Databus

设计文档

1.0

黄世钱

本文为 databus 设计文档,部署调优等请参考部署文档

概述

Databus 是由 linkedin 开源的低延迟的关系型数据库变化数据捕获系统。提供了以下特性

- 隔离和解耦数据源和消费者
- 保证数据变化的有序性,保证被捕获数据至少一次发布
- 基于随机时间点访问变化数据流,同时引导 client 容灾恢复和持续消费数据
- 消费分片
- 源数据一致性

源码获取地址 https://github.com/linkedin/databus
部署请参考示例 https://github.com/linkedin/databus/wiki/Databus-2.0-Example

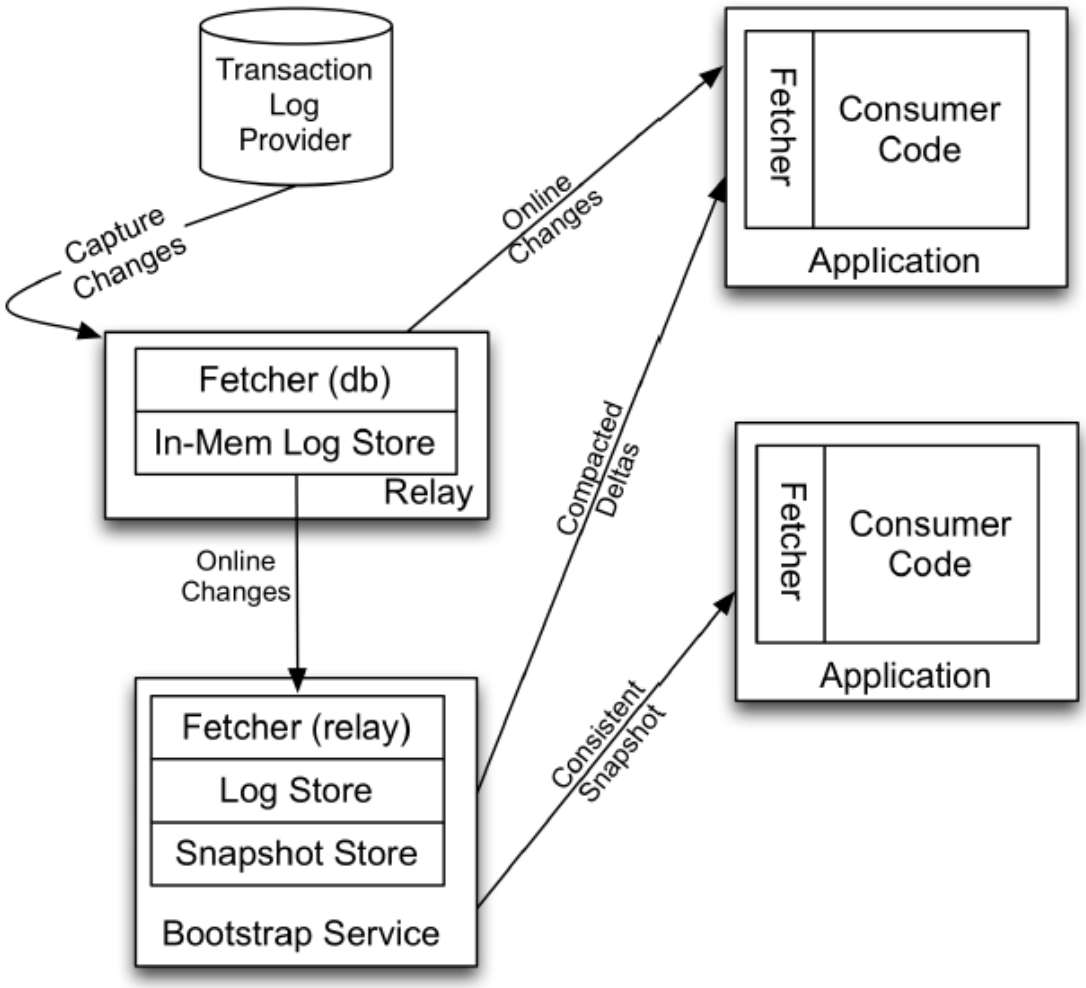
名词定义

名词	定义
Databus source	变化数据来源,通常指数据库中的表和视图,也可指 chain relay 数据来源 event buffer
Master source	主数据源,指真正的数据来源,也就是数据库中的表和视图
Databus relay	从数据源中捕获变化数据的组件
Databus consumer	变化数据的真正消费者程序,需要遵守 databus consumer 规范
Databus consumer group	捕获同类 events 的一组 consumer
Databus client	Consumer 与 relay 数据交互的中间件,relay 对于 consumer 而言是透明的.client 会回调 sonsumer 逻辑

Data change event	变化数据捕获后封装为 change event
Consistency window	一致性数据窗口,系统保证同一窗口批次的事件处理状态的一致性。类似于将批量数据处理作为一个整体事务进行处理。
Scn	系统数据事务变化编号,64 位 long 类型数据

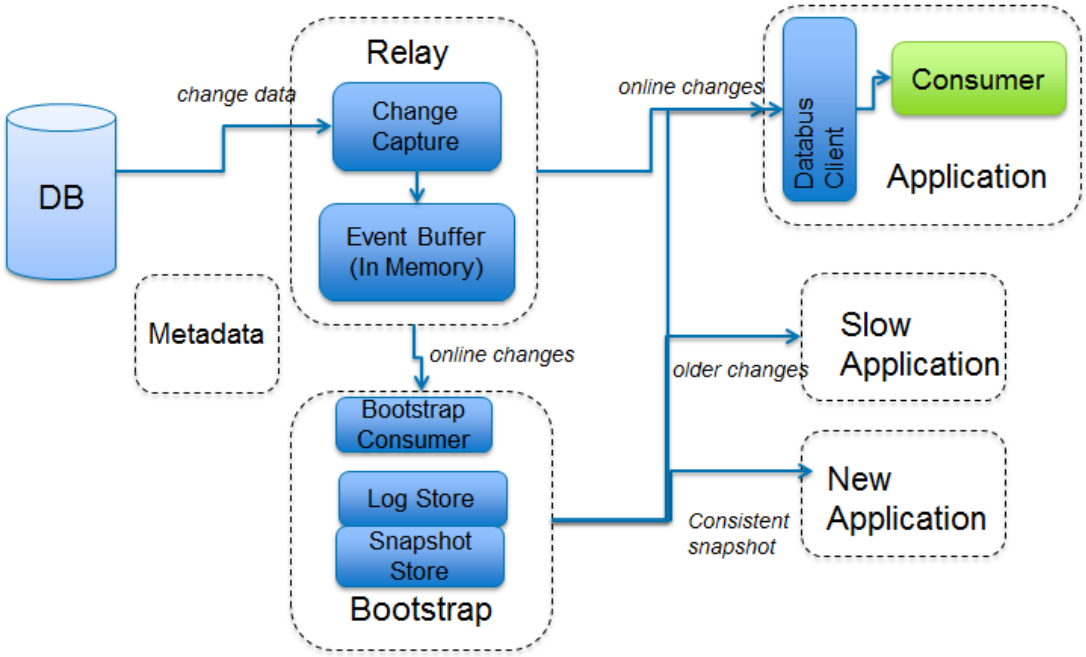
架构

逻辑结构

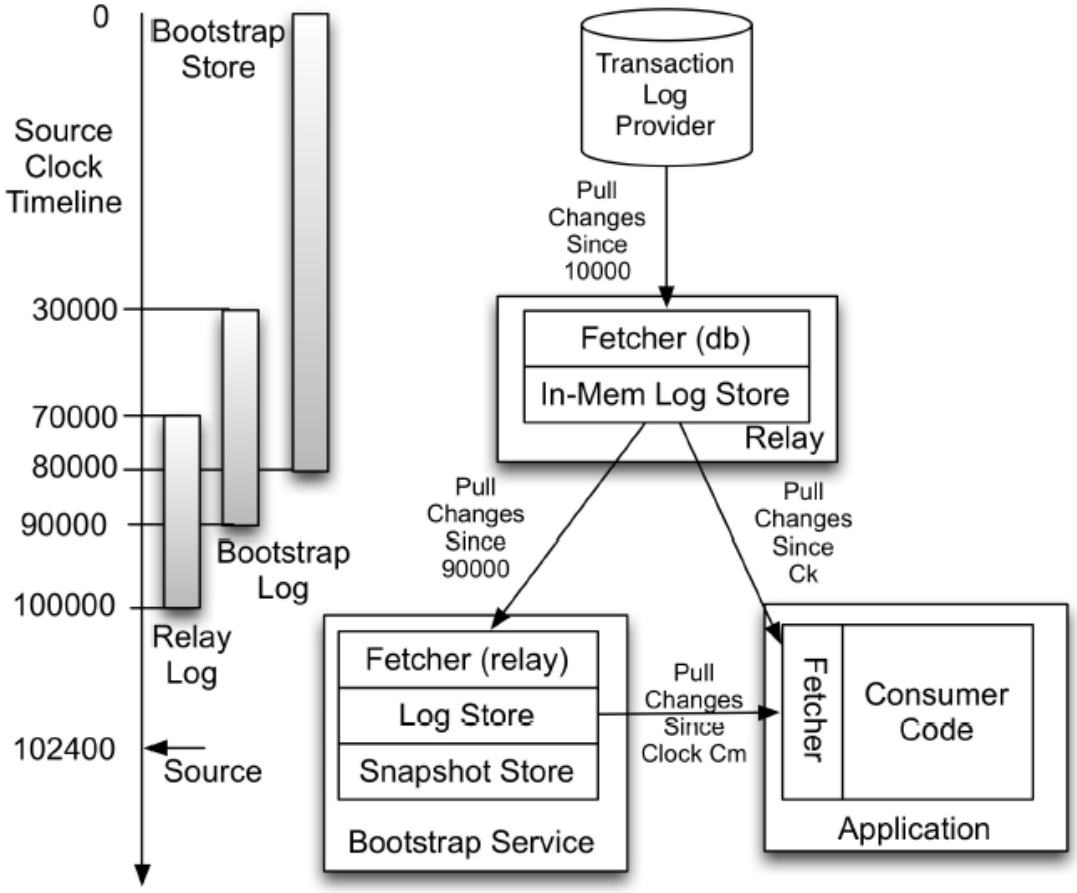


Databus 分为四个逻辑功能

- Fetcher:从源系统拉取数据
- Log Store:存储变化数据流
- Snapshot Store:存储历史数据快照
- Consumer:消费和使用数据



数据在 Databus 中以 pull 模式进行流转,databus 通过此不同的方式实现 Fetcher&Store 等四个组件满足数据存储订阅流转的需求。



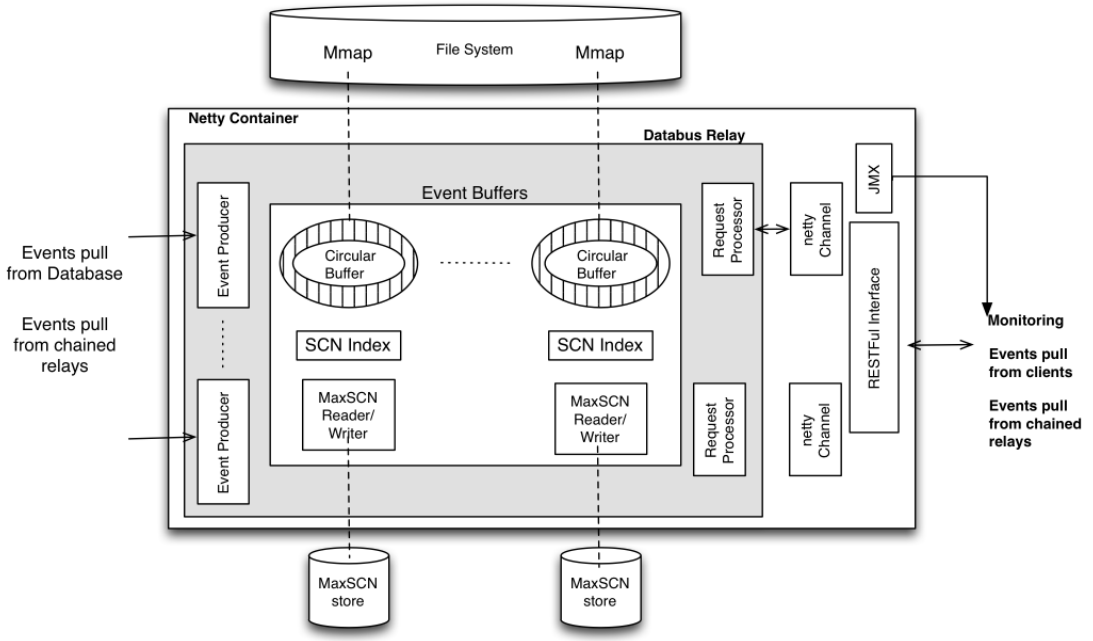
为了满足灵活的数据读取需求,databus 将使用不同的组件对实时和历史数据分开服务。如上图所示,源系统中的当前事务变化 SCN 为 102400,Relay Log Store 中当前存储了 SCN：70000-100000 的事务数据,Bootstrap Log Store 中存储了 SCN:30000-90000 的数据,Snapshot Store 中存储了 SCN:0-80000 的事务数据. Consumer 会根据自身当前处理的 SCN 号,从不同的组件中读取数据

Databus Relays(中继器)设计

功能

- 捕获数据变化
- 将变化数据打包为事件序列化到内存 buffer
- 响应 client/bootstrap producer 的请求,传输数据

设计



Relay 从数据源拉取数据并封装为事件,每个事件拥有一个自增的 scn(系统变化自增列,参考 oracle 中的 ora_rowscn)来记录事务顺序,有序存储在一个或多个环形缓冲区(Circular Buffer)中。Buffer 分为直接内存缓冲(direct)和文件系统映射缓冲(mmap)两种模式。但是不管使用哪种 buffer,在内存中都有相应的 SCN 稀疏索引和对应的 MaxSCN Reader/Writer。MaxSCN Reader/Writer 会定期持久化 relay 中新拉取的最大的 SCN。

当 buffer 被配置为内存映射模式时,relay 会在关闭时将 scn 索引以及元数据存储到本地硬盘以便于重启时恢复数据。Relay 通过 Request Processor 监听 netty channel 来处理客户端的请求。对外通过 restful api 提供管理和事件拉取服务。

Databus Event Producer

EventProducer 定期的轮询和捕获所有数据源中的主 oracle 数据源的数据变化，将捕获到的变化的数据行读取出来并且封装为 avro 记录。Avro 的格式会由 databus 根据 oracle schema 自动生成。

Avro 记录会继续序列化为包含更多 databus 所需元数据的 databus event(参见 DbusEvent 类)。DvusEvent 可以通过 avro 的二进制序列化机制进行荷载(payload:参见 http 的 payload 含义与作用)。

AbstractEventProducer 类提供了生成 event producer 的框架, OracleEventProducer 读取 oracle 中变化数据来生成 DbusEvent。Oracle 中需要创建 txlog 表用来记录数据变化,并且通过触发器更新 txlog table.详情参考 txlog 表结构以及源数据库设计。

MaxSCN Reader/Writer

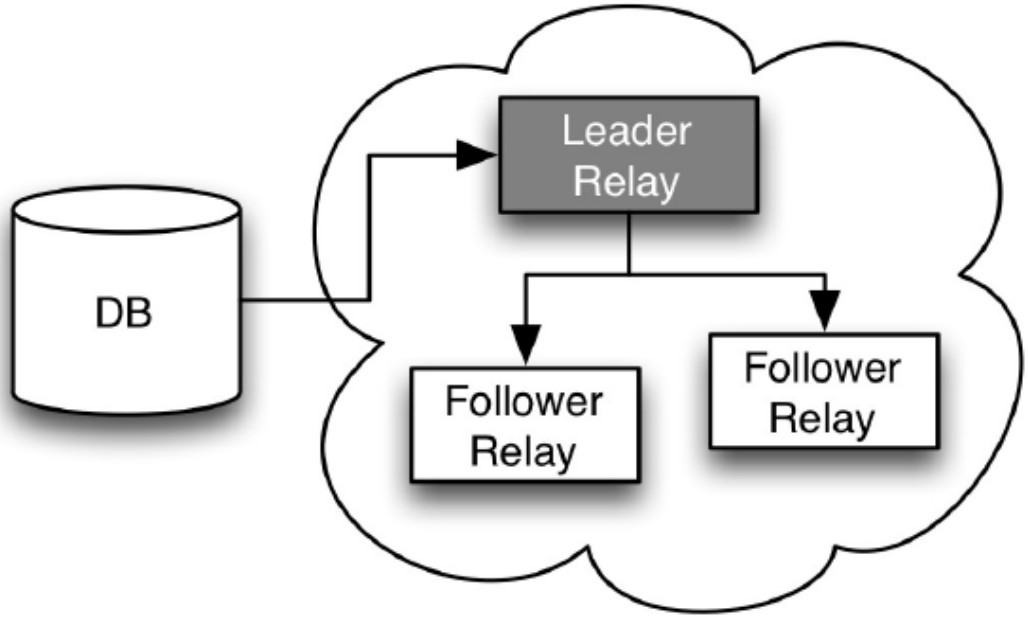
MaxSCN Reader/Writer 的作用是跟踪 DBEP(Database Event Producer)的过程，在 DBEP 的启动时,如果 SCN 没有赋值,则会使用 Reader 读取已经存储的 SCN 来恢复数据。当数据被捕获并存储到 Event Buffer 后,DBEP 使用 writer 来保存最后处理的 SCN。目前 SCN 数据保存在本地文件中,也可以使用 RDBMS 或者 Zookeeper 来进行保存。本地文件的 SCN 数据格式如下:

```
5984592768094:Thu Dec 13 00:54:07 UTC 2012
```

Chain Relay&Relay Cluster

功能

- 提供 relay 容灾
- 为 relay 添加从其他 Relay 获取数据的能力
- 提供对 relay 工作过程的感知功能
- 提供多 relay 对同一源数据链的管理能力



设计

- 为减少对源系统的压力,使用一主多从的 Relay Cluster
- 为 relay 提供一个从任意给定的 event buffer 读取数据的 consumer(类似于 Client 的 consumer)
    - 优点:通过对 buffer 的管理提供 relay 之间感知的能力
    - 优点:可以扩展提供源系统非主键数据过滤的功能

    - 缺点:相比较于通过 Avro 的 payload 传输数据,DbusEvent 的反序列化会降低性能
- 使用配置中数据源 physicalSource 的 URI 属性决定 relay chain 的顺序,例如将 URN 设置为为 *abc.foo.com:9993*,则会将订阅域名 *abc.foo.com* 的 Relay 分配到同一 cluster
- Chain relay 会读取 source relay 的所有数据
    - 在启用 checkpoint 功能时,chain relay 从 checkpoint 之后读取数据,否则会读取 source relay 的 buffer 中所有数据,但是 chain relay 目前没有提供类似于正常 relay 启动时 `restartScnOffset`(参考 `relay` 功能设计)的功能。
- Chain relay 会被禁用掉从 bootstrap producer 获取引导数据的功能
- 在 chain relay 启动时,需要指定初始的 SCN 值(保证数据的低水位线)
- 快速启动读取追踪 source relay 数据，无限制的重试失败的连接尝试
    - consumer relay 与 source relay 使用同样的 scn 保存恢复机制，使用 maxscn file 实现 maxscn 的 checkpoint
    - 与普通 relay 不同在于在 endDataSequence()调用时保存 scn checkpoint
    - Consumer 并行度为 1
- 使用与普通 relay 相同的 log 信息

Relay 选举

当前通过配置来分配不同的 relay 功能。未来的版本会加入自动分配功能。实现自动功能分配需要系统提供如下特性

- 高层结构
    - 物理 DB 定义为 master source(主数据源),Relay 的数目为初始为 N 个,k 个 relay 直接从主数据源读取数据,命名为 Leader 节点,N-k 个 relay 从 Leader 节点读取数据,命名为 follower 节点
    - 当其中一个 leader 节点挂掉时,监听 leader 的 follower 节点会转而从主数据源读取数据并且将自身转化为新的 leader 节点
    - 将 relay 的 buffer 当作一种 data source,以实现 Leader/Follower 数据交互
- 功能
    - 每个 relay 都可以从主数据源和 buffer 数据读取数据
    - 每个 relay 都可以方便的进行 Leader/Follower 模式转换
- 实现
    - 提供 cluster 概念,将订阅同一个 master source 的 relay 分为一个 cluster 以方便管理
    - 提供 leader/follower 分配的决策工具以方便 relay 在任意时刻进行模式转换
    - 提供自动化 leader 容灾处理
    - 提供可配置化的 leader 指定功能
- 扩展
    - 根据 server 数量自动调整 cluster 大小
    - 系统需要保证 leader 的高可用性(容灾的处理时间和自动数据源切换)
    - 通过 helix(类似于 pinot 中的 helix 使用)进行资源状态管理
    - 提供调整 cluster 节点功能
    - 不再需要 bootstrap producer 来提供容灾、回溯数据和引导 client

Bootstrap Service

Bootstrap service 由以下组件组成

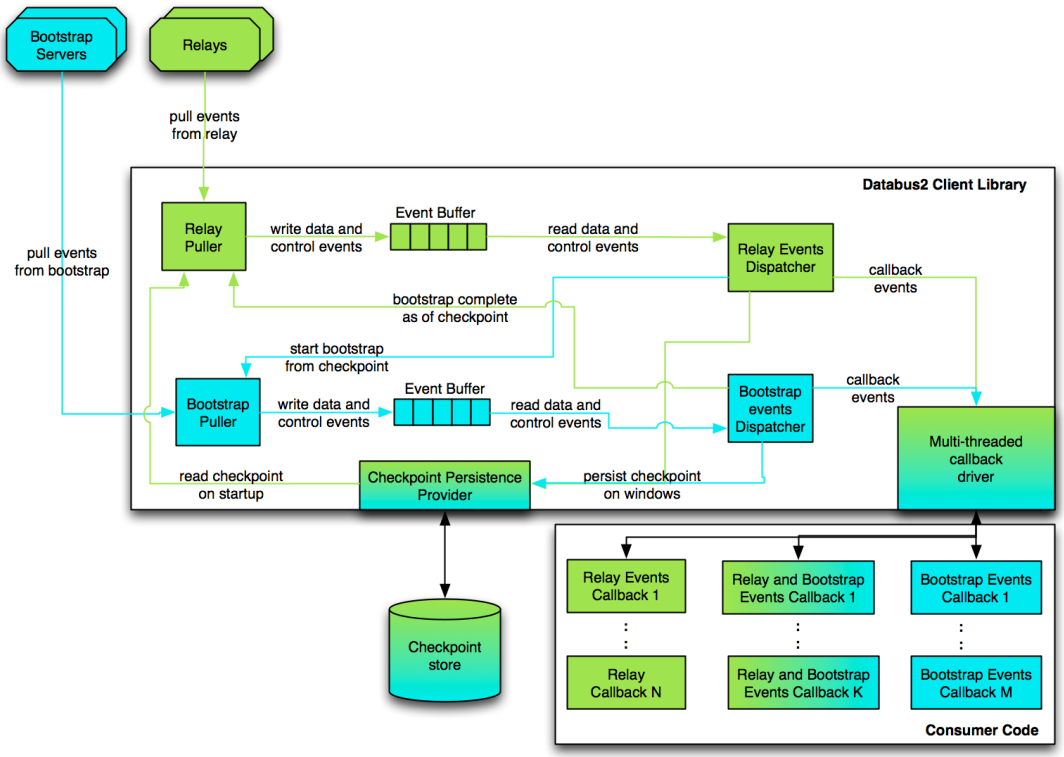
- Bootstrap producer:一个特殊的 databus client,从 relay 中读取数据并写入到 log store
- Bootstrap applier:从 log store 中读出数据并序列化到 db 中
- Snapshot Store DB:存储历史数据变化事件的数据库
- Bootstrap Server:相应 client 拉取历史引导数据的请求

Bootstrap service 屏蔽掉读取历史数据变化对原系统造成的压力,同时 bootstrap service 允许 client 限定单次读取的数据量以减小自身压力。

Databus Clients 设计

- 检测数据变化事件,回调业务逻辑
- 与 relay 严重不同步时,从 bootstrap server 同步所需的延迟
- 新增 client 从 bootstrap server 一次性读取历史数据,再从 relays 中读取最近变化的数据
- 可以单个 client 处理全部数据流,也可多个 client 分片处理

组件



Relay Connection

连接 Relay 获取变化数据流,将数据保存到 Event Buffer

Bootstrap Connection

连接 bootstrap producer 获取历史数据,并将数据保存到 Event Buffer

Dispatcher

分为 Relay 和 Bootstrap Event Dispatcher 两种,分别从对应的 Event Buffer 消费事件。是 client 端的核心事件调度处理组件。详细功能为

- 注册和回调正确的 consumer 逻辑
- 错误和超时处理

代码请参考 com.linkedin.databus.client.BootstrapDispatcher/RelayDispatcher

Consumer Code Callbacks

消费者回调接口

Checkpoint persistence

保存数据消费的状态,使用 JSON 格式存储数据

Event Model and Consumer API

Data Change Events

源数据库中每条数据变化都会封装为一个 event,event 分为三个模块

- 元数据
    - 操作[insert/update/delete]
    - 序列:记录 event 发生的顺序
- 主键
- 变化后数据

详细实现请参考 DbusEvent 代码

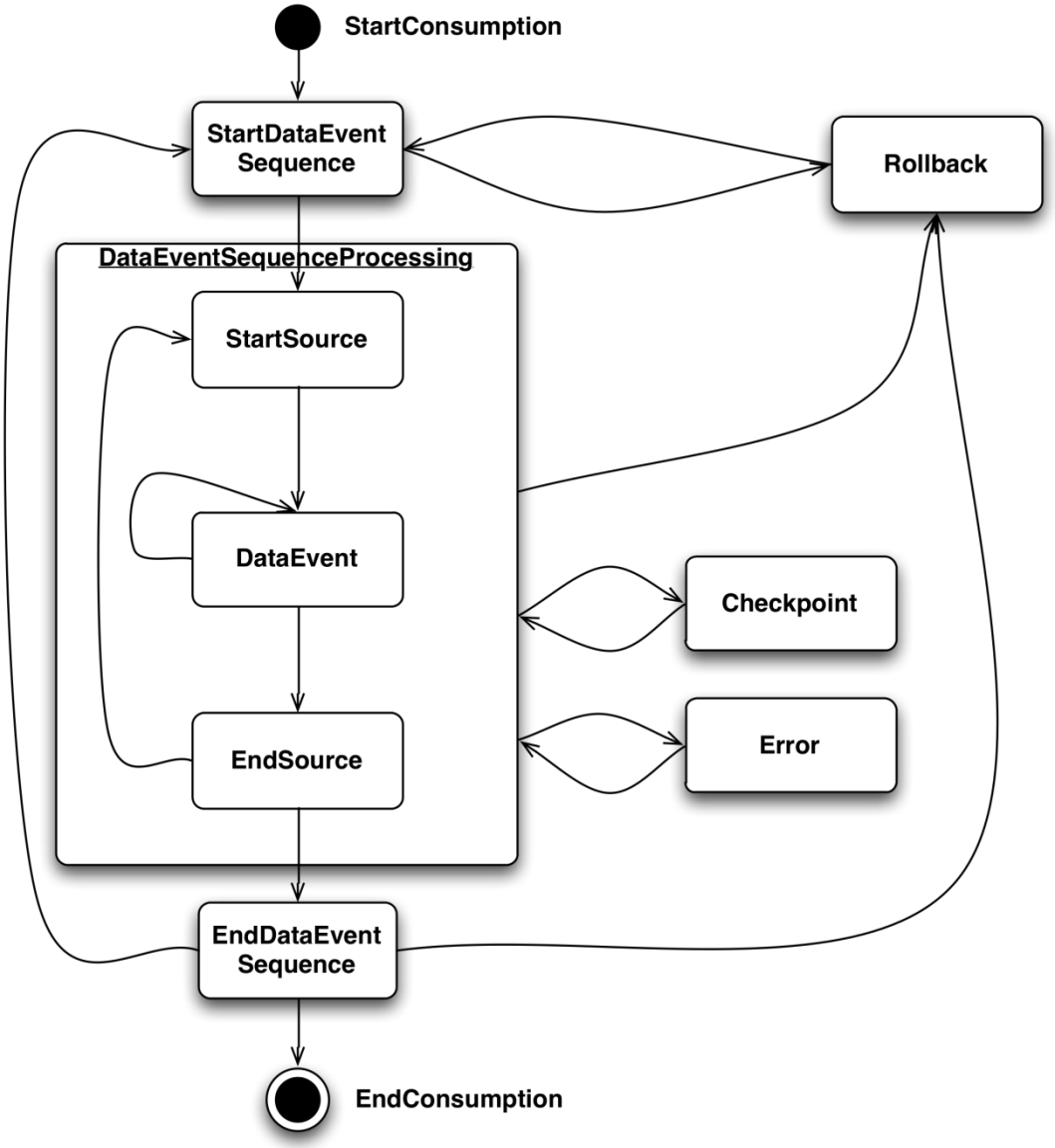
Event Type

Event 作为组件之间信息交互的基础,databus 提供了多种 Event Type 以协同各组件之间的工作

- StartDataEventSequence:启动数据一致性处理事务
- StartSource:开始从单个数据源读取数据
- DataEvent:开始传输当前数据源数据
- EndSource:当前数据源数据读取完成
- EndDataEventSequence:数据一致性处理事务结束
- RollBack:回滚

这些 Event 协同工作以实现数据的传输和事务保证,工作流程如下

```
StartDataEventSequence(startSCN) --> // 从起始 scn 开启数据处理事务
    StartSource(Source1) --> //开始从单个数据源读取数据
        DataEvent(Source1.data1) --> ... --> DataEvent(Source1.dataN) -->
    EndSource(Source1)  --> //当前数据源读取完毕
    StartSource(Source2) --> //从其他数据源读取数据
        DataEvent(Source2.data1) --> ... --> DataEvent(Source2.dataM) -->
    EndSource(Source2) --> //当前数据源读取完毕
    ... -->
EndDataEventSequence(endSCN) //在最后的 scn 处理完成后事务完成
```



首先 client 通知 consumer 开启数据处理事务,然后 Consumer 注册的 source 订阅顺序向 consumer 循环传输不同表(source)的变化数据,同时保证数据状态的一致性。Consumer 之间不需要事务协调。

Databus 需要保证单个 source 的 data event 的传输顺序.在未来的版本可能会新增 CheckPoint Event 以通知 consumer 立即保存当前事务状态以实现 consumer 重用上个 checkpoint 数据来进行容灾处理。这可以保证系统处理海量源数据变化时(例如单次更新某大数据源 20%以上数据)的系统稳定性。

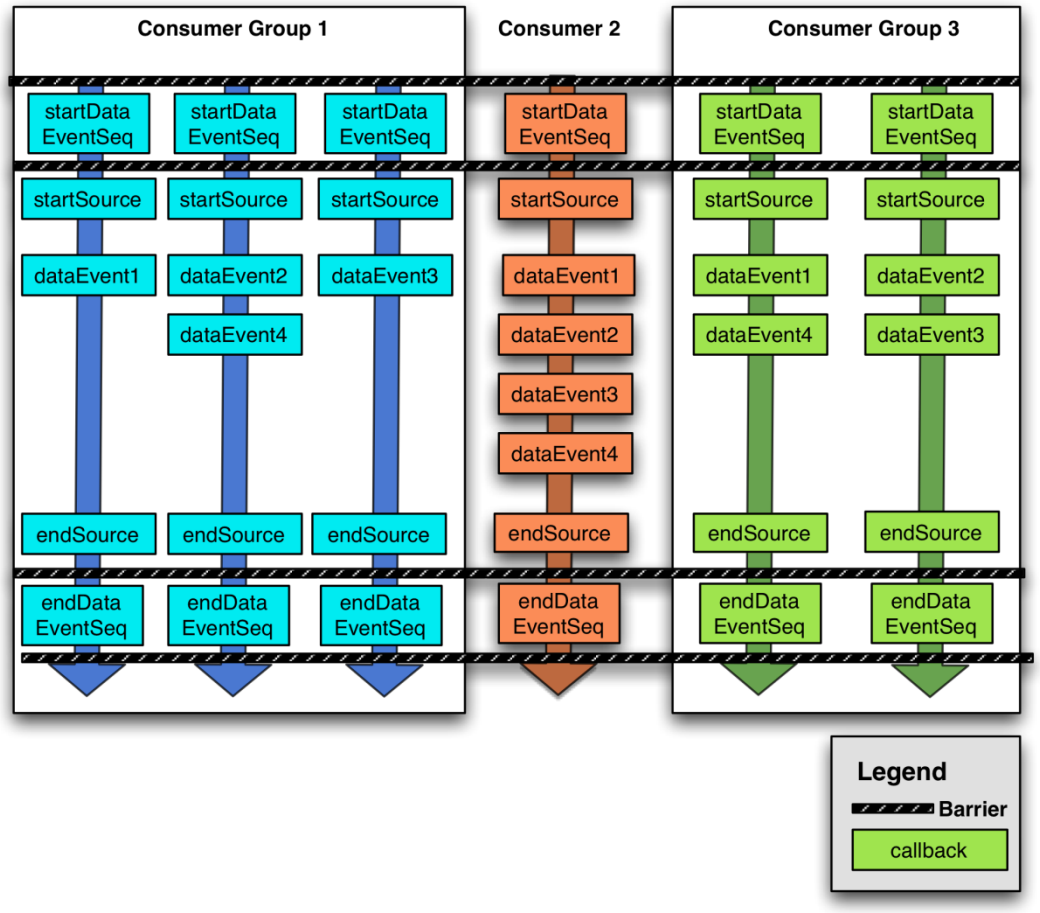
当事务处理(Sequence Event)过程中发生不可修复的错误时,client 会向 consumer 发送 rollback Event 使状态回滚,并且从错误的事务的起始 SCN 重启数据流程以保证不会遗漏数据。Client 会使用额外的 event buffer 保存一定数量的 events 状态以支持 rollback

当 consumer 多线程读取数据时, 每个线程都会接收到 StartDataEventSequence, EndDataEventSequence, StartSource, EndSource events,但是只接收分配到此线程的 DataEvent 事件。线程之间通过 CyclicBarrier 协调 StartSource 和 EndSource 事件

Databus Consumers API

当 client 检测到数据处理异常时会终止掉整个数据处理事务并尝试从失败事务的起始点进行恢复, Consumer 需要设置良好的异常处理机制以规避掉不必要的异常传播

Execution Model



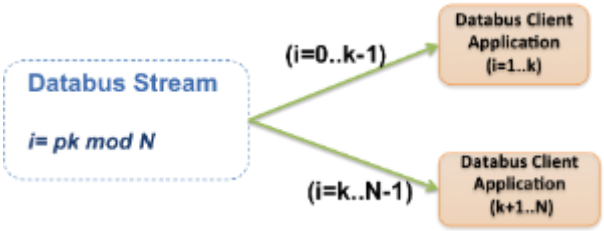
consumer 事件的分配机制如图片所示

回调函数	传输事件	group 分发机制	协调 Barrier
onStartDataEventSequence	StartDataEventSequence	广播	回调前/后
onStartSource	StartSource	广播	回调前
onDataEvent	DataEvent	分发	无
onCheckpoint	CheckPointEvent	广播	回调前/后
onError	ErrorEvent	广播	回调前/后
onEndSource	EndSource	广播	回调前/后
onEndDataEventSequence	EndDataEventSequence	广播	回调前/后
onRollback	RollBack	广播	回调后

负载均衡

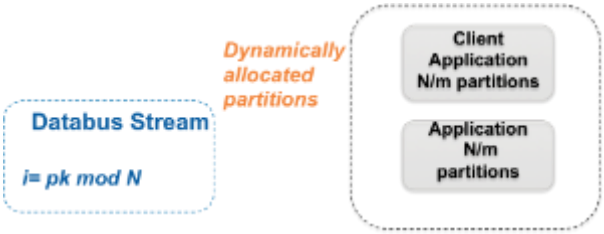
多个 client 可以组成一个 cluster 来并行的处理 relay 中不同分片的数据。Databus 支持静态分片和动态分片两种模式,动态分片模式下, cluster 中新增 client 时会触发重分片(重新计算数据分片)操作。Client 使用此分片信息去连接 relay/bootstrap server 获取对应的数据。Relay 会使用分桶分片分发数据以更高效的利用网络带宽。当重分片发生时,client 会自动创建新的 cosumer callbacks 以处理新的分片数据。

静态分片



- 策略:范围或者 hash
- 通过配置设置数据分片数量 N,不支持频繁的增删 client
- 使用 server-side-filter 的分桶方式实现
- 每个分片只由一个 client 实例处理

动态分片



- 可以向 cluster 中动态增删 client
- Client 上下线会触发自动重分片
- 分片数量根据 cluster 中 client tokens 决定
- Client 之间使用 helix 沟通和交互分片信息

Partition&Server Side Filtering

可以在 relay 或者 bootstrap server 上开启 Server side Filtering 过滤数据来减少网络 IO。当前支持基于主键的范围和分桶两种数据分片方式。不同的 client 来读取不同分分数据以加快数据处理速度。Databus 还支持在 server 端对主键进行过滤。准确来说,databus 中分片是基于过滤的方式实现的。Relay 会根据配置预先计算 DbusEvent 所属的分片,当 consumer 的分

片方式和 Relay 中相同时则可以直接基于 DbusEvent 的 partitionid 字段划分数据,不符合时再根据 primary key 进行数据过滤。

范围分片

需要在 consumer 的请求中显示指定分片类型和分片大小 rangeSize。预先分片需要单独配置

- FilterType=Range
- IntervalSize=1M(例:1000000)
- PartitionIds:consumer 将[1-1M],[1M-2M]等数据范围自动转化为分片范围[1,2,5-10]此类格式

当使用范围分片时,client 需要在请求中显式的指定期望访问的分片,当前支持的请求格式如下

1. List(列表):例如 \[1, 5, 10\],代表 client 需要读取第 1、5、10 这三个分片的数据
2. Range(范围):例如\[1-10\],代表 client 需要第 1 到第 10 个分区的所有分片中数据
3. 组合:例如\[1,2,5-10],代表 client 需要读取第 1,2 和第 5 到第 10 个分片的所有数据

桶分片

需要在 consumer 的请求中显示指定分片模式为 mod 和分桶数目,Relay 中预先分区需要单独配置

- FilterType=mod
- Bucket Size=10(例:10 个桶)
- BucketId:类似于范围分片的格式, \[1, 5, 10\],\[1-10\],\[1,2,5-10],只不过分片变为了桶

Databus Event Buffer

- 跨平台
- 可控大小
- 索引
- 锁粒度
- 过滤
- 无状态
-

Databus For Oracle

介绍

databus 通过 oracle 的 ora_rowscn 虚拟列(不支持创建索引)来追踪数据行的变化。默认情况下 ora_rowscn 只支持块级别的数据更新记录,意味着当一行数据被更新时,当前数据所在块的其余数据行的 ora_rowscn 也会发生变化。可以通过重建表指定 rowdependencies 属性使 oracle 实现行级别的数据变动追踪，但是这会严重影响数据库性能。Databus 通过自身的设计实现在块级别下高效的数据变化捕获功能，同时为了保证数据变动的顺序一致性和可重放性,在一个时间周期内一行数据被多次改动时,databus 只记录其最终修改后的状态。

对数据源的改动

- 源表中添加名为 txn 的列,同时为此列创建索引以服务于原始数据查询语句
- 源数据库中会创建 sy$txlog 表(已经开启 rowdependencies),用来记录源库中的事务性变化,此表主要的列为(scn, txn, mask, timestamp)
- 源库中为每个表创建 before insert/update 触发器用来追踪事务
    - 触发器使用 databus 提供的函数 sync_core.getTxn()函数获取当前事务 ID 并且更新到目标表的 txn 列
    - 新增或更新 sy$txlog 中的记录,任何操作都会用 txn 保存获取的事务 ID,同时将 scn 设置为无穷大(9999999),并且使用 mask 字段来记录此次事务操作的目标数据
- 源库中为每个表创建对应视图以方便数据查询
- 创建 databus 运行所需的各种包、函数、存储过程
- 后台合并作业周期性的(默认 2 秒)操作 sy$txlog 表中 scn 字段值为无穷大的数据行,将其值设置为事务的真实 ora_rowscn

工作流程

以下是一个事务启动、提交时的工作流程。

- 事务 t1 开启,更新源表 S1 的行 R1 记录
- S1 表上 before insert/update 触发器触发
- 调用 sync_core.gettxn()获取当前事务 ID t1,并且将 t1 更新到 R1 记录的 txn 字段
- 向 sy$txlog 表添加一条记录 R2,它的 txn=t1,scn=INFINITY(9999999),mask 字段标明被更新的表为 S1
- 事务 t1 提交,源表 S1 的 ora_rowscn 被更新, sy$txlog 表中记录 R2 的 ora_rowscn 字段同时被更新为当前事务提交的 scn 号
- 后台合并作业启动,将 sy$txlog 中 scn 字段值为 INFINITY 记录的 ora_rowscn 值回写回 scn 字段。

注意事项

- 必须使用 SCN 字段而不是 TXN 字段对事件进行排序,原因是 txn 是事务启动时填充,scn 为提交时填充,数据的变化应该以事务提交为准
- 通过 sy$txlog 的 rowdependencies 来实现行级 ora_rowscn 准确

实现

sync_core 包

- 描述:databus 自定义的核心函数过程库
- 属性

名称	描述
lastTxID	the DBMS_TRANSACTION.LOCAL_TRANSACTION_ID of the lastest transaction; used in getTxn() to determine transaction boundaries
currentTxn	the txn column value in sy$txlog for the last updated/inserted row
currentMask	the mask column value in sy$txlog for the last updated/inserted row
source_bits	map from a databus source name to a source bitmask

- 方法

定义	描述
function getScn(v_scn in number, v_ora_rowscn in number) return number	当 v_scn 为无穷大时,则返回 v_ora_rowscn,否则返回 v_scn
function getMask(source in varchar) return number	Returns the bitmask corresponding to a databus source name.
function getTxn(source in varchar) return number	Denotes that source is being updated by a transaction. Updates sy$txlog accordingly. Returns the txn of the transaction.
procedure coalesce_log	更新 sy$txlog 表中 scn 与 ora_rowscn 值不同步的记录,因为 ora_rowscn 在事务提交后才生成。
procedure signal_beep	当 sync_core_settings 表的 RAISE_DBMS_ALERTS 字段值为'Y'时启用

unconditional_signal_beep	触发 sy$alert alert

sync_alert 包

- 描述:管理 databus 运行中的事件警告
- 属性

Variable name	Description
is_registered	Boolean: 标记当前 sy$alert 是否已经注册

- 方法

定义	描述
function registerSourceWithVersion(source in varchar, version in number) return number	将 source(源表)注册进 databus,当 waitForEvent 方法被调用时,如果当前 source 未注册则会返回 null,否则返回当前 source 字符串
procedure unregisterAllSources	清除所有 sources. 此操作会导致 waitForEvent 不再返回 event
function waitForEvent(maxWait in number) return varchar	定时执行(秒). 返回事件信息

sy$scn_seq 序列

- 描述:为 sy$txlog 表生成 txid
- 脚本位置:db/*/schema/ddl_seq.sh

sync_core_settings 表

- 描述:databus 配置表

Column	Description
RAISE_DBMS_ALERTS	当值为'N'时,会禁用 sync_core 包中的 signal_beep 函数会被禁用

sy$sources 表

- 描述:将源表映射为 bit 数字
- 目前最大支持 126 个源表映射,因为源表位掩码需要保存为数字
- 脚本位置: db/*/schema/ddl_tab.sh

Column	Description
name	the name of the Databus source
bitnum	the bit number corresponding to this source in the sy$txlog mask column

sy$txlog 表

- 描述:记录数据源的事务变化,由触发器和合并作业维护和更新数据
- 提示:此表开启了 rowdependencies 以实现行级别 ora_rowscn 追踪

Column	Description
txn	transaction id (generated from sy$scn_req)
scn	System Change Number; the ora_rowscn of the first change in the transaction
mask	a mask with the bits of all source that got updated in the transaction; see also sy$sources
ts	the transaction timestamp

J_COALESCE_LOG 定时作业

- 功能:调用 sync_core.coalesce_log 函数实现 sy$txlog 表的 scn 字段和虚拟 ora_rowscn 列数据的同步
- 周期:2 秒 REPEAT_INTERVAL => 'FREQ=SECONDLY;INTERVAL=2'
- 脚本位置: db/*/schema/ddl_prc.sh

J_CALL_SIGNAL 定时作业

- 功能:调用 `sync_core.unconditional_signal_beep` 函数实现事件追踪
- 周期:每秒触发 `repeat_interval => 'FREQ=SECONDLY'`
- 脚本位置: db/*/schema/ddl_prc.sh

触发器

- 功能:在 insert/update 发生前触发,生成和记录事务 txn 信息
- 作用域:每张表创建一个

```
CREATE TRIGGER T_DATABUS_TRG
     before insert or update on T
     referencing old as old new as new
```

```
        for each row
begin
        if (updating and :new.txn < 0) then
                :new.txn := -:new.txn;
        else
                :new.txn := sync_core.getTxn('T');
        end if;
end;
```

视图

- 功能:databus 中源表视图,方便 databus 进行数据查询
- 作用域:每张 source 表对应一个 sy$T, T 对应源表名称

```
create or replace force view sy$T as
  select
    txn,
    f1,
    f2,
    :
    :
  from
    T
```

数据查询

简单查询

- 功能:databus 正常的变化数据查询
- 说明
    - SOURCE_VIEW_NAME 视图封装的源表名称,参见视图章节描述
- 参数
    - sinceScn:relay/client 中设置的查询起始 scn 号

```
放大
SELECT /*+ first_rows LEADING(tx) */
     sync_core.getScn(tx.scn, tx.ora_rowscn) scn,
     tx.ts event_timestamp,
     src.*
FROM sy$<SOURCE_VIEW_NAME> src, sy$txlog tx
WHERE src.txn=tx.txn AND tx.scn > :sinceScn AND tx.ora_rowscn > :sinceScn
```

批量查询

- 功能:批量(固定条数)查询数据
- 描述
  - SOURCE_VIEW_NAME 视图封装的源表名称,参见视图章节描述
- 参数
  - sinceScn:起始 scn 码
  - rowsPerChunk:每次查询出的数据条数

```
SELECT scn, event_timestamp, src.*
FROM sy$<SOURCE_VIEW_NAME> src,
   (SELECT /*+ first_rows LEADING(tx) */
       sync_core.getScn(tx.scn, tx.ora_rowscn) scn,
       tx.ts event_timestamp,
       tx.txn,
       row_number() OVER (ORDER BY TX.SCN) r
    FROM sy$txlog tx
    WHERE tx.scn > :sinceScn
    AND tx.ora_rowscn > :sinceScn
    AND tx.scn < 99999999999999999999999999999) t
WHERE src.txn = t.txn AND r <= :rowsPerChunk
ORDER BY r
```

Databus For MySQL

Databus 可以从 mysql 的 binlog 实时读取数据变化日志捕获数据变化。Binlog 格式与设置详情可参考 mysql 官方文档。Databus 利用 mysql 的 binlog 数据同步备份机制读取 binlog 事件流并将其转化为 databus 数据变化事件,binlog 的解析使用了第三方开源库 OpenReplicator。项目地址 https://github.com/whitesock/open-replicator

实现

Mysql 需要开启 binlog,并且关闭 binlog checksum 功能。当前面向 mysql 的 fetcher 尚未部署到生产环境中。由于不同 mysql 版本的 binlog 格式会有差别,databus 通过 mysql 引擎 api 读取数据。Databus 使用 mysql 的 offset 机制来生成 scn,scn 是 64 位的 long 类型数据,类似于 twitter 的 snowflake 机制,前 32 位用来记录数据来源的 binlog 文件,后 32 位用来记录 binlog 中的偏移量(offset)。这个方式的缺点是 databus 无法保证从多从库同步数据时的一致性,原因在于 mysql 的 binlog 的 copy 并不能保证多个 slave 中同样数据的存储顺序,也就是说在不同机器上的相同的事务可能会在不同文件的不同位置,甚至相同文件名的不同位置,因此造成 scn 不一致,同时在容灾切换时甚至会丢失一部分数据。例如 reset master 命令清空服务器上现有的 binlog,因此当其余 slave server 没有清空时,此 server 的数据会产生不同步,而当 relay 从其余的 slave 切换至此 slave 时,很可能会丢失数据,因为此 slave 产生的 scn 会变小。

当前只支持单节点 mysql 数据读取,未来版本会添加对 mysql cluster 的支持。同时 mysql binlog 变化数据捕获和处理还可以参考阿里开源的 canal。