

尚硅谷大数据项目之手机 APP 数据统计分析系统

(作者：章鹏)

官网：www.atguigu.com

版本：V1.0

第一章 项目框架

1.1 项目概述

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，因此，手机 APP 对于各类互联网公司的重要性不言而喻，因此，越来越多的互联网公司开始重视 APP 的研发和升级。

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、沉默用户数、回流用户数等；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；为了了解用户对于这款 APP 的依赖程度，我们需要获取每个用户每天使用这款 APP 的时长。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。



图 1-1 大数据技术框架

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，为 APP 的改进与升级提供了有力的参考依据。

1.2 项目框架

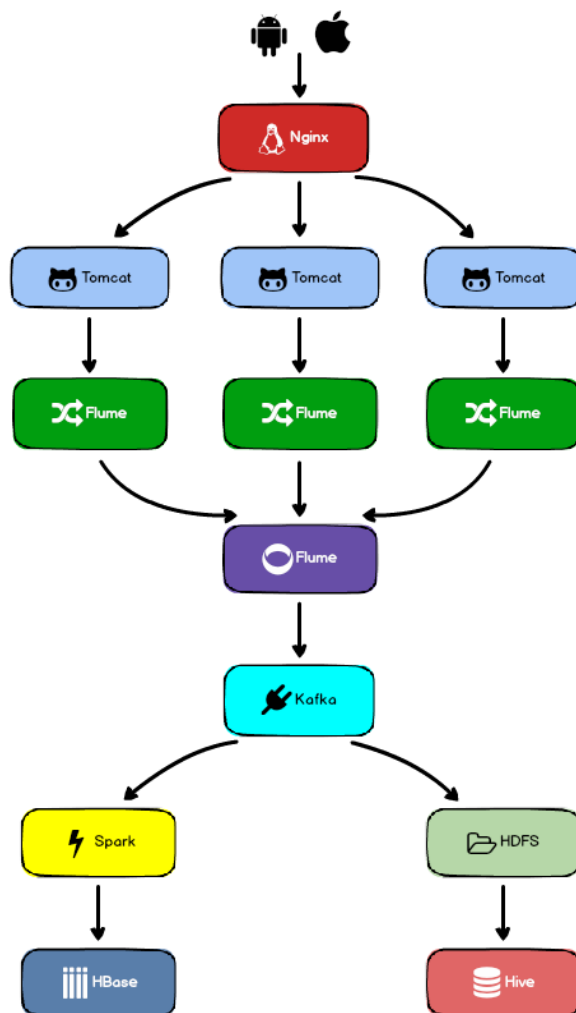


图 1-2 项目框架 1

根据图 1-2 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）向手机 APP 厂商服务器进行发送，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个 Tomcat 上，Tomcat 服务器通过 log4j 将日志数据写入日志文件中，通过日志数据的落盘实现了业务系统与数据采集系统的解耦。

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

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

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。

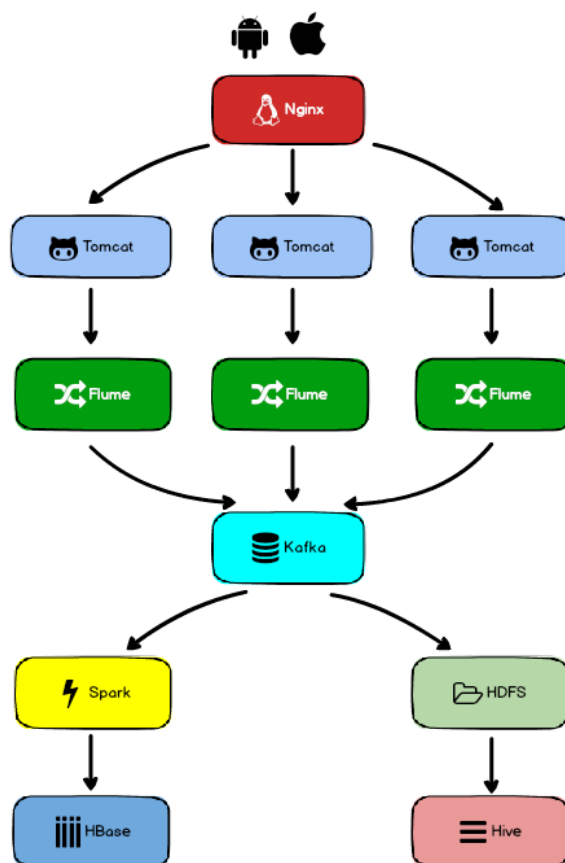


图 1-3 项目框架 2

由于双层 Flume 结构存在大量的网络传输,对于系统的整体性能会造成影响,并且 Kafka 集群宕机的可能行非常低, Flume 的缓冲作用可能无法派上用场,与此同时,多个 Flume 的数据汇总到一台 Flume 上也会导致此聚合 Flume 的数据量过大,因此,我们优先考虑使用单层 Flume 对数据进行采集,并统一发送给 Kafka。

1.2.1 日志上报流程

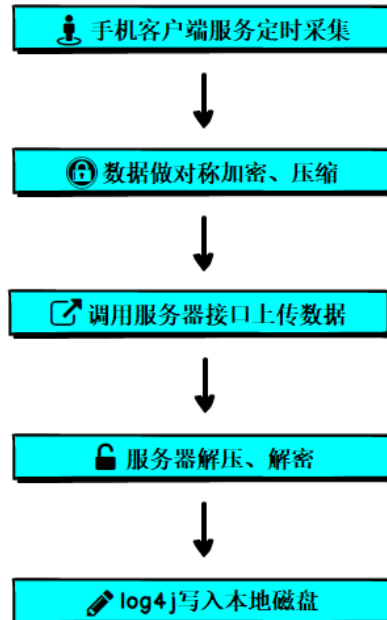


图 1-4 日志上报流程

用户在使用 APP 的过程中，会不断产生日志数据。当用户打开 APP 的时候，会产生 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；当用户浏览不同页面时，会产生页面访问日志，页面访问日志记录了当前页面和上一访问页面，以及页面浏览时长等信息；当用户在使用过程中 APP 发生故障时，会产生错误日志，错误日志记录了错误概要以及详细错误信息。

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 log4j 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）

1.3.2 日志数据概述

本项目中的日志数据由三种类型的日志组成，分别为手机 APP 的启动上报日志（StartupReportLogs）、页面访问日志（PageVisitReportLogs）以及错误上报日志（ErrorReportLogs）组成。

三种日志中共有的数据内容是：userId、userPlatform、appId。

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

表 1-1 日志共有字段表

字段名称	解析
userId	唯一地标识一个 APP 的用户
userPlatform	标识了当前 APP 用户手机的操作系统类型，有 Android 和 IOS 两种类型
appId	唯一地标识公司的一款 APP，如果公司只有一款 APP，那么所有日志中的 appId 均相同

1.3.3 启动上报日志

启动上报日志格式如下：

```
"StartupReportLogs": [  
  {  
    "userId": "user1188",  
    "userPlatform": "android",  
    "appId": "app00001",  
    "appVersion": "1.0.1",  
    "startTimeInMs": 1434534683  
    "activeTimeInMs":157683  
    "city":"Beijing"  
  }  
]
```

启动上报日志中的专有数据项有 4 个，分别为：appVersion、startTimeInMs、activeTimeInMs、city。

表 1-2 启动上报日志字段表

字段名称	解析
appVersion	标识 APP 的版本信息
startTimeInMs	标识启动上报日志的生成时间
activeTimeInMs	标识本次 APP 的使用时长
city	标识 APP 用户所在地区（在实际工业环境中，通过 IP 地址解析用户的详细位置信息，不需要在日志中添加对应的数据项）

1.3.4 页面访问日志

页面访问日志格式如下：

```
"pageVisitReportLogs": [  
  {  
    "userId": "user1188",  
    "userPlatform": "android",  
    "appId": "app00001",  
    "currentPage": "page1.html",  
    "pageVisitIndex": 0,  
    "nextPage": "page2.html",  
    "stayDurationInSec": 60,  
    "createTimeInMs": 1434534683  
  }  
]
```

```
}}]
```

页面访问日志中的专有数据项有 5 个，分别为：currentPage、pageVisitIndex、nextPage、stayDurationInSec、createTimeInMs。

表 1-3 页面访问日志字段表

字段名称	解析
currentPage	标识用户的当前访问页面
pageVisitIndex	标识用户访问页面索引
nextPage	标识用户的下一访问页面
stayDurationInSec	标识用户在页面的停留时间
createTimeInMs	标识日志的产生时间

1.3.5 错误上报日志

```
"errorReportLogs": [
{
  "userId": "user1188",
  "userPlatform": "android",
  "appId": "app00001",
  "errorMajor": "at cn.appstore.appIn.web.AbstractBaseController.validInbound(AbstractBaseController.java:72)",
  "errorInDetail": "java.lang.NullPointerException\n at cn.appstore.appIn.web.AbstractBaseController.validInbound(AbstractBaseController.java:72)\n",
  "createTimeInMs": 1434534683
}]
```

错误上报日志中的专有数据项有 3 个，分别为：errorMajor、errorInDetail、createTimeInMs。

表 1-4 错误日志字段表

字段名称	解析
errorMajor	标识 APP 发生错误的主要信息
errorInDetail	标识 APP 发生错误的详细信息
createTimeInMs	标识日志的产生时间

1.3 项目需求

1.4.1 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。

2. 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。

3. 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。

4. 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。

5. 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。

6. 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。

7. 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。

8. 本周回流用户

上周末启动过应用，本周启动了应用的用户。

9. 连续 n 周活跃用户

连续 n 周，每周至少启动一次。

10. 忠诚用户

连续活跃 5 周以上的用户

11. 连续活跃用户

连续 2 周及以上活跃的用户

12. 近期流失用户

连续 $n(2 \leq n \leq 4)$ 周没有启动应用的用户。

13. 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%。

14. 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。

15. 单次使用时长

每次启动使用的时间长度。

16. 日使用时长

累计一天内的使用时间长度。

1.4.2 离线系统项目需求

表 1-5 离线系统项目需求

离线项目需求	细分项
新增用户统计	日新增用户统计
	周新增用户统计
	月新增用户统计
活跃用户统计	日、周、月活跃用户统计
	指定时间日、周、月活跃用户统计
	过去 N 周活跃用户统计
	过去 N 月活跃用户统计
	连续 N 周活跃用户统计
	忠诚用户统计
沉默用户统计	-
启动次数统计	-
版本分布统计	-
留存分析统计	本周回流用户统计
	连续 N 周末启动用户统计
	留存用户统计
新鲜度分析	-

1.4.3 实时系统项目需求

表 1-6 实时系统项目需求

实时项目需求	细分项
各城市点击次数实时统计	-

1.5 产品选型

1.5.1 Kafka 选型

Spark Streaming + Kafka Integration Guide

Apache Kafka is publish-subscribe messaging rethought as a distributed, partitioned, replicated commit log service. Please read the [Kafka documentation](#) thoroughly before starting an integration using Spark.

The Kafka project introduced a new consumer API between versions 0.8 and 0.10, so there are 2 separate corresponding Spark Streaming packages available. Please choose the correct package for your brokers and desired features; note that the 0.8 integration is compatible with later 0.9 and 0.10 brokers, but the 0.10 integration is not compatible with earlier brokers.

	spark-streaming-kafka-0-8	spark-streaming-kafka-0-10
Broker Version	0.8.2.1 or higher	0.10.0 or higher
API Stability	Stable	Experimental
Language Support	Scala, Java, Python	Scala, Java
Receiver DStream	Yes	No
Direct DStream	Yes	Yes
SSL / TLS Support	No	Yes
Offset Commit API	No	Yes
Dynamic Topic Subscription	No	Yes

图 1-5 Kafka 选型依据（项目经验在 2018 年 3 月前）

由于需要使用 Spark Streaming 对接 Kafka，因此我们需要参照 Spark 官网的说明去选择 Kafka 版本，

2018 年 3 月份之前，由于 Spark Streaming 与 Kafka 0.10 版本尚处于实验阶段，不能保证稳定性，因此，本项目使用 0.8 版本的 Kafka。

Spark Streaming + Kafka Integration Guide

Apache Kafka is publish-subscribe messaging rethought as a distributed, partitioned, replicated commit log service. Please read the [Kafka documentation](#) thoroughly before starting an integration using Spark.

The Kafka project introduced a new consumer API between versions 0.8 and 0.10, so there are 2 separate corresponding Spark Streaming packages available. Please choose the correct package for your brokers and desired features; note that the 0.8 integration is compatible with later 0.9 and 0.10 brokers, but the 0.10 integration is not compatible with earlier brokers.

Note: Kafka 0.8 support is deprecated as of Spark 2.3.0.

	spark-streaming-kafka-0-8	spark-streaming-kafka-0-10
Broker Version	0.8.2.1 or higher	0.10.0 or higher
API Maturity	Deprecated	Stable
Language Support	Scala, Java, Python	Scala, Java
Receiver DStream	Yes	No
Direct DStream	Yes	Yes
SSL / TLS Support	No	Yes
Offset Commit API	No	Yes
Dynamic Topic Subscription	No	Yes

图 1-6 Kafka 选型依据（项目经验在 2018 年 3 月后）

2018 年 3 月份之后，Spark Streaming 稳定支持的 Kafka 版本升级为 0.10 版本，0.8 版本不再推荐使用。

1.5.2 Flume 选型

Kafka Sink

This is a Flume Sink implementation that can publish data to a Kafka topic. One of the objective is to integrate Flume with Kafka so that pull based processing systems can process the data coming through various Flume sources. This currently supports Kafka 0.9.x series of releases.

This version of Flume no longer supports Older Versions (0.8.x) of Kafka.

Required properties are marked in bold font.

图 1-6 Flume 选型依据

如果使用的是 0.8 版本的 Kafka，由于 1.7 版本的 Flume 只支持 0.8 以上版本（不包括 0.8 版本）的 Kafka Sink，因此我们使用 1.6 版本的 Flume。

如果使用的是 0.10 版本的 Kafka，可以直接使用 1.7 及以上版本的 Kafka。

1.5.3 产品选型列表

表 1-7 产品选型列表

名称	版本
Nginx	nginx-1.12.2
Tomcat	apache-tomcat-7.0.72
Hadoop	hadoop-2.7.2
Spark	spark-2.1.1-bin-hadoop2.7
Flume	apache-flume-1.6.0-bin
Kafka	kafka_2.11-0.8.2.1
Hive	apache-hive-1.2.2-bin
Hbase	hbase-1.2.6

第二章 预备知识

2.1 Flume

2.1.1 Flume 组件介绍

表 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

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】



	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
Sink	从 Channel 中读取并移除 Event，将 Event 传递到 Flow Pipeline 中的下一个 Agent 或者其他存储系统

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 提供的 Source 组件，在 1.6 中并没有，因此，需要从 1.7 中移植到 1.6 中。

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（上一跳）Flume 代理的内置 Avro Sink 配对时，它可以创建分层收集拓扑。

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

2) Taildir Source

在 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。

在 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。

2. 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。

2) 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。

3) 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。

3. 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 事件变为 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

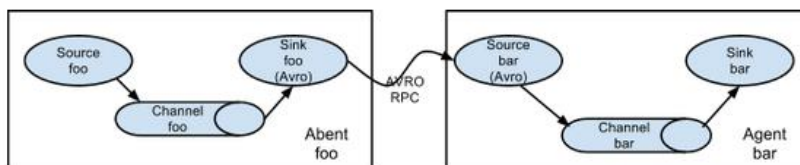


图 2-1 Flume Agent 连接

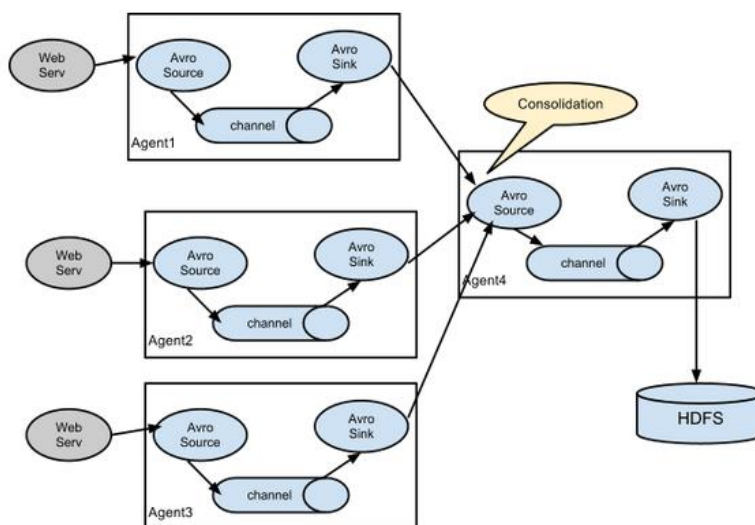


图 2-2 Flume Agent 聚合

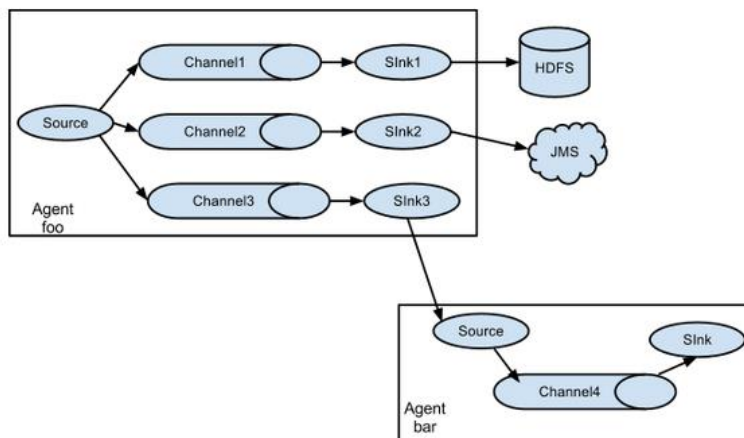


图 2-3 Flume 多路 Flow

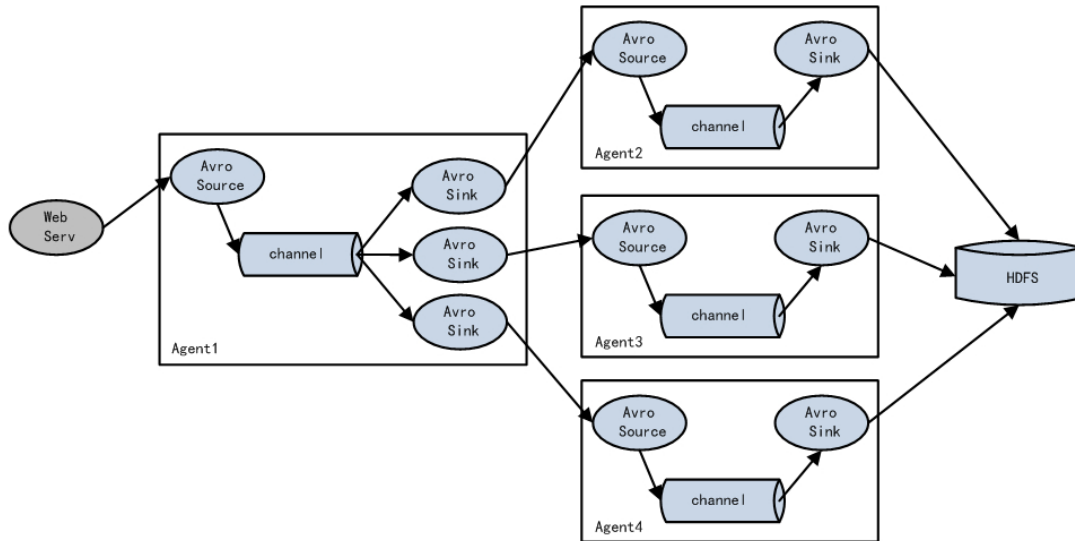


图 2-4 Flume 负载均衡

2.1.5 Flume Source、Channel 处理器、拦截器和 Channel 选择器之间的交互

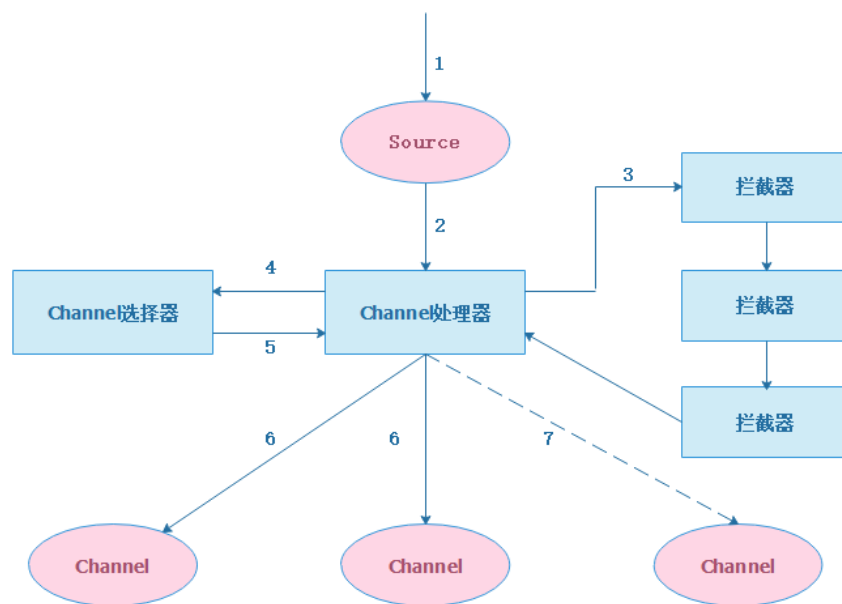


图 2-5 Flume 负载均衡

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；

5. 返回写入事件的 Channel 列表;
6. 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;
7. 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些 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**

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷 (中国) 官网下载区】

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。

2. 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` 并不是两阶段提交（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` 选择器可以指定一组 `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。

`mutiplexing 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

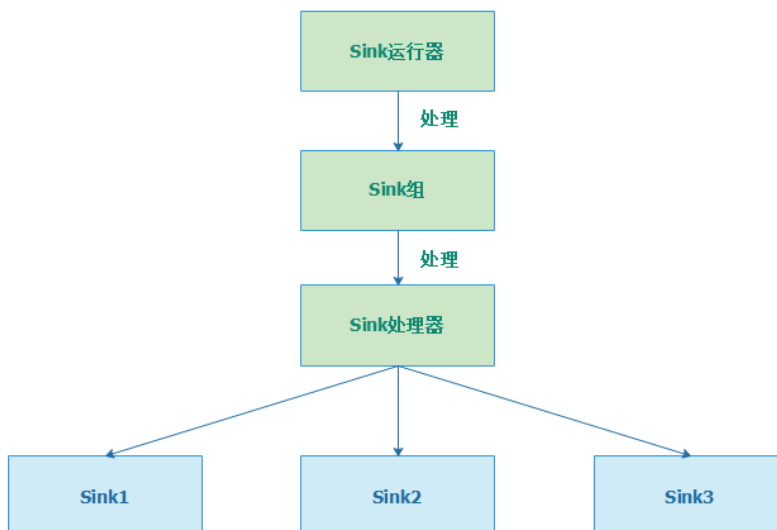


图 2-6 Flume Sink、Sink 运行器、Sink 组合 Sink 处理器

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 组中建立了几十个 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。

2. 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。

在 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。

在 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：

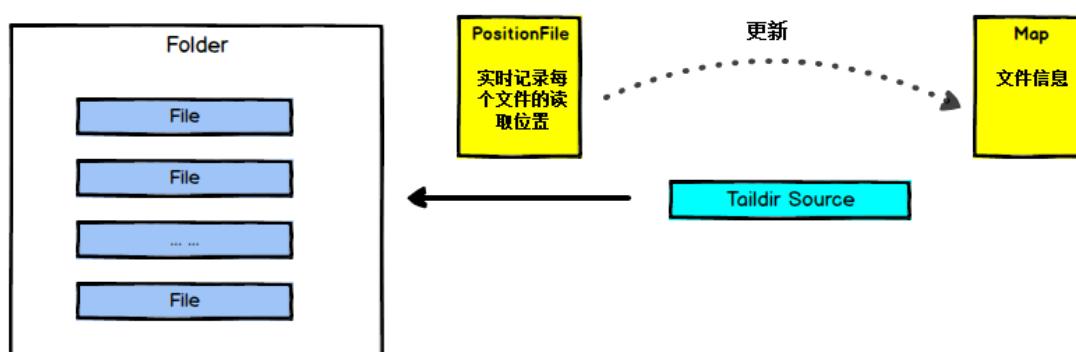
```
private RandomAccessFile raf;
private final String path;
private final long inode;
private long pos;
private long lastUpdated;
private boolean needTail;
private final Map<String, String> headers;
```

```
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。



在 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()

```
/**  
 * Update tailFiles mapping if a new file is created or appends are detected  
 * to the existing file.  
 * 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();
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```
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
        //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
            //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());
                }

                // 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
                if (f.length() < tf.getPos()) {
                    logger.info("Pos " + tf.getPos() + " is larger than file size! "
                        + "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()

```
/**
 * Load a position file which has the last read position of each file.
 * 加载并解析记录了每个文件最新读取位置的 position file
 * If the position file exists, update tailFiles mapping.
 * 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 {
    // 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。

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

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。

2. 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。

3. 生产者参数设置

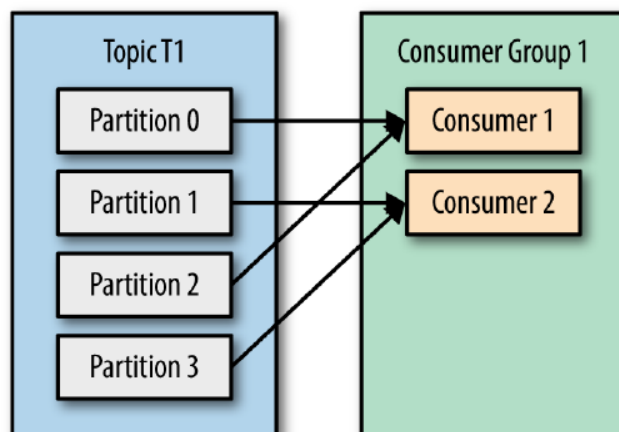
request.required.acks:

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



在 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 的个数除以消费者线程的总数来决定每个消费者线程消费几个分区。**如果除不尽，那么前面几个消费者线程将会多消费一个分区。**

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

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

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) => {
    /*
     * Randomize the order by taking the hashCode to reduce the likelihood of all partitions
     * of a given topic ending
     * 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;

2.2.4 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。

1) 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。

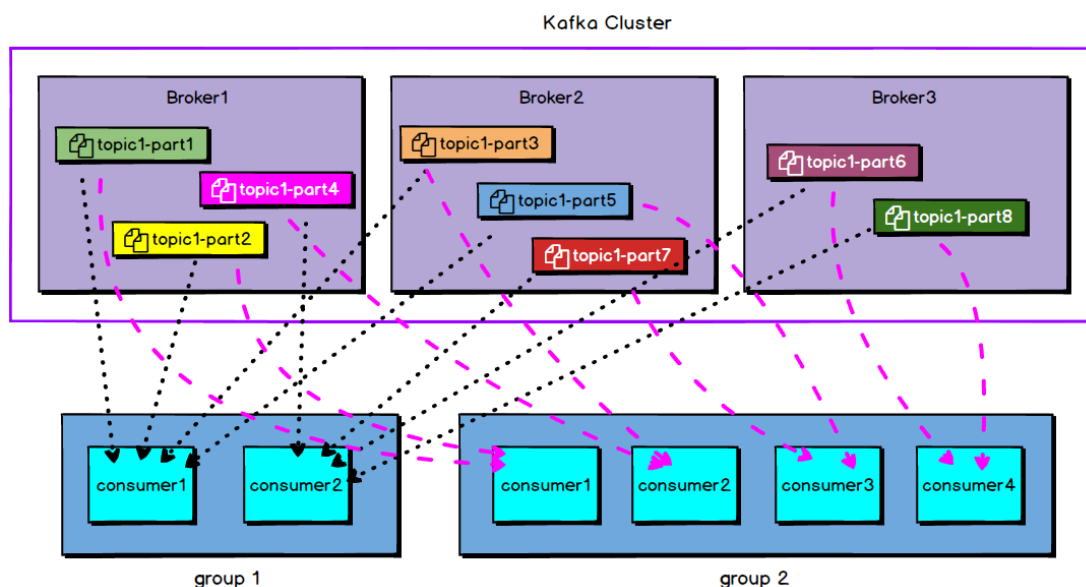


图 2-7 Kafka 高级消费者自动分配分区

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分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。

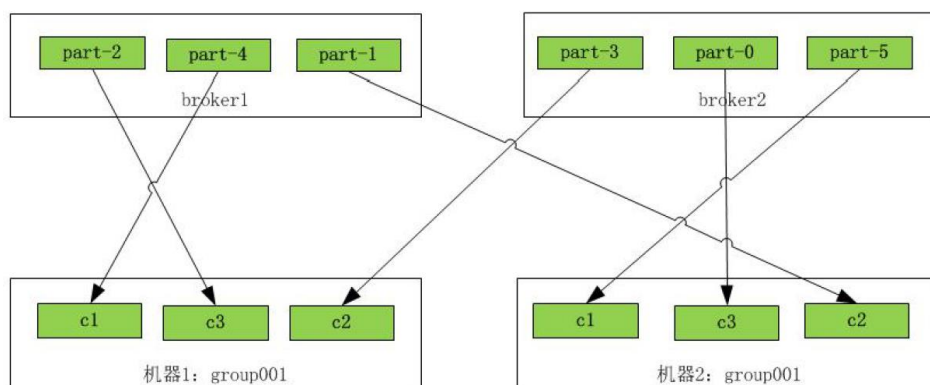


图 2-8 Kafka 高级消费者自动分配分区

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 的 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。

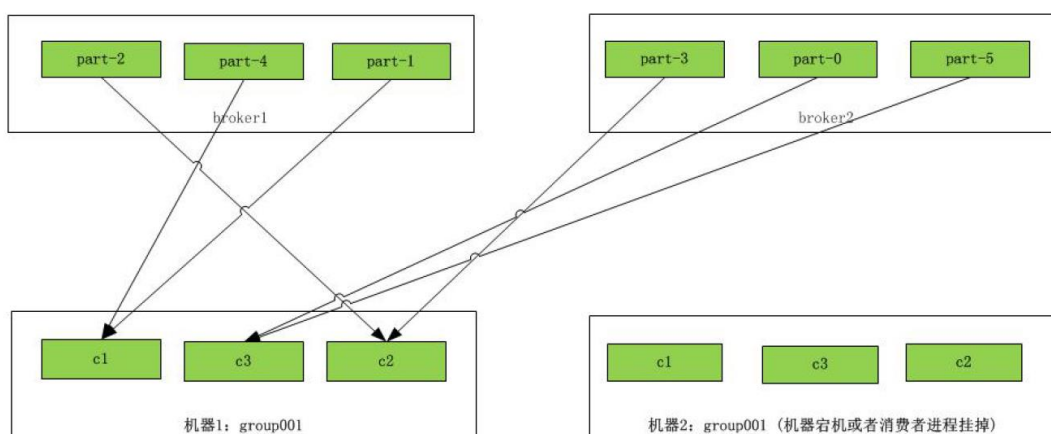


图 2-9 Kafka 高级消费者 rebalance

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。

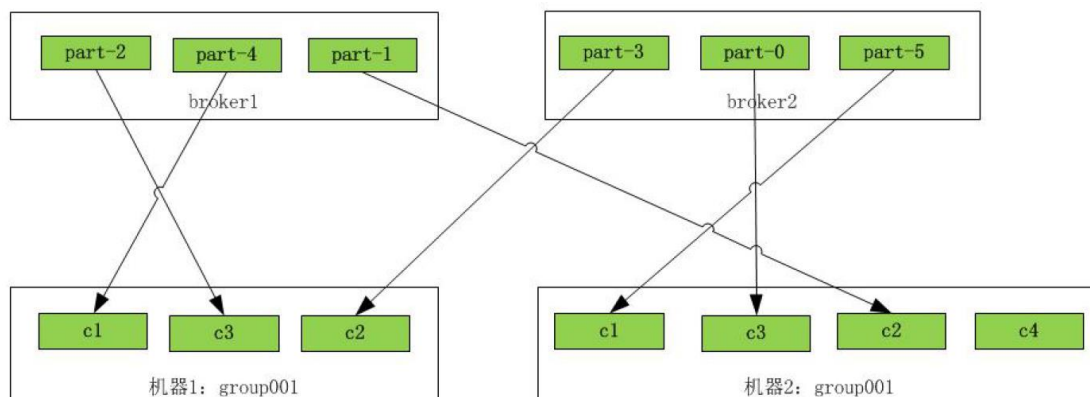


图 2-10 增加消费者线程不能提高并发度

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；

2. 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的 Leader 在哪个 broker 上。

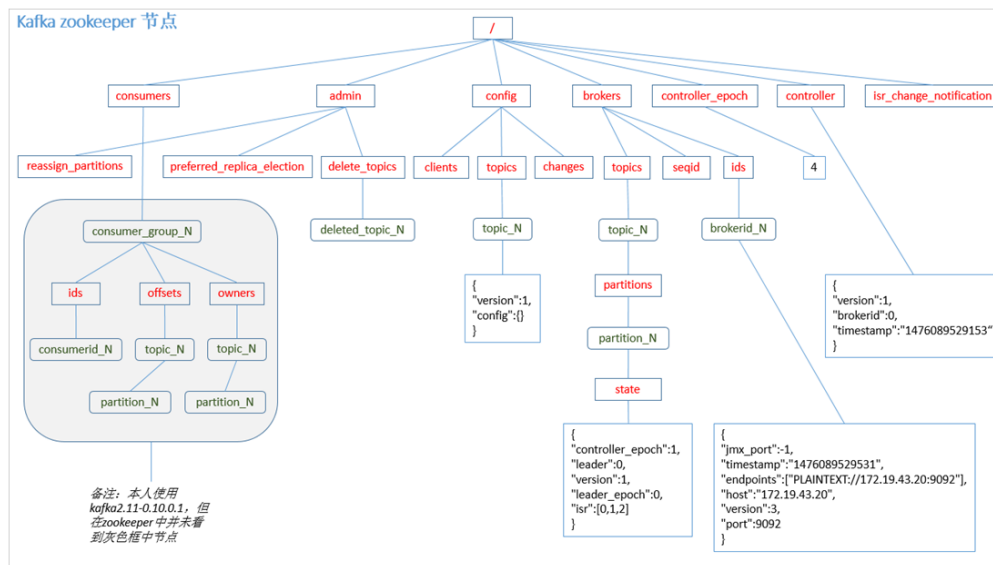


图 2-11 Kafka 在 ZooKeeper 上的布局

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。

2.2.3 Kafka 消费者

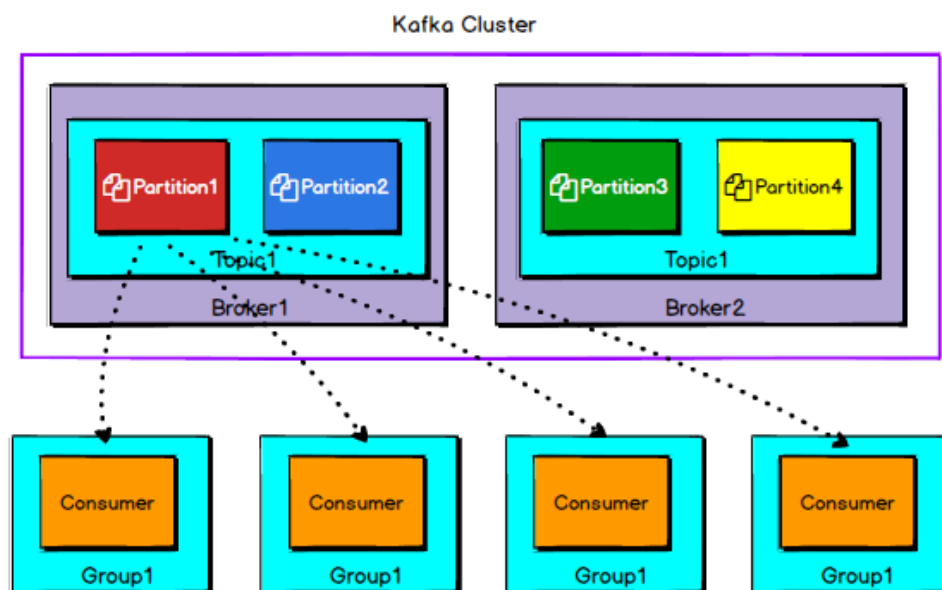
1. 使用消费者组实现消息队列的两种模式

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。

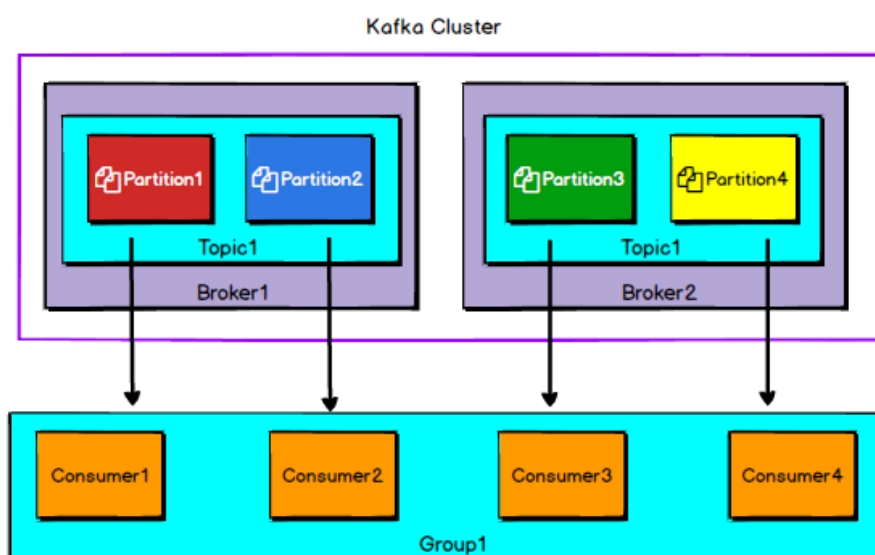
Kafka 采用消费组保证了“一个分区只可被消费组中的一个消费者所消费”，这意味着：

- (1) 在一个消费组中，一个消费者可以消费多个分区。
- (2) 不同的消费者消费的分区分一定不会重复，所有消费者一起消费所有的分区。
- (3) 在不同消费组中，每个消费组都会消费所有的分区。
- (4) 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



队列模式

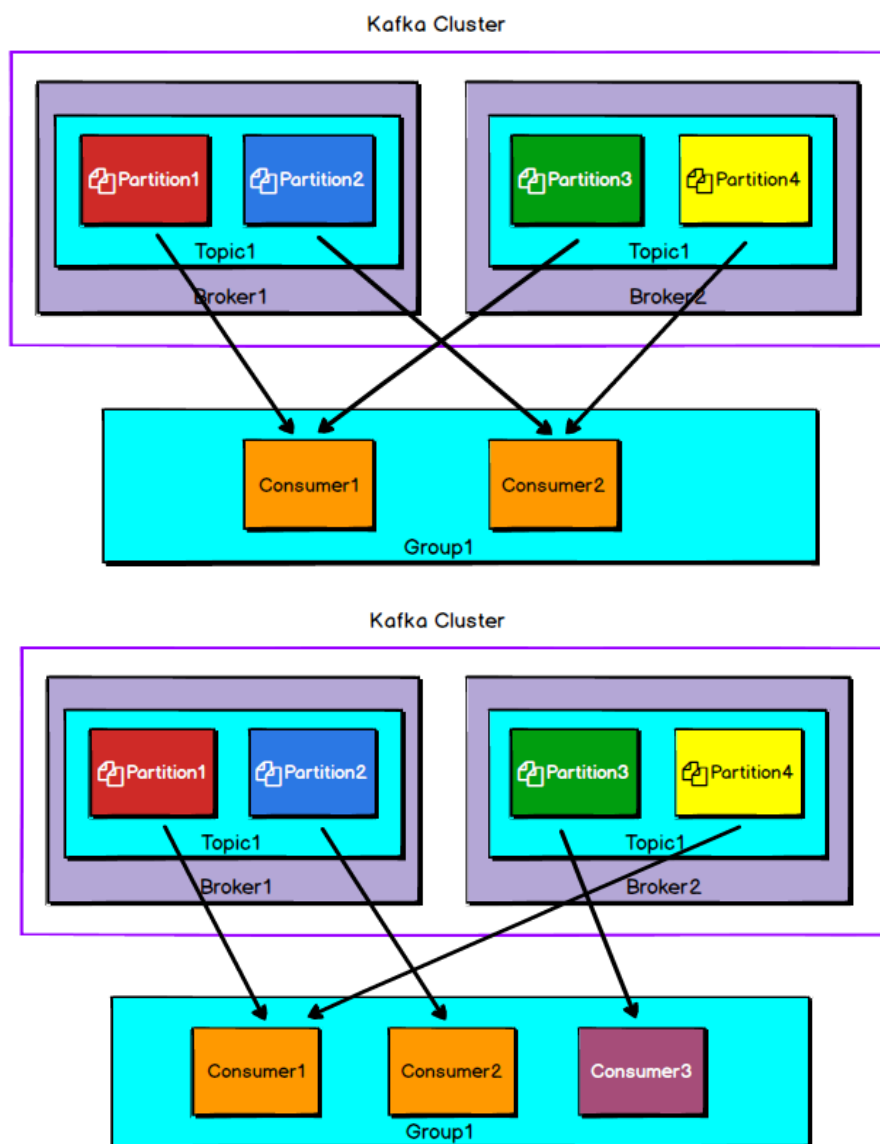
发布-订阅模式：同一条消息会被多个消费组消费，每个消费组只有一个消费者，实现广播。

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。

2. 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。

1. 保存消费进度

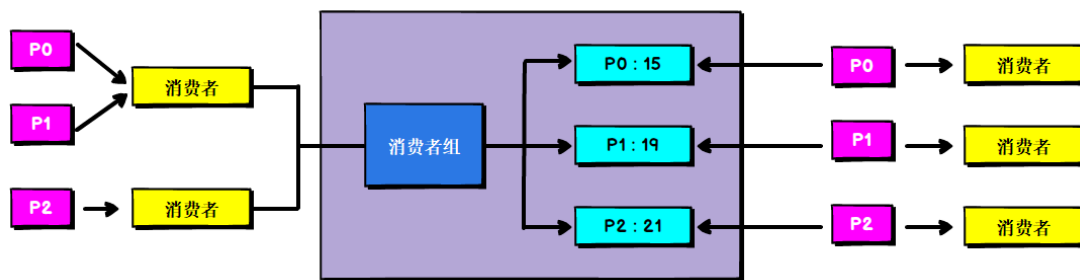
生产者的提交日志采用递增的偏移量，连同消息内容一起写入本地日志文件。生产者客

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

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 1，再平衡后消费者 C2 可以正常地读取消费组 1 的分区 P1 进度，还是可以准确还原出这个分区在消费组 1 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`)。这样分区不同所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。如下图所示，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。

2. 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信息由消费组来决定，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所有者，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信息。

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【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

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。

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:

- **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。
- **发送 (Send)** - 发送消息给经纪人
- **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信息。
- **偏移量 (Offsets)** - 获取有关给定主题分区的可用偏移量的信息。
- **偏移提交 (Offset Commit)** - 为消费者组提供一组偏移量
- **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.factor)。

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
```

【更多 Java、HTML5、Android、python、大数据 资料下载,可访问尚硅谷 (中国) 官网下载区】

在 Kafka 文件存储中，同一个 topic 下有多个不同的 partition，每个 partition 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：

```
00000000000000000000.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：

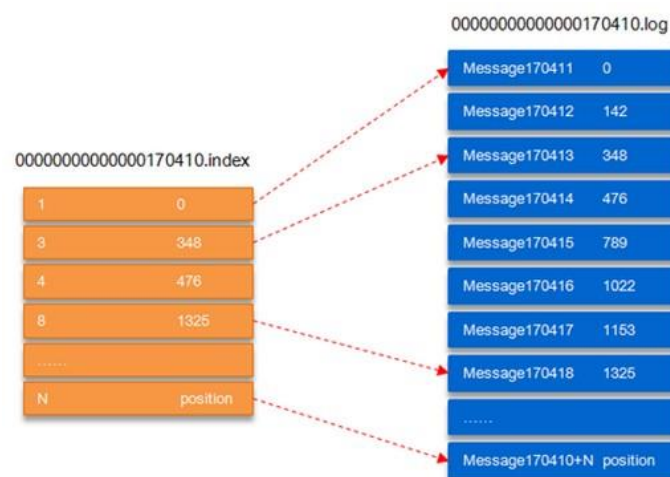


图 2-12 segment 存储机制

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。

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

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。

2. 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。

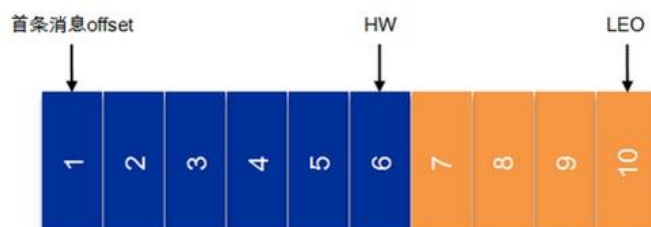


图 2-13 HW 与 LEO

图 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 factor) 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:

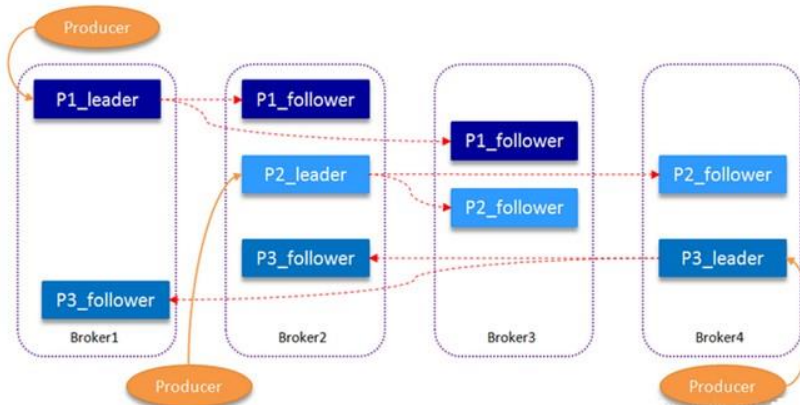


图 2-14 Kafka 副本机制

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，或者说 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。

3. 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 从 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.time.max.messages 表示当前某个副本落后 leader 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.time.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.time.max.messages 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.time.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，【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

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，而 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）

图 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：

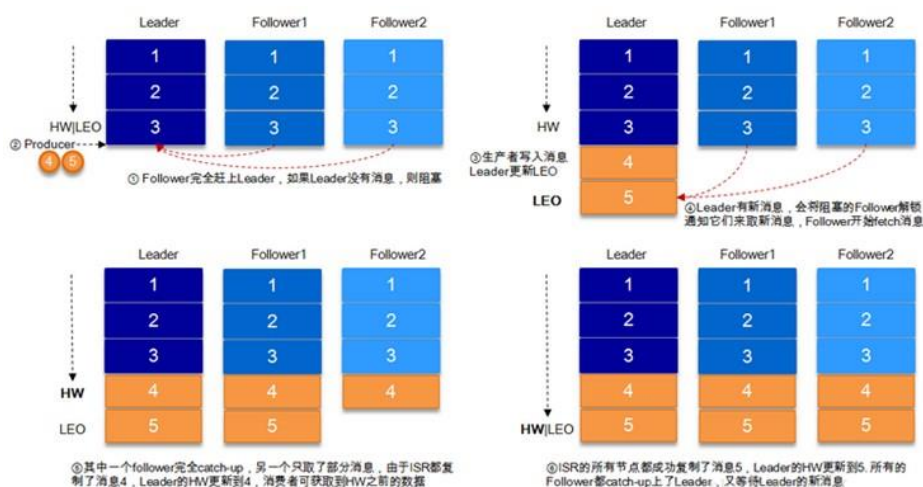


图 2-15 ISR、HW、LEO 流转过程

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：

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

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。

4. 数据可靠性和持久性保证

当 **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.NotEnoughReplicasException: 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。

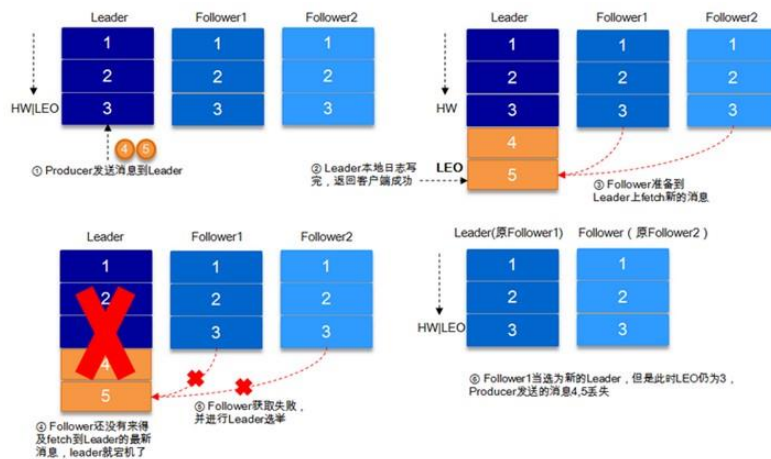


图 2-16 Request.required.acks = 1

2) 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。

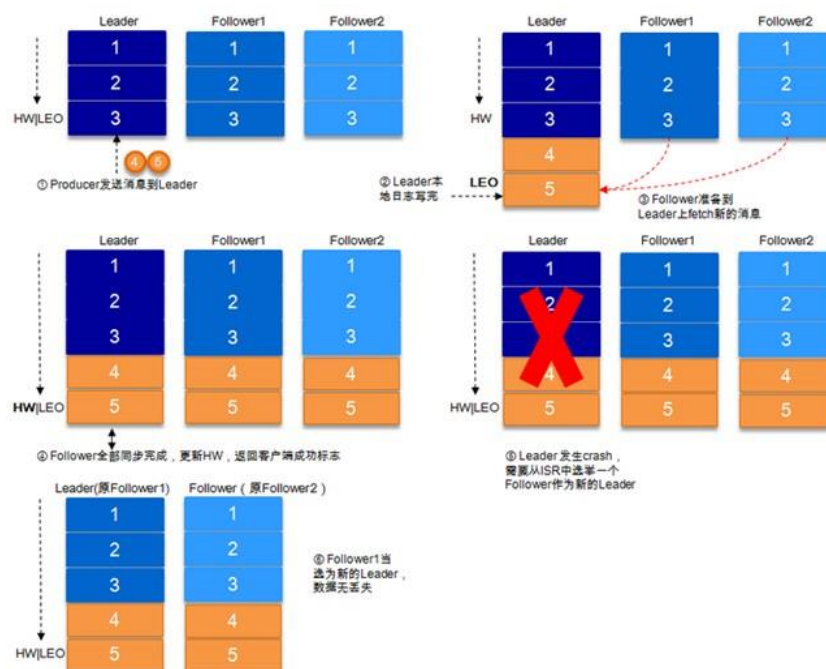


图 2-17 Request.required.acks = -1

`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。比如 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。

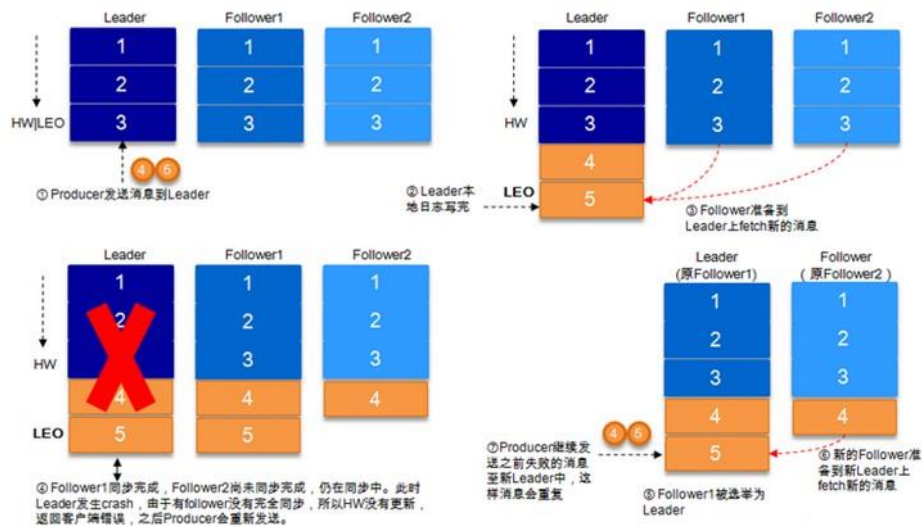


图 2-18 Leader 宕机的情况

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。

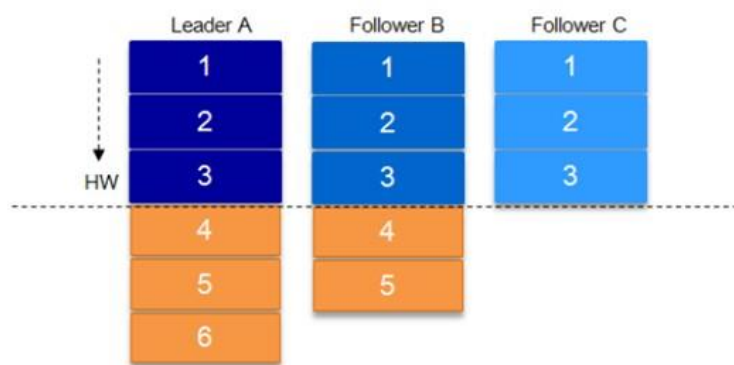


图 2-19 HW 的作用

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 肯定是 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 A 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 B 中拉取消息进行同步。

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

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 时刻的 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

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;
```

- 4) 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) lowname from emp;
```

2.4 Hbase

2.4.1 Hbase 客户端

Hbase 客户端与 Hbase 建立一个连接 (connection)，一般创建一个，因为 connection 是重量级连接，创建表时通过 connection 返回一个 table，table 很轻量，每次写操作时都可以打开一个 table，然后通过这个 table 写数据，但是 table 是一个非线程安全的，多个线程同时写 table 的话会发生线程安全问题。

2.4.2 Hbase 预分区

HBase 的 Region 在数据量达到一定程度时，会进行分裂，分裂时会消耗一定的资源（例如文件拆分后 HDFS 上文件的创建、移动、复本的生成等，数据在集群内的复制，带宽的占用等），此时可以预分区，将不同 RowKey 的数据放在不同的分区中，也就是预先创建 Region，此时在前期就不会发生 Region 的分裂了（当然，预分区后当数据达到一定程度时也会分裂，但是会大幅减少 Region 的分裂频率）。

2.4.3 Hbase 热点问题

Region 是 HBase 中最小的分布式单元，一个 Region 只能存在于一台机器上。

当突然来了一批连续的 RowKey（例如 10 条），那么这批连续的 RowKey 就会进入同一个 Region，由于一个 Region 只能在一台机器上，在很短的时间内就写入了 10 万条数据甚至更多，其他的机器就比较空闲，那么我们说在这段时间内这台机器产生了热点。

那么如何解决热点？

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】



可以给 **RowKey** 的前面加上前缀，例如原始 **RowKey** 的 MD5 值的前三位，这样的话 **RowKey** 的第一个字母就会改变，那么这些原本连续的 **RowKey** 就基本不会被分配到一个 **Region** 中，这些数据就被分散了。查询时，计算出 MD5 值后拼接好就可以进行查询了。但是，虽然解决了热点问题，但是就不能直接进行范围查询了。

还有其他的一些解决思路，例如反序，或者加随机数。

2.5 Scala

2.5.1 Left 与 Right

Scala 中有 **Left**, **Right** 两个类，继承于 **Either**，主要用途是表示两个可能不同的类型（它们之间没有交集），**Left** 主要是表示 **Failure**，即获取失败，**Right** 表示获取成功。

Left 与 **Right** 非常类似于 **None** 与 **Some**，是对于成功/失败的运行结果的一种表示。

```
object EitherLeftRightExample extends App {  
  
  // y 为被除数, x 为除数  
  def divideXByY(x: Int, y: Int): Either[String, Int] = {  
    if (y == 0) Left("Dude, can't divide by 0")  
    else Right(x / y)  
  }  
  
  //Left(Dude, can't divide by 0)  
  println(divideXByY(1, 0))  
  
  //Right(1)  
  println(divideXByY(1, 1))  
  
  //Answer: Dude, can't divide by 0  
  divideXByY(1, 0) match {  
    case Left(s) => println("Answer: " + s)  
    case Right(i) => println("Answer: " + i)  
  }  
  
  //Answer: 1  
  divideXByY(1, 1) match {  
    case Left(s) => println("Answer: " + s)  
    case Right(i) => println("Answer: " + i)  
  }  
}
```

2.6 Spark Core

2.6.1 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：

【更多 **Java**、**HTML5**、**Android**、**python**、**大数据** 资料下载，可访问尚硅谷（中国）官网下载区】

- 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。

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

```
//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);
    }
}
```

2.6.2 Spark 运行模式

当 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。

当 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。当工作在 YARN 模式下时，可以使用 yarn application kill applicationID 杀死指定程序。用这种方式

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

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。

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。

2.7 Spark Streaming

2.7.1 Checkpoint

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 会 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：

- **配置** —— 用于创建该 streaming application 的所有配置；
- **DStream 操作** —— DStream 一系列的操作；
- **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

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

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。

2.7.2 Receiver 与 Direct

1. Receiver

Receiver 是使用 Kafka 的高层次 Consumer API 来实现的。Receiver 每隔一段 batch 时间去 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 100 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 文件中也有 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

【更多 Java、HTML5、Android、python、大数据 资料下载,可访问尚硅谷(中国)官网下载区】

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。

2. 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。

2.7.3 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地点开始处理。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

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);
    // 将 Scala 的 mutable.Map 转化为 immutable.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;
            }
        });

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

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】



```
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();

// 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
// 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()) {
    // 将 Scala 的 Set 结构转换为 Java 的 Set 结构
    Set<TopicAndPartition> topicAndPartitionSet1 = JavaConversions.setAsJavaSet(topic
AndPartitionSet2);

    // 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 {
    // offset 已存在, 获取 Zookeeper 上的 offset
    // 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) {

    // 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());
        // 创建 Map 结构保持 TopicAndPartition 和对应的 offset 数据
        Map<TopicAndPartition, Object> topicAndPartitionObjectMap = new HashMap();
        // 将当前 offsetRange 的 topicAndPartition 信息和 untilOffset 信息写入 Map
        topicAndPartitionObjectMap.put(topicAndPartition, o.untilOffset());

        // 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);  
  
// 将 Scala 的 mutable.Map 转化为 immutable.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)
```

2.8 crontab 任务调度

2.8.1 crontab 常用命令

1. 查看状态

```
service crond status
```

2. 停止状态:

```
service crond stop
```

3. 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
```

6. 删除当前用户所有的 crontab 任务

```
crontab -r
```

2.8.2 编写 crontab 调度

1. 进入编写 crontab 调度

```
crontab -e
```

2. 实现每分钟执行一次

```
* * * * * 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
```

第三章 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
```

2) 解压安装包

```
tar zxvf pcre-8.35.tar.gz
```

3) 进入安装包目录

```
cd pcre-8.35
```

4) 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：

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】



```
yum install -y gcc gcc-c++
```

安装 pcre:

```
make && make install
```

5) 查看 pcre 版本

```
pcgrep -V
```

2. 安装 Nginx

1) 解压安装包

```
tar zxvf nginx-1.12.2.tar.gz
```

2) 进入安装包目录

```
cd nginx-1.12.2
```

3) 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
```

4) 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" '
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```
#          '$status $body bytes sent "$http referer" '
#          '"$http_user_agent" "$http_x_forwarded_for"';

#access log logs/access.log main;

sendfile      on;
#tcp nodelay  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;

    # 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 Tomcat

3.2.1 Tomcat 安装

将 Tomcat 压缩包 apache-tomcat-7.0.72.tar.gz 解压到指定目录即可。

3.2.2 Tomcat 配置

1) Tomcat1 配置

代码清单 3-2 Tomcat1 配置文件

```
<?xml version='1.0' encoding='utf-8'?>

<Server port="18005" shutdown="SHUTDOWN">
  <Listener className="org.apache.catalina.startup.VersionLoggerListener" />
  <Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
  <Listener className="org.apache.catalina.core.JasperListener" />
  <Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener" />
  <Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
  <Listener className="org.apache.catalina.core.ThreadLocalLeakPreventionListener" />

  <Resource name="UserDatabase" auth="Container"
    type="org.apache.catalina.UserDatabase"
    description="User database that can be updated and saved"
    factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
    pathname="conf/tomcat-users.xml" />
</GlobalNamingResources>

<Service name="Catalina">

  <Connector port="18080" protocol="HTTP/1.1"
    connectionTimeout="20000"
    redirectPort="8443" />

  <Connector port="18009" protocol="AJP/1.3" redirectPort="8443" />

  <Engine name="Catalina" defaultHost="localhost">

    <Realm className="org.apache.catalina.realm.LockOutRealm">

      <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
        resourceName="UserDatabase"/>
    </Realm>

    <Host name="localhost" appBase="webapps"
      unpackWARs="true" autoDeploy="true">

      <Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"
        prefix="localhost_access_log." suffix=".txt"
        pattern="%h %l %u %t &quot;%r&quot; %s %b" />
    </Host>
  </Engine>
</Service>
</Server>
```

2) Tomcat2 配置

代码清单 3-3 Tomcat1 配置文件

```
<?xml version='1.0' encoding='utf-8'?>

<Server port="28005" shutdown="SHUTDOWN">
  <Listener className="org.apache.catalina.startup.VersionLoggerListener" />
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】


```
<Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
<Listener className="org.apache.catalina.core.JasperListener" />
<Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener" />
<Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
<Listener className="org.apache.catalina.core.ThreadLocalLeakPreventionListener" />

    <Resource name="UserDatabase" auth="Container"
        type="org.apache.catalina.UserDatabase"
        description="User database that can be updated and saved"
        factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
        pathname="conf/tomcat-users.xml" />
</GlobalNamingResources>

<Service name="Catalina">

    <Connector port="28080" protocol="HTTP/1.1"
        connectionTimeout="20000"
        redirectPort="8443" />

    <Connector port="28009" protocol="AJP/1.3" redirectPort="8443" />

    <Engine name="Catalina" defaultHost="localhost">

        <Realm className="org.apache.catalina.realm.LockOutRealm">

            <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
                resourceName="UserDatabase"/>
        </Realm>

        <Host name="localhost" appBase="webapps"
            unpackWARs="true" autoDeploy="true">

            <Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"
                prefix="localhost_access_log." suffix=".txt"
                pattern="%h %l %u %t &quot;%r&quot; %s %b" />

        </Host>
    </Engine>
</Service>
</Server>
```

3.2.3 Tomcat 部署

将打包为 war 包的 log-analysis.war 放入 Tomcat 安装目录的 webapps 目录下，之后会自动解压，生成 log-analysis 目录，完成部署后需要重启 Tomcat 从而使部署生效。

3.3 Flume

3.3.1 多层 Flume 部署

1. 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 = com.atguigu.flume.source.TaildirSource
a1.sources.r1.channels = c1
a1.sources.r1.positionFile = /opt/modules/flume/checkpoint/behavior/taildir position.js
on
a1.sources.r1.filegroups = f1
a1.sources.r1.filegroups.f1 = /opt/modules/apache-tomcat-7.0.72-1/logs/LogsCollect/atgu
igu.log
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
```

2. 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
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】



```
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 =
hadoop-senior01.itguigu.com:9092,hadoop-senior02.itguigu.com:9092,hadoop-senior03.itguigu.com: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 = com.atguigu.flume.source.TaildirSource
a1.sources.r1.channels = c1
a1.sources.r1.positionFile = /opt/modules/flume/checkpoint/behavior/taildir_position.js
on
a1.sources.r1.filegroups = f1
a1.sources.r1.filegroups.f1 = /opt/modules/apache-tomcat-7.0.72-1/logs/LogsCollect/atgu
igu.log
a1.sources.r1.fileHeader = true

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 =
hadoop-senior01.itguigu.com:9092,hadoop-senior02.itguigu.com:9092,hadoop-senior03.itguigu.com:9092
a1.sinks.k1.requiredAcks = 1
a1.sinks.k1.kafka.producer.type = sync
a1.sinks.k1.batchSize = 1
a1.sinks.k1.channel = c1
```

3.4 Kafka

3.4.1 创建 Kafka 主题

```
kafka-topics.sh --zookeeper hadoop102:2181 --create --replication-factor 3 --partitions 1 --topic topic_app_startup
```

3.5 Hive

3.5.1 配置 Hive 支持 JSON 存储

在 Hive 中采用 Json 作为存储格式,需要建表时指定 Serde。Insert into 时, Hive 使用 json 格式进行保存,查询时,通过 json 库进行解析。Hive 默认输出是压缩格式,这里改成不压缩。

具体操作步骤如下:

- 1) 将 json-serde-1.3.8-jar-with-dependencies.jar 导入到 hive 的/opt/module/hive/lib 路径下。
- 2) 在/opt/module/hive/conf/hive-site.xml 文件中添加如下配置:

代码清单 3-9 Hive支持JSON存储配置

```
<property>
<name>hive.aux.jars.path</name>
<value>file:///opt/module/hive/lib/json-serde-1.3.8-jar-with-dependencies.jar</value>
</property>

<property>
<name>hive.exec.compress.output</name>
<value>>false</value>
</property>
```

3.5.2 Hive 创建数据库及分区表

1. 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;
```

2. 创建数据库

```
hive (default)> create database applogsdb;
```

【更多 Java、HTML5、Android、python、大数据 资料下载,可访问尚硅谷(中国)官网下载区】



使用 applogs_db 数据库:

```
hive (default)> use applogsdb;
```

3. 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;
```

4. 查看数据库中的分区表

```
hive (applogsdb)> show tables;
```

5. 退出 Hive

```
hive (applogsdb)> quit;
```

3.5.3 Hive 执行脚本

代码清单 3-10 Hive执行脚本

```
#!/bin/bash
# 获取三分钟之前的时间
systime=`date -d "-3 minute" +%Y%m-%d-%H%M`
# 获取年月
ym=`echo ${systime} | awk -F '-' '{print $1}'`
# 获取日
day=`echo ${systime} | awk -F '-' '{print $2}'`
# 获取小时分钟
hm=`echo ${systime} | awk -F '-' '{print $3}'`

# 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}')" "
```

3.5.3 crontab 调度脚本

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

3.5.4 自定义 UDF 函数

1. 需求:

根据输入的时间信息, 返回当天的起始时间;

根据输入的时间信息, 返回本周的起始时间;

根据输入的时间信息, 返回本月的起始时间;

根据输入的时间和时间格式化信息, 返回按照格式化要求显示的信息。

3. 工具类实现:

代码清单 3-11 时间工具类

```
public class DateUtil {  
    /**  
     * 得到指定 date 的零时刻。  
     */  
    public static Date getDayBeginTime(Date d) {  
  
        try {  
            // 通过 SimpleDateFormat 获得指定日期的零时刻  
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd 00:00:00");  
  
            return sdf.parse(sdf.format(d));  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
  
        return null;  
    }  
  
    /**  
     * 得到指定 date 的偏移量零时刻。  
     */  
    public static Date getDayBeginTime(Date d, int offset) {  
  
        try {  
            // 通过 SimpleDateFormat 获得指定日期的零时刻  
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd 00:00:00");  
            Date beginDate = sdf.parse(sdf.format(d));  
  
            Calendar c = Calendar.getInstance();  
            c.setTime(beginDate);  
            // 通过 Calendar 实例实现按照天数偏移  
            c.add(Calendar.DAY_OF_MONTH, offset);  
  
            return c.getTime();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
  
        return null;  
    }  
  
    /**  
     * 得到指定 date 所在周的起始时刻。  
     */  
    public static Date getWeekBeginTime(Date d) {  
  
        try {  
            // 得到指定日期 d 的零时刻  
            Date beginDate = getDayBeginTime(d);  
            Calendar c = Calendar.getInstance();  
            c.setTime(beginDate);  
            // 得到当前日期是所在周的第几天  
            int n = c.get(Calendar.DAY_OF_WEEK);  
            // 通过减去周偏移量获得本周的第一天  
            c.add(Calendar.DAY_OF_MONTH, -(n - 1));  

```

```
        return c.getTime();
    } catch (Exception e) {
        e.printStackTrace();
    }

    return null;
}

/**
 * 得到距离指定 date 所在周 offset 周之后的一周的起始时刻。
 */
public static Date getWeekBeginTime(Date d, int offset) {

    try {
        //得到 d 的零时刻
        Date beginDate = getDayBeginTime(d);
        Calendar c = Calendar.getInstance();
        c.setTime(beginDate);
        int n = c.get(Calendar.DAY_OF_WEEK);

        //定位到本周第一天
        c.add(Calendar.DAY_OF_MONTH, -(n - 1));
        //通过 Calendar 实现按周进行偏移
        c.add(Calendar.DAY_OF_MONTH, offset * 7);

        return c.getTime();
    } catch (Exception e) {
        e.printStackTrace();
    }

    return null;
}

/**
 * 得到指定 date 所在月的起始时刻。
 */
public static Date getMonthBeginTime(Date d) {

    try {
        //得到 d 的零时刻
        Date beginDate = getDayBeginTime(d);
        //得到 date 所在月的第一天的零时刻
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/01 00:00:00");

        return sdf.parse(sdf.format(beginDate));
    } catch (Exception e) {
        e.printStackTrace();
    }

    return null;
}

/**
 * 得到距离指定 date 所在月 offset 个月之后的月的起始时刻。
 */
public static Date getMonthBeginTime(Date d, int offset) {

    try {
        //得到 d 的零时刻
        Date beginDate = getDayBeginTime(d);
```



```
//得到 date 所在月的第一天的零时刻
SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/01 00:00:00");

//d 所在月的第一天的零时刻
Date firstDay = sdf.parse(sdf.format(beginDate));

Calendar c = Calendar.getInstance();
c.setTime(firstDay);

//通过 Calendar 实现按月进行偏移
c.add(Calendar.MONTH, offset);

return c.getTime();
} catch (Exception e) {
    e.printStackTrace();
}

return null;
}
}
```

4. 编写 UDF 函数

1) DayBeginUDF

代码清单 3-12 DayBeginUDF

```
public class DayBeginUDF extends UDF {

    // 获取当天的起始时刻 (毫秒数)
    public long evaluate() throws ParseException {
        return evaluate(new Date());
    }

    // 计算距离当天 offset 天之后的起始时间 (毫秒数)
    public long evaluate(int offset) throws ParseException {
        return DateUtil.getDayBeginTime(new Date(), offset).getTime();
    }

    // 计算某天的起始时间 (毫秒数)
    public long evaluate(Date d) throws ParseException {
        return DateUtil.getDayBeginTime(d).getTime();
    }

    // 计算距离某天 offset 天之后的起始时间 (毫秒数)
    public long evaluate(Date d, int offset) throws ParseException {
        return DateUtil.getDayBeginTime(d, offset).getTime();
    }

    // 按照默认格式对输入的 String 类型日期进行解析, 计算某天的起始时间 (毫秒数)
    public long evaluate(String dateStr) throws ParseException {

        SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
        Date d = sdf.parse(dateStr);

        return evaluate(d);
    }

    // 按照默认格式对输入的 String 类型日期进行解析, 计算距离某天 offset 天之后的起始时间 (毫秒数)
    public long evaluate(String dateStr, int offset) throws ParseException {
```

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
Date d = sdf.parse(dateStr);

return DateUtil.getDayBeginTime(d, offset).getTime();
}

// 按照指定的 fmt 格式对输入的 String 类型日期进行解析, 计算某天的起始时间 (毫秒数)
public long evaluate(String dateStr, String fmt) throws ParseException {

    SimpleDateFormat sdf = new SimpleDateFormat(fmt);
    Date d = sdf.parse(dateStr);

    return DateUtil.getDayBeginTime(d).getTime();
}

// 按照指定的 fmt 格式对输入的 String 类型日期进行解析, 计算距离某天 offset 天之后的起始时间 (毫秒数)
public long evaluate(String dateStr, String fmt, int offset) throws ParseException {

    SimpleDateFormat sdf = new SimpleDateFormat(fmt);
    Date d = sdf.parse(dateStr);

    return DateUtil.getDayBeginTime(d, offset).getTime();
}
}
```

2) WeekBeginUDF

代码清单 3-13 WeekBeginUDF

```
public class WeekBeginUDF extends UDF {

    // 计算本周的起始时间 (毫秒数)
    public long evaluate() throws ParseException {
        return DateUtil.getWeekBeginTime(new Date()).getTime();
    }

    // 计算距离本周 offset 周的一周起始时间 (毫秒数)
    public long evaluate(int offset) throws ParseException {
        return DateUtil.getWeekBeginTime(new Date(), offset).getTime();
    }

    // 计算某周的起始时间 (毫秒数)
    public long evaluate(Date d) throws ParseException {
        return DateUtil.getWeekBeginTime(d).getTime();
    }

    // 计算距离指定周 offset 周的一周起始时间 (毫秒数)
    public long evaluate(Date d, int offset) throws ParseException {
        return DateUtil.getWeekBeginTime(d, offset).getTime();
    }

    // 按照默认格式对输入的 String 类型日期进行解析, 计算某周的起始时间 (毫秒数)
    public long evaluate(String dateStr) throws ParseException {

        SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
        Date d = sdf.parse(dateStr);

        return DateUtil.getWeekBeginTime(d).getTime();
    }

    // 按照默认格式对输入的 String 类型日期进行解析, 计算距离某周 offset 周之后的起始时间 (毫秒数)
```

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷(中国)官网下载区】

```
public long evaluate(String dateStr,int offset) throws ParseException {
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
    Date d = sdf.parse(dateStr);
    return DateUtil.getWeekBeginTime(d, offset).getTime();
}

// 按照指定的 fmt 格式对输入的 String 类型日期进行解析, 计算某周的起始时间 (毫秒数)
public long evaluate(String dateStr, String fmt) throws ParseException {

    SimpleDateFormat sdf = new SimpleDateFormat(fmt);
    Date d = sdf.parse(dateStr);

    return DateUtil.getWeekBeginTime(d).getTime();
}

// 按照指定的 fmt 格式对输入的 String 类型日期进行解析,计算距离某周 offset 天之后的起始时间(毫秒数)
public long evaluate(String dateStr, String fmt,int offset) throws ParseException {

    SimpleDateFormat sdf = new SimpleDateFormat(fmt);
    Date d = sdf.parse(dateStr);

    return DateUtil.getWeekBeginTime(d, offset).getTime();
}
}
```

3) MonthBeginUDF

代码清单 3-14 MonthBeginUDF

```
public class MonthBeginUDF extends UDF {

    // 计算本月的起始时间(毫秒数)
    public long evaluate() throws ParseException {
        return DateUtil.getMonthBeginTime(new Date()).getTime();
    }

    // 计算距离本月 offset 个月的月起始时间(毫秒数)
    public long evaluate(int offset) throws ParseException {
        return DateUtil.getMonthBeginTime(new Date(),offset).getTime();
    }

    // 计算某月的起始时间(毫秒数)
    public long evaluate(Date d) throws ParseException {
        return DateUtil.getMonthBeginTime(d).getTime();
    }

    // 计算距离指定月 offset 个月的月起始时间(毫秒数)
    public long evaluate(Date d,int offset) throws ParseException {
        return DateUtil.getMonthBeginTime(d,offset).getTime();
    }

    // 按照默认格式对输入的 String 类型日期进行解析, 计算某月的起始时间 (毫秒数)
    public long evaluate(String dateStr) throws ParseException {

        SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
        Date d = sdf.parse(dateStr);

        return DateUtil.getMonthBeginTime(d).getTime();
    }

    // 按照默认格式对输入的 String 类型日期进行解析, 计算距离某月 offset 个月之后的起始时间 (毫秒数)
}
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】



```
public long evaluate(String dateStr,int offset) throws ParseException {

    SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
    Date d = sdf.parse(dateStr);

    return DateUtil.getMonthBeginTime(d, offset).getTime();
}

// 按照指定的 fmt 格式对输入的 String 类型日期进行解析, 计算某月的起始时间 (毫秒数)
public long evaluate(String dateStr, String fmt) throws ParseException {

    SimpleDateFormat sdf = new SimpleDateFormat(fmt);
    Date d = sdf.parse(dateStr);

    return DateUtil.getMonthBeginTime(d).getTime();
}

// 按照指定的 fmt 格式对输入的 String 类型日期进行解析, 计算距离某月 offset 月之后的起始时间 (毫秒数)
public long evaluate(String dateStr, String fmt,int offset) throws ParseException {

    SimpleDateFormat sdf = new SimpleDateFormat(fmt);
    Date d = sdf.parse(dateStr);

    return DateUtil.getMonthBeginTime(d, offset).getTime();
}
}
```

4) FormatTimeUDF

代码清单 3-15 FormatTimeUDF

```
public class FormatTimeUDF extends UDF{

    // 根据输入的时间毫秒值 (long 类型) 和格式化要求, 返回时间 (String 类型)
    public String evaluate(long ms,String fmt) throws ParseException {

        SimpleDateFormat sdf = new SimpleDateFormat(fmt) ;
        Date d = new Date();
        d.setTime(ms);

        return sdf.format(d) ;
    }

    // 根据输入的时间毫秒值 (String 类型) 和格式化要求, 返回时间 (String 类型)
    public String evaluate(String ms,String fmt) throws ParseException {

        SimpleDateFormat sdf = new SimpleDateFormat(fmt) ;
        Date d = new Date();
        d.setTime(Long.parseLong(ms));

        return sdf.format(d) ;
    }

    // 根据输入的时间毫秒值 (long 类型)、格式化要求, 返回当前周的起始时间 (String 类型)
    public String evaluate(long ms ,String fmt, int week) throws ParseException {

        Date d = new Date();
        d.setTime(ms);

        // 获取周内第一天
        Date firstDay = DateUtil.getWeekBeginTime(d) ;
    }
}
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```
SimpleDateFormat sdf = new SimpleDateFormat(fmt);

return sdf.format(firstDay);
}
}
```

5. 导出 jar 包 (app_logs_hive.jar)

6. 添加 app_logs_hive.jar 到类路径/opt/modules/hive/lib 下

1) 临时添加 jar 包:

```
hive (applogsdb)> add jar /opt/module/hive/lib/app_logs_hive.jar;
```

2) 永久添加 jar 包:

在 hive-site.xml 文件中添加:

代码清单 3-16 hive-site.xml配置

```
<property>
  <name>hive.aux.jars.path</name>
  <value>file:///opt/module/hive/lib/app_logs_hive.jar</value>
</property>
```

由于之前添加过 json 的 jar 包所以修改为如下方式:

代码清单 3-17 hive-site.xml配置

```
<property>
  <name>hive.aux.jars.path</name>
  <value>file:///opt/module/hive/lib/json-serde-1.3.8-jar-with-dependencies.jar,file:///opt/module/hive/lib/app_logs_hive.jar</value>
</property>
```

1. 注册永久函数

```
hive (default)>create function getdaybegin AS 'com.atguigu.hive.DayBeginUDF';
hive (default)>create function getweekbegin AS 'com.atguigu.hive.WeekBeginUDF';
hive (default)>create function getmonthbegin AS 'com.atguigu.hive.MonthBeginUDF';
hive (default)>create function formattime AS 'com.atguigu.hive.FormatTimeUDF';
```

2. 验证函数

登录 mysql

```
$ mysql -uroot -p000000
mysql> show databases;
mysql> use metastore;
mysql> show tables;
mysql> select * from FUNCS;
```

3. 删除函数

```
hive (applogsdb)> drop function getdaybegin;
hive (applogsdb)> drop function getweekbegin;
hive (applogsdb)> drop function getmonthbegin;
hive (applogsdb)> drop function formattime;
```

注意: 在哪个数据库中注册的永久函数, 必须在哪个数据库下将该方法删除。

比如在 applogsdb 数据库中创建的方法, 必须在该数据中调用 drop 方法才能实现删除功能。

3.6 HBase

3.6.1 Hbase 建表

1. 各城市用户数量统计表

hbase 结构设计:

表名: online_city_click_count

key 设计: [city_name]

列族: StatisticData

列: clickCount

列值: 用户点击次数

建表语句: create 'online_city_click_count','StatisticData'

表 1-11 各城市用户数量统计表

RowKey	StatisticData
	clickCount
Beijing	1800
Shanghai	1500
...	...
Xinjiang	230

第四章 代码解析

4.1 离线数据处理系统

4.1.1 Kafka 高级消费者程序

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, 随后, 通过 crontab 周期性地将 HDFS 中存储的日志文件加载到 Hive 数据仓库中。

代码清单 4-1 离线系统Kafka高级消费者程序

```
public class KafkaConsumer {  
  
    private static Configuration configuration = new Configuration();  
    private static FileSystem fs = null;  
    private static FSDataOutputStream outputStream = null;
```

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷(中国)官网下载区】



```
private static Path writePath = null;
private static String hdfsBasicPath = "hdfs://hadoop-senior01.itguigu.com:8020/user/atguigu/test/";

public static void main(String[] args) {
    // 创建配置
    Properties properties = new Properties();
    properties.put("zookeeper.connect", "192.168.10.200:2181");
    properties.put("group.id", "group1");
    properties.put("zookeeper.session.timeout.ms", "1000");
    properties.put("zookeeper.sync.time.ms", "250");
    properties.put("auto.commit.interval.ms", "1000");

    // 创建消费者连接器
    // 消费者客户端会通过消费者连接器（ConsumerConnector）连接 ZK 集群
    // 获取分配的分区，创建每个分区对应的消息流（MessageStream），最后迭代消息流，读取每条消息
    ConsumerConnector consumer = Consumer.createJavaConsumerConnector(new ConsumerConfig(properties));

    try {
        // 获取 HDFS 文件系统
        fs = FileSystem.get(new URI("hdfs://hadoop-senior01.itguigu.com:8020"), configuration, "crazyjack");
    } catch (Exception e) {
        e.printStackTrace();
    }

    // 创建 HashMap 结构，用于指定消费者订阅的主题和每个主题需要的消费者线程数
    // Key: 主题名称    Value: 消费线程个数
    // 一个消费者可以设置多个消费者线程，一个分区只会被分配给一个消费者线程
    HashMap<String, Integer> topicCount = new HashMap<>();

    // 消费者采用多线程访问的分区都是隔离的，所以不会出现一个分区被不同线程同时访问的情况
    // 在上述线程模型下，消费者连接器负责处理分区分配和拉取消息

    // 每一个 Topic 至少需要创建一个 Consumer thread
    // 如果有多个 Partitions，则可以创建多个 Consumer thread 线程
    // Consumer thread 数量 > Partitions 数量，会有 Consumer thread 空闲

    // 设置每个主题的线程数量
    // 设置 analysis-test 的线程数量为 1
    topicCount.put("analysis-test", 1);

    // 每个消费者线程都对应了一个消息流
    Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer.createMessageStreams(topicCount);

    // 消费者迭代器，从消息流中取出数据
    // consumerMap.get("analysis-test") 获取 analysis-test 主题的所有数据流
    // 由于只有一个线程，只有一个消息流，因此 get(0) 获取这个唯一的消息流
    KafkaStream<byte[], byte[]> stream = consumerMap.get("analysis-test").get(0);

    // 获取 MessageStream 中的数据迭代器
    ConsumerIterator<byte[], byte[]> it = stream.iterator();

    // 获取当前时间
    Long lastTime = System.currentTimeMillis();

    // 获取数据写入全路径
    String totalPath = getTotalPath(lastTime);
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```
// 根据路径创建 Path 对象
writePath = new Path(totalPath);

// 创建文件流
try{
    if (fs.exists(writePath)) {
        outputStream = fs.append(writePath);
    } else {
        outputStream = fs.create(writePath, true);
    }
} catch (Exception e){
    e.printStackTrace();
}

while (it.hasNext()) {
    // 收集两分钟的数据后更换目录
    if (System.currentTimeMillis() - lastTime > 6000) {
        try{
            outputStream.close();
            Long currentTime = System.currentTimeMillis();
            String newPath = getTotalPath(currentTime);
            writePath = new Path(newPath);
            outputStream = fs.create(writePath);
            lastTime = currentTime;
        } catch (Exception e){
            e.printStackTrace();
        }
    }

    String jsonStr = new String(it.next().message());

    if (jsonStr.contains("appVersion")) {
        System.out.println("startupPage");
        save(jsonStr);
    } else if (jsonStr.contains("currentPage")) {
        System.out.println("PageLog");
    } else {
        System.out.println("ErrorLog");
    }
}

private static void save(String log) {
    try {
        // 将日志内容写入 HDFS 文件系统
        String logEnd = log + "\r\n";
        outputStream.write(logEnd.getBytes());
        // 一致性模型
        outputStream.hflush();
        outputStream.hsync();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private static String timeTransform(Long timeInMills){
    Date time = new Date(timeInMills);
    String formatDate = "";
    try {
```

```
        SimpleDateFormat sdf = new SimpleDateFormat("yyyyMM-dd-HH:mm");
        formatDate = sdf.format(time);
    } catch (Exception e) {
        e.printStackTrace();
    }

    return formatDate;
}

private static String getDirectoryFromDate(String date) {
    String[] directories = date.split("-");
    String directory = directories[0] + "/" + directories[1];
    return directory;
}

private static String getFileName(String date) {
    String[] dateSplit = date.split("-");
    String fileName = dateSplit[2];
    return fileName;
}

private static String getTotalPath(Long lastTime) {
    // 时间格式转换
    String formatDate = timeTransform(lastTime);
    // 提取目录
    String directory = getDirectoryFromDate(formatDate);
    // 提取文件名称
    String fileName = getFileName(formatDate);
    // 全路径
    String totalPath = hdfsBasicPath + directory + "/" + fileName;

    return totalPath;
}
}
```

4.2 实时数据处理系统

4.2.1 SparkStreaming 程序

实时数据处理系统中的 SparkStreaming 使用了 Kafka 低级消费者对 Kafka 中的日志数据进行消费，并通过 CheckPoint 机制和 ZooKeeper 保存机制对 Kafka 中主题各个分区的 offset 进行手动保存，通过这两种机制的融合实现了 offset 的可靠性存储。

代码清单 4-2 main函数

```
public static void main(String[] args) throws Exception {

    final Properties serverProps = PropertiesUtil.getProperties("spark-streaming/src/main/resources/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(check
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```
pointPath, createContext(serverProps));

    javaStreamingContext.start();

    javaStreamingContext.awaitTermination();
}
```

代码清单 4-3 createContext函数

```
/**
 * 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>() {
        @Override
        public JavaStreamingContext call() throws Exception {

            //获取配置中的 topic
            String topicStr = serverProps.getProperty("kafka.topic");
            Set<String> topicSet = new HashSet(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, String> kafkaParams = new HashMap();
            kafkaParams.put("metadata.broker.list", kafkaBrokerList);
            kafkaParams.put("group.id", groupId);

            //从 zookeeper 中获取每个分区的消费到的 offset 位置
            final KafkaCluster kafkaCluster = getKafkaCluster(kafkaParams);
            Map<TopicAndPartition, Long> consumerOffsetsLong = getConsumerOffsets(kafkaCluster, groupId, topicSet);
            printZkOffset(consumerOffsetsLong);

            // 创建 SparkConf 对象
            SparkConf sparkConf = new SparkConf().setMaster("local[*]").setAppName("online");

            // 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
        }
    };
}
```

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷(中国)官网下载区】



```
sparkConf.set("spark.serializer","org.apache.spark.serializer.KryoSerializer");
//增加 MyRegistrar 类，注册需要用 Kryo 序列化的类
sparkConf.set("spark.kryo.registrator", "com.atguigu.registrator.MyKryoRegistrar");

// 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 5s 执行一次
JavaStreamingContext javaStreamingContext = new JavaStreamingContext(sparkConf, Durations.seconds(streamingInterval));
javaStreamingContext.checkpoint(checkpointPath);

//创建 kafka DStream
JavaInputDStream<String> kafkaMessage = KafkaUtils.createDirectStream(
    javaStreamingContext,
    String.class,
    String.class,
    StringDecoder.class,
    StringDecoder.class,
    String.class,
    kafkaParams,
    consumerOffsetsLong,
    new Function<MessageAndMetadata<String, String>, String>() {
        public String call(MessageAndMetadata<String, String> vl) throws Exception{
            return vl.message();
        }
    });

//需要把每个批次的 offset 保存到此变量
final AtomicReference<OffsetRange[]> offsetRanges = new AtomicReference();

JavaDStream<String> kafkaMessageDStreamTransform = kafkaMessage.transform(new
Function<JavaRDD<String>, JavaRDD<String>>() {
    public JavaRDD<String> call(JavaRDD<String> rdd) throws Exception {
        // 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[] offsets = ((HasOffsetRanges) rdd.rdd()).offsetRanges();
        offsetRanges.set(offsets);
        for (OffsetRange o : offsetRanges.get()) {
            log.info("rddoffsetRange:=====
=====");
            log.info("rddoffsetRange:topic=" + o.topic()
                + ",partition=" + o.partition()
                + ",fromOffset=" + o.fromOffset()
                + ",untilOffset=" + o.untilOffset()
                + ",rddpartitions=" + rdd.getNumPartitions()
                + ",isempty=" + rdd.isEmpty()
                + ",id=" + rdd.id());
        }
        return rdd;
    }
});

//将 kafka 中的消息转换成对象并过滤不合法的消息
JavaDStream<String> kafkaMessageFilter = kafkaMessageDStreamTransform.filter
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】



```
(new Function<String, Boolean>() {
    @Override
    public Boolean call(String message) throws Exception {
        try {
            // 第一步：将 topic 中不合法的 message 全部滤除
            if(!message.contains("activeTimeInMs") && !message.contains("stayDurationInSec") && !message.contains("errorMajor")){
                return false;
            }

            // 由于只对 startup 日志进行处理，因此去除其他的日志
            if(!message.contains("activeTimeInMs") || !message.contains("appVersion")){
                return false;
            }

            // 第二步：完成 JSON 到对象的转换
            StartupReportLogs requestModel = JSONUtil.json2Object(message, StartupReportLogs.class);

            // 第三步：滤除不合法的 Log 对象
            if (requestModel == null ||
                MyStringUtil.isEmpty(requestModel.getUserId()) ||
                MyStringUtil.isEmpty(requestModel.getAppId())) {
                return false;
            }
            return true;
        } catch (Exception e) {
            log.error("find illegal message", e);
            return false;
        }
    }
});

//将每条用户行为转换成键值对，建是我们自定义的 key，值是使用应用的时长，并统计时长
JavaPairDStream<UserHourPackageKey, Long> kafkaStatic = kafkaMessageFilter.
mapToPair(new PairFunction<String, UserHourPackageKey, Long>() {
    @Override
    public Tuple2<UserHourPackageKey, Long> call(String s) throws Exception {
        StartupReportLogs requestModel = null;
        try {
            //将 JSON 字符串数据转换为 StartupReportLogs.class 对象
            requestModel = JSONUtil.json2Object(s, StartupReportLogs.class);
        } catch (Exception e) {
            e.printStackTrace();
        }

        //以 userId、Hour (yyyyMMddHH)、PackageName 组合为 key
        UserHourPackageKey key = new UserHourPackageKey();
        String userId = requestModel.getUserId();
        key.setUserId(Long.valueOf(userId.substring(4, userId.length())));
        Long startTime = requestModel.getStartTimeInMs();
        key.setHour(DateUtils.getDateStringByMillisecond(DateUtils.HOUR_FORMAT, startTime));
        key.setPackageName(requestModel.getAppId());
        key.setCity(requestModel.getCity());

        //以 Package 活跃时间为 value
        Tuple2<UserHourPackageKey, Long> t = new Tuple2<UserHourPackageKey, Long>(key, requestModel.getActiveTimeInMs() / 1000);
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】


```
        return t;
    }
    //将一个批次中的同一 Package 的 activeTime 聚合到一起
    }).reduceByKey(new Function2<Long, Long, Long>() {
        @Override
        public Long call(Long v1, Long v2) throws Exception {
            return v1 + v2;
        }
    });

    kafkaStatic.print();

    //将每个用户的统计时长写入 hbase
    kafkaStatic.foreachRDD(new VoidFunction<JavaPairRDD<UserHourPackageKey, Long>>() {
        @Override
        public void call(JavaPairRDD<UserHourPackageKey, Long> rdd) throws Exception {
            rdd.foreachPartition(new VoidFunction<Iterator<Tuple2<UserHourPackageKey, Long>>>() {
                @Override
                public void call(Iterator<Tuple2<UserHourPackageKey, Long>> it) throws Exception {
                    BehaviorStatService service = BehaviorStatService.getInstance(serverProps);

                    while (it.hasNext()) {
                        Tuple2<UserHourPackageKey, Long> t = it.next();
                        UserHourPackageKey key = t._1();

                        //创建用户行为统计模型
                        UserBehaviorStatModel model = new UserBehaviorStatModel();
                        //根据 key 中的数据填充统计模型
                        model.setUserId(MyStringUtil.getFixedLengthStr(key.getUserId() + "",
10));

                        model.setHour(key.getHour());
                        model.setPackageName(key.getPackageName());
                        model.setTimeLen(t._2());
                        model.setCity(key.getCity());

                        //根据统计模型中的数据，对 HBase 表中的数据进行更新
                        service.addUserNumOfCity(model);
                    }
                }
            });

            //kafka offset 写入 zk
            offsetToZk(kafkaCluster, offsetRanges, groupId);
        }
    });
    return javaStreamingContext;
};
return createContextFunc;
}
```

代码清单 4-4 getKafkaCluster函数

```
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);
    // 将 Scala mutable Map 结构转化为 Scala immutable Map
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】



```
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;  
        }  
    });  
// 传入 Scala immutable Map 类型的 Kafka 参数创建 KafkaCluster  
// 返回创建完成的 KafkaCluster  
return new KafkaCluster(scalaKafkaParam);  
}
```

代码清单 4-5 offsetToZk函数

```
public static void offsetToZk(final KafkaCluster kafkaCluster,  
    final AtomicReference<OffsetRange[]> offsetRanges,  
    final String groupId) {  
    for (OffsetRange o : offsetRanges.get()) {  
        // 创建存储 topic 与 partition 的数据结构  
        TopicAndPartition topicAndPartition = new TopicAndPartition(o.topic(), o.partition());  
        // 创建 Map 结构  
        Map<TopicAndPartition, Object> topicAndPartitionObjectMap = new HashMap();  
        // 将存储 topic 与 partition 的数据结构 TopicAndPartition 作为 key, 将 offset 作为 value  
        // key 指明了 topic 和 partition, value 指明了对应的 offset  
        topicAndPartitionObjectMap.put(topicAndPartition, o.untilOffset());  
  
        // 将 java 的 HashMap 转化为 Scala 的 mutable Map  
        scala.collection.mutable.Map<TopicAndPartition, Object> testMap = JavaConversions.map  
        AsScalaMap (topicAndPartitionObjectMap);  
  
        // 将 Scala mutable Map 结构转化为 Scala immutable Map  
        scala.collection.immutable.Map<TopicAndPartition, Object> scalatopicAndPartitionObj  
        ctMap = testMap.toMap(new Predef.$less$colon$less<Tuple2<TopicAndPartition, Object>, Tup  
        le2<Topic AndPartition, Object>>()) {  
            public Tuple2<TopicAndPartition, Object> apply(Tuple2<TopicAndPartition, Object> v1) {  
                return v1;  
            }  
        });  
  
        // 更新 offset 到 kafkaCluster  
        kafkaCluster.setConsumerOffsets(groupId, scalatopicAndPartitionObjectMap);  
    }  
}
```

代码清单 4-6 getConsumerOffsets函数

```
public static Map<TopicAndPartition, Long> getConsumerOffsets(KafkaCluster kafkaCluster,  
    String groupId, Set<String> topicSet)  
{  
    // 将 java 的 Set 类型转换为 Scala 的 Set 类型  
    scala.collection.mutable.Set<String> mutableTopics = JavaConversions.asScalaSet(top  
    icSet);  
    // 将 mutable 类型转换为 immutable 类型  
    scala.collection.immutable.Set<String> immutableTopics = mutableTopics.toSet();  
    // 根据 topic 名称通过 kafkaCluster.getPartitions 获取 topic 的所有分区  
    scala.collection.immutable.Set<TopicAndPartition> topicAndPartitionSet2 =  
    (scala.collection.immutable.Set <TopicAndPartition>) kafkaCluster.getPartitions(immutab  
    leTopics).right().get();  
}
```

```
// 以 key-value 的形式记录每个 topic 的每个 partition 的 offset
Map<TopicAndPartition, Long> consumerOffsetsLong = new HashMap();

// 调用 KafkaCluster 的 getConsumerOffsets() 方法获取 offset
// isLeft 为 True 代表没有保存 offset (该 group 首次消费时), 各个 partition offset 默认为 0
if (kafkaCluster.getConsumerOffsets(groupId, topicAndPartitionSet2).isLeft()) {
    // 将 Scala 的 Set 类型转换为 Java 的 Set 类型
    Set<TopicAndPartition> topicAndPartitionSet1 = JavaConversions.setAsJavaSet(topicAndPartitionSet2);
    // 设置 offset 为 0
    for (TopicAndPartition topicAndPartition : topicAndPartitionSet1) {
        consumerOffsetsLong.put(topicAndPartition, 0L);
    }
} else {
    // offset 已存在, 使用 getConsumerOffsets() 获取已保存的 offset
    scala.collection.immutable.Map<TopicAndPartition, Object> consumerOffsetsTemp =
        (scala.collection.immutable.Map<TopicAndPartition, Object>) kafkaCluster.getConsumerOffsets(groupId, topicAndPartitionSet2).right().get();

    // 将 Scala 的 Map 类型转换为 java 的 Map 类型
    Map<TopicAndPartition, Object> consumerOffsets = JavaConversions.mapAsJavaMap(consumerOffsetsTemp);
    // 将 Scala 的 Set 类型转换为 java 的 Set 类型
    Set<TopicAndPartition> topicAndPartitionSet1 = JavaConversions.setAsJavaSet(topicAndPartitionSet2);

    // 建立分区与 offset 一一映射的 Map 结构
    for (TopicAndPartition topicAndPartition : topicAndPartitionSet1) {
        Long offset = (Long) consumerOffsets.get(topicAndPartition);
        consumerOffsetsLong.put(topicAndPartition, offset);
    }
}
return consumerOffsetsLong;
}
```

第五章 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 Tomcat 启动

1. Tomcat1 启动

```
/opt/modules/tomcat-1/bin/startup.sh
```

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷(中国)官网下载区】



2. Tomcat2 启动

```
/opt/modules/tomcat-1/bin/startup.sh
```

5.1.3 模拟日志发送程序启动

启动 data_producer 项目中的 GenBeahavior 程序，开始模拟日志的发送。

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-tailedirsource.jar --conf /opt/modules/flume/conf/ -f /opt/modules/flume/conf/flume-analysis.conf -n a1
```

5.2.3 聚合层 Flume 启动

```
flume-ng agent --conf /opt/modules/flume/conf/ -f /opt/modules/flume/conf/flume-analysis.conf -n a1
```

5.2.4 Kafka 集群启动

```
kafka-server-start.sh 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
```

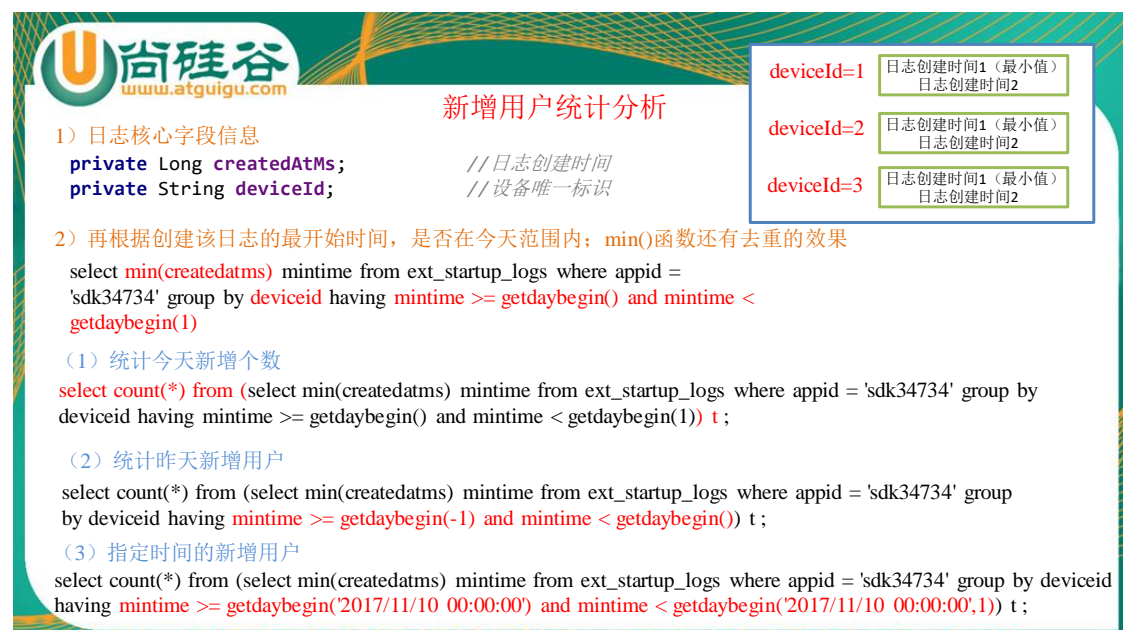
5.3.3 crontab 任务调度启动

```
service crond start
```

5.3.4 Kafka 高级消费者启动

启动 Log-processing 项目中 log-processing 模块中 com.atguigu.offline 包下的 KafkaConsumer 程序，开始消费 Kafka 集群中的日志数据，完成处理后将数据写入 HDFS 文件中。

5.3.5 新增用户统计



尚硅谷 www.atguigu.com

新增用户统计分析

1) 日志核心字段信息

```
private Long createdAtMs; // 日志创建时间
private String deviceId; // 设备唯一标识
```

2) 再根据创建该日志的最开始时间，是否在今天范围内；min()函数还有去重的效果

```
select min(createdatms) mintime from ext_startup_logs where appid = 'sdk34734' group by deviceId having mintime >= getdaybegin() and mintime < getdaybegin(1)
```

(1) 统计今天新增个数

```
select count(*) from (select min(createdatms) mintime from ext_startup_logs where appid = 'sdk34734' group by deviceId having mintime >= getdaybegin() and mintime < getdaybegin(1)) t;
```

(2) 统计昨天新增用户

```
select count(*) from (select min(createdatms) mintime from ext_startup_logs where appid = 'sdk34734' group by deviceId having mintime >= getdaybegin(-1) and mintime < getdaybegin()) t;
```

(3) 指定时间的新增用户

```
select count(*) from (select min(createdatms) mintime from ext_startup_logs where appid = 'sdk34734' group by deviceId having mintime >= getdaybegin('2017/11/10 00:00:00') and mintime < getdaybegin('2017/11/10 00:00:00',1)) t;
```

deviceId	日志创建时间1 (最小值)	日志创建时间2
deviceId=1	日志创建时间1 (最小值)	日志创建时间2
deviceId=2	日志创建时间1 (最小值)	日志创建时间2
deviceId=3	日志创建时间1 (最小值)	日志创建时间2

图 5-1 新增用户统计分析

1. 任意日新增用户

1) 今天新增用户

```
select
count(*)
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】



```
from
(select min(startTimeInMs) mintime
from ext_startup_logs
where appId = 'app00001'
group by userId
having mintime >= getdaybegin() and mintime < getdaybegin(1)
)t ;
```

2) 昨天新增用户

```
select
count(*)
from
(select min(startTimeInMs) mintime
from ext_startup_logs
where appId = 'app00001'
group by userId
having mintime >= getdaybegin(-1) and mintime < getdaybegin()
)t ;
```

3) 指定天新增用户

```
select
count(*)
from
(select min(startTimeInMs) mintime
from ext_startup_logs
where appId = 'app00001'
group by userId
having mintime >= getdaybegin('2017/11/10 00:00:00') and mintime < getdaybegin('2017/11/10 00:00:00',1)
)t ;
```

2. 任意周新增用户

1) 本周新增用户

```
select
count(*)
from
(select min(startTimeInMs) mintime
from ext_startup_logs
where appId = 'app00001'
group by userId
having mintime >= getweekbegin() and mintime < getweekbegin(1)
)t ;
```

2) 上一周新增用户

```
select
count(*)
from
(select min(startTimeInMs) mintime
from ext_startup_logs
where appId = 'app00001'
group by userId
having mintime >= getweekbegin(-1) and mintime < getweekbegin()
)t ;
```

3) 指定周时间的新增用户

```
select
count(*)
```

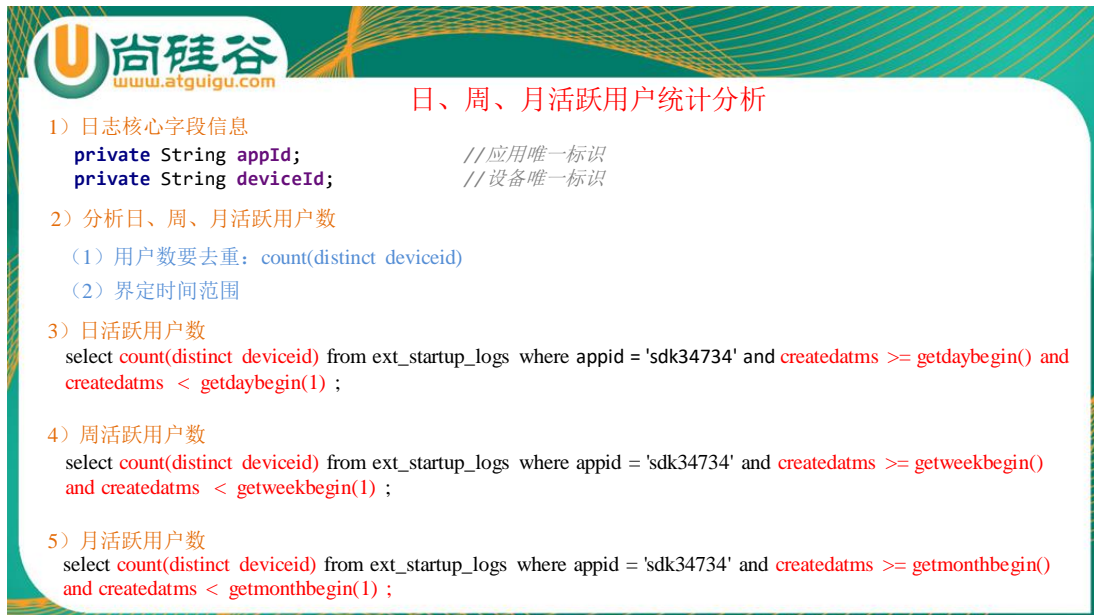
```
from
(select min(startTimeInMs) mintime
from ext_startup_logs
where appId = 'app00001'
group by userId
having mintime >= getweekbegin('2017/10/10 00:00:00') and mintime <
getweekbegin('2017/10/10 00:00:00',1)
)t ;
```

3. 月新增用户

```
select
count(*)
from
(select min(startTimeInMs) mintime
from ext_startup_logs
where appId = 'app00001'
group by userId
having mintime >= getmonthbegin(-3) and mintime < getmonthbegin(0)
)t ;
```

5.3.6 活跃用户统计

1. 日、周、月活跃用户



尚硅谷
www.atguigu.com

日、周、月活跃用户统计分析

- 日志核心字段信息
`private String appId;` // 应用唯一标识
`private String deviceId;` // 设备唯一标识
- 分析日、周、月活跃用户数
 - 用户数要去重: `count(distinct deviceId)`
 - 界定时间范围
- 日活跃用户数
`select count(distinct deviceId) from ext_startup_logs where appId = 'sdk34734' and createdatms >= getdaybegin() and createdatms < getdaybegin(1) ;`
- 周活跃用户数
`select count(distinct deviceId) from ext_startup_logs where appId = 'sdk34734' and createdatms >= getweekbegin() and createdatms < getweekbegin(1) ;`
- 月活跃用户数
`select count(distinct deviceId) from ext_startup_logs where appId = 'sdk34734' and createdatms >= getmonthbegin() and createdatms < getmonthbegin(1) ;`

图 5-2 日、周、月活跃用户统计分析

1) 日活跃用户数

```
select
count(distinct userId)
from ext_startup_logs
where appId = 'app0001'
and startTimeInMs >= getdaybegin() and startTimeInMs < getdaybegin(1) ;
```

2) 周活跃用户数

```
select
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】


```
count(distinct userId)
from ext_startup_logs
where appId = 'app0001'
and startTimeInMs >= getweekbegin() and startTimeInMs < getweekbegin(1);
```

3) 月活跃用户数

```
select
count(distinct userId)
from ext_startup_logs
where appId = 'app0001'
and startTimeInMs >= getmonthbegin() and startTimeInMs < getmonthbegin(1);
```

2. 指定时间内查询日活、周活、月活



1) 日志核心字段信息

- `private Long createTime;` // 日志创建时间
- `private String appId;` // 应用唯一标识
- `private String deviceId;` // 设备唯一标识

2) 分析指定时间内查询日活、周活、月活

- (1) 用户数要去重: `count(distinct deviceId)`
- (2) 界定时间范围
- (3) 分别根据天、周、月分组

3) 一次查询出一周内，每天的日活跃数

```
select formattime(createdatms,'yyy/MM/dd') day ,count(distinct deviceId) from ext_startup_logs where appId = 'sdk34734' and createdatms >= getweekbegin() and createdatms < getweekbegin(1) group by formattime(createdatms,'yyy/MM/dd');
```

4) 一次查询出过去的5周，每周的周活跃数

```
select formattime(createdatms,'yyy/MM/dd',0) week ,count(distinct deviceId) from ext_startup_logs where appId = 'sdk34734' and createdatms >= getweekbegin(-6) and createdatms < getweekbegin(-1) group by formattime(createdatms,'yyy/MM/dd',0);
```

5) 一次查询出过去的三个月内，每周的月活跃数

```
select formattime(createdatms,'yyy/MM',0) month ,count(distinct deviceId) from ext_startup_logs where appId = 'sdk34734' and createdatms >= getmonthbegin(-4) and createdatms < getmonthbegin(-1) group by formattime(createdatms,'yyy/MM',0);
```

图 5-3 指定时间内查询日活、周活、月活

1) 一次查询出一周内，每天的日活跃数

```
select
formattime(startTimeInMs,'yyy/MM/dd') day ,count(distinct userId)
from ext_startup_logs
where appId = 'app0001'
and startTimeInMs >= getweekbegin() and startTimeInMs < getweekbegin(1)
group by formattime(startTimeInMs,'yyy/MM/dd');
```

2) 一次查询出过去的 5 周，每周的周活跃数

```
select
formattime(startTimeInMs,'yyy/MM/dd',0) week ,count(distinct userId)
from ext_startup_logs
where appId = 'app0001'
and startTimeInMs >= getweekbegin(-6) and startTimeInMs < getweekbegin(-1)
group by formattime(startTimeInMs,'yyy/MM/dd',0);
```

3) 一次查询出过去的三个月内，每月的月活跃数

```
select
formattime(startTimeInMs,'yyy/MM') month ,count(distinct userId)
from ext_startup_logs
where appId = 'app0001'
```

```
and startTimeInMs >= getmonthbegin(-4) and startTimeInMs < getmonthbegin(-1)
group by formattime(startTimeInMs, 'yyyy/MM');
```

3. 优化活跃查询

根据时间分区表去查询，避免全表扫描

```
select
count(distinct userId)
from ext_startup_logs
where appId = 'app0001'
and ym = formattime(getdaybegin(), 'yyyyMM') and day = formattime(getdaybegin(), 'dd');
```

4. 过去五周周活跃用户数

过去的五周(包含本周)某个 app 每周的周活跃用户数

连接函数测试: select concat(ym,day) from ext_startup_logs;

```
select
formattime(startTimeInMs, 'yyyyMMdd', 0) stdate, count(distinct userId) stcount
from ext_startup_logs
where concat(ym, day) >= formattime(getweekbegin(-4), 'yyyyMMdd') and appId = 'app0001'
group by formattime(startTimeInMs, 'yyyyMMdd', 0);
```

5. 过去六个月活跃用户统计

最近的六个月(包含本月)每月的月活跃数

```
select
formattime(startTimeInMs, 'yyyyMM') stdate, count(distinct userId) stcount
from ext_startup_logs
where ym >= formattime(getmonthbegin(-5), 'yyyyMM') and appId = 'app00001'
group by formattime(startTimeInMs, 'yyyyMM');
```

6. 连续 n 周活跃用户统计



连续n周活跃用户统计

1) 日志核心字段信息

```
private Long createdAtMs; // 日志创建时间
private String appId; // 应用唯一标识
private String deviceId; // 设备唯一标识
```

2) 分析连续3周活跃用户

(1) 按照设备id分组

(2) 查询每个分组中的时间在3周内

```
where concat(ym, day) >= formattime(getweekbegin(-2), 'yyyyMMdd')
```

(3) 对符合查询条件的结果都转换为周一时间，然后去重

```
count(distinct(formattime(createdatms, 'yyyyMMdd', 0))) c
```

(4) 最后判断，统计结果是否为3

```
having c = 3
```

(5) 最终的查询语句

```
select deviceid, count(distinct(formattime(createdatms, 'yyyyMMdd', 0))) c from ext_startup_logs where appId = 'sdk34734' and concat(ym, day) >= formattime(getweekbegin(-2), 'yyyyMMdd') group by deviceid having c = 3;
```

deviceId=1	日志创建时间（周一、周二。。。）
deviceId=2	日志创建时间（周一、周二。。。）
deviceId=3	日志创建时间（周一、周二。。。）
deviceId=4	日志创建时间（周一、周二。。。）

图 5-4 连续 n 周活跃用户统计

连续活跃 3 周用户统计

```
select userId, count(distinct(formattime(startTimeInMs, 'yyyyMMdd', 0))) c
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```
from ext startup logs
where appId = 'app00001'
and concat(ym,day) >= formattime(getweekbegin(-2),'yyyyMMdd')
group by userId
having c = 3;
```

7. 忠诚用户

忠诚用户（连续活跃 5 周）

```
select userId , count(distinct(formattime(startTimeInMs,'yyyyMMdd',0))) c
from ext startup logs
where appId = 'app00001'
and concat(ym,day) >= formattime(getweekbegin(-4),'yyyyMMdd')
group by userId
having c >= 5;
```

5.3.7 沉默用户统计




图 5-5 展示了尚硅谷关于沉默用户统计的分析内容。左侧列出了日志核心字段信息（deviceId, appId, deviceId）、沉默用户条件（条件1：只有一条日志；条件2：刚安装的2天内不算沉默用户）、分析沉默用户的方法（按照设备id分组，日志产生时间只有一次，刚安装的2天内不算沉默用户）以及最终的查询语句。右侧是一个表格，列出了deviceId=1到4，每个设备ID都对应一个“日志创建时间”的输入框。

尚硅谷
www.atguigu.com

沉默用户统计分析

1) 日志核心字段信息

- private Long createdAtMs; // 日志创建时间
- private String appId; // 应用唯一标识
- private String deviceId; // 设备唯一标识

2) 沉默用户条件

- (1) 条件1：只有一条日志
- (2) 条件2：刚安装的2天内不算沉默用户

3) 分析沉默用户

- (1) 按照设备id分组
- (2) 日志产生时间只有一次
`count(createdatms) dcount` `having dcount = 1`
- (3) 刚安装的2天内不算沉默用户
`min(createdatms) dmin` `dmin < getdaybegin(-1)) t`
- (4) 最终的查询语句
`select count(*) from (select deviceid , count(createdatms) dcount,min(createdatms) dmin from ext_startup_logs where appid = 'sdk34734' group by deviceid having dcount = 1 and dmin < getdaybegin(-1)) t;`

deviceId=1	日志创建时间
deviceId=2	日志创建时间
deviceId=3	日志创建时间
deviceId=4	日志创建时间

图 5-5 沉默用户统计

查询沉默用户数（一共只有一条日志；且安装时间超过 2 天）

```
select
count(*)
from
(select userId , count(startTimeInMs) dcount,min(startTimeInMs) dmin
from ext startup logs
where appId = 'app00001'
group by userId
having dcount = 1 and dmin < getdaybegin(-1))t;
```

5.3.8 启动次数统计

今天 app 的启动次数

启动次数类似于活跃用户数，活跃用户数去重，启动次数不需要去重。

```
select
count(userId)
from ext_startup_logs
where appId = 'app00001'
and ym = formattime(getdaybegin(),'yyyyMM') and day = formattime(getdaybegin(),'dd');
```

5.3.9 版本分布统计

1. 今天 appId 为 app00001 的不同版本的活跃用户数

```
select
appVersion,count(distinct userId)
from ext_startup_logs
where appId = 'app00001'
and ym = formattime(getdaybegin(),'yyyyMM') and day = formattime(getdaybegin(),'dd')
group by appVersion;
```

2. 本周内每天各版本日活跃数

- 1) 本周内: where concat(ym,day) >= formattime(getweekbegin(),'yyyyMMdd')
- 2) 每天: group by formattime(createdatms,'yyyyMMdd')
- 3) 各个版本: group by appversion
- 4) 日活跃数: count(distinct userId)

```
select
formattime(startTimeInMs,'yyyyMMdd'),appVersion , count(distinct userId)
from ext_startup_logs
where appId = 'app00001'
and concat(ym,day) >= formattime(getweekbegin(),'yyyyMMdd')
group by formattime(startTimeInMs,'yyyyMMdd'), appVersion;
```

5.3.10 留存分析统计

1. 本周回流用户统计

本周回流用户: 上周没有启动过，本周启动过

 尚硅谷
www.atguigu.com

本周回流用户统计分析

1) 日志核心字段信息

```
private String appId;           // 应用唯一标识
private String deviceId;        // 设备唯一标识
```

2) 本周回流概念：本周启动过，但上周没有启动过，

3) 分析本周回流用户

(1) 本周启动过、且去重

```
distinct s.deviceid
where concat(ym,day) >= formattime(getweekbegin(),'yyyyMMdd')
```

(2) 且不在上周启动过的结果中

```
and deviceid not in 上周启动过
```

(3) 上周启动过

```
select distinct t.deviceid from ext_startup_logs t where t.appid = 'sdk34734' and concat(t.ym,t.day) >=
formattime(getweekbegin(-1),'yyyyMMdd') and concat(t.ym,t.day) < formattime(getweekbegin(),'yyyyMMdd')
```

(4) 最终的查询语句

```
select distinct s.deviceid from ext_startup_logs s where appId = 'sdk34734' and concat(ym,day) >=
formattime(getweekbegin(),'yyyyMMdd') and deviceid not in (select distinct t.deviceid from ext_startup_logs t where
t.appid = 'sdk34734' and concat(t.ym,t.day) >= formattime(getweekbegin(-1),'yyyyMMdd') and concat(t.ym,t.day) <
formattime(getweekbegin(),'yyyyMMdd'));
```



图 5-6 本周回流用户统计

```
select
distinct s.userId
from ext_startup_logs s
where appId = 'app00001' and concat(ym,day) >= formattime(getweekbegin(),'yyyyMMdd') and
userId not in (
select
distinct t.userId
from ext_startup_logs t
where t.appId = 'app00001' and concat(t.ym,t.day) >= formattime(getweekbegin(-1),
'yyyyMMdd') and concat(t.ym,t.day) < formattime(getweekbegin(),'yyyyMMdd')
);
```

2. 连续 n 周没有启动的用户

连续 2 周内没有启动过：本周和上周没启动过，大上周之前启动过

```
select
distinct s.userId
from ext_startup_logs s
where appId='app00001'
and concat(ym,day) < formattime(getweekbegin(-1),'yyyyMMdd')
and userId not in (
select
distinct(t.userId)
from ext_startup_logs t
where t.appId='app00001'
and concat(t.ym,t.day) >= formattime(getweekbegin(-1),'yyyyMMdd')
);
```

3. 留存用户统计

本周留存用户=上周新增用户和本周活跃用户的交集

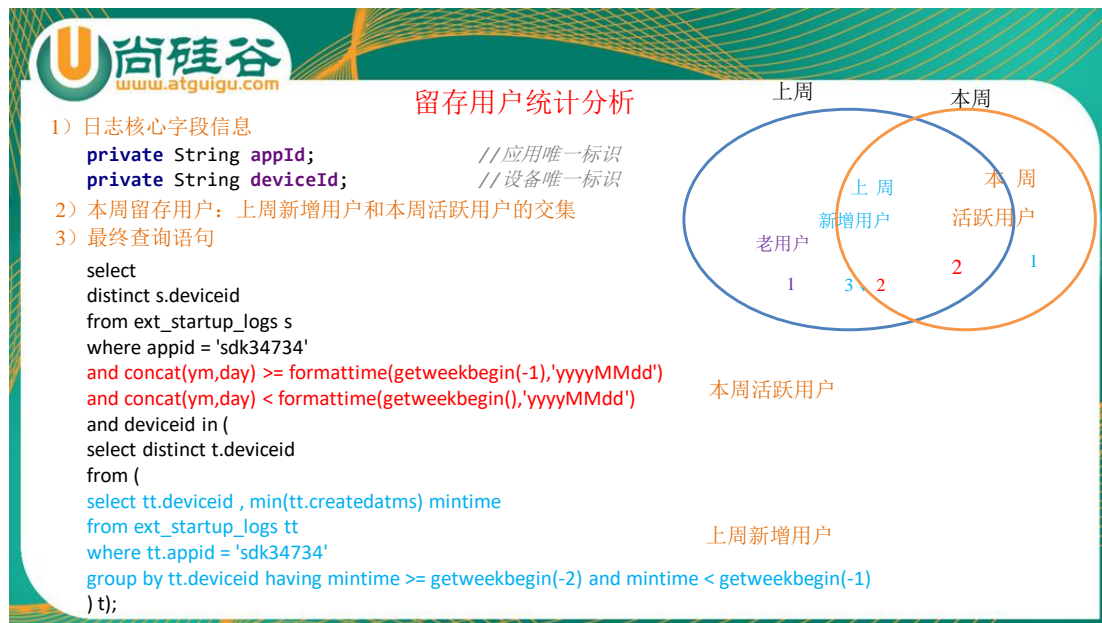


图 5-7 留存用户统计

```
select
distinct s.userId
from ext_startup_logs s
where appId = 'app00001'
and concat(ym,day) >= formattime(getweekbegin(-1),'yyyyMMdd')
and concat(ym,day) < formattime(getweekbegin(),'yyyyMMdd')
and userId in (
select distinct t.userId
from (
select tt.userId, min(tt.startTimeInMs) mintime
from ext_startup_logs tt
where tt.appId = 'app00001'
group by tt.userId having mintime >= getweekbegin(-2) and mintime < getweekbegin(-1)
) t);
```

5.3.11 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

1. 今天新增用户（为 n）

```
select
count(*)
from
(select min(startTimeInMs) mintime
from ext_startup_logs
where appId = 'app00001'
group by userId
having mintime >= getdaybegin() and mintime < getdaybegin(1)
) t;
```

2. 今天活跃用户（m）

```
select
count(distinct userId)
```




```
from ext startup logs
where appId = 'app00001'
and startTimeInMs >= getdaybegin() and startTimeInMs < getdaybegin(1);
```

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 Hbase 集群启动

```
Start-hbase.sh
```

5.4.2 SparkStreaming 程序启动

启动 Log-processing 项目中 log-processing 模块中 com.atguigu.streaming 包下的 SparkStreamingDirect 程序，开始消费 Kafka 集群中的日志数据并进行实时处理，实时处理后的数据写入 Hbase 中完成统计。

5.4.3 各城市用户点击次数实时统计

在实时数据处理系统中，日志采集系统采集到的日志数据经过处理后，被存储到 Hbase 中。实时数据处理系统根据 Startup 中 city 字段的不同，对不同城市用户的点击次数进行统计，并不断更新各个城市用户用户的点击次数，Hbase 中的数据信息如图 1-25 所示。

```
hbase(main):049:0> scan 'online_city_users'
COLUMN+CELL
ROW
Beijing      column=StatisticData:userNum, timestamp=1518168908484, value=\x00\x00\x00\x00\x00\x00\x00\x04
Guangzhou    column=StatisticData:userNum, timestamp=1518168908672, value=\x00\x00\x00\x00\x00\x00\x00\x05
Hangzhou     column=StatisticData:userNum, timestamp=1518168908680, value=\x00\x00\x00\x00\x00\x00\x00\x03
Hunan        column=StatisticData:userNum, timestamp=1518168908691, value=\x00\x00\x00\x00\x00\x00\x00\x05
Shanghai     column=StatisticData:userNum, timestamp=1518168909272, value=\x00\x00\x00\x00\x00\x00\x00\x05
Shenyang     column=StatisticData:userNum, timestamp=1518168908657, value=\x00\x00\x00\x00\x00\x00\x00\x08
Shenzhen     column=StatisticData:userNum, timestamp=1518168907996, value=\x00\x00\x00\x00\x00\x00\x00\x01
Tianjin      column=StatisticData:userNum, timestamp=1518168908393, value=\x00\x00\x00\x00\x00\x00\x00\x08
Xian         column=StatisticData:userNum, timestamp=1518168914219, value=\x00\x00\x00\x00\x00\x00\x00\x05
Xinjiang     column=StatisticData:userNum, timestamp=1518168907582, value=\x00\x00\x00\x00\x00\x00\x00\x02
10 row(s) in 0.1390 seconds
```

图 5-8 Hbase 数据信息

第六章 项目总结

手机 APP 数据统计分析项目将离线数据分析系统与实时数据分析系统相融合，通过公共的日志采集模块进行日志的采集工作，随后将相同的日志数据分别发送给离线数据分析系统和实时数据分析系统，实现在不同维度对 APP 日志数据进行分析。

本项目有以下几个关键点：

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

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 Flume agent 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 Taildir Source 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 Flume agent 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 Flume agent 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 agent 的 channel 中的数据。

6) 当第二层聚合 Flume agent 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 Kafka offset 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 SparkStreaming。

10) 当写入 HBase 时, 需要在 rdd.foreachPartition 里面创建 HBase Client 实例, 以免创建过多 HBase Client 实例对集群产生影响。

11) 根据需求建 HBase 模型时尽量通过一个 RowKey 查询就获取所需数据。