：

```
CREATE TABLE `t_hive` (
    `k1` int NOT NULL,
    `k2` char(10) NOT NULL,
    `k3` datetime NOT NULL,
    `k5` varchar(20) NOT NULL,
    `k6` double NOT NULL
) ENGINE=HIVE
PROPERTIES (
    'hive.metastore.uris' = 'thrift://hadoop1:9083',
    'database' = 'test',
    'table' = 'test11'
);
```

### 10.3 Lateral View语法

```
-- 开启功能
SET enable_lateral_view=true;

-- explode_split展开字符串
SELECT k1, e1 FROM test3
LATERAL VIEW explode_split(k2, ',') tmp1 AS e1
ORDER BY k1, e1;

-- explode_json_array展开JSON数组
SELECT k1, e1 FROM test3
LATERAL VIEW explode_json_array_int('[1,2,3]') tmp1 AS e1
ORDER BY k1, e1;
```

### 10.4 mysqldump导出

```
# 导出表数据
mysqldump -h127.0.0.1 -P9030 -uroot --no-tablespaces \
--databases test_db --tables user > dump1.sql

# 导出表结构
mysqldump -h127.0.0.1 -P9030 -uroot --no-tablespaces \
--databases test_db --tables user --no-data > dump2.sql

# 导出整个数据库
mysqldump -h127.0.0.1 -P9030 -uroot --no-tablespaces \
--databases test_db

# 导入数据
source /opt/module/doris-1.0.0/dump1.sql
```

# 2、Doris相关面试题

## 1.介绍一下Doris

Doris 是一款基于 MPP（大规模并行处理）架构的高性能、实时分析型数据库。它以其高效、简单、统一的特点而闻名，能够在亚秒级响应时间内处理海量数据的查询。Doris 不仅支持高并发的点查询场景，还能支持高吞吐的复杂分析场景，适用于多种数据分析和查询需求。

主要应用场景：

1. 实时数据分析：Doris 支持高效的实时数据分析，可以处理快速增长的数据，并且支持低延迟的查询响应，非常适合大规模实时数据分析的场景，比如广告监控、点击流分析、数据挖掘等。
2. 数据仓库：它常用于构建大规模的数据仓库，支持 OLAP（在线分析处理）查询，能够高效处理海量的数据。
3. 报表与可视化分析：通过 Doris 提供的高效查询性能，许多 BI 工具（如 Tableau、Power BI）可以直接连接并进行实时的数据查询分析，帮助企业做决策。
4. 用户行为日志分析：由于其高效的聚合和查询能力，Doris 也适用于日志数据分析，可以实时生成和展示用户行为或系统监控数据。

## 2.Doris基本架构？主要组成部分作用？

Doris 采用 MySQL 协议，高度兼容 MySQL 语法，支持标准 SQL，用户可以通过各类客户端工具来访问 Doris，并支持与 BI 工具的无缝对接。

Doris 存算一体架构精简易于维护，如上图所示，只有两类进程：

**Frontend（FE）**：主要负责用户请求的接入、查询解析规划、元数据的管理、节点管理相关工作。

FE是 Doris 集群的控制节点，负责处理查询请求的调度、路由、解析和计划生成等任务。前端通常包含多个节点，以实现高可用和负载均衡。

主要作用：

- 查询调度和解析：前端接收用户的 SQL 查询请求，解析并生成查询计划。它将查询任务分解并路由到后端节点。
- 查询优化：根据查询语句的不同，前端会选择最优的执行计划，以提高查询效率。
- 元数据管理：前端还负责管理和存储 Doris 的元数据，包括数据库、表、列和分区等结构信息。

**Backend（BE）**：主要负责数据存储、查询计划的执行。数据会被切分成分片，在 BE 中多副本存储。

BE是 Doris 集群的执行节点，负责实际的数据存储、查询执行和计算。后端节点是集群的核心，通常由多个节点组成，通过水平扩展来应对大数据量和高并发的需求。

主要作用：

- 数据存储与管理：后端节点负责数据的持久化存储。Doris 使用列式存储格式，数据被分为多个分区（Partions），每个分区包含多个数据块（Tablet）。每个 Tablet 会存储一个数据表的部分数据。
- 查询执行：当前端将查询任务分发到后端时，后端节点会根据查询计划进行数据读取、计算和聚合。后端节点会并行执行查询操作，充分利用分布式计算的优势。
- 数据分布与容错：Doris 通过数据分片（Sharding）机制将数据均匀分布到多个后端节点，保证数据的高可用性和容错能力。当某个节点出现故障时，其他节点可以接管任务，确保系统的稳定性。

在生产环境中可以部署多个 FE 节点做容灾备份，每个 FE 中都会维护全量的元数据副本。

FE 分为三种角色：

| 角色 | 功能 |
| --- | --- |
| Master | FE Master 节点负责元数据的读写，在 Master 元数据发生变更后，会通过 BDB JE 协议同步给 Follower 或 Observer 节点。 |
| Follower | Follower 节点负责读取元数据，在 Master 节点发生故障时，Follower 节点可以被选取作为新的 Master 节点。 |
| Observer | Observer 节点负责读取元数据，主要为了增加集群的查询并发行。不参加集群选主。 |

FE 与 BE 进程都是可以横向扩展的，单集群可以支持到数百台机器，数十 PB 的存储容量。FE 与 BE 进程通过一致性协议来保证服务的高可用和数据的高可靠。存算一体架构高度集成，大幅降低了分布式系统的运维成本。

# 3.Doris的分区、分片、分桶?

## 1. Doris 的 分区（Partition）

- **概念**：分区是逻辑层面的划分，通常用于**大表**，把数据按一定规则（比如日期、地域、ID 范围）切分成多个子表。
- **目的**：
  - 提高查询性能（只扫所需分区，减少扫描量，类似"分区裁剪"）。
  - 提高管理灵活性（可以单独导入/删除某个分区的数据）。
- **方式**：
  - **Range 分区**：按范围划分（如 `20210101-20210131`）。
  - **List 分区**：按枚举值划分（如 `region in ('bj', 'sh', 'gz')`）。
  - **Dynamic Partition（动态分区）**：Doris 支持自动创建/删除分区（常用于按天、按月）。

👉 类似于 **数据库表的分区表**。

## 2. Doris 的 分片（Tablet / Shard）

- **概念**：分片是物理层面的划分，Doris 的底层存储是 **Tablet**。
  - 每个 **分区** 会被拆分为若干个 **分片（Tablet）**。
  - Tablet 是 Doris 数据存储和副本管理的最小单位。
- **副本**：
  - 每个分片通常有 **3 个副本**（配置可改），分布在不同的 BE 节点上，用于高可用。
- **目的**：
  - 实现 **数据分布在多节点**，充分利用集群资源。
  - 保证高可用（某个副本挂掉还能从其他副本恢复）。

👉 可以理解为 Doris 的 **HDFS Block** 或 **ES Shard**。

## 3. Doris 的 分桶（Bucket，哈希分桶 Bucketing）

- **概念**：在分区内部，数据会根据 **哈希分桶列** 进行划分，分成多个 **Bucket**，每个 Bucket 对应一个 Tablet。
- **作用**：
  - 决定数据在集群节点之间的分布，保证负载均衡。
  - 提高查询性能（例如 Join 时，保证相同 key 的数据在同一个分桶内，减少 shuffle）。
- **配置**：
  - `DISTRIBUTED BY HASH(col) BUCKETS N`
  - 其中 `col` 是哈希分桶列，`N` 是桶数。
- **举例**：
  - 如果按 `user_id` 做 HASH 分桶，桶数为 32，则相同 `user_id` 的数据必然落在同一个 Bucket（Tablet）里。

👉 类似于 **Hive 的分桶表**。

### 4. 三者关系图（层级）

```
表（Table）
 └── 分区（Partition）    ← 按时间、地域等逻辑划分
        └── 分桶（Bucket）← 哈希分桶，决定数据分布
               └── 分片（Tablet）← 底层物理存储，副本单位
```

### 5. 举个例子

比如有一个用户日志表：

```
CREATE TABLE user_log (
    user_id BIGINT,
    event_time DATETIME,
    region STRING,
    action STRING
)
PARTITION BY RANGE (event_time) (    -- 分区：按日期
    PARTITION p20210901 VALUES [('2021-09-01'), ('2021-09-02')),
    PARTITION p20210902 VALUES [('2021-09-02'), ('2021-09-03'))
)
DISTRIBUTED BY HASH(user_id) BUCKETS 16; -- 分桶：按 user_id 做哈希分桶
```

- **分区**：每天一个分区，比如 `p20210901`、`p20210902`。
- **分桶**：在每个分区内部，数据根据 `user_id` 哈希分成 16 个桶。
- **分片**：每个桶对应一个 Tablet，Tablet 有多个副本分布在不同 BE 节点。

## 4.说一下Doris的Broker?

Doris的资源调度组件，负责管理集群中的BE节点资源。它根据系统的负载情况，动态调整BE节点的数量，实现集群的扩容和缩容。

Broker扩容：Broker负责资源的调度和负载均衡。当需要增加BE节点时，Broker会自动将新的BE节点纳入资源池中，并根据负载情况分配任务。

Broker缩容：Broker负责资源的调度和管理。当需要减少Broker节点时，应确保集群中的BE节点资源得到合理利用，避免出现资源浪费的情况。

## 5.Doris 支持哪些数据导入方式？各自适用场景是什么?

Doris 支持多种数据导入方式，主要包括以下几种。具体如下：

**Stream Load**

- 适用场景：实时数据导入，小批量、高频导入场景；适用于需快速反映业务变化的在线服务，如实时监控、在线交易。
- 特点：低延时、高频次，适用于实时数据分析。
- 使用技巧：控制数据量，设置合适内存缓存和并发数，避免给集群造成过大压力。

**Broker Load**

- 适用场景：大批量数据导入，可并行导入多个超大文件，支持 HDFS/S3 等分布式存储；适用于离线数据分析、批量数据处理，如每天定时导入前一天业务数据。

- 特点：支持大批量导入，利用 Broker 节点集成外部存储系统。
- 使用技巧：调优 Broker 节点配置，如并行度、缓存大小等提升导入性能。

**Routine Load**

- 适用场景：自动化和持续的数据流导入，可从 Kafka 等流式数据平台消费数据；适用于日志数据导入、IoT 设备数据采集等需持续流式处理数据的应用。
- 特点：自动化、持续数据流导入，支持流式数据平台。
- 使用技巧：合理分配 Kafka 分区，设置适当消费速率，避免数据丢失或重复。

**Insert Load**

- 适用场景：表间数据复制、按需数据导入，小批量数据导入或表数据复制。
- 特点：简单直接，类似传统 INSERT 语句，但性能相对较低。
- 使用技巧：避免大数据量直接插入，可分批次或采用其他导入方式提升效率。

**Spark Load**

- 适用场景：与 Apache Spark 集成，适用于大规模数据处理和导入；用于数据预处理、大数据分析、复杂计算后数据导入。
- 特点：结合 Spark 计算能力处理大规模数据。
- 使用技巧：合理利用 Spark 并行计算能力，优化 Spark 作业参数配置

## 6.Doris 中怎么实现数据的自动分区和手动分区？

1、手动分区

用户手动创建分区（如建表时指定或通过 `ALTER` 语句增加）。

在 `CREATE TABLE` 语句中，使用 `PARTITION` 语法手动定义每个分区的名称和范围。

2、自动分区

数据写入时，系统根据需要自动创建相应的分区，使用时注意脏数据生成过多的分区。

通过 `AUTO PARTITION` 语法支持自动创建分区，常用于基于时间字段的 RANGE 分区。

3、自动分区 VS 手动分区

| 方式 | 适用场景 | 特点 | 优势 | 适用数据类型 |
| --- | --- | --- | --- | --- |
| **自动分区** | **时间序列数据（如日志、订单）** | 按时间字段自动创建分区 | 维护简单，适用于流式数据导入 | `DATE`、`DATETIME` |
| **手动分区** | **固定类别、地区、时间范围数据** | 需手动定义每个分区 | 完全可控，适用于分区数较少的情况 | `DATE`、`STRING` |

## 7.Doris的Rollup表？

Rollup 表是基于基表某些列组合预先聚合和存储数据的辅助表，主要用于提升查询性能，尤其针对含聚合计算与复杂维度过滤条件的查询。

Rollup 表在查询优化中的作用：

- 减少计算开销：预先计算和存储特定维度汇总数据，减少查询实时计算，加快查询响应速度。
- 提高查询响应速度：基于不同维度创建多个 Rollup 表，供查询优化器选最优 Rollup 表加速查询。

- 减少 I/O 操作：因 Rollup 表存储聚合数据量通常小于基表，可减小数据扫描量，降低 I/O 开销。

以下是一些典型的应用场景：

- 高频聚合查询：在涉及大量数据的聚合查询中（如按日期、地区、产品等维度聚合数据），Rollup 表能够提前计算好聚合结果，从而避免每次查询时重新计算，提升查询效率。例如，电商平台上的订单数据需要按日期和地区汇总销售额，使用 Rollup 表可以减少实时查询时的计算负担。
- 报表和统计分析：对于报表系统，Rollup 表可以显著加速生成报表的过程，避免重复计算汇总数据。例如，财务报表系统经常需要对不同维度的销售数据进行汇总，使用 Rollup 表可以大大提高响应速度。
- 低延迟查询：对于需要低延迟响应的查询，Rollup 表能够提前完成数据的汇总，查询时直接返回结果而不需要再进行计算。适用于实时监控、实时风控等场景。
- 数据仓库中的多维分析：在数据仓库环境中，Rollup 表常用于 OLAP（联机分析处理）查询优化，例如销售数据的多维分析，Rollup 表可以为查询提供快速的聚合结果。

## 5.Doris如何支持多维分析?

Doris 是一款面向分析型负载的分布式 SQL 数据库，其OLAP（在线分析处理）能力是支持多维分析的关键。Doris 主要依赖列式存储、动态索引、物化视图和复杂查询优化机制，使其能够高效处理和查询海量数据，从而实现高性能的多维分析。

列式存储（Columnar Storage）

- 优化点：Doris 使用列式存储格式，将相同列的数据存放在一起，方便按列读取和聚合，大幅提升查询效率。
- 优势：在多维分析中，经常需要对特定维度进行聚合、过滤，列式存储可以减少 I/O 操作，因为只需要读取相关列的数据。

Rollup 表（物化视图）

- 优化点：通过 Rollup 表，Doris 可以预先根据常用的维度和聚合方式创建预计算的表，避免在查询时重复计算。
- 优势：在执行多维分析时，Doris 会智能选择最优的 Rollup 表，直接返回结果，极大地提高了查询性能。

Bitmap 索引和 Bitmap 类型

- 优化点：Bitmap 索引可以快速定位某些维度上的数据，Bitmap 类型则适合做集合运算（如去重计数、用户画像）。
- 优势：在用户行为分析、去重统计等场景中，Bitmap 索引可以让查询速度提升数倍。

查询优化器（Query Optimizer）

- 优化点：Doris 的查询优化器能智能选择索引、使用谓词下推、进行列裁剪、自动选择最优的 Rollup 表。
- 优势：通过优化器，Doris 可以自动调整查询计划，针对多维分析自动选择最优执行路径。

## 6.Doris四种数据模型?

| 模型 | 是否聚合 | 是否去重 | 更新方式 | 典型场景 |
| --- | --- | --- | --- | --- |
| Aggregate聚合模型 | ✅ 聚合 | 按 Key 聚合 | 不支持更新 | 报表、统计 |
| Unique唯一模型 | ❌ 不聚合 | ✅ 覆盖 | 导入时覆盖 | 用户表、订单表 |

| 模型 | 是否聚合 | 是否去重 | 更新方式 | 典型场景 |
| --- | --- | --- | --- | --- |
| Duplicate明细模型 | ❌ 不聚合 | ❌ 保留 | 不更新 | 日志、明细存储 |
| Primary Key主键模型 | ❌ 不聚合 | ✅ 最新值 | 实时更新 | IoT、监控 |

## 7.Doris物化视图怎么加速查询的?

> 导入时预计算 + 查询时自动改写：通过在导入阶段提前算好常用聚合/维度，查询时直接用结果表替代大表扫描，从而实现 **秒级查询**。

通过预计算和存储查询结果，物化视图能够显著提升查询性能，尤其是对于复杂查询、聚合操作、以及多维分析场景中的查询。

具体步骤如下：（透明加速、实时更新、多视图选择、空间换时间）

1. 确定热点查询模式：识别频繁查询的模式或热点查询。
2. 设计物化视图：根据常见查询模式设计合适的物化视图。
3. 创建物化视图：使用 SQL 语法创建物化视图。
4. 自动优化查询：Doris 自动优化器根据查询条件决定是否使用物化视图，从而加速查询。

## 8.Doris数据高可用？哪些机制保障数据安全?

Doris 是一个现代化的 MPP 列存储数据库，主要用于 OLAP 分析场景。为了实现数据高可用性，Doris 设计了一系列机制：

- 数据分片和副本：Doris 通过数据分片和多个副本的方式确保数据高可用。每个数据分片有多个副本，分布在不同节点上，确保即使某个节点故障，数据仍可从其他节点读取。
- 节点自动故障转移：Doris 提供智能的节点故障检测机制，节点不可用时，系统会自动将任务和数据转移到其他正常节点，确保服务连续性。
- 元数据冗余存储：Doris 将元数据存储在多个节点上，保证元数据的高可用性和一致性。
- 数据持久化：所有数据变更（如插入、更新、删除）都会记录到日志文件中，并定期持久化到物理存储，确保系统故障后可以通过日志恢复数据。
- 多种数据恢复机制：Doris 提供快照恢复、基于时间点恢复等多种数据恢复机制，确保在任何情况下都能迅速恢复数据。

## 9.说一下Compaction机制?

> 定期小合并，周期大合并

Doris（底层基于 **Palo/StarRocks 类似的存储引擎**）采用 **LSM-Tree 思想**：

- 数据导入时不会直接改原文件，而是以 **增量文件（Rowset/Segment 文件）** 的形式追加写入。
- 这样写入快，但随着时间推移：
  - 同一个 Key 可能在多个文件中有不同版本（比如更新、删除标记）。
  - 文件数量越来越多，查询时需要扫描大量文件，性能下降。
- **Compaction 就是把这些小文件合并成大文件，并清理旧版本数据。**

👉 一句话：**Compaction 负责"文件合并 + 数据清理"，让查询更高效。**

**Doris 有两类主要的 Compaction：**

（1）**Cumulative Compaction（累计合并）**

- **目标**：把**最近导入的小文件**合并成中等大小的文件。
- **触发条件**：当某个 Tablet 的小文件数量超过阈值。
- **特点**：
  - 合并的数据量相对较小，执行频率高。
  - 保证写入的数据逐渐被整合，避免 Tablet 中文件数量过多。

👉 类似于 “小扫除”，快速合并新写入的小文件。

（2）**Base Compaction（基线合并）**

- **目标**：把 Tablet 中累积的中等文件，和历史的 Base 文件一起合并成一个新的 Base 文件。
- **触发条件**：当累计合并后的数据量比较大，或者版本数量太多时。
- **特点**：
  - 数据量大，执行频率低。
  - 会清理掉过期版本、已删除的数据（例如 `DELETE` 标记的数据）。

👉 类似于 “大扫除”，彻底整理出一个干净的基准版本。

## 1）Compaction 的流程

以一个 Tablet（分片）为例：

1. **数据写入**：每次导入会生成一个新的 Rowset（小文件）。
2. **累计合并（Cumulative Compaction）**：
   - 定期把多个小 Rowset 合并成一个中等大小 Rowset。
   - 这样 Tablet 的 Rowset 数量不会无限增加。
3. **基线合并（Base Compaction）**：
   - 当累计 Rowset 太多，或者版本太复杂时，把 Base Rowset 和累计 Rowset 合并为一个新的 Base Rowset。
   - 同时清理过期数据。

👉 合并后的文件替代旧文件，Tablet 始终保持较少的文件数量。

## 2）Compaction调度策略

Doris 的 **BE（Backend）节点**上有 Compaction Scheduler（调度器），负责：

- 哪些 Tablet 优先做合并。
- 控制并发度，避免 Compaction 占用过多 CPU/IO 影响查询。

策略：

- 优先合并小文件多的 Tablet（Cumulative）。
- 当 Tablet 版本太复杂时，触发 Base Compaction。

## 10.Doris常见的查询优化方法?

1、设计合理表结构，避免不必要宽列冗余字段，合适数据类型，按需查询列
2、索引优化，常见索引有覆盖索引和二级索引，对文本或数字类型的列应用倒排索引或 Bitmap 索引
3、分区分桶设计，按照查询模式对表划分，减少全表扫描；将数据分散到多个桶中，能够更好地进行并行处理
4、收集和定期更新统计信息，帮助查询优化器生成更优的执行计划
5、SQL优化，避免复杂子查询和嵌套查询
6、参数调优，内存和缓存调整，并发度调整
7、CBO成本优化器，CBO 会自动决定查询的执行顺序、选择合适的 Join 算法、决定是否使用索引等，减少无谓的计算和数据扫描。
8、向量化执行引擎，通过批处理来提高查询执行效率，充分利用cpu和内存资源，能并行执行多个子任务
9、物化视图
10、分区裁剪，doris自动根据where条件进行分区裁剪，只扫描需要的分区
11、Join优化：小表广播和大表分布式Join；调整Join顺序和合理的Join算法
12、查询换成和冷数据淘汰

## 11.Doris多租户?

多租户支持主要通过 资源池（Resource Pool）和资源标签（Resource Tag）来实现。这些机制确保了租户之间的资源隔离和管理，避免资源竞争对系统性能产生负面影响。

资源池（Resource Pool）：

- 作用：资源池是实现多租户隔离的核心。Doris 将计算资源（如 CPU、内存等）划分为多个资源池，每个租户会被分配一个独立的资源池。这样做的目的是防止单个租户消耗过多资源，影响其他租户的正常使用。
- 优势：通过将资源池分配给不同租户，能够避免资源争用和性能下降。

资源标签（Resource Tag）：

- 作用：资源标签用于标识计算节点，并通过标签与资源池相结合来调度查询请求。每个计算节点可以被分配一个或多个标签，确保特定租户的查询请求被调度到特定的计算节点。
- 优势：通过资源标签的灵活配置，可以更精细地控制租户的计算任务分配，实现更好的资源隔离。

配额管理（Quota Management）：

- 作用：Doris 允许为每个资源池设置配额，控制每个租户可使用的资源量。例如，可以设置最大并发查询数、内存使用限制等。
- 优势：配额管理帮助避免某个租户过度消耗资源，确保公平性，同时提升系统的整体可用性。

## 12.Doris负载均衡?

负载均衡策略配置：Doris 提供了 Leaderbalance 和 Diskbalance 等参数来实现负载均衡。 Leaderbalance 主要用于平衡 MetaServiceMaster 节点的负载，而 Diskbalance 关注节点的存储均衡。 这些参数需要根据实际场景进行微调，以确保集群的负载均衡。

分区和分桶设计：

- 分区（Partition）： Doris 支持通过 RANGE 和 HASH 分区策略，以业务逻辑为基础，创建细粒度的分区。 合理的分区设计有助于避免热点数据问题，提升查询和写入效率。
- 分桶（Bucket）： 对大型表格进行分桶策略，可以进一步优化数据的分布，避免数据倾斜。 尤其在写多读多的场景下，合理配置分桶策略可以明显提升查询和写入效率。

监控工具：Doris 内置了丰富的监控指标，如节点 CPU 使用率、内存使用率、I/O 负载等。 可以通过 Doris 自带的 Web UI 界面、与 Prometheus 集成以及 Grafana 展示这些监控指标。 结合这些工具，可以实时捕捉节点的负载状态，发生异常时及时告警并采取应急措施。

数据迁移功能： 如果发现有节点的负载过高，可以通过数据迁移功能，将部分数据从高负载节点迁移到低负载节点，达到负载均衡。 同时，Doris 也可以自动执行数据再分布，不需要太多的人工干预，但需要在集群健康检查和定期维护时加以监督。

## 13.Doris如何使用外表来查询不同数据源?

Doris中可以通过创建外部表来查询其他数据源的数据。外部表允许在不将数据导入 Doris 的情况下，直接对外部数据源进行查询和分析。

Doris 支持多种外部数据源，包括 MySQL、PostgreSQL、Elasticsearch、Apache Hive、Apache Iceberg 和 Apache Hudi 等。

创建外部表：在 Doris 中创建外部表，映射到外部数据源中的表。外部表的结构应与目标表的结构一致。<

```
CREATE EXTERNAL TABLE `external_table` (
  `id` INT,
  `name` VARCHAR(255),
  `age` INT
)
 ENGINE=JDBC
 COMMENT "MySQL External Table"
 PROPERTIES (
   "resource" = "mysql_resource",
   "table" = "your_table"
 );
```

`resource` 是外部资源的名称，`table` 是目标数据库中的表名。

## 14.说一下Doris支持的分布式查询引擎?

Doris支持的分布式查询引擎主要包括：

B.E (Backend Engine)：

- 作用：B.E（Backend Engine）是 Doris 的计算引擎，负责实际的查询执行。它处理数据的读取、计算、聚合等操作。数据存储和计算都发生在 B.E 节点上，这些节点是 Doris 集群中的工作节点。（一句话解释：处理数据存储和计算的计算引擎，负责执行实际查询操作）
- 职责：B.E 负责执行由 F.E（Frontend Engine）传递来的查询任务，并将计算结果返回给 F.E。

F.E (Frontend Engine)：

- 作用：F.E（Frontend Engine）是 Doris 的前端引擎，主要负责查询解析、优化和调度。
- 查询解析和优化：F.E 接收用户的查询请求，解析 SQL 语句，并将其转换为计算任务。接着，它会调用查询优化器（如 CBO）生成最优的执行计划。
- 协调任务：F.E 会将查询任务分发给 B.E 节点进行实际的数据计算和执行。

Doris 通过以下几种机制来实现高效的分布式查询：

- 分布式查询引擎：支持并行计算，利用集群的分布式资源加速查询。
- 查询优化器：通过 CBO 和规则优化器，自动选择最佳执行计划。

- 向量化执行引擎：提高计算效率，减少资源消耗。
- 列式存储：减少 I/O 操作，适合分析型查询。
- 物化视图：加速查询，避免重复计算。
- 分区裁剪：减少扫描无关数据，提高效率。
- 查询缓存：避免重复计算，提升响应速度。
- 负载均衡与故障转移：确保查询高可用和高性能。

# 15.Doris的Join?

## 1）Join执行方式?

在执行引擎层，Doris 主要实现了以下几类 Join 算法：

### （1）Hash Join（最常用）

- **原理**：
  - 将 **小表构建哈希表**，
  - 再对大表扫描时，用哈希表快速匹配。
- **特点**：
  - 适合大表 + 小表的场景。
  - 小表会被完整加载到内存（如果太大可能 OOM）。
- **示例**：

```
SELECT o.order_id, c.name
FROM orders o
JOIN customers c ON o.cust_id = c.cust_id;
```

👉 一般会把 `customers` 当作小表加载到内存。

### （2）Broadcast Join（广播 Join）

- **原理**：
  - 将 **小表广播到所有 BE 节点**，每个节点本地做 Hash Join。
- **适用场景**：
  - 小表（几 MB ~ GB 级别）+ 大表。
  - 避免 Shuffle 大表数据。
- **优点**：减少网络传输，大表不用打散。

### （3）Shuffle Join（分布式 Join）

- **原理**：
  - 大表 + 大表 Join 时，把两张表按 Join Key 做 **哈希分区（Shuffle）**，
  - 保证相同 Key 的数据被分配到同一个 BE 节点，再做本地 Join。
- **适用场景**：
  - 大表对大表。
- **缺点**：需要 Shuffle，网络 IO 成本大。

**（4）Colocate Join（共置 Join）**

- **原理**：
  - 如果两张表在建表时，使用了相同的 **分桶列** 和 **分桶数**，
  - 那么相同 Key 的数据天然落在相同的 BE 节点。
  - 这样 Join 可以直接在本地完成，无需 Shuffle。
- **优点**：性能最佳，几乎等于本地 Join。
- **要求**：
  - 两张表的分布方式完全一致（colocate group）。

👉 这是 Doris 在分布式 Join 上的一个 **杀手锏优化**。

**（5）Nested Loop Join（嵌套循环 Join，少用）**

- **原理**：一张表的行逐条和另一张表匹配。
- **缺点**：效率低，OLAP 场景几乎不用。

## 2） Doris 的 Join 优化策略

1. **自动选择执行方式**
   - Doris 的优化器会根据表大小、分布情况，自动选择 Broadcast、Shuffle 或 Colocate Join。
   - 你可以通过 `EXPLAIN` 看实际执行计划。
2. **Join Reorder（Join 重排序）**
   - 多表 Join 时，优化器会调整执行顺序，把小表尽量提前 Join。
3. **Predicate Pushdown（谓词下推）**
   - Join 前会把过滤条件尽量下推到扫描阶段，减少数据量。
4. **Colocate 优化**
   - 如果能命中 Colocate Join，则不做 Shuffle，大幅加速。

## 3）总结一句话

- Doris 的 Join = **Hash Join 为主**，结合 **Broadcast / Shuffle / Colocate** 三种分布式策略。
- 小表 + 大表 ⇒ **Broadcast Join**
- 大表 + 大表 ⇒ **Shuffle Join**
- 分布相同的大表 Join ⇒ **Colocate Join（最快）**
- 多表 Join ⇒ 优化器自动重排序。

# 16.Doris查询机制?

1、并行扫描：多个扫描任务可以并行地读取不同的数据块，从而加快数据读取速度
2、并行计算：针对某个查询任务，Doris 会将其分解为多个子任务，这些子任务在多个计算节点（BE）上并行执行，实现计算资源的最大化利用
3、并行聚合：在执行聚合操作时，Doris 会将数据集分成多个部分，并在多个节点上进行并行聚合，最后合并结果。

**提高并行度：**

1）增加计算节点
2）调整并行度配置
3）优化数据分布

## 17.Doris的优化导入?

| 优化方向 | 关键措施 | 适用场景 / 效果 |
| --- | --- | --- |
| **批量导入** | - **Stream Load**：实时小批量 - **Broker Load**：离线大规模导入 - **Routine Load**：实时 Kafka 消费 | 根据数据源与实时性选择合适方式，提高写入效率 |
| **数据排序与表模型** | - 选择合理的 `sort key` - **Aggregate 表**：预聚合，按聚合键排序 - **Unique 表**：精确去重，按主键排序 | 减少导入冲突，提高查询性能 |
| **参数调优** | - 增大 `spark.doris.batch.size`、`spark.doris.batch.max.size` - 合理分配内存 | 降低网络与 I/O 开销，加快大批量导入 |
| **资源隔离** | - 使用 K8s/容器分配独立配额 - 专用 BE 节点处理导入 | 避免导入与查询抢占资源，保障导入稳定性 |
| **压缩与编码** | - 压缩算法：Snappy、Zlib - 文件格式：Parquet、ORC | 降低传输与存储开销，提高读写速度 |

## 18.Doris分区裁剪?

分区裁剪（Partition Pruning）是一种优化技术，用于减少查询过程中需要扫描的数据量，从而提高查询性能。它通过分析查询条件，智能地过滤掉与查询无关的分区，只访问相关的分区数据。这种技术特别适用于大规模数据集和复杂的 OLAP 查询场景。

Doris 支持按指定列对表进行分区（Partition），常见的分区方式包括：

- 范围分区（Range Partition）：按某个字段的值范围划分分区，例如按日期字段分区。
- 列表分区（List Partition）：按某个字段的离散值划分分区，例如按地区字段分区。

分区的设计使得数据可以按逻辑分组存储，便于管理和查询。

当用户执行查询时，Doris 的查询优化器会根据查询条件（如 `WHERE` 子句）判断哪些分区与查询相关，并跳过无关分区。这种优化能够显著减少扫描的数据量，降低 I/O 开销，从而提升查询性能。

## 19.说一下 Doris 中查询计划生成和优化的步骤?

```
SQL
 |
 ▼
解析 (Parse) → 语义分析 (Analyze)
 |
 ▼
逻辑计划 (Logical Plan)
 |        └─ 谓词下推、投影裁剪、子查询改写、Join 重排、MV 改写
 ▼
```

```
物理计划 (Physical Plan)
 |        └─ Join 算法选择、分区裁剪、数据分发策略
 ▼
执行计划 (Execution Plan)
 |
 ▼
BE 节点并行执行
```

## 20.Doris水平拓展方式?

增加计算节点：通过增加计算节点来提升查询处理能力。Doris 会将新的计算任务分配到新加入的节点上，提高查询吞吐量。

增加存储节点：通过增加存储节点来扩展数据存储能力。新的存储节点会存储数据副本，增强系统的容错能力，同时增加存储空间。

增加副本：通过增加数据的副本数量，可以提升数据的容错能力和并发查询能力。副本会自动分配到不同节点，保证高可用性。

## 21.Doris如何处理分布式事务?

多版本并发控制 (MVCC)：Doris 确实使用了 MVCC 来实现事务的并发控制。在 MVCC 模型下，数据会有多个版本，读操作会读取最新的已提交数据，不会读取到未提交的修改，保证了事务的隔离性。这种方式确保了数据的一致性和并发性，尤其在读写操作频繁的场景中，能有效防止脏读、不可重复读等问题。

分布式共识协议 (Raft)：Doris 使用 Raft协议来保证数据的一致性。所有的写操作会先通过 Raft 协议进行同步，确保副本之间的一致性。在数据写入时，Raft 协议会确保大多数副本成功写入之后，事务才会被认为提交成功。Raft 协议的优势是能够在分布式环境下提供强一致性，避免单点故障对数据一致性的影响。

## 22.Doris多副本机制?

Doris的多副本机制通过数据副本（replica）确保高可用性和容错性。

Doris中的每个表都会有多个副本，每个副本存储相同的数据。这些副本分布在不同的节点上，以保证系统的高可用性。

工作机制：

- 副本分配：数据会根据分区规则分布到多个节点上，每个分区会有多个副本。副本数量可以在创建表时指定，一般推荐使用3个副本，以平衡性能和可靠性。
- 数据同步：主副本负责处理写请求（如插入、更新、删除），而从副本会异步地同步主副本的数据。通过 Raft协议，Doris确保副本之间的数据一致性。Raft协议保证写入操作在多数副本上成功提交后才算完成。
- 副本一致性<：Doris使用Raft协议来管理副本的一致性。每次写操作都会先在主副本上执行，并由Raft协议确保所有从副本同步数据。如果主副本失效，系统会自动选举一个新的主副本，确保数据持续可用。
- 容错机制：如果某个副本发生故障或不可用，Doris能够自动将查询请求转发到其他可用副本，确保系统高可用。同时，Raft协议会自动进行副本恢复，确保副本的数量和一致性。

## 23.Doris压缩算法?

| 压缩算法 | 压缩率 | 压缩/解压速度 | 适用场景 | Doris 默认 |
| --- | --- | --- | --- | --- |
| **LZ4** | 中等 | **最快**（解压极快） | 实时导入、查询性能优先的 OLAP 场景 | ✅ |
| **Snappy** | 较低 | 快（略慢于 LZ4） | 查询延迟敏感，存储空间不是瓶颈 | ❌ |
| **Zlib** | **最高** | 慢（CPU 开销大） | 存储成本敏感、大数据量但低频访问 | ❌ |

# 十、ELK

# 1、ELK基础

# 2、ELK相关面试题1

## 1.FileBeat如何收集容器日志?

Filebeat 是一个轻量级的日志采集器，能够有效地收集容器日志。以下是使用 Filebeat 收集容器日志的几种常见方法：

### 1. 使用 Docker 日志驱动

Docker 提供了多种日志驱动，其中 `json-file` 是默认选项。Filebeat 可以直接从 Docker 容器的日志文件中读取日志。

### 步骤：

1. **安装 Filebeat**：确保 Filebeat 已在你的主机上安装。
2. **配置 Filebeat**：编辑 Filebeat 的配置文件（通常是 `filebeat.yml`），添加以下输入配置：

```
filebeat.inputs:
  - type: container
    paths:
      - "/var/lib/docker/containers/*/*.log"
    json.keys_under_root: true
    json.add_error_key: true
```

   这里的配置会读取 Docker 容器的 JSON 日志文件，并将其解析为 JSON 格式。

3. **启动 Filebeat**：运行 Filebeat，将日志发送到 Elasticsearch 或 Logstash。

### 2. 使用 Kubernetes

如果你在 Kubernetes 环境中运行容器，可以使用 `kubernetes` 输入类型来收集日志。

**步骤：**

1. **安装 Filebeat**：可以通过 Helm Chart 或直接部署 YAML 文件的方式安装 Filebeat。
2. **配置 Filebeat**：在 Filebeat 的配置文件或 Helm Chart 中，使用以下输入配置：

```
filebeat.autodiscover:
  providers:
    - type: kubernetes
      hints.enabled: true
      templates:
        - condition:
            contains:
              kubernetes.namespace: "default"
          config:
            - type: container
              paths:
                - /var/log/containers/*.log
              processors:
                - add_kubernetes_metadata: {}
```

   这里的配置会自动发现运行在 Kubernetes 集群中的容器，并收集对应的日志。

3. **启动 Filebeat**：将配置应用到 Kubernetes 集群中，Filebeat 会开始收集日志。

## 3. 使用 Docker Compose

如果使用 Docker Compose 进行容器编排，可以在 Compose 文件中配置 Filebeat 服务。

**步骤：**

1. **创建** `docker-compose.yml`：在你的 Compose 文件中添加 Filebeat 服务。

```
version: '3.8'
services:
  filebeat:
    image: docker.elastic.co/beats/filebeat:7.10.0
    volumes:
      - /var/lib/docker/containers:/var/lib/docker/containers:ro
      - /var/run/docker.sock:/var/run/docker.sock:ro
      - ./filebeat.yml:/usr/share/filebeat/filebeat.yml
    command: ["-e", "-strict.perms=false"]
```

2. **配置 Filebeat**：确保 `filebeat.yml` 配置文件包含适当的输入设置，如前面提到的配置。
3. **启动服务**：运行 `docker-compose up` 启动 Filebeat。

## 4. 与其他工具结合

- **与 Logstash 配合**：可以将 Filebeat 配置为将日志发送到 Logstash，进一步处理和过滤日志。
- **与 Elasticsearch 结合**：Filebeat 可以直接将日志发送到 Elasticsearch，便于后续分析和可视化。

### 总结

通过上述方式，Filebeat 可以有效地收集 Docker 和 Kubernetes 环境中的容器日志。根据你的具体架构和需求，选择合适的方法进行配置和部署。同时，确保监控 Filebeat 的运行状态，以保证日志的高效采集。

## 2、如何提升FileBeat的性能?

提高 Filebeat 的性能可以通过多个方面进行优化。以下是一些有效的方法和最佳实践：

### 1. 配置并发

- **增加 `harvester` 数量**：Filebeat 可以为每个文件启动一个 harvester。通过合理配置 `max_file_size` 和 `scan_frequency`，确保对大文件的处理不会造成延迟。

```
filebeat:
  inputs:
    - type: log
      paths:
        - /var/log/*.log
      harvester:
        max_bytes: 1048576  # 每个 harvester 最多处理的字节数
```

### 2. 批量发送

- **使用批量输出**：Filebeat 支持批量发送数据，可以通过设置 `bulk_max_size` 来提高发送效率。

```
output.elasticsearch:
  hosts: ["localhost:9200"]
  bulk_max_size: 2048  # 每次批量发送的最大文档数
```

### 3. 调整内存使用

- **调整内存限制**：Filebeat 在处理大量日志时可能会消耗较多内存。可以通过调整系统的内存限制和 Filebeat 的配置来优化性能。

### 4. 使用多实例

- **横向扩展**：在大型环境中，可以运行多个 Filebeat 实例，将负载分散到不同的实例上。这可以通过 Docker 或 Kubernetes 等容器化技术实现。

### 5. 选择合适的输入类型

- **优先使用 `filestream` 输入**：在 Filebeat 7.0 及以上版本，推荐使用 `filestream` 输入类型，它比老旧的 `log` 输入类型更高效。

```
filebeat.inputs:
  - type: filestream
    paths:
      - /var/log/*.log
```

### 6. 减少不必要的处理

- **使用轻量级的处理器**：尽量避免复杂的处理，如 `grok` 或 `json` 解析等。如果不需要，可以省略这些步骤，直接发送原始日志。
- **条件过滤**：如果有条件地发出事件，可以使用条件语句，从而减少不必要的数据处理。

### 7. 优化输出配置

- **使用适当的输出插件**：根据需求选择最适合的输出插件。例如，使用 `elasticsearch` 时，可以配置连接池参数。

### 8. 监控与调优

- **使用监控工具**：利用 Elastic Stack 的监控工具，监测 Filebeat 的性能指标，如日志处理速度、延迟等，及时发现瓶颈。

### 9. 配置注册表

- **调整注册表设置**：配置注册表的路径和大小，以确保 Filebeat 能够在重启后快速恢复状态。

```
filebeat:
  registry:
    path: /var/lib/filebeat/registry
    clean_inactive: 72h  # 清理不活跃的注册表条目
```

### 10. 优化文件扫描频率

- **调整文件扫描频率**：通过配置 `scan_frequency` 调整文件扫描的频率，确保 Filebeat 不会过于频繁地检查文件。

### 11. 使用队列

- **引入消息队列**：在高流量环境中，可以使用消息队列（如 Kafka 或 Redis）作为中间层，帮助平衡负载。

## 总结

通过以上优化措施，可以显著提升 Filebeat 的性能。在实际应用中，根据具体的使用场景和需求，灵活调整配置和架构是提高性能的关键。同时，持续监测 Filebeat 的运行状态，以确保其高效稳定地处理日志数据。

分类: 运维面试题 / ELK面试题

## 3、描述ES的写入过程？包括refresh、commit、flush和merge操作

### 1）写入请求的基本流程

当你向 Elasticsearch 发送一个 **写请求（index、update、delete）** 时，大致过程如下：

1. **路由**
   - 请求先到协调节点（coordinating node）。
   - 根据文档 `_id` 通过一致性 hash 算法计算出对应的 **主分片（primary shard）**。
   - 请求转发到该主分片所在的节点。

2. **写入内存缓冲区（In-Memory Buffer）**
    - 文档不会立刻写入磁盘，而是先写到 **内存 buffer**。
    - buffer 中的数据是不可搜索的。
3. **写入 translog**
    - 同时写一份操作日志到 **translog（事务日志）**，保证即使节点宕机，也能从日志恢复。
4. **副本同步**
    - 主分片写入成功后，会将操作同步到副本分片（replica shard）。
    - 主副分片都写入成功后，才算一次写操作完成并返回。

### 2）四个关键动作总结

| 操作 | 触发时机 / 方式 | 作用 | 是否持久化 | 是否可搜索 |
| --- | --- | --- | --- | --- |
| **refresh** | 默认 1s，或手动 `_refresh` | 将内存缓冲区写入新的 segment，打开供搜索 | ❌ 否 | ✅ 是 |
| **commit** | flush 时触发 | 把 segment 元信息和 translog checkpoint 落盘 | ✅ 是 | ✅ 是 |
| **flush** | 手动 `_flush`，或 translog 超限/定时 | 清空缓冲区，强制 commit，重置 translog | ✅ 是 | ✅ 是 |
| **merge** | 后台异步 | 合并小 segment → 大 segment，清理已删除文档 | ✅ 是 | ✅ 是 |

### 3）一句话理解

- **refresh**：数据可搜索，但不保证持久化。
- **commit**：数据落盘，保证崩溃恢复。
- **flush**：强制 commit + 清空 translog。
- **merge**：优化 segment，提升搜索性能，清理删除数据。

## 4、ES的核心概念有哪些?

- **Index（索引）**：相当于关系型数据库的Database，是文档的容器
- **Document（文档）**：存储的基本单位，以JSON格式表示
- **Type（类型）**：7.x版本后已废弃，之前用于在索引中对文档进行逻辑分类
- **Field（字段）**：文档中的属性
- **Mapping（映射）**：定义文档和字段的存储和索引方式
- **Shard（分片）**：索引的物理存储单元，分为主分片和副本分片
- **Node（节点）**：Elasticsearch实例
- **Cluster（集群）**：一组节点的集合

## 5.如何设计高可用ES集群?

高可用集群设计要点:

- **节点配置**:至少3个Master节点(防止脑裂),多个Data节点,可选配置Coordinating节点
- **分片策略**:主分片数量根据数据量确定,每个主分片至少1个副本
- **硬件配置**:SSD硬盘、充足内存(建议堆内存不超过32GB)
- **网络设计**:节点间低延迟网络,考虑跨机房部署
- **监控告警**:配置集群健康状态监控、磁盘空间监控、性能指标监控
- **备份策略**:定期快照,异地备份

## 6.如何处理ES的脑裂问题?

脑裂是指集群中出现多个Master节点的情况。解决方案:

- **7.x版本之前,设置最小主节点数**:`discovery.zen.minimum_master_nodes = (master节点数/2) + 1`
- **7.x版本后**:自动处理,使用新的集群协调算法(基于投票的选举机制Zen2,自动保证多派选举)
- **网络隔离**:确保节点间网络稳定
- **合理的节点角色分配**:专用Master节点不存储数据,分离master和data角色
- 部署奇数个master节点,保证多数派
- 监控告警第一时间人工手动切换

## 7.如何优化写入性能?

优化策略:

- **批量写入**:使用Bulk API,合理设置批次大小(5-15MB)
- **关闭不必要的功能**:写入时禁用refresh和replica
- **增加索引缓冲区**:调整`indices.memory.index_buffer_size`
- **优化分片数量**:避免过多分片,建议每个分片20-40GB
- **使用SSD**:提升I/O性能
- **调整translog**:增加`index.translog.flush_threshold_size`
- **避免深度嵌套**:减少文档复杂度

## 8.如何做冷热数据分离架构?

在日志 / 监控场景中:

- **热数据(Hot Data)**:最近几天或几周,查询频繁,需要高性能(SSD)。
- **温数据(Warm Data)**:历史数据,查询频率降低,但仍可能使用。
- **冷数据(Cold Data)**:很少被查询,只保留存档,用廉价存储(SATA/HDD/对象存储)。

👉 直接把所有数据都放在 SSD + 高性能节点上成本太高。
👉 冷热分离能**平衡性能和成本**。

**实现方案:**

- **节点标签**:配置节点角色,为节点设置hot/warm/cold标签

- **索引生命周期管理（ILM）**：自动管理索引的生命周期，通过ILM自动迁移到 warm/cold 节点。
- **分片分配**：通过 `index.routing.allocation` 控制分片位置
- **硬件配置**：热节点使用SSD和高配CPU，冷节点使用HDD
- Rollover：当索引超过一定大小/天数时，自动新建索引

配置示例：

json

```
{
  "index.routing.allocation.require.temperature": "hot",
  "index.number_of_replicas": 1
}
```

## 9、Logstash的Pipeline是如何工作的？

Logstash Pipeline分为三个阶段：

- **Input阶段**：从数据源获取数据（file、kafka、beats等）
- **Filter阶段**：数据处理和转换（grok、mutate、date等）
- **Output阶段**：将数据发送到目标（elasticsearch、file、kafka等）

工作流程：

1. Input插件将外部数据转换为事件
2. 事件进入队列（内存或持久化队列）
3. Filter按顺序处理事件
4. Output将处理后的事件发送到目标

## 10、ES性能问题如何排查优化？

排查步骤：

1. **检查集群状态**：

```
GET /_cluster/health
GET /_cat/nodes?v
```

2. 分析慢日志：
    - 开启慢查询日志
    - 分析耗时操作
3. 检查资源使用：
    - CPU使用率
    - 内存和GC情况
    - 磁盘I/O
    - 网络流量
4. **查看热点线程**：

```
GET /_nodes/hot_threads
```

5. **使用Profile API**：分析查询执行计划
6. **检查分片分布**：是否均衡
7. 优化建议：
    - 调整JVM参数
    - 优化索引设置
    - 重构查询语句
    - 扩容或调整架构

# 3、ELK相关面试题2

## 一、基础原理

### 1. ELK/EFK 架构是什么？各自的作用？

- **E = Elasticsearch**：分布式搜索与分析引擎，用来存储和查询日志。
- **L = Logstash / F = Fluentd**：日志收集、解析和转发工具。
- **K = Kibana**：可视化平台，用于日志查询、仪表盘、告警。
  👉 区别：EFK 用 **Fluentd** 代替 Logstash，更轻量、适合容器化（K8s）。

---

### 2. Elasticsearch 倒排索引原理是什么？

- 倒排索引：存储"词项 → 文档 ID 列表"的映射。
- 查询时先定位关键词对应的文档集合，再做聚合或过滤。
- 适合全文搜索，避免逐文档扫描。

---

### 3. Elasticsearch 的分片和副本机制？

- **Primary Shard**：存储原始数据，数据分布式存储的基本单位。
- **Replica Shard**：Primary 的拷贝，用于高可用和分担查询压力。
- 写入：先写 Primary，再同步到 Replica。
- 查询：Primary 和 Replica 都可以响应。

---

### 4. 写入流程（refresh / commit / flush / merge）？

1. 文档写入 → 内存 buffer + translog。
2. **refresh**（默认 1s）：将 buffer 写成 segment，打开供搜索，但不落盘。
3. **commit**：把 segment 元数据 + translog checkpoint 持久化。
4. **flush**：清空 buffer，强制 commit，重置 translog。
5. **merge**：后台合并小 segment → 大 segment，删除标记文档被物理清理。
   👉 **refresh = 可搜索，commit/flush = 持久化，merge = 性能优化。**

---

### 5. Elasticsearch 是强一致还是最终一致？

- 默认 **最终一致性**：写入 Primary 成功 → 异步复制到 Replica。
- 如果 Primary 崩溃，未同步的写入可能丢失。
- 可通过 `wait_for` 参数或 `refresh` 提高一致性。

---

## 二、查询与索引

**6. ES 如何实现高效查询?**

- 倒排索引 + segment 文件不可变（无锁并发）。
- filter cache：过滤条件可缓存。
- 分片并行查询，结果再合并。
- 使用 doc values 支持聚合和排序。

---

**7. 更新和删除文档的底层实现?**

- **删除**：打删除标记（tombstone），不立即物理删除，merge 时清理。
- **更新**：实际操作是"删除旧文档 + 新写入文档"。

---

**8. 如何优化查询性能?**

- 使用 `keyword` 类型做精确匹配，避免 text。
- 避免深度分页，推荐 `search_after` 或 `scroll`。
- 避免通配符前缀查询 `*xxx`。
- 合理使用 filter（可缓存），少用 script。
- 限制返回字段 `_source`。
- 控制分片数量，避免过小或过多。

---

**9. 什么是索引模板（Index Template）?**

- 定义索引的 **mapping** 和 **settings**（分片、副本数、分词器）。
- 新建索引时会自动应用匹配的模板，保证一致性。
- 适用于日志系统按时间滚动生成的索引。

---

**10. 什么是 ILM（Index Lifecycle Management）?**

- 索引生命周期管理策略，自动执行：
  - **Hot**（写入频繁）
  - **Warm**（只读，副本数减少）
  - **Cold**（历史数据，迁移到便宜存储）
  - **Delete**（过期删除）
- 减少运维成本，提高集群稳定性。

---

# 三、Logstash / Fluentd

**11. Logstash 架构是什么?**

- **Input**：数据源（file、beats、kafka）。
- **Filter**：数据解析和清洗（grok、mutate、date）。
- **Output**：输出到 ES、Kafka、文件。
  👉 属于 ETL 工具。

---

**12. Fluentd 与 Logstash 的区别?**

- Fluentd 用 Ruby/C 写，轻量、资源消耗小，更适合 K8s DaemonSet 部署。
- Logstash 功能更强大，但资源占用高。
- 常见实践：EFK = Fluentd + ES + Kibana。

---

**13. grok 插件的作用?**

Grok是Logstash最重要的插件之一，用于结构化非结构化日志。

- 用正则表达式提取日志字段，转为结构化数据。
- 例如：把 `127.0.0.1 GET /index.html 200` → `{ip:127.0.0.1, method:GET, uri:/index.html, status:200}`。

---

**14. 如何优化 Logstash 性能?**

- 增加 `pipeline.workers`，提高并行度。
- 使用 `persistent queue` 保证容错。
- 能用 Beats 采集的日志尽量不用 file input。
- 避免复杂 grok，多用 dissect 或 kv 插件。

---

**15. Filebeat 的优势?**

- Go 实现，单进程低资源占用。
- 原生支持 Docker/K8s 日志采集。
- 可直接输出到 ES 或 Logstash。
- 模块化配置，支持 Nginx、MySQL 等常见日志格式。

---

# 四、Kibana 与运维

**16. Kibana 的主要功能?**

- 搜索与过滤日志（DSL / Lucene 语法）。
- 数据可视化（图表、Dashboard）。
- Dev Tools（执行 ES DSL 查询）。
- Alerting（告警）。
- 管理 ES 索引。

---

**17. Kibana 是如何与 Elasticsearch 交互的?**

- 通过 HTTP/REST API。
- Kibana 的查询最终会转换成 ES DSL。
- 所有可视化面板底层都是 ES 查询结果。

---

**18. 如果 Kibana 查询慢，可能原因是什么?**

- ES 查询本身慢（分片过多、聚合复杂）。
- Kibana 取回了过多字段/数据量太大。
- ES 节点资源不足（CPU、heap）。
  👉 解决：优化索引设计，合理分片，使用 filter cache，升级集群配置。

---

**19. Split Brain 问题是什么？如何避免?**

- 多个 ES 节点因网络分区，各自选举出主节点，导致数据不一致。
- 解决：
  - ES7+ 通过 `master_nodes` 和投票机制避免；
  - 确保 master 节点数为奇数；
  - 网络和 ZooKeeper 等外部一致性工具也能辅助。

---

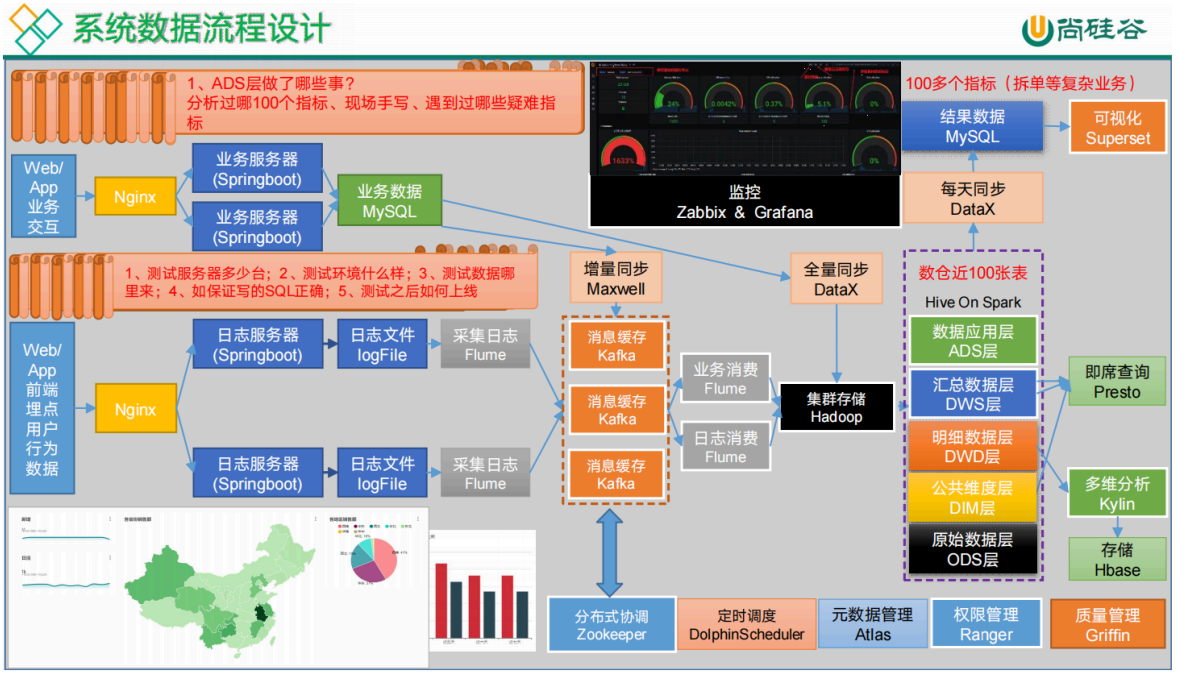
**20. 生产环境如何保证 ELK/EFK 集群稳定?**

- 分片规划：单分片不超过 50GB，避免过小或过多。
- 索引生命周期管理（ILM）：冷热分离，过期删除。
- JVM heap < 50% 内存，避免 GC 频繁。
- 监控 ES heap、segment 数、查询耗时。
- 日志采集用 Filebeat/Fluentd，减少 Logstash 压力。
- 定期 rollover/merge 索引，避免 segment 过多。

# 十一、数仓

## 1、离线数仓


## 1.项目架构




（1）输入系统：前端埋点产生的用户行为数据、JavaEE 后台产生的业务数据、个别公司有爬虫数据。

（2）输出系统：报表系统、用户画像系统、推荐系统

## 1.1 集群规模

### 1）磁盘方某考虑


### 2）cpu方面考虑

20 核物理 CPU 40 线程 * 8 = 320 线程

（指标 100-200）

### 3）内存方某考虑

内存 128g * 8 台 = 1024g （计算任务内存 800g，其他安装框架需要内存）

128m =》512M 内存

=》

200g 数据 、800g 内存

## 1.2 集群参考案例


**Master 节点：**管理节点，保证集群的调度正常进行；主要部署 NameNode、ResourceManager、HMaster 等进程；非 HA 模式下数量为 1，HA 模式下数量为2。

**Core 节点**：为计算及存储节点，我们在 HDFS 中的数据全部存储于 core 节点中，因此为了保证数据安全，扩容 core 节点后不允许缩容；主要部署 DataNode、NodeManager、RegionServer 等进程。非 HA 模式下数量≥2，HA 模式下数量≥3。

**Common 节点**：为 HA 集群 Master 节点提供数据共享同步以及高可用容错服务；主要部署分布式协调器组件，如 ZooKeeper、JournalNode 等节点。非 HA 模式数量为 0，HA 模式下数量≥3。

> 1、数据传输数据比较紧密的放在一起（Kafka、clickhouse）
>
> 2、客户端尽量放在一到两台服务器上，方便外部访问
>
> 3、有依赖关系的尽量放到同一台服务器（例如：Ds-worker 和 hive/spark，ClickHouse必须单独部署）

| Master | Master | core | core | core | common | common | common |
|---|---|---|---|---|---|---|---|
| nn | nn | dn | dn | dn | JournalNode | JournalNode | JournalNode |
| rm | rm | nm | nm | nm | | | |
| | | | | | zk | zk | zk |
| hive | hive | hive | hive | hive | | | |
| | | kafka | kafka | kafka | | | |
| spark | spark | spark | spark | spark | | | |
| datax | datax | datax | datax | datax | | | |
| Ds-master | Ds-master | Ds-worker | Ds-worker | Ds-worker | | | |
| maxwell | | | | | | | |
| superset | | | | | | | |
| mysql | | | | | | | |
| flume | flume | | | | | | |
| flink | flink | | | | | | |
| | | | redis | | | | |
| hbase | | | | | | | |

# 2.数仓建模

## 2.0 数据建模流程

**1、数据调研**，明确业务和字段、指标、粒度

**2、明确数据域**，明确业务分类，比如：用户域、流量域、交易域、工具域、互动域...

**3、构建业务矩阵**

(1) 用户域:

登录、注册

(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、取消下单、取消支付

(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:

点赞、评论、收藏

**4、底层数据建模，自下而上**

- **ODS层**：原始表，直接从业务库抽取的**原始数据层**

  ①保持数据原貌不做任何修改 起到备份作用
  ②采用压缩 减少磁盘空间，采用 Gzip 压缩
  ③创建分区表，防止后续全表扫描

- **DWD层**：事实表，对ODS数据进行轻度清洗、规范化、去重的**明细数据层**

  1）事务型事实表：每个事件都记录下来

  找原子操作

  a）选择业务过程，选择感兴趣的业务过程。 产品经理提出的指标中需要的。

  b）声明粒度，粒度：一行信息代表什么含义。可以是一次下单、一周下单、一个月下单。如果是一个月 的下单，就没有办法统计一次下单情况。保持最小粒度。只要你自己不做聚合操作就可以。

  c）确定维度，确定感兴趣的维度。 产品经理提出的指标中需要的。例如：用户、商品、活动、时间、地 区、优惠卷

  d）确定事实，确定事实表的度量值。 可以累加的值，例如，个数、件数、次数、金额。

  2）事务型快照事实表：每隔一段时间周期采样存储快照

  ①选择业务过程
  ②声明粒度 =》 1 天
  ③确定维度
  ④确定事实

  3）累积型快照事实表：一张表记录业务流程完整生命周期

  ①选择业务过程
  ②声明粒度
  ③确定维度
  ④确定事实 ，确定多个事实表度量值

| 类型 | 定义 | 特点 | 电商例子 | 表结构示例 | 应用场景 |
| --- | --- | --- | --- | --- | --- |
| **事务型事实表** | 记录每一条最细粒度的业务事件 | 数据量大、只追加不修改、可追溯 | 订单明细事实表（fact_order_detail） | 订单ID、用户ID、商品ID、下单时间、数量、金额 | 明细分析：用户下单行为、商品销量统计 |
| **周期型快照事实表** | 按固定周期采样，存储快照 | 每周期一条记录，累积存储，趋势分析 | 每日订单统计事实表（fact_order_day） | 日期、订单总数、销售总额、下单用户数 | 趋势分析：日销售额、周活跃用户数 |
| **累积型快照事实表** | 记录业务过程全生命周期，状态推进时更新 | 一行覆盖全过程，会更新，不是只追加 | 订单生命周期事实表（fact_order_lifecycle） | 订单ID、下单时间、支付时间、发货时间、收货时间、完成时间 | 流程分析：订单转化率、平均发货/收货时长 |

- **DIM层**：维度表，存储元数据/基础数据的**公共维度层**

  ①维度整合 减少 join
  - a）商品表、商品品类表、spu、商品一级分类、二级分类、三级分类=》商品维度表
  - b）省份表、地区表 =》 地区维度表
  - c）活动信息表、活动规则表 =》 活动维度表

  ②拉链表：对用户表做了拉链。缓慢变化维 场景

**5、指标体系建设，自上而下**

- **ADS层**：应用表，直接服务报表提供分析结果的**数据应用层**

  统计指标、需求、日活、月活、总计、新增、留存、转化率、GMV
- **DWS层**：聚合表，按分析主题数据域建模进行汇总聚合的**服务数据层/汇总数据层**

  需求：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 + 统计粒度 + 业务限定

  > 例如，统计，每天各个省份手机品牌交易总额

  交易总额 （下单 + 金额 + sum ） + 每天 + 省份 + 手机品牌

  找公共的：业务过程 + 统计周期 + 统计粒度 建宽表

## 2.1 数仓建模意义

1. **统一视角，消除数据孤岛**，统一口径，统一维度
2. **提升查询性能**，通过雪花、星型模型面向分析设计，避免重复计算，支持OLAP多维分析（切片、钻取、聚合）
3. **支撑业务分析和决策**，报表和BI分析
4. **保证数据一致性和可追溯性**，明确数据血缘
5. 降低数据使用门槛，形成 **标准化的数据层次**（ODS → DWD → DWS → ADS）

## 2.2 ER建模（面向事务OLTP）

ER 模型是数据库设计的一种方法，用来描述 **实体、属性和关系。**
在 ER 图中，实体用矩形，属性用椭圆，关系用菱形。
它常见的关系有 **一对一、一对多、多对多**，在数据库中分别通过外键或中间表来实现。
ER 模型的意义是 **把业务需求转化为数据模型**，帮助我们构建清晰、规范的数据库结构。

## 2.3 维度建模（面向分析OLAP）

维度建模是一种 **面向分析的建模方法**，主要用于数据仓库和 OLAP 系统。
核心思想：
👉 **事实表（Fact Table）记录业务过程，维度表（Dimension Table）描述业务背景。**

**1）核心要素**

**事实表（Fact Table）**：事务型事实表、周期快照事实表、累积型快照事实表

- 存储业务过程的度量值（可统计的指标，可以累加的）。
- 例：销售额、订单数量、点击次数；个数、件数、金额、次数
- 一般包含 **外键（维度关联）+ 度量值**；数据量大，通常增量

**维度表（Dimension Table）**

- 存储业务分析的"角度"或"上下文信息"；没有度量值，全是描述信息
- 例：时间维度、客户维度、产品维度、地域维度；身高、体重、年龄、性别
- 一般字段较多，冗余存储，利于查询；数据量小，通常全量

**度量（Measure）**

- 可汇总的数值指标（如销售额、利润）。

**2）常见建模方式**

**星型模型（Star Schema）**

- 事实表在中心，维度表围绕在外。
- 查询简单，性能好，最常用。

**雪花模型（Snowflake Schema）**

- 维度表再做规范化，形成多层级关系。
- 节省存储，但查询复杂，性能差。

```
          年-月-日表
            |
          时间维度
            |
客户基本信息表 —— 客户维度
            |
     产品类别表 —— 产品维度 —— 品牌表
            |
       城市表 —— 地区维度 —— 省份表
```

**星座模型（Fact Constellation）**

- 多个事实表共享部分维度。
- 适合复杂业务场景。

```
          时间维度
            |
客户维度 — 销售事实表 — 产品维度
            |
          地区维度
            |
          退货事实表
```

## 2.4 数仓每层做了什么事?

**1）ODS层做了哪些事?**

（1）保持数据原貌，不做任何修改
（2）压缩采用 gzip，压缩比是 100g 数据压缩完 10g 左右。
（3）创建分区表

**2）DIM/DWD 层做了哪些事?**

建模里面的操作，正常写。
（1）数据清洗的手段
HQL、MR、SparkSQL、Kettle、Python（项目中采用 SQL 进行清除）

（2）清洗规则
1、金额必须都是数字，[0-9]、手机号、身份证、匹配网址 URL
2、解析数据、核心字段不能为空、过期数据删除、重复数据过滤
3、json => 很多字段 =》一个一个判断 =》取数，根据规则匹配

（3）清洗掉多少数据算合理
参考，1 万条数据清洗掉 1 条。

（4）脱敏
对手机号、身份证号等敏感数据脱敏。

①加*
1350013 互联网公司经常采用

②加密算法 md5 需要用数据统计分析，还想保证安全

美团 滴滴 md5（12334354809）=》唯一值

③加权限 需要正常使用

军工、银行、政府

（5）压缩 snappy

（6）orc 列式存储

**3）DWS层做了哪些事?**

指标体系建设里面的内容再来一遍。

**4）ADS 层做了哪些事？**

一分钟至少说出 30 个指标。

> 日活、月活、周活、留存、留存率、新增（日、周、年）、转化率、流失、回流、七天内连续 3 天登录（点赞、收藏、评价、购买、加购、下单、活动）、连续 3 周（月）登录、GMV、复购率、复购率排行、点赞、评论、收藏、领优惠卷人数、使用优惠卷人数、沉默、值不值得买、退款人数、退款率 topn 热门商品
>
> 产品经理最关心的：留转 G 复活

## 3.项目中遇到问题

### 1）Flume 零点漂移

- **问题**：由于 Flume agent 收集日志时时间戳解析异常，导致落盘时间与实际业务时间有偏移（比如零点数据写到前一天分区）。
- **解决**：
  - 使用 `TimestampInterceptor` 校正时间字段；
  - 确保日志源时间格式统一，落库前做时间字段标准化。

### 2）Flume 挂掉及优化

- **问题**：Agent 不稳定，内存溢出、channel 堵塞导致服务挂掉。
- **解决**：
  - 调整 channel 类型（memory + file channel 结合），保证高可用；
  - 启用监控报警，agent 异常自动重启；
  - 优化 sink 并发写入，避免单点瓶颈。

### 3）DataX 空值、调优

- **问题**：导入导出时字段为空或类型不匹配，容易失败。
- **解决**：
  - 配置 `nullFormat` 参数，统一空值处理；
  - 增大 `channel` 数量、并行度，减少单批数据量，提升吞吐。

### 4）HDFS 小文件处理

- **问题**：过多小文件导致 NameNode 内存压力大。
- **解决**：
  - 使用 Hive 合并小文件（insert overwrite + 合并文件大小）；
  - 上游 ETL 预先做文件合并；
  - 采用 `CombineInputFormat` 读取小文件。

### 5-9）Kafka 常见问题

- **挂掉**：部署多 broker，多副本机制，启用监控自动重启。
- **丢失**：生产端设置 `acks=-1`，开启 ISR 副本确认；消费者开启 `enable.auto.commit=false`，手动提交 offset。
- **重复**：借助幂等生产者（`enable.idempotence=true`）；消费端保证幂等写入。
- **数据积压**：扩容 partition、增加消费者并行度，或提高消费批次大小。
- **乱序/顺序**：
  - 全局乱序 → Kafka 本身无法保证；
  - 分区内顺序 → 由 partition 保证，只要生产端按 key 投递即可。

## 10-12）Kafka 优化与底层原理

- **提高吞吐量**：批量发送（`batch.size`）、压缩（`compression.type`）、增加 partition。
- **高效读写原理**：
  - 顺序写磁盘（零拷贝 + 页缓存）；
  - segment 文件结构 + 索引查找；
  - 拉取模型（消费者主动拉）。
- **单条日志大小**：默认限制 1MB，可在 broker 配置 `message.max.bytes` 调整。

## 13-15）Hive 优化 & 倾斜

- **优化（Hive on Spark）**：开启 `tez` 或 `spark` 执行引擎，使用 `orc/parquet` 列存格式，合理设置 `mapreduce.split`，避免过度切分。
- **数据倾斜**：
  - 过滤极端值；
  - MapJoin 小表广播；
  - 使用 `salting` 技术打散热点 key；
  - 分桶表。

## 19）疑难指标编写

- **连续活跃**：使用 `row_number()` 窗口函数 + 日期差分计算；
- **1/7/30 日指标**：建宽表 + 窗口聚合；
- **路径分析**：基于用户行为日志 session 化 + 窗口排序；
- **留存率**：当日活跃与历史活跃取交集；
- **复购率**：分组统计品牌 + 去重；
- **优惠券补贴率**：已使用金额 / 发放金额；
- **同时在线人数**：利用区间展开表，统计任意时间点并发用户数。

## 20）DS 任务挂了怎么办？

- **解决方案**：
  - 设置任务失败自动重试；
  - 配置任务依赖、失败处理策略（跳过/重跑）；
  - 人工介入时支持手动 rerun；
  - 日志快速定位问题（资源不足 / 脚本错误 / 下游不可用）。

## 21）DS 故障报警

- **手段**：
  - 集成邮件、钉钉/飞书机器人报警；
  - 配置 SLA，任务超时/失败自动通知；
  - 可对关键任务配置短信/电话告警。

# 4.业务相关

## SKU SPU

SKU：一台银色、128G 内存的、支持联通网络的 iPhoneX。
SPU：iPhoneX。（指代产品）
Tm_id：品牌 Id 苹果，包括 IPHONE，耳机，MAC 等。

## 订单表和订单详情表

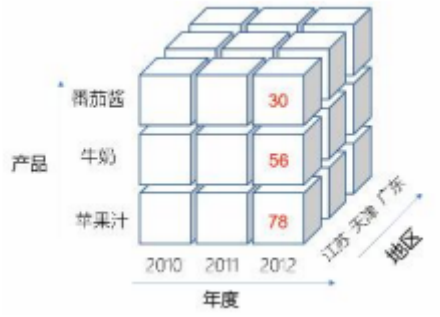
1、订单表的订单状态会变化，订单详情表不会，因为没有订单状态。
2、订单表（一般用在外部关联）记录 user_id，订单 id 订单编号，订单的总金额 order_status，支付方式，订单状态等。
3、订单详情表记录 user_id，商品 sku_id，具体的商品信息（商品名称 sku_name，价格order_price，数量 sku_num）

## 上卷和下钻

**上卷：**上卷是沿着维度的层次向上聚集汇总数据。

**下探（钻）：**下探是上卷的逆操作，它是沿着维度的层次向下，查看更详细的数据。

比如这个经典的数据立方体模型：




维度有产品、**年度**、**地区**等，统计销售额。实际上，维度还可以更细粒度，如时间维可由**年、季、月、日**构成，地区也可以由**国家、省份、市、区县**构成等。

下钻可以理解为由粗粒度到细粒度来观察数据，比如对产品销售情况分析时，可以沿着时间维从年到月到日更细粒度的观察数据。

> 上卷和下钻是相逆的操作，所以上卷可以理解为删掉维的某些粒度，由细粒度到粗粒度观察数据，向上聚合汇总数据。

## TOB和TOC

TOB（toBusiness）：表示面向的用户是企业。
TOC（toConsumer）：表示面向的用户是个人。

## 流转G复活指标

**1）活跃**

日活：100 万；月活：是日活的 2-3 倍 300 万

总注册的用户多少？1000 万-3000 万之间。

渠道来源：app 公众号 抖音 百度 36 氪 头条 地推

**2）GMV**（商品交易总额，成交额）

GMV：每天 10 万订单 （50 – 100 元） 500 万-1000 万

10%-20% 100 万-200 万（人员：程序员、人事、行政、财务、房租、收电费）

**3）复购率**

某日常商品复购；（手纸、面膜、牙膏）10%-20%

电脑、显示器、手表 1%

**4）转化率**

商品详情 =》 加购物车 =》下单 =》 支付

1%-5% 50-60% 80%-95%

**5）留存率**

1/2/3-60 日、周留存、月留存

搞活动： 10-20%

## 数仓中使用哪种的文件存储格式

常用的包括：textFile，ORC，Parquet，一般企业里使用 ORC 或者 Parquet，因为是列式存储，且压缩比非常高，所以相比于 textFile，查询速度快，占用硬盘空间少。

## 拉链表的退链如何实现

拉链表用于记录维度表中的历史变化。在拉链表中，当某个维度属性发生变化时，会插入一条新的记录，同时将原记录的有效期设置为截至。退链是指将一个已经生效的变更恢复到上一个状态。实现思路如下：

（1）定位要退链的记录：例如找到用户最近一次信息更新

（2）查询上一条记录：查询这条记录之前的一条记录（主键相同，不同版本记录，而不是单指上一条）

（3）更新有效期：将当前记录的生效时间或者有效开始时间更新为无效，将上条记录截至日期改为最大值。

## map任务从 0%到1%很慢，但是1%到100%很快，期间出了什么问题?

1）申请资源
2）加载数据
3）如果存在于外部系统交互，获取连接慢

## 数据倾斜场景中除了group by和join外，还有哪些场景?

（1）使用 Snappy 压缩，原始文件大小不等，Map 阶段数据倾斜。
（2）over 开窗，partition by 导致数据倾斜。

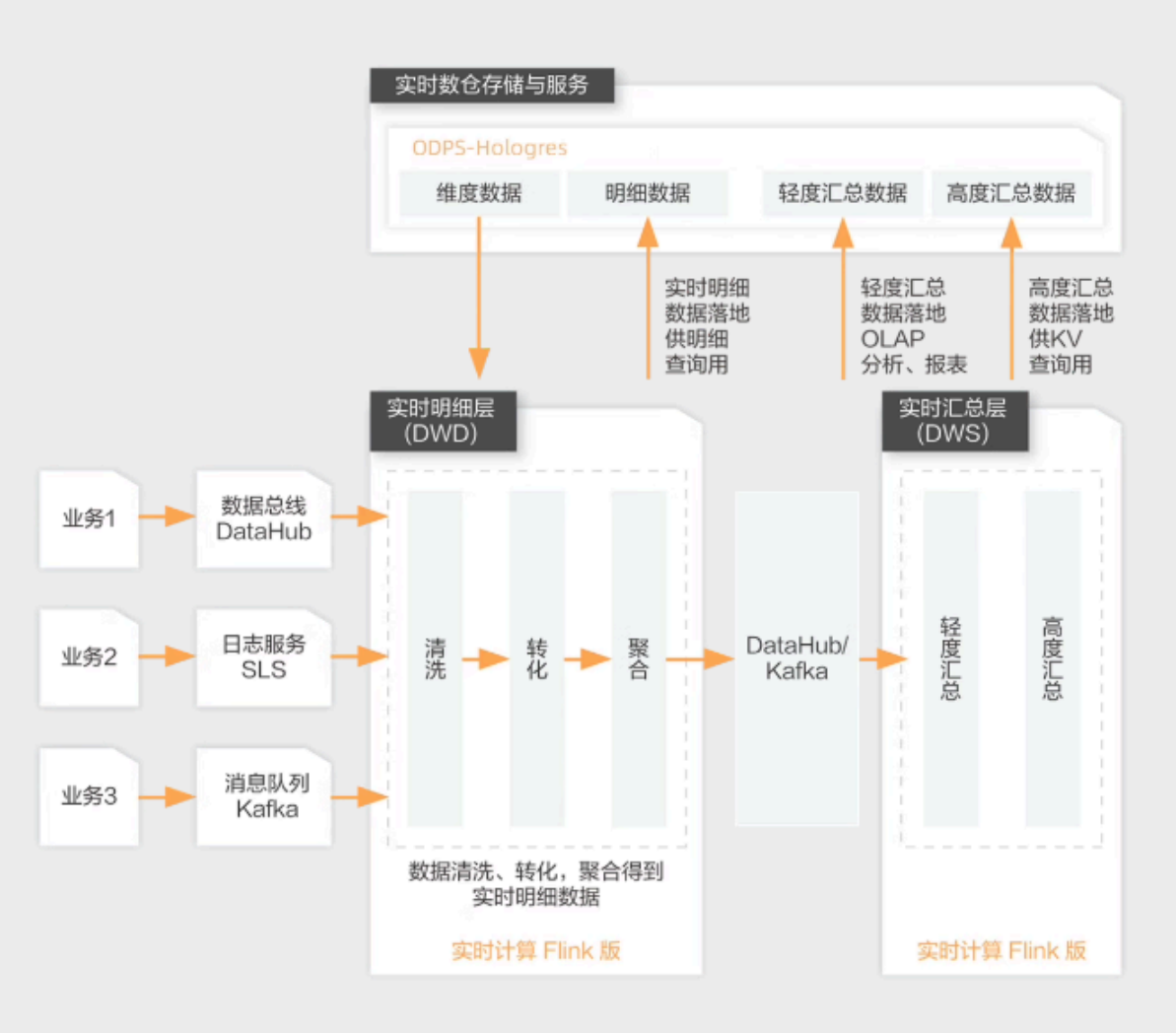
## 当ADS计算完，如何判断指标是正确的?

（1）样本数据验证：从计算结果抽取部分样本数据，与业务部门实际数据对比。
（2）逻辑验证：检查指标的计算 sql 是否正确
（3）指标间关系验证：比较不同指标间关系，检查它们是否符合预期。例如：某个指标是另外一个指标的累计值，那么这两个指标一定存在关系
（4）历史数据对比：将计算结果和过去数据进行对比，观察指标的变化趋势
（5）异常值检测：检查计算结果中是否存在异常值
（6）跨数据部门对比：可以将计算结果与其它部门或团队的数据进行对比，进一步验证

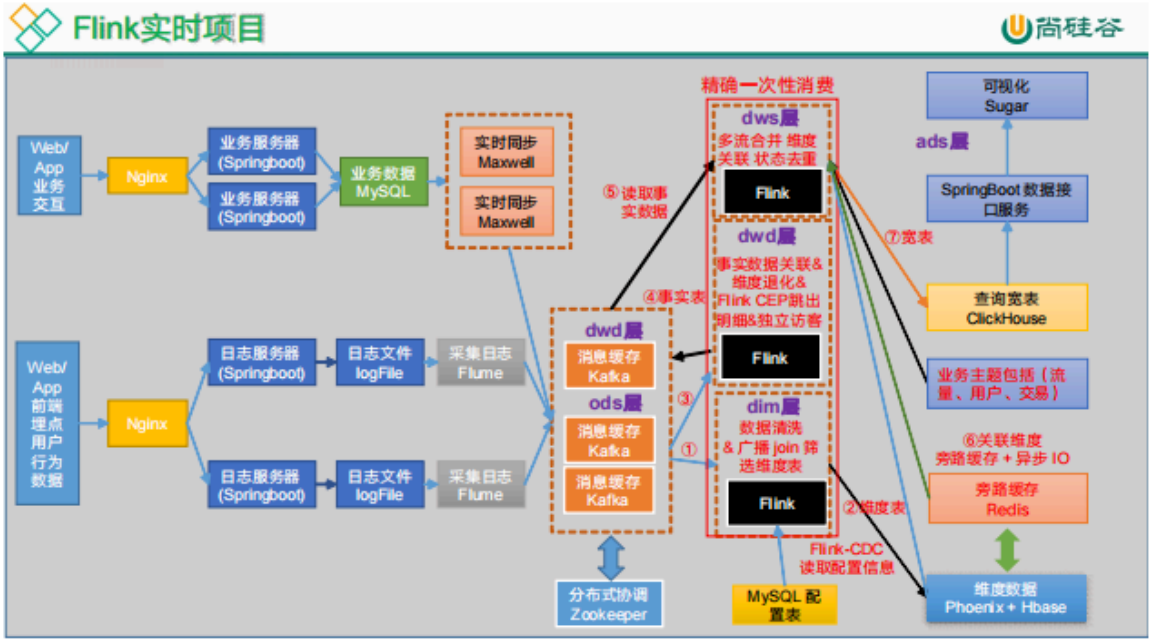
**ADS指标计算错误，如何解决?**

（1）确定错误范围：找出指标计算错误的时间范围，指标及相关维度，缩小排查范围
（2）检查数据处理逻辑：从 ods 层开始排查，找出可能导致计算错误的数据清洗，转换和聚合等步骤，确认处理逻辑不出错误
（3）审查数据质量：确保每层数据完整性，一致性，准确性和时效性
（4）重新计算指标：修复数据质量问题和处理逻辑后，重新计算

# 2、实时数仓



# 1.项目架构




# 2.数据建模

同离线数仓

## 2.1 实时组件存储数据量

**1）Kafka：**

ods 层和 dwd 层数据

ods 和 dwd 数据一致，每天约 200G 数据

考虑 kafka 副本 2 个，保存三天，kafka 存储 400G 数据

**2）Hbase：**

存储 dim 层数据，与离线一致

**3）Clickhouse：**

存储 dws 层数据，每天约 20~30G 数据

考虑 dws 层数据保存一年，Clickhouse 三个副本

数据约 15~20T

## 2.2 实时QPS

QPS 峰值：20000 条/s 或者 2M/s

# 3.项目中遇到问题

## 3.1 项目中哪有用到状态编程，状态如何存储，怎么解决大状态问题？

> **维度动态分流**：
>
> - 场景：比如订单流要根据维度表（商品类目、地区）动态路由到不同下游；
> - 解决：把维度表数据做成广播流（Broadcast State），每个算子副本都能拿到完整维度信息，做到“动态分流”。
>
> **新老访客修复**：
>
> - 场景：判断用户是新访客还是老访客，需要记录该用户是否出现过；
> - 解决：用 **Keyed State（键控状态）** 按用户 ID 存储访问记录，后续到来数据直接查询状态即可。

**1）DIM维度 动态分流使用广播状态，新老访客修复使用键控状态**

状态中数据少使用 HashMap，状态中数据多的使用 RocksDB（把状态序列化后存到本地磁盘 + 部分缓存内存，保证不会 OOM。）

**2）大状态优化手段**

当状态特别大（百 GB ~ TB 级别），就需要一些专门优化：

1. **使用 RocksDB**
   - 支持超大状态存储，内存放索引，数据落盘。
   - 适合大 key/大 value 的业务场景。
2. **开启增量检查点（Incremental Checkpoint）**
   - 默认检查点会全量写快照，开销大；
   - 增量 checkpoint 只写变化部分，速度快，资源开销小。
3. **本地恢复（Local Recovery）**
   - 在节点本地磁盘保存 checkpoint 数据；
   - 作业重启时优先从本地恢复，减少远程存储下载时间。
4. **设置多目录（RocksDB 多目录存储）**
   - RocksDB IO 密集，可以配置多个磁盘目录分散压力。
5. **预定义 RocksDB 选项（memory + disk 策略）**
   - 设置 writer buffer、block cache 等参数，合理利用内存和磁盘；
   - 避免 RocksDB 频繁 compact、性能抖动。

## 3.2 项目中哪里用到了反压，造成的危害，定位解决？

> **什么是反压？**
>
> 反压指的是：**下游算子处理不过来时，向上游“反向施压”，限制上游继续发送数据，从而避免系统崩溃的一种机制**。换句话说，就是 **消费者速度 < 生产者速度**，导致数据在中间环节堆积，系统自动减缓上游的处理速度，防止无限制积压。

**1）项目中反压造成的原因**

流量洪峰：如果是暂时性的则不需要解决
频繁 GC：比如代码中大量创建临时对象导致 JVM GC 频繁，处理速度下降
大状态：新老访客修复（数据延迟导致的错误区分新老用户）需要存储大量用户 ID 状态，RocksDB IO 变慢
关联外部数据库：

1.从 HBase 读取维度数据 → IO 慢；

2.Sink ClickHouse/MySQL 时单条写入，速度不足。数据倾斜：keyby 之后不同分组数据量不一致
数据倾斜：`keyBy` 后某些 key 数据量过大，单个并行度算子处理不过来

**2）反压的危害**

1.Checkpoint 超时失败 → 作业挂掉
2.内存压力增大 → OOM → 作业失败
3.处理延迟增加 → 时效性降低，无法满足实时需求。

**3）定位反压**

（1）利用 Web UI 定位

1.打开 Flink Web UI → 找反压指标；。Flink 现在在 UI 上通过颜色和数值来展示繁忙和反压的程度。

2.上游都是 high，UI 颜色提示：红色(high) 表示算子有严重反压，找到第一个为 ok 的节点就是瓶颈节点。

3.operator chain 禁用，可以更细粒度地看到哪个算子反压；

（2）利用 Metrics 定位

1.可以根据指标分析反压： buffer.inPoolUsage、buffer.outPoolUsage
2.可以分析数据传输，throughput / busyTime 等指标 → 判断是否处理不过来。

**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。

（1）查看是否数据倾斜：热点 key 打散（加盐 salting）；广播 join 替代 shuffle join。
（2）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。
（3）分析 GC 日志，调整代码。减少临时对象创建，优化序列化方式。
（4）资源不合理造成的：调整资源
（5）与外部系统交互：写 MySQL、Clickhouse：攒批写入；读 HBase：异步 IO、旁路缓存

## 3.3 数据倾斜问题如何解决？

**1）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。

**2）数据倾斜解决**

（1）数据源倾斜

比如消费 Kafka，但是 Kafka 的 Topic 的分区之间数据不均衡
读进来Flink之后立即调用重分区算子：rescale、rebalance、shuffle 等打散数据

> `rebalance()`：轮询分发，最均匀，但 shuffle 开销大；
> `rescale()`：按并行度比例分发，轻量；
> `shuffle()`：随机分发。

（2）单表分组聚合（纯流式）倾斜，直接对 keyBy 的数据聚合，热点 key 把一个 subtask 撑爆

API：用 `flatMap` 做 **预聚合**，在 map 端先攒一批再发给下游；
SQL：开启 **MiniBatch**（延迟一会儿，批量触发计算）；开启 **Local-Global 优化**（先局部聚合，再全局聚合）。

（3）单表分组开窗聚合倾斜，窗口聚合时某些 key 太集中，导致一个窗口计算特别重。

**解决**：两阶段聚合（俗称 **打散 key**）：

1. **第一阶段**：
   1）把原始 key 加一个随机前缀/后缀，比如 `key+"_"+random(0, N)`；
   2）先按这个“打散 key”开窗聚合，避免热点集中。
2. **第二阶段**：
   1）聚合结果输出时，携带原始 key 和 windowEnd；
   2）再按 `(原始 key, windowEnd)` 分组，做最终聚合。

（4）查询层可以在最终结果Clickhouse SQL做最终聚合

## 3.4 数据一致性问题？

### （1）上游：Kafka

- Kafka 的 **offset 管理**机制可以保证消费的可重发：
  - 消费者提交 offset，失败时不提交 → 下次还能重新消费；
  - 结合 Flink 的 checkpoint（状态 + offset 一起保存），保证 **端到端精确一次。**
- 默认情况下就是 **at-least-once**，结合 checkpoint 才能做到 **exactly-once。**

### （2）中间：Flink

- **Checkpoint/Savepoint**：把算子状态（如 keyed state、广播状态）和 Kafka offset 一起保存。
- **Exactly-once 模式**：
  - Source（Kafka） → Flink → Sink 的整个链路都和 checkpoint 绑定；
  - 只有当 checkpoint 成功时，才算 offset 已提交。
    👉 这样即便失败重启，也能从一致的位置恢复，不会丢/重。

### （3）下游：Sink（事务 & 幂等）

- **写 Kafka**：
  - 用 **事务性写入**（Kafka 事务 producer），保证数据不会部分提交；
  - 和 Flink checkpoint 协同，确保“要么写入成功，要么回滚”。
- **写 ClickHouse**：
  - ClickHouse 本身不支持分布式事务，但我们用 **幂等写入**（比如用唯一主键 idempotent insert/update，或替换写）；
  - 查询时加 `FINAL`，确保物化视图/合并树（MergeTree）最终一致，不出现脏读。

## 3.5 FlinkSQL性能比较慢如何优化？

> 1、通过设置 **空闲状态保留时间** 清理无效 state，减少大状态；
> 2、开启 **MiniBatch**，批量触发计算，减少状态更新次数；
> 3、使用 **LocalGlobal 两级聚合**，降低 shuffle 压力；
> 4、对 **Distinct** 类操作，开启 **Split Distinct** 提高并行度，多维 distinct 拆成带 Filter 的多个计算，避免单算子过重。
> 这些手段组合起来，整体吞吐量和稳定性提升明显。

### 📌 1. 设置空闲状态保留时间（Idle State Retention Time）

> - **场景**：Flink SQL 做 `GROUP BY`、`JOIN`、`窗口` 等算子时会用到状态（state）。
> - 如果 key 很多，但部分 key 已经长时间不活跃，状态还保留在内存/RocksDB，就会拖慢整体性能。
> - **优化**：通过设置 `idle state retention time`，自动清理长时间未更新的 key 状态，减少状态膨胀。
>   👉 本质：**减少大状态**，提升查询性能。

### 📌 2. 开启 MiniBatch

> - **场景**：Flink SQL 默认是逐条数据触发计算，开销很大。
> - **MiniBatch**：把一段时间内的多条数据攒成一批，再触发聚合/更新。
> - **好处**：
>   - 降低状态更新次数（比如聚合从 1 次/条 → 1 次/批）；
>   - 吞吐量明显提升。
>     👉 本质：**小批量流式计算**，折中实时性和性能。

### 📌 3. 开启 LocalGlobal 聚合

> - **场景**：直接在全局做 `GROUP BY`，所有相同 key 的数据会集中到少数 task，容易倾斜。
> - **Local-Global 模式**：
>   1. **Local 聚合**：先在本地 task 上预聚合；
>   2. **Global 聚合**：再把局部结果汇总。
> - **好处**：
>   - 避免数据倾斜；
>   - 大大减少 shuffle 数据量。
>     👉 本质：**两级聚合，降低 shuffle 压力。**

### 📌 4. 开启 Split Distinct

> - **场景**：`COUNT(DISTINCT user_id)` 这类操作最容易慢，因为需要去重，压力很大。
> - **Split Distinct**：把一个大 Distinct 拆成多个子任务并行处理，再合并结果。
> - **好处**：
>   - 分而治之，提高并行度；
>   - 降低单个算子的状态大小。
>     👉 本质：**把一个热点 Distinct 拆成多个小 Distinct。**

### 📌 5. 多维 Distinct 使用 Filter

- **场景**：`COUNT(DISTINCT user_id)`、`COUNT(DISTINCT device_id)`、`COUNT(DISTINCT order_id)` 等多个 distinct 在一个 SQL 里会非常重。
- **优化**：用 `FILTER` 子句拆开，例如：

```
COUNT(DISTINCT user_id) FILTER (WHERE type = 'user'),
COUNT(DISTINCT device_id) FILTER (WHERE type = 'device')
```

- **好处**：避免多个 distinct 相互影响，提升并行处理能力。
  👉 本质：**把多个 distinct 分开计算，而不是堆在一个大算子里。**

## 3.6 Kafka动态分区增加，Flink监控不到新分区导致数据丢失?

在项目中，我们遇到过 Kafka 动态增加分区导致 Flink Job 无法消费新分区数据的问题。
这是因为 Flink Kafka Source 默认只在 Job 启动时获取分区信息。
我们通过开启 **动态分区发现**：设置 `partition.discovery.interval-millis` 参数，让 Flink 定期扫描 Kafka Broker 的分区信息，这样新增分区就能被消费，避免数据丢失。
同时配合 checkpoint 和幂等写入，保证端到端数据不丢失、不重复。

## 3.7 Flink Checkpoint?

**Checkpoint（检查点）** 是 Flink **状态一致性和容错机制**的核心。
它的作用是：

- **保存算子状态（Keyed State / Operator State）** 的快照；
- **保存 Source 偏移量**（比如 Kafka offset）；
- 当作业失败重启时，从最近的 checkpoint 恢复，保证 **Exactly-Once 或 At-Least-Once** 语义。

**工作原理（端到端）**

1. **触发 Checkpoint**
   - JobManager 定期触发 checkpoint（周期性，默认 10s/30s 可配置）；
2. **Barrier 插入流**
   - Flink 会在每个 Source 的输出流中插入 **Checkpoint Barrier**（标记分界线）；
3. **状态快照**
   - 当 barrier 到达某个算子实例时，算子会把当前状态快照写到 **状态后端**（Memory / RocksDB / FsStateBackend）；
4. **Barrier 传播**
   - Barrier 继续向下游传播，保证整条流上所有算子在同一时间点的状态一致；
5. **完成 Checkpoint**
   - 所有算子状态和 source offset 成功写入后，checkpoint 被标记为 **成功**；
   - 如果失败，checkpoint 被丢弃，不影响下一个周期。

## 3.8 Kafka某个分区没有数据，导致下游水位线没法抬升，窗口无法计算

在注入 Watermark 水位线时，可以设置一个 **最小等待时间/最大空闲等待时间**，让空闲分区不拖慢整体计算

> **背景：**

> 在 Flink 里，如果你要做 **窗口计算**（比如 5 分钟聚合），通常是用 **事件时间 (event time)**。
> 事件时间计算依赖 **Watermark（水位线）** 来推进。如果某个分区完全没数据，那么它的 Watermark 就不会前进（一直卡住）。结果：**全局 Watermark 被拖住**，窗口迟迟不触发计算。
>
> - **Watermark** 的作用 → 告诉 Flink：“我已经收齐了时间戳小于 X 的数据，可以触发窗口了”。
> - Flink 在 **Kafka 多分区**场景下，会取 **所有分区的 Watermark 的最小值** 作为全局水位线。

## 3.9 Redis和HBase数据不一致问题

**① 写入顺序 + 幂等**

- **策略**：先写 HBase，再更新 Redis（保证持久化先完成）；
- 写入必须 **幂等**：重复写不会产生错误。
  - 设计唯一key，redis采用set key value，利用主键唯一覆盖写
  - 写入时带上版本号（乐观锁），HBase和Redis都保存同样版本号，只有当新版本号 > 旧版本号 时才更新

**② 事务/两阶段提交**

- 对 Kafka → Flink → Redis/HBase 的链路，可以用 **Flink 两阶段提交 Sink**：
  1. 写 Redis/HBase 都是第一阶段 “预提交”
  2. Checkpoint 成功后第二阶段 “提交”
- 保证 **Exactly-Once** 语义

**③ 异常重试 + 补偿**

- 失败后重试 HBase 或 Redis
- 定期对比 Redis 和 HBase，做 **数据校验** 和 **补偿修复**

**④ 延时双删（使用缓存失效策略）**

- 更新完数据库立马删除缓存，然后过段时间再删除一遍缓存
- 对 Redis 设置合理 TTL 或 **延迟删除 + 更新**，减少脏数据被读取

## 3.10 双流join关联不上如何解决?

> 双流join本质就是**把一条流的数据先存下来**，等另一条流的数据来了再关联
>
> 在数据库里：`select * from a join b on a.id = b.id`。
> 在 Flink 里：数据是流的，不是静态表，所以 Flink 要把两条流在某个时间段内“对齐”，才能做 Join。

Flink 常见的双流 Join 有三种：

1. **Inner Join**：两边流都有匹配才输出。
2. **Left/Right Join**：一边有数据就输出，另一边没有就补空。
3. **Interval Join**：带上时间范围（eventTime），比如 A 事件要和 B 事件在 ±5 秒内的才能匹配。

**1）为什么会出现双流join关联不上?**

在流处理里，有几个典型原因：

1. **时间对不齐**
   - Flink Join 是基于 EventTime 的，两个流的 watermark 水位线决定了窗口什么时候触发。

- 如果一边数据迟到，超过了窗口等待时间，就无法匹配。

2. **窗口/区间设置不合理**
   - 比如 interval join 设置 `between -5s and 5s`，但是数据实际延迟 10s，结果关联不上。
3. **数据源有丢失/乱序**
   - Kafka 分区没对齐、消费延迟、部分消息没发到流里。

**5）解决方案**

在实际项目中常见的三种思路：

> （1）使用 interval join 调整上下限时间，但是依然会有迟到数据关联不上
> （2）使用 left join，带回撤关联
> （3）可以使用 Cogroup+connect 关联两条流

（1）调整时间区间 / 允许迟到

- 使用 **Interval Join** 时，把上下限时间调大一些，让迟到数据能进来。
- 在水位线里设置 **allowedLateness**，允许一定迟到。

缺点：等待太久会增加状态大小。

（2）用 Left Join + 回撤修正

- 如果一定要保证尽量多的关联结果，可以先做 **Left Join**：
  - 左流的数据立刻输出，即使右流还没到。
  - 当右流数据晚点到达，再做“回撤”（更新之前的结果）。
- 这种方式比较适合 **先出结果，再修正** 的场景，比如报表、埋点统计。

（3）使用 CoGroup（依赖窗口） + ProcessFunction 自定义实现（解决上述灵活性差问题）

- 比如：
  - 将两条流数据用 `keyBy` 把相同的 id 的数据聚合到一起，
  - 然后用 `CoProcessFunction` 或 `CoGroup` 把两边流的数据都存到状态里，
  - 等到另一边的数据来了，再手动输出匹配结果。
- 这种方式灵活，可以自己控制状态保存多久、如何清理、如何补偿迟到数据。

> 本质：存状态 + 等另一边数据 + 匹配 + 清理状态

## 3.11 多流join?

**2 流**：直接用 Flink 提供的 join 或自己写 CoProcessFunction。
**3 流**：推荐链式 join（先 AB，再跟 C）。
**n 流（复杂场景）**：用 KeyedProcessFunction/CoProcessFunction，所有流都放进状态，自己控制 join和状态清理。
**小表 join 大流**：广播流（Broadcast State）。

# 4.生产经验

## 4.1 Flink任务提交使用的哪种模式?

项目中提交使用的 per-job 单作业模式，因为每个 job 资源隔离、故障隔离、独立调优，缺点是启动延迟高

> Session会话模式：启动一个集群，保持一个会话，所有的提交的作业会竞争集群中的资源，适合单个作业规模小、执行时间短的大量作业
>
> Application应用模式：用户的 **main 函数直接在 JobManager 中运行**，而不是先在客户端运行再提交。更适合 **K8s 原生部署**，能减少客户端依赖。

## 4.2 Flink任务提交参数，JobManager和TaskManager分别给多少?

我们的 JobManager 一般给 **1 核 + 2G 内存**，因为主要是负责调度和 checkpoint，不需要太多资源。
TaskManager 的资源配置和业务数据量、Kafka 分区数密切相关。
我们的经验公式是：

- **并行度和 Kafka 分区一致**（1 分区 = 1 并行度 = 1 slot）
- **CPU:Slot = 1:3**
- **CPU:内存 = 1:4**

比如有 20M/s 的流量，Kafka 3 个分区，我们就会配置 1 核 4G，3 个 slot 的 TaskManager。
平均每个 Flink 作业大概需要 **2 核 6G 内存**。
对于大流量日志分流 Job，TaskManager 会给到 8G 内存；而小流量的 DB 分流 Job，只需 4G。

## 4.3 Flink Task、Slot、并行度、算子关系

1、一个Task运行时需要占用一个Slot
2、每个 TaskManager 有 N 个 Slot，同一个 TaskManager 可以运行多个 Task（只要 Slot 足够）
3、Flink作业或算子并行度=Task数量
4、总 Task 数 = 并行度 × 算子数量（每个算子可能不同）
5、一个 Task 可以消费多个 Kafka Partition（取决于 Source 并行度与 Partition 数量）。
6、算子分类：Source算子（KafkaSource/FileSource），Transformation算子（Map/FlatMap/Filter），Sink算子（KafkaSink/JDBC Sink），算子是逻辑概念，Task是物理执行单元
7、一个算子可以有多个Task
8、算子链：为了减少 Task 数和网络开销，Flink 会把多个算子链到同一个 Task 上执行（前置条件：算子之间没有shuffle、并行度相同）

> **核心理解：**
>
> **算子**：逻辑处理单元（Source / Map / Filter / KeyBy / Sink）
> **算子链**：没有 shuffle 的算子可以合并成一个 Task
> **Task**：算子的物理执行单元，运行在 Slot 上
> **Slot**：TaskManager 的资源单元，每个 Task 占用一个 Slot
> **并行度**：Task 数量 = 算子并行度
> **Kafka Partition** 决定 Source Task 数据来源

**作业调度表示例：**

| TaskManager | Slot | Task编号 | Kafka Partition | 算子链 | 并行度 |
|---|---|---|---|---|---|
| TM1 | 0 | Task0 | P0 | Source → Map → Filter | 4 |
| TM1 | 1 | Task1 | P1 | Source → Map → Filter | 4 |
| TM1 | 2 | Task0 | shuffle分区0 | KeyBy + Sum | 2 |
| TM1 | 3 | Task0 | — | Sink | 2 |
| TM2 | 0 | Task2 | P2 | Source → Map → Filter | 4 |
| TM2 | 1 | Task3 | P3 | Source → Map → Filter | 4 |
| TM2 | 2 | Task1 | shuffle分区1 | KeyBy + Sum | 2 |
| TM2 | 3 | Task1 | — | Sink | 2 |

**数据流示例：**

Kafka Partitions: P0 P1 P2 P3
Source Tasks → Map + Filter (链成一个 Task)
| | | |
v v v v
KeyBy + Sum Tasks (并行度2, shuffle重分区)
| |
v v
Sink Tasks (并行度2)
|
MySQL

> 参考阅读：http://blog.csdn.net/be_racle/article/details/136049363

## 4.4 Flink任务并行度如何设置

> **并行度**：某个算子同时运行的 Task 数量。
> **全局并行度**：作用于整个作业，默认情况下，算子会继承这个并行度。
> **算子并行度**：可以单独指定某个算子的并行度，覆盖全局设置。

全局并行度设置和 kafka 分区数保持一致为 5，Keyby 后计算偏大的算子，单独指定算子的并行度。

## 4.5 Flink作业中的Checkpoint参数如何设置?

> Checkpoint作用：**容错恢复 + 精准一次性处理 + 节省回放**（能够保证正确从kafka offset处恢复）

Checkpoint 间隔：作业多久触发一次 Checkpoint，由 job 状态大小和恢复调整，大多数生产环境一般建议 3~5 分钟，时效性要求高的可以设置 s 级别。

Checkpoint 超时：限制 Checkpoint 的执行时间，超过此时间，Checkpoint 被丢弃，建议10 分钟。

Checkpoint 最小间隔：避免 Checkpoint 过于频繁，可以设置分钟级别。（设置最小间隔可以让作业有喘息时间）

Checkpoint 的执行模式：Exactly_once 或 At_least_once，选择 Exactly_once强一致性。

Checkpoint 的存储后端：一般存储 HDFS 。大状态就用RocksDB+HDFS

## 4.6 迟到数据如何解决？

（1）**设置乱序时间**：延迟一点再触发窗口计算，能吸收一部分乱序数据，但是会延迟计算结果
（2）窗口允许迟到时间
（3）侧输出流：迟到数据放入Kafka/HDFS→ 后续离线补算，或者ClickHouse/MySQL → 触发二次修正

生产中侧输出流，需要 Flink 单独处理，在写入 Clickhouse，通过接口再次计算

## 4.7 实时数仓延迟多少？

反压，算子逻辑复杂度，状态大小，资源偏少，机器性能，checkpoint 时间都会影响数仓延迟。

一般影响最大就是窗口大小，一般是 5s。

如果启用两阶段提交写入 Kafka，下游设置读已提交，那么需要加上 CheckPoint 间隔时间。

## 4.8 如何处理缓存冷启动问题？

缓存冷启动时，Redis 没数据会导致大量请求打到 HBase，类似缓存雪崩。实际解决方案包括：**离线任务+脚本来预热热门数据、Flink 内建本地缓存、异步批量加载和HBase限流保护、热点 key 单独兜底**，生产环境一般是"离线预热 + 本地缓存"结合来解决。

## 4.9 如何处理动态分流冷启动问题？（主流数据先到，丢失数据怎么处理）

> 在 Flink 实时作业里常见模式：
>
> - **主流（业务流）**：比如用户点击日志、订单事件
> - **配置流（Broadcast 流）**：动态规则、维度映射（如路由规则、过滤条件、分流配置）
>
> 作业逻辑：主流事件进来时，需要根据配置流规则进行分流或处理。
>
> **问题**：
>
> - 作业刚启动时，**主流数据先到**，但配置流的规则（Broadcast 状态）还没到；
> - 导致主流数据找不到对应的规则，可能会被丢弃或者错误路由。
> - 这就是 **动态分流冷启动问题**。

**（1）在 `open()` 方法中预加载配置**

- 在算子 `open()` 钩子初始化时，从外部存储（如 MySQL、Redis、HBase）一次性加载初始配置，存入 **HashMap** 或 **Operator State**；
- 这样即使 Broadcast 流的配置还没来，主流数据也能先用 **预加载的规则**处理，避免丢失。

**（2）主流数据暂存（Buffer 缓冲）**

- 如果配置流延迟较大，可以先把主流数据写入 **侧输出流** 或 **State 缓存**；
- 等到配置流（Broadcast）到来后，再重新处理这些缓存的主流数据。

👉 适合配置更新不频繁、数据对齐要求高的场景。

**（3）配置流优先启动**

- 在生产部署时，保证 **配置流任务先启动**，甚至先跑一个 **预热任务**，把配置写入 Redis / HBase；
- 主流任务启动时，`open()` 方法先去拉取 Redis 配置，再订阅 Broadcast 流做动态更新。

**（4）默认降级规则**

- 如果配置还没到，可以先走默认规则，比如写到一个 **“待处理” Topic / ClickHouse 表**，等后续再补算

## 4.10 代码升级，修改代码，如何上线？

Flink 作业上线或升级时，如果只是算子内部逻辑的小改动，可以先触发 Savepoint，停止旧作业，再用 Savepoint 恢复新作业，保留历史状态。

如果代码改动较大，Savepoint 无法直接恢复，需要判断历史数据是否必须保留：

- 必须保留 → 需要做状态迁移
- 不需要 → 可以直接提交新作业，从头处理数据

总结就是：**Savepoint 用于状态兼容的升级，状态不可兼容时就要看业务需求决定是否从头跑。**

## 4.11 如果现在做了 5 个 Checkpoint，Flink Job 挂掉之后想恢复到第三次 Checkpoint保存的状态上，如何操作？

Flink 默认 Checkpoint 是用于容错的，但会随作业取消被清理。
如果我们启用了 **Externalized Checkpoint**，Checkpoint 会保留在外部存储（HDFS/S3）。

如果想从第三次 Checkpoint 恢复作业，需要：

1. 找到第三次 Checkpoint 的目录（例如 `chk-3`）
2. 提交新作业时，使用 `-s` 参数指定这个 Checkpoint

作业启动后就会从第三次 Checkpoint 保存的状态继续执行，后续的第四、第五次 Checkpoint 会被忽略。

> 外部化 Checkpoint 必须提前开启，否则挂掉后无法恢复
>
> 如果 Checkpoint 状态不兼容新代码（比如算子状态结构变了），可能需要用 Savepoint 或做状态迁移

## 4.12 需要使用 flink 记录一群人，从北京出发到上海，记录出发时间和到达时间，同时要显示每个人用时多久，需要实时显示，如果让你来做，你怎么设计？

按照每个人 KeyBy把同一个的事件路由到同一个算子实例，将出发时间存入状态，当到达时使用到达时间减去出发时间。

## 4.13 Flink内部的数据质量和数据时效是怎么把控的？

在 Flink 作业中，**数据质量**通过以下方式保证：

- 脏数据定义以及通过 **侧输出流** 单独处理
- 使用 **KeyedState 去重**，保证唯一性
- 字段校验和规范化，保证数据完整性
- Checkpoint/Savepoint 提供容错和 Exactly-once 语义，Kafka 结合 Flink 的事务性写入保证 **端到端一致性**

**数据时效**通过以下方式保障：

- 使用 **事件时间 + Watermark** 控制窗口输出，处理乱序数据
- **迟到数据**可通过允许迟到和侧输出流处理

- 作业并行度和资源合理配置，避免反压
- Prometheus + Grafana实时监控作业吞吐和延迟，确保数据及时性

### 4.13 实时任务问题（延迟）怎么排查?

实时任务出现延迟时，可以从以下几个方面进行排查：

（1）监控指标：看是否反压

（2）日志信息：查看任务运行时的日志信息，定位潜在的问题和异常情况。例如，网络波动、硬件故障、不当的配置等等。

（3）外部事件：如果延迟出现在大量的外部事件后，则可能需要考虑其他因素（如外部系统故障、网络波动等）。框架混部，资源争抢!

### 4.14 维度数据并发量?

未做优化之前，有几千 QPS，做完 Redis 的缓存优化，下降到几十

> 把维度表数据预加载到 Redis（或者异步更新）
>
> Flink 任务里查 Redis → 内存级别查询，几百微秒就能拿到结果
>
> 只有缓存 miss 的时候才回源查 HBase/MySQL，并写回 Redis

### 4.15 Prometheus+Grafana 是自己搭的吗，监控哪些指标?

是我们自己搭建的，用来监控 Flink 任务和集群的相关指标

1）TaskManager Metrics：这些指标提供有关 TaskManager 的信息，例如 CPU 使用率、内存使用率、网络 IO 等。

2）Task Metrics：这些指标提供有关任务的信息，例如任务的延迟时间、记录丢失数、输入输出速率等。

3）Checkpoint Metrics：这些指标提供有关检查点的信息，例如检查点的持续时间、成功/失败的检查点数量、检查点大小等。

4）Operator Metrics：这些指标提供有关 Flink 操作符的信息，例如操作符的输入/输出记录数、处理时间、缓存大小等。

### 4.16 怎么在不停止任务情况下修改Flink参数?

在生产中，如果要在不停止任务的情况下修改 Flink 参数：

- 对于 **运行时配置（如 checkpoint 间隔、重启策略）**，可以通过 Flink Web UI 或 REST API 动态修改;
- 对于 **业务自定义参数（如规则、阈值）**，一般接入外部配置中心（Apollo、Nacos），Flink 作业在运行中定期拉取或监听配置变化，实现动态调整;
- 对于 **并行度和资源参数**，常用方式是 **Savepoint保存状态快照停掉旧任务 + 作业恢复**，这样可以平滑升级，不会丢失状态。

### 4.17 hbase中有表，里面的1月份到3月份的数据我不要了，我需要删除它（彻底删除），要这么做?

> 在 HBase 里，单纯用 delete 只是写入 tombstone 删除标记，真正物理删除要等 major compaction（HBase合并压缩）。
> 如果只是不要 1~3 月的数据，可以有两种方案：
>
> - 一种是通过 TTL 配置，让数据自动过期；
> - 另一种是建一个新表，把 4 月之后的数据迁过去，再 drop 掉旧表，这样能彻底清理掉不要的数据。
>   如果一定要彻底删除，最直接的办法就是 disable + drop 表，再新建。
>   生产里一般会结合数据备份，避免误删。

在 HBase 中彻底删除表中的数据，需要执行以下步骤：

（1）禁用表
（2）创建一个新表
（3）复制需要保留的数据，将需要保留的数据从旧表复制到新表。
（4）删除旧表
（5）重命名新表

在执行这些步骤之前，建议先进行数据备份以防止意外数据丢失。此外，如果旧表中的数据量非常大，复制数据到新表中的过程可能会需要很长时间。

### 4.18 如果 flink 程序的数据倾斜是偶然出现的，可能白天可能晚上突然出现，然后几个月都没有出现，没办法复现，怎么解决?

Flink 本身存在反压机制，短时间的数据倾斜问题可以自身消化掉，所以针对于这种偶然性数据倾斜，不做处理。

### 4.19 维度数据改变之后，如何保证新 join的维度数据是正确的数据?

（1）我们采用的是低延迟增量更新，本身就有延迟，没办法保证完全的正确数据。

（2）如果必须要正确结果，只能直接读取 MySQL 数据，但是需要考虑并发，MySQL机器性能

> 在实时场景下，维度数据会更新，如果不处理，join 到的可能是旧数据。
> 常见的做法有三种：
>
> - 一种是把维度数据放到 Redis，保证维度更新时 Redis 也更新，这样 Flink join 查 Redis 就是最新的；
> - 另一种是使用广播流，把维度表的变更实时推送到 Flink，每个算子都维护一份最新的维度状态；
> - 还有一种是定时全量刷新维度，适合更新不频繁的情况。
>   生产里最常用的就是 **Redis 缓存 + 广播流** 两种方案。

## 5.业务相关

### 5.1 数据采集到ODS层

#### （1）前端埋点的行为数据为什么又采集一份?

- **时效性原因**：前端埋点的数据有些直接进离线仓库（比如 Hive），但是那是批处理，延迟高。
- 如果要做实时统计（比如实时 UV、实时点击量），就要单独采集一份流式数据，进入 Kafka → Flink → 实时数仓。
- 这样离线和实时两份数据就都有了，能保证既能跑报表，也能实时展示。

👉 面试回答：

> 前端埋点数据会采集两份：一份进入离线数仓用于离线分析，另一份进入 Kafka 做实时计算，主要是为了满足实时性需求。

**（2）为什么选择 Kafka？**

- Kafka 是分布式消息队列，特点是 **高吞吐、低延迟、支持分区并行、支持实时写实时读。**
- 如果用 MySQL/HDFS 直接承接前端的实时写入，根本撑不住高并发。
- Kafka 正好能扛住高并发写入，并且下游（Flink、Spark）可以实时消费。

👉 面试回答：

> 我们选择 Kafka 作为采集层的承载，因为它支持高吞吐实时写入、分区并行消费，能够很好支撑实时数仓的高并发采集场景，而传统数据库没法承受这种实时写读压力。

**（3）为什么用 Maxwell？历史数据同步怎么保证一致性？**

- **背景**：要把 MySQL 里的业务表数据同步到 Kafka/Flink 里，常见工具有 Canal、Maxwell、FlinkCDC。
- **时间点**：FlinkCDC 在 2020 年 7 月才正式发布，很多项目之前都在用 Maxwell 或 Canal。
- **区别**：
  - Canal 偏向于只做增量 binlog 订阅；
  - Maxwell 除了 binlog，还能支持**历史全量数据导入**，而且它的位点存储在自己的元数据库里，可以断点续传；
  - Maxwell 输出的数据格式比较轻量（JSON），下游 Flink、Kafka 都好处理。
- **一致性**：Maxwell 保证至少一次，不会丢数据；如果下游做幂等处理（比如主键覆盖），就能做到最终一致。

👉 面试回答：

> 当时我们选 Maxwell，是因为 FlinkCDC 还没发布稳定版。相比 Canal，Maxwell 支持全量 + 增量同步，支持断点恢复，保证至少一次不丢数据，格式轻量，方便下游接入。

**（4）Kafka 保存多久？如果需要以前的数据怎么办？**

- Kafka 本质是个消息队列，不是数据库，保存数据的时间主要取决于磁盘和配置。
- 一般保存 3 天（这样磁盘占用不大，下游任务也有容错空间）。
- 如果需要更早的数据，有两个来源：
  - **离线仓库（Hive）**：历史的原始明细数据都会存在 Hive，可以查。
  - **ClickHouse**：实时明细或者宽表可能也有存储，可以查历史。

👉 面试回答：

> 我们 Kafka 配置保存 3 天，这样下游任务如果挂了还能回溯数据。磁盘也没压力，从原来 1T 扩到 2T 够用了。如果需要更早的数据，可以去 Hive 查离线的原始表，或者去 ClickHouse 查实时的宽表数据。

## 5.2 ODS层

> 2 个 topic：埋点的行为数据 ods_base_log、业务数据 ods_base_db

特点：**不做加工，原汁原味保存。**

1）**存储原始数据**

- 前端埋点行为数据 → 存到Kafka 的 `ods_base_log`
- 业务数据库（MySQL 等）的 binlog → Maxwell 采集，存到 Kafka 的 `ods_base_db`
  👉 这样 Kafka 就保存了两类原始数据。

2）**业务数据的有序性**

- Maxwell 采集 MySQL binlog 后发到 Kafka，如果不指定 key，可能会乱序。
- 通过配置 **分区 key = 表名**，保证同一张表的数据落到同一个分区，消费时有序。

## 5.3 DWD层+DIM层

DWD = 明细事实层
DIM = 维度层

**1）为什么维度表存 HBase？事实表存 Kafka？**

- **事实表（订单、支付、点击…）**：数据量大、只追加不更新 → 存 Kafka，方便流式计算。
- **维度表（用户、商品…）**：数据量相对小，但会频繁更新，要支持随机读写 → 存 HBase 更合适。
  👉 实时 join 时，可以从 Kafka 事实流 + HBase 维表，做维度补充。

**2）埋点行为数据分流**

Flink 读 `ods_base_log`，做拆分：

- **修复新老访客标识**：前端埋点打的标签可能不准，Flink 再次判断修正。
- **侧输出流分流**：按日志类型拆到不同 Topic：启动（start）、页面（page）、曝光（display）。
- **统计指标**：比如用户跳出率、独立访客数。

**3）业务数据处理**

- **动态分流**：
  FlinkSQL 消费 `ods_base_db`，把不同的表拆出来，写回不同 Kafka topic。
  （比如订单表写到 `dwd_order_info`，用户表写到 `dwd_user_info`）。
- **订单预处理表**：
  需要双流 join，比如订单表 + 订单详情表 → left join 得到宽表。
- **字典表维度退化**：
  比如性别（男/女），直接存成字符串，不用建维度表，减少 join。

**4）维度数据写入 HBase**

（1）**动态配置**

- 维度表变化怎么办？如果每次都重启 Flink Job，很麻烦。
- 方案：建一张配置表（在 MySQL），里面写清楚：
  - source（数据来源）
  - sink（写入哪里）
  - 操作类型（insert/update/delete）
  - 字段、主键...
- Flink 通过 **CDC + 广播状态** 机制实时同步这张配置表。
  👉 一旦配置变了，任务就能动态感知。

（2）**写 HBase 的方式**

- Flink 不能直接写 HBase → 借助 **Phoenix**，用 SQL 方式写 HBase，操作简单。

（3）**RowKey 设计 & 数据热点问题**

- 用户维表最大（假设 2000 万注册用户，每条 1KB，约 20GB）。
- RowKey 如果是自增 ID，容易热点 → 解决方法是 **加盐（Salt）**，即在 RowKey 前加随机前缀，打散写入。
- Phoenix 提供盐表机制。

## 5.4 DWS层

> DWS 层是汇总层，主要存宽表和轻度聚合后的数据，适合实时查询。
> 我们选择 ClickHouse，因为它擅长大宽表聚合计算，宽表 join 在 DWS 已经完成，查询快。
> 维度 join 采用 Async I/O + 旁路缓存优化 HBase 查询，异步查询减少阻塞，缓存命中减少重复查库，同时通过删除缓存或双写保证缓存一致性。
> 为了减轻 ClickHouse 写入压力和查询压力，会做 5 秒滚动窗口的轻度聚合，把数据整理成访客、商品、地区、关键词等宽表。

**1）为什么选择 ClickHouse**

- **大宽表、数据量多、聚合统计分析快**
  - DWS 层的数据通常是汇总后的宽表（例如订单 + 用户 + 商品 + 地区维度），字段多、行多。
  - ClickHouse 擅长列式存储 + 聚合计算，能快速查询和统计。
- **宽表已经不再需要 Join**
  - 在 DWS 层，维度数据多数已经在前面 DWD/DIM 层 join 好了，数据是宽表，查询时不再做复杂 join，ClickHouse 适合直接聚合查询。

**2）关联维度数据**

**维度关联方案**

- **预加载**：把维度表加载到内存（广播流）
- **读取外部数据库**：直接查 HBase、Redis
- **双流 Join / LookupJoin**：实时流 + 维度流 join

**项目实践**

- Flink 读取 **HBase 中的维度数据**

**优化方案 1：异步 IO（Async I/O）**

- **问题**：直接查 HBase 时，如果每条数据都阻塞等待，吞吐量低。
- **解决**：异步 IO，把维表查询托管到线程池，单个并行可以同时发送多个请求，哪个先返回就先处理。
- **效果**：减少网络延迟阻塞，提高 Flink 流处理效率。
- **Flink 版本**：从 1.2 开始支持 Async I/O，这是阿里贡献的特性。

**优化方案 2：旁路缓存（Cache Aside）**

- **流程**：
  1. 先查缓存（Redis），命中就返回
  2. 如果未命中，去 HBase 查，再写入缓存，方便下次查
- **缓存一致性**：
  - **方案 1**：维表更新时（update），同时删除 Redis 旧值 + 设置 24h 过期
  - **方案 2**：双写（HBase + Redis 同步更新）

**3）轻度聚合**

- **为什么**：DWS 层要处理大量实时查询，如果都是明细表，查询压力太大。
- **做法**：
  - 将实时数据按主题组合（宽表）
  - 减少维度查询次数
  - 开小窗口（如 5 秒滚动窗口）进行轻度聚合
  - 减轻写 ClickHouse 压力，减少后续聚合时间
- **涉及表**：访客、商品、地区、关键词
- **字段**：根据业务统计需求设计，都是宽表字段

## 5.5 ADS层

**1）实现方案**

为可视化大屏服务，提供一个数据接口用来查询 ClickHouse 中的数据。

**2）怎么保证 ClickHouse 的一致性?**

ReplacingMergeTree 只能保证最终一致性，查询时的 sql 语法加上去重逻辑。

> **ReplacingMergeTree**：ClickHouse 的一种表引擎，支持按主键更新数据，但只保证**最终一致性**（最终只保留最新一条数据）。
>
> **问题**：在写入过程中，可能会出现重复记录或旧数据没清理。
>
> **解决方案**：
>
> - 查询时加 `GROUP BY` 或 `DISTINCT` 等去重逻辑
> - 确保最终返回结果准确

# 3、用户画像

## 3.1 画像系统主要做的事情

1. **用户信息标签化**
   - 将用户的各种属性、行为、偏好做成标签。
   - 例如：性别、年龄段、购买偏好、浏览偏好等。
2. **标签数据的应用**
   - **用户分群**：按标签组合划分人群（例如女性 + 90 后）。
   - **洞察分析**：分析用户行为、偏好，支持运营和决策。
3. **标签建模**
   - 四级标签结构：
     1. 一级、二级：分类（如人口属性、行为属性）
     2. 第三级：具体标签（性别、年龄段）
     3. 第四级：标签值（女性、90 后）

## 3.2 项目整体架构

**数据仓库 → 用户画像管理平台 → 外部应用**

- **数据仓库**：抽取、计算、重组、导出，存储标签基础数据
- **画像平台**：
  - 标签规则定义
  - 标签任务调度（自动/手动）
  - 任务监控
  - 分群管理、用户标签分析
- **技术栈**：
  - 前端：Vue.js
  - 后端：SpringBoot + MySQL
  - 计算：Spark + SparkMLlib + Yarn
  - 存储：ClickHouse、Elasticsearch、HBase、Redis

## 3.3 标签计算调度过程

1. 在画像平台填写标签任务（任务定义 + SQL + 参数）
2. 上传 Spark 程序 jar 到 HDFS
3. 平台通过 `spark-submit` 提交任务到 Yarn 集群
4. Spark 作业加载 jar，读取任务定义和规则
5. 计算完成后写入 Hive 画像库
6. 平台跟踪作业状态，回调作业执行结果

## 3.4 标签批处理流程

四个主要任务：

1. **生成标签单表**：每个标签独立计算，生成单表
2. **合并为标签宽表**：将各标签单表合并成用户维度宽表
3. **导出到 ClickHouse**：方便实时查询和下游服务调用
4. **转储为 Bitmap 表**：提高多标签交集运算性能

- **流程特点**：
  - 新增标签只需在平台填写定义和 SQL
  - Spark 程序自动完成计算
  - 平台调度统一管理

## 3.5 画像平台功能

1. 标签定义
2. 标签任务设定
3. 任务调度
4. 任务监控
5. 分群创建与维护
6. 用户洞察分析

## 3.6 Web 应用开发

- **画像平台分群功能**：根据标签组合进行人群划分
- **实时数据接口**：从 ClickHouse 查询数据给业务系统或广告系统使用

## 3.7 画像平台上下游

- **上游**：数仓系统，提供计算好的标签数据
- **下游**：Redis，供广告系统、运营系统访问

## 3.8 Bitmap 原理与性能优势

- **原理**：使用二进制数组表示用户是否属于某个标签
  - 0 = 不存在，1 = 存在
- **优势**：
  - 求交集只需做位运算（AND），无需遍历所有用户
  - 复杂度从 $O(N^2)$ 降到 $O(N)$，大幅提高性能
- **应用**：用户分群（如女性 + 90 后）通过 Bitmap 交集快速筛选

# 4、数据湖项目

## 4.1 数据湖 vs 数据仓库

| 特性 | 数据仓库 | 数据湖 |
| --- | --- | --- |
| 数据类型 | 结构化、清洗后 | 原始数据（结构化、半结构化、非结构化） |
| 处理方式 | ETL 批量 | 批 + 流、增量、全量 |
| 存储成本 | 较高 | 较低，可海量存储 |
| 使用场景 | 报表、分析 | 数据探索、建模、机器学习、实时分析 |
| 典型技术 | Hive、ClickHouse、Doris | Hudi、Iceberg、DeltaLake |

- 数据湖是企业存储**各种原始数据**的大仓库，可供处理、分析、传输。
- Hudi/Iceberg 支持**增量处理、Upsert**，适合流批一体场景。

## 4.2 为什么做这个项目 / 解决痛点

### 1️⃣ 离线数仓痛点

- **时效性差**：T+1 批处理模式
- **数据更新困难**：只能 overwrite，资源消耗大

### 2️⃣ 实时数仓痛点

- **数据一致性维护麻烦**
- **历史数据修正复杂**：没有明细持久化，流程繁琐，需要重跑

### 3️⃣ 传统数仓发展方向

- **流批一体**：一套架构、一套代码，既能跑批也能跑流
- **优势**：节省资源和人力

### 4️⃣ Hudi 数据湖优势

- 实时性提升到分钟级别（5~10 分钟）
- 支持增量处理
- 数据更新支持 **Upsert**

> 总结：目标是**融合数据湖和数据仓库**，在数据湖低成本存储基础上，继承数仓的处理和管理能力。

## 4.3 项目架构

**湖仓一体核心架构**

```
业务系统
├─ MySQL / 业务数据
├─ Log 文件 / 日志数据
|
├─ Flume / Kafka
|
└─ FlinkCDC 同步
      ├─ ODS（原始数据层）
      ├─ DWD（明细层）
      ├─ DIM（维度层）
      ├─ DWS（宽表聚合层）
      └─ ADS / 可视化查询接口
```

- 技术栈：FlinkCDC、Hudi、Hive、ClickHouse、Kafka、Flume
- 支持流批统一处理，ODS → DWD/DIM → DWS → 可视化查询

## 4.4 业务

- 与实时数仓保持一致
- 支持批量分析和实时查询

## 4.5 遇到的问题及优化

### 1️⃣ 断点续传采集

- FlinkCDC 分全量 + binlog
- 依赖 Flink **state** 存储进度
- **恢复机制**：失败后，从 state 恢复，不重复采集

### 2️⃣ 写 Hudi 表数据倾斜

- 问题：全量阶段一张表写完再写下一张表，下游多个 Sink 可能只有一个有数据
- 解决方案：**多表混合读取**，同时写多个 Sink

### 3️⃣ 大状态处理

- 问题：regular join + 无 TTL，会导致 RocksDB 状态膨胀
- 解决方案：RocksDB + 增量更新，避免全量状态

### 4️⃣ Hudi 优化

- **MOR 表**：离线 compaction，不绑定写入
- 并发优化：Compaction / write 并发 = 4
- 内存配置：Compaction = 1G

### 5️⃣ Hudi 二期规划

- **大状态优化**：部分列更新方案，避免全表 join
  - Hudi <= 0.12 需要自定义 Payload 类
  - Hudi 0.13 已原生支持
- **DWS 层升级**：ClickHouse 存储明细宽表，支持自助分析

- 目标：未来实时项目可直接读取 Hudi → ClickHouse，不再依赖复杂流处理

# 5、数据治理

数据治理是企业为了保证数据的**可靠性、可用性、一致性和安全性**，在数据全生命周期中实施的一系列管理和技术措施。

核心模块包括：

1. 元数据管理
2. 数据质量监控
3. 权限管理
4. 用户认证
5. 数据资产健康度量

## 5.1 元数据管理（Metadata Management）

**作用**：记录数据资产信息（表、字段、血缘关系等），便于**影响分析、异常排查、数据血缘管理。**

### 实现方式

1. **开源框架**：Apache Atlas
2. **自研系统**：公司内部元数据管理工具

### 底层原理

- 对数据处理作业进行解析（如 HQL/SQL）：

```
insert into table ads_user
select id, name from dws_user
```

- 可获取：
  - **表级依赖**：ads_user 依赖 dws_user
  - **字段级依赖**：id、name 字段关联
- 存储方式常用 **Neo4j** 实现图结构血缘

### 用途

- 当作业失败时，快速评估影响范围
- 支持版本管理和数据血缘查询

## 5.2 数据质量监控（Data Quality Monitoring）

数据质量监控确保数据符合**准确性、完整性、唯一性、及时性、一致性**等指标。

### 5.2.1 实现步骤

1. **确定数据源**
   - 数据库表、文件、API 等
2. **定义质量规则**

| 维度 | 示例规则 |
| --- | --- |
| 准确性 | 数据值在合理范围或符合预期分布 |
| 完整性 | 字段非空或无缺失值 |
| 唯一性 | 主键或唯一字段无重复 |
| 及时性 | 数据按预期时间更新 |
| 一致性 | 格式符合标准，如日期格式统一 |

3. **实现检测功能**
    - 数据导入
    - 数据清理（去空格、类型转换）
    - 应用规则检测（缺失值、重复值、范围检查）
    - 输出报告（表格或可视化图表）
4. **自动化与监控**
    - 集成到 ETL/数据管道中
    - 设置报警机制：发现异常及时通知

## 5.2.2 监控原则

**1. 单表数据量监控**

- SQL 示例：`SELECT COUNT(*) FROM table WHERE date='2025-09-21'`
- 报警条件：
    - 数据量不在 [下限, 上限]
    - 环比/同比偏离过大

**2. 单表空值检测**

- SQL 示例：`SELECT COUNT(*) FROM table WHERE field IS NULL`
- 异常数据量超阈值触发报警

**3. 单表重复值检测**

- SQL 示例：

```
SELECT COUNT(*) - COUNT(DISTINCT key_field) AS duplicate_count
FROM table
```

- 超阈值触发报警

**4. 单表值域检测**

- 检查字段值是否在允许范围
- 异常触发报警

**5. 跨表数据量对比**

- 检查同步流程一致性
- SQL 示例：`SELECT COUNT(*) FROM table1 - COUNT(*) FROM table2`

## 5.2.3 数据质量实现示例

- 电商数仓项目采用 **SQL/HQL** 脚本 + 数据质量监控平台（如 QDC）
- 自动化执行，定期生成报告

## 7.3 权限管理（Ranger）

- 对数据访问进行**授权、审计、监控**
- 控制表级、列级、行级访问
- 解决数据安全合规性问题

## 7.4 用户认证（Kerberos）

- 确保**用户身份合法**
- 通过票据机制保护集群访问
- 防止非授权访问

## 7.5 数据资产健康度量（Data Asset Health Score）

量化企业数据资产健康状况，用于评估**数据质量、规范性、安全性、可用性。**

### 1️⃣ 基本逻辑

- 健康分采用 **百分制（0-100）**
- 最小粒度为表，每个表都有健康分
- 个人、团队、业务部门健康分为所属表加权平均

### 2️⃣ 计算公式

```
表资产健康分 score = 规范合规*10% + 存储健康*30% + 计算健康*30% + 数据质量*15% + 数据安全*15%
```

### 3️⃣ 健康分维度示例

| 维度 | 特征 | 评分规则 |
| --- | --- | --- |
| 规范 | 有技术 owner、业务 owner、表命名合规、分区信息等 | 0/1，按完成度加权 |
| 存储 | 生命周期合理性、访问情况、冷备表等 | 0-100% |
| 计算 | HDFS 路径存在、产出非空、任务运行正常、无重复表、避免暴力扫描 | 0/1 |
| 数据质量 | 表产出时效监控、字段内容监控、SLA 达成率 | 0-100% |

| 维度 | 特征 | 评分规则 |
| --- | --- | --- |
| 安全 | 数据分类、资产分级、字段安全等级 | 0-100% |

> 核心思想：通过量化每个表的指标，形成**资产健康度评分体系**，便于团队管理和监控数据质量风险。

### 面试答题模板

> 数据治理主要保证数据**可靠、可用、安全、一致**。
> 包含：
>
> 1. 元数据管理：用 Atlas 或自研系统记录表、字段、血缘信息，便于影响分析和异常排查。
> 2. 数据质量监控：定义准确性、完整性、唯一性、及时性、一致性规则，通过 SQL/HQL 或 ETL 集成检测，异常触发报警。
> 3. 权限管理和用户认证：Ranger + Kerberos 确保访问合规。
> 4. 数据资产健康度量：根据规范、存储、计算、数据质量、安全五大维度，给每张表打分形成资产健康分，支持团队和业务部门监控和管理。

# 十二、算法题

## 一、复杂度分析

### 1. 时间复杂度

- 衡量运行时间随 **n** 增长的趋势。
- 常见复杂度（快 → 慢）：
  - **O(1)** → 哈希查找、位运算。
  - **O(log n)** → 二分查找。
  - **O(n)** → 遍历数组、链表。
  - **O(n log n)** → 排序（快排、归并）。
  - **O(n²)** → 双重循环，冒泡。
  - **O(2^n)** → 子集、递归枚举。
  - **O(n!)** → 全排列。

### 2. 空间复杂度

- 衡量程序所需额外空间。
- 常见：
  - O(1) → 原地排序。
  - O(n) → 辅助数组、递归。
  - O(n²) → DP 二维表。

👉 面试套路：答题时说明 **最坏情况** 和 **是否可优化**。

---

# 二、常见解题思想

1. **暴力枚举**
   全部尝试，保证正确但效率低。
   例：全排列、最长子串。
2. **贪心**
   每步取局部最优，期望得到全局最优。
   例：区间调度、最少会议室、哈夫曼编码。
3. **分治**
   大问题拆成子问题 → 合并解。
   例：快排、归并、二分查找。
4. **动态规划（DP）**
   存储子问题解，避免重复计算。
   - 自顶向下（记忆化递归）。
   - 自底向上（迭代 DP）。
     例：最长公共子序列、背包问题。
5. **回溯（DFS）**
   穷举所有解，遇到不合法就剪枝。
   例：N 皇后、IP 地址划分。
6. **广度优先搜索（BFS）**
   层次遍历，常用于最短路径。
   例：迷宫最短路径、单词接龙。
7. **分支限界**
   搜索时加约束，减少无效遍历。
   例：整数规划、约束搜索问题。

---

# 三、常见常考算法题（分类）

## 1. 排序

- **冒泡排序**：$O(n^2)$，简单但慢。
- **选择排序**：$O(n^2)$，不稳定。
- **插入排序**：$O(n^2)$，数据接近有序时快。
- **归并排序**：$O(n \log n)$，稳定。
- **快速排序**：$O(n \log n)$，最坏 $O(n^2)$。
- **堆排序**：$O(n \log n)$，不稳定，基于堆。

👉 面试常问：**快排 vs 归并** 区别。

## 2. 查找

- **二分查找**（有序数组）→ $O(\log n)$
- **二分变种**：
  - 第一个 >= target
  - 最后一个 <= target

- **哈希查找** → O(1) 平均复杂度。

## 3. 数组 & 字符串

1. **两数之和**（哈希表 O(n)）。
2. **最长子串**（滑动窗口 O(n)）。
3. **最长回文子串**（中心扩展 $O(n^2)$，Manacher O(n)）。
4. **字符串匹配**：KMP（O(n+m)）。
5. **数组去重**（快慢指针 O(n)）。
6. **旋转数组最小值**（二分 O(log n)）。
7. **盛最多水的容器**（双指针 O(n)）。

## 4. 链表

1. **反转链表**（迭代 O(n) / 递归 O(n)）。
2. **链表环检测**（快慢指针）。
3. **合并两个有序链表**（双指针）。
4. **LRU 缓存**（哈希表 + 双向链表）。

## 5. 栈 & 队列

1. **最小栈**（辅助栈记录最小值）。
2. **用栈实现队列 / 用队列实现栈。**
3. **滑动窗口最大值**（单调队列）。

## 6. 二叉树

1. 遍历：前序 / 中序 / 后序 / 层序（递归 & 非递归）。
2. **最大深度 / 最小深度**（递归）。
3. **是否对称**（递归 + 队列）。
4. **二叉搜索树（BST）**：
    - 中序遍历有序。
    - 判断合法性。
    - 最近公共祖先（LCA）。

## 7. 图论

1. **BFS**：最短路径（O(V+E)）。
2. **DFS**：拓扑排序、连通分量。
3. **最短路径算法**：Dijkstra、Floyd。
4. **最小生成树**：Prim、Kruskal。

### 8. 动态规划经典题

1. **斐波那契数列**（青蛙跳台阶）。
2. **爬楼梯**（f(n)=f(n-1)+f(n-2)）。
3. **最长公共子序列（LCS）**。
4. **最长上升子序列（LIS）**。
5. **背包问题**：
   - 01 背包。
   - 完全背包。
6. **编辑距离**。
7. **最大子数组和**（Kadane 算法 O(n)）。

### 9. 回溯 & 搜索

1. **全排列 / 子集 / 组合**。
2. **N 皇后**。
3. **数独求解**。
4. **电话号码字母组合**。
5. **IP 地址划分**。

### 10. 高频面试题

1. **两数之和 / 三数之和**。
2. **接雨水问题**（单调栈 / 双指针）。
3. **区间合并**。
4. **旋转图像**（矩阵转置 + 翻转）。
5. **岛屿数量**（DFS/BFS）。
6. **单词接龙**（BFS）。

## 四、答题套路总结

面试答算法题时，建议这样说：

1. **先描述暴力解法**（保证思路正确）。
2. **再提出优化思路**（比如用 DP / 双指针 / 二分）。
3. **分析复杂度**（时间 & 空间）。
4. **提到边界条件**（空输入、极值、大数据）。

## 五、完整题解

### 1.排序

#### 🔹 1. 冒泡排序 (Bubble Sort)

```
public static void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
```

```
        boolean swapped = false; // 优化：如果一轮没有交换，说明已经有序
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
```

👉 **思路**：每轮把最大的数"冒泡"到最后。
时间复杂度：O(n²)，最好 O(n)（已排序时）。

---

### 🔹 2. 选择排序 (Selection Sort)

```
public static void selectionSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}
```

👉 **思路**：每轮选择剩余部分的最小值，放到前面。
时间复杂度：O(n²)，不稳定。

---

### 🔹 3. 插入排序 (Insertion Sort)

```
public static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

👉 **思路**：把当前元素插入到前面已经有序的部分。
适合「基本有序」的情况，O(n)。最坏 O(n²)。

### 🔹 4. 归并排序 (Merge Sort)

```
public static void mergeSort(int[] arr, int left, int right) {
    if (left >= right) return;
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

private static void merge(int[] arr, int left, int mid, int right) {
    int[] temp = new int[right - left + 1];
    int i = left, j = mid + 1, k = 0;

    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) temp[k++] = arr[i++];
        else temp[k++] = arr[j++];
    }
    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];
    System.arraycopy(temp, 0, arr, left, temp.length);
}
```

👉 **思路**：分治 → 拆分到单个元素 → 合并两个有序数组。
时间 O(n log n)，稳定。空间 O(n)。

---

### 🔹 5. 快速排序 (Quick Sort)

```
public static void quickSort(int[] arr, int left, int right) {
    if (left >= right) return;
    int pivot = partition(arr, left, right);
    quickSort(arr, left, pivot - 1);
    quickSort(arr, pivot + 1, right);
}

private static int partition(int[] arr, int left, int right) {
    int pivot = arr[right]; // 选最后一个元素作为基准
    int i = left - 1;
    for (int j = left; j < right; j++) {
        if (arr[j] <= pivot) {
            i++;
            int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
        }
    }
    int temp = arr[i + 1]; arr[i + 1] = arr[right]; arr[right] = temp;
    return i + 1;
}
```

👉 **思路**：分治 → 选基准 → 小的放左边，大的放右边。
平均 O(n log n)，最坏 O($n^2$)，不稳定。

---

### 🔹 6. 堆排序 (Heap Sort)

```
public static void heapSort(int[] arr) {
    int n = arr.length;
    // 建大顶堆
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
    // 依次取出堆顶元素
    for (int i = n - 1; i > 0; i--) {
        int temp = arr[0]; arr[0] = arr[i]; arr[i] = temp;
        heapify(arr, i, 0);
    }
}

private static void heapify(int[] arr, int n, int i) {
    int largest = i;
    int l = 2 * i + 1, r = 2 * i + 2;
    if (l < n && arr[l] > arr[largest]) largest = l;
    if (r < n && arr[r] > arr[largest]) largest = r;
    if (largest != i) {
        int temp = arr[i]; arr[i] = arr[largest]; arr[largest] = temp;
        heapify(arr, n, largest);
    }
}
```

👉 **思路**：利用堆（大顶堆/小顶堆）的性质，每次取堆顶。
时间 O(n log n)，空间 O(1)，不稳定。

---

### ⚖️ 快排 vs 归并

- **快排**：原地排序，空间 O(1)，平均快，但最坏退化 $O(n^2)$。
- **归并**：稳定，最坏也能保证 O(n log n)，但需要额外 O(n) 空间。

## 2.查找

### 🔹 1. 二分查找（Binary Search）：

- 思路：对有序数组，每次取中间元素和目标比较，缩小一半区间。
- 时间复杂度：O(log n)，空间复杂度：O(1)。

```
public class BinarySearch {
    // 标准二分查找：找到 target 的索引，找不到返回 -1
    public static int binarySearch(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2; // 防止溢出
            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
```

```
        }
        return -1;
    }
}
```

---

🔹 **2. 二分查找变种**：

1）查找第一个 >= target 的位置（lower bound）。

- 应用：求区间、找插入点。

```
public static int lowerBound(int[] nums, int target) {
    int left = 0, right = nums.length; // 注意这里 right = length
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid; // 收缩右边
        }
    }
    return left; // 返回第一个 >= target 的索引
}
```

2）查找最后一个 <= target 的位置（upper bound - 1）。

- 应用：找区间右边界。

```
public static int upperBound(int[] nums, int target) {
    int left = 0, right = nums.length;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] <= target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left - 1; // 返回最后一个 <= target 的索引
}
```

---

🔹 **3. 哈希查找（Hash Search）**：

- 思路：利用哈希表的 **O(1) 平均查找时间。**
- 场景：查找是否存在、统计频率。

```
import java.util.HashMap;

public class HashSearch {
    public static void main(String[] args) {
        int[] nums = {1, 3, 5, 7};
        int target = 5;
```

```
        // 用 HashMap 存储值 -> 索引
        HashMap<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            map.put(nums[i], i);
        }

        if (map.containsKey(target)) {
            System.out.println("找到索引：" + map.get(target));
        } else {
            System.out.println("未找到");
        }
    }
}
```

---

解题思路总结：

1. **二分查找**：用于有序数组，适合静态数据查询。常见变种是找左右边界。
2. **哈希查找**：用于快速存在性判断，典型应用是两数之和、频率统计。
3. 面试常考：二分的边界处理（`left <= right` vs `left < right`），以及 lowerBound / upperBound 写法。

## 3.数组和字符串

### 🔹 1. 两数之和 (Two Sum)

- 思路：用哈希表记录已经访问过的数字及索引，遍历数组，检查 `target - nums[i]` 是否存在。
- 时间复杂度：O(n)，空间复杂度：O(n)。

```
import java.util.HashMap;

public class TwoSum {
    public static int[] twoSum(int[] nums, int target) {
        HashMap<Integer, Integer> map = new HashMap<>();
        for(int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if(map.containsKey(complement)) {
                return new int[]{map.get(complement), i};
            }
            map.put(nums[i], i);
        }
        return new int[]{};
    }
}
```

---

### 🔹 2. 最长子串 (Longest Substring Without Repeating Characters)

- 思路：滑动窗口 + 哈希表/Set，维护窗口内不重复字符。
- 时间复杂度：O(n)，空间复杂度：O(min(n, charset))。

```
import java.util.HashSet;

public class LongestSubstring {
```

```
    public static int lengthOfLongestSubstring(String s) {
        HashSet<Character> set = new HashSet<>();
        int left = 0, maxLen = 0;
        for(int right = 0; right < s.length(); right++) {
            while(set.contains(s.charAt(right))) {
                set.remove(s.charAt(left++));
            }
            set.add(s.charAt(right));
            maxLen = Math.max(maxLen, right - left + 1);
        }
        return maxLen;
    }
}
```

---

### 🔹 3. 最长回文子串 (Longest Palindromic Substring)

- 思路：中心扩展法，枚举每个字符作为中心，向两边扩展。
- 时间复杂度：O(n²)，空间复杂度：O(1)。

```
public class LongestPalindrome {
    public static String longestPalindrome(String s) {
        if(s == null || s.length() < 1) return "";
        int start = 0, end = 0;
        for(int i = 0; i < s.length(); i++) {
            int len1 = expandAroundCenter(s, i, i);        // 奇数回文
            int len2 = expandAroundCenter(s, i, i + 1);    // 偶数回文
            int len = Math.max(len1, len2);
            if(len > end - start) {
                start = i - (len - 1) / 2;
                end = i + len / 2;
            }
        }
        return s.substring(start, end + 1);
    }

    private static int expandAroundCenter(String s, int left, int right) {
        while(left >= 0 && right < s.length() && s.charAt(left) ==
s.charAt(right)) {
            left--;
            right++;
        }
        return right - left - 1;
    }
}
```

---

### 🔹 4. 字符串匹配 (KMP)

- 思路：先构建部分匹配表（next数组），然后匹配过程中遇到不匹配直接跳过部分前缀。
- 时间复杂度：O(n+m)，空间复杂度：O(m)。

```
public class KMP {
    public static int kmpSearch(String text, String pattern) {
```

```
        int[] lps = buildLPS(pattern);
        int i = 0, j = 0;
        while(i < text.length()) {
            if(text.charAt(i) == pattern.charAt(j)) {
                i++; j++;
            }
            if(j == pattern.length()) return i - j;
            else if(i < text.length() && text.charAt(i) != pattern.charAt(j)) {
                if(j != 0) j = lps[j - 1];
                else i++;
            }
        }
        return -1;
    }

    private static int[] buildLPS(String pattern) {
        int[] lps = new int[pattern.length()];
        int len = 0, i = 1;
        while(i < pattern.length()) {
            if(pattern.charAt(i) == pattern.charAt(len)) {
                lps[i++] = ++len;
            } else {
                if(len != 0) len = lps[len - 1];
                else lps[i++] = 0;
            }
        }
        return lps;
    }
}
```

---

### 🔹 5. 数组去重 (Remove Duplicates from Sorted Array)

- 思路：快慢指针，慢指针指向最后一个不重复元素位置，快指针遍历。
- 时间复杂度：O(n)，空间复杂度：O(1)。

```
public class RemoveDuplicates {
    public static int removeDuplicates(int[] nums) {
        if(nums.length == 0) return 0;
        int slow = 0;
        for(int fast = 1; fast < nums.length; fast++) {
            if(nums[fast] != nums[slow]) {
                nums[++slow] = nums[fast];
            }
        }
        return slow + 1; // 新数组长度
    }
}
```

---

### 🔹 6. 旋转数组最小值 (Find Minimum in Rotated Sorted Array)

- 思路：二分查找，判断 mid 与 right 的大小决定下一步缩区间。
- 时间复杂度：O(log n)，空间复杂度：O(1)。

```
public class FindMin {
    public static int findMin(int[] nums) {
        int left = 0, right = nums.length - 1;
        while(left < right) {
            int mid = left + (right - left) / 2;
            if(nums[mid] > nums[right]) left = mid + 1;
            else right = mid;
        }
        return nums[left];
    }
}
```

---

### 🔹 7. 盛最多水的容器 (Container With Most Water)

- 思路：双指针，从两边开始，移动较小高度指针，计算面积并更新最大值。
- 时间复杂度：O(n)，空间复杂度：O(1)。

```
public class MaxWater {
    public static int maxArea(int[] height) {
        int left = 0, right = height.length - 1, maxArea = 0;
        while(left < right) {
            int h = Math.min(height[left], height[right]);
            int w = right - left;
            maxArea = Math.max(maxArea, h * w);
            if(height[left] < height[right]) left++;
            else right--;
        }
        return maxArea;
    }
}
```

## 4.链表

### 🔹 1. 反转链表 (Reverse Linked List)

- **思路**：
  - **迭代**：逐步改变 `next` 指针，保存前驱节点。
  - **递归**：递归到末尾节点，逐层返回时翻转指针。
- **时间复杂度**：O(n)，**空间复杂度**：迭代 O(1)，递归 O(n)。

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int val){ this.val = val; }
}

public class ReverseLinkedList {
    // 迭代
    public static ListNode reverseIter(ListNode head) {
        ListNode prev = null, curr = head;
        while(curr != null) {
            ListNode next = curr.next;
```

```
            curr.next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }

    // 递归
    public static ListNode reverseRec(ListNode head) {
        if(head == null || head.next == null) return head;
        ListNode newHead = reverseRec(head.next);
        head.next.next = head;
        head.next = null;
        return newHead;
    }
}
```

---

### 🔹 2. 链表环检测 (Linked List Cycle Detection)

- **思路**：使用快慢指针，如果链表有环，快指针最终会追上慢指针。
- **时间复杂度**：O(n)，**空间复杂度**：O(1)。

```
public class LinkedListCycle {
    public static boolean hasCycle(ListNode head) {
        if(head == null) return false;
        ListNode slow = head, fast = head;
        while(fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if(slow == fast) return true;
        }
        return false;
    }
}
```

---

### 🔹 3. 合并两个有序链表 (Merge Two Sorted Lists)

- **思路**：使用双指针，依次比较两个链表节点，连接小的节点。
- **时间复杂度**：O(n+m)，**空间复杂度**：O(1)。

```
public class MergeSortedList {
    public static ListNode merge(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode tail = dummy;
        while(l1 != null && l2 != null) {
            if(l1.val < l2.val) {
                tail.next = l1;
                l1 = l1.next;
            } else {
                tail.next = l2;
                l2 = l2.next;
            }
```

```
            tail = tail.next;
        }
        tail.next = (l1 != null) ? l1 : l2;
        return dummy.next;
    }
}
```

---

### 🔹 4. LRU 缓存 (LRU Cache)

- **思路**：
  - 使用 **哈希表 + 双向链表** 实现 O(1) 的 `get` 和 `put`。
  - 哈希表存 key->节点映射，双向链表维护最近使用顺序。

```
import java.util.*;

class LRUCache {
    class Node {
        int key, value;
        Node prev, next;
        Node(int k, int v) { key = k; value = v; }
    }

    private Map<Integer, Node> map;
    private Node head, tail;
    private int capacity;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        map = new HashMap<>();
        head = new Node(0,0);
        tail = new Node(0,0);
        head.next = tail;
        tail.prev = head;
    }

    private void remove(Node node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }

    private void addToFront(Node node) {
        node.next = head.next;
        node.prev = head;
        head.next.prev = node;
        head.next = node;
    }

    public int get(int key) {
        if(!map.containsKey(key)) return -1;
        Node node = map.get(key);
        remove(node);
        addToFront(node);
        return node.value;
```

```
    }

    public void put(int key, int value) {
        if(map.containsKey(key)) {
            Node node = map.get(key);
            node.value = value;
            remove(node);
            addToFront(node);
        } else {
            if(map.size() == capacity) {
                Node toRemove = tail.prev;
                remove(toRemove);
                map.remove(toRemove.key);
            }
            Node node = new Node(key, value);
            addToFront(node);
            map.put(key, node);
        }
    }
}
```

## 5.栈 & 队列

### 🔹 1. 最小栈 (Min Stack)

- **思路**：
  - 使用两个栈：一个存数据，一个存每个状态下的最小值。
  - `push` 时将当前值与辅助栈栈顶最小值比较，更新辅助栈。
  - `pop` 时两个栈同时弹出。
- **时间复杂度**：O(1)，**空间复杂度**：O(n)。

```
import java.util.Stack;

class MinStack {
    private Stack<Integer> stack = new Stack<>();
    private Stack<Integer> minStack = new Stack<>();

    public void push(int x) {
        stack.push(x);
        if(minStack.isEmpty() || x <= minStack.peek()) {
            minStack.push(x);
        } else {
            minStack.push(minStack.peek());
        }
    }

    public void pop() {
        stack.pop();
        minStack.pop();
    }

    public int top() {
        return stack.peek();
    }
```

```
    public int getMin() {
        return minStack.peek();
    }
}
```

---

### 🔹 2. 用栈实现队列 (Queue via Stacks)

- **思路**：
  - 使用两个栈 `in` 和 `out`，`in` 入队，`out` 出队。
  - 当 `out` 为空时，将 `in` 栈所有元素倒入 `out`，保证队列顺序。
- **时间复杂度**：均摊 O(1) 入/出队。

```
import java.util.Stack;

class MyQueue {
    private Stack<Integer> in = new Stack<>();
    private Stack<Integer> out = new Stack<>();

    public void push(int x) {
        in.push(x);
    }

    public int pop() {
        if(out.isEmpty()) {
            while(!in.isEmpty()) out.push(in.pop());
        }
        return out.pop();
    }

    public int peek() {
        if(out.isEmpty()) {
            while(!in.isEmpty()) out.push(in.pop());
        }
        return out.peek();
    }

    public boolean empty() {
        return in.isEmpty() && out.isEmpty();
    }
}
```

- **用队列实现栈**的思路类似：使用两个队列维护顺序，每次 `push` 将新元素放到空队列，再把另一队列元素全部转移。

---

### 🔹 3. 滑动窗口最大值 (Sliding Window Maximum)

- **思路**：
  - 使用双端队列 `Deque` 保存索引，保持队列头为当前窗口最大值。
  - 入队时，移除队尾比当前值小的索引。

- 出队时，移除窗口外的索引。
- **时间复杂度**：O(n)，**空间复杂度**：O(k)。

```
import java.util.*;

public class SlidingWindowMax {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if(nums == null || k <= 0) return new int[0];
        int n = nums.length;
        int[] res = new int[n - k + 1];
        Deque<Integer> deque = new LinkedList<>(); // 存索引

        for(int i = 0; i < n; i++) {
            // 移除队尾比当前值小的
            while(!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
                deque.pollLast();
            }
            deque.offerLast(i);

            // 移除队头过期元素
            if(deque.peekFirst() <= i - k) {
                deque.pollFirst();
            }

            // 记录结果
            if(i >= k - 1) {
                res[i - k + 1] = nums[deque.peekFirst()];
            }
        }
        return res;
    }
}
```

## 6.二叉树

### 🔹 1. 二叉树遍历 (Traversal)

- **思路**：
  - 前序：根 → 左 → 右
  - 中序：左 → 根 → 右
  - 后序：左 → 右 → 根
  - 层序：队列 BFS
  - 非递归使用栈模拟递归过程

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int val){ this.val = val; }
}

// 递归遍历
```

```
class BinaryTreeTraversal {
    public void preorder(TreeNode root) {
        if(root == null) return;
        System.out.print(root.val + " ");
        preorder(root.left);
        preorder(root.right);
    }

    public void inorder(TreeNode root) {
        if(root == null) return;
        inorder(root.left);
        System.out.print(root.val + " ");
        inorder(root.right);
    }

    public void postorder(TreeNode root) {
        if(root == null) return;
        postorder(root.left);
        postorder(root.right);
        System.out.print(root.val + " ");
    }

    public void levelOrder(TreeNode root) {
        if(root == null) return;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        while(!q.isEmpty()){
            TreeNode node = q.poll();
            System.out.print(node.val + " ");
            if(node.left != null) q.offer(node.left);
            if(node.right != null) q.offer(node.right);
        }
    }
}

// 非递归前序
class IterativeTraversal {
    public void preorder(TreeNode root) {
        if(root == null) return;
        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);
        while(!stack.isEmpty()){
            TreeNode node = stack.pop();
            System.out.print(node.val + " ");
            if(node.right != null) stack.push(node.right);
            if(node.left != null) stack.push(node.left);
        }
    }
}
```

---

### 🔹 2. 最大深度 / 最小深度 (Max / Min Depth)

- **思路**：
  - 使用递归，深度 = 1 + 左右子树深度的最大/最小值

- 最小深度特殊处理空子树

```
class TreeDepth {
    public int maxDepth(TreeNode root){
        if(root == null) return 0;
        return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
    }

    public int minDepth(TreeNode root){
        if(root == null) return 0;
        if(root.left == null) return 1 + minDepth(root.right);
        if(root.right == null) return 1 + minDepth(root.left);
        return 1 + Math.min(minDepth(root.left), minDepth(root.right));
    }
}
```

---

### 🔹 3. 是否对称 (Symmetric Tree)

- **思路**：
  - 递归：比较左右子树镜像
  - 队列：BFS，每次比较队列两两节点是否相等

```
class SymmetricTree {
    public boolean isSymmetric(TreeNode root){
        if(root == null) return true;
        return dfs(root.left, root.right);
    }

    private boolean dfs(TreeNode t1, TreeNode t2){
        if(t1 == null && t2 == null) return true;
        if(t1 == null || t2 == null) return false;
        return t1.val == t2.val && dfs(t1.left, t2.right) && dfs(t1.right,
t2.left);
    }

    public boolean isSymmetricBFS(TreeNode root){
        if(root == null) return true;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root.left);
        q.offer(root.right);
        while(!q.isEmpty()){
            TreeNode t1 = q.poll();
            TreeNode t2 = q.poll();
            if(t1 == null && t2 == null) continue;
            if(t1 == null || t2 == null || t1.val != t2.val) return false;
            q.offer(t1.left);
            q.offer(t2.right);
            q.offer(t1.right);
            q.offer(t2.left);
        }
        return true;
    }
}
```

### 🔹 4. 二叉搜索树 (BST)

- **思路**：
  - BST 中序遍历有序
  - 判断合法性：递归 + min/max 或中序遍历判断升序
  - 最近公共祖先（LCA）：递归判断两个节点位置

```
class BST {
    private TreeNode prev = null;

    // 判断 BST 合法性
    public boolean isValidBST(TreeNode root){
        if(root == null) return true;
        if(!isValidBST(root.left)) return false;
        if(prev != null && root.val <= prev.val) return false;
        prev = root;
        return isValidBST(root.right);
    }

    // 最近公共祖先
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q){
        if(root == null) return null;
        if(root.val > p.val && root.val > q.val) return
lowestCommonAncestor(root.left, p, q);
        if(root.val < p.val && root.val < q.val) return
lowestCommonAncestor(root.right, p, q);
        return root; // 分岔点即为 LCA
    }
}
```

## 7.图论

### 🔹 1. BFS 最短路径 (Breadth-First Search)

- **思路**：
  - 用队列进行层次遍历
  - 记录从起点到各节点的距离
  - 适合无权图最短路径

```
import java.util.*;

class GraphBFS {
    public int[] bfsShortestPath(int n, List<List<Integer>> adj, int start){
        int[] dist = new int[n];
        Arrays.fill(dist, -1);
        Queue<Integer> q = new LinkedList<>();
        dist[start] = 0;
        q.offer(start);

        while(!q.isEmpty()){
            int u = q.poll();
            for(int v : adj.get(u)){
```

```
                if(dist[v] == -1){
                    dist[v] = dist[u] + 1;
                    q.offer(v);
                }
            }
        }
        return dist;
    }
}
```

---

### 🔹 2. DFS (Depth-First Search)

- **思路**：
  - 拓扑排序：DFS 后序遍历逆序
  - 连通分量：每个未访问节点启动 DFS

```
class GraphDFS {
    public void dfs(int u, boolean[] visited, List<List<Integer>> adj){
        visited[u] = true;
        for(int v : adj.get(u)){
            if(!visited[v]){
                dfs(v, visited, adj);
            }
        }
    }

    public List<Integer> topoSort(int n, List<List<Integer>> adj){
        boolean[] visited = new boolean[n];
        LinkedList<Integer> order = new LinkedList<>();
        for(int i=0;i<n;i++){
            if(!visited[i]){
                dfsTopo(i, visited, adj, order);
            }
        }
        return order;
    }

    private void dfsTopo(int u, boolean[] visited, List<List<Integer>> adj,
LinkedList<Integer> order){
        visited[u] = true;
        for(int v : adj.get(u)){
            if(!visited[v]) dfsTopo(v, visited, adj, order);
        }
        order.addFirst(u);
    }
}
```

---

### 🔹 3. 最短路径算法

- **Dijkstra（单源最短路径，非负权）**
  - 贪心 + 最小堆 (优先队列)
  - 更新起点到各点的最短距离

```
class Dijkstra {
    class Edge { int to, weight; Edge(int t,int w){to=t;weight=w;} }

    public int[] dijkstra(int n, List<List<Edge>> graph, int start){
        int[] dist = new int[n];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[start] = 0;
        PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -
> a[1]));
        pq.offer(new int[]{start,0});

        while(!pq.isEmpty()){
            int[] cur = pq.poll();
            int u = cur[0], d = cur[1];
            if(d > dist[u]) continue;

            for(Edge e : graph.get(u)){
                int v = e.to;
                if(dist[u]+e.weight < dist[v]){
                    dist[v] = dist[u]+e.weight;
                    pq.offer(new int[]{v, dist[v]});
                }
            }
        }
        return dist;
    }
}
```

- **Floyd（任意两点最短路径，O(V³)）**

```
class Floyd {
    public int[][] floyd(int n, int[][] graph){
        int[][] dist = new int[n][n];
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                dist[i][j] = graph[i][j];
            }
        }
        for(int k=0;k<n;k++){
            for(int i=0;i<n;i++){
                for(int j=0;j<n;j++){
                    if(dist[i][k] < Integer.MAX_VALUE && dist[k][j] <
Integer.MAX_VALUE)
                        dist[i][j] = Math.min(dist[i][j], dist[i][k]+dist[k][j]);
                }
            }
        }
        return dist;
    }
}
```

---

### 🔹 4. 最小生成树 (MST)

- **Prim（起点扩展，O(V²) 或 O(E log V) 堆优化）**

```
class PrimMST {
    class Edge { int to, weight; Edge(int t,int w){to=t;weight=w;} }

    public int prim(int n, List<List<Edge>> graph){
        boolean[] visited = new boolean[n];
        PriorityQueue<Edge> pq = new PriorityQueue<>(Comparator.comparingInt(e -> e.weight));
        pq.add(new Edge(0,0));
        int sum = 0;

        while(!pq.isEmpty()){
            Edge e = pq.poll();
            if(visited[e.to]) continue;
            visited[e.to] = true;
            sum += e.weight;

            for(Edge next : graph.get(e.to)){
                if(!visited[next.to]) pq.offer(next);
            }
        }
        return sum;
    }
}
```

- **Kruskal（按边权排序 + 并查集）**

```
class KruskalMST {
    class Edge implements Comparable<Edge>{
        int u,v,weight;
        Edge(int u,int v,int w){this.u=u;this.v=v;this.weight=w;}
        public int compareTo(Edge e){return this.weight - e.weight;}
    }

    class UnionFind {
        int[] parent;
        UnionFind(int n){ parent = new int[n]; for(int i=0;i<n;i++) parent[i]=i; }
        int find(int x){ return parent[x]==x?x:(parent[x]=find(parent[x])); }
        boolean union(int x,int y){
            int px = find(x), py = find(y);
            if(px==py) return false;
            parent[px]=py;
            return true;
        }
    }

    public int kruskal(int n, List<Edge> edges){
        Collections.sort(edges);
        UnionFind uf = new UnionFind(n);
        int sum = 0;
        for(Edge e : edges){
            if(uf.union(e.u, e.v)) sum += e.weight;
        }
        return sum;
    }
```

```
}
```

## 8.动态规划

### 🔹 1. 斐波那契数列 / 青蛙跳台阶

- **思路**：递归或 DP。状态转移：`f(n) = f(n-1) + f(n-2)`

```
// 递归
public int fibRec(int n){
    if(n <= 1) return n;
    return fibRec(n-1) + fibRec(n-2);
}

// 动态规划
public int fibDP(int n){
    if(n <= 1) return n;
    int[] dp = new int[n+1];
    dp[0] = 0; dp[1] = 1;
    for(int i=2;i<=n;i++){
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}

// 滚动数组优化
public int fibOptimized(int n){
    if(n <= 1) return n;
    int a=0, b=1;
    for(int i=2;i<=n;i++){
        int tmp = a + b;
        a = b;
        b = tmp;
    }
    return b;
}
```

---

### 🔹 2. 爬楼梯

- **思路**：每次可走 1 或 2 步，转化为斐波那契数列

```
public int climbStairs(int n){
    if(n <= 2) return n;
    int a=1, b=2;
    for(int i=3;i<=n;i++){
        int tmp = a + b;
        a = b;
        b = tmp;
    }
    return b;
}
```

---

### 🔹 3. 最长公共子序列（LCS）

- **思路**：状态 `dp[i][j]` 表示 `s1[0..i]` 与 `s2[0..j]` 的 LCS 长度
- 转移方程：
  - `s1[i] == s2[j]` → `dp[i][j] = dp[i-1][j-1]+1`
  - `s1[i] != s2[j]` → `dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1])`

```
public int lcs(String s1, String s2){
    int m = s1.length(), n = s2.length();
    int[][] dp = new int[m+1][n+1];
    for(int i=1;i<=m;i++){
        for(int j=1;j<=n;j++){
            if(s1.charAt(i-1) == s2.charAt(j-1))
                dp[i][j] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[m][n];
}
```

---

### 🔹 4. 最长上升子序列（LIS）

- **思路**：状态 `dp[i]` 表示以 `nums[i]` 结尾的 LIS 长度
- 转移方程：`dp[i] = max(dp[j]+1) for j<i && nums[j]<nums[i]`

```
public int lis(int[] nums){
    int n = nums.length;
    int[] dp = new int[n];
    Arrays.fill(dp, 1);
    int res = 1;
    for(int i=1;i<n;i++){
        for(int j=0;j<i;j++){
            if(nums[j]<nums[i]) dp[i] = Math.max(dp[i], dp[j]+1);
        }
        res = Math.max(res, dp[i]);
    }
    return res;
}
```

---

### 🔹 5. 背包问题

- **01 背包**
  - `dp[i][w]` 表示前 i 件物品，容量为 w 时的最大价值
  - 转移方程：`dp[i][w] = max(dp[i-1][w], dp[i-1][w-weight[i]]+value[i])`

```
public int knapsack01(int[] weight, int[] value, int W){
    int n = weight.length;
    int[] dp = new int[W+1];
    for(int i=0;i<n;i++){
        for(int w=W; w>=weight[i]; w--){
            dp[w] = Math.max(dp[w], dp[w-weight[i]] + value[i]);
        }
    }
    return dp[W];
}
```

- **完全背包**
  - 可以多次选择，每个物品无限
  - 遍历方向改为从小到大

```
public int knapsackComplete(int[] weight, int[] value, int W){
    int n = weight.length;
    int[] dp = new int[W+1];
    for(int i=0;i<n;i++){
        for(int w=weight[i]; w<=W; w++){
            dp[w] = Math.max(dp[w], dp[w-weight[i]] + value[i]);
        }
    }
    return dp[W];
}
```

---

### 🔹 6. 编辑距离

- **思路**：状态 `dp[i][j]` 表示 `word1[0..i-1]` 变成 `word2[0..j-1]` 的最少操作次数
- 操作：插入、删除、替换

```
public int minDistance(String word1, String word2){
    int m = word1.length(), n = word2.length();
    int[][] dp = new int[m+1][n+1];
    for(int i=0;i<=m;i++) dp[i][0] = i;
    for(int j=0;j<=n;j++) dp[0][j] = j;

    for(int i=1;i<=m;i++){
        for(int j=1;j<=n;j++){
            if(word1.charAt(i-1) == word2.charAt(j-1))
                dp[i][j] = dp[i-1][j-1];
            else
                dp[i][j] = 1 + Math.min(dp[i-1][j-1], Math.min(dp[i-1][j], dp[i]
[j-1]));
        }
    }
    return dp[m][n];
}
```

---

### 🔹 7. 最大子数组和（Kadane 算法）

- **思路**：维护 `currentSum`，若为负则从下一个元素重新开始
- 时间复杂度 O(n)

```
public int maxSubArray(int[] nums){
    int maxSum = nums[0], currentSum = nums[0];
    for(int i=1;i<nums.length;i++){
        currentSum = Math.max(nums[i], currentSum + nums[i]);
        maxSum = Math.max(maxSum, currentSum);
    }
    return maxSum;
}
```

## 9.回溯 & 搜索

### 🔹 1. 全排列（Permutations）

- **思路**：DFS + 回溯
- 每次选择一个元素，递归生成剩余元素的排列，回溯时撤销选择

```
public List<List<Integer>> permute(int[] nums){
    List<List<Integer>> res = new ArrayList<>();
    boolean[] used = new boolean[nums.length];
    backtrack(res, new ArrayList<>(), nums, used);
    return res;
}

private void backtrack(List<List<Integer>> res, List<Integer> temp, int[] nums,
boolean[] used){
    if(temp.size() == nums.length){
        res.add(new ArrayList<>(temp));
        return;
    }
    for(int i=0;i<nums.length;i++){
        if(used[i]) continue;
        temp.add(nums[i]);
        used[i] = true;
        backtrack(res, temp, nums, used);
        temp.remove(temp.size()-1);
        used[i] = false;
    }
}
```

---

### 🔹 2. 子集（Subsets）

- **思路**：DFS 回溯或迭代
- 每个元素有“选/不选”两种选择

```
public List<List<Integer>> subsets(int[] nums){
    List<List<Integer>> res = new ArrayList<>();
    backtrackSubsets(res, new ArrayList<>(), nums, 0);
    return res;
}
```

```
private void backtrackSubsets(List<List<Integer>> res, List<Integer> temp, int[] nums, int start){
    res.add(new ArrayList<>(temp));
    for(int i=start;i<nums.length;i++){
        temp.add(nums[i]);
        backtrackSubsets(res, temp, nums, i+1);
        temp.remove(temp.size()-1);
    }
}
```

---

### 🔹 3. 组合（Combination）

- **思路**：长度固定的子集（例如从 n 个数中选 k 个）
- 回溯 + 起点索引控制顺序

```
public List<List<Integer>> combine(int n, int k){
    List<List<Integer>> res = new ArrayList<>();
    backtrackCombine(res, new ArrayList<>(), 1, n, k);
    return res;
}

private void backtrackCombine(List<List<Integer>> res, List<Integer> temp, int start, int n, int k){
    if(temp.size() == k){
        res.add(new ArrayList<>(temp));
        return;
    }
    for(int i=start;i<=n;i++){
        temp.add(i);
        backtrackCombine(res, temp, i+1, n, k);
        temp.remove(temp.size()-1);
    }
}
```

---

### 🔹 4. N 皇后

- **思路**：DFS + 回溯 + 检查皇后放置合法性
- 用数组 `cols[i]` 表示列占用情况，`diag1[i+j]` 表示主对角线占用，`diag2[i-j+n-1]` 表示副对角线占用

```
public List<List<String>> solveNQueens(int n){
    List<List<String>> res = new ArrayList<>();
    int[] cols = new int[n];
    Arrays.fill(cols, -1);
    backtrackNQueens(res, cols, 0, n);
    return res;
}

private void backtrackNQueens(List<List<String>> res, int[] cols, int row, int n){
    if(row == n){
```

```
        List<String> board = new ArrayList<>();
        for(int i=0;i<n;i++){
            char[] line = new char[n];
            Arrays.fill(line, '.');
            line[cols[i]] = 'Q';
            board.add(new String(line));
        }
        res.add(board);
        return;
    }
    for(int c=0;c<n;c++){
        boolean valid = true;
        for(int r=0;r<row;r++){
            if(cols[r]==c || Math.abs(row-r)==Math.abs(c-cols[r])){
                valid = false;
                break;
            }
        }
        if(!valid) continue;
        cols[row] = c;
        backtrackNQueens(res, cols, row+1, n);
        cols[row] = -1;
    }
}
```

---

### 🔹 5. 数独求解（Sudoku Solver）

- **思路**：DFS + 回溯
- 每空格尝试 1-9，递归解决剩余空格，回溯撤销

```
public void solveSudoku(char[][] board){
    backtrackSudoku(board, 0, 0);
}

private boolean backtrackSudoku(char[][] board, int row, int col){
    if(col == 9) { row++; col=0; }
    if(row == 9) return true;
    if(board[row][col] != '.') return backtrackSudoku(board,row,col+1);

    for(char c='1';c<='9';c++){
        if(isValid(board,row,col,c)){
            board[row][col]=c;
            if(backtrackSudoku(board,row,col+1)) return true;
            board[row][col]='.';
        }
    }
    return false;
}

private boolean isValid(char[][] board, int row, int col, char c){
    for(int i=0;i<9;i++){
        if(board[row][i]==c || board[i][col]==c) return false;
        if(board[3*(row/3)+i/3][3*(col/3)+i%3]==c) return false;
    }
```

```
    return true;
}
```

---

### 🔹 6. 电话号码字母组合（Letter Combinations of a Phone Number）

- **思路**：回溯 + 映射数字到字母

```
String[] map = {"","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};
public List<String> letterCombinations(String digits){
    List<String> res = new ArrayList<>();
    if(digits.length()==0) return res;
    backtrackPhone(res, new StringBuilder(), digits, 0);
    return res;
}

private void backtrackPhone(List<String> res, StringBuilder sb, String digits,
int index){
    if(index == digits.length()){
        res.add(sb.toString());
        return;
    }
    String letters = map[digits.charAt(index)-'0'];
    for(char c : letters.toCharArray()){
        sb.append(c);
        backtrackPhone(res, sb, digits, index+1);
        sb.deleteCharAt(sb.length()-1);
    }
}
```

---

### 🔹 7. IP 地址划分

- **思路**：回溯 + 剪枝
- 每次尝试 1~3 位，值 ≤255，且不能有前导零

```
public List<String> restoreIpAddresses(String s){
    List<String> res = new ArrayList<>();
    backtrackIP(res, new ArrayList<>(), s, 0);
    return res;
}

private void backtrackIP(List<String> res, List<String> temp, String s, int
index){
    if(temp.size()==4){
        if(index==s.length()) res.add(String.join(".", temp));
        return;
    }
    for(int len=1;len<=3;len++){
        if(index+len>s.length()) break;
        String seg = s.substring(index,index+len);
        if(seg.startsWith("0") && seg.length()>1) continue;
        int val = Integer.parseInt(seg);
        if(val>255) continue;
        temp.add(seg);
```

```
            backtrackIP(res,temp,s,index+len);
            temp.remove(temp.size()-1);
        }
    }
```

## 10.高频面试题

### 🔹 10.1 两数之和 / 三数之和

**问题描述**

- 两数之和：给定数组 `nums` 和目标值 `target`，找出两个数之和等于 `target` 的索引。
- 三数之和：给定数组 `nums`，找出所有和为 0 的三元组，且不重复。

**解题思路**

- 两数之和 → 哈希表存已经遍历过的数，O(n) 时间复杂度。
- 三数之和 → 排序 + 双指针避免重复，O($n^2$) 时间复杂度。

**Java 核心代码**

```
// 两数之和
public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement)) {
            return new int[]{map.get(complement), i};
        }
        map.put(nums[i], i);
    }
    return new int[0];
}

// 三数之和
public List<List<Integer>> threeSum(int[] nums) {
    List<List<Integer>> res = new ArrayList<>();
    Arrays.sort(nums);
    for (int i = 0; i < nums.length - 2; i++) {
        if (i > 0 && nums[i] == nums[i - 1]) continue; // 去重
        int left = i + 1, right = nums.length - 1;
        while (left < right) {
            int sum = nums[i] + nums[left] + nums[right];
            if (sum == 0) {
                res.add(Arrays.asList(nums[i], nums[left], nums[right]));
                while (left < right && nums[left] == nums[left + 1]) left++; //
去重
                while (left < right && nums[right] == nums[right - 1]) right--;
// 去重
                left++; right--;
            } else if (sum < 0) left++;
            else right--;
        }
    }
    return res;
```

```
}
```

**复杂度分析**

- 两数之和：O(n) 时间 + O(n) 空间
- 三数之和：O(n²) 时间 + O(1) 或 O(n) 输出空间

---

### 🔹 10.2 接雨水问题（Trapping Rain Water）

**问题描述**
给定数组 `height`，表示每个柱子的高度，求能够接多少雨水。

**解题思路**

1. 双指针 → 从两边向中间遍历，维护左右最大值。
2. 单调栈 → 用栈保存可能的边界，遇到比栈顶高的柱子就计算雨水。

**Java 核心代码**

```
// 双指针
public int trap(int[] height) {
    int left = 0, right = height.length - 1;
    int leftMax = 0, rightMax = 0, res = 0;
    while (left < right) {
        if (height[left] < height[right]) {
            leftMax = Math.max(leftMax, height[left]);
            res += leftMax - height[left];
            left++;
        } else {
            rightMax = Math.max(rightMax, height[right]);
            res += rightMax - height[right];
            right--;
        }
    }
    return res;
}
```

**复杂度分析**

- 时间：O(n)，空间：O(1)

---

### 🔹 10.3 区间合并

**问题描述**
给定一组区间 `[start, end]`，合并所有重叠区间。

**解题思路**

- 先按左端点排序，再遍历合并重叠区间。

**Java 核心代码**

```
public int[][] merge(int[][] intervals) {
    if (intervals.length <= 1) return intervals;
    Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
```

```
    List<int[]> res = new ArrayList<>();
    int[] current = intervals[0];
    for (int i = 1; i < intervals.length; i++) {
        if (intervals[i][0] <= current[1]) {
            current[1] = Math.max(current[1], intervals[i][1]);
        } else {
            res.add(current);
            current = intervals[i];
        }
    }
    res.add(current);
    return res.toArray(new int[res.size()][]);
}
```

**复杂度分析**

- 时间：O(n log n)（排序），空间：O(n)

---

### 🔹 10.4 旋转图像（矩阵顺时针旋转 90°）

**问题描述**
给定 n×n 矩阵，将其旋转 90°。

**解题思路**

- 先转置矩阵，再翻转每一行。

**Java 核心代码**

```
public void rotate(int[][] matrix) {
    int n = matrix.length;
    // 转置
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = temp;
        }
    }
    // 翻转每一行
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n / 2; j++) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[i][n - 1 - j];
            matrix[i][n - 1 - j] = temp;
        }
    }
}
```

**复杂度分析**

- 时间：$O(n^2)$，空间：O(1)

---

### 🔹 10.5 岛屿数量（Number of Islands）

**问题描述**

二维网格 `'1'` 表示陆地，`'0'` 表示水，求岛屿数量。

**解题思路**

- DFS 或 BFS 遍历，每遇到 `'1'` 递归/队列将整个岛标记为已访问。

**Java 核心代码**

```
// DFS
public int numIslands(char[][] grid) {
    int m = grid.length, n = grid[0].length;
    int count = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == '1') {
                dfs(grid, i, j);
                count++;
            }
        }
    }
    return count;
}

private void dfs(char[][] grid, int i, int j) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] != '1') return;
    grid[i][j] = '0';
    dfs(grid, i + 1, j);
    dfs(grid, i - 1, j);
    dfs(grid, i, j + 1);
    dfs(grid, i, j - 1);
}
```

**复杂度分析**

- 时间：O(m × n)，空间：O(m × n) 递归栈

---

🔹 **10.6 单词接龙**（Word Ladder）

**问题描述**

给定 `beginWord`、`endWord` 和字典 `wordList`，每次只能改变一个字母，求最短转换序列长度。

**解题思路**

- BFS 寻找最短路径，使用字典进行快速匹配。

**Java 核心代码**

```
public int ladderLength(String beginWord, String endWord, List<String> wordList)
{
    Set<String> wordSet = new HashSet<>(wordList);
    if (!wordSet.contains(endWord)) return 0;
    Queue<String> queue = new LinkedList<>();
    queue.offer(beginWord);
    int level = 1;
```

```
    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            String word = queue.poll();
            char[] chs = word.toCharArray();
            for (int j = 0; j < chs.length; j++) {
                char old = chs[j];
                for (char c = 'a'; c <= 'z'; c++) {
                    chs[j] = c;
                    String next = new String(chs);
                    if (next.equals(endWord)) return level + 1;
                    if (wordSet.contains(next)) {
                        queue.offer(next);
                        wordSet.remove(next);
                    }
                }
                chs[j] = old;
            }
        }
        level++;
    }
    return 0;
}
```

**复杂度分析**

- 时间：O(N × L × 26)，N 为字典大小，L 为单词长度
- 空间：O(N)

# 十三、场景题

## 🔹 1. 手写 Flink 的 UV（独立访客统计）

**问题描述**
统计某段时间内访问某网站或应用的独立用户数（UV）。

**解题思路**

1. 数据流：用户访问事件 → 提取 userId → 去重 → 统计数量。
2. Flink 常用方法：
    - 使用 KeyedStream 按照用户 ID 去重
    - 可以用 **窗口**统计 UV（如滚动窗口、滑动窗口）
    - 可以用 **HyperLogLog** 进行近似去重，降低内存占用

**核心示例（Java/Flink）**

```
DataStream<Event> events = env.addSource(...);

events
    .keyBy(Event::getUserId)
    .window(TumblingEventTimeWindows.of(Time.hours(1)))
    .process(new ProcessWindowFunction<Event, Long, String, TimeWindow>() {
        @Override
```

```
        public void process(String key, Context context, Iterable<Event>
elements, Collector<Long> out) {
            out.collect(1L); // 每个 key 只计一次
        }
    })
    .windowAll(TumblingEventTimeWindows.of(Time.hours(1)))
    .reduce(Long::sum)
    .print();
```

- **时间复杂度**：O(n)，每条事件只处理一次
- **空间复杂度**：O(unique userId)

---

## 🔹 2. Flink 的分组 TopN

**问题描述**
统计每个分组（如商品分类）的 TopN 数据（如销量前 3 的商品）。

**解题思路**

1. keyBy 分组
2. 窗口或状态存储每个 key 的 TopN 数据
3. 使用 PriorityQueue 维护当前 TopN

**核心示例**

```
KeyedStream<Sale, String> keyed = sales.keyBy(Sale::getCategory);
keyed.process(new KeyedProcessFunction<String, Sale, List<Sale>>() {
    private transient PriorityQueue<Sale> topN;
    @Override
    public void open(Configuration parameters) {
        topN = new PriorityQueue<>(Comparator.comparingInt(Sale::getCount));
    }
    @Override
    public void processElement(Sale sale, Context ctx, Collector<List<Sale>> out)
{
        topN.offer(sale);
        if (topN.size() > 3) topN.poll(); // 保留前三
        out.collect(new ArrayList<>(topN));
    }
});
```

---

## 🔹 3. Spark 的分组 TopN

**问题描述**
统计每个 key 的 TopN 数据，数据量大时如何处理。

**解题思路**

- 方法 1：groupByKey + mapValues 排序 → 数据量大容易 OOM
- 方法 2：对 key 遍历排序 → 时间复杂度高
- 方法 3：自定义分区器 + 分区内排序 → 高效，避免 OOM

**关键点**

- 尽量使用 `reduceByKey` 或 `aggregateByKey` 避免 groupByKey 大量数据落地
- 使用 `mapPartitions` 在分区内完成排序

---

## 🔹 4. 如何快速从 40 亿条数据中判断数据 123 是否存在

**解题思路**

- 使用 **Bloom Filter**：空间效率高，支持快速判断是否存在（可能有误判，但不存在一定准确）
- 或使用数据库索引/分布式 KV 存储（如 HBase、Redis）

---

## 🔹 5. 给你 100G 数据，1G 内存，如何排序

**解题思路**

- 外部排序（External Sort）

1. 将大数据切分成若干块，每块可以在内存中排序（如 1G 一块）
2. 将每块排序结果写入磁盘
3. 使用归并算法（取最小）将所有块归并成有序文件

---

## 🔹 6. 公平调度器容器集中在同一服务器上?

**问题描述**
使用公平调度器时，容器可能分配不均导致集中在同一服务器。

**解题思路**

- 可能原因：调度器只按队列公平，不考虑节点负载
- 解决方案：开启 **资源感知调度（Resource-aware scheduling）** 或设置 **节点亲和性**

---

## 🔹 7. 匹马赛跑，1 个赛道，每次 5 匹进行比赛，最少赛几次找出前三名?

**问题描述**

- 25 匹马，1 条赛道，每次 5 匹，无法计时，只能知道相对名次。

**解题思路**

1. 第 1 轮 → 5 组各赛一次 → 5 次
2. 第二轮 → 每组冠军赛一次 → 1 次
3. 第三轮 → 根据前两轮结果，最多 5 匹马参与，进行第 3 次比赛

- **最少 7 次**比赛可以确定前三名

---

## 🔹 8. 给定一个点、一条线、一个三角形、有向无环图，用 Java 面向对象建模

```
// 点
class Point {
    double x, y;
    public Point(double x, double y){ this.x = x; this.y = y; }
}

// 线段
class Line {
    Point start, end;
    public Line(Point start, Point end){ this.start = start; this.end = end; }
}

// 三角形
class Triangle {
    Point a, b, c;
    public Triangle(Point a, Point b, Point c){ this.a = a; this.b = b; this.c = c; }
}

// 有向无环图
class DAG {
    Map<Node, List<Node>> edges = new HashMap<>();
    void addEdge(Node from, Node to) { edges.computeIfAbsent(from, k -> new ArrayList<>()).add(to); }
}
class Node { String name; Node(String name){ this.name=name; } }
```

---

## 🔹 9. SQL 优化

**原 SQL**

```
SELECT 2d FROM t_order WHERE 2d IN (SELECT 2d FROM t_order_f);
```

**优化方案**

- 改用 `INNER JOIN`，避免子查询性能问题

```
SELECT t1.2d
FROM t_order t1
INNER JOIN t_order_f t2
ON t1.2d = t2.2d;
```

**验证优化效果**

- 查看执行时间差异
- 可使用 `EXPLAIN` 查看执行计划
- 性能提升比例 = `(原时间 - 优化后时间) / 原时间 * 100%`

## 🔹 10. TOP K 算法（10 个 1G 文件的 query 排序）

**问题描述**

- 有 10 个文件，每个 1G，每行存放用户 query，可能重复。
- 目标：按 query 出现频度排序，找出最热门的 query。

**解题思路 & 方案**

**方案 1：分文件 + 内存统计 + 排序**

1. 顺序读取 10 个文件，对每个 query 做 `hash(query) % 10`，写入对应的 10 个新文件。
   - 保证相同 query 在同一个文件中
2. 每个新文件约 1G，可以在一台 2G 内存机器上统计每个 query 出现次数（`HashMap<String, Integer>`）
3. 对统计结果用快速排序 / 堆排序 / 归并排序得到每个文件内的局部 Top K
4. 对 10 个文件的局部 Top K 进行归并排序，得到全局排序

**方案 2：内存统计法**

- 如果 query 总量有限，可以直接使用 `Trie` 或 `HashMap` 将所有 query 加入内存
- 对出现次数排序即可

**方案 3：分布式处理（MapReduce / Flink）**

- Map 阶段：按 key 统计次数
- Reduce 阶段：每个分区计算局部 Top K
- 最终合并得到全局 Top K

---

## 🔹 11. 不重复的数据（2.5 亿整数，内存不足）

**问题描述**

- 在 2.5 亿个整数中找出不重复的整数，内存不足以存全部整数。

**解题思路 & 方案**

**方案 1：2-bit Bitmap**

- 每个数分配 2 bit：
  - 00：不存在
  - 01：出现一次
  - 10：出现多次
- 扫描 2.5 亿整数，更新 Bitmap
- 最后扫描 Bitmap，把值为 01 的整数输出

**方案 2：分文件 + 去重**

- 将大文件分割成若干小文件（hash 分桶）
- 每个小文件内找不重复的整数，排序
- 最后归并排序输出，注意去掉重复

---

## 🔹 12. 判断数据是否存在（40 亿 unsigned int）

**问题描述**

- 有 40 亿未排序的整数，给定一个数，快速判断其是否存在。

**解题思路 & 方案**

**方案 1：位图法**

- 申请 512M 内存，用 1 bit 表示一个整数
- 扫描 40 亿整数，设置相应 bit
- 查询某数是否存在：查看对应 bit

**方案 2：递归折半法（文件分块）**

- 40 亿整数分为两类（最高位 0 或 1），写入两个文件
- 对目标数，选择相应文件继续折半分割（次高位 0 或 1），逐步缩小范围
- 时间复杂度 O(log n)，无需全部加载内存

---

## 🔹 13. 重复最多的数据（千万条短信，找前 10 条）

**问题描述**

- 1 千万条短信，每行一条，可能重复
- 目标：找重复出现最多的前 10 条

**解题思路 & 方案**

**方案 1：排序 + 遍历**

- 排序短信，遍历一次统计重复次数
- 复杂度至少 O(n log n)

**方案 2：哈希表统计 + 小顶堆**

- 创建 `HashMap<String, Integer>`，统计每条短信出现次数
- 同时维护一个大小为 10 的最小堆（top 10）
- 扫描所有短信一次即可找到前 10
- 时间复杂度 O(n)

**Java 核心逻辑示例**

```
Map<String, Integer> freq = new HashMap<>();
PriorityQueue<String> top10 = new PriorityQueue<>(
    (a,b) -> freq.get(a) - freq.get(b)
);

for(String msg : messages){
    freq.put(msg, freq.getOrDefault(msg,0)+1);
    if(!top10.contains(msg)){
        top10.offer(msg);
        if(top10.size() > 10) top10.poll();
    }
}
```

-

-