信息流数据仓库

(作者：罗圣杰)

# 第1章 项目框架

## 1.1 概述

随着人们对手机依赖程度的增加，手机已经成为了绝大多数人社交、购物、休闲娱乐、学习、发表见解、获取时事新闻等需求的主要实现渠道。移动端已经是绝大数用户流量的入口

* 当一款APP上线后，为了更好地改进这款APP，公司需要了解到这款APP的详细使用情况，例如新增用户数、活跃用户数等；为了了解用户对于这款APP不同版本的接受程度，需要了解这款APP在用户群体中的版本分布情况。
* 目前移动端作为手机的巨大入口，一些工具类的app用户停留时长不高。为了延长用户在手机上的停留时长，方法之一是在app中嵌入内容。信息流包括音频，视频，文字，图片等。丰富的内容，能极大限度的留住用户。这需要对用户进行个性化分析，所以信息流推荐系统就起到非常大的作用。

以上这些需求的实现，都要建立在一个稳定的分布式日志采集和统计分析系统之上。

在大数据技术趋于成熟的今天，大数据技术的发展使采集海量用户信息并分析用户行为进而有目的的改进APP这一需求的实现成为可能，本项目就致力于打造一套成熟的日志数据统计分析系统，通过离线数据分析系统和实时数据分析系统两个模块的协同作用，完成APP各项指标的分析，同时为信息流提供相关的数据支持与实现。

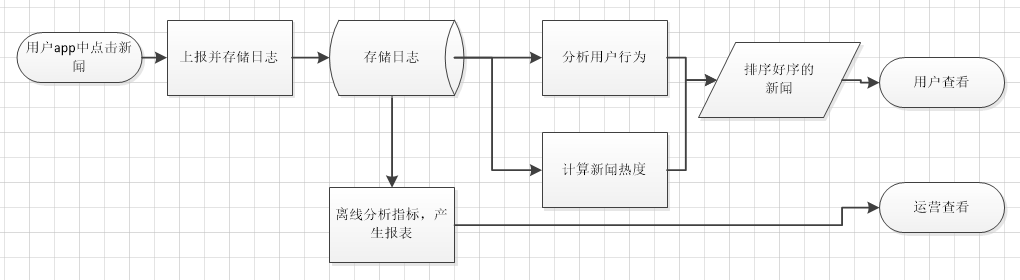
## 1.2 项目需求

### 1.2.1 日志业务流程

现在我们需要这样一个系统，来承载信息流，实现下面的功能：

分析用户行为功能：

1. 收集用户观看新闻的日志
2. 每日分析用户的行为
3. 实时对用户的行为作出反馈和推荐



图为业务流程

**对象说明：**

* 用户:用户由设备上传的唯一ID辨别。Id由前端产生，用来标注每一个用户。
* 新闻:包括图片，文字组合成的新闻，带有唯一id，计算过程以id处理
* 新闻质量:与点击率相关联
* 用户中心：数据仓库中维护所有点击过新闻的用户。
* 新闻中心：数据仓库中维护所有新闻的集合



图为数据流程

**报表说明：**

* 新增用户：首次联网使用应用的用户。如果一个用户首次打开某app，那这个用户定义为新增用户；卸载再安装的设备，不会被算作一次新增。新增用户包括日新增用户、周新增用户、月新增用户。
* 活跃用户：打开应用的用户即为活跃用户，不考虑用户的使用情况。每天一台设备打开多次会被计为一个活跃用户。
* 月活跃用户：某个自然周（月）内启动过应用的用户，该周（月）内的多次启动只记一个活跃用户。

### 1.2.2 离线报表需求

* 月活跃用户，日活跃用户: 用户
* 新增用户
* 新闻每日点击率，展示量，点击量
* 活跃用户信息（用户最近3天，7天，30天对新闻产生点击）
* 历史用户信息
* 历史新闻信息（最前和最后展示时间）
* 版本分布（各个版本用户的分布情况）

### 1.2.3 业务术语介绍

1. 用户

用户以设备为判断标准，在移动统计中，每个独立设备认为是一个独立用户。Android系统根据 IMEI 号，IOS 系统根据 OpenUDID 来标识一个独立用户，每部手机一个用户。对应的是user\_id。我们采取前端生成的方式来上报。

1. 新增用户

首次联网使用应用的用户。如果一个用户首次打开某 app，那这个用户定义为新增用户；卸载再安装的设备，不会被算作一次新增。新增用户包括日新增用户、周新增用户、月新增用户。

1. 活跃用户

**前台活跃用户**：打开应用的用户即为活跃用户，不考虑用户的使用情况。每天一台设备打开多次会被计为一个活跃用户。

**后台活跃用户**：app在后台运行的用户。

1. 周（月）活跃用户

某个自然周（月）内启动过应用的用户，该周（月）内的多次启动只记一个活跃用户。

1. 月活跃率

月活跃用户与截止到该月累计的用户总和之间的比例。

1. 沉默用户

用户仅在安装当天（次日）启动一次，后续时间无再启动行为。该指标可以反映新增用户质量和用户与 APP 的匹配程度。

1. 版本分布

不同版本的周内各天新增用户数，活跃用户数和启动次数。利于判断 App 各个版本之间的优劣和用户行为习惯。

1. 留存用户

某段时间内的新增用户，经过一段时间后，仍然使用应用的被认作是留存用户；这部分

用户占当时新增用户的比例即是留存率。例如，5 月份新增用户 200，这 200 人在 6 月份启动过应用的有 100 人，7 月份启动过应用的有 80 人，8 月份启动过应用的有 50 人；则 5 月份新增用户一个月后的留存率是 50%，二个月后的留存率是 40%，三个月后的留存率是 25%。

1. 用户新鲜度

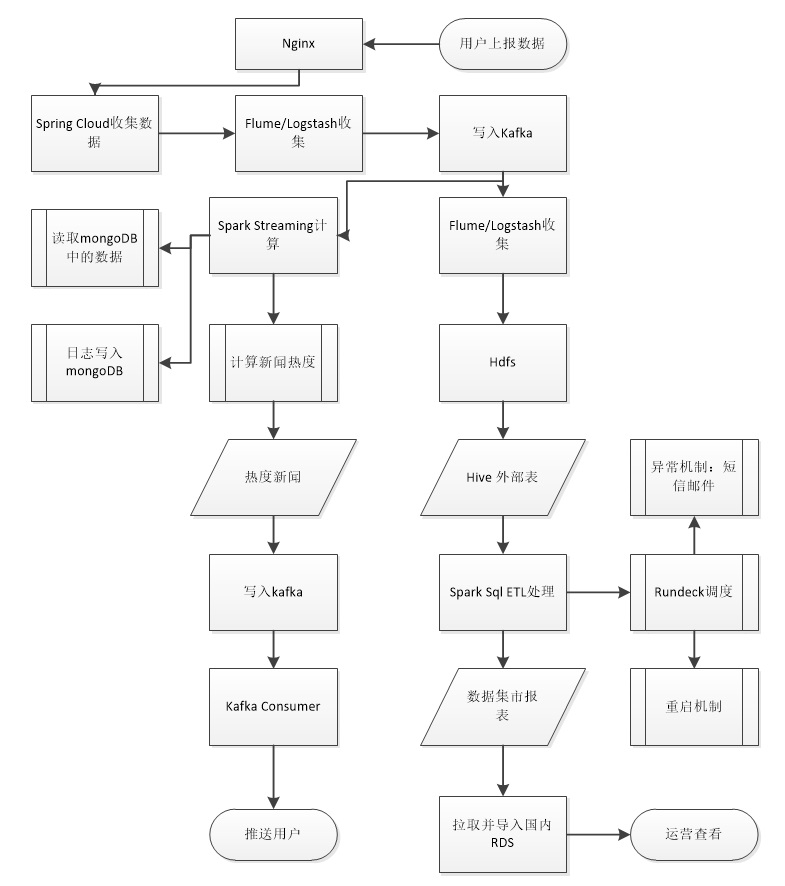
每天启动应用的新老用户比例，即新增用户数占活跃用户数的比例。

### 1.2.4 实时报表需求

* 最近24小时新闻点击率
* 根据点击计算热度，根据个性化排序，推送给后台。

## 1.3 项目框架

### 1.3.1 数据仓库流程



图：整个数据仓库的数据流程

**注意：**

1. **其中RDS可以使用Elasticsearch**
2. **Logstash与Flume可以互换**
3. **Spring Boot是Tomcat的替代方案，也可以用tomcat+Javaweb**
4. **Flume，kafka，spark可以使用老版本**

根据图中的项目框架可知，本项目由离线数据处理系统和实时数据处理系统两个部分组成。

离线数据处理系统与实时数据处理系统共用一套日志采集系统，日志采集系统采用了双层Flume拓扑结构，第一层实现数据的采集，第二层实现数据的集中聚合处理；手机APP客户端的日志数据被日志采集系统采集完成后，被分别输送到离线和实时数据处理系统中进行处理。

首先，APP客户端的日志数据定时（如0.5小时/次）服务器发送日志，服务器端通过Nginx实现负载均衡，Nginx将日志数据负载均衡到多给web应用上，然后通过日志写在磁盘。

随后，双层Flume架构中的第一层数据采集Flume将对应的Tomcat生成的日志文件采集到其拓扑结构中，随后多个第一层数据采集Flume的数据汇总到第二层的数据聚合Flume上，完成数据的聚合和集中处理。

然后，第二层的数据聚合Flume根据日志数据的类型，将日志数据发送到不同的Kafka主题中，在Kafka中完成数据的分布式存储。

最后，离线数据处理系统和实时数据处理系统分别从Kafka中消费消息，各自完成数据的离线分析处理和实时分析处理。

### 1.3.2 技术组件



数据收集层组件

用户在使用APP的过程中，会不断产生日志数据。当用户后台运行的APP的时候，会产生后台日志。打开APP的时候，会产生前台APP启动日志，启动日志中记录了APP启动时间，运行时长等信息；

APP中运行的服务会定时将手机系统中产生的日志提取出来，然后经过一定的处理，例如合并等，目的是减少对服务端的压力，数据合并完成后，**在APP中有一个数据库，合并的数据会放入这一数据库中**，因为用户有可能断网，导致数据不能及时的发送到服务器，因此先存储在APP的本地数据库中，等到用户联网的时候再一并发送出来。

当手机客户端需要发送数据时，会对数据进行对称加密和压缩，压缩可以减少服务器的带宽，然后调用服务器的接口上传数据，服务端对数据进行解压和解密，然后用SLF4J写入本地磁盘中。（**写入本地磁盘的目的是让业务系统与采集系统完全解耦，同时缓冲峰值数据**）

**所以有几类产品：**



* 数据集成:Flume，DataX，Kafka
* 数据计算:Spark，Hive
* 数据存储:Hive，S3，Mysql,HDFS
* 数据管理:Rundeck，元数据管理
* 数据查询:Presto

**数仓设计的几个考虑点：**

* 容灾性：数据备份
* 高可用: 分布式应用。一些应用宕机之后能很快恢复
* 即时性：流式计算的处理时间需要在业务要求的时间以内。
* 预警：有相关的检查，短信，电话预警
* 文档：包括元数据，注释或电子文档，对数据有相关说明

### 1.3.3 数据集市

数据集市就是数据仓库的产品。运营有时候需要进行一些自定义的查询，或报表查看这个时候需要看一些可视化的报表。这个时候就要获取数据集市的数据产品。可以分为:

* 可视化与非可视化数据报表
* 即时查询与周期性查询数据报表

## 1.4 原始日志概述

本项目中的日志数据由三种类型的日志组成，分别为

* 新闻点击/展示日志。
* 用户后台启动日志
* 用户前台启动日志

三种日志中共有的数据内容是：userId、area、appVersion,appKey。

### 1.4.1 新闻点击/展示日志

格式如下：

1537619705000| **{**

**"appkey"**: "browser"**,**

**"he"**: **{**

**"userId"**: "a12234sd"**,**

**"area"**: "BR"**,**

**"appVersion"**: "V1.0.2",

**"appTime"**: "1537533305000"

**},**

**"et"**: [**{**

**"eventName "**: "display"**,**

**"kv "**: **{**

**"action"**: "2"，**"newsId"**: "2343234"

**}**

**}]**

**}**

|  |  |
| --- | --- |
| 标签 | 含义 |
| 最前面的时间 | 服务端加上的时间 |
| appkey | APP的ID |
| he | 公共字段集合 |
| userId | 用户唯一Id，根据imei生成 |
| area | 国家 |
| appVerson | app版本号 |
| appTime | 客户端日志产生时间 |
| et | 业务字段集合，是一个json数组，多个事件共用 |
| eventName | 事件名称 |
| kv | 事件的键值对 |
| action | 1，展示 2，点击 |
| newsId | 新闻Id |

### 1.4.2 用户后台上报日志

1537619705000| **{**

**"appkey"**: "browser"**,**

**"he"**: **{**

**"userId"**: "a12234sd"**,**

**"area"**: "BR"**,**

**"appVersion"**: "V1.0.2",

**"appTime"**: "1537533305000"

**},**

**"et"**: [**{**

**"eventName "**: "background"，

**"kv "**: **{**

**"createTime"**: "1537533305000"

**}**

**}]**

**}**

|  |  |
| --- | --- |
| 标签 | 含义 |
| 最前面的时间 | 服务端加上的时间 |
| eventName | background后台上报日志 |
| createTime | 后台活跃产生时间，即该后台启动时间 |

### 1.4.3 用户前台上报日志

1537619705000| **{**

**"appkey"**: "browser"**,**

**"he"**: **{**

**"userId"**: "a12234sd"**,**

**"area"**: "BR"**,**

**"appVersion"**: "V1.0.2",

**"appTime"**: "1537533305000"

**},**

**"et"**: [**{**

**"eventName "**: "foreground"，

**"kv "**: **{**

**"createTime"**: "1537533305000"

**}**

**}]**

**}**

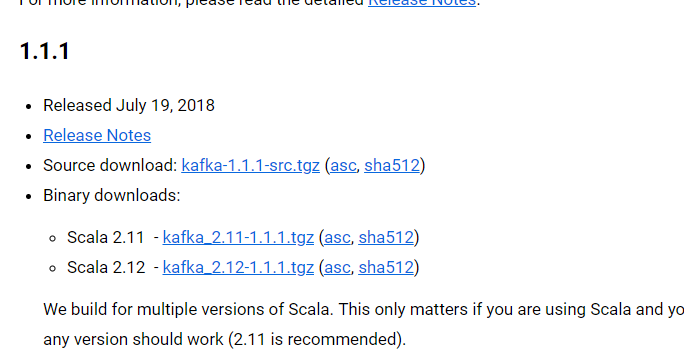
|  |  |
| --- | --- |
| 标签 | 含义 |
| 最前面的时间 | 服务端加上的时间 |
| eventName | foreground后台上报日志 |
| createTime | 前台活跃产生时间，即用户打开该应用的时候 |

## 1.5 产品选型

### 1.5.1 Kafka版本

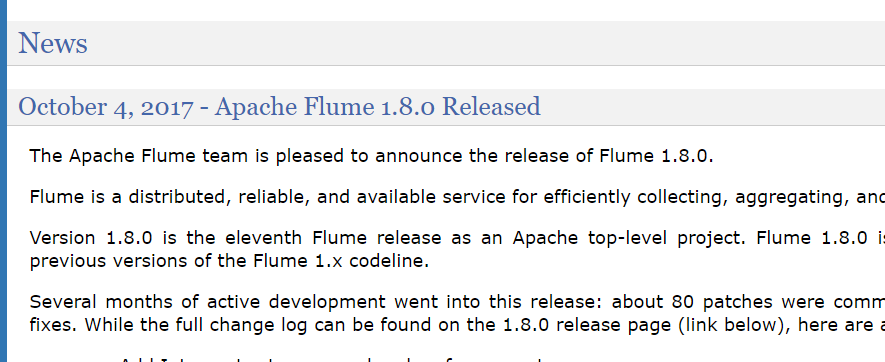
由于0.8的支持已经过时，所以我们选择1.1.1的kafka版本





### 1.5.2 Flume版本

选择1.8版本。而1.7版本只支持0.8的kafka



### 1.5.3 SpringBoot版本

Spring与后台大数据是解耦的，所以选最新的版本就可以了

### 1.5.4 产品选型列表

|  |  |
| --- | --- |
| 产品 | 版本 |
| Nginx | 1.12.2 |
| Hadoop | 2.7.2 |
| Spring Cloud | Finchley.SR1 |
| Spark | 2.3.1 |
| Flume | 1.8 |
| Kafka | 1.1.1 |
| Hive | 1.2.2 |
| Presto | presto-server-0.211, |
| Mongo | 4.0.2 |

# 第2章 预备知识

## 2.1 Flume

### 2.1.1 Flume 组件介绍

1. 1-8 Flume组件介绍

|  |  |
| --- | --- |
| 名称 | 解析 |
| Event | 一个数据单元，带有一个可选的消息头，其实就是一条消息，一个日志；（Kafka 的消息没有消息头，因此，Flume 的消息进入 Kafka 后，消息头就丢失了） |
| Flow | 数据流，Event 从源点到达目的点的迁移的抽象 |
| Client | 操作位于源点处的 Event，将其发送到 Flume Agent |
| Agent | 一个独立的 Flume 进程，包含组件 Source、Channel、Sink |
| Source | 用来获取 Event 并写入 Channel |
| Channel | 中转 Event 的一个临时存储，保存有 Source 组件传递过来的Event，可以认为是一个队列 |

### 2.1.2 Flume 组件选择

1. 多层 Flume

第一层 agent：

**Source**：TailDirSource

**Channel**：FileChannel

**Sink**：AvroSink

注意，TailDirSource 是 Flume 1.7 开始提供的 Source 组件，在 1.6 中并没有.

第二层 agent：

**Source**：AvroSource

**Channel**：FileChannel

**Sink**：KafkaSink

2. 单层 Flume

**Source**：TailDirSource

**Channel**：FileChannel

**Sink**：KafkaSink

### 2.1.3 Flume 采集系统组件解析

* 1. Source

1. Avro Source

侦听 Avro 端口并从外部 Avro 客户端流接收事件。当与另一个（上一层）Flume 代理的内置 Avro Sink 配对时，它可以创建分层收集拓扑。

1. Taildir Source
2. Flume1.7 之前如果想要监控一个文件新增的内容，我们一般采用的 source 为 exec tail ,但是这会有一个弊端，就是当你的服务器宕机重启后，此时数据读取还是从头开始，这显然不是我们想看到的 在 Flume1.7 没有出来之前我们一般的解决思路为：当读取一条记录后，就把当前的记录的行号记录到一个文件中，宕机重启时，我们可以先从文件中获取到最后一次读取文件的行数，然后继续监控读取下去。保证数据不丢失、不重复。
3. Flume1.7 时新增了一个 source 的类型为 taildir，它可以监控一个目录下的多个文件，并且实现了实时读取记录保存的断点续传功能。

但是 Flume1.7 中如果文件重命名，那么会被当成新文件而被重新采集。

* 1. Channel

1. Memory Channel

Memory Channel 把 Event 保存在内存队列中，该队列能保存的 Event 数量有最大值上限。由于 Event 数据都保存在内存中，Memory Channel 有最好的性能，不过也有数据可能会丢失的风险，如果 Flume 崩溃或者重启，那么保存在 Channel 中的 Event 都会丢失。同时由于内存容量有限，当 Event 数量达到最大值或者内存达到容量上限，Memory Channel 会有数据丢失。

1. File Channel

File Channel 把 Event 保存在本地硬盘中，比 Memory Channel 提供更好的可靠性和可恢复性，不过要操作本地文件，性能要差一些。

1. Kafka Channel

Kafka Channel 把 Event 保存在 Kafka 集群中，能提供比 File Channel 更好的性能和比 Memory Channel 更高的可靠性。

* 1. Sink

1. Avro Sink

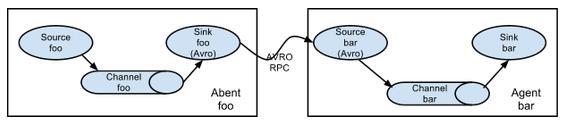
Avro Sink 是 Flume 的分层收集机制的重要组成部分。发送到此接收器的 Flume 事件变

1. Avro 事件，并发送到配置指定的主机名/端口对。事件将从配置的通道中按照批量配置的批量大小取出。
2. Kafka Sink

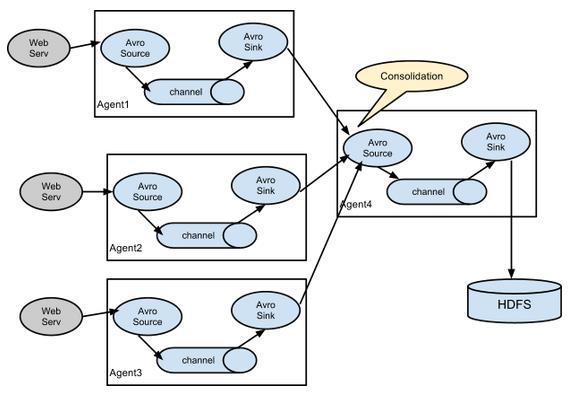
Kafka Sink 将会使用 FlumeEvent header 中的 topic 和 key 属性来将 event 发送给 Kafka。如果 FlumeEvent 的 header 中有 topic 属性，那么此 event 将会发送到 header 的 topic 属性指定的 topic 中。如果 FlumeEvent 的 header 中有 key 属性，此属性将会被用来对此 event 中的数据指定分区，具有相同 key 的 event 将会被划分到相同的分区中，如果 key 属性 null，那么 event 将会被发送到随机的分区中。

可以通过自定义拦截器来设置某个 event 的 header 中的 key 或者 topic 属性。

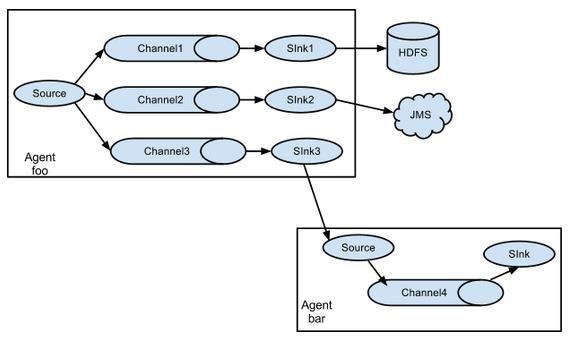
### 2.1.4 Flume 拓扑结构

****

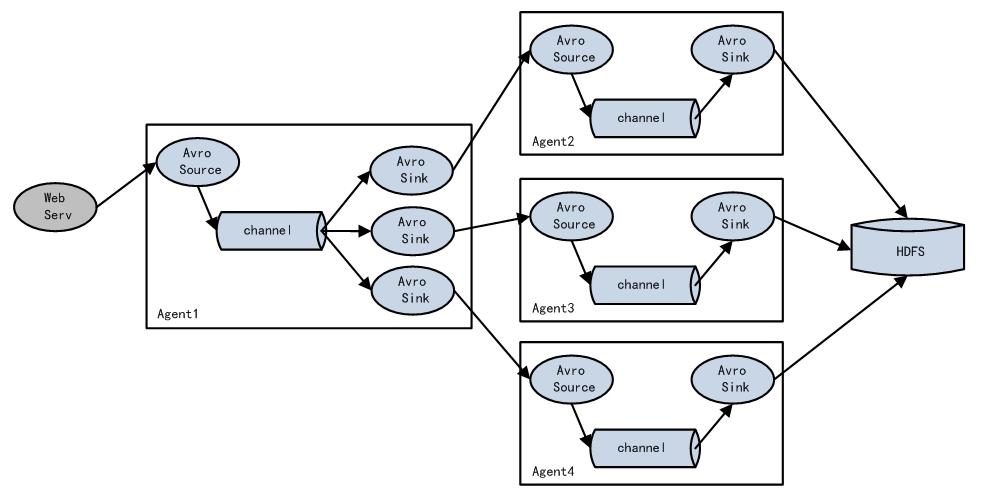
1. 2-1 Flume Agent 连接



1. 2-2 Flume Agent 聚合



1. 2-3 Flume 多路 Flow

****

1. 2-4 Flume 负载均衡

### 2.1.5 Flume Source,Channel处理器,拦截器和 Channel 选择之间的交互

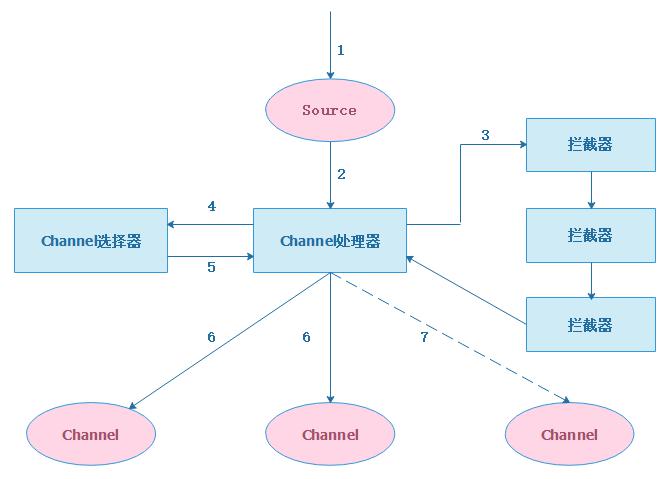
****

图 2-5 Flume 负载均衡

1. 接收事件；
2. 处理事件；
3. 将事件传递给拦截器链；
4. 将每个事件传递给 Channel 选择器；

1. 返回写入事件的 Channel 列表；
2. 将所有事件写入每个必需的 Channel，只有一个事务被打开；对于每个 Channel，所有事件都写为事务的一部分；
3. 利用可选 Channel 重复相同动作。

Flume 本身不限制 Agent 中 Source、Channel 和 Sink 的数量。因此 Flume Source 可以接收事件，并可以通过配置将事件复制到多个目的地。这使得 Source 通过 Channel 处理器、拦截器和 Channel 选择器，写入数据到 Channel 成为可能。

每个 Source 都有自己的 Channel 处理器。每次 Source 将数据写入 Channel，它是通过委派该任务到其 Channel 处理器来完成的。然后，Channel 处理器将这些事件传到一个或多个 Source 配置的拦截器中。

拦截器是一段代码，可以基于某些它完成的处理来读取时间和修改或删除时间。基于某些标准，如正则表达式，拦截器可以用来删除事件，为事件添加新报头或移除现有的报头等。每个 Source 可以配置成使用多个拦截器，按照配置中定义的顺序被调用，将拦截器的结果传递给链的下一个单元。这就是所谓的责任链的设计模式。一旦拦截器处理完事件，拦截器链返回的事件列表传递到 Channel 列表，即通过 Channel 选择器为每个事件选择的 Channel。

Source 可以通过处理器-拦截器-选择路由器写入多个 Channel。Channel 选择器是决定每个事件必须写入到 Source 附带的哪个 Channel 的组件。因此拦截器可以用来插入或删除事件中的数据，这样 Channel 选择器可以应用一些条件在这些事件上，来决定事件必须写入哪

1. Channel。Channel 选择器可以对事件应用任意过滤条件，来决定每个事件必须写入哪些

Channel，以及哪些 Channel 是必需的或可选的。

写入到必需的 Channel 失败将会导致 Channel 处理器抛出 ChannelException，表明 Source 必须重试该事件（实际上，所有的时间都在那个事务中），而未能写入可选 Channel 失败仅仅忽略它。一旦写出事件，处理器将会对 Source 指示成功状态，可能发送确认（ACK）给发送该事件的系统，并继续接受更多的事件。

1. 拦截器

拦截器（Interceptor）是简单插件式组件，设置在 Source 和 Source 写入数据的 Channel

之间。Source 接收到的事件在写入到对应的 Channel 之前，拦截器都可以转换或删除这些事件。每个拦截器实例只处理同一个 Source 接收到的事件。拦截器可以基于任意标准或转换事件，但是拦截必须返回尽可能多（或尽可能少）的事件，如同原始传递过来的事件。

多个拦截器组成一个有序的拦截器链。在一个链条中，可以添加任意数量的拦截器去转换从单个 Source 中来的事件。Source 将同一个事务的所有事件传递给 Channel 处理器，进而传递给拦截器链条，然后事件被传递给链条中的第一个拦截器。通过拦截器转换时间产生的一系列事件，传递到链条的下一个拦截器，以此类推。链条最后一个拦截器返回的最终事

件列表写入到 Channel 中。

因为拦截器必须在事件写入 Channel 之前完成转换操作，只有当拦截器已成功转换事件

后，RPC Source（和任何其他可能产生超时的 Source）才会响应发送事件的客户端或 Sink。

因此，在拦截器中进行大量重量级的处理并不是和一个好主意。如果拦截器中的处理时重量

级的、耗时的，那么需要相应地调整超时时间属性。

Flume 配置文件中，所有拦截器通用的唯一配置参数是 type 参数，改参数必须是拦截器的别名或者 Builder 类的完全限定类名（FQCN），该 Builder 类用于创建拦截器。正如前面提到的，可以设置任意数量的拦截器连接到单个的 Source。

拦截器是需要命名的组件，每个拦截器实例必须限定一个名字。为了给 Source 添加拦截器，需要列出 Source 应该连接的拦截器名字，这些拦截器就是 Source 应该连接到 Source 配置中 interceptors 参数的值代表的拦截器。原配置中以 interceptors. 开头的、后面跟着拦截器名称和参数的所有值都传递给拦截器。

1. Channel 选择器

Channel 选择器是决定 Source 接收的一个特定事件写入哪些 Channel 的组件。它们告知 Channel 处理器，然后将事件写入到每个 Channel。

由于 Flume 并不是两阶提交（不会等所有事件都写入成功后再一起提交，而是写一个提交一个），事件被写入到一个 Channel，然后在事件被写入到下一个 Channel 之前提交。如果写入一个 Channel 时出现故障，可能已经发生在其他 Channel 的相同事件的写入不能被

回滚。当这样的故障发生时，Channel 处理器抛出 ChannelException 并且事务失败。如果 Source 试图再次写入相同的事件（在大多数情况下，它会重试，只有类似 Syslog、Exec 等 Source 不能重试，因为没有办法再次生成相同的数据），重复的事件将写入到 Channel，而先前的提交实际上是成功的，这是在 Flume 管道发生重复的一种情况。

Channel 选择器配置是通过 Channel 处理器完成的，虽然配置看起来像 Source 子组件的配置。传递到 Channel 选择器的所有参数作为 Source 的上下文中的参数使用 selector 后缀传递。对于每个 Source，选择器通过使用一个配置参数 type 指定。Channel 选择器可以指定一

1. Channel 是必需的（required），另一组是可选的（optional）。

Flume 内置两种 Channel 选择器：replicating 和 mutiplexing。如果 Source 的配置中没有指定选择器，那么会自动使用复制 Channel 选择器。

replicating Channel 选择器复制每个事件到 Source 的 channels 参数所指定的所有 Channel 中。

multiplexing Channel 选择器是一种专门用于动态路由事件的 Channel 选择器，通过选择事件应该写入的 Channel，基于一个特定的事件头的值进行路由。

### 2.1.6 Flume Sink、Sink 运行器、Sink 组和 Sink 处理器

1. 2-6 Flume Sink、Sink 运行器、Sink 组合 Sink 处理器

Sink 运行器（Sink Runner）运行一个 Sink 组（Sink Group），Sink 组可以含有一个或多个 Sink。如果组中只存在一个 Sink，那么没有组将会更有效率。Sink 运行器仅仅是一个询问 Sink 组（或 Sink）来处理下一批事件的线程。每个 Sink 组有一个 Sink 处理器（Sink Processor），处理器选择组中的 Sink 之一去处理下一个事件集合。每个 Sink 只能从一个

Channel 获取数据（一个 Sink 只能有一个 Channel），尽管多个 Sink 可以从同一个 Channel 获取数据。选定的 Sink（或如果没有组，唯一的 Sink）从 Channel 中接受事件，并将事件写入到下一阶段或最终目的地。

1. Sink 组

Flume 配置框架为每个 Sink 组实例化一个 Sink 运行器，来运行 Sink 组。每个 Sink 组可以包含任意数量的 Sink。Sink 运行器持续请求 Sink 组，要求其中的一个 Sink 从自己的 Channel 中读取事件。Sink 组通常用于 RPC Sink，在层之间以负载均衡或故障转移方式发送

数据。

Sink 组中的每个 Sink 必须单独进行配置。这包括：Sink 从哪个 Channel 读取，写数据到哪些主机或者集群。在理想情况下，如果 Sink 组中建立了几个 Sink，所有的 Sink 将从相同的 Channel 读取，这将有利于在当前层以合理的速度清除数据，确保将要被发送到多台集群的数据，以一种支持负载均衡和故障转移的方式进行发送。

1. Load-Balancing Sink 处理器

Load-Balancing Sink 处理器从所有的 Sink 中选择一个 Sink，处理来自 Channel 的事件。

Sink 选择的顺序可以为 random 或者 round-robin。如果顺序被设置为 random，那么将随机从 Sink 组的 Sink 中选择一个，用来从自己的 Channel 中移除事件并将它们写出。round-robin选项使 Sink 以循环的方式被选择：每个选择循环调用定义 Sink 组中指定顺序 Sink 的 process

方法。如果 Sink 写入到一个失败的 Agent 或者速度太慢的 Agent，会导致超时，Sink 处理器会选择另一个 Sink 写数据。

Sink 处理器可以配置将失败的 Sink 加入黑名单，回退时间以指数方式增长直到达到上限值。这能确保相同的 Sink 不会循环重复尝试且不浪费资源，直到回退时间过期。

### 2.1.7 Flume Taildir Source 移植

如果使用 0.8 版本 Kafka 并配套 1.6 版本 Flume，由于 Flume 1.6 版本没有 Taildir Source 组件，因此，需要将 Flume 1.7 中的 Taildir Source 组件源码编译打包后，放入 Flume1.6 安装目录的 lib 文件目录下。

所谓移植，就是将 Flume1.7 版本中 Taildir Source 的相关文件全部提取出来，然后将这些文件放入新建的项目中进行编译打包，将打包出的 jar 包放入 Flume1.6 安装目录的 lib 目录下即可。

在本项目中，已经将 Taildir Source 的源码放入了 taildirsource 项目中，直接编译项目，打包后放入 Flume1.6 安装目录的 lib 文件目录下即可。

1. Flume 配置文件中指定全类名即可使用 Taildir Source 组件。

a1.sources.r1.type = com.atguigu.flume.source.TaildirSource

如果使用 0.10 版本 Kafka 并配套 1.7 及以上版本的 Flume，那么无需手动移植，1.7 及以上版本的 Flume 自带 Taildir Source。

Taildir Source 有三个坑：不支持 Windows，只支持 UTF-8，文件重命名后重新读取。

### 2.1.8 Flume Taildir Source 源码修改

TaildirSource 中维护着两个核心结构，一个是记录着所有监控文件相关信息的 Map 结构，一个是实时记录每个文件读取位置的 position file。

1. TaildirSource 的源码中，维护了一个<inode，TailFile>为元素的 Map，用来记录指定目录下的所有采集文件，当有新文件时，会写入此 Map 中，因此，所有被监控文件的信息都集中记录在这一 Map 结构中，当有新文件时，此 Map 中会产生新的 K-V 对。

（Linux 中储存文件元数据的区域就叫做 inode，每个 inode 都有一个号码，操作系统用 inode 号码来识别不同的文件，Unix/Linux 系统内部不使用文件名，而使用 inode 号码来识别文件，当文件名称改变时，inode 并不会改变）

Map 的 key 为 inode，value 如下：

private RandomAccessFile raf;

private final String path;

private final long inode;

private long pos;

private long lastUpdated;

private boolean needTail;

private final Map<String, String> headers;

private byte[] buffer;

private byte[] oldBuffer;

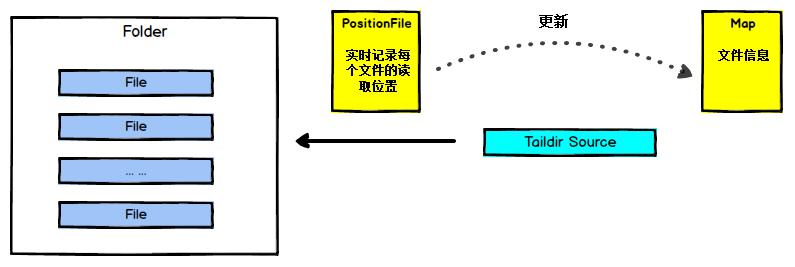
private int bufferPos;

private long lineReadPos;

可以看到 value 中记录了文件路径、inode、读取位置 pos、更新时间等信息。

在维护上述的 Map 结构时，判断文件是否为新文件时加入了 inode 和文件名的双重判断，这使得一旦文件重命名，将会被判定为新文件，因此此处需要修改 updateTailFiles()。

与此同时，TaildirSource 在运行过程中维护了一个 position file，对于文件的读取位置会实时记录到 position file 中，包括 inode、pos 和 path，然后根据 position file 中的内容更新 Map 中的数据。



1. TaildirSource 的源码中，通过 inode 和文件名来进行唯一文件的判定，也就是说，一旦文件改名，那么 TaildirSource 会将它当成新文件而重新读取，这是不合理的，会造成数据的重复读取。

为了解决这个问题，我们对 ReliableTaildirEventReader.java 文件进行修改，具体是对此文件中的 updateTailFiles 函数和 loadPositionFile 函数进行修改。

1. updateTailFiles()

UpdateTailFiles 函数扫描指定的监控目录是否产生了新文件或者文件是否被追加了内容。

UpdateTailFiles 函数使用了一个 Map 结构保存所有被监视文件的信息，对 Map 的每个 TailFile 中的 pos 进行更新时，传入的是 position file 中的旧文件名，而在 updateTailFiles()中已经修改了源码，修改后重命名会覆盖 Map 中的原始文件，TailFile 中的信息同步更新，不会再将重命名的文件当成新文件，而原始的 loadPositionFile()会将 json 中的旧文件名传入

updatePos()，如果不修改会导致名称不匹配而更新 pos 失败，因此也应该对 loadPositionFile() 进行修改。

代码清单 3-4 updateTailFiles()

/\*\*

1. Update tailFiles mapping if a new file is created or appends are detected
2. to the existing file.
3. 扫描指定的监控目录是否产生了新文件或者文件是否被追加了内容

\*/

public List<Long> updateTailFiles (boolean skipToEnd) throws IOException { updateTime = System.currentTimeMillis();

List<Long> updatedInodes = Lists.newArrayList();

for (TaildirMatcher taildir : taildirCache) {

Map<String, String> headers = headerTable.row(taildir.getFileGroup());

//遍历所有匹配的文件

for (File f : taildir.getMatchingFiles()) {

//得到本地文件的 inode(储存文件元信息的区域就叫做 inode，inode 包含除了文件名以外的所有文件信息) //文件由唯一的 inode，不论文件是否重命名，inode 不变

long inode = getInode(f);

//tailFiles 是一个 Map，以 inode 为 key，以 TailFile 为 value //第一次遍历，此 inode 对应的 Map 项肯定不存在

TailFile tf = tailFiles.get(inode);

//源码中导致文件重命名后被重新读取的罪魁祸首

1. 当文件重命名后， !tf.getPath().equals(f.getAbsolutePath())为 True，那么就会创建新的TailFile，然后覆盖 Map 中原有的 key-value 对

//if (tf == null || !tf.getPath().equals(f.getAbsolutePath())) { if (tf == null) {

//如果 Map 中对应文件为空，那么就创建一个 TailFile 对象

//skipToEnd 可配置，决定是否从文件开始位置读取数据还是直接跳到文件结尾 long startPos = skipToEnd ? f.length() : 0;

//openFile 中根据传入的参数 new 了一个新的 TailFile tf = openFile(f, headers, inode, startPos);

} else { //不为空时进入

//如果文件重命名则进入此分支，由于是对于源码的修改导致重命名后进入，必须再次修改源码以处理重命名

情况

//判断此文件的更新时间是否比 Map 中存储的文件更新时间要新

boolean updated = tf.getLastUpdated() < f.lastModified(); if (updated) {

//如果 Map 含有对应项，但是得到的 tf 中封装的文件为 null，需要重新创建 tf if (tf.getRaf() == null) {

tf = openFile(f, headers, inode, tf.getPos());

}

* 1. 如果 Map 中记录的读取位置 Pos 已经超过了文件长度，那么设置 Map 中的 Pos 值为 0，即重新从 0 开

始

if (f.length() < tf.getPos()) {

logger.info("Pos " + tf.getPos() + " is larger than file size! "

1. "Restarting from pos 0, file: " + tf.getPath() + ", inode: " + inode); tf.updatePos(tf.getPath(), inode, 0);

}

}

//重命名后，Map 中的文件名还是老的文件名，因此使用 openFIle 重新创建 TailFile 用来替换原数据 //modify by zhangpeng

if (!tf.getPath().equals(f.getAbsolutePath())) { tf = openFile(f, headers, inode, tf.getPos());

}

//modify by zhangpeng end

tf.setNeedTail(updated);

}

//将 inode 及其对应的 tf 加入 Map 中

tailFiles.put(inode, tf);

updatedInodes.add(inode);

}

}

return updatedInodes;

}

2. loadPositionFile ()

根据 position file 更新 TailFile Map 中每个文件的 pos：

代码清单 3-5 loadPositionFile()

/\*\*

1. Load a position file which has the last read position of each file.
2. 加载并解析记录了每个文件最新读取位置的 position file
3. If the position file exists, update tailFiles mapping.
4. 如果 position file 存在则更新 tailFiles 映射

\*/

public void loadPositionFile(String filePath) {

Long inode, pos;

String path;

FileReader fr = null;

JsonReader jr = null;

//对 position file 进行读取和解析

try {

fr = new FileReader(filePath);

jr = new JsonReader(fr);

jr.beginArray();

while (jr.hasNext()) {

inode = null;

pos = null;

path = null;

jr.beginObject();

while (jr.hasNext()) {

switch (jr.nextName()) {

case "inode":

inode = jr.nextLong();

break;

case "pos":

pos = jr.nextLong();

break;

case "file":

path = jr.nextString();

break;

}

}

jr.endObject();

for (Object v : Arrays.asList(inode, pos, path)) { Preconditions.checkNotNull(v, "Detected missing value in position file. "

+ "inode: " + inode + ", pos: " + pos + ", path: " + path);

}

//判断 position file 中的 inode 是否存在于 TailFile Map 中 TailFile tf = tailFiles.get(inode);

//根据对 updatePos 的分析，当出现重命名时，position file 中的 path 项对应的文件名是旧文件名，而通过 updateTailFiles()已经将 Map 中的文件名更新成了重命名后的文件名

//因此，为了 updatePos 能够顺利更新 pos，应该传入 tf.getPath()，即新文件名，tailfile 与 tailfile

自身的文件名的比较必然是相等的

//modify by zhangpeng

//if (tf != null && tf.updatePos(path, inode, pos)) {

if (tf != null && tf.updatePos(tf.getPath(), inode, pos)) { tailFiles.put(inode, tf);

} else {

logger.info("Missing file: " + path + ", inode: " + inode + ", pos: " + pos);

}

}

jr.endArray();

} catch (FileNotFoundException e) {

logger.info("File not found: " + filePath + ", not updating position"); } catch (IOException e) {

logger.error("Failed loading positionFile: " + filePath, e);

} finally { try {

if (fr != null) fr.close(); if (jr != null) jr.close();

} catch (IOException e) {

logger.error("Error: " + e.getMessage(), e);

}

}

}

代码清单 3-6 loadPositionFile()

public boolean updatePos(String path, long inode, long pos) throws IOException {

1. 注意 updatePos 判断是否更新读取位置时检查的是 inode 和文件名，如果重命名而传入的仍是 position file 中记录的原来的名称，那么此处将不会更新 pos

if (this.inode == inode && this.path.equals(path)) {

setPos(pos);

updateFilePos(pos);

logger.info("Updated position, file: " + path + ", inode: " + inode + ", pos: " + pos);

return true;

}

return false;

}

### 2.1.9 Flume 的停止

使用 kill 停止 Flume 进程。

不可使用 kill -9，因为 Flume 内部注册了很多钩子函数执行善后工作，如果使用 kill -9 会导致钩子函数不执行，使用 kill 时，Flume 内部进程会监控到用户的操作，然后调用钩子函数，执行一些善后操作，正常退出。

### 2.1.10 Flume 数据丢失问题的讨论

在一些网络资料中提出当 Flume 的数据量达到 70MB/s 以上时，就会出现丢失数据的情况，但是根据 Flume 的架构原理，Flume 是不可能丢失数据的，其内部有完善的事务机制，

Source 到 Channel 是事务性的，Channel 到 Sink 是事务性的，因此这两个环节不会出现数据的丢失，唯一可能丢失数据的情况是 Channel 采用 memoryChannel，agent 宕机导致数据丢

失，或者 Channel 存储数据已满，导致 Source 不再写入，未写入的数据丢失。并且，在实际的项目开发和运行过程中，并没有出现过 Flume 丢失数据的情况（以滴滴为例），因此， Flume 在数据量大的时候丢失数据的论断还有待商榷。

Flume 不会丢失数据，但是有可能造成数据的重复，例如数据已经成功由 Sink 发出，但是没有接收到响应，Sink 会再次发送数据，此时可能会导致数据的重复。

## 2.2 Kafka

### 2.2.1 Kafka 生产者

1. 生产者模式

Kafka 生产者有两种工作模式，即同步模式和异步模式。

异步模式下，生产者客户端应用程序不需要提供回调函数，生产者客户端应用程序发送完一条消息后，不需要关心服务器端处理完了没有，可以接着发送下一条消息。服务端在处理完每一条消息后，会自动触发回调函数，返回响应结果给客户端。

同步模式下，生产者发送完一条消息后，必须等待服务端返回响应结果，然后才能发送下一条消息。

1. 生产者消息的发送

KafkaProducer 只用了一个 send 方法，就可以完成同步和异步两种模式的消息发送。生产者客户端对象 KafkaProducer 的 send 方法的处理逻辑是：首先序列化消息的 key 和value（消息必须序列化成二进制流的形式才能在网络中传输），然后为每一条消息选择对应的分区（表示要将消息存储到 Kafka 集群的哪个节点上），最后通知发送线程发送消息。

1. 生产者参数设置

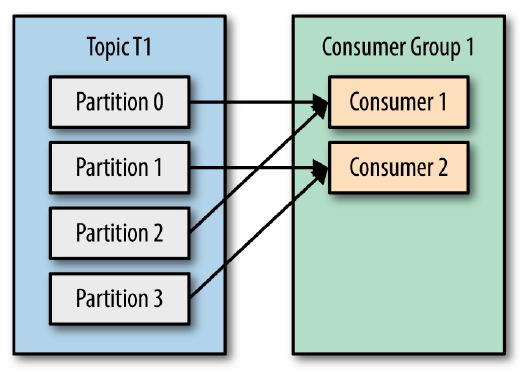
request.requred.acks：

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

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

-1：producer 等待 broker 的 ack，partition 的 leader 和 follower 全部落盘成功后才返回 ack，数据一般不会丢失，延迟时间长但是可靠性高；

### 2.2.2 Kafka 分区分配策略



1. Kafka 内部存在两种默认的分区分配策略：Range 和 RoundRobin。当以下事件发生时，Kafka 将会进行一次分区分配：

* 同一个 Consumer Group 内新增消费者
* 消费者离开当前所属的 Consumer Group，包括 shuts down 或 crashes
* 订阅的主题新增分区

将分区的所有权从一个消费者移到另一个消费者称为重新平衡（rebalance），如何 rebalance 就涉及到下面提到的分区分配策略。下面我们将详细介绍 Kafka 内置的两种分区分配策略。本文假设我们有个名为 T1 的主题，其包含了 10 个分区，然后我们有两个消费

者（C1，C2）来消费这 10 个分区里面的数据，而且 C1 的 num.streams = 1，C2 的 num.streams = 2。

1. Range strategy

Range 策略是对每个主题而言的，首先对同一个主题里面的分区按照序号进行排序，并

对消费者按照字母顺序进行排序。在我们的例子里面，排完序的分区将会是 0, 1, 2, 3, 4, 5, 6,

7, 8, 9；消费者线程排完序将会是 C1-0, C2-0, C2-1。然后将 partitions 的个数除于消费者线程的总数来决定每个消费者线程消费几个分区。如果除不尽，那么前面几个消费者线程将会多消费一个分区。

在我们的例子里面，我们有 10 个分区，3 个消费者线程， 10 / 3 = 3，而且除不尽，那么消费者线程 C1-0 将会多消费一个分区，所以最后分区分配的结果看起来是这样的：

|  |  |  |
| --- | --- | --- |
| C1-0 | 将消费 0, 1, 2, | 3 分区 |
| C2-0 | 将消费 4, 5, 6 | 分区 |
| C2-1 | 将消费 7, 8, 9 | 分区 |

假如我们有 11 个分区，那么最后分区分配的结果看起来是这样的：

C1-0 将消费 0, 1, 2, 3 分区

C2-0 将消费 4, 5, 6, 7 分区

C2-1 将消费 8, 9, 10 分区

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C1-0 | 将消费 T1 | 主题的 0, 1, 2, 3 分区以及 T2 主题的 | | 0, 1, 2, 3 分区 |
| C2-0 | 将消费 T1 | 主题的 4, 5, 6 | 分区以及 T2 主题的 4, | 5, 6 分区 |
| C2-1 | 将消费 T1 | 主题的 7, 8, 9 | 分区以及 T2 主题的 7, | 8, 9 分区 |

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

2. RoundRobin strategy

使用 RoundRobin 策略有两个前提条件必须满足：

* 同一个 Consumer Group 里面的所有消费者的 num.streams 必须相等；
* 每个消费者订阅的主题必须相同。

所以这里假设前面提到的 2 个消费者的 num.streams = 2。RoundRobin 策略的工作原理：

将所有主题的分区组成 TopicAndPartition 列表，然后对 TopicAndPartition 列表按照

hashCode 进行排序，这里文字可能说不清，看下面的代码应该会明白：

val allTopicPartitions = ctx.partitionsForTopic.flatMap { case(topic, partitions) => info("Consumer %s rebalancing the following partitions for topic %s: %s"

.format(ctx.consumerId, topic, partitions))

partitions.map(partition => {

TopicAndPartition(topic, partition)

})

}.toSeq.sortWith((topicPartition1, topicPartition2) => { /\*

1. Randomize the order by taking the hashcode to reduce the likelihood of all partitions of a given topic ending
2. up on one consumer (if it has a high enough stream count).

\*/

topicPartition1.toString.hashCode < topicPartition2.toString.hashCode })

最后按照 round-robin 风格将分区分别分配给不同的消费者线程。

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

|  |  |  |
| --- | --- | --- |
| C1-0 | 将消费 T1-5, T1-2, T1-6 | 分区； |
| C1-1 | 将消费 T1-3, T1-1, T1-9 | 分区； |
| C2-0 | 将消费 T1-0, T1-4 分区； | |

C2-1 将消费 T1-8, T1-7 分区；

### 2.2.3 Kafka 消费者

使用消费者组实现消息队列的两种模式

Kafka 集群的数据需要被不同类型的消费者使用，而不同类型的消费者处理逻辑不

同。Kafka 使用消费组的概念，允许一组消费者进程对消费工作进行划分。每个消费者都可

以配置一个所属的消费组，并且订阅多个主题。Kafka 会发送每条消息给每个消费组中的一

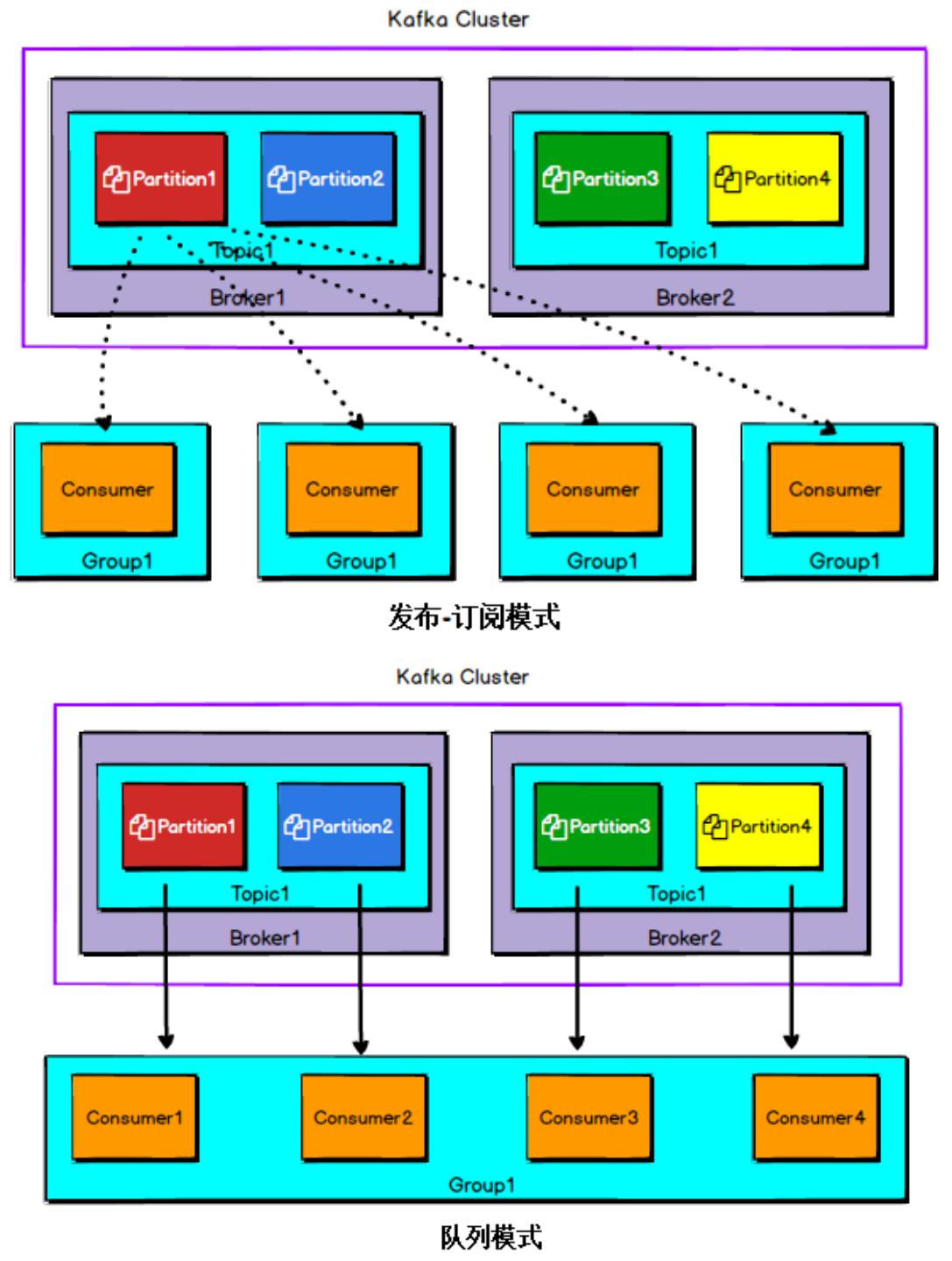
个消费者进程（ 同一条消息广播给多个消费组，单播给同一组中的消费者）。被订阅主题

的所有分区会平均地负载给订阅方，即消费组中的所有消费者。

Kafka 采用消费组保证了“一个分区只可被消费组中的一个消费者所消费” ，这意味着：

* 在一个消费组中，一个消费者可以消费多个分区。
* 不同的消费者消费的分区一定不会重复，所有消费者一起消费所有的分区。
* 在不同消费组中，每个消费组都会消费所有的分区。
* 同一个消费组下消费者对分区是互斥的，而不同消费组之间是共享的。

下图给出了多个消费者都在同一个消费组中，或者各向组成一个消费组的不同消费场景，这样 Kafka 也可以实现传统消息队列的发布——订阅模式和队列模式。



**发布－订阅模式**：同一条消息会被多个消费组消费，每个消费组只有一个消费者，实现

**广播**。

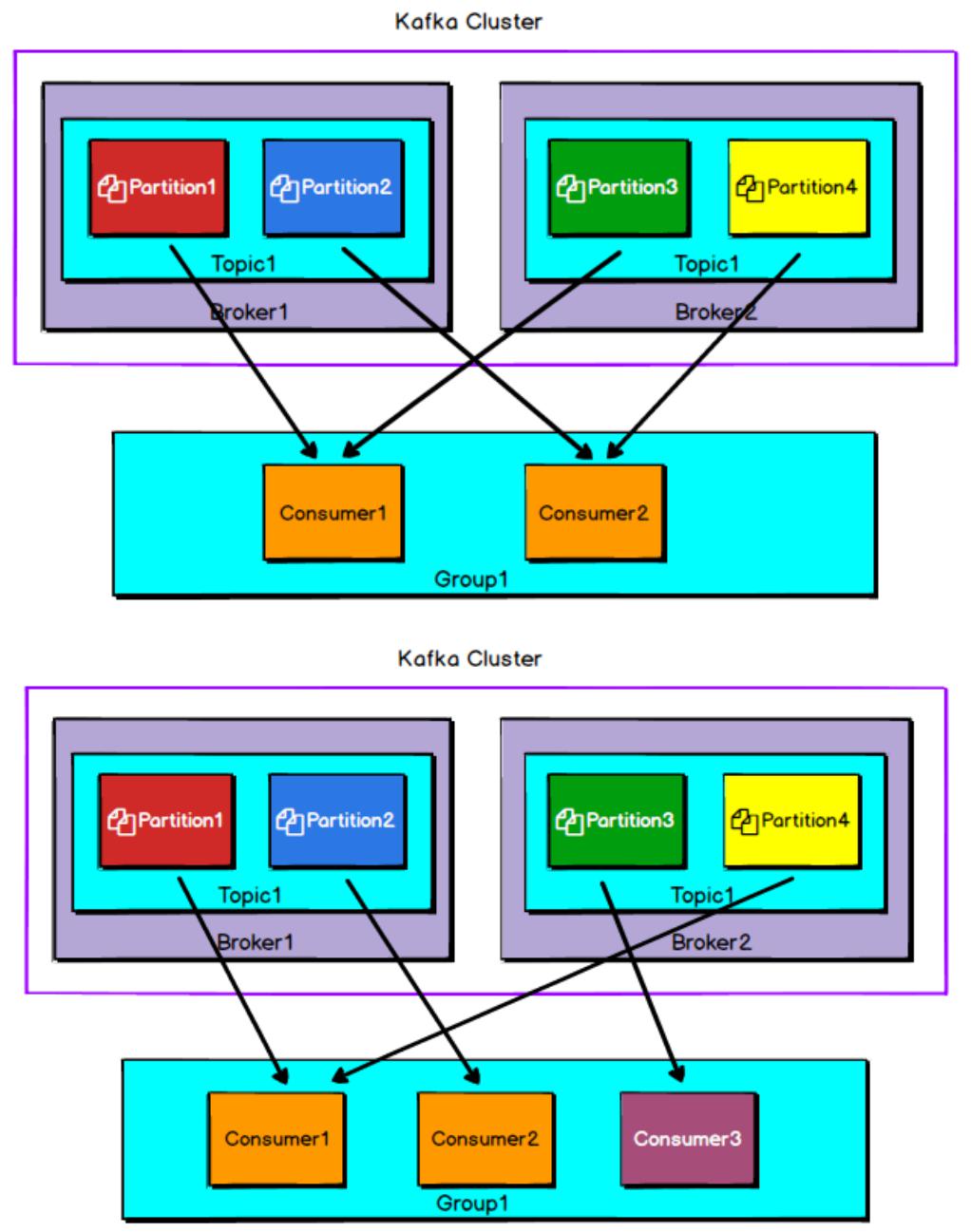
**队列模式**：只有一个消费组、多个消费者一条消息只被消费组的一个消费者消费，实现

**单播**。

消费者组再平衡实现故障容错

消费者是客户端的监务处理逻辑程序，因此要考虑消费者的故障容错。一个消费组有多个消费者，因此消费组需要维护所有的消费者。如果一个消费者宕机了，分配给这个消费者的分区需要重新分配给相同组的其他消费者；如果一个消费者加入了同一个组，之前分配给其他消费组的分区需要分配给新加入的消费者。

一旦有消费者加入或退出消费组，导致消费组成员列表发生变化，消费组中所有的消费者就要执行再平衡（ rebalance ） 工作。如果订阅主题的分区有变化，所有的消费者也都要再平衡。如下图所示，在加入一个新的消费者后，需要为所有的消费者重新分配分区， 因此所有消费者都会执行再平衡。



消费者再平衡前后分配到的分区会完全不同，那么消费者之间如何确保各向消费消息的平滑过渡呢？假设分区 P1 原先分配给消费者 C1 ，再平衡后被分配给消费者 C2 。如果再

平衡前消费者 C1 保存了分区 P1 的消费进度，再平衡后消费者 C2 就可以从保存的进度位置继续读取分区 P1，保证分区 P1 不管分配给哪个消费者，消息都不会丢失，实现了消费者的故障容错。

保存消费进度

生产者的提交日志采用递增的偏移量，连同消息内容一起写入本地日志文件。生产者客

户端不需要保存偏移量相关的状态，消费者客户端则要保存消费消息的偏移量即消费进度。

消费进度表示消费者对一个分区已经消费到了哪里。

由于消费者消费消息的最小单元是分区，因此每个分区都应该记录消费进度，而且消费进度应该面向消费组级别。假设面向的是消费者级别，再平衡前分区 P1 只记录到消费者 C1

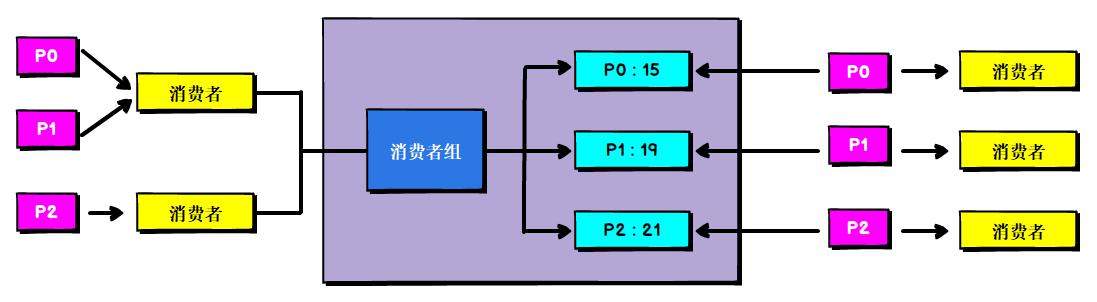
中，再平衡后分区 P1 属于消费者 C2。但是这样一来，分区 P1 和消费者 C2 之前没有记录任何信息，就无法做到无缝迁移。而如果针对消费组，因为消费者 C1 和消费者 C2 属于同一个消费组，再平衡前记录分区 P1 到消费组 l ，再平衡后消费者 C2 可以正常地读取消费

1. l 的分区 P1 进度，还是可以准确还原出这个分区在消费组 l 中的最新进度的。总结下，虽然分区是以消费者级别被消费的，但分区的消费进度要保存成消费组级别的。

消费者对分区的消费进度通常保存在外部存储系统中，比如 ZK 或者 Kafka 的内部主

题(\_\_consumer\_offsets)。这样分区的不同拥有者总是可以读取同一个存储系统的消费进度，即使消费者成员发生变化，也不会影响消息的消费和处理。如下体所示，消费者消费消息时，

需要定时将分区的最新消费进度保存到 ZK 中。当发生再平衡时，消费者拥有的新分区消费进度都可以从 ZK 中读取出来，从而恢复到最近的消费状态。



由消费者保存消费进度的另一个原因是： 消费者消费消息是主动从服务端拉取数据，而不是由服务端向消费者推送数据。如果由服务端推送数据给消费者，消费者只负责接收数据，就不需要保存状态了。但后面这种方法会严重影响服务端的性能，因为要在服务端记录每条消息分配给哪个消费者，还要记录消费者消费到哪里了。

1. 消费者与 ZK 的关系

消费者除了需要保存消费进度到 ZK 中，它分配的分区也是从 ZK 读取的。ZK 不仅存储了 Kafka 的内部元数据，而且记录了消费组的成员列表、分区的消费进度、分区的所有者。

消费者要消费哪些分区的消息由消费组来决定，因为消费组管理所有的消费者，所以它需要知道集群中所有可用的分区和所有存活的消费者，才能执行分区分配算法，而这些信息都需要保存到 ZK 中。每个消费者都要在 ZK 的消费组节点下注册对应的消费者节点，在分配到不同的分区后，才会开始各自拉取分区的消息。

通常，客户端代码并不直接完成上面那些复杂的操作步骤，而是由服务端暴露出一个

API 接口，让客户端可以透明地和集群交互。这个 API 接口实际上属于客户端进程范畴，用

来和管理员以及数据存储节点通信。Kafka 提供了两种层次的客户端 API ： 如果消费者不

太关心消息偏移量的处理，可以使用高级 API ；如果想自定义消费逻辑，可以使用低级 API 。

**高级 API**：消费者客户端代码不需要管理偏移量的提交，并且采用了消费组的自动负载

均衡功能，确保消费者的增减不会影响消息的消费。高级 API 提供了从 Kafka 消费数据的

高层抽象。

**低级 API**：通常针对特殊的消费逻辑，比如消费者只想消费某些特定的分区。低级API

的客户端代码需要自己实现一些和 Kafka 服务端相关的底层逻辑，比如选择分区的主副本、处理主副本的故障转移等。

高级 API 的特点

优点

● 高级API写起来简单

● 不需要去自行去管理offset，系统通过zookeeper自行管理

● 不需要管理分区，副本等情况，系统自动管理

● 消费者断线会自动根据上一次记录在 zookeeper中的offset去接着获取数据（默认设置5s更新一下 zookeeper 中存的的offset）,版本为0.10.2

● 可以使用group来区分对访问同一个topic的不同程序访问分离开来（不同的group记录不同的offset，这样不同程序读取同一个topic才不会因为offset互相影响）

缺点

● 不能自行控制 offset（对于某些特殊需求来说）

● 不能细化控制如分区、副本、zk 等

低级 API 的特点

优点

● 能够开发者自己控制offset，想从哪里读取就从哪里读取。

● 自行控制连接分区，对分区自定义进行负载均衡

● 对 zookeeper 的依赖性降低（如：offset 不一定非要靠 zk 存储，自行存储offset 即可，比如存在文件或者内存中）

缺点

● 太过复杂，需要自行控制 offset，连接哪个分区，找到分区 leader 等

### 2.2.4 Kafka 高级消费者

高阶消费者是一把双刃剑，一方面简化了编程，一方面也由于编程者参与的功能过少，可控内容过少而造成很多问题。

自动负载均衡

高阶消费者为了简化编程，封装了一系列 API，这套 API 会均匀地将分区分配给消费者线程，消费者消费哪个分区不由消费者决定，而是由高阶 API 决定，如果有消费者线程挂掉了，高阶 API 会检测到，进而进行重新分配。高阶消费者 API 将大部分功能已经实现，因此，编程者编写高阶消费者的难度也随之降低，不需要关注分区的分配，只需要读取数据就好了。

高阶消费者 API 下，固定分区个数时如果消费者个数大于分区个数，那么将会有消费者空转，造成资源的浪费。

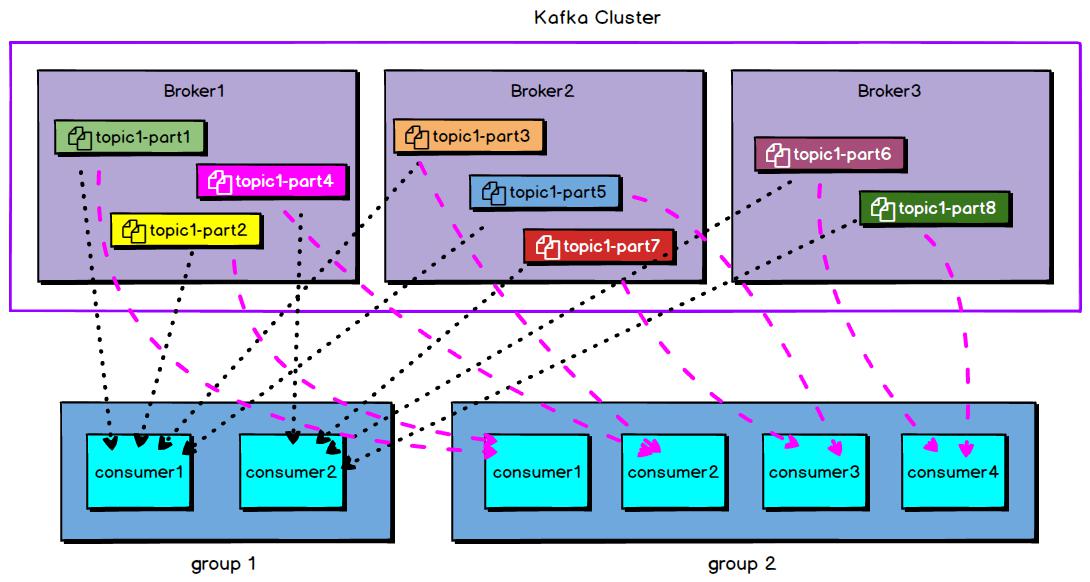


图 2-7 Kafka 高级消费者自动分配分区

如图 2-7 所示，在高级消费者的作用下，同一个 topic 的 8 个分区均匀的分布到了 3 个 broker 上。有两个消费者组，它们互不相干，对于每个 partition 每个消费者组维护着自己的offset，不同的消费者组可以同时消费同一个 partition。高级消费者 API 将每个 partition 均匀地分配到了每个消费者组的每个消费者上，push-group 是两个消费者，每个消费者消费 4 个， token-group 是四个消费者，每个消费者消费 2 个。

于此同时，高阶消费者 API 为用户提供了一个高可用机制，在不同机器运行相同消费

组时，如果有机器宕机，高阶消费者 API 会检测到，然后将宕机的机器所消费的分区重新分配给其他机器上的消费组的消费者，不会影响业务系统。

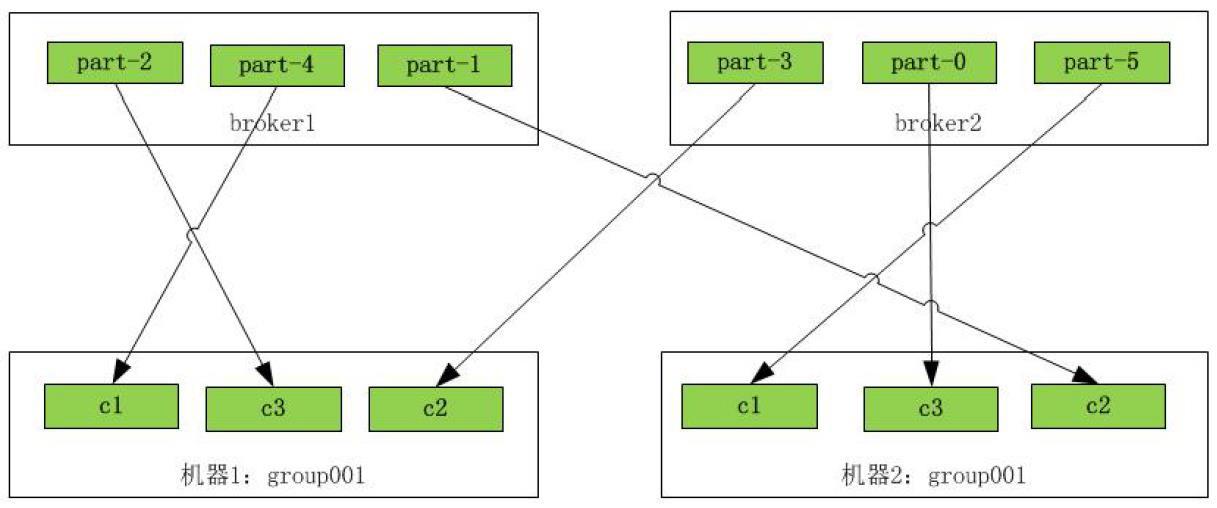
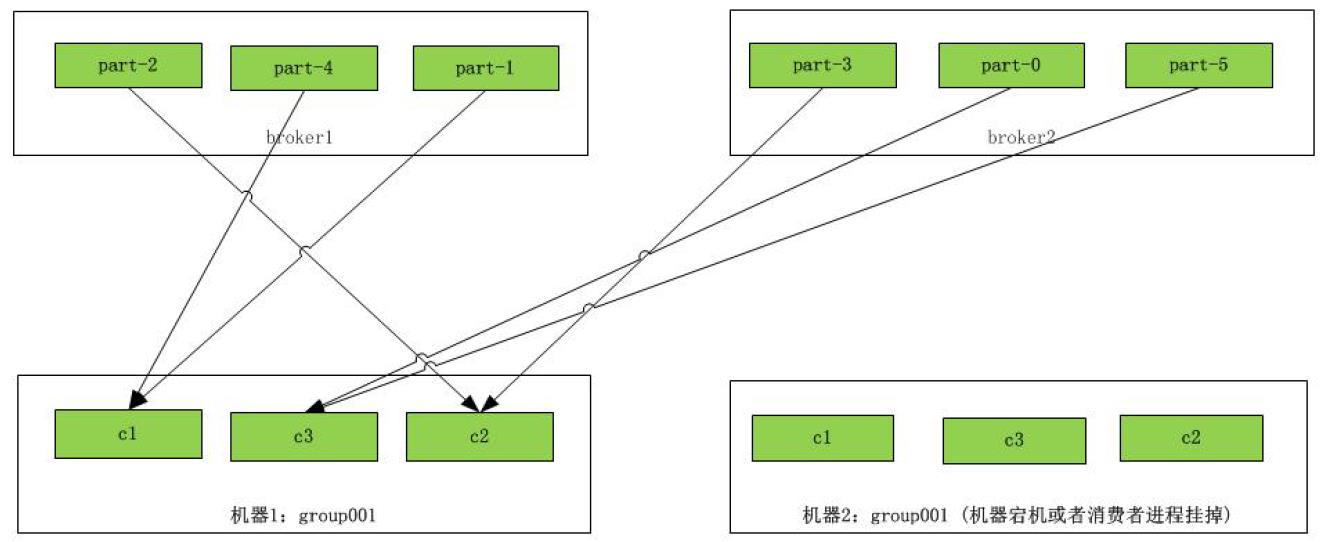


图 2-8 Kafka 高级消费者自动分配分区

如图 2-8 所示，同一套程序在两台机器上同时跑，均为 group001，那么高级消费者将会将两台机器中的 group001 视为同一个组，进而将两个 broker 的 6 个分区均匀分配到 group001

1. 6 个线程中。也就是说，高级消费者关注组名，即使是不同机器上的相同组名的消费者，也全部视为一个组的所有消费者。



1. 2-9 Kafka 高级消费者 rebalance

如图 2-9 所示，当有一台机器宕机或者消费者进程挂掉时，会进行 rebalance，即重新进行负载均衡，此时 group001 有 3 个消费者，因此高级消费者会将 6 个分区均匀分配给 3 个消费者，每个消费者 2 个分区。

通过以上的部署实现了 Kafka 的高可用。

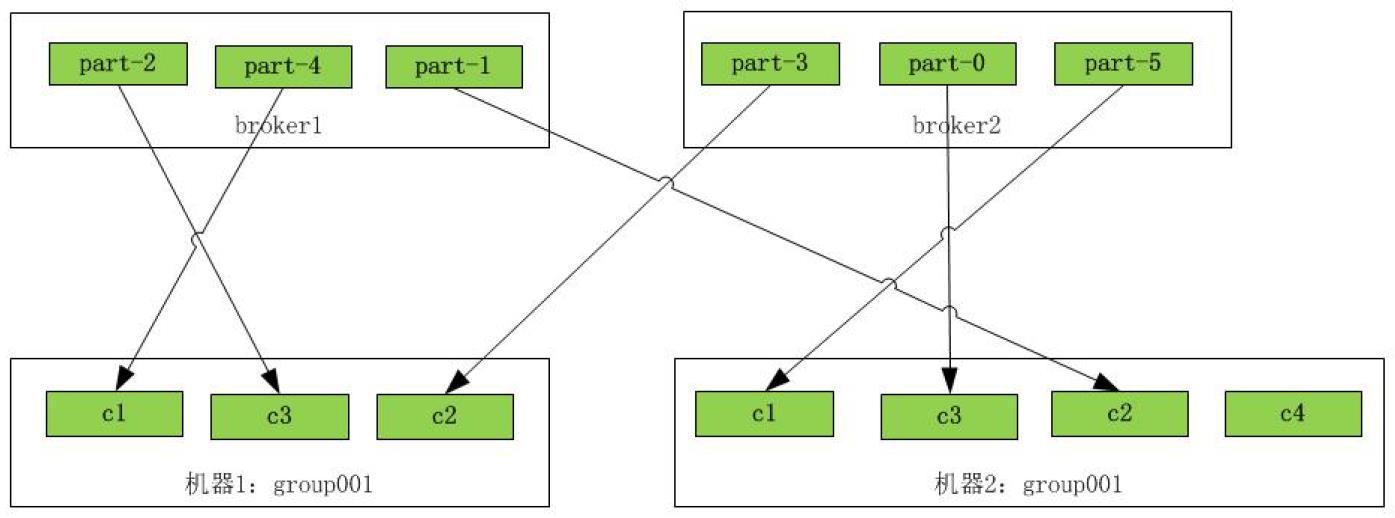


图 2-10 增加消费者线程不能提高并发度

当第一台机器有 3 个消费者，而第二台机器有 4 个消费者时，由于每个 partition 只能被一个分组内的一个消费者同时消费，因此，消费者 c4 线程空闲，拿不到数据，这就意味着，在分区不变的情况下，增加消费者线程不能提高并发度。

问题一：Kafka 高级消费者怎样才能达到最大吞吐量？

答：分区数量与线程数量一致。

问题二：消费者消费能力不足时，如果提高并发？

答：1. 增加分区个数；

1. 增加消费者线程数；
2. 自动提交 offset

在高阶消费者中，Offset 采用自动提交的方式。

自动提交时，假设 1s 提交一次 offset 的更新，设当前 offset=10，当消费者消费了 0.5s 的数据，offset 移动了 15，由于提交间隔为 1s，因此这一 offset 的更新并不会被提交，这时候我们写的消费者挂掉，重启后，消费者会去 ZooKeeper 上获取读取位置，获取到的 offset 仍为 10，它就会重复消费，这就是一个典型的重复消费问题。

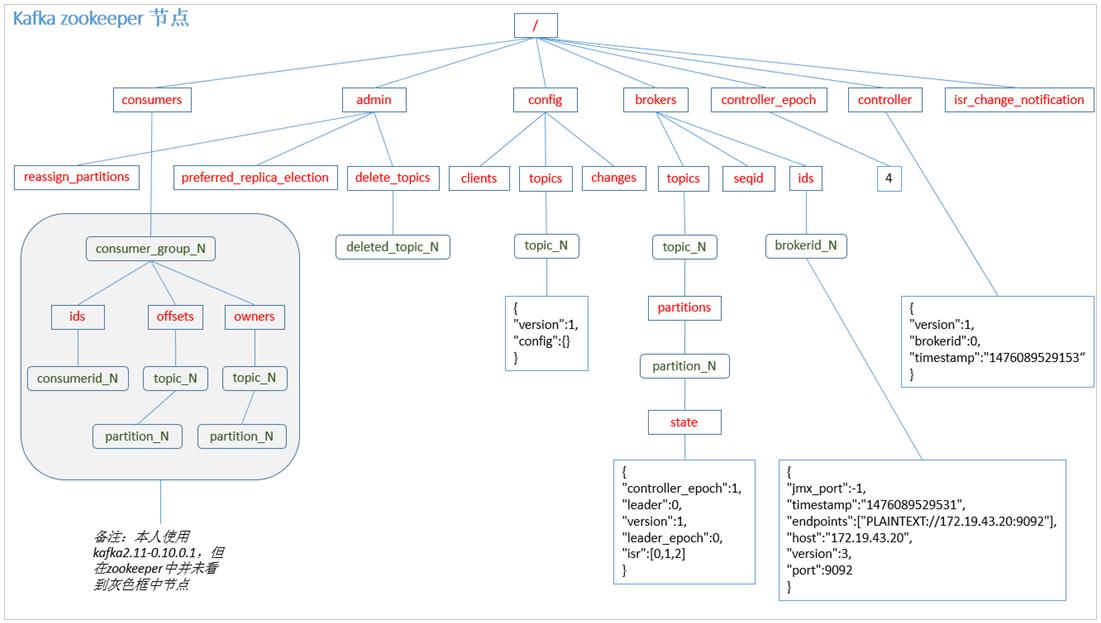
高阶消费者存在一个弊端，即消费者消费到哪里由高阶消费者 API 进行提交，提交到ZooKeeper，消费者线程不参与 offset 更新的过程，这就会造成数据丢失（消费者读取完成，高级消费者 API 的 offset 已经提交，但是还没有处理完成 Spark Streaming 挂掉，此时 offset 已经更新，无法再消费之前丢失的数据），还有可能造成数据重复读取（消费者读取完成，高级消费者 API 的 offset 还没有提交，读取数据已经处理完成后 Spark Streaming 挂掉，此时 offset 还没有更新，重启后会再次消费之前处理完成的数据）。

### 2.2.5 Kafka 低级消费者

对于低阶消费者就不再有分区到消费者之间的 API 中间层了，由消费者直接找到分区进行消费，即消费者通过 ZooKeeper 找到指定分区的 Leader 在哪个 broker 上。

首先，在 ZooKeeper 中能够找到 Kafka 所有 topic 的分区列表，并且可以找到指定分区

1. Leader 在哪个 broker 上。



1. 2-11 Kafka 在 ZooKeeper 上的布局

消费者消费一条消息后，可以选择提交或者不提交，offset 可以缓存在 Redis 中，也可以自己存到 ZooKeeper 上。

当正在读取的分区挂掉了，此时读取会出现异常，由于 Kafka 存在副本机制，需要从 ZooKeeper 重新获取元数据，更新列表，从而继续消费分区数据。

**2.2.6 KafkaCluster**

在使用 Kafka 低阶消费者时，可以通过 KafkaCluster 类实现 offset 向 ZooKeeper 的提交和获取。

Kafka 协议非常简单，只有六个核心客户端请求 API：

1. **元数据（Metadata）** -描述当前可用的代理，主机和端口信息，并提供有关哪个代理主机分区的信息。
2. **发送（Send）** -发送消息给经纪人
3. **获取（Fetch）** -从代理获取消息，获取数据，获取集群元数据，获取关于主题的偏移信息的消息。
4. **偏移量（Offsets）** -获取有关给定主题分区的可用偏移量的信息。
5. **偏移提交（Offset Commit）**-为消费者组提供一组偏移量
6. **偏移获取（Offset Fetch）** -为消费者组取得一组偏移量

**2.2.7 Kafka 高可靠性存储**

Kafka 的高可靠性的保障来源于其健壮的副本（replication）策略。通过调节其副本相关

参数，可以使得 Kafka 在性能和可靠性之间运转的游刃有余。Kafka 从 0.8.x 版本开始提供

partition 级别的复制，replication 的数量可以在$KAFKA\_HOME/config/server.properties 中配置（default.replication.refactor）。

1. Kafka 文件存储机制

Kafka 中消息是以 topic 进行分类的，生产者通过 topic 向 Kafka broker 发送消息，消费者通过 topic 读取数据。

为了便于说明问题，假设这里只有一个 Kafka 集群，且这个集群只有一个 Kafka broker，即只有一台物理机。在这个 Kafka broker 中配置（$KAFKA\_HOME/config/ server.properties 中）log.dirs=/tmp/kafka-logs，以此来设置 Kafka 消息文件存储目录，与此同时创建一个 topic： topic\_zzh\_test ， partition 的数量为 4 （ $KAFKA\_HOME/bin/kafka-topics.sh – create – zookeeper localhost:2181 –partitions 4 –topic topic\_zzh\_test –replication-factor 4）。那么我们此时可以在/tmp/kafka-logs 目录中可以看到生成了 4 个目录：

drwxr-xr-x 2 root root 4096 Apr 10 16:10 topic\_zzh\_test-0 drwxr-xr-x 2 root root 4096 Apr 10 16:10 topic\_zzh\_test-1 drwxr-xr-x 2 root root 4096 Apr 10 16:10 topic\_zzh\_test-2 drwxr-xr-x 2 root root 4096 Apr 10 16:10 topic\_zzh\_test-3

1. Kafka 文件存储中，同一个 topic 下有多个不同的 partition，每个 partiton 为一个目录， partition 的名称规则为：topic 名称+有序序号，第一个序号从 0 开始计，最大的序号为 partition 数量减 1，partition 是实际物理上的概念，而 topic 是逻辑上的概念。

partition 还可以细分为 segment，这个 segment 又是什么？

如果就以 partition 为最小存储单位，我们可以想象当 Kafka producer 不断发送消息，必然会引起 partition 文件的无限扩张，这样对于消息文件的维护以及已经被消费的消息的清理带来严重的影响，所以这里以 segment 为单位又将 partition 细分。每个 partition(目录)相当于一个巨型文件被平均分配到多个大小相等的 segment(段)数据文件中（每个 segment 文件中

消息数量不一定相等），这种特性也方便 old segment 的删除，即方便已被消费的消息的清理，提高磁盘的利用率。每个 partition 只需要支持顺序读写就行，segment 的文件生命周期由服务端配置参数（log.segment.bytes，log.roll.{ms,hours}等若干参数）决定。

segment 文件由两部分组成，分别为“.index”文件和“.log”文件，分别表示为 segment

索引文件和数据文件（引入索引文件的目的就是便于利用二分查找快速定位 message 位置）。这两个文件的命令规则为：partition 全局的第一个 segment 从 0 开始，后续每个 segment 文件名为上一个 segment 文件最后一条消息的 offset 值，数值大小为 64 位，20 位数字字符长度，没有数字用 0 填充，如下：

000000000000000000000.index

00000000000000000000.log

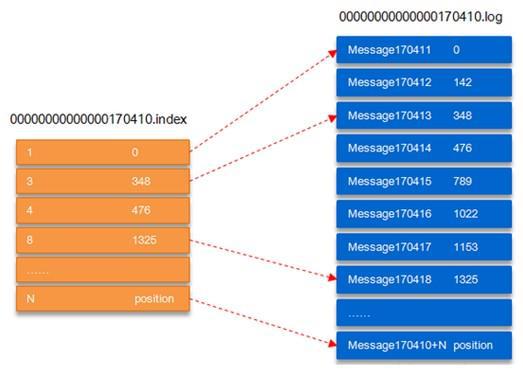
00000000000000170410.index

00000000000000170410.log

00000000000000239430.index

00000000000000239430.log

以上面的 segment 文件为例，展示出 segment：00000000000000170410 的“.index”文件和“.log”文件的对应的关系，如下图：



1. 2-12 segment 存储机制

如图 2-12 所示，“.index”索引文件存储大量的元数据，“.log”数据文件存储大量的消息，索引文件中的元数据指向对应数据文件中 message 的物理偏移地址。其中以“.index”索引文件中的元数据[3, 348]为例，在“.log”数据文件表示第 3 个消息，即在全局 partition 中表示 170410+3=170413 个消息，该消息的物理偏移地址为 348。

那么如何从 partition 中通过 offset 查找 message 呢？

以上图为例，读取 offset=170418 的消息，首先查找 segment 文件，其中 00000000000000000000.index 为最开始的文件，第二个文件为 00000000000000170410.index （起始偏移为 170410+1=170411），而第三个文件为 00000000000000239430.index（起始偏

移为 239430+1=239431），所以这个 offset=170418 就落到了第二个文件之中。其他后续文件可以依次类推，以其偏移量命名并排列这些文件，然后根据二分查找法就可以快速定位到

具体文件位置。其次 根据 00000000000000170410.index 文件中的 [8,1325] 定位到 00000000000000170410.log 文件中的 1325 的位置进行读取。

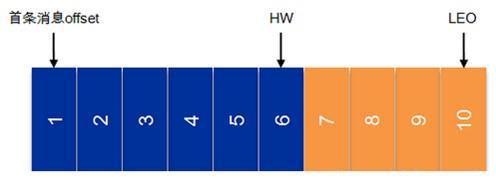
要是读取 offset=170418 的消息，从 00000000000000170410.log 文件中的 1325 的位置进行读取，那么怎么知道何时读完本条消息，否则就读到下一条消息的内容了？

这个就需要联系到消息的物理结构了，消息都具有固定的物理结构，包括：offset（8 Bytes）、消息体的大小（4 Bytes）、crc32（4 Bytes）、magic（1 Byte）、attributes（1 Byte）、 key length（4 Bytes）、key（K Bytes）、payload(N Bytes)等等字段，可以确定一条消息的大

小，即读取到哪里截止。

1. 复制原理和同步方式

Kafka 中 topic 的每个 partition 有一个预写式的日志文件，虽然 partition 可以继续细分为若干个 segment 文件，但是对于上层应用来说可以将 partition 看成最小的存储单元（一个由多个 segment 文件拼接的“巨型”文件），每个 partition 都由一些列有序的、不可变的消息组成，这些消息被连续的追加到 partition 中。



* 1. 2-13 HW 与 LEO

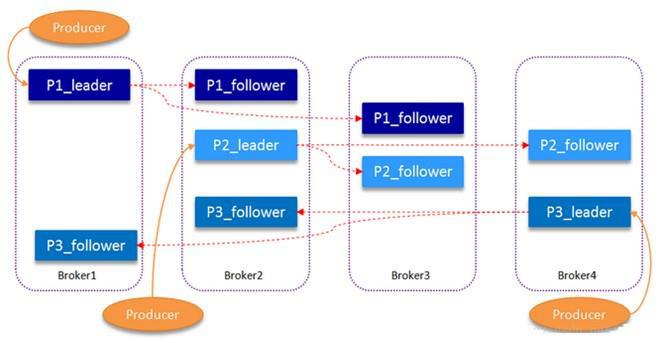
1. 2-13 中有两个新名词：HW 和 LEO。这里先介绍下 LEO，LogEndOffset 的缩写，表示每个 partition 的 log 最后一条 Message 的位置。HW 是 HighWatermark 的缩写，是指 consumer

能够看到的此 partition 的位置，这个涉及到多副本的概念（参见第 3.4.3 节）。

言归正传，为了提高消息的可靠性，Kafka 每个 topic 的 partition 有 N 个副本（replicas），其中 N(大于等于 1)是 topic 的复制因子（replica fator）的个数。Kafka 通过多副本机制实现故障自动转移，当 Kafka 集群中一个 broker 失效情况下仍然保证服务可用。在 Kafka 中发生

复制时确保 partition 的日志能有序地写到其他节点上，N 个 replicas 中，其中一个 replica 为leader，其他都为 follower，leader 处理 partition 的所有读写请求，与此同时，follower 会被动定期地去复制 leader 上的数据。

如下图所示，Kafka 集群中有 4 个 broker, 某 topic 有 3 个 partition,且复制因子即副本个数也为 3：



1. 2-14 Kafka 副本机制

Kafka 提供了数据复制算法保证，如果 leader 发生故障或挂掉，一个新 leader 被选举并

被接受客户端的消息成功写入。Kafka 确保从同步副本列表中选举一个副本为 leader，或者

1. follower 追赶 leader 数据。leader 负责维护和跟踪 ISR（In-Sync Replicas 的缩写，表示副本同步队列，具体可参考下节）中所有 follower 滞后的状态。当 producer 发送一条消息到

broker 后，leader 写入消息并复制到所有 follower。消息提交之后才被成功复制到所有的同步副本。消息复制延迟受最慢的 follower 限制，重要的是快速检测慢副本，如果 follower“落后”太多或者失效，leader 将会把它从 ISR 中删除。

ISR

ISR（In-Sync Replicas），副本同步队列。ISR 中包括 leader 和 follower。副本数对 Kafka 的吞吐率是有一定的影响，但极大的增强了可用性。默认情况下 Kafka 的 replica 数量为 1，即每个 partition 都有一个唯一的 leader，为了确保消息的可靠性，通常应用中将其值(由 broker 的参数 offsets.topic.replication.factor 指定)大小设置为大于 1，比如 3。 所有的副本（replicas）统称为 Assigned Replicas，即 AR。ISR 是 AR 中的一个子集，由 leader 维护 ISR 列表，follower

1. leader 同步数据有一些延迟（包括延迟时间 replica.lag.time.max.ms 和延迟条数 replica.lag.max.messages 两个维度, 当前的 0.10.x 及以上版本中只支持 replica.lag.time.max.ms 这个维度），任意一个超过阈值都会把 follower 剔除出 ISR, 存入 OSR（Outof-Sync Replicas）

列表，新加入的 follower 也会先存放在 OSR 中。

AR=ISR+OSR

Kafka 0.10.x 版 本 后 移 除 了 replica.lag.max.messages 参 数 ， 只 保 留 了 replica.lag.time.max.ms 作为 ISR 中副本管理的参数。为什么这样做呢？

replica.lag.max.messages 表示当前某个副本落后 leaeder 的消息数量超过了这个参数的值，那么 leader 就会把 follower 从 ISR 中删除。假设设置 replica.lag.max.messages=4，那么如果 producer 一次传送至 broker 的消息数量都小于 4 条时，因为在 leader 接受到 producer 发送的消息之后而 follower 副本开始拉取这些消息之前，follower 落后 leader 的消息数不会

超过 4 条消息，故此没有 follower 移出 ISR，所以这时候 replica.lag.max.message 的设置似乎是合理的。但是 producer 发起瞬时高峰流量，producer 一次发送的消息超过 4 条时，也就是超过 replica.lag.max.messages，此时 follower 都会被认为是与 leader 副本不同步了，从而被

踢出了 ISR。但实际上这些 follower 都是存活状态的且没有性能问题，那么在之后追上 leader,

并被重新加入了 ISR，于是就会出现它们不断地剔出 ISR 然后重新回归 ISR，这无疑增加了无谓的性能损耗。而且这个参数是 broker 全局的。设置太大了，影响真正“落后”follower的移除； 设置的太小了，导致 follower 的 频繁 进出 。 无法 给定一个合 适的 replica.lag.max.messages 的值，故此，新版本的 Kafka 移除了这个参数。

HW，HighWatermark 的缩写，俗称高水位，取一个 partition 对应的 ISR 中最小的 LEO 作为 HW，consumer 最多只能消费到 HW 所在的位置。每个 replica 都有 HW，leader 和 follower 各自负责更新自己的 HW 的状态。对于 leader 新写入的消息，consumer 不能立刻消费，leader会等待该消息被所有 ISR 中的 replicas 同步后更新 HW，此时消息才能被 consumer 消费，这样就保证了如果 leader 所在的 broker 失效，该消息仍然可以从新选举的 leader 中获取。对于来自内部 broker 的读取请求，没有 HW 的限制。

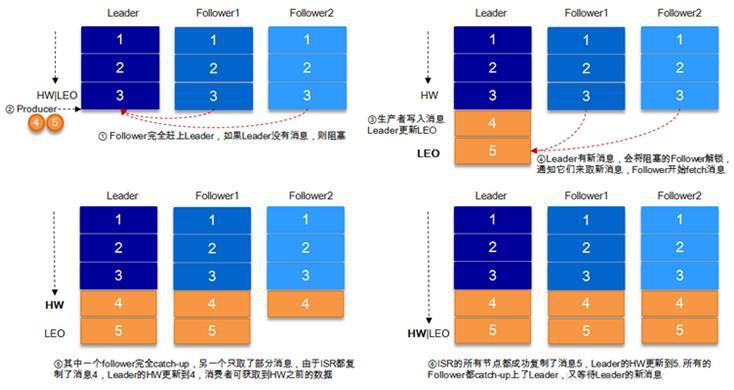
（一个 partition 的副本分为 Leader 和 Follower，Leader 和 Follower 都维护了 HW 和 LEO，

1. partition 的读写都在 Leader 完成，Leader 的 HW 是所有 ISR 列表里副本中最小的那个的LEO，当 Leader 新写入消息后，Leader 的 LEO 更新到加入新消息后的位置，Leader 的 HW

仍在原位置，当所有的 Follower 都同步完成后，HW 更新到最新位置，此时最新写入的消息才能被 Consumer 消费）

1. 2-15 详细的说明了当 producer 生产消息至 broker 后，ISR 以及 HW 和 LEO 的流转过

程：



1. 2-15 ISR、HW、LEO 流转过程

由此可见，Kafka 的复制机制既不是完全的同步复制，也不是单纯的异步复制。事实上，同步复制要求所有能工作的 follower 都复制完，这条消息才会被 commit，这种复制方式极大的影响了吞吐率。而异步复制方式下，follower 异步的从 leader 复制数据，数据只要被 leader 写入 log 就被认为已经 commit，这种情况下如果 follower 都还没有复制完，落后于 leader 时，突然 leader 宕机，则会丢失数据。而 Kafka 的这种使用 ISR 的方式则很好的均衡了确保数据不丢失以及吞吐率。

Kafka 的 ISR 的管理最终都会反馈到 Zookeeper 节点上 。具体位置为：/brokers/topics/[topic]/partitions/[partition]/state。目前有两个地方会对这个 Zookeeper 的节点进行维护：

Controller：Kafka 集群中的其中一个 Broker 会被选举为 Controller，主要负责 Partition 管理和副本状态管理，也会执行类似于重分配 partition 之类的管理任务。在符合某些特定条件下，Controller 下的 LeaderSelector 会选举新的 leader，将 ISR 和新的 leader\_epoch 及 controller\_epoch 写入 Zookeeper 的相关节点中。同时发起 LeaderAndIsrRequest 通知所有的replicas。

Leader：Leader 有单独的线程定期检测 ISR 中 Follower 是否脱离 ISR，如果发现 ISR 变化，则会将新的 ISR 的信息返回到 Zookeeper 的相关节点中。

数据可靠性和持久性保证

1. producer 向 leader 发送数据时，可以通过 request.required.acks 参数来设置数据可靠性的级别：

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

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

-1：producer 等待 broker 的 ack，partition 的 leader 和 follower 全部落盘成功后才返回 ack，数据一般不会丢失，延迟时间长但是可靠性最高。但是这样也不能保证数据不丢失，比如当

ISR 中只有 leader 时（前面 ISR 那一节讲到，ISR 中的成员由于某些情况会增加也会减少，最少就只剩一个 leader），这样就变成了 acks=1 的情况；

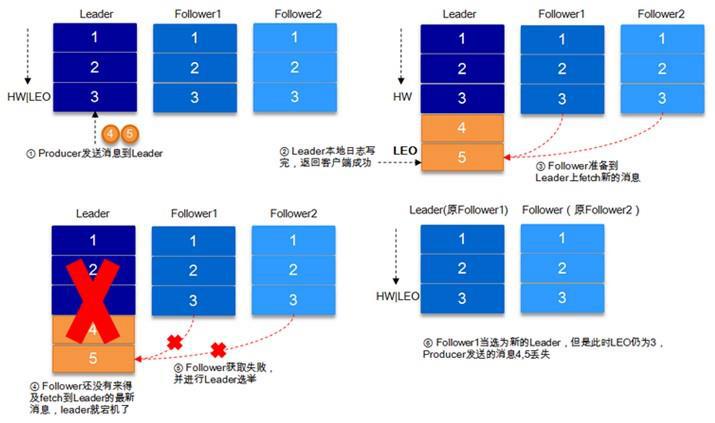
如 果 要 提 高 数 据 的 可 靠 性 ， 在 设 置 request.required.acks=-1 的 同 时 ， 也 要 min.insync.replicas 这个参数(可以在 broker 或者 topic 层面进行设置)的配合，这样才能发挥最大的功效。min.insync.replicas 这个参数设定 ISR 中的最小副本数是多少，默认值为 1，当且仅当 request.required.acks 参数设置为-1 时，此参数才生效。如果 ISR 中的副本数少于

min.insync.replicas 配置的数量时，客户端会返回异常：org.apache.kafka. common.errors. NotEnoughReplicasExceptoin: Messages are rejected since there are fewer in-sync replicas than required。

接下来对 ack=1 和-1 的两种情况进行详细分析：

1. Request.required.acks = 1

producer 发送数据到 Leader，Leader 写本地日志成功，返回客户端成功；此时 ISR 中的副本还没有来得及拉取该消息，Leader 就宕机了，那么此次发送的消息就会丢失 producer 发送数据到 Leader，Leader 写本地日志成功，返回客户端成功；此时 ISR 中的副本还没有来得及拉取该消息，Leader 就宕机了，那么此次发送的消息就会丢失。



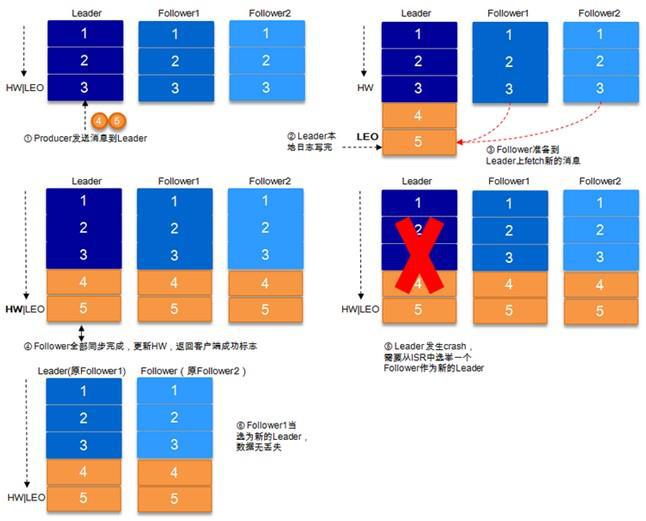
* 1. 2-16 Request.required.acks = 1

1. Request.required.acks = -1

同步（Kafka 默认为同步，即 producer.type=sync）的发送模式，replication.factor>=2 且 min.insync.replicas>=2 的情况下，不会丢失数据。

有两种典型情况。 acks=-1 的情况下（如无特殊说明，以下 acks 都表示为参数request.required.acks），数据发送到 leader，ISR 的 Follower 全部完 1 成数据同步后，Leader

此时挂掉，那么会选举出新的 Leader，数据不会丢失。

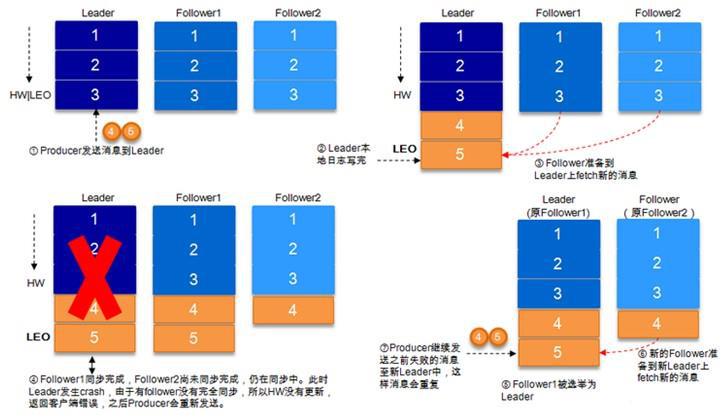


1. 2-17 Request.required.acks = -1

acks=-1 的情况下，数据发送到 leader 后 ，部分 ISR 的副本同步，leader 此时挂掉。比

1. follower1 和 follower2 都有可能变成新的 leader, producer 端会得到返回异常，producer 端

会重新发送数据，数据可能会重复。



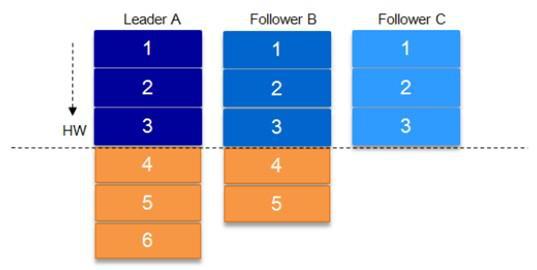
1. 2-18 Leader 宕机的情况

当然图 2-17 中所示，如果在 leader crash 的时候，follower2 还没有同步到任何数据，而且 follower2 被选举为新的 leader 的话，这样消息就不会重复。

考虑图 2-18（即 acks=-1,部分 ISR 副本同步）中的另一种情况，如果在 Leader 挂掉的时候，follower1 同步了消息 4,5，follower2 同步了消息 4，与此同时 follower2 被选举为 leader，那么此时 follower1 中的多出的消息 5 该做如何处理呢？

这里就需要 HW 的协同配合了。如前所述，一个 partition 中的 ISR 列表中，leader 的 HW 是所有 ISR 列表里副本中最小的那个的 LEO。类似于木桶原理，水位取决于最低那块

短板。



1. 2-19 HW 的作用

如图 2-19，某个 topic 的某 partition 有三个副本，分别为 A、B、C。A 作为 leader 肯定

LEO 最高，B 紧随其后，C 机器由于配置比较低，网络比较差，故而同步最慢。这个时候 A 机器宕机，这时候如果 B 成为 leader，假如没有 HW，在 A 重新恢复之后会做同步(makeFollower)操作，在宕机时 log 文件之后直接做追加操作，而假如 B 的 LEO 已经达到了

的 LEO，会产生数据不一致的情况，所以使用 HW 来避免这种情况。

A 在做同步操作的时候，先将 log 文件截断到之前自己的 HW 的位置，即 3，之后再从

中拉取消息进行同步。

如果失败的 follower 恢复过来，它首先将自己的 log 文件截断到上次 checkpointed 时刻

HW 的位置，之后再从 leader 中同步消息。leader 挂掉会重新选举，新的 leader 会发送“指令”让其余的 follower 截断至自身的 HW 的位置然后再拉取新的消息。

注意：当 ISR 中的个副本的 LEO 不一致时，如果此时 leader 挂掉，选举新的 leader 时并不是按照 LEO 的高低进行选举，而是按照 ISR 中的顺序选举，新的 leader 会发送“指令”让其余的 follower 截断至自身的 HW 的位置然后再拉取新的消息。

## 2.3 Hive/ Spark SQL

**2.3.1 Hive 用户自定义函数**

Hive 根据用户自定义函数类别分为以下三种：

表 1-9 Hive自定义函数类型

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **自定义函数类型** | |  | **解析** | |
|  |  |  |  |  |  |
|  | **UDF（User-Defined-Function）** |  |  | 一进一出 |  |
|  | **UDAF（User-Defined Aggregation Function）** | |  | 聚集函数，多进一出 | |
|  |  | |  |  | |
|  | **UDTF（User-Defined Table-Generating Functions）** |  |  | 一进多出 |  |
|  |  |  |  |  |  |

* 1. 编程步骤：

1. 继承 org.apache.hadoop.hive.ql.UDF
2. 需要实现 evaluate 函数；evaluate 函数支持重载；
3. 在 hive 的命令行窗口创建函数

a）添加 jar

add jar linux\_jar\_path

b）创建 function

create [temporary] function [dbname.]function\_name AS class\_name;

1. 在 hive 的命令行窗口删除函数

drop [temporary] function [if exists] [dbname.]function\_name;

2. UDF 实例

1）创建一个 java 工程，并创建一个 lib 文件夹

2）将 hive 的 jar 包解压后，将 apache-hive-1.2.1-bin\lib 文件下的 jar 包都拷贝到 java 工

程中。

3）创建一个类

public class Lower extends UDF {

public String evaluate (final String s) {

if (s == null) {

return null;

}

return s.toString().toLowerCase();

}

}

4）打成 jar 包上传到服务器/opt/module/jars/udf.jar

5）将 jar 包添加到 hive 的 classpath

hive (default)> add jar /opt/modules/datas/udf.jar;

6）创建临时函数与开发好的 java class 关联

hive (default)> create temporary function my\_lower as "com.atguigu.hive.Lower";

7）即可在 hql 中使用自定义的函数 strip

hive (default)> select ename, my\_lower(ename) lowername from emp;

## 2.4 Java & Maven

### 2.4.1 Maven

Apache Maven是一个软件项目管理和综合工具。基于项目对象模型（POM）的概念，Maven可以从一个中心资料片管理项目构建，报告和文件。

管理依赖包。

Maven的标准工程结构如下：

|-- pom.xml(maven的核心配置文件)

|-- src

|-- main

|   `-- java(java源代码目录)

|   `-- resources(资源文件目录)

|-- test

        `-- java(单元测试代码目录)

|-- target(输出目录，所有的输出物都存放在这个目录下)

    |--classes(编译后的class文件存放处)

<https://www.cnblogs.com/jingmoxukong/p/5591368.html>

一个pom.xml 的例子

<projectxmlns=*"http://maven.apache.org/POM/4.0.0"*xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

  xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.zp.maven</groupId>

  <artifactId>MavenDemo</artifactId>

  <version>0.0.1-SNAPSHOT</version>

  <packaging>jar</packaging>

  <name>MavenDemo</name>

  <url>http://maven.apache.org</url>

  <properties>

    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

    <junit.version>3.8.1</junit.version>

  </properties>

  <dependencies>

    <dependency>

      <groupId>junit</groupId>

      <artifactId>junit</artifactId>

      <version>${junit.version}</version>

      <scope>test</scope>

    </dependency>

    <dependency>

      <groupId>log4j</groupId>

      <artifactId>log4j</artifactId>

      <version>1.2.12</version>

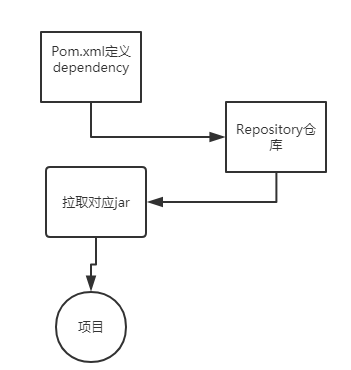
      <scope>compile</scope>

    </dependency>

  </dependencies>

</project>

通过groupId ,artifactId,version来拉取jar包,而repository是jar包服务器。而scope是他的生命周期



Build 中是用来定义项目如何构建的，如：

* 依赖jar包（放外面还是里面）
* 编译的java版本

<build>  
 <plugins>  
 <plugin>  
 <artifactId>maven-assembly-plugin</artifactId>  
 <configuration>  
 <descriptorRefs>  
 <descriptorRef>jar-with-dependencies</descriptorRef>  
 </descriptorRefs>  
 </configuration>  
 <!-- <executions> <execution> <id>make-assembly</id> <phase>package</phase>   
 <goals> <goal>single</goal> </goals> </execution> </executions> -->  
 </plugin>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <version>3.8.0</version>  
 <configuration>  
 <source>1.8</source>  
 <target>1.8</target>  
 </configuration>  
 </plugin>  
 </plugins>

### 2.4.2 Java 编码

OOP思想：

## 2.5 Spark Core

### 2.5.1 Spark 序列化

在 Spark 的架构中，在网络中传递的或者缓存在内存、硬盘中的对象需要进行序列化操作，序列化的作用主要是利用时间换空间：

* 分发给 Executor 上的 Task
* 需要缓存的 RDD（前提是使用序列化方式缓存）
* 广播变量
* Shuffle 过程中的数据缓存
* 使用 receiver 方式接收的流数据缓存
* 算子函数中使用的外部变量

上面的六种数据，通过 Java 序列化（默认的序列化方式）形成一个二进制字节数组，大大减少了数据在内存、硬盘中占用的空间，减少了网络数据传输的开销，并且可以精确的推测内存使用情况，降低 GC 频率。

序列化有一定的好处，但是缺点也比较明显：

把数据序列化为字节数组、把字节数组反序列化为对象的操作，是会消耗 CPU、延长作业时间的，从而降低了 Spark 的性能。

至少默认的 Java 序列化方式在这方面是不尽如人意的。Java 序列化很灵活但性能较差，同时序列化后占用的字节数也较多。

所以官方也推荐尽量使用 Kryo 的序列化库。官文介绍，Kryo 序列化机制比 Java 序列化机制性能提高 10 倍左右，Spark 之所以没有默认使用 Kryo 作为序列化类库，是因为它不支持所有对象的序列化，同时 Kryo 需要用户在使用前注册需要序列化的类型，不够方便。

Spark 2.0.0 版本开始，简单类型、简单类型数组、字符串类型的 Shuffling RDDs 已经默认使用 Kryo 序列化方式了。

//使用 Kryo 序列化库，如果要使用 Java 序列化库，需要把该行屏蔽掉

conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");

//在 Kryo 序列化库中注册自定义的类集合，如果要使用 Java 序列化库，需要把该行屏蔽掉

conf.set("spark.kryo.registrator", MyKryoRegistrator.class.getName());

Kryo 中的序列化类型注册如下所示：

public class MyKryoRegistrator implements KryoRegistrator {

public void registerClasses(Kryo kryo)

{

kryo.register(StartupReportLogs.class);

}

}

### 2.5.2 Spark 运行模式

Spark 工作在 Client 模式下时，由于 Driver 在作业提交机器上运行，Driver 进程是可以看到的，可以用 kill（不是 kill -9）杀死 Driver 进程，此时，如果设置了优雅停止，就会调用钩子函数进行优雅地停止。

Spark 工作在 Cluster 模式下时，Driver 运行在集群的那一台机器上我们是无法确定的（YARN 模式下由 ResourceManager 决定），因此无法用 kill 取杀死 Driver 进程。当工作

YARN 模式下时，可以使用 yarn application kill applicationID 杀死指定程序。用这种方式停止程序，ResourceManager 会给定一定的时间（如 1s）让 Driver 停止，但是如果在给定的时间内作业没有完成，那么 ResourceManager 会将其强制杀死，但是这不是我们希望看到的，我们希望 Driver 优雅地退出。

那么如何在 YARN 的 Cluster 模式下优雅退出？

我们采用的方法是启动一个监控进程，每 20s 查看一次 stopPath，如果 stopPath 存在，则停止 streaming，停止后 StreamingContextState 就变成了 STOPPED，监控进程检测到 StreamingContextState 变化为 STOPPED 后，就会停止。

停止代码：

/\* 如果stop path出现,就停止系统 \*/  
Configuration conf = javaStreamingContext.sparkContext().sc().hadoopConfiguration();  
FileSystem fs = DistributedFileSystem.*get*(conf);  
if(fs.exists(new org.apache.hadoop.fs.Path(serverProps.getProperty("streaming.stop.path")))){  
 *logger*.info("开始停止spark程序");  
 javaStreamingContext.stop(true);  
}

在提交作业时，不论 stopPath 是否存在，都要尝试删除此路径。由于此路径存在于 HDFS，因此为了验证此路径是否存在，需要连接 HDFS。

当采用 YARN Cluster 模式时，SparkSubmit 进程提交作业，然后就只负责接受任务是否正常运行的消息，不再负责任何其他功能，提交作业后，会在某一台机器上执行 Driver 进程，此时如果想停止 SparkStreamin，kill SparkSubmit 进程已经没有意义了，必须让 Driver 进程自己停止，因此，我们通过线程监控目录的方式停止 Driver。

## 2.6 Spark Streaming

### 2.6.1 Checkpoint

checkpoint是什么：

一个 Streaming Application 往往需要 7\*24 不间断的跑，所以需要有抵御意外的能力（比如机器或者系统挂掉，JVM crash 等）。为了让这成为可能，Spark Streaming 需要 Checkpoint 足够多信息至一个具有容错设计的存储系统才能让 Driver 从失败中恢复。

Spark Streaming Checkpoint 两种类型的数据。

Metadata（元数据） Checkpointing - 保存定义了 Streaming 计算逻辑至类似 HDFS 的支持容错的存储系统。用来恢复 Driver，元数据包括：

1. **配置** —— 用于创建该 streaming application 的所有配置；
2. **DStream 操作** ——DStream 一系列的操作；
3. **未完成的 batches** ——那些提交了 job 但尚未执行或未完成的 batches。

Data（数据） Checkpointing - 保存已生成的 RDD 至可靠的存储。这在某些 stateful 转换中是需要的，在这种转换中，生成 RDD 需要依赖前面的 batches，会导致依赖链随着时间而变长。为了避免这种没有尽头的变长，要定期将中间生成的 RDDs 保存到可靠存储来切断依赖链。

总之，Metadata Checkpointing 主要用来恢复 Driver； Data Checkpointing 对于 stateful 转换操作是必要的。

1. 什么时候该启用 Checkpoint 呢？

满足以下任一条件：

* 使用了有状态的 transformation 操作——比如 updateStateByKey（强制），或者reduceByKeyAndWindow 操作（非强制），被使用了，那么 Checkpoint 目录要求是必须提供的，也就是必须开启 Checkpoint 机制，从而进行周期性的 RDD Checkpoint
* 希望能从意外中恢复 Driver。

如果 streaming app 没有 stateful 操作，也允许 driver 挂掉后再次重启的进度丢失，就没有启用 Checkpoint 的必要了

2. Checkpoint 间隔设置

Checkpoint 的时间间隔设置方法如下：

dstream.checkpoint(checkpointInterval)

Checkpoint 时间间隔设置原则：一般设置为 batch 时间间隔的 5-10 倍。

Checkpoint 会增加存储开销、增加批次处理时间。当批次间隔较小（如 1 秒）时，checkpoint可能会减小 operation 吞吐量；反之，checkpoint 时间间隔较大会导致 lineage 和 task 数量增长。

SparkContext是：

### 2.6.2 Receiver 与 Direct

Receiver

Receiver 是使用 Kafka 的高层次 Consumer API 来实现的。Receiver 每隔一段 batch 时间

1. Kafka 获取那段时间最新的消息数据，Receiver 从 Kafka 获取的数据都是存储在 Spark Executor 的内存中的，然后 Spark Streaming 启动的 job 会去处理那些数据。

对于高阶消费者，谁来消费分区不是由 Spark Streaming 决定的，有一个高阶消费者 API，由高阶消费者决定分区向消费者的分配，即由高阶消费者 API 决定消费者消费哪个分区，

而消费者读取数据后什么时候提交 offset 也不是由它们自己决定的，高阶消费者 API 会根据参数配置隔几秒提交一次。

这会引起一个问题，当 Spark Streaming 中的 Receiver 读取 Kafka 分区数据时，假设读取了 100 条数据，高阶消费者 API 会执行 offset 的提交，例如每隔 3 秒，这 100 条数据就是RDD，假设此 RDD 还没有处理完，高阶消费者 API 执行了 offset 提交，但是 Spark Streaming 挂掉了，由于 RDD 在内存中，那么 RDD 的数据就丢失了，如果想重新拿数据，从哪里去

拿不是由 Spark Streaming 说了算的，是由高阶 API 决定的，由于 offset 已经提交，高阶 API 认为这个数据 Spark Streaming 已经拿过了，再拿要拿 100 条以后的数据，那么之前丢失的

1. 条数据就永远丢失了。

针对这一问题，Spark Streaming 设计了一个规则，即 Spark Streaming 预写日志规则（Write Ahead Log，WAL），每读取一批数据，会写一个 WAL 文件，在 WAL 文件中，读了多少条就写多少条，WAL 文件存储于 HDFS 上。假设 RDD 中有 100 条数据，那么 WAL 文件中也100 条数据，此时如果 Spark Streaming 挂掉，那么回去读取 HDFS 上的 WAL 文件，把WAL 文件中的 100 条数据取出再生成 RDD，然后再去消费。由于这一设计需要写 HDFS，会对整体性能造成影响。

假设有 6 个分区，高阶消费者的话会在 Spark 集群的 Worker 上启动 Receiver，有 6 个

分区则会用 6 个线程去读取分区数据，这是在一个 Worker 的一个 Receiver 中有 6 个线程同时读取 6 个分区的数据，随着数据量越来越大，数据读取会成为瓶颈，此时可以创建多个

Receiver 分散读取分区数据，然后每个 Receiver 创建一个 Dstream，再把这些流全部都合并起来，然后进行计算。读取时，一方面把 RDD 放在内存中，一方面写 HDFS 中的 WAL 文件。

根据上面的情景，又要创建多个 Receiver，又要进行合并，又要在内存中存储 RDD，又要写 HDFS 上的 WAL 文件，高级 API 的缺点还是比较多的。

高阶消费者是由高阶消费者 API 自己提交 offset 到 ZooKeeper 中。

Direct

低阶消费者需要自己维护 offset，Spark Streaming 从分区里读一部分数据，然后将 offset

保存到 CheckpointPath 目录中，比如 5s 生成一个 Spark Streaming job（每个 action 操作启动一次 job），每个 job 生成的时候，会写一次 CheckpointPath 下的文件，Checkpoint 中有 job 信息和 offset 信息（当然还有 RDD 依赖关系等其他信息），即保存了未完成的 job 和分区读取的 offset，一旦 Spark Streaming 挂掉后重启，可以通过从 CheckpointPath 中的文件中反序列化来读取 Checkpoint 的数据。

### 2.6.3 Spark Streaming 代码升级

如果正在运行的 Spark Streaming 应用程序需要使用新的应用程序代码进行升级，则有两种可能的机制：

升级的 Spark Streaming 应用程序启动并与现有应用程序并行运行。一旦新的应用程序（接收到的数据与旧的应用程序相同）已经被预热并准备好进入独立运行阶段，旧的应用程序就可以被取消。 请注意，这可以为支持将数据发送到两个目标（即早期和已升级的应用程序）的数据源完成。

现 有 应 用 程 序 正 常 关 闭 （ 请 参 阅 StreamingContext.stop （ ... ） 或JavaStreamingContext.stop（...）以获取正常关闭选项），以确保已收到的数据在关闭前

已完全处理。然后可以启动升级的应用程序，该应用程序将从旧版应用程序中断的同一点开始处理。 请注意，只有使用支持 Source 端缓冲的输入源（如 Kafka 和 Flume）才能完成此操作，因为数据需要在先前的应用程序关闭并且升级的应用程序尚未启动时进行缓冲。新的应用程序不能够在旧版应用程序的 Checkpoint 信息基础上启动，因为 Checkpoint 信息本质上包含序列化的 Scala / Java / Python 对象，试图用新的修改后的类来反序列化对象可能会导致错误，在这种情况下，可以使用不同的 Checkpoint 目录来启动升级的应用程序，也可以删除以前的 Checkpoint 目录。

代码修改后，如果仍然使用原来的 Checkpoint 目录，并且有序列化文件，那么新的应用程序启动后可能引起错误，比如旧版本序列化了一个对象，但是新版本对这个类进行了修改，那么反序列化时很有可能报错。其实如果没有删除原来的 Checkpoint 目录，那么原来的 offset 仍然存储于 Checkpoint

目录下，而序列化后的文件都有一个序列化版本号，当新的程序启动后会尝试反序列化 Checkpoint 目录下的序列化文件，但是会发现序列化版本号不一致，导致无法反序列化。

但是有个问题，如果修改了代码，并且不再使用原来的 CheckpointPath（删除或修改），那么新的应用程序如何获取原来的 offset 呢？？

可以同时使用 Checkpoint 机制和 ZooKeeper 机制：

当程序只是挂掉之后重启而没有修改代码的时候，通过 Checkpoint 机制反序列化信息；

当应用程序升级了代码的时候，首先需要 graceful stop 我们的 Spark Streaming，所谓 graceful stop 就是 Spark Streaming 不再产生新的作业，让所有未完成的作业执行完成，此时 Checkpoint 是最后一个批次生成的，ZooKeeper 也是最后一个批次更新的，此时

ZooKeeper 中的 offset 与 Checkpoint 中的 offset 是一致的。新版应用程序启动前会删除原始的 Checkpoint 目录，程序执行 getOrCreate()，此时没有 CheckpointPath 目录，那么就会执行函数，函数新建 JavaStreamingContext，新建 JavaStreamingContext 时会自己去读取 ZooKeeper 中的 offset，然后把 offset 信息写入 CheckpointPath，随后开始运行。

**2.7.4 Spark Streaming 实现 offset 在 Zookeeper 的读取（老）**

1. KafkaCluster 的创建

public static KafkaCluster getKafkaCluster(Map<String, String> kafkaParams) {

// 将 Java 的 HashMap 转化为 Scala 的 mutable.Map

scala.collection.mutable.Map<String, String> testMap = JavaConversions.mapAsScalaMap

(kafkaParams);

1. 将 Scala 的 mutable.Map 转化为 imutable.Map scala.collection.immutable.Map<String, String> scalaKafkaParam =

testMap.toMap(new Predef.$less$colon$less<Tuple2<String, String>, Tuple2 <String,

String>>() {

public Tuple2<String, String> apply(Tuple2<String, String> v1) { return v1;

}

});

1. 由于 KafkaCluster 的创建需要传入 Scala.HashMap 类型的参数，因此要进行上述的转换
2. 将 immutable.Map 类型的 Kafka 参数传入构造器，创建 KafkaCluster

return new KafkaCluster(scalaKafkaParam);

}

2. 从 Zookeeper 读取 offset

public static Map<TopicAndPartition, Long> getConsumerOffsets( KafkaCluster kafkaCluster,

String groupId,

Set<String> topicSet) {

// 将 Java 的 Set 结构转换为 Scala 的 mutable.Set 结构

scala.collection.mutable.Set<String> mutableTopics = JavaConversions.asScalaSet (topicSet);

1. 将 Scala 的 mutable.Set 结构转换为 immutable.Set 结构 scala.collection.immutable.Set<String> immutableTopics = mutableTopics.toSet();
2. 根据传入的分区，获取 TopicAndPartition 形式的返回数据 scala.collection.immutable.Set<TopicAndPartition> topicAndPartitionSet2 = (scala.collection.immutable.Set<TopicAndPartition>)kafkaCluster.getPartitions(immutableTopics ).right().get();

// 创建用于存储 offset 数据的 Hashmap

Map<TopicAndPartition, Long> consumerOffsetsLong = new HashMap();

1. kafkaCluster.getConsumerOffsets：通过 kafkaCluster 的 getConsumerOffsets 方法获取指定消费者组合，指定主题分区的 offset
   1. 如果返回 Left，代表获取失败，Zookeeper 中不存在对应的 offset，因此 HashMap 中对应的 offset 应

该设置为 0

if (kafkaCluster.getConsumerOffsets(groupId, topicAndPartitionSet2).isLeft()) {

* + 1. 将 Scala 的 Set 结构转换为 Java 的 Set 结构

Set<TopicAndPartition> topicAndPartitionSet1 = JavaConversions.setAsJavaSet(topic AndPartitionSet2);

* 1. 由于没有保存 offset（该 group 首次消费时）, 各个 partition offset 默认为 0 for (TopicAndPartition topicAndPartition : topicAndPartitionSet1) {

consumerOffsetsLong.put(topicAndPartition, 0L);

}

} else {

* + 1. offset 已存在, 获取 Zookeeper 上的 offset
    2. 获取到的结构为 Scala 的 Map 结构 scala.collection.immutable.Map<TopicAndPartition, Object> consumerOffsetsTemp = (scala.collection.immutable.Map<TopicAndPartition, Object>) kafkaCluster.

getConsumerOffsets(groupId, topicAndPartitionSet2).right().get();

// 将 Scala 的 Map 结构转换为 Java 的 Map 结构

Map<TopicAndPartition, Object> consumerOffsets = JavaConversions.mapAsJavaMap (consumerOffsetsTemp);

// 将 Scala 的 Set 结构转换为 Java 的 Set 结构

Set<TopicAndPartition> topicAndPartitionSet1 = JavaConversions.setAsJavaSet (topicAndPartitionSet2);

// 将 offset 加入到 consumerOffsetsLong 的对应项

for (TopicAndPartition topicAndPartition : topicAndPartitionSet1) { Long offset = (Long) consumerOffsets.get(topicAndPartition); consumerOffsetsLong.put(topicAndPartition, offset);

}

}

return consumerOffsetsLong;

}

3. Offset 写入 Zookeeper

public static void offsetToZk(final KafkaCluster kafkaCluster,

final AtomicReference<OffsetRange[]> offsetRanges, final String groupId) {

1. 遍历每一个偏移量信息

for (OffsetRange o : offsetRanges.get()) {

// 提取 offsetRange 中的 topic 和 partition 信息封装成 TopicAndPartition

TopicAndPartition topicAndPartition = new TopicAndPartition(o.topic(), o.partition());

1. 创建 Map 结构保持 TopicAndPartition 和对应的 offset 数据 Map<TopicAndPartition, Object> topicAndPartitionObjectMap = new HashMap();
2. 将当前 offsetRange 的 topicAndPartition 信息和 untilOffset 信息写入 Map topicAndPartitionObjectMap.put(topicAndPartition, o.untilOffset());
   1. 将 Java 的 Map 结构转换为 Scala 的 mutable.Map 结构

scala.collection.mutable.Map<TopicAndPartition, Object> testMap = JavaConversions. mapAsScalaMap(topicAndPartitionObjectMap);

1. 将 Scala 的 mutable.Map 转化为 imutable.Map scala.collection.immutable.Map<TopicAndPartition, Object> scalatopicAndPartition

ObjectMap =

testMap.toMap(new Predef.$less$colon$less<Tuple2<TopicAndPartition, Object>, Tuple2 <TopicAndPartition, Object>>() {

public Tuple2<TopicAndPartition, Object> apply(Tuple2<TopicAndPartition, Object> v1) {return v1;}

});

// 更新 offset 到 kafkaCluster

kafkaCluster.setConsumerOffsets(groupId, scalatopicAndPartitionObjectMap);

}

}

final class OffsetRange private(

val topic: String,

val partition: Int,

val fromOffset: Long,

val untilOffset: Long)

### 2.6.4 Spark Streaming 手动提交offset

stream.foreachRDD(**new** VoidFunction<JavaRDD<ConsumerRecord<String, String>>>() {

**@Override**

**public** void call(JavaRDD<ConsumerRecord<String, String>> rdd) {

OffsetRange[] offsetRanges = ((HasOffsetRanges) rdd.rdd()).offsetRanges();

*// 业务逻辑处理*

((CanCommitOffsets) stream.inputDStream()).commitAsync(offsetRanges);

}

});

在业务逻辑完成之后再提交到zk中，否则下次还是以zk中存在的consumer的offset为准。此代码较为简洁，建议使用

## 2.7 Crontab 任务调度

### 2.7.1 crontab 常用命令

1. 查看状态

service crond status

1. 停止状态：

service crond stop

1. 启动状态：

service crond start

4. 编辑 crontab 定时任务

crontab –e

5. 查询 crontab 任务

crontab –l

1. 删除当前用户所有的 crontab 任务

crontab –r

### 2.7.2 编写 crontab 调度

1. 进入编写 crontab 调度

crontab –e

* 1. 实现每分钟执行一次

1. \* \* \* \* source /etc/profile; /opt/module/shell/hdfstohive.sh

表 1-10 Hive自定义函数类型

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **项目** | |  | **含义** | |  | **范围** | |
|  |  |  |  |  |  |  |  |  |
|  | **第一个“\*”** |  |  | 一小时当中的第几分钟 |  |  | 0-59 |  |
|  | **第二个“\*”** | |  | 一天当中的第几小时 | | 0-23 | |  |
|  |  | |  |  | |  | |  |
|  | **第三个“\*”** |  |  | 一个月当中的第几天 |  |  | 1-31 |  |
|  | **第四个“\*”** | |  | 一年当中的第几月 | | 1-12 | |  |
|  |  | |  |  | |  |  | |
|  | **第五个“\*”** |  |  | 一周当中的星期几 |  |  | 0-7（0 和 7 都代表星期日） |  |
|  |  |  |  |  |  |  |  |  |

3. 查看 crontab

crontab –l

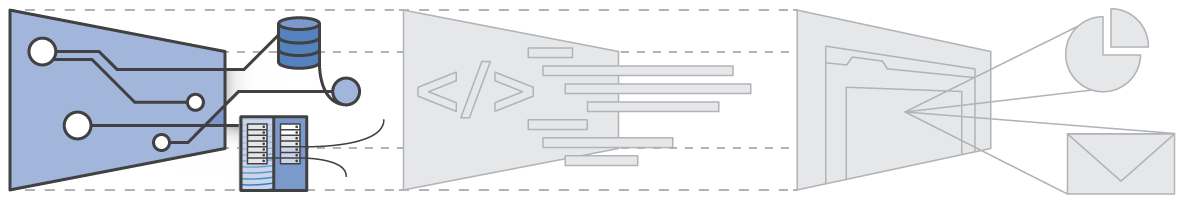
## 2.8 云计算

### 2.8.1 基本概念

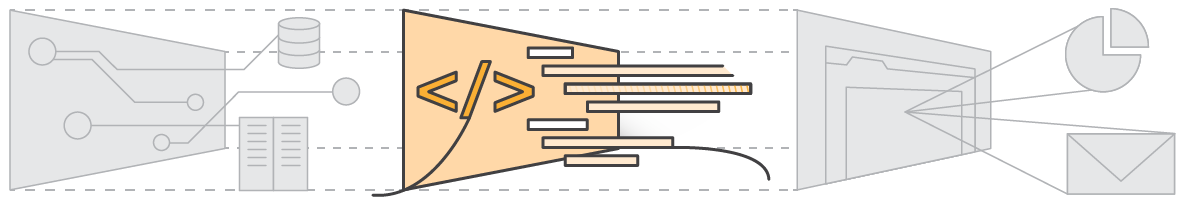
**云计算模型**

云计算的模型主要有三种。每种模型代表着云计算世界的一个独特部分。

基础设施即服务 (IaaS)

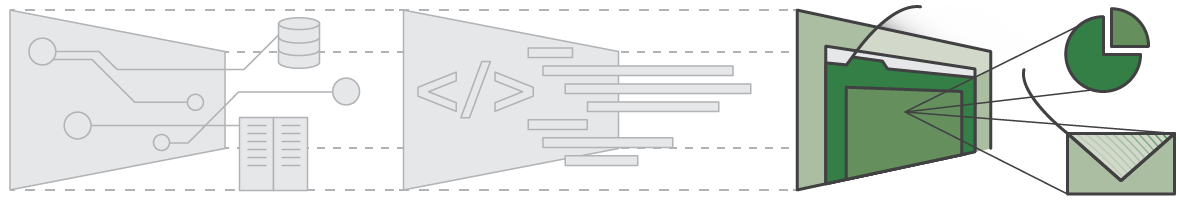


基础设施即服务有时缩写为 IaaS，包含云 IT 的基本构建块，通常提供对联网功能、计算机（虚拟或专用硬件）以及数据存储空间的访问。基础设施即服务提供最高等级的灵活性和对 IT 资源的管理控制，其机制与现今众多 IT 部门和开发人员所熟悉的现有 IT 资源最为接近。

平台即服务 (PaaS)  


平台即服务消除了组织对底层基础设施（一般是硬件和操作系统）的管理需要，让您可以将更多精力放在应用程序的部署和管理上面。这有助于提高效率，因为您不用操心资源购置、容量规划、软件维护、补丁安装或任何与应用程序运行有关的不能产生价值的繁重工作。

软件即服务 (SaaS)



软件即服务提供一种完善的产品，其运行和管理皆由服务提供商负责。通常人们所说的软件即服务指的是终端用户应用程序。使用 SaaS 产品时，服务的维护和底层基础设施的管理都不用您操心，您只需要考虑怎样使用 SaaS 软件就可以了。SaaS 的常见应用是基于 Web 的电子邮件，在这种应用场景中，您可以收发电子邮件而不用管理电子邮件产品的功能添加，也不需要维护电子邮件程序所运行的服务器和操作系统。

就直接拿来用的软件了。

比如

**Infrastructure as a Service (IaaS) by AWS:**

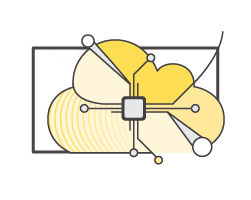
* **Amazon EC2**
* **Amazon Elastic Block Store**
* **AutoScaling**
* **Elastic Load Balancing**

**Platform as a Service (PaaS) by AWS:**

* **AWS Elastic Beanstalk**

云计算部署模型

[云](https://aws.amazon.com/cn/what-is-cloud-computing/)

[](https://aws.amazon.com/cn/what-is-cloud-computing/)

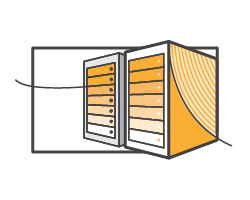
基于云的应用程序完全部署在云中且应用程序的所有组件都在云中运行。云中的应用程序分为两种，一种是在云中创建，另一种是从现有基础设施迁移到云中以利用云计算的优势。基于云的应用程序可以构建在基础设施组件上，也可以使用较高级的服务，这些服务提供了从核心基础设施的管理、架构和扩展要求中抽象提取的能力。

混合

[](https://aws.amazon.com/cn/hybrid/)

混合部署是一种在基于云的资源和非云现有资源之间连接基础设施和应用程序的方法。混合部署最常见的方法是在云和现有内部基础设施之间将组织的基础设施扩展到云中，同时将云资源与内部系统进行连接。

内部



使用虚拟化和资源管理工具在内部部署资源往往被称作“私有云”。内部部署无法提供云计算的诸多优势，但有时采用这种方案是为了能够提供专用资源。大多数情况下，这种部署模型与旧式 IT 基础设施无异，都通过应用程序管理和虚拟化技术尽可能提高资源利用率。

目前主流的云计算服务供应商有

* 亚马逊云
* 微软云
* 谷歌云
* 阿里云
* 腾讯云

AWS中云产品中的服务器为EC2，硬盘存储为EBS，分布式文件系统为S3。

### 2.8.2 重要产品

以下以AWS云为例子

**VPC：**

Amazon Virtual Private Cloud。

VPC指的是一种云(Cloud)，这与它的字面意思相符。对于基础架构服务(IaaS)，云就是指资源池。你或许听过公有云(Public Cloud)、私有云(Private Cloud)、混合云(Hybrid Cloud)。不过，VPC不属于这三种云中任一种。这是一种运行在公有云上，将一部分公有云资源为某个用户隔离出来，给这个用户私有使用的资源的集合。VPC是这么一种云，它由公有云管理，运行在公共资源上，但是保证每个用户之间的资源是隔离，用户在使用的时候不受其他用户的影响，感觉像是在使用自己的私有云一样。

**EC2**

Amazon Elastic Compute Cloud (Amazon EC2) 是一种 Web 服务，可以在云中提供安全并且可调整大小的计算容量。该服务旨在让开发人员能够更轻松地进行 Web 规模的云计算。

Amazon EC2 的 Web 服务接口非常简单，您可以最小的阻力轻松获取和配置容量。使用该服务，您可以完全控制您的计算资源，并可以在成熟的 Amazon 计算环境中运行。Amazon EC2 将获取并启动新服务器实例所需要的时间缩短至几分钟，这样一来，在您的计算要求发生变化时，您便可以快速扩展或缩减计算容量。Amazon EC2 按您实际使用的容量收费，改变了计算的成本结算方式。Amazon EC2 还为开发人员提供了创建故障恢复应用程序以及排除常见故障情况的工具。

**EC2 可以理解为云计算机**

EBS

Amazon Elastic Block Store (Amazon EBS) 可在 AWS 云中提供用于 Amazon EC2 实例的持久性块存储卷。每个 Amazon EBS 卷都会在其可用区内自动复制，以保护您免受组件故障的影响，同时提供高可用性和持久性。Amazon EBS 卷为您提供处理工作所需的稳定低延迟性能。通过 Amazon EBS，您可在几分钟内调整用量大小 - 所有这些您只需为配置的资源量支付低廉的价格。

Amazon EBS 设计用于优化性能、成本和容量即可受益的应用程序工作负载。典型使用案例包括：大数据分析引擎（如 Hadoop/HDFS 生态系统和 Amazon EMR 集群）、关系和 NoSQL 数据库（如 Microsoft SQL Server 和 MySQL 或 Cassandra 和 MongoDB）、流和日志处理应用程序（如 Kafka 和 Splunk），以及数据仓库应用程序（如 Vertica 和 Teradata）。

**EBS就可以理解为挂在在云计算机上的云硬盘**

**RDS**

Amazon Relational Database Service (Amazon RDS) 让您能够在云中轻松设置、运行和扩展关系数据库。它在自动执行耗时的管理任务（如硬件预置、数据库设置、修补和备份）的同时，可提供经济实用的可调容量。这使您能够腾出时间专注于应用程序，为它们提供所需的快速性能、高可用性、安全性和兼容性。

Amazon RDS 在多种类型的数据库实例（针对内存、性能或 I/O 进行了优化的实例）上均可用，并提供六种常用的数据库引擎供您选择，包括 Amazon Aurora、PostgreSQL、MySQL、MariaDB、Oracle 和 Microsoft SQL Server。您可以使用 AWS Database Migration Service 轻松将您现有的数据库迁移或复制到 Amazon RDS。

**亚马逊提供的关系型数据库**

**S3**

Amazon Simple Storage Service (Amazon S3) 是一种对象存储，它具有简单的 Web 服务接口，可用于从 Web 上的任何位置存储和检索任意数量的数据。它能够提供 99.999999999% 的持久性，并且可以在全球大规模传递数万亿对象。

客户将 S3 用于批量存储库、“数据湖”(用于分析)、备份和还原、灾难恢复和无服务器计算。许多原生云应用程序甚至使用 S3 作为主要存储。

借助 Amazon 的云数据迁移选项，客户可轻松地将大量数据移入或移出 S3。数据在存储到 Amazon S3 中之后，会自动采用成本更低、存储期限更长的云存储类（如 S3 Standard-Infrequent Access、S3 One Zone - Infrequent Access 和 Amazon Glacier）进行存档。

**S3是一个无服务器的数据服务。可以理解为云硬盘，同时它提供了网络接口**

**AutoScaling：自动扩展**

AWS Auto Scaling 可以监控您的应用程序并自动调整容量，以便以尽可能低的成本来保持稳定、可预测的性能。使用 AWS Auto Scaling，您可以在几分钟内为多项服务中的多个资源轻松设置应用程序扩展。该服务可以提供一个简单而功能强大的用户界面，让您可以为 Amazon EC2 实例和 Spot 队列、Amazon ECS 任务、Amazon DynamoDB 表和索引以及 Amazon Aurora 副本等资源制定扩展计划。AWS Auto Scaling 可以提供建议，让您能够优化性能、成本或实现二者的平衡，从而使扩展变得简单。如果您已经在使用 Amazon EC2 Auto Scaling 来动态扩展 Amazon EC2 实例，那么现在可以将其与 AWS Auto Scaling 结合使用，为其他 AWS 服务扩展其他资源。有了 AWS Auto Scaling，您的应用程序就始终能在合适的时间获得合适的资源。

您可以通过 AWS 管理控制台、命令行界面 (CLI) 或软件开发工具包轻松开始使用 AWS Auto Scaling。使用 AWS Auto Scaling 不会产生额外费用。您仅需支付运行应用程序所需的 AWS 资源费用和 Amazon CloudWatch 监控费用。

Autoscaling就是一种策略，在峰值的时候复制机器，在不用的时候自动释放机器。符合亚马逊pay as you go的策略。

**ELB:**

Elastic Load Balancing 可以在多个目标（如 Amazon EC2 实例、容器和 IP 地址）之间自动分配传入的应用程序流量。它可以在单个可用区内处理不断变化的应用程序流量负载，也可以跨多个可用区处理此类负载。Elastic Load Balancing 提供三种负载均衡器，它们均能实现高可用性、自动扩展和可靠的安全性，因此能让您的应用程序获得容错能力。

nginx负载均衡可以用ELB来实现。

**EMR**:

AWS的 大数据套件，Paas服务。国内用的是阿里云的E-MapReduce.

Amazon EMR 提供的托管 Hadoop 框架可以让您快速、轻松、经济高效地在多个动态可扩展的 Amazon EC2 实例中处理大量数据。您还可以运行其他常用的分布式框架（例如 Amazon EMR 中的 Apache Spark、HBase、Presto 和 Flink），以及与其他 AWS 数据存储服务（例如 Amazon S3 和 Amazon DynamoDB）中的数据进行交互。

**集成了大数据的套件。但大都昂贵，所以用EC2自己搭建。**

## 2.9 大数据仓库

**一、数据仓库是什么**

**可以理解为：**面向分析的存储系统。

也就是说数仓是存数据的，企业的各种数据往里面塞，主要目的是为了有效分析数据，后续会基于它产出供分析挖掘的数据，或者数据应用需要的数据，如企业的分析性报告和各类报表，为企业的决策提供支持。

上面的意思先理解着，接下来从头讲起，先看关系型数据库，它可以被划分为两大基本类型：操作型数据库和分析型数据库。

**1. 操作型数据库**

主要面向应用，用于业务支撑，支持对实际业务的处理，也可以叫业务型数据库。

可以理解为通常意义上的数据库(后端开发同学口中的经常提到的就是这种)。

**2. 分析型数据库**

主要面向数据分析，侧重决策支持，作为公司的单独数据存储，负责利用历史数据对公司各主题域进行统计分析。

由于分析型数据库中的操作都是查询，因此也就不需要严格满足关系型数据库一些设计规范，这样的情况下再将它归为数据库不太合适，也容易不引起混淆，所以称之为数据仓库。

这里可以说一下，数据处理大致可以分成两大类：OLTP(联机事务处理)和OLAP(联机分析处理)。

* OLTP(联机事务处理)就是操作型数据库的主要应用，更侧重于基本的、日常的事务处理，包括数据的增删改查。
* OLAP(联机分析处理)就是分析型数据库的主要应用，以多维度的方式分析数据， 这个后续会整理。

**二、数据仓库有什么特点**

相对于数据库，数据仓库有以下特点

**(1)面向主题**

数据仓库通过一个个主题域将多个业务系统的数据加载到一起，为了各个主题(如：用户、订单、商品等)进行分析而建，操作型数据库是为了支撑各种业务而建立。

**(2)集成性**

数据仓库会将不同源数据库中的数据汇总到一起。

**(3)历史性**

较之操作型数据库，数据仓库的数据是为企业数据分析而建立，所以数据被加载后一般情况下将被长期保留，前者通常保存几个月，后者可能几年甚至几十年。

**(4)时变性**

是指数据仓库包含来自其时间范围不同时间段的数据快照，有了这些数据快照以后，用户便可将其汇总，生成各历史阶段的数据分析报告。

**(5)稳定性**

数据仓库中的数据一般仅执行查询操作，很少会有删除和更新。但是需定期加载和刷新数据。

**三、为什么搭建数据仓库**

简单来说，就是为了有效分析数据 。

你说直接从业务数据库中取数据来做分析?

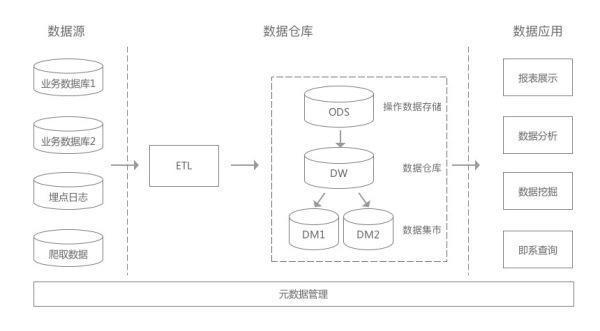
也不是不可以，就是业务系统多，业务复杂时，会发现结构复杂，数据脏乱，难以理解，缺少历史，大规模查询缓慢这些问题。

业务到一定规模，大家需要面临的问题越来越复杂和深入，数据需求不再只是昨日的营收，上月的uv这些，而是“28到45岁女性在社区的活跃度与公司策划的专题内容活动的关系”这类精细化的分析，而从数据库是很难取出这类数据的。

毕竟业务型数据库是为了支撑业务设计的，不是为了查询和分析数据。

**四、数据仓库结构**

简单来说，就是把各数据源的数据ETL到数仓中，数仓再对数据进行集成和统计，然后再输出给各数据应用，图中涉及的模块，接下来会分别介绍。

[](http://s5.51cto.com/oss/201807/13/5c765e00dffcd96554285b6ec2758abf.jpg-wh_651x-s_418910562.jpg)

**五、ETL**

ETL分别代表：抽取extraction、转换transformation、加载load。

**(1)抽取(Extract)**

从数据来源提取指定数据，数据是需要指定的，不是所有的数据都要抽取过来， 某些源数据对于分析而言没有价值，或者其可能产生的价值，远低于储存这些数据所需要的数据仓库的实现和性能上的成本，就不会抽取了。

**(2)转换(Transform)**

将数据转换为指定格式并进行数据清洗保证数据质量。

数据转换，如包括编码转换(m/f->男/女)，字段转换(balance->bal)，度量单位的转换(cm->m)，数据粒度的转换。业务系统数据存储非常明细的数据，而数据仓库中数据是用分析的，不需要非常明细，会将业务系统数据按照数据仓库粒度进行聚合。

数据清洗，如会对不完整数据，错误数据和重复数据等脏数据进行清洗。

**(3)加载(Load)**

将转换过后的数据加载到目标数据仓库，加载可分为两种：

* **全量加载：**一次对全部数据进行加载。
* **增量加载：**一般首次需要全量加载，但是在第二次周期或者第三次周期的时候仍然全量加载的话，耗费了极大的物理和时间资源。有可能部分数据源并未发生变化，而有的数据源可能只是增加了少量的数据。 对数据源中的数据只考虑新修改的记录和新插入的记录就是增量加载。

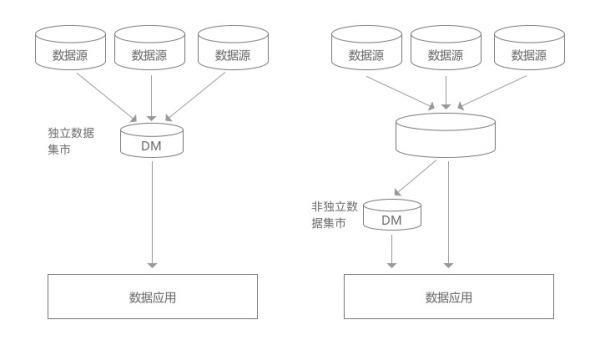
ETL很可能是数据仓库开发中最耗时最耗资源的一个环节，因为该环节要整理各大业务系统中杂乱无章的数据，并协调元数据上的差别，工作量很大，但也是构建数据仓库的重要环节，对数据仓库的后续环节影响比较大。

**六、数据集市**

数据集市(DM)可以理解为是一种“小型数据仓库”，一般面向部门、单个主题或特定应用，且之间互不影响。

**可以分为以下两种：**

* **独立数据集市：**有自己的源数据库和ETL架构;
* **非独立数据集市：**没有自己的源数据，它的数据来自数据仓库。当用户或者应用程序不需要/不必要/不允许访问整个数仓数据时，就可以直接访问数据集市，为用户提供一个数据仓库的“子集”。

[](http://s3.51cto.com/oss/201807/13/aece75e1e85d8c2b86557e7fa03742af.jpg)

简单理解就是一个结构完全和数仓一样，有ETL，然后自己存储和计算;另一种就是直接用数仓处理过的数据，再次进行组合集成。可能后面结合数据分层更好理解。

**七、ODS**

**ODS：**全称是Operational Data Store，操作数据存储。明细数据

存储各大业务型数据库ETL后的数据，是最接近数据源中数据的一层，主要目的是为了数据集中。

总体上大多是按照源业务系统的分类方式而分类的，因此会具有鲜明的业务数据库的特征，甚至还具有一定的关系数据库中的数据范式的组织形式。

但是不等同于原始数据，数据格式按照数仓要求统一，并经过简单的清洗。

**八、元数据**

元数据(Meta Date)，即数据的数据，元数据可分为技术元数据和业务元数据。

技术元数据为开发和管理数据仓库的IT 人员使用，描述了与数据仓库开发、管理和维护相关的数据，包括数据源信息、数据转换描述、数据仓库模型、数据清洗与更新规则、数据映射和访问权限等。

而业务元数据为管理层和业务分析人员服务，从业务角度描述数据，包括商务术语、数据仓库中有什么数据、数据的位置和数据的可用性等，帮助业务人员更好地理解数据仓库中哪些数据是可用的以及如何使用。

### **几个概念**

维度:就是数据的聚合。

**热数据**：是需要被计算节点频繁访问的在线类数据。  
**冷数据**：是对于离线类不经常访问的数据，比如企业备份数据、业务与操作日志数据、话单与统计数据。

热数据就近计算，冷数据集中存储

结构化数据：结构化数据也称作行数据，是可以由**二维表结构来逻辑表达和实现的数据**, 对于表结构的**每一列，都有着清晰的定义**

非结构化的数据：非结构化数据是**不方便用数据库二维逻辑表来表现的数据**，它们的特点是**数据结构不规则或不完整，没有预定义的数据模型。**

 数据处理大致可以分成两大类：联机事务处理OLTP（on-line transaction processing）、联机分析处理OLAP（On-Line Analytical Processing）。OLTP是传统的关系型[数据库](http://lib.csdn.net/base/mysql)的主要应用，主要是基本的、日常的事务处理，例如银行交易。OLAP是数据仓库系统的主要应用，支持复杂的分析操作，侧重决策支持，并且提供直观易懂的查询结果。

**OLTP**系统强调数据库内存效率，强调内存各种指标的命令率，强调绑定变量，强调并发操作；  
**OLAP**系统则强调数据分析，强调SQL执行市场，强调磁盘I/O，强调分区等。

**OLTP与OLAP之间的比较**：



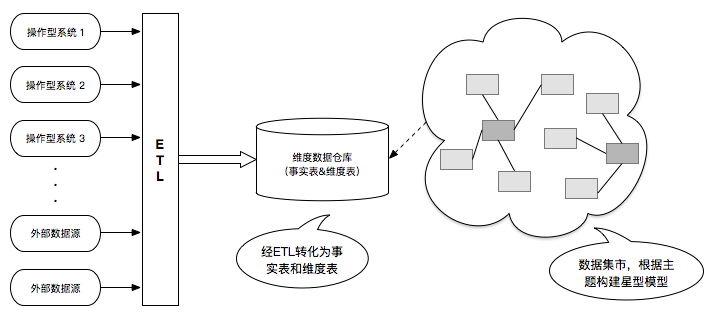
数据立方：一般是时间，地点，产品。

### 2.9.1 kimball建模

1）Kimball 的模型的数据源往往是给定的若干个数据库表，数据较为稳定但是数据之间的关联关系比较复杂，需要从这些OLTP中产生的事务型数据结构抽取出分析型数据结构。Kimball 是以最终任务为导向，将数据按照目标拆分出不同的表需求，通过ETL导入数据集市层

2）**Kimball模型将分散异构的数据源经ETL转化为事实表和维度表导入数据集市**，数据集市由若干个事实表和维度表组成

3）在数据集市将事实表和维度表根据分析主题组合后导入数据仓库中，用于数据分析

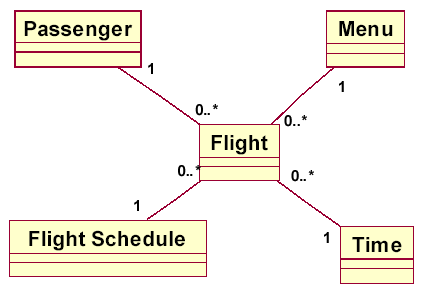


雪花模型和星型模型:

### 星型模式

星型模式的基本思想就是保持立方体的多维功能，同时也增加了小规模数据存储的灵活性。

##### 图3 一个星型模式



在图3中，星型模式使用事实 Flight 表示了一个 4 维方块（Passenger、Menu、Flight Schedulet 和 Time）。基本上，事实必须指定一个维度，以将其放入立方体的单元中。

我们的例子中的维度是：

* Passenger，描述了飞行航程中的每位乘客，由经常飞行号（frequent flyer number）指定。不是经常乘坐飞机的乘客不是数据仓库的一部分。
* Flight Schedule，是指所有常规飞行的日程。
* Menu，是用于飞行的菜单。只有对菜单进行基本的分类才会对数据挖掘有重要意义。
* Time，是指飞行的时间。

事实 Flight 描述了乘客在唯一的 Time 的单程飞行上选择 Menu。

分析空间可以是完整的方块，或者我们可以根据维度将分析空间分割成小片。

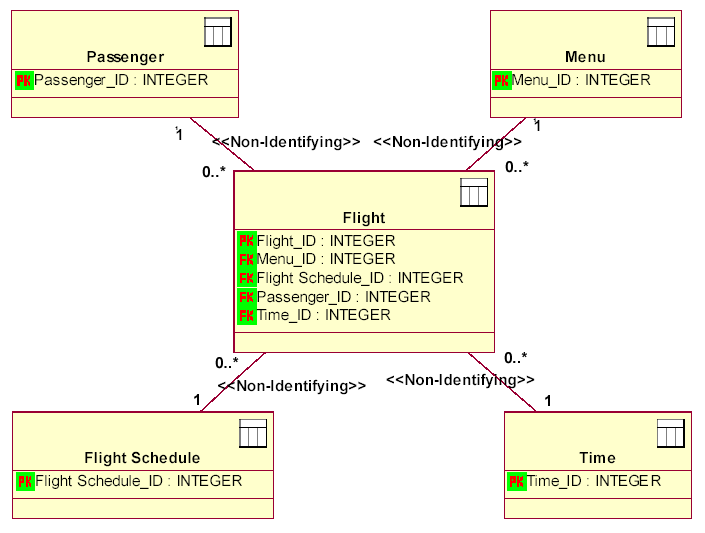
每个维度根据一个对象进行描述，对象可以用类表示，这些类就是有关业务主题的名称。这一点对于成功建立数据仓库来说是很重要的，因为仓库的用户（经理、分析员、市场）对于信息技术的术语并不是很熟悉。

事实本身就是商业智能的另一个对象，仍然通过类进行表示。

事实指每个维度。事实与维度的关联常常是一对任意，这也就意味着每个事实都与单个维度的一个单元准确对应，而维度的每个单元（每个Passenger、Time等）可以与任意数量的事实发生关联（包括0个事实）。

使用 Rational Rose 将对象模型转换为数据模型即完成了星型模式的实现。这里我们可以看到转换后的结果。

##### 图4 使用Rational Rose实现星型模式



在图4中，没有显示自动创建的主键和外键约束。

星型模式的维度是独立的表。当对象模型转换为数据模型时，Rational Rose 可以生成维度的主键。

事实表指从维度表中使用键迁移的维度，当生成数据模型时 Rational Rose 可以生成外键。

在星型模式中切片和切块是对维度的限制（选择）。这是一个运行时问题，而不是建模问题，但是模型必须分辨其需要。

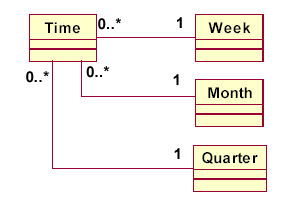
### 雪花模式

基本的星型模式并不能满足数据挖掘的所有需要。我们需要更复杂的维度，例如时间。分析员希望根据周、月、季度等识别模式。

维度必须进行规范化。我们不需要冗余的维度表，这只会使数据切片变得更加复杂。这种过程中我们得到的模式被称为雪花模式。

我们来看一个简单的雪花模式例子。我们将时间维度规范化为周、月和季度。

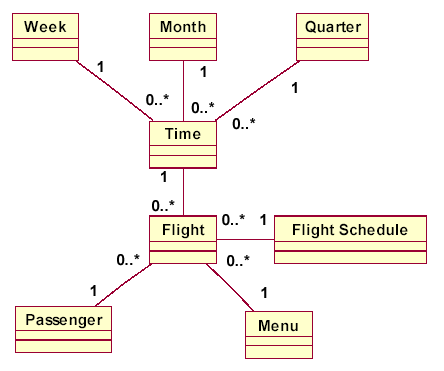
##### 图5 规范化的 Time 维度



我们希望能够使用附加的规范化维度将立方体切片：周、月和季度。在本例中，我们假定季度是月的平行层次，这也就意味着我们不能将季度假定为若干月的聚合。由于这个原因，我们将使用一张范化表（是对 OLAP 查询的一项简单附加）预先选择时间维度。

最终雪花模式添加了规范化维度。

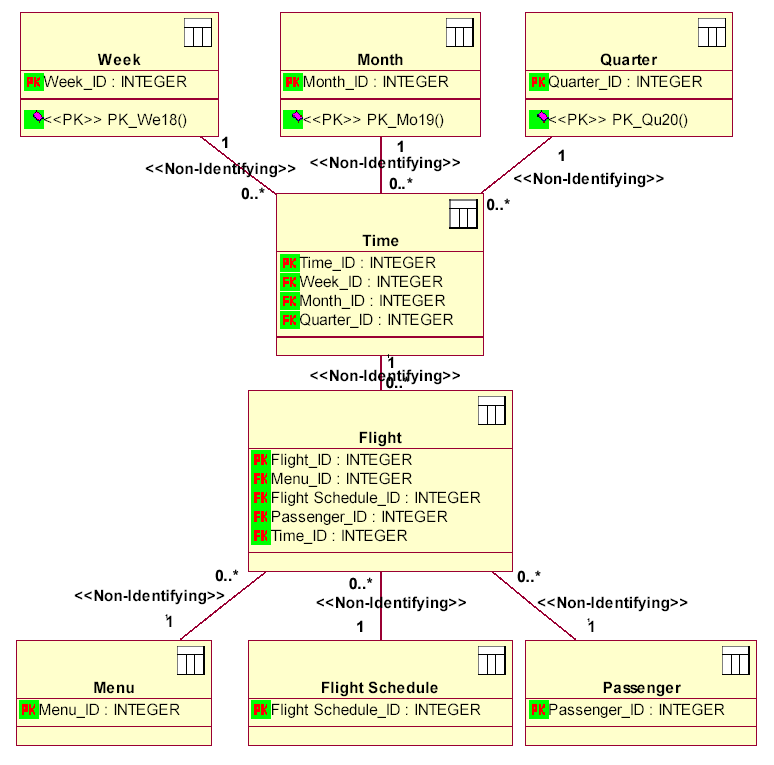
##### 图6 带有范化维度的 Time 和事实 Flight 的雪花模式



当然，所有的维度都可以像时间例子那样进行规范化，这就导致了比较复杂的数据集市模式的出现。

由 Rational Rose 从雪花模式中开发的实现模式（数据模型）是完善的。

##### 图7 带有范化 Time 维度的雪花模式的数据模型



创建的约束在图中也没有显示。

雪花模式中可以存在切片，不仅仅在基本的 Time 维度上，也可以在规范化的 Week、Month 和 Quarter 维度上。

|  |  |  |
| --- | --- | --- |
| 属性 | 星型模型 | 雪花模型 |
| 数据总量 | 多 | 少 |
| 可读性 | 容易 | 差 |
| 表个数 | 少 | 多 |
| 查询速度 | 快 | 慢 |
| 冗余度 | 高 | 低 |
| 对实时表的情况 | 增加宽度 | 字段比较少，冗余低 |
| 扩展性 | 差 | 好 |

其实这两种数据都是基于事实表，里面存的都是最小维度

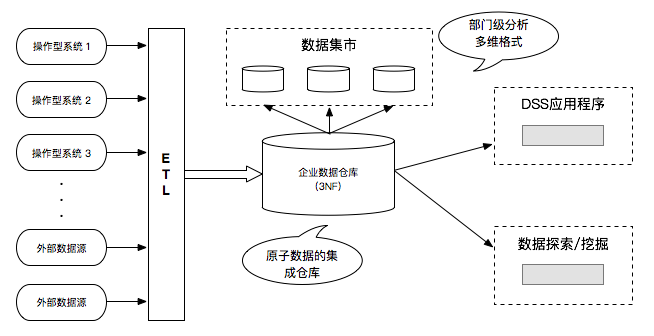
### 2.9.2 Inmon建模

1）操作型系统的数据和体系外数据需要经过ETL过程，加载到企业数据仓库中

2）企业数据仓库是企业信息化工厂的枢纽，是原子数据的集成仓库，其目的是将附加的数据存储用于各类分析型系统；在数据仓库中会对数据进行清洗，并抽取实体-关系。

3）数据集市是针对不同主题的聚集区域

基本上就是将数据库的数据一对一导入进数据仓库中。要满足数据库的三大范式（3NF）



3NF:

1. 第一范式是最基本的范式。如果数据库表中的所有字段值都是不可分解的原子值，就说明该数据库表满足了第一范式。
2. 第二范式在第一范式的基础之上更进一层。第二范式需要确保数据库表中的每一列都和主键相关，而不能跟主键没有关系。也就是说在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。
3. 第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

### 2.9.3 逻辑层架构

**分为：**

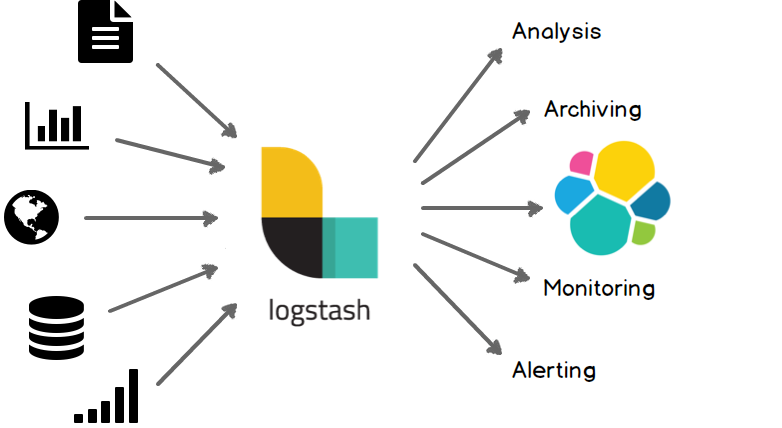
**STAGE**接口信息模型、ODS/DWD细节数据层，MID轻度综合层、DM数据集市、元数据信息模型。

|  |  |  |
| --- | --- | --- |
| 数据模型规范要求的层次 | 数据中心物理模型层次名称 | 物理表前缀（参考） |
|  | 维度数据层 | DIM\_ |
| 数据集市层(DM) | 展示层 | DM\_ |
| 主题域 | DW\_F\_ |
| 轻度汇总层(MID) | 轻度汇总层 | DW\_M\_ |
| 细节数据层（ODS/DWD） | 细节数据层 | DW\_V\_ |
| 接口层（STAGE） | 接口层 | SRC\_ |

**STAGE**接口信息模型、ODS/DWD细节数据层，MID轻度综合层、DM数据集市、元数据信息模型。

## 2.10 Logstash

### 简介



logstash是一个实时流水式开源数据收集引擎。具有强大的plugin。可以根据自己的业务场景选择不同的input filter output。绝大多数情况下都是结合ElasticSearch Kibana一起使用的，俗称ELK。

### 模块介绍

Logstash使用管道方式进行日志的搜集处理和输出。有点类似\*NIX系统的管道命令 xxx | ccc | ddd，xxx执行完了会执行ccc，然后执行ddd。

在logstash中，包括了三个阶段:

输入input –> 处理filter（不是必须的） –> 输出output

### 配置文件说明

前面介绍过logstash基本上由三部分组成，input、output以及用户需要才添加的filter，因此标准的配置文件格式如下：

input { } filter { } output { }

### 执行说明

|  |  |  |
| --- | --- | --- |
| 1 |  | bin/logstash -f demo.conf |

## 2.11 MySQL(RDS)

数据库事务的四个级别：

**·**未提交读(Read Uncommitted)：允许脏读，也就是可能读取到其他会话中未提交事务修改的数据

**·**提交读(Read Committed)：只能读取到已经提交的数据。Oracle等多数数据库默认都是该级别 (不重复读)

**·**可重复读(Repeated Read)：可重复读。在同一个事务内的查询都是事务开始时刻一致的，InnoDB默认级别。在SQL标准中，该隔离级别消除了不可重复读，但是还存在幻象读。就读的时候随便也还是这个，是一个事务

**·**串行读(Serializable)：完全串行化的读，每次读都需要获得表级共享锁，读写相互都会阻塞

### 关系型数据库

    关系型数据库，是指采用了关系模型来组织数据的数据库。  
    关系模型是在1970年由IBM的研究员E.F.Codd博士首先提出的，在之后的几十年中，关系模型的概念得到了充分的发展并逐渐成为主流数据库结构的主流模型。  
    简单来说，关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。  
**关系模型中常用的概念：  
    关系**：可以理解为一张二维表，每个关系都具有一个关系名，就是通常说的表名  
    **元组**：可以理解为二维表中的一行，在数据库中经常被称为记录  
    **属性**：可以理解为二维表中的一列，在数据库中经常被称为字段  
    **域**：属性的取值范围，也就是数据库中某一列的取值限制  
    **关键字**：一组可以唯一标识元组的属性，数据库中常称为主键，由一个或多个列组成  
    **关系模式**：指对关系的描述。其格式为：关系名(属性1，属性2， ... ... ，属性N)，在数据库中成为表结构  
  
关系型数据库的优点：  
    容易理解：二维表结构是非常贴近逻辑世界的一个概念，关系模型相对网状、层次等其他模型来说更容易理解  
    使用方便：通用的SQL语言使得操作关系型数据库非常方便  
    易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率  
  
**关系型数据库瓶颈**  
  1).高并发读写需求  
    网站的用户并发性非常高，往往达到每秒上万次读写请求，对于传统关系型数据库来说，硬盘I/O是一个很大的瓶颈  
  2).海量数据的高效率读写  
    网站每天产生的数据量是巨大的，对于关系型数据库来说，在一张包含海量数据的表中查询，效率是非常低的  
  3).高扩展性和可用性  
    在基于web的结构当中，数据库是最难进行横向扩展的，当一个应用系统的用户量和访问量与日俱增的时候，数据库却没有办法像web server和app server那样简单的通过添加更多的硬件和服务节点来扩展性能和负载能力。对于很多需要提供24小时不间断服务的网站来说，对数据库系统进行升级和扩展是非常痛苦的事情，往往需要停机维护和数据迁移。  
  
**对网站来说，关系型数据库的很多特性不再需要了**：  
  事务一致性: 关系型数据库在对事物一致性的维护中有很大的开销，而现在很多web2.0系统对事物的读写一致性都不高  
  读写实时性: 对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比如发一条消息之后，过几秒乃至十几秒之后才看到这条动态是完全可以接受的  
  复杂SQL，特别是多表关联查询: 任何大数据量的web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的复杂SQL报表查询，特别是SNS类型的网站（SNS，专指社交网络服务，包括了社交软件和社交网站。），从需求以及产品阶级角度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能极大的弱化了  
  
    在关系型数据库中，导致性能欠佳的最主要原因是多表的关联查询，以及复杂的数据分析类型的复杂SQL报表查询。为了保证数据库的ACID特性，我们必须尽量按照其要求的范式进行设计，关系型数据库中的表都是存储一个格式化的数据结构。每个元组字段的组成都是一样，即使不是每个元组都需要所有的字段，但数据库会为每个元组分配所有的字段，这样的结构可以便于标语表之间进行链接等操作，但从另一个角度来说它也是关系型数据库性能瓶颈的一个因素。

### NoSQL

    NoSQL一词首先是Carlo Strozzi在1998年提出来的，指的是他开发的一个没有SQL功能，轻量级的，开源的关系型数据库。这个定义跟我们现在对NoSQL的定义有很大的区别，它确确实实字如其名，指的就是“没有SQL”的数据库。但是NoSQL的发展慢慢偏离了初衷，我们要的不是“no sql”，而是“no relational”，也就是我们现在常说的非关系型数据库了。  
    2009年初，Johan Oskarsson举办了一场关于开源分布式数据库的讨论，Eric Evans在这次讨论中再次提出了NoSQL一词，用于指代那些非关系型的，分布式的，且一般不保证遵循ACID原则的数据存储系统。Eric Evans使用NoSQL这个词，并不是因为字面上的“没有SQL”的意思，他只是觉得很多经典的关系型数据库名字都叫“\*\*SQL”,所以为了表示跟这些关系型数据库在定位上的截然不同，就是用了“NoSQL“一词。  
注：数据库事务必须具备ACID特性，ACID是Atomic原子性，Consistency一致性，Isolation隔离性，Durability持久性。  
    非关系型数据库提出另一种理念，例如，以键值对存储，且结构不固定，每一个元组可以有不一样的字段，每个元组可以根据需要增加一些自己的键值对，这样就不会局限于固定的结构，可以减少一些时间和空间的开销。使用这种方式，用户可以根据需要去添加自己需要的字段，这样，为了获取用户的不同信息，不需要像关系型数据库中，要对多表进行关联查询。仅需要根据id取出相应的value就可以完成查询。但非关系型数据库由于很少的约束，他也不能够提供像SQL所提供的where这种对于字段属性值情况的查询。并且难以体现设计的完整性。他只适合存储一些较为简单的数据，对于需要进行较复杂查询的数据，SQL数据库显的更为合适。  
  
**2-1.非关系型数据库分类**  
    由于非关系型数据库本身天然的多样性，以及出现的时间较短，因此，不想关系型数据库，有几种数据库能够一统江山，非关系型数据库非常多，并且大部分都是开源的。  
    这些数据库中，其实实现大部分都比较简单，除了一些共性外，很大一部分都是针对某些特定的应用需求出现的，因此，对于该类应用，具有极高的性能。依据结构化方法以及应用场合的不同，主要分为以下几类：  
    **1).面向高性能并发读写的key-value数据库**：key-value数据库的主要特点即使具有极高的并发读写性能，Redis,Tokyo Cabinet,Flare就是这类的代表  
    **2).面向海量数据访问的面向文档数据库**：这类数据库的特点是，可以在海量的数据中快速的查询数据，典型代表为MongoDB以及CouchDB  
    **3).面向可扩展性的分布式数据库**：这类数据库想解决的问题就是传统数据库存在可扩展性上的缺陷，这类数据库可以适应数据量的增加以及数据结构的变化

### 关系型数据库 V.S. 非关系型数据库

    关系型数据库的最大特点就是事务的一致性：传统的关系型数据库读写操作都是事务的，具有ACID的特点，这个特性使得关系型数据库可以用于几乎所有对一致性有要求的系统中，如典型的银行系统。  
    但是，在网页应用中，尤其是SNS应用中，一致性却不是显得那么重要，用户A看到的内容和用户B看到同一用户C内容更新不一致是可以容忍的，或者说，两个人看到同一好友的数据更新的时间差那么几秒是可以容忍的，因此，关系型数据库的最大特点在这里已经无用武之地，起码不是那么重要了。  
    相反地，关系型数据库为了维护一致性所付出的巨大代价就是其读写性能比较差，而像微博、facebook这类SNS的应用，对并发读写能力要求极高，关系型数据库已经无法应付(在读方面，传统上为了克服关系型数据库缺陷，提高性能，都是增加一级memcache来静态化网页，而在SNS中，变化太快，memchache已经无能为力了)，因此，必须用新的一种数据结构存储来代替关系数据库。  
    关系数据库的另一个特点就是其具有固定的表结构，因此，其扩展性极差，而在SNS中，系统的升级，功能的增加，往往意味着数据结构巨大变动，这一点关系型数据库也难以应付，需要新的结构化数据存储。  
    于是，非关系型数据库应运而生，由于不可能用一种数据结构化存储应付所有的新的需求，因此，非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。

    必须强调的是，数据的持久存储，尤其是海量数据的持久存储，还是需要一种关系数据库这员老将。

## 2.12 MongoDB

MongoDB 是由C++语言编写的，是一个基于分布式文件存储的开源数据库系统。

在高负载的情况下，添加更多的节点，可以保证服务器性能。

MongoDB 旨在为WEB应用提供可扩展的高性能数据存储解决方案。

MongoDB 将数据存储为一个文档，数据结构由键值(key=>value)对组成。MongoDB 文档类似于 JSON 对象。字段值可以包含其他文档，数组及文档数组

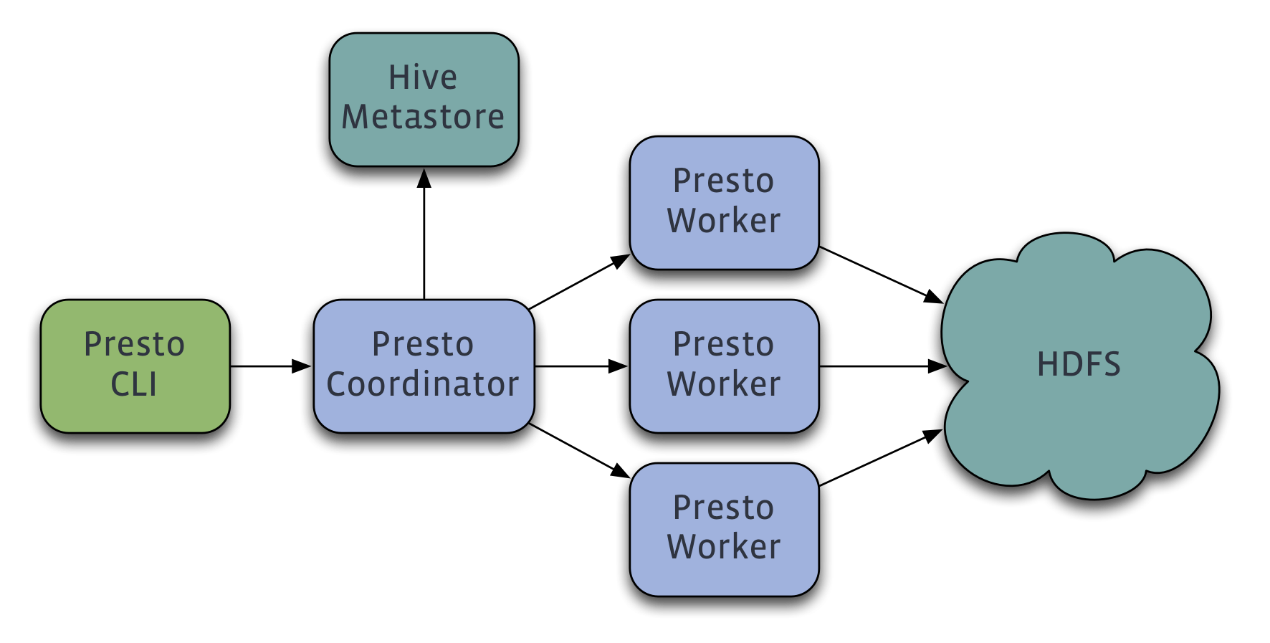


Mongo属于NoSQL数据库

## 2.13 Presto

Presto是一个分布式sql查询引擎，支持响应式查询，多数据源，大数据数据量。它源于facebook。

包含coordinator和多个worker。Coordinator解析sql，变成查询计划，然后分发给worker进行处理。它具有低延时性。



## 2.14 SpringBoot

使用Spring Boot可以让我们快速创建一个基于Spring的项目，而让这个Spring项目跑起来我们只需要很少的配置就可以了。Spring Boot主要有如下核心功能：

1.独立运行的Spring项目

Spring Boot可以以jar包的形式来运行，运行一个Spring Boot项目我们只需要通过java -jar xx.jar类运行。非常方便。

2.内嵌Servlet容器

Spring Boot可以内嵌Tomcat，这样我们无需以war包的形式部署项目。

3.提供starter简化Maven配置

使用Spring或者SpringMVC我们需要添加大量的依赖，而这些依赖很多都是固定的，这里Spring Boot 通过starter能够帮助我们简化Maven配置。

4.自动配置Spring

5.准生产的应用监控

6.无代码生成和xml配置

## 2.15 SLF4J

SLF4J的全称是Simple Logging Facade for Java，即简单日志门面。SLF4J并不是具体的日志框架，而是作为一个简单门面服务于各类日志框架，如java.util.logging, logback和log4j。

SLF4J提供了统一的记录日志的接口，对不同日志系统的具体实现进行了抽象化，只要按照其提供的方法记录即可，最终日志的格式、记录级别、输出方式等通过绑定具体的日志系统来实现。

使用SLF4J的好处在于，你只需要按统一的方式写记录日志的代码，如：

public class LoggerTest {

private static final Logger logger = LoggerFactory.getLogger(Tester.class);

public static void main(String[] args) {

logger.info("Current Time: {}", System.currentTimeMillis());

}

}

*SLF4J支持{}作为占位符，等价于C语言中的%s，而不必再进行字符串的拼接，效率有显著的提升（见后面运行结果）。*

而无需关心日志是通过哪个日志系统，以什么风格输出的。因为它们取决于部署项目时绑定的日志系统。   
例如，在项目中使用了SLF4J记录日志，并且绑定了log4j，则日志会以log4j的风格输出；后期需要改为以logback的风格输出日志，只需要将log4j替换成logback即可，不用修改项目中的代码

# 第3章 项目配置

## 3.1 Nginx

### 3.1.1 Nginx 安装

注意：以下操作均使用 root 用户。

* 1. 安装 PCRE

1. 下载 PCRE 安装包

wget http://downloads.sourceforge.net/project/pcre/pcre/8.35/pcre-8.35.tar.gz

1. 解压安装包

tar zxvf pcre-8.35.tar.gz

1. 进入安装包目录

cd pcre-8.35

1. 编译安装

./configure

如果编译过程中报错：error: You need a C++ compiler for C++ support，这是由于没有安装 gcc，输入以下指令进行安装：

yum install -y gcc gcc-c++

安装 pcre：

make && make install

1. 查看 pcre 版本

pcregrep -V

* 1. 安装 Nginx

1. 解压安装包

tar zxvf nginx-1.12.2.tar.gz

1. 进入安装包目录

cd nginx-1.12.2

1. 编译安装

./configure --prefix=/usr/local/webserver/nginx --with-http\_stub\_status\_module

--with-http\_ssl\_module --with-pcre=/usr/local/src/pcre-8.35

make && make install

如果报出以下错误，需要安装 OpenSSL

./configure: error: SSL modules require the OpenSSL library.

安装指令如下：

yum -y install openssl openssl-devel

1. 查看 Nginx 版本

/usr/local/webserver/nginx/sbin/nginx -v

### 3.1.2 Nginx 负载均衡配置

代码清单 3-1 Nginx负载均衡配置

#user nobody;

worker\_processes 1;

#error\_log logs/error.log;

#error\_log logs/error.log notice;

#error\_log logs/error.log info;

#pid logs/nginx.pid;

events {

worker\_connections 1024;

}

http {

include mime.types;

default\_type application/octet-stream;

#log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '

1. '$status $body\_bytes\_sent "$http\_referer" '
2. '"$http\_user\_agent" "$http\_x\_forwarded\_for"';

#access\_log logs/access.log main;

sendfile on;

#tcp\_nopush on;

#keepalive\_timeout 0;

keepalive\_timeout 65;

#服务器的集群

upstream netitcast.com {

#服务器集群名字

#服务器配置 weight 是权重的意思，权重越大，分配的概率越大。

#server 127.0.0.1:18080;

#server 127.0.0.1:28080;

server 192.168.10.200:18080;

server 192.168.10.200:28080;

}

server {

listen 80;

server\_name localhost;

location / {

proxy\_pass http://netitcast.com;

proxy\_redirect default;

}

#error\_page 404 /404.html;

1. redirect server error pages to the static page /50x.html

error\_page 500 502 503 504 /50x.html; location = /50x.html {

root html;

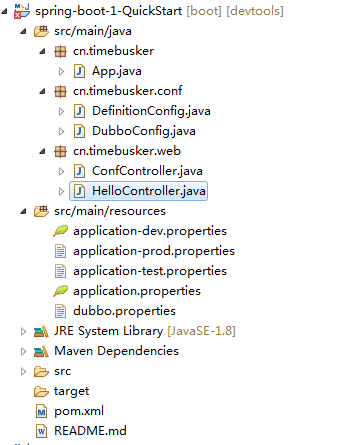
}

}

}

## 3.2 SpringBoot

（使用Spring Boot，无需再做tomcat配置）



## SL4J日志配置

集成logback与SL4J一起使用，在磁盘打印日志，然后收集。

<?xml version="1.0" encoding="UTF-8"?>  
<!--debug="true" : 打印logback内部状态（默认当logback运行出错时才会打印内部状态 ），配置该属性后打印条件如下（同时满足）：   
 1、找到配置文件 2、配置文件是一个格式正确的xml文件 也可编程实现打印内部状态，例如： LoggerContext lc = (LoggerContext)   
 LoggerFactory.getILoggerFactory(); StatusPrinter.print(lc); -->  
<!-- scan="true" ： 自动扫描该配置文件，若有修改则重新加载该配置文件 -->  
<!-- scanPeriod="30 seconds" : 配置自动扫面时间间隔（单位可以是：milliseconds, seconds, minutes   
 or hours，默认为：milliseconds）， 默认为1分钟，scan="true"时该配置才会生效 -->  
<configuration debug="false" scan="true" scanPeriod="30 seconds" packagingData="true">  
 <!-- 设置 logger context 名称,一旦设置不可改变，默认为default -->  
 <contextName>myAppName</contextName>  
  
 <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
 <!-- encoders are by default assigned the type ch.qos.logback.classic.encoder.PatternLayoutEncoder -->  
 <encoder>  
 <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>  
 </encoder>  
 </appender>  
  
 <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">  
 <!-- 当前活动日志文件名 -->  
 <file>./my\_log.log</file>  
 <!-- 文件滚动策略根据%d{patter}中的“patter”而定，此处为每天产生一个文件 -->  
 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  
 <!-- 归档文件名“.zip或.gz结尾”,表示归档文件自动压缩 -->  
 <FileNamePattern>./my\_log%d{yyyyMMdd}.log.zip</FileNamePattern>  
 <maxHistory>30</maxHistory>  
 </rollingPolicy>  
  
 <!--rollingPolicy class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">  
 <fileNamePattern>renhai%i.log</fileNamePattern>  
 <minIndex>1</minIndex>  
 <maxIndex>10</maxIndex>  
 </rollingPolicy>  
   
 <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">  
 <maxFileSize>20MB</maxFileSize>  
 </triggeringPolicy-->  
  
 <!-- <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">  
 rollover daily  
 <fileNamePattern>mylog-%d{yyyy-MM-dd}.%i.log</fileNamePattern>  
 each file should be at most 30MB, keep 60 days worth of history, but at most 20GB  
 <maxFileSize>30MB</maxFileSize>   
 <maxHistory>60</maxHistory>  
 <totalSizeCap>20GB</totalSizeCap>  
 </rollingPolicy> -->  
  
  
 <encoder>  
 <pattern>%d{HH:mm:ss.SSS}[%-5level][%thread]%logger{36} - %msg%n</pattern>  
 <!-- <pattern>%d{HH:mm:ss.SSS}[%-5level][%thread] - %msg%n</pattern> -->  
 </encoder>  
 </appender>  
  
 <!-- 日志级别若没显示定义，则继承最近的父logger（该logger需显示定义level,直到rootLogger）的日志级别-->  
 <!-- logger的appender默认具有累加性（默认日志输出到当前logger的appender和所有祖先logger的appender中），可通过配置 “additivity”属性修改默认行为-->  
 <logger name="com.yinz" level="debug" additivity="false" >  
 <appender-ref ref="FILE"/>  
 </logger>  
  
 <!-- 至多只能配置一个root -->  
 <root level="debug">  
 <appender-ref ref="STDOUT" />  
 <appender-ref ref="FILE" />  
 </root>  
</configuration>

## 3.4 Flume

### 3.3.1 多层 Flume 部署

数据采集层 Flume 配置

代码清单 3-7 数据采集层Flume配置

a1.sources = r1

a1.channels = c1

a1.sinkgroups = g1

a1.sinks = k1 k2

a1.sources.r1.type = TAILDIR

a1.sources.r1.channels = c1

a1.sources.r1.positionFile=/mnt/flume/taildir\_position.json

a1.sources.r1.filegroups = f1

a1.sources.r1.filegroups.f1=/tmp/logs/app.+

a1.sources.r1.fileHeader = true

a1.channels.c1.type = file

a1.channels.c1.checkpointDir = /opt/modules/flume/checkpoint/behavior

a1.channels.c1.dataDirs = /opt/modules/flume/data/behavior/

a1.channels.c1.maxFileSize = 104857600

a1.channels.c1.capacity = 90000000

a1.channels.c1.keep-alive = 60

a1.sinkgroups.g1.sinks = k1 k2

a1.sinkgroups.g1.processor.type = load\_balance

a1.sinkgroups.g1.processor.backoff = true

a1.sinkgroups.g1.processor.selector = round\_robin

a1.sinkgroups.g1.processor.selector.maxTimeOut=10000

a1.sinks.k1.type = avro

a1.sinks.k1.channel = c1

a1.sinks.k1.batchSize = 1

a1.sinks.k1.hostname = hadoop-senior02.itguigu.com

a1.sinks.k1.port = 1234

a1.sinks.k2.type = avro

a1.sinks.k2.channel = c1

a1.sinks.k2.batchSize = 1

a1.sinks.k2.hostname = hadoop-senior03.itguigu.com

a1.sinks.k2.port = 1234

聚合层 Flume 配置

代码清单 3-8 聚合层Flume配置

a1.sources = r1

a1.channels = c1

a1.sinks = k1

a1.sources.r1.type = avro

a1.sources.r1.channels = c1

a1.sources.r1.bind = 0.0.0.0

a1.sources.r1.port = 1234

a1.channels.c1.type = file

a1.channels.c1.checkpointDir = /opt/modules/flume/checkpoint/behavior\_collect

a1.channels.c1.dataDirs = /opt/modules/flume/data/behavior\_collect

a1.channels.c1.maxFileSize = 104857600

a1.channels.c1.capacity = 90000000

a1.channels.c1.keep-alive = 60

a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink

a1.sinks.k1.topic = analysis-test

a1.sinks.k1.brokerList= localhost:9092

a1.sinks.k1.requiredAcks = 1

a1.sinks.k1.kafka.producer.type = sync

a1.sinks.k1.batchSize = 1

a1.sinks.k1.channel = c1

### 3.3.2 单层 Flume 部署

代码清单 3-9 单层Flume配置

a1.sources = r1

a1.channels = c1

a1.sinks = k1

a1.sources.r1.type = TAILDIR

a1.sources.r1.channels = c1

a1.sources.r1.positionFile=/mnt/flume/taildir\_position.json

a1.sources.r1.filegroups = f1

a1.sources.r1.filegroups.f1=/tmp/logs/app.+

a1.sources.r1.fileHeader = true

a1.channels.c1.type = file

a1.channels.c1.checkpointDir = /mnt/flume/checkpoint/behavior\_collect

a1.channels.c1.dataDirs = /mnt/flume/data/behavior\_collect

a1.channels.c1.maxFileSize = 104857600

a1.channels.c1.capacity = 90000000

a1.channels.c1.keep-alive = 60

a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink

a1.sinks.k1.topic = analysis-test

a1.sinks.k1.brokerList= localhost:9092

a1.sinks.k1.requiredAcks = 1

a1.sinks.k1.kafka.producer.type = sync

a1.sinks.k1.batchSize = 1

a1.sinks.k1.channel = c1

## 3.5 Kafka

### 3.5.1 创建 Kafka 主题

kafka-topics.sh --zookeeper hadoop102:2181 --create --replication-factor 3 --partitions 1 --topic topic\_app\_startup

## 3.6 Hive

### 3.6.1 Hive 创建数据库及分区表

查看数据库

hive (default)> show databases;

如果 applogs\_db 存在则删除数据库：

hive (default)> drop database applogs\_db;

创建数据库

hive (default)> create database applogsdb;

使用 applogs\_db 数据库：

hive (default)> use applogsdb;

创建分区表

--startup

CREATE external TABLE ext\_startup\_logs(userId string,appPlatform string,appId

string,startTimeInMs bigint,activeTimeInMs bigint,appVersion string,city string)PARTITIONED BY (ym string, day string,hm string) ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe' STORED AS TEXTFILE;

1. 查看数据库中的分区表

hive (applogsdb)> show tables;

5. 退出 Hive

hive (applogsdb)> quit;

### 3.6.2 Hive 执行脚本

代码清单 3-10 Hive执行脚本

#!/bin/bash

1. 获取三分钟之前的时间

systime=`date -d "-3 minute" +%Y%m-%d-%H%M`

1. 获取年月

ym=`echo ${systime} | awk -F '-' '{print $1}'`

1. 获取日

day=`echo ${systime} | awk -F '-' '{print $2}'`

1. 获取小时分钟

hm=`echo ${systime} | awk -F '-' '{print $3}'`

1. 执行 hive 命令

hive -e "load data inpath '/user/atguigu/test/${ym}/${day}/${hm}' into table loganalysisdb.ext\_startup\_logs partition(ym='${ym}',day='${day}',hm='${hm}')"

## 3.7 Spark SQL

Spark SQL 的命令基本同hive，spark执行脚本的命令

spark-sql -e "$sql" --driver-java-options "-Dlog4j.configuration=file:///mnt/log4j.properties"

## 3.8 Logstah

（未完）

# 第4章 代码和脚本解析

## 4.1 离线数据处理系统

### 4.1.1 Flume读取Kafka数据写hdfs

离线数据处理系统中的 Kafka 高级消费者程序将消息从 Kafka 集群中消费出来，然后写入指定的 HDFS 文件中。采用lzo压缩文件

代码清单 4-1 Flume 读取kafka的数据

agent.sources=r1

agent.sinks=k1

agent.channels=c1

agent.sources.r1.type= org.apache.flume.source.kafka.KafkaSource

agent.sources.r1.kafka.bootstrap.servers=localhost:9092

agent.sources.r1.kafka.topics=test

#agent.sources.r1.serializer.class=kafka.serializer.StringEncoder

agent.sources.r1.kafka.consumer.group.id = flumetest

agent.sources.r1.flumeBatchSize=1000

agent.sources.r1.useFlumeEventFormat=false

agent.sources.r1.restart=true

agent.sources.r1.batchSize=1000

agent.sources.r1.batchTimeout=3000

agent.sources.r1.channels=c1

agent.channels.c1.type=memory

agent.channels.c1.capacity=102400

agent.channels.c1.transactionCapacity=1000

agent.channels.c1.byteCapacity=134217728

agent.channels.c1.byteCapacityBufferPercentage=80

agent.sinks.k1.channel=c1

agent.sinks.k1.type=hdfs

agent.sinks.k1.hdfs.path=hdfs:///tmp/flume/%Y%m%d

agent.sinks.k1.hdfs.writeFormat=Text

agent.sinks.k1.hdfs.rollSize=0

agent.sinks.k1.hdfs.rollCount=0

agent.sinks.k1.hdfs.rollInterval=120

agent.sinks.k1.hdfs.threadsPoolSize=30

agent.sinks.k1.hdfs.fileType = CompressedStream

agent.sinks.k1.hdfs.fileSuffix=.lzo

agent.sinks.k1.hdfs.codeC = lzop

### 4.1.2 Hive函数

编写udf函数

1. 聚合函数，找出非空的值

*/\*\*  
 \* 判断不是 null和空串这两种情况  
 \*/*public class NotNullValueUDAF extends AbstractGenericUDAFResolver {  
 static final Log *LOG* = LogFactory.*getLog*(NotNullValueUDAF.class.getName());  
  
 public GenericUDAFEvaluator getEvaluator(GenericUDAFParameterInfo info) throws SemanticException {  
 return new GenericUDAFHistogramNumericEvaluator();  
 }  
  
 public static class GenericUDAFHistogramNumericEvaluator extends GenericUDAFEvaluator {  
  
 private PrimitiveObjectInspector inputOI;  
  
 //这个方法返回了UDAF的返回类型，这里确定了sum自定义函数的返回类型是Long类型  
 @Override  
 public ObjectInspector init(Mode m, ObjectInspector[] parameters) throws HiveException {  
 assert (parameters.length == 1);  
 super.init(m, parameters);  
 inputOI = (PrimitiveObjectInspector) parameters[0];  
 return PrimitiveObjectInspectorFactory.*javaStringObjectInspector*;  
 }  
 @Override  
 public AbstractAggregationBuffer getNewAggregationBuffer(){  
 NotNullAgg buffer = new NotNullAgg();  
 reset(buffer);  
 return buffer;  
 }  
  
 @Override  
 public void reset(AggregationBuffer agg) {  
 NotNullAgg notNullAgg=(NotNullAgg)agg;  
 notNullAgg.val="";  
 }  
  
 @Override  
 public void iterate(AggregationBuffer agg, Object[] parameters) {  
 assert (parameters.length == 1);  
 merge(agg,parameters[0]);  
 }  
  
 @Override  
 public Object terminatePartial(AggregationBuffer agg) {  
 return terminate(agg);  
 }  
  
 @Override  
 public void merge(AggregationBuffer agg, Object partial) {  
 if(partial!=null){  
 NotNullAgg notNullAgg=(NotNullAgg)agg;  
 String input= PrimitiveObjectInspectorUtils.*getString*(partial, inputOI).trim();  
 if(StringUtils.*isNotBlank*(input)){  
// 还要判断是否存在 null nul 这两种情况  
 if(!input.equalsIgnoreCase("null") && ! input.equalsIgnoreCase("none")) {  
 notNullAgg.val = input;  
 }  
 }  
 }  
 }  
  
 @Override  
 public Object terminate(AggregationBuffer agg) {  
 NotNullAgg myagg = (NotNullAgg) agg;  
 return new Text(myagg.val);  
 }  
  
 }  
  
 @GenericUDAFEvaluator.AggregationType(estimable = true)  
 static class NotNullAgg extends GenericUDAFEvaluator.AbstractAggregationBuffer {  
 String val="";  
 }

1. UDTF 一列转多列

import java.util.ArrayList;  
  
import org.apache.commons.lang3.StringUtils;  
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;  
import org.apache.hadoop.hive.ql.metadata.HiveException;  
import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;  
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;  
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;  
import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;  
import org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInspectorFactory;  
import org.json.JSONArray;  
import org.json.JSONException;  
  
*/\*\*  
 \* 用于将json串中的json数组展平，将eventname字段提取出来，并且保存成event\_name 和event\_json  
 \*  
 \** ***@author*** *tomluo  
 \*  
 \*/*public class EventnameListFlatter extends GenericUDTF {  
  
 public static final String *EVENT\_NAME\_FIELD* = "eventname";  
  
 @Override  
 public StructObjectInspector initialize(StructObjectInspector argOIs) throws UDFArgumentException {  
  
 ArrayList<String> fieldNames = new ArrayList<String>();  
 ArrayList<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>();  
 fieldNames.add("event\_name");  
 fieldOIs.add(PrimitiveObjectInspectorFactory.*javaStringObjectInspector*);  
 fieldNames.add("event\_json");  
 fieldOIs.add(PrimitiveObjectInspectorFactory.*javaStringObjectInspector*);  
 return ObjectInspectorFactory.*getStandardStructObjectInspector*(fieldNames, fieldOIs);  
 }  
  
 @Override  
 public void process(Object[] objects) throws HiveException {  
 if (objects == null || objects[0] == null) {  
 forward(new Object[] { null, null });  
 return;  
 }  
 String input = objects[0].toString();  
 JSONArray ja = null;  
 if (StringUtils.*isNotBlank*(input)) {  
 try {  
 ja = new JSONArray(input);  
 } catch (JSONException e) {  
 e.printStackTrace();  
 }  
 if (ja == null)  
 return;  
 for (int i = 0; i < ja.length(); i++) {  
 String[] result = new String[2];  
 try {  
 result[0] = ja.getJSONObject(i).getString(*EVENT\_NAME\_FIELD*);  
 result[1] = ja.getString(i);  
 } catch (JSONException e) {  
 continue;  
 }  
 forward(result);  
 }  
 } else {  
 forward(new Object[] { "", "" });  
 }  
 }  
  
 @Override  
 public void close() throws HiveException {  
  
 }  
  
}

### 4.1.3 Hive流程语句

1.创建外部表，连接原始

-- 创建外部表，连接原始日志，对应数仓的原始数据层

CREATE EXTERNAL TABLE `stage\_originlog\_lzo\_dt`(`line` string)  
PARTITIONED BY (`dt` string)  
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'  
WITH SERDEPROPERTIES (  
 'serialization.format' = '1'  
)  
STORED AS  
 INPUTFORMAT 'com.hadoop.mapred.DeprecatedLzoTextInputFormat'  
 OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'  
LOCATION 'hdfs:/tmp/flume/';、

– 添加分区，让数据可以查看

alter table stage\_originlog\_lzo\_dt add partition (dt='2018-09-01') location 'hdfs:/tmp/flume/20180901/';

1. 创建细节表，一行转多行，同时提取数据

use test;  
add jar /home/hadoop/teach.jar;  
create temporary function base\_analizer as 'com.test.BaseFieldUDF';  
create temporary function flat\_analizer as 'com.test.EventnameUDTF';  
set hive.exec.dynamic.partition.mode=nonstrict;  
  
insert overwrite table ods\_basedata\_dt  
PARTITION (dt)  
select  
 user\_id ,   
 area ,   
 app\_version ,   
 app\_time ,   
 event\_name ,   
 event\_json ,  
 server\_time ,   
 sdk\_log.dt   
 from  
(  
select  
split(base\_analizer(line,'userId,area,appVersion,appTime'),'\t')[0] as user\_id ,  
split(base\_analizer(line,'userId,area,appVersion,appTime'),'\t')[1] as area ,  
split(base\_analizer(line,'userId,area,appVersion,appTime'),'\t')[2] as app\_version ,  
split(base\_analizer(line,'userId,area,appVersion,appTime'),'\t')[3] as app\_time ,  
split(base\_analizer(line,'userId,area,appVersion,appTime'),'\t')[4] as server\_time ,  
split(base\_analizer(line,'userId,area,appVersion,appTime'),'\t')[5] as ops ,  
dt  
from stage\_originlog\_lzo\_dt where dt='2018-09-01'   
) sdk\_log lateral view flat\_analizer(ops) tmp\_k as event\_name, event\_json;

创建用户表

CREATE EXTERNAL TABLE `dw\_history\_users`(`uid` string COMMENT 'uid', `area` string COMMENT 'area',   
`first\_dat` string COMMENT 'first appear time', `first\_version` string COMMENT 'first appear version,default 0.0.0',   
 `current\_dat` string COMMENT 'latest time', `current\_version` string COMMENT 'latest version')  
PARTITIONED BY (`dt` string);

### 4.1.4 Mongo流程语句

1.创建新闻表

db.createCollection('user\_news')

2.结构,\_id为（newsid+user\_id）

{"\_id":news\_id+user\_id, "create\_time":记录的时间，用long类型}

3.创建时间索引

db.user\_news.ensureIndex('create\_time')

## 4.2实时数据处理系统

### 4.2.1 SparkStreaming程序

流式计算做的事情是

1. 计算出新闻的热度。
2. 将新闻排序

import com.mongodb.spark.MongoSpark;  
import org.apache.kafka.clients.consumer.ConsumerRecord;  
import org.apache.spark.SparkConf;  
import org.apache.spark.api.java.JavaRDD;  
import org.apache.spark.api.java.function.Function0;  
import org.apache.spark.streaming.Durations;  
import org.apache.spark.streaming.api.java.JavaInputDStream;  
import org.apache.spark.streaming.api.java.JavaStreamingContext;  
import org.apache.spark.streaming.kafka010.\*;  
import org.bson.Document;  
  
import java.util.\*;  
  
  
public class ReadNewsStream {  
 public static void main(String[] args) throws Exception {  
 final Properties serverProps = PropertiesUtils.*getProperties*("config.properties");  
//获取 checkpoint 的 hdfs 路径  
 String checkpointPath = serverProps.getProperty("streaming.checkpoint.path");  
// 如果 checkpointPath hdfs 目录下的有文件，则反序列化文件生产 context, 否则使用函数 createContext 返回的 context 对象  
 JavaStreamingContext javaStreamingContext = JavaStreamingContext.*getOrCreate*(checkpointPath, *createContext*(serverProps));  
 javaStreamingContext.start();  
 javaStreamingContext.awaitTermination();  
 }  
  
 */\*\*  
 \* 根据配置文件以及业务逻辑创建 JavaStreamingContext  
 \*  
 \** ***@param*** *serverProps  
 \** ***@return*** *\*/* public static Function0<JavaStreamingContext> createContext(final Properties serverProps) {  
 Function0<JavaStreamingContext> createContextFunc = new Function0<JavaStreamingContext>() {  
 public JavaStreamingContext call() throws Exception {  
//获取配置中的 topic  
 String topicStr = serverProps.getProperty("kafka.topic");  
 Collection<String> topics = Arrays.*asList*(topicStr.split(","));  
 //获取配置中的 groupId  
 final String groupId = serverProps.getProperty("kafka.groupId");  
//获取批次的时间间隔，比如 5s  
 final Long streamingInterval = Long.*parseLong*(serverProps.getProperty("streaming.interval"));  
//获取 checkpoint 的 hdfs 路径  
 final String checkpointPath = serverProps.getProperty("streaming.checkpoint.path");  
//获取 kafka broker 列表  
 final String kafkaBrokerList = serverProps.getProperty("kafka.broker.list");  
//组合 kafka 参数  
 final Map<String, Object> kafkaParams = new HashMap();  
 kafkaParams.put("metadata.broker.list", kafkaBrokerList);  
 kafkaParams.put("group.id", groupId);  
  
// 创建 SparkConf 对象  
 SparkConf sparkConf = new SparkConf().setMaster("local[\*]").setAppName("spark-news");  
  
*/\*\*  
 优雅停止 Spark. 暴力停掉 sparkstreaming 是有可能出现问题的，比如你的数据源是 kafka，  
 已经加载了一批数据到 sparkstreaming 中正在处理，如果中途停掉，  
 这个批次的数据很有可能没有处理完，就被强制 stop 了，  
 下次启动时候会重复消费或者部分数据丢失。  
 \*/* sparkConf.set("spark.streaming.stopGracefullyOnShutdown", "true");  
  
 */\*\*在 Spark 的架构中，在网络中传递的或者缓存在内存、硬盘中的对象需要进行序列化操作，序列化的作用主要是利用时间换空间\*/* sparkConf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");  
  
  
 */\*\*增加 MyRegistrator 类，注册需要用 Kryo 序列化的类,Kryo 的序列化会比java的要更省空间\*/*// sparkConf.set("spark.kryo.registrator", "com.tom.MyKryoRegisrator");  
  
  
 */\*\* 每秒钟对于每个 partition 读取多少条数据如果不进行设置，Spark Streaming 会一开始就读取 partition 中的所有数据到内存，给内存造成巨大压力  
 设置此参数后可以很好地控制 Spark Streaming 读取的数据量，也可以说控制了读取的进度 \*/* sparkConf.set("spark.streaming.kafka.maxRatePerPartition", "100");  
  
 /\* 创建 javaStreamingContext，设置 每隔5s 执行一次\*/  
 JavaStreamingContext javaStreamingContext = new JavaStreamingContext(sparkConf, Durations.*seconds*(streamingInterval));  
 javaStreamingContext.checkpoint(checkpointPath);  
//创建 kafka DStream  
 final JavaInputDStream<ConsumerRecord<String, String>> kafkaMessage = KafkaUtils.*createDirectStream*(javaStreamingContext, LocationStrategies.*PreferConsistent*(),  
 ConsumerStrategies.<String, String>*Subscribe*(topics, kafkaParams));  
  
//需要把每个批次的 offset 保存  
 kafkaMessage.foreachRDD(rdd -> {  
 /\*表示具有[[OffsetRange]]集合的任何对象，这可以用来访问由直 Direct Kafka DStream 生成的 RDD 中的偏移量范围\*/  
 OffsetRange[] offsetRanges = ((HasOffsetRanges) rdd.rdd()).offsetRanges();  
 /\*逻辑处理在这里\*/  
 *executeData*(rdd);  
 /\*kafka offset 写入 zk\*/  
 ((CanCommitOffsets) kafkaMessage.inputDStream()).commitAsync(offsetRanges);  
 });  
//将 kafka 中的消息转换成对象并过滤不合法的消息  
 return javaStreamingContext;  
 }  
 };  
 return createContextFunc;  
 }  
  
 static private void executeData(JavaRDD<ConsumerRecord<String, String>> rdd) {  
//1.计算威尔逊热度  
 JavaRDD<String> result = rdd.filter(stringStringConsumerRecord -> {  
 /\*过滤出来只有点击的日志\*/  
 String content = stringStringConsumerRecord.value();  
 return content.contains("action");  
 }).map(stringStringConsumerRecord -> {  
 /\*组装成新闻的点击格式\*/  
 Document document = Document.parse();  
 return stringStringConsumerRecord.value();  
 });  
// 2.写入mongoDB  
// MongoSpark.save(result);  
 }  
}

# 第5章 项目调试与运行

## 5.1 Nginx负载均衡

### 5.1.1 Nginx启动

sudo /usr/local/webserver/nginx/sbin/nginx -c /opt/modules/nginx-1.12.2/conf/nginx.conf

### 5.1.2 SpringBoot启动

mvn install

cd target

java -jar xxxx.jar

## 5.2 公共日志采集系统

### 5.2.1 Zookeeper集群启动

ZkServer.sh start

### 5.2.2 数据采集层Flume启动

flume-ng agent --classpath /opt/modules/flume/lib/flume-taildirsource.jar --conf /opt/modules/flume/conf/ -f /opt/modules/flume/conf/flume-analysis.conf -n a1

### 5.2.3 Kafka 集群启动

kafka-server-start.sh –daemon config/server.properties

## 5.3 离线系统

### 5.3.1 Hadoop集群启动

start-dfs.sh

start-yarn.sh

### 5.3.2 Hive数据仓库启动

hive –-service hiveserver2

hive –-service metastore

hive

### 5.3.3 调度任务启动

service crond start

### 5.3.4 新增用户统计

此处需要先计算出历史用户表，然后再根据用户**第一次出现的时间**为当天时间，算出

每日新增：

select count(user\_id) from dw\_history\_users where dt='2018-09-26' and current\_dat=dt

每周新增：

select count(user\_id) from dw\_history\_users where dt='2018-09-26' and first\_dat<=dt  
and first\_dat> date\_add(dt, -7)

每月新增：

select count(user\_id) from dw\_history\_users where dt='2018-09-26' and first\_dat<=dt  
and first\_dat> date\_add(dt, -30)

### 5.3.5 每日新闻展示总量，平均新闻展示，新闻点击，点击率计算

通过聚合函数来计算每日新闻的总量，新闻展示

select sum(news\_id) news\_total,sum(display)/sum(news\_id) avg\_display,sum(click)/sum(news\_id) avg\_click, sum(display)/sum(click) avg\_rate  
from (  
select  
 t1.display display,  
 case when t2.click is not null then t2.click else 0 end click,  
 t1.news\_id  
 from (select count(distinct user\_id) display,news\_id from user\_display group by news\_id) t1 left join  
(select count(distinct user\_id ) click,news\_id from user\_click group by news\_id) t2 on  
t1.news\_id=t2.news\_id  
) t3;

### 5.3.6 前台，后台活跃用户数统计

此处有前端的日志上报，可以分开分析。统计每个报表的**唯一用户数.**

select t1.forground\_total,t2.background\_total,t1.dt from  
(select count(distinct uid) forground\_total,dt from dw\_foreground where dt='2018-09-25' group by dt) t1  
join (select count(distinct uid) background\_total,dt from dw\_background where dt='2018-09-25' group by dt) t2  
on t1.dt=t2.dt ;

### 5.3.7 用户版本分布统计

需要依赖每日历史用户表。聚合算出用户在每个版本的分布情况。

select count(distinct user\_id) num,app\_version from mid\_user\_history\_dt  
 where dt='2018-09-30' group by app\_version order by app\_version

### 5.3.8 沉默用户统计

沉默用户是这样。可以分析**第一次出现和最后一次出现**的日期相等，并且是2天前的用户。此报表依然依赖于每日历史用户分析表。

* 从历史用户表中找出前天出现，并且最后一天也是前天的用户
* 计算出用户数

select count(distinct user\_id) num from mid\_user\_history\_dt  
 where dt='2018-09-30' and first\_dat=current\_dat and first\_dat=date\_add(dt,-2);

### 5.3.9 每日历史用户分析

所有用户的拉链表。存储所有用户的信息，并且每日更新。

* 跟昨天的用户进行比较
* 更新用户的第一次和最后一次出现的时间。所以需要先union再统计
* insert overwrite table mid\_user\_history\_dt   
  partition (dt='2018-09-30')  
  select user\_id,area,app\_version,min(first\_dat) first\_dat,max(last\_dat) last\_dat from (  
  select distinct user\_id,area,dt first\_dat,dt last\_dat from ods\_basedata\_dt where dt='2018-09-30'   
  union  
  select user\_id,area,app\_version,first\_dat,last\_dat from mid\_user\_history\_dt where dt='2018-09-29'   
  )group by user\_id,area;

### 5.4.0 每日历史新闻分析

所有新闻的历史信息，并且保存。每天进行更新。用来分析新闻

* 跟昨天的新闻进行比较，更新新闻的出现时间
* 需要统计新闻的总共点击和展示

create temp view tmp\_today\_news as  
 select t1.\*,t2.total\_click,t2.total\_display from  
 (  
select news\_id,area,min(first\_display\_time) first\_display\_time, max(last\_display\_time) last\_display\_time, min(first\_click\_time) first\_click\_time, max(last\_click\_time) last\_click\_time from (  
select news\_id,area,min(server\_time) first\_display\_time,max(server\_time) last\_display\_time,null first\_click\_time,null last\_click\_time from ods\_display\_dt where dt='2018-09-30' and action='1' group by news\_id,area  
union  
select news\_id,area,null first\_display\_time,null last\_display\_time,min(server\_time) first\_click\_time,max(server\_time) last\_click\_time from ods\_display\_dt where dt='2018-09-30' and action='2' group by news\_id,area  
)  
group by news\_id,area  
) t1 join mid\_daily\_news\_dt t2  
on t1.news\_id =t2.news\_id;  
  
insert overwrite table mid\_news\_history\_dt  
partition (dt='2018-09-30')  
select  
news\_id,area,  
min(first\_display\_time) first\_display\_time,  
max(last\_display\_time) last\_display\_time,  
min(first\_click\_time) first\_click\_time,  
max(last\_click\_time) last\_click\_time,  
sum(total\_display) total\_display,  
sum(total\_click) total\_click  
from  
(  
select \* from tmp\_today\_news  
union all  
select news\_id,area,first\_display\_time,last\_display\_time,first\_click\_time,last\_click\_time,total\_display,total\_click from mid\_news\_history\_dt where dt='2018-09-29'  
) group by news\_id,area;

### 5.4.1 新鲜度分析

用户新鲜度 = 某段时间的新增用户数/某段时间的活跃的用户数

今天新增用户（为 n）

select count(user\_id) from dw\_history\_users where dt='2018-09-26' and current\_dat=dt

今天活跃用户（m）

select count(distinct uid) forground\_total,dt from dw\_foreground where dt='2018-09-25' group by select count(distinct uid) forground\_total,dt from dw\_foreground where dt='2018-09-25' group by dt;

3. 新鲜度 = n / m

注意判断 m 等于 0 的情况

## 5.4 实时系统

### 5.4.1 SparkStreaming 程序启动

其中 driver-memory, executor 数，可以用来调节spark程序

spark-submit \

--class com.tom.spark.SparkNewsCaculator \

--master yarn \

--deploy-mode cluster \

--queue migsz \

--driver-memory 5G \

--executor-memory 10G \

--num-executors 10 \

--executor-cores 4 \

--conf spark.shuffle.io.maxRetries=6

Spark-news.jar

### 5.4.2 实时计算新闻热度

威尔逊函数+阈值，然后排序质量新闻和热度新闻。调用算法函数。最后得出的数据

{"user\_id":"xxxx", "news\_ids":["1","2","3"]},{"user\_id":"xxxx", "news\_ids":["2","3","1"]}

# 第6章 项目总结

## 6.1 面试题

### 6.1.2 项目相关

整个数据流程中，如何保证数据一致性？

从日志到kafka过程，和从kafka到hdfs的过程中，分别用到了kafka的ack事务性和flume的事务性。而从数据事实表到数据产品，中间通过脚本的依赖，来保证数据的一致。

这个信息流是实时性的么，瓶颈在哪里？

准实时。整个时间是从api收集日志到计算完成新闻排序，中间经过了springboot，kafka，然后数据需要写一次RDS，读一次RDS，读一次HDFS最后得到相关的结果。

瓶颈在spark与rds，hdfs的多次交互，读取kafka全量数据。

为什么不用hbase而用RDS做热度统计？

Hbase：吞吐量高，可以直接去重，但聚合函数复杂,所以聚合函数是瓶颈

RDS：吞吐量略低，插入时候要创建索引。但在统计和计算点击率join的时候可以走索引，同时清楚历史数据也可以走索引，速度非常快。



这个系统有什么可以优化的地方？

* RDS可以用elasticsearch代替
* 可以在flume层对日志进行分离。将实时数据和离线数据分开，这样能减少spark读取量
* Flume可以考虑用其他的组件
* 数据仓库流程可视化，引入kettle

在这个系统中遇到的最大的问题是什么？

数据仓库：整个清洗流程是8小时左右。几亿条的数据。然后

* 多个报表的并行，yarn队列的划分能大大提高速度
* Sparksql替代tez能提高效率

你们系统是怎么容灾的呢？

* 我们将原始日志gzip压缩，然后传送到我们自己托管的服务器。这样哪怕亚马逊出现问题，也没有关系。
* 用亚马逊的s3多区域存储。将数据同步存在两个区域

### 6.1.3 技术相关

Java

* Java1.8的垃圾回收机制
* Compares ,equals和 =的区别
* HashMap查询的时间复杂度
* Lock 和synchronized 的区别
* Thread 和 runnable的区别
* Sleep和wait的区别
* Viotile关键字

Spark：（问的最多）

* Shuffle的过程，为什么要shuffle
* Shuffle ,Join的 实现方式(HashJoin ,Sorted-merge,broadcase join):
* Spark stage和job的划分
* Kyto序列化
* Spark sql执行的流程
* Coalesce 和 repartition 的区别
* Spark碰到数据倾斜的办法
* MemoryOverHead对应什么

Hive:

* Clustered by ,sort by ,distribute by 的区别
* 如何分组取前几条（partition by）
* Hive 如何自定义map数？

Kafka

* 解释一下ISR
* 解释一下LEO和Highwater Mark

数据库：

* 解释一下B+ tree和B-tree，各有什么优缺点
* 什么是数据库的三范式

算法：

* 说一下快排的时间复杂度
* 堆排的时间复杂度

数据仓库：

* 说一下拉链表
* 什么是开窗函数

## 6.2 总结

1. 用户行为数据通过http接口上报到web服务器，web服务型将上报的用户行为数据写入本地磁盘，由此实现了业务与数据处理的解耦，之后再由Flume实时监控文件并进行采集。
2. Flume数据采集构架采用双层Flume拓扑架构，第一层实现数据的采集，第二层实现数据的聚合，并通过Flume拓扑结构中的Sinkgroup实现了Load Balance，充分利用了资源。
3. 第一层Flumeagent在每台web服务器上部署一个，当挂掉重启后不能丢失数据，所以我们用了TaildirSource数据源，此数据源会记录每个文件采集到的位置，重启后会从记录的位置采集，但是此数据源有缺陷(当文件重命名后会重复采集)，故我们针对此缺陷进行了源码修改。
4. 第二层聚合Flumeagent需要部署2个及以上，我们的实战项目中有两个聚合agent,其中一个挂掉后，系统可以继续运行，不丢数据，不影响结果。
5. 第二层聚合Flumeagent将数据写入kafka, 当由于某种原因比如网络问题或者kafka停服不能写入时，数据会在第二层agent中的channel中累积，不影响第一层agent的采集，当kafka能够写入时，kafkasink会继续消费第二层angent的channel中的数据。
6. 当第二层聚合Flumeagent全部挂掉时，数据会在第一层的采集agent中的channel中累积，不影响数据采集。
7. Spark Streaming需要7x24小时运行，一旦Driver挂掉后能够自动重启,我们让其在YARN中以Cluster模式运行，此时Driver运行在ApplicationMaster中，当ApplicationMaster挂掉后会重新启动。
8. 为了Driver重启后需要能恢复到之前的状态，所以我们要设置Checkpoint目录，以持久化Kafka offset、未完成的job等到HDFS。
9. 当Spark Streaming的代码修改后启动时，反序列化Checkpoint目录中的数据失败，所以Kafkaoffset会丢失，此时不知道从哪里消费Kafka的数据，所以我们要将Kafka offset保存到ZooKeeper中一份，当Spark Streaming优雅停止后，删除Checkpoint目录然后从ZooKeeper中读取Kafka offset再启动SparkSteaming。