**DTS设计文档**

## 修订记录

|  |  |  |  |
| --- | --- | --- | --- |
| 修订版本号. | 修订内容简述 | 修订日期 | 修订人 |
| **1.0** | **概述，架构设计** | **2016.11.09** | **马进** |
| **1.1** | **Engines实现** | **2016.11.18** | **周劲松** |
| **1.2** | **总体设计** | **2016.11.22** | **马进** |
| **1.3** | **Center实现** | **2016.11.24** | **杨宇佳** |
| **1.4** | **Applier实现** | **2016.11.25** | **刘晨** |

## 评审记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 提交评审的修订版本号 | 评审意见/结果 | 评审形式 | 评审日期 | 评审人 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 第一章 概述

## 1. 应用场景

DTS全称Data Transfer System（暂定名），是网易大数据平台组计划研发的一套用于结构化数据传输和订阅的中间件系统，与DataStream一类的数据传输系统相比，它的特点是只作用于结构化数据，所以源端都是数据库，目标端可以是各种异构数据存储系统，未来DataStream用于结构化数据传输的DBCollector会慢慢退出历史舞台。

DTS可以且不限于适用以下几种场景：

1. 异构数据迁移

目前大多数应用对olap的解决方案是，用sqoop定期将线上的oltp业务数据库全量或按照某种过滤条件，导入到hive，hadoop或hbase中，这种做法一方面导入数据很慢，另一方面在导入数据的时段，一般每天凌晨，会对业务库产生巨大压力，并且造成网络拥塞。

而且，随着impala，kudu等更好的olap系统崛起，sqoop这类工具将不再实用，我们迫切需要一套对业务库压力小，实时性强，迁移过程平滑的结构化数据迁移解决方案。

1. 同构数据同步

异构数据迁移是将结构化数据从数据库中实时导入到olap数据仓库，目的是基于业务做大数据分析，查询，挖掘有价值的信息。还有一种实用数据迁移场景，用于异地灾备，多活，这种场景下，异地部署的业务一般不会变，因而底层数据库同构居多，我们将这种实时数据迁移的场景定义为“数据同步”，相比于数据迁移，数据同步对端都要支撑oltp业务，因而对实时性和一致性要求更高，再加上异地网络的不可控，需要压缩算法降低传输延迟，数据库如mysql自身提供的主备同步方案往往不是最优选择。

除了数据同步的实时性和一致性之外，对于一些复杂的异地多活案例，还需要解决星状拓扑和数据回环问题。

3．结构化数据订阅

数据库的目的是数据存储和持久化，它可以被动接收应用各种查询请求并予以回复，却不能主动告知外界自身状态的变化，数据订阅功能可以很好地弥补数据库的这个短板。结构化数据订阅**（以下简称数据订阅）**是指将数据库中每时每刻产生的数据增量变化实时拉取出来，主动交付给相关业务方，一般以SDK的方式提供服务。结构化数据订阅可以实现缓存异步更新，数据驱动的业务解耦等，例如在电商应用中，通过订阅订单数据，可以将下单流程中的库存，物流业务抽离出来，降低业务耦合度。

目前我们对DTS的需求主要在**异构数据迁移和数据订阅**上，数据迁移可以打通MySQL，oracle到各种olap系统的壁垒，同时也可以作为DDB 做数据重分布，在线扩缩容，在线修改表结构的底层依赖。数据订阅可以给云音乐，考拉这类大应用做业务治理：将数据驱动的业务线隔离出来，或者通过订阅数据库增量数据实现多机房缓存更新和淘汰。

## 2. 设计目标

目前DTS主要服务于网易内部产品，数据迁移功能作为BDMS，DDB的依赖组件，数据订阅功能开放给网易内部应用做缓存更新淘汰和异步业务解耦。

由于DTS会给多个系统做支撑，不可避免的会涉及到适配问题，我们希望DTS系统本身可以作为一个独立系统存在，有自己的运维工具，平台管理，节点管理等模块，通过RPC或http接口向BDMS或DDBAdmin提供服务。BDMS中集成DTS的管理界面，DDBAdmin将DDB的扩容缩容，在线修改表结构等操作转换为一个个单机数据库的迁移任务，并向DTS系统发起请求。

同时，DTS也要支持多租户，但是接入哪套租户认证还有待商榷。

DTS中包含的基本组件有：

* 管控模块/节点：节点管理，任务调度，元数据管理
* 执行模块/节点：迁移，订阅任务的具体执行模块
* Dashboard：可视化运维工具
* SDK：数据订阅组件

## 3. 功能需求

目前设计中，DTS支持数据迁移和数据订阅两类功能。源端先支持MySQL和Oracle，目的端先支持Kudu, MySQL, Oracle和DDB。

对外，DTS通过Dashboard以web的形式为产品管理员提供各项管理功能：

1. **租户认证**：由于DTS是面向多租户的，需要接入云计算或猛犸的账号系统，并将各个迁移和订阅任务按照租户统计和管理。
2. **任务调度**：管控节点负责迁移和订阅任务的创建，预检查和分发到执行节点。目前DTS要求迁移任务必须有一个源端实例和一个目的端实例，迁移和订阅对象以表为单位，一个任务可以包含一个源端实例内的多个表对象。
3. **任务监控**：迁移或订阅任务在执行过程中，会定时向管控节点上报任务执行状态，以及任务执行过程中产生的各个状态值，如MySQL增强迁移时源端的binlog位置点。DTS对任务的监控要多维度的：除了不同迁移过程，如全量和增量会产生不同的状态值之外，对每个迁移对象表也会分开记录状态值。
4. **任务管理：**用户可以通过管控平台将创建任务，或将迁移和订阅任务暂停，启动，停止或释放，停止后的任务再也无法启动，只会在系统中保留任务最后时刻的状态值，释放后的任务会彻底从DTS系统中删除。
5. **动态更改迁移对象**：无论是迁移任务还是订阅任务，都要求DTS具备动态更改迁移对象的功能，即可以在迁移或订阅任务执行过程中，增加或减少迁移对象表。

除了面向产品管理员的各种功能外，DTS还需要具备面向平台管理员的管理功能，主要是执行节点的管理和监控：

1. **Engine管理：**查看所有执行节点和相关状态信息，执行节点优雅下线，释放等。平台管理员功能，不按租户区分
2. **SDK管理：**查看订阅任务关联SDK节点列表，以及相关状态信息

## 4. 非功能需求

除了功能性需求外，NTS3.0对非功能需求的考量如下：

1. **管控高可用**：管控模块一般是中心组件，可采用冷备实现高可用
2. **任务高可用**：在执行节点宕机，或发生网络分区时，通过管控节点将任务迁移到其他执行节点重试，实现任务的自动take over

## 5. 竞品调研

网易DTS的竞品主有Oracle golden gate（以下简称OGG），阿里Otter，阿里DTS。

OGG是一个典型的数据迁移软件，是oracle为异构数据迁移场景提供的企业级解决方案，优点是支持的异构系统很多，缺点是不开源，没有平台化，部署和使用方法复杂。

Otter是阿里B2B开源的一套数据同步平台，对远距离，如中美机房的数据同步场景有较多的优化，只支持MySQL和Oracle。优点是有平台化，缺点是概念很多，使用方法较复杂，不支持全量迁移。

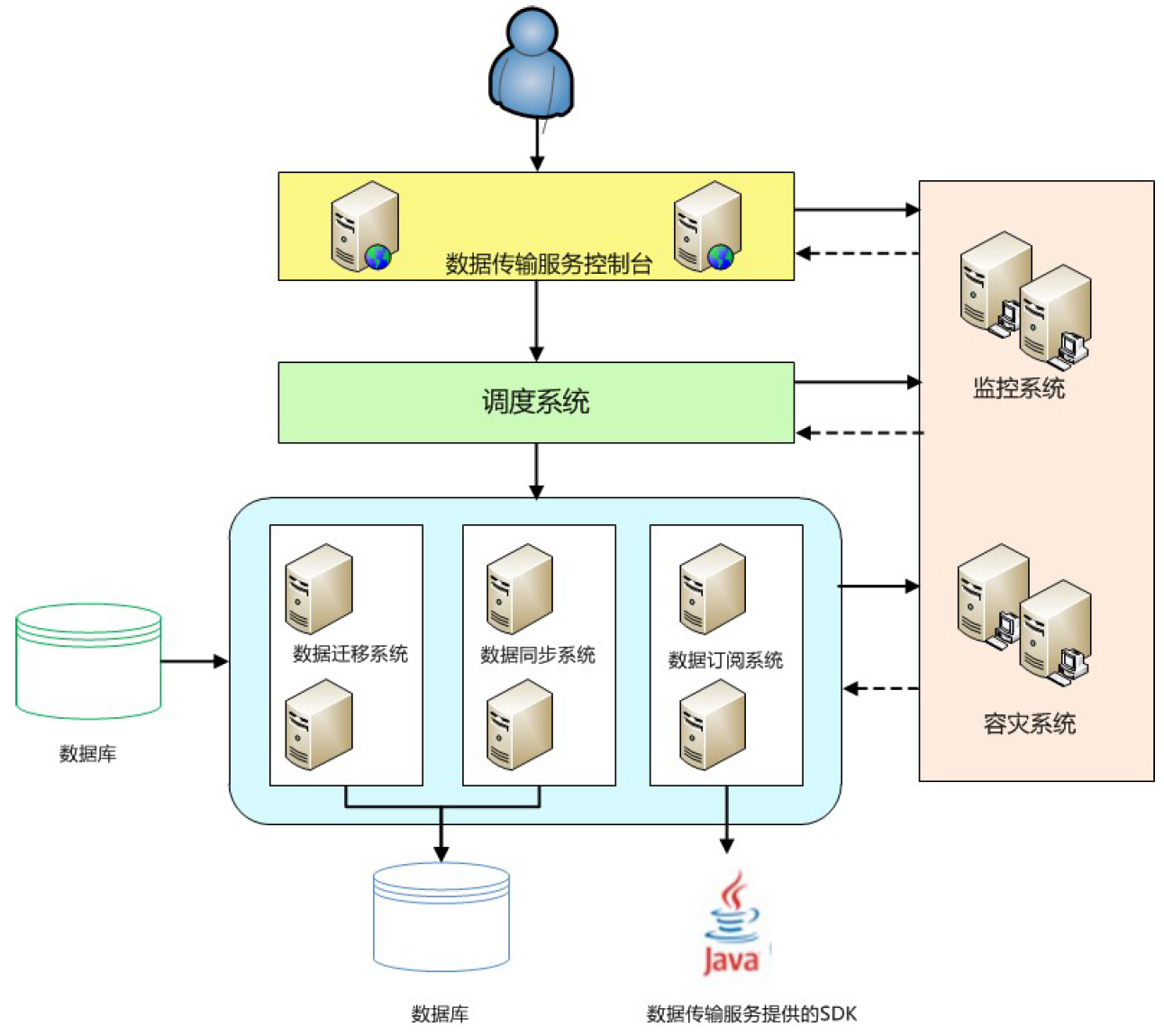
阿里DTS是阿里云在云端提供的一套数据迁移，数据同步和数据订阅的SaaS服务，必须承认阿里在数据库，数据传输领域已经走在了很靠前的位置。网易DTS在产品概念和形态上有很多借鉴于阿里DTS。阿里DTS应该是最早将数据迁移，数据同步和数据订阅三种结构化数据传输场景严格定义并提供相应云服务的供应商。



# 第二章 架构设计

网易DTS目前的设计目标首要是为网易内部产品提供数据迁移和订阅服务，但并不排除未来通过蜂巢提供公有云服务，在这方面，阿里DTS已经走在了行业前面。阿里DTS的产品定位与目标用户与我们的DTS基本一致，不同的是阿里DTS支持数据同步功能，并且支持的迁移和同步目标仅限于阿里的各类数据存储系统。而网易DTS虽然在设计上源端仅支持MySQL和Oracle，目标端支持的系统也没有阿里DTS多，但网易DTS旨在帮助应用解决实际的数据迁移和订阅问题，未来会根据应用实际需求支持更多的系统。

作为标杆性的竞品，我们先来看下阿里DTS的架构：



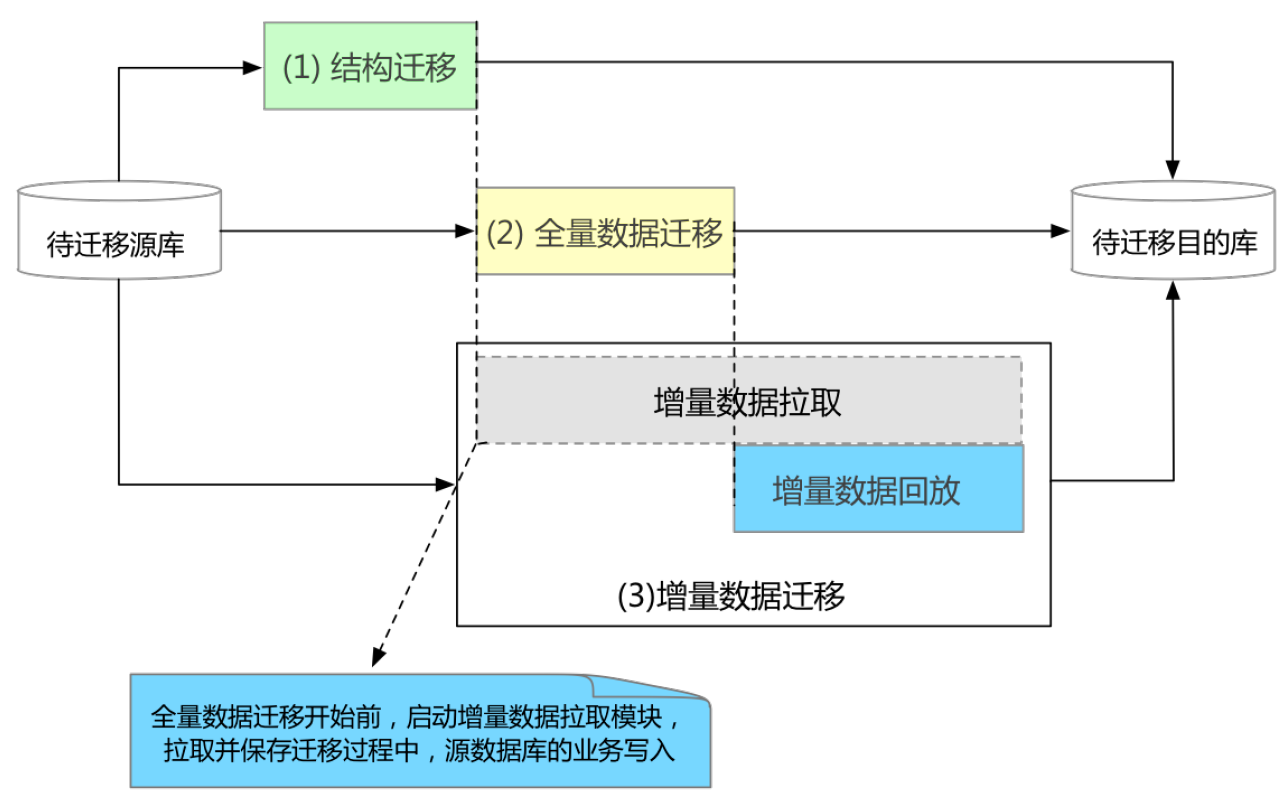
与真实的架构图相比，上图更像是一张概念架构图，即便如此，我们也可以看到阿里DTS和网易DTS在基础模块上的划分非常类似，我们有专门的Dashboard作为DTS的管理工具，与图种的“服务控制台”相对应，我们设计了center模块作为调度中心，同时也是整个系统的监控中心，作为中心模块，我们未来会用ZK一类的分布式协调组件实现高可用，这样center对应了“调度系统”和“监控系统”，ZK对应了“容灾系统”，图种的数据迁移子系统和数据订阅子系统分别对应了两种不同类型的engine。综上，上面的概念架构图完全适用于网易DTS，下面将分别介绍网易DTS中的各个模块和术语，数据迁移和数据订阅的基础架构以及全系统架构。

## 1. 术语与模块

|  |  |  |
| --- | --- | --- |
| 名词 | 解释 | 使用类型 |
| Engine | 执行节点统称，一个执行节点可执行多个任务，为了方便管理，同一个Engine不能同时执行迁移和订阅任务 | Both |
| SDK | SDK通常是指数据订阅中应用获取增量数据依赖的JAR包，在本文档中也可能指依赖SDK订阅数据的应用节点 | 数据订阅 |
| Center | DTS中的管控节点，接收所有Engine的状态上报，负责所有任务的管理和调度。 | Both |
| Dashboard | DTS的web运维工具，可以在Dashboard中创建任务，更改任务对象，查看任务各个对象，各个阶段的状态值等 | Both |
| 对象 | 本文档中的对象一般是指迁移和订阅对象，**DTS中以表为迁移和订阅单位**，这一方面是大部分应用场景所要求的，另一方面方便以表为维度做统计监控 | Both |
| Pipeline | 一个对象的迁移或订阅过程被称为Pipeline，Pipeline是任务状态统计的X维度，一个任务可以包含数十数百个Pipeline不等 | Both |
| Process | 一个任务虽然可以包含多个对象，但多个对象同时只会处于同一个过程，如所有迁移对象都处于全量复制过程，或增量复制过程，这些不同的过程抽象为Process的概念，Process是任务统计监控的Y维度 | Both |
| 结构迁移 | 指源端schema的迁移，即DDL的迁移。结构迁移同时作用于全量复制过程和增量复制过程 | 数据迁移 |
| 全量复制  /迁移 | 将源端的所有存量数据迁移到目标端的过程，全量复制默认串行执行，按照配置对象的顺序将源表逐个迁移到目标表，可以配置每批迁移的行数 | 数据迁移 |
| 增量复制  /迁移 | 将源端的增量数据实时迁移到目标端，或将源端的实时更改从元旦抽取出来，交付给订阅端。在迁移场景中，增量复制一般指迁移全量迁移过程中新产生的数据变更，在迁移不停止的情况下，增量迁移将源端的实时更改应用到目标端。而我们所说的数据订阅，专指订阅增量数据 | Both |

## 2. 数据迁移

数据迁移一般包括结构迁移，全量迁移和增量迁移三个过程（参考上节术语解释），以源端MySQL为例，三者的关系大致如下所示：



在一个完整的迁移过程中，执行节点会先将源端所有需要迁移的表结构在目标端中创建，即结构迁移，结构迁移完成后，会先启动增量数据的拉取模块，再开始全量迁移，这样做是为了保证全量迁移过程中的增量更新不会丢失。

全量迁移的原理是单个线程分批从源端拉取存量数据，通过目标系统的驱动将存量数据插入到目标端中，目前全量迁移提供的配置项比较简单，只有一次批量拉取数据的数量。而在全量迁移过程中可能出现两种情况：拉取对源端压力过高，或迁移过程过慢。对于前者，可以增加一个批量拉取的时间间隔配置项，减轻对源端的压力；对后者可以增加一个并发拉取的选项，以加速全量迁移速度。这两种配置都需要对业务情况做全面权衡，而单线程不间断全量迁移是一种普遍接受的做法。

与全量迁移相同，增量迁移也需要将增量数据从源端拉取到本地，再通过目标系统驱动应用到目标端中，不同的地方在于：增量数据拉取模块对不同的源端系统可能有截然不同的实现，例如对MySQL而言，增量拉取就是dump binlog，使用MySQL特有的通信协议，而对Oracle而言，可能是通过JDBC将CDC中的数据定时拉出来；其次，增量迁移中数据回放和数据拉取必须异步进行，因为源端的增量拉取往往比回放速度快很多，如果因回放速度慢而导致源端增量日志无法及时消费，可能影响源端的服务稳定性。另外，为了实现异步回放且不会OOM，拉取模块会把增量数据落盘，回放模块读盘来异步消费。DTS默认采用了Hamal的并行复制算法，用户在创建含有增量迁移的迁移任务时，需要指定增量迁移的并发度。

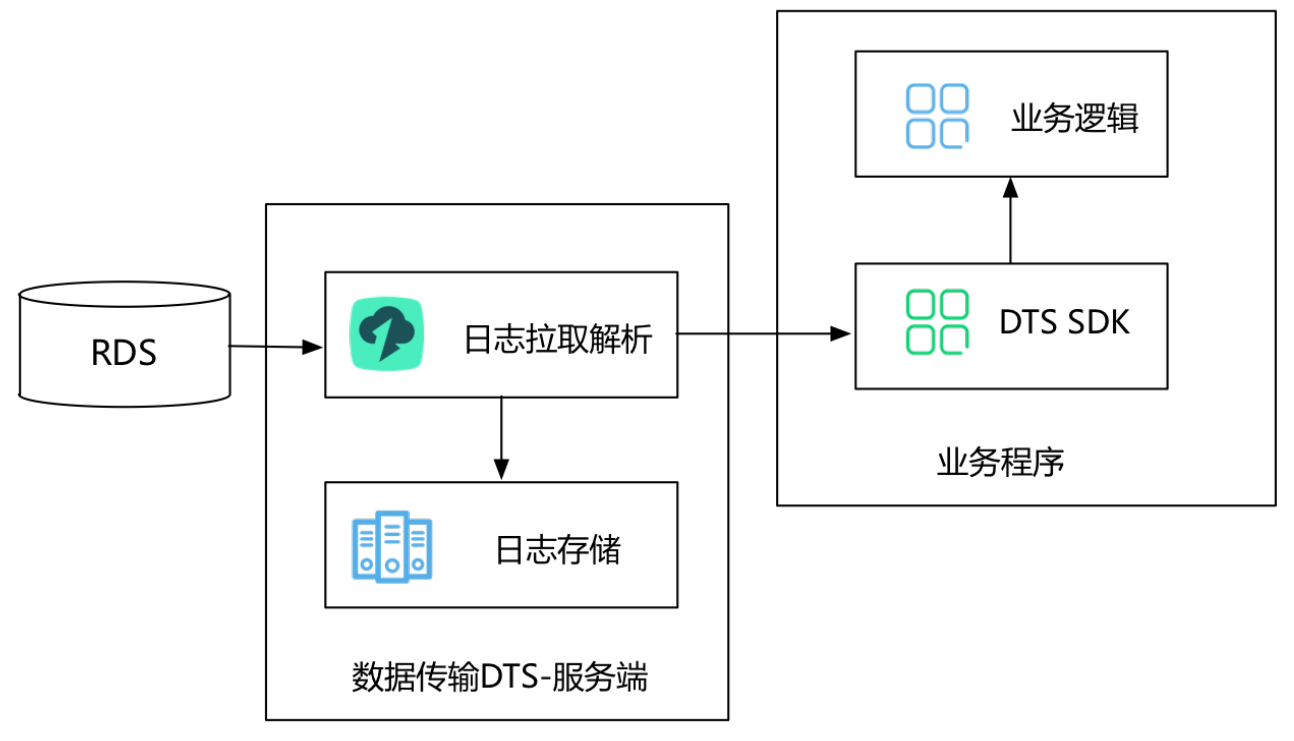
在上一节中有提到，结构迁移同时作用于全量复制和增量复制过程，所以结构迁移不能算作一个Process，而是作为全量迁移和增量迁移过程中需要handle的一个特性。全量迁移中，结构迁移是在真正的全量迁移开始之前完成，增量迁移中，engine要确保每个结构迁移必须发生在相关结构事件迁移之前。

考虑到结构迁移的业务复杂性：传统的DDL会锁表，而pt-osc一类的在线修改表结构机制也无法立即生效，这种结构迁移极容易造成整个迁移过程不可控。且结构迁移在一些没有schema的系统中是一个伪命题，例如HBase。所以我们建议前期先将结构迁移交给运维人员处理，这样一方面规避了结构迁移自动化给系统带来的不可控因素，另一方面也允许了schema不同但兼容的情况出现（当然也可以通过提供一个是否结构迁移的选项来实现）。

DTS一期在创建一个任务时可以选择是否全量迁移和增量迁移，系统允许对一个迁移任务只做全量迁移或增量迁移。

## 2. 数据订阅

数据订阅是将源端数据库的增量数据拉取出来，并通过DTS的SDK实时交付给应用端进行定制化处理，它的基础架构如下图所示：



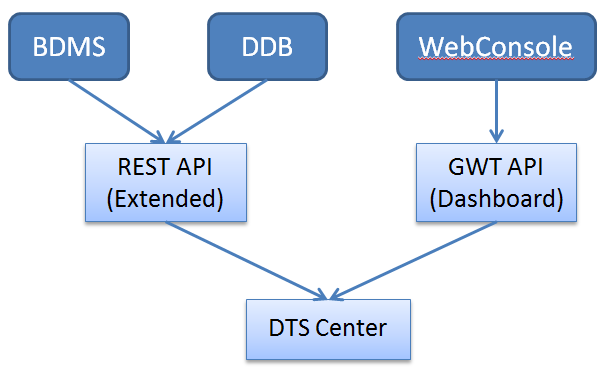
数据订阅与数据迁移的增量迁移过程类似，不过数据订阅的Apply过程是将增量日志转换为消息丢进消息队列，交由SDK处理。

由于数据库的增量更新必须保证有序，所以选用的消息队列必须可以保障消息的有序性，而下游消费数据的SDK同时只能有一个接收消息，否则可能出现一条数据先后的更新操作由两个SDK节点接收，顺序难以保障。

目前的方案选型是用kafka的单partition策略，日志拉取后是否保存在engine本地可选。

## 3. 可视化运维

DTS为用户提供WEB版Dashboard作为可视化管理工具，用户可以通过Dashbaord创建，监控和跟踪每个数据迁移和数据订阅任务。Dashboard是DTS独立于BDMS和DDB的工具，是DTS独立的入口，而DTS在作为一个独立服务的同时，也是大数据平台BDMS和DDB管理工具做数据迁移的依赖组件，为此我们需要为BDMS和DDB提供方便适配的REST接口。



如上图所示，我们在开发Dashboard时选用了vaadin作为WEB开发框架，vaadin内嵌了很多GWT的控件，与纯GWT相比，vaadin的强大之处在于是全JAVA开发，不用编辑html，类似于swing开发窗口程序，是前后端一站式开发的利器。而GWT与REST完全两种风格，这里我们把他们作为并列的组件。REST API为BDMS和DDB提供REST接口来适配（DDB由于我们自己维护，可以和Dashboard一样通过RPC与center交互），GWT API则以Dashboard的形式为DTS提供独立的管理服务。

在实现上，REST API和GWT API可以置于一个容器中。

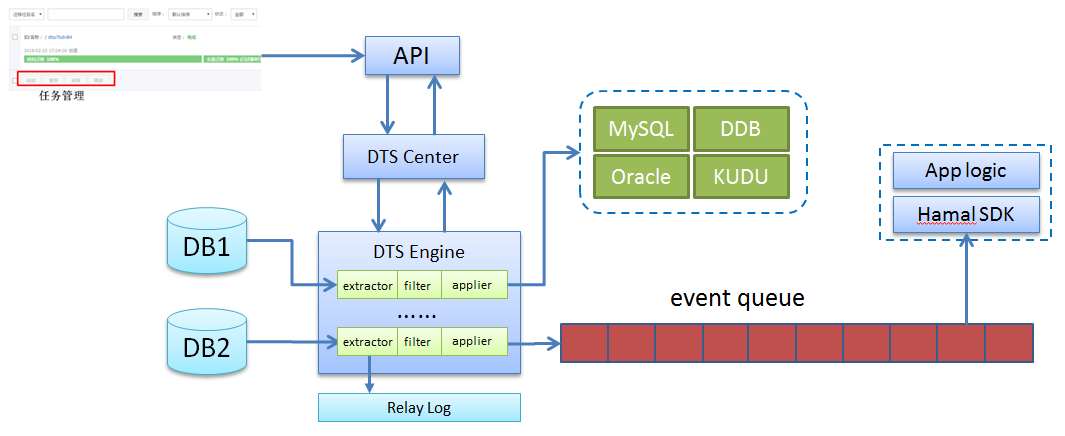
## 4. 平台接入

DTS未来可能会接入不同平台，不同平台有不同的用户认证服务，比如BDMS和蜂巢就是两种不同的用户体系。所谓平台接入就是接入不同的用户认证体系。

为了最大限度保持DTS的独立性，我们把DTS的用户认证放在API层，而不是在center中，center中的用户统一以int类型标志，并存储用户以及相关的平台信息，以供查阅。而实际的用户认证操作放在API层去扩展。这样平台接入只需要在API层进行适配，而不需要更改center和engine。

## 5. 总体架构

DTS的整体架构如下图所示：



API模块接收来自WEB或其他系统的请求，创建任务时，API节点收到请求后通过RPC的方式转发给center，center负责将任务描述信息记录到元数据库中。启动任务时，center先执行一些固有的预检查，如网络连通性检查，用户权限检查，然后选择合适的engine节点将下发。engine启动相应任务后，会定期将任务运行时状态上报给center，center再将这些运行时信息保存到元数据库中，以便center重启后可以从元数据库中恢复所有job信息。

Center作为DTS的中心模块，未来会使用ZK或keepAlived一类的高可用组件以冷备方式实现高可用。

# 第三章 总体设计

在DTS的整体架构中，center节点作为配置中心和调度中心，具有集群管理，任务管理和任务调度三大功能。

集群管理又叫节点管理，包含响应SDK和Engine节点的注册请求，维护节点元数据信息和状态信息，通过Dashboard向用户提供节点统计信息以及优雅下线等功能。

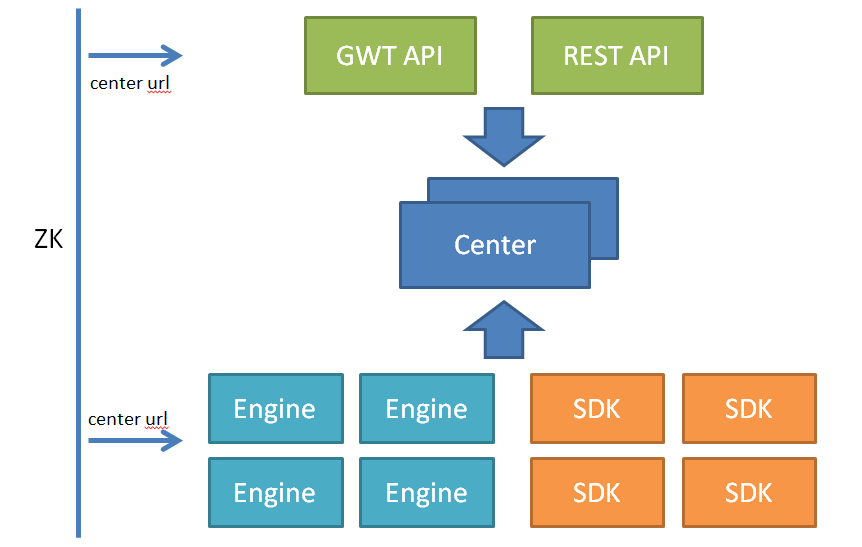
任务管理主要是指任务元数据管理和状态管理，由于任务在运行过程中的状态实际由各个Engine节点和SDK节点维护，center需要定期向各个执行节点收集任务状态并持久化在系统库中。

任务调度发生在两个过程中，第一个过程是任务启动时，center会根据所有Engine的负载情况选择节点下发任务，第二个过程是在发现Engine节点宕机，或执行优雅下线时，center会将目标节点上的所有任务take over到其他Engine节点，实现任务高可用。

本章前三节将分别就集群管理，任务管理和任务调度三大功能展开讨论，后三节分别描述数据迁移和数据订阅的核心流程，以及DTS的异常处理机制。

## 集群管理

Center是DTS的配置中心，所有角色节点（SDK和Engine）都需要向Center注册，状态上报和同步元数据。Center的元数据信息，包括节点信息和任务信息都持久化在一个MySQL数据库中，Center启动后会首先从这个DTS系统库加载所有元数据。如下图所示：



为了实现管控高可用，我们会为Center部署一到两个冷备，通过ZK实现主备切换。所有角色节点和API节点通过ZK获取最新的Center地址来注册和状态上报。

角色节点和Center之间通过租约机制来实现注册，状态上报和元数据同步功能。具体流程如下图所示：



这里重点对注册（建立租约）和状态上报（续租）这两个过程展开讨论。

注册的过程又叫建立租约，由角色节点通过RPC向Center发起，Center接到角色节点的注册请求后做三件事：

1. 认证，主要针对SDK节点，会对SDK的租户进行必要的认证操作。
2. 生成节点元数据，Center首先会在已有节点列表中查找IP和port相匹配的元数据，若没有，说明是节点第一次注册，Center将节点类型，IP，端口以及时间戳等信息打成一个元数据结构置于节点管理的模块中，并记录系统库。
3. 建立租约，Center为该节点生成一个全局唯一的sessionId（UUID），并返回给注册节点，之后该节点所有续租行为都以这个sessionId作为凭据。Center可以通过从内存中删除相应sessionId来强制节点重新注册，以刷新全局配置信息。

从流程图中可以看到，角色节点在注册后会立即发起一次配置信息同步，也就是说，一次成功注册可以触发一次元数据同步，DTS通过这个机制来实现实时性要求不高的配置信息同步，对于实时性要求高的同步，由RPC来实现。

节点状态上报也是节点续租的过程，以sessionId和可持久化的状态信息为参数。Center收到续租请求后，首先验证sessionId是否与内存中保存的一致，一致的情况下会将节点上报的状态信息更新到内存和系统库中，否则返回角色节点续租失败，让节点重新进入注册流程，以拉取最新配置信息。Engine节点上报的状态中包含了任务状态信息，以实现Center的任务管理功能，这部分内容下一节展开介绍。

节点的状态管理也通过租约机制实现，角色节点状态至少有alive和dead两种，在注册和租约超时转换：在节点完成注册后，状态更新为alive，而这个节点的租约超时，即节点超过一段时间没有续租成功后，节点状态更新为dead，租约超时的时间一般为节点续租时间间隔的3倍。

总体来说，租约机制是DTS集群管理的核心，业界有许多系统实现了租约机制，比如ZK通过租约机制监控节点状态，通知事件变更，hadoop通过租约机制实现DataNode的任务管理和调度。

## 任务管理

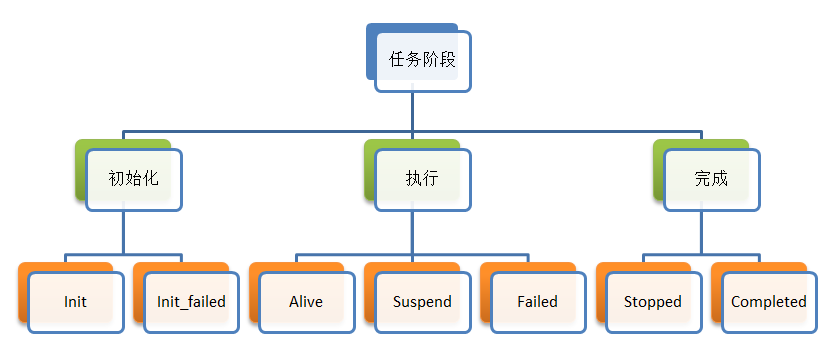
DTS任务管理包含以下几点：

1. 任务创建：对应center生成一项任务的元数据信息，并持久化到系统库中
2. 启动任务：启动任务后，Center会先对源和目标库进行一边预检查，预检查内容包括网络连通性，用户权限，白名单，库表是否存在，数据是否兼容这五项。预检查完成后由center选择节点调度任务，任务进入Engine后开始执行。
3. 悬挂任务：任务正常执行状态中，可以通过Dashboard暂停任务，也叫悬挂任务，悬挂后的任务依旧占用Engine资源，但是不会继续执行任务。这里需要注意的是，不只有悬挂任务这项操作才能让任务暂停执行，若任务执行过程中遇到需要人为干预的错误，例如网络长时间不可达，则会进行失败状态，失败状态下的任务也会暂停执行，区别在于失败任务的状态比悬挂任务多了错误码和错误信息。
4. 恢复任务：任务在暂停状态下，可以通过恢复任务这项操作让任务继续执行，暂停状态包括悬挂和失败两种情况。一般如果任务执行出错，在解决掉引发错误的问题之后，可以恢复任务让其继续执行。
5. 停止任务：在使用“停止任务”这项功能后，任务会从Engine节点删除，并将任务状态更新为Stopped，停止后的任务永不可恢复，系统只会为任务保留元数据信息以供查阅，若希望任务可恢复使用悬挂功能。
6. 更改任务：可以在动态为任务添加或删除表对象，注意能动态更改的只有表对象，源端和目的端只有在任务初始化阶段可以更改。
7. 释放任务：释放任务对应着任务从DTS系统中永久性删除。

任务的生命周期可分为三个部分：

初始化—>执行—>完成。

其中每个部分又有各自的状态，如下图所示：



其中Init对应创建任务后，未启动任务前的状态，Init\_failed表示预检查失败或下发任务失败的状态，Alive表示任务下发后，在Engine节点中正在执行的状态，Failed表示执行过程中出现了无法自动恢复的异常状态，Suspend表示用户手动悬挂的状态，Completed表示任务完成执行，并退出Engine的状态，Stopped是用户通过Dashboard手动停止任务的状态。由于创建任务，预检查和调度任务都是在Center中完成，Init和Init\_failed状态由Center维护，Completed和Stopped状态下任务会退出Engine，其状态也是由Center维护。其他执行状态由Engine节点维护，并通过租约机制定期上报给Center。

状态机描述请参加Engine详细设计。

Center中会实现一套状态补偿机制，状态补偿主要作用于任务执行时，对任务的悬挂，停止等操作会先把任务的状态更新为对应的悬挂和停止状态，但是任务最终的状态以Engine上报为准，例如用户向悬挂一个任务，Center在向Engine执行悬挂任务的RPC操作时抛了网络异常，这时任务可能已经悬挂成功，可能没有。界面上虽然会提示悬挂任务失败，但经过最多一个续租等待的时间后，任务的状态会更新为Engine中实际的任务状态，这种情况下界面上可能会显示任务悬挂。

单个任务的状态会从两个维度统计，Process维度和Pipeline维度，具体到数据迁移任务，就是全量迁移和增量迁移各维护一套任务状态，比如全量迁移状态有迁移进度，已迁移表，也就是Pipeline的数量，全量迁移时间，预估结束时间等，增量迁移状态有迁移延迟，源端增量日志位置，本地缓存大小等。Pipeline维度就是各个表对象的状态值，例如某个表已经完成的全量迁移数，某个表增量迁移中apply的insert事件等。两个维度的的状态由Engine通过租约机制上报，然后持久化系统库，可以在Dashboard中查询。

## 任务调度

DTS任务调度的原则比较简单：以各个Engine本地配置的最大任务数，和当前Engine正在执行的任务数这两项指标作为调度依据。调度过程首先会将所有活跃状态节点按正在执行的任务数做升序排序，然后按照次序选择没有到达最大任务数的Engine节点作为本次任务下发的节点。若当前没有满足条件的Engine节点存在，则抛异常，若选择的节点调用RPC下发任务失败，则按照之前的排序原则重新选择其他节点下发，最多重试两次，不成功则抛最后RPC异常。

有一种特殊情况是在Engine与Center失联一段时间后，租约过期，节点状态更新为Dead，而这个节点上所有任务需要判断是否take over，而take over的过程是将任务重新选择节点调度。Engine失联有两种情况：宕机和网络分区，对宕机情况，take over是必要的，但是对网络分区，Engine上的任务依然在执行，若take over，则一个任务可能同时在两个Engine中运行，造成不必要的资源浪费。

这里我们的做法是在take over前去源端延迟（5-20s）判断一下拉取线连接是否健在，若不健在，才会执行take over，流程如上图所示：



## 数据迁移流程

数据迁移，以MySQL的数据迁移为例，按照顺序包含以下步骤：



在Engine收到下发任务后，首先启动增量拉取binlog的线程，拉取binlog线程会将源端的binlog在本地生成冗余的relay log，以供后续的增量迁移消费，也就是说，拉取binlog和消费binlog做增量迁移，是一个异步的过程（详细解释见架构设计）。

启动binlog拉取线程后，开始执行全量迁移。全量迁移的过程是不断从源端select一批数据，insert到目的端。每次scan数据的起点（不包含）是上一次select结果的最后一个主键值，终点是开始全量迁移前获取的最大主键值（包含），每次select带有limit限制，默认1000，前端可配。

需要注意的是，全量迁移由于没有使用事务执行所有select，各个批次的scan不具有一致性读的特性，换句话说，全量迁移的过程可能会引入不定量的增量内容。这是我们故意为之，因为将所有select放在一个事务中会导致源端回滚段不断累积，影响巨大。而即便全量迁移中带入了增量的内容，只要在增强迁移时保证数据迁移具有幂等性，最终得到的迁移结果是一致的。

为此，增量迁移的幂等性是我们必须保障的一点，我们增量迁移的insert和update事件采用replace语句拼装，delete事件依然使用delete语句，这样能够保证同样的CRUD事件应用多次，得到的结果必然是一致的。

另外，对结构化数据到目标端的迁移，我们构造了基本的并行复制框架，基于有向无环图的数据结构来表达多个增量事件的依赖关系，在图种入度为0的事件节点都可以并发执行，这个设计是从原先Hamal数据迁移工具中保留下来的，吞吐率大概是MySQL官方并行复制的5-10倍。

## 数据订阅流程

略（二期完成）

## 异常处理

DTS中，任务执行的异常分为两种：可恢复的异常和不可恢复的异常。

可恢复的异常是指发生网易瞬断或迁移超时（拉数据导数据）时，通过重试可以恢复的异常，这类异常在Engine只需记录本地日志。不可恢复的异常是指在任务执行过程中，在一段时间内多次重试后依旧无法恢复的异常，比如源库目标库发生宕机，需要人工介入的情况，任务状态会由Alive转变为Failed，并触发报警。

可恢复和不可恢复的区别仅在于在有限时间内，有限的重试次数下，任务执行是否可以自发恢复。同一类异常可能是可恢复异常也可能不可恢复，如网络断开异常，若网络很快恢复，则为可恢复异常，否则为不可恢复异常，需要报警和人工介入。

另外，在任务由于不可恢复异常进入Faild状态，Center会定期尝试重新启动这些任务，这样可以在运维人员无法及时介入的情况下，尽可能让任务自动恢复。也就是说，Alive状态下的异常重试在Engine中进行，Failed状态下的重试在Center中触发，一般Center中的重试时间间隔是Engine中的5-10倍。

报警由Center完成。

# 第四章 DTS Center实现

## 1. 概述

DTSCenter是DTS集群的管理中心，功能包括节点管理（节点在内存以及数据库中的状态维护）、租约管理（节点的注册、过期、续租以及状态上报）、任务管理（任务的状态维护、任务新建、调度、启动、停止等操作）、用户管理（暂无），同时还负责为 API节点提供后台支持。

Center在收到用户创建任务的请求后，对任务的原信息进行一些预检查，预检查通过后基于一定的负载均衡算法选择一个Engine将任务下发，此后Engine将定时上报自己负责运行的任务的运行时信息给Center，同时用户对任务的修改、暂停、恢复、停止等操作也是如此。Center还需要负责对engine的上报的job状态进行管理，进行持久化处理。在重启时候，通过读取数据库来恢复集群中的任务状态信息，这其中包括一些异常情况下的状态补全以及修正。

## 2. 模块划分

从对外的接口到数据持久化看， Center的模块划分可如下图所示：



RPC层提供以RPC的方式向外部提供相应操作接口，包括集群中各个节点的管控操作以及用户的各种任务管控操作。实际业务逻辑层主要分为三大块：租约管理、节点管理、任务（job）管理。最后通过MyBatis将数据持久化到系统数据库。

## 3. 租约管理

DTS Center中所有节点的信息的有效性都以session为依据，session过期，则节点视为租约过期，相关的数据则失效，如节点配置信息以及该节点原处于运行状态的任务。而session过期的逻辑表现为Center内存中是否有该sessionId的记录。

对于Center来说，由于节点与Center租约的改变，可引发如下三类session变更（SessionEvent）：

* CONNECTED: 新建session，node在session过期后注册或者configserver重启。
* DISCONNECTED: session过期，在设定的session过期时间内，未收到node的探活。
* RESET:session重置，node在其session有效期内重新注册（node重启或者网络抖动）。

对于发起注册的节点来说，Center从节点管理模块中获取一个与（IP:port）对应的Node对象。如当前IP:Port未出现过，则该节点为一个全新的node，无租约信息。否则则该节点会映射到Center中的某个历史node，携带原来的租约信息（session）。若无租约信息或者原租约已过期，则为Connect事件；若原租约信息仍有效，则为Reset事件。DTS中对session事件敏感的模块称为sessionWatcher，在发生SessionEvent时候，SesssionManager会通知watcher这些session事件，让其自行处理。

租约管理模块还需要对当前注册的node进行session检测，若session的 touchTime间隔超过设定的过期阈值，则判定节点 session过期，租约失效，从sessionMap中移除并产生相应的SessionEvent. DISCONNECTED事件，发送给相应的sessionWatcher。

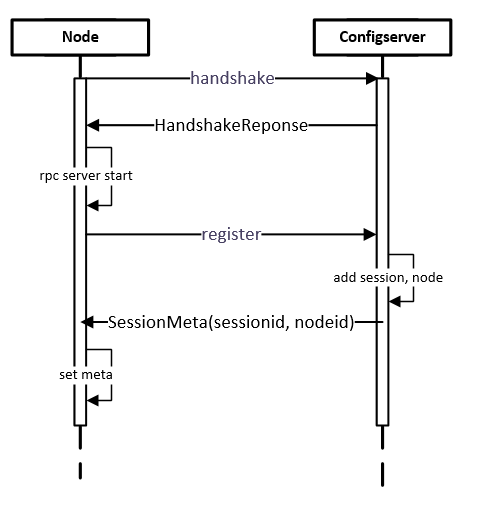
同时，由于center宕机后的engine状态并不能被center感知，因为重启时候加载的alive的engine可能已经dead。租约管理还需要检查可疑节点是否在有效租约期内。

## 4. 节点管理

节点管理包含响应SDK和Engine节点的注册请求，维护节点元数据信息和状态信息，通过Dashboard向用户提供节点统计信息以及优雅下线等功能。Center中又ClusterMananger 负责节点管理。

Center的节点元数据信息，都持久化在一个系统数据库中，Center启动后，节点模块从系统库加载所有节点信息，恢复节点状态信息。Center运行中，节点管理模块通过监听SessionEvent. DISCONNECTED事件来更新节点状态为Dead。对于节点管理来说，（ip:port）与nodeId是一对一的，这保证了同一engine重启后能对其重启前的状态进行追溯以及整理。

以新node的注册为例，说明下注册流程：



## 5. 任务状态

DTS中任务的生命周期可分为初始化—>执行—>完成三个阶段，其中各个阶段又包含不同的任务状态，具体说明可参见总体设计中关于任务管理的说明。用户对任务的主动操控以及engine中job的执行流程都会改变任务的状态。Center通过收集job的状态进行分析，可维护任务的运行状态给前端用户，对job状态进行持久化，同时job状态的改变也会影响到任务调度中的权重状态。

对于Center而言，能引发job状态更改的事件有两类：engine上报jobruntime列表、engine失联。Engine在每次向Center续约时候，会附带其上正在运行的job runtime信息列表，通过对比记录的该engine上一次的job runtime 列表，可得知哪些job是新下发运行的，哪些job已经完成，哪些job运行状态需更新，哪些 job已经丢失。Engine失联，即该engine租约过期，此时engine可能是网络分区或者engine宕机。对于网络分区来说，该engine上任务依然在正常运行；而若宕机，则任务已经不在运行，需要做后续处理（take over），一期中先直接标记为failed。Center中由JobStatTracker负责进行 job event事件的生成以及分发。

Center中将job状态的变更定义为JobEvent事件，job event 事件描述了job 的事件类型、产生该事件的engine、job当前确定的jobruntime信息（如发生lost事件，job当前状态并不确定，则此时job event中携带的jobruntime为上一次内存中存放的runtime信息）。其类型包括：

* ADD: job新出现在engine的上报中，为新下发的job
* UPDATE: job运行状态被engine上报更新
* LOST：运行的job的engine与center失联，job的状态不确定
* REMOVE：job已经不在原engine的上报列表中

Center中对job状态变化敏感的模块有任务调度模块（JobScheduler）和任务管控模块（JobController）。任务调度模块需要掌握各个可用engine上的负载情况，如engine上新启动了一个job，则负载增加，影响该engine在调度中的权重。任务管控模块需要负责内存中job状态的更新、数据库中的持久化以及与任务状态强相关的一些job操作等。

鉴于jobevent数据来源有两类，JobEvent的产生逻辑也可分为两类：

1. 从engine上报的jobruntime列表（new）于该engine上一次保存的jobruntime列表（old）进行对比。old中无，new中有-> add；old中有，new中有-> update；old中有，new中无-> remove。
2. 若job的状态超过设定时间未上报更新，即运行job的engine与center发生失联（网络分区或者engine宕机）,则-> lost

## 6. 任务管控

任务管控主要包含对job的管理操作以及job状态维护，其中任务状态的维护又可细分状态更新以及状态补偿。实现中这部分由DefaultJobController负责。

### 6.1 管理操作

Center对外提供的job的管理操与job当前的任务状态强相关，如任务挂起只支持对Alive状态的任务。

对于已经下发的任务的操作，如任务挂起、释放、更改、恢复，Center基本的处理逻辑为［执行（数据库更改、rpc调用），内存更新］。但任务的停止操作稍加区别，对于stop操作，执行逻辑为［数据库更改、内存更新、rpc调用］，即可能存在数据库中已经为stopped状态，但engine rpc 调用失败，任务实际依然alive的情况。为此，Center在状态维护时候若发现预期为stopped，实际engine上报不为stopped的情况下，会继续调用rpc操作。Stop如此设计等于为DTS提供了一种简单的让任务自动停止的方法，为后序运维以及engine的优雅下线做基础。另外，对于已经下发的任务的更新，DTS 只支持部分修改，如对于数据迁移任务只支持迁移对象的增删，以及增量、全量的一些执行配置，不支持其他元数据信息的修改。

未下发的任务的操作包括任务新建、更改、启动。新建即在数据库中增加一条job记录，默认job初始化状态为init。对于未下发任务的更改，DTS支持其所有信息的修改。未下发的任务启动需要执行预检查，预检查分为两部分：任务配置检查（网络连通性，用户权限，白名单，库表是否存在，数据是否兼容），下发检查。只有任务配置检查通过后，才会进行下发检查

### 6.2 状态维护

任务管控模块还需要对job的运行状态进行维护，可将运行状态的维护分为正常的状态更新、状态补偿两类。

状态更新：前面提到过基本的任务操作流程为：执行操作，内存更新。若整个操作正常执行，则下次engine 上报时候，会上报目标任务的最新的jobruntime ，则任务管控模块直接更新最新状态，并写入数据库。需要注意的是当job 被上报为stopped 或completed时，说明任务在engine里已经执行结束了， Center会先写入数据库，成功后再调用engine的release接口，通知engine后续可以不用上报该任务信息。

状态补偿：若任务操作中，执行rpc调用异常，此时内存中并不能更新任务状态。而此次操作engine到底有没有成功并不确定，只能通过engine下次上报时得知该job状态，并更新为Engine中实际的任务状态。需要注意的是stop操作会先更新内存，当rpc调用异常后，进行状态补偿时通过对比预期状态为stopped而实际并不是时，center会继续调用rpc操作。

除去正常的 engine上报外，状态补偿还发生在engine异常时。一期中，当engine失联时候，则center将收不到其上原来运行的job的状态上报，在超过job上报超时时间后，center会将 job均改成failed。若只是网络分区，后续engine重连上依旧上报正确的job信息，center收到后将 job状态进行更正。若engine确实宕机，则下次engine重启时上报的job信息为空，触发jobevent的remove 事件，此时center会清除job的engine信息，job 变成未下发状态。

## 7. 任务调度

Center中任务调度模块为JobScheduler ，负责集群可用engine的管理，任务的调度、takeover。

JobScheduler在初始化时，会从数据库中恢复当前alive状态的engine及其上的job运行信息。当engine新连上（Sessionevent.CONNECT）时，JobScheduler将初始化数据库中记录的该engine上正在运行的job信息。反之失联时候，JobScheduler会将其剔除。

目前任务调度的原则比较简单，将当前alive状态的engine以调度优先级排序，选优先级最高的engine进行任务下发。若当前无可用engine可用，则返回异常下发无资源可用。若下发rpc 失败，则继续按照优先级选取另外一个engine进行下发，重试三次，继续失败则返回rpc失败。调度优先级指标有［是否达到最大任务并发数、当前活跃任务数、有效时间下发失败次数］，这三者为递进考量关系。

JobScheduler中将每个engine封装成一个Scheduler对象，会记录其当前running（正在运行）、pending（剩余任务槽占用数）、下发失败次数。当前engine可用来调度即（running+pending < maxRunning）。每次尝试下发到engine之前，会尝试先占用engine的剩余任务槽，通过则rpc下发给该engine。若下发失败，则释放资源。若下发成功，则等下次收到该engine的jobevent.add时，会将运行数＋1、占用槽释放。同时engine在运行过程中，job执行结束，要进行running -1。

JobScheduler在调度到当前优先级最高的engine上，执行下发失败时，会继续重新尝试调度。由于未发生任务调度，engine 上的运行状况未改变，若只按照最小运行数作为优先级调度，则重试的调度选择依旧会选择为下发失败的engine。为此，scheduler还需要记录一个failtimes，记录最近时间段内下发失败的次数，下发失败数越少优先级越高。这样即可避免 engine宕机而center未察觉的这段时间内，job被重复调度到一个宕机的engine上而导致调度不可用。

# 第五章 DTS Engine详细设计

## 1. 概述

DTS Engine是DTS系统中的任务执行组件，Center在收到用户创建任务的请求后，对任务的原信息进行一些预检查，预检查通过后基于一定的负载均衡算法选择一个Engine将任务下发，此后Engine将定时上报自己负责运行的任务的运行时信息给Center，同时用户对任务的修改、暂停、恢复、停止等操作也是经由Center，最后下发给Engine来操作具体的任务的。

Engine作为任务的执行组件接受来自Center有关任务操作的请求，上报任务执行的情况，但在与Center由于网络故障无法通信时，可以继续任务的执行，等待网络故障恢复后接续接收请求和上报信息，在Engine宕机重启后会丢失其上执行的任务的信息，这是Engine会重新和Center建立租约关系，接受新的任务分配，而之前运行在Engine上的任务可能已经被分配给了其他的Engine执行。

## 2. 模块划分

DTS Engine组件的模块划分如下图所示：



Engine的模块可以分为两大类：与Center通信相关的和与任务执行相关的。

通讯相关模块主要包括：RPC通信模块、租约维护模块与状态上报模块。

任务执行相关模块中由包括：任务管理模块、任务组装模块、结构迁移过程模块、全量迁移过程模块、增量迁移过程模块，各个过程中又可以分为拉取、过滤和应用三个子模块。

## 3. 任务管理

在Engine中一个任务被划分为多个过程，各个过程之间相互独立，且按照顺序挨个执行。比如一个数据迁移任务，可以分为预检查过程、结构迁移过程、全量迁移过程和增量迁移过程，任务的执行就是对这四个过程的挨个执行。

任务组装模块的主要工作就是按照任务的描述信息，生成各个任务执行过程对象，将执行过程组装进任务执行体对象。

### 3.1 JobExecutor接口

**public** **interface** JobExecutor **extends** Runnable{

**public** JobRuntime getJobRuntime() **throws** DTSException;

**public** **void** alterJob(DtsJob job) **throws** DTSException;

**public** **void** stop() **throws** DTSException;

**public** **void** suspend() **throws** DTSException;

**public** **void** resume() **throws** DTSException;

**public** **void** preCheck() **throws** DTSException;

}

JobExecutor是任务执行体对外暴露的接口，任务管理模块通过这个接口操作任务。

* getJobRuntime:获取Job的运行时信息,JobRuntime一般包含多个jobPhase。
* alterJob:修改Job的内容，现阶段支持的修改内容只包含增加和减少迁移对象。
* stop:停止Job,停止后Job占用的资源将被回收，且不可恢复。
* suspend:挂起Job,Job占用的资源将不被回收，可以被恢复。
* resume:恢复Job,挂起和错误的Job都可以被恢复。
* preCheck:预检查Job，一般是对一些网络连通性进行检查。
* run:运行Job，只有预检查通过的Job才能被运行。

### 3.2 任务状态

**public** **enum** JobStatus {

***Init***,

***Alive***,

***Completed***,

***Failed***,

***Suspend***,

***Stopped***;

}

任务包含以下五种状态：

* Init：初始化状态，任务还未被下发给Engine时一般处于这个状态。
* Alive：执行中状态，任务已经在Engine上执行。
* Completed：已完成状态，任务已经完成了执行，已经不再占有Engine的资源。
* Failed：执行出错状态，任务在执行过程中遇到了错误，任务被挂起，但占用的资源不被释放，等待错误被修复后重新执行。
* Suspend：挂起状态，任务被用户主动挂起。
* Stopped：停止状态，任务被用户主动停止，已经不再占有Engine的资源。

五种状态存在以下的装换关系：



任务创建之初处于Init状态，此时任务管理模块调用preCheck完成任务的预检查，如果预检查失败则变为Failed状态，预检查成功后，调用run方法开始执行任务，任务变为Alive状态；此时用户如果发起暂停操作，管理模块调用suspend方法，任务状态变为Suspend，用户再次发起恢复操作，调用resume方法，任务状态恢复为Alive；如果任务在执行过程中遇到错误，则会变为Failed状态，此时等待用户解决错误然后发起恢复操作，恢复成功则重新回到Alive状态；任务执行完成后，状态最终将变为Completed状态；处理最初的Init和最后的Completed两个状态之外，任务在其他状态时都是可以由用户发起停止操作，进入Stopped状态的。

### 3.3 DTSProcess接口

**public** **interface** DTSProcess {

**public** **void** init(JobContext context, JobRuntime runtime) **throws** DTSException;

**public** **void** execute() **throws** DTSException;

**public** **void** suspend() **throws** DTSException;

**public** **void** release() **throws** DTSException;

**public** JobPhase getProcessRuntime() **throws** DTSException;

}

DTS系统中任务都是由不同的过程组成的，所有的过程都实现了DTSProcess接口

* init:初始化Process，一般在Process第一次调用前或任务发生了修改，需要Process也发生变化时被调用。
* execute:Process最重要的执行方法，如果正常返回表示Process执行成功，如果抛出异常表示执行过程中发生了错误。
* suspend:挂起Process,一般在用户发起暂停操作时会挂起当前正在执行的Process，或者当用户发起修改操作时，也要挂起当前正在执行的Process，然后重新从第一个Process开始执行。
* release:释放Process，如果Process正在执行则终止执行释放资源，如果没有再执行则单纯释放资源。
* getProcessRuntime:获得Process的运行时信息，具体的数据每个Process可能存在较大差异。

### 3.4 JobExecutor实现

JobExecutor具体的实现类为DefaultJobExecutor，它主要维护当前Job的状态和所有Process的列表两个类成员，并对接口中的方法按照以下思路实现：

* getJobRuntime：挨个获取各个Process的运行时信息，即JobPhase，获取当前状态的状态信息status，组装后返回。
* alterJob:如果任务处于Alive状态则挂起当前正在执行的Process，更新JobContext并使用更新后的JobContext和JobRuntime重新初始化所有Process并从第一个Process开始重新执行所有任务。现阶段支持的任务修改主要是修改迁移的表，所以当一个新的迁移对象加入时，无论结构、全量、增量迁移过程都是需要重新执行一遍的。
* stop:当任务被用户停止时，如果任务处于Alive状态则中断当前的Process并调用所有Process的release接口，将任务状态修改为Stopped。
* suspend:挂起当前正在执行的Process，修改任务状态为Suspend。
* resume:只有Suspend和Failed两种状态的任务才可以被恢复，此时重新执行上一个执行的Process，并且修改任务状态为Alive。
* preCheck:每个任务会有自己的预检查Process列表，依次执行这些Process即可。
* run:如果任务的状态为Suspend或者Failed，则挂起等待恢复，如果为Init则修改状态为Alive从第一个Process开始执行，如果为Stopeed或者Completed则不做任何操作然后退出。之所以要在run方法里考虑各种状态是因为一个任务可能在一个Engine节点上执行过程中由于Engine宕机等原因被重新分配到一个新的Engine上执行，此时任务可能出现任何一种状态。

## 4. 全量迁移过程

全量迁移过程负责将源端数据库中当前迁移对象现在的数据全部迁移到目标系统中。在一个迁移任务中，全量迁移一般发生在结构化迁移之后，增量迁移之前。

### 4.1 MySQL全量迁移过程

MySQL的全量迁移过程实现思路比较简单，只需要从源端Select迁移对象的所有数据然后应用到不同的目标系统即可。但实现上还是有一些细节可以考虑。首先在Select数据的时候是否使用事务，如果使用事务是加一个大事务还是分段加事务。在一个事务里读取所有的数据可以保证读到的数据都是事务开始时的数据，由于MySQL MVCC特性的支持，读事务是不会阻塞后面的写操作和读操作的，但是会在所有记录上往后面建立更多的副本数据，如果迁移对象数据量非常大的情况下，这一个读事务可能持续时间非常长，导致副本数据持续增大，一方面会拖慢后面的查询操作，另一方面可能由于占用存储资源过大导致系统崩溃；另外一种方法是分段加锁，按照主键的范围分段加锁，这样能减少第一种方法导致的各种问题，但却无法保证全量获取到的数据都是全量开始时的数据，会掺杂一些增量的数据；还有一种方法是彻底不使用事务，只是按照主键排序后一批一批得读取数据，此时不保证所有数据与迁移开始时一致，其中会掺杂增量过程中的数据，但这样做的好处是对源端的侵入最少。

由于DTS数据迁移的业务场景一般是拉取线上库的数据交由OLAP系统进行数据分析，所以对源端数据的影响越小越好，所以最终MySQL的全量迁移过程暂选择上述的最后一种实现思路，这样只要保证在后面的增量迁移过程使用类似“replace”这样有幂等语言的操作进行数据更改，就可以保证最终数据一致。

另外全量迁移时的并发度与一次从源端拉取多少条数据也是应该考虑到的。还是按照对源端影响尽量小的原则，现在的全量迁移实现定位单线程挨个执行，用户可以配置一次拉取的数据行数，另外用户还可以配置拉取之间的等待时间间隔来再次减少全量迁移过程对源端的影响。

MySQL全量复制过程的实现类为MySQLFullReplProcess，它的具体实现方法如下：

* Init:初始化过程，从JobContext对象中获取源端MySQL的ip、port、username、password等连接信息和batchNum、doTables等运行时信息，如果runtime不为null，则从中获取上一次复制到的位置。
* execute:按照doTable中的顺序，挨个按照batchNum批量获取表记录信息，查询按照表的主键进行排序，下次查询的主键起始值是上一次查询的最后一条记录的主键取值，用limit关键字控制记录数量，当某次获取到的数据行数小于batchNum时表示已经获取到了表的所有数据，使用最简单的select语句获取数据，不进行任何加锁处理。获取到行数据信息后封装进FullReplEvent中，并根据目的端系统的不同交由不同的applier将数据插入不同的目的端系统中。
* suspen:修改当前Process的状态为SUSPEND。
* releae:修改当前Process的状态为Stopped，等待执行过程结束，释放源端连接信息和Applier所占用的资源
* getProcessRuntime:获取Process的一些运行时信息，供用户参考，主要内容包括：

status:Process当前的状态

startTime:Process的开始时间

progress:Process的进度

failReason:失败原因

tableCount:总共要迁移的表的数量

migCount:已经完成迁移的表的数量

currentTable:当前迁移的表的数量

## 5. 增量迁移过程

增量迁移过程是将源端数据库随着时间推移，所迁移对象上的一些数据变化迁移到目标端系统中，数据变化包括数据的新增、删除和修改，在配置了结构迁移的任务中还包括结构的变化。增量迁移一般处于全量迁移之后执行，但一个任务也可以只包含增量迁移过程。

### 5.1 MySQL全量迁移过程

MySQL的全量迁移过程的实现方式是与源端数据库建立复制关系，engine作为复制关系中的slave拉取源端MySQL的binlog存储在本地，然后再读取这些binlog过滤掉不关心的event后交由应用模块应用到不同的目的端中。

全量迁移实现过程中依然有一些细节需要注意，首先由于一个engine上会同时运行多个任务，它们将共同使用engine所在主机上的磁盘资源，如果一个任务堆积的binlog过大会影响其他任务的执行，所以给每个任务分配一个磁盘使用上限是有必要的，由于binlog的拉取与读取是异步的，就有可能读取延后于拉取位置，这一般出现在应用端应用速度赶不上源端的数据产生速度，此时这部分binlog无法被清理，且有可能越来越大，当达到配置的上限时将阻塞拉取的线程。

另外当一个任务同时有全量迁移与增量迁移时，全量迁移之前，增量迁移过程需要记录下源端MySQL的binlog位置，并以这个位置作为增量开始时的起始位置。一种优化实现时这个时候同时开启拉取源端binlog的任务，这样当全量复制开始后就可以直接使用这部分数据了，同时还能解决由于全量迁移过程太长，导致源端的binlog已经被清楚的问题。

在DTS Engine中使用tungsten作为拉取源端MySQL binlog的工具，tungsten提供的Filter组件同时也满足了我们过滤数据的需求。

MySQLIncReplProcess是MySQL增量复制的实现类，其具体实现如下：

* init:初始化过程中，从JobContex对象中获取源端MySQL的ip、port、username、password等连接信息和parallelCount、doTables等运行时信息，如果runtime不为null，则从中获取上一次复制到的位置。
* execute:MySQLIncReplProcess使用tungsten作为其增量复制引擎，本身的执行过程就是对tungsten数据迁移过程的封装。
* suspend:挂起tungsten的执行线程，并退出过程的执行，修改过程的状态为Suspend
* release:终止tungsten的执行，清理tungsten与applier占用的资源。
* getProcessRuntime:获取Process的一些运行时信息，供用户参考，主要内容包括：

status:Pocess当前的状态

startTime:Process开始的时间

progress:Process的进度

failReason:失败原因

sourcePos:当前源端的binlog位置

applyingPos:当前apply的最新的binlog位置

delayTime:最新apply的事件距离时间产生时的延迟

leftTime:预计的追赶上源端的时间

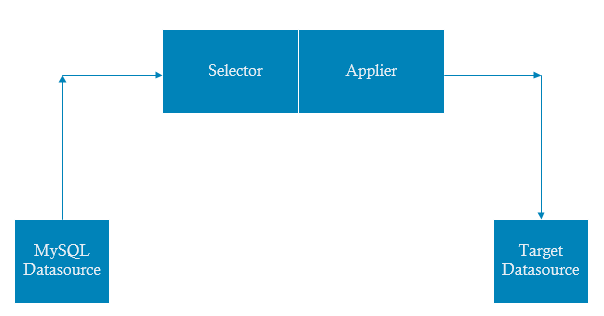
leftRelayLogByte:剩余还需apply的relay log大小

totalRelayLogByte:当前多有relay log占用的大小

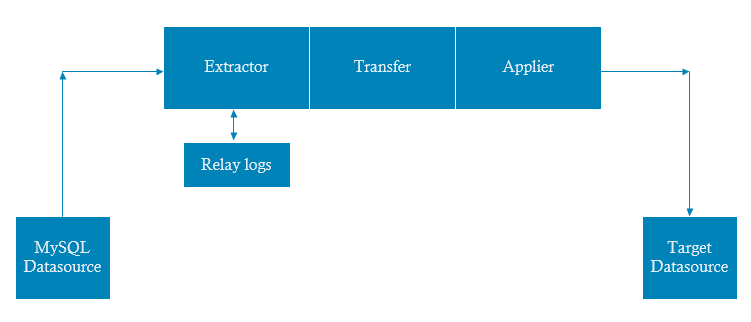
# 第六章 DTS Applier详细设计

## 1. 概述

DTS的Engine是进行数据迁移的实际节点，在其上面可以执行限定数量的任务。Applier是其中非常重要的模块，全量迁移比较简单，全量Applier负责将全量过程中从源端select出来的数据应用到目标数据库中，其流程图如下：



增量迁移比较复杂，在Extractor和Filter完成相应逻辑后，数据会最终交由增量Applier负责处理事件转换，事件并发处理，以及到目标端的应用。其流程图如下：



增量Applier模块是决定数据迁移任务性能的关键，提供如下特性：

支持行级和事务级并行复制，追赶binlog速度远快于MySQL原生复制；

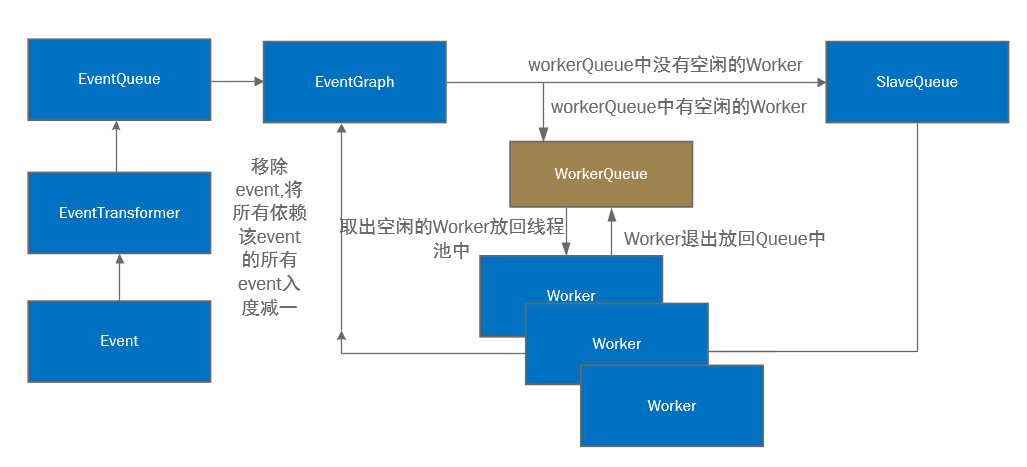
支持断点续传，在源端或目标出现问题后，可以暂停任务，修复问题后根据上次服务记录的重启点恢复任务；

基于事件依赖图和replace语法保证数据一致性，其中行级事务并行复制适用于迁移场景，只保证最终一致性，在牺牲完全实时一致性的同时满足了高效的复制需求

下面主要介绍一下增量Applier模块的实现。

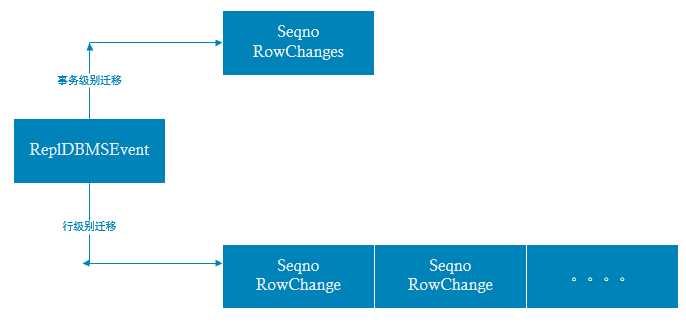
## 2. Applier的组成

增量Applier包含事件转换器EventTransformer，事件依赖图EventGraph，事件消费队列SlaveQueue，任务执行器队列WorkerQueue，任务执行器Worker，进度更新ApplierPoint等模块，其详细设计如下图所示：



### 2.1转换器

EventTransformer用于将Tungsten Replicator中的数据格式ReplDBMSEvent封装生成所需的HamalEvent，每个HamalEvent就是Hamal Engine的处理单元，具有全局唯一的事件编号seqno，用户可配置是按照行级改变还是按照事务来迁移数据。如果选择按照事务来迁移数据，一个HamalEvent对应一个数据库事务；如果选择按照行级改变来迁移数据，一个ReplDBMSEvent封装成多个HamalEvent，每个HamalEvent对应一个行级改变。如下图所示：



选择用什么方式来进行迁移，决定了并行复制的执行粒度和一致性级别，需要指出的当设置为行级改变时，事务的原子特性会被破坏，但不会影响数据同步的最终一致性，同时复制的并行度将有效提升。

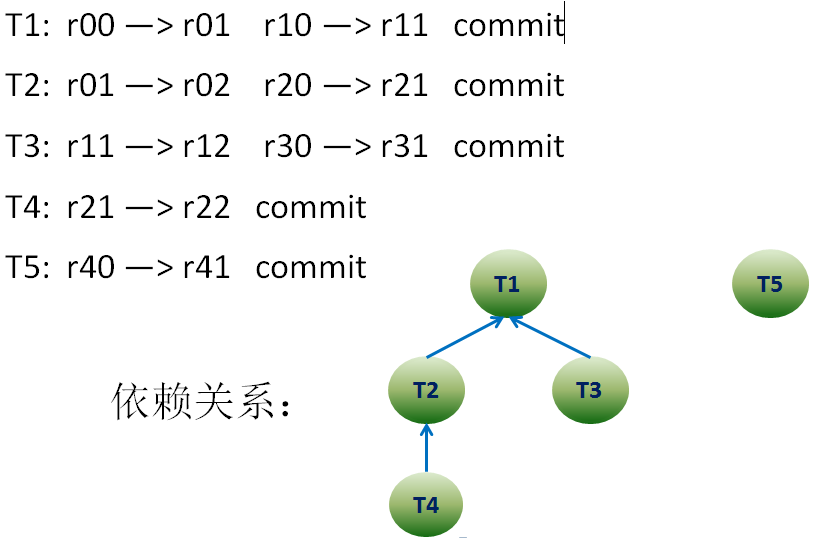
### 2.2依赖图

事件依赖图是DTS实现高效并行复制的关键，通过构建事件彼此依赖关系，生成依赖图，只有互相无依赖的事件才可以并行处理以保证数据一致，而有依赖的事件必须串行化处理。因此，事件的执行顺序无法保证与binlog中一致，但可满足最终的数据一致性。

#### 2.2.1依赖关系

事件之间的依赖关系反映在若干有向无环图中，我们将其称之为依赖图，每张表的每个唯一性索引对应一张依赖图。因为唯一性索引具有天然的约束关系，后进入依赖图中的事件（binlog中位置靠后的事件）如果在某唯一性索引的值上与之前依赖图中其他事件相同，则意味着该事件在该唯一性索引上存在依赖，必须等待之前的全部事件都依次执行完（并从依赖图中移除）后方可被执行。而不存在唯一性索引值冲突的事件则表示它们的数据修改没有任何交集，完全可以同时执行，无需等待。

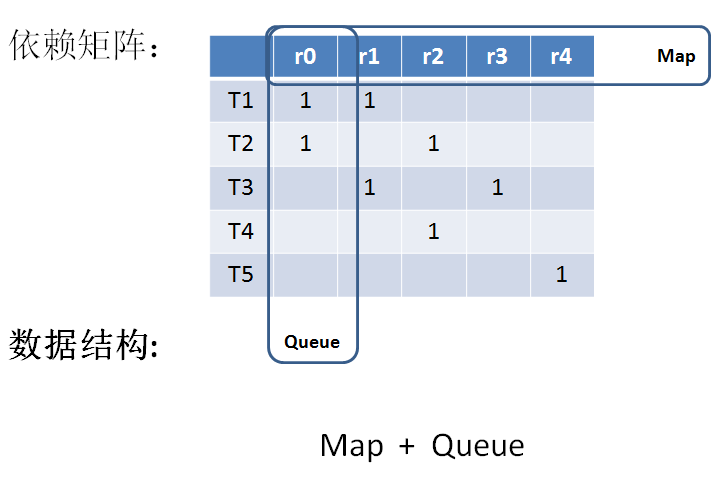
基于图论的概念，事件的依赖关系由其维护的入度表示，入度为零则表示该事件不依赖其余事件，不为零则表示其依赖其他事件。用如下的一组事务提交记录举例说明。假设迁移过程以唯一性索引X构建了一张事务依赖图。事件T1有两条修改记录，Record1（将唯一性索引值从0修改为1），Record2（从10修改为11）。事件T2有两条修改记录，Record3（将唯一性索引值从1修改为2），Record4（从20修改为21）。此时根据发现，Record3的旧值依赖于Record1的新值，说明Record1在图中，Record3就无法执行。剩下的事务据此类推，形成如下依赖关系图。



直到T1移除，T2，T3，T4才能从事件依赖图中移除。需要特别说明的是，上图所说明的是只有一个依赖图的情形，如果一个Event依赖多张图，必须保证所有图都不存在依赖关系，才能被用于应用到目标端数据库。

#### 2.2.2实现方式

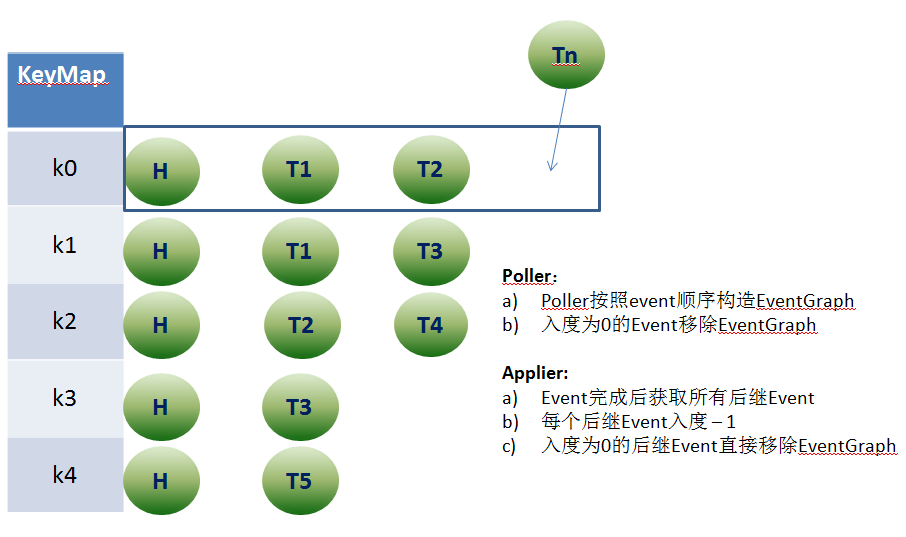
依赖图通过Map+Queue实现， Map键值为某行数据的该唯一索引值计算出的哈希值，其数据值为一个链表，依次串联与这行数据相关的依赖事件。由此，通过该有向无环图算法就形成了一个执行队列，这个队列中只有上一个节点执行完成才可能触发下一个节点的执行。任意一个事件只有满足其在所有相关的依赖图中均不存在依赖关系时才算解除依赖，允许交由工作线程处理或投入执行队列。



#### 2.2.3执行流程

在事件依赖图中，Poller负责将Event放入到依赖图中，为了防止依赖图无限增长，当DataGraph中的Event达到上限时，Poller会被阻塞。

EventGraph交由线程EventGraphManager管理，对依赖图进行更新，对入度为0的Event进行移除，交由任务执行器Worker处理或者放入slaveQueue。



### 2.3 任务执行

任务的执行由对应于不同目标源的Worker来完成，如果需要处理的事件比较多，使用SlaveQueue存放可处理的事件，由Worker进行处理 。

SlaveQueue是在Event移除依赖图时，没有空闲的Worker可以使用时存放的队列，只有彼此不依赖的事件才可能同时存在于该队列中，EventGraph是队列的生产者，Worker是队列的消费者。

Worker，对应不同的目标源有不同的Worker，是Applier执行并行复制的工作线程，由从依赖图移除的Event触发，并能继续从SlaveQueue中获取事件，应用到目标端。一个事件执行完成后， Worker还会通过EventGraphManager将该事件从相应的EventGraph中删除，并更新其余依赖事件的入度，并再次触发对入度为零的事件的处理。

WokrerQueue维护了当前Applier可用的Worker，当依赖图有事件移除，首先查看WorkerQueue中是否有空闲的Worker可以使用。如果有，从队列中取出一个Worker放入到整个Engine共享的线程池中运行。运行中的Worker在处理完触发它的Event之后，会处理slaveQueue中的Event，当无法从slaveQueue中获取到Event处理时会结束，重新放回到WorkerQueue中，等待下一次依赖图有事件移除时触发。

需要指出的是Worker执行过程是保证幂等性的，即同一个事件多次执行最终的数据一致性不受影响。而实现幂等的核心思想是在apply阶段构建语句时使用REPLACE INTO语句代替原本的INSERT INTO和UPDATE语句实现数据插入和更新，保证数据最终的值一定会生效在目标数据库里。

我们根据从binlog文件中解析出来的事件不同类型分别具体讨论：

1. Insert事件

Insert事件只有新值，没有旧值。在此采用REPLACE INTO句式将新值插入目标数据库，即使该行记录已存在，也可以通过唯一性索引的约束保证结果正确性。

2. Delete事件

Delete事件只有旧值，没有新值。采用DELETE FROM句式删除旧值，其中将主键作为WHERE条件指定。

3. Update事件

Update事件同时有旧值和新值，它的处理是最为复杂的，因为涉及到唯一性索引（DDB还包括均衡字段），主要分两种情况讨论：

1）如果所有唯一性索引（包括主键）都发生了改变，则采用DELETE FROM句式删除旧值，REPLACE INTO句式插入新值。否则无法依靠REPLACE INTO句式实现旧值到新值的更新（因为唯一性索引都改变了，新值与旧值理论上并不冲突）。

2）除此之外的情况，均采用REPLACE INTO句式直接进行旧值到新值的更新操作

## 3. 执行详情

在DTS的Engine中Applier模块是负责将过滤之后的源端数据导入到目标数据库中。在Applier模块的执行流程如下：

EventTransfer将抽取的事件转换为HamalEvent；

对HamalEvent中行记录进行处理，将主键和唯一键索引信息（用于生成导入到目标数据库的语法）记录在Event的HamalRowChange当中，随后由EventGraphManager进行处理；

EventGraphManager不断的从EventQueue获取到HamalEvent，根据该事件中所有涉及的主键和唯一性索引构建EventGraph，然后放入到对应的所有EventGraph中，计算其入度。如果在所有依赖图中的入度为0，根据特定选择策略启动一个Worker放入线程池中执行，或者直接放入SlaveQueue中等待其他Worker执行

每个Worker都是一个Runnable，需要放到全局共享的线程池中执行，根据系统设置的并发数初始化放入到一个WorkerQueue中，当EventGraph中存在入度为0的Event，判断WorkerQueue是否有闲置的Worker，如果存在取出这个Worker绑定这个Event放入到线程池中执行；如果不存在放入SlaveQueue中。

Worker的实现首先是执行触发其启动的Event，然后循环的从SlaveQueue中获取Event执行，直到没有Event可以处理，退出，放回到WorkerQueue中。

Worker执行完一个Event，Worker会通过EventGraphManager将该事件从所有涉及的EventGraph中移除，并且所有直接依赖于该事件的其余事件的入度均减一，此时会判断是否有入度为零的新事件被选出，如果是则采用上述同样逻辑进行处理，反之则等待。随后更新ApplyPoint。

## 4. ApplierPoint更新

我们定义ApplyPoint这一数据结构来保存该记录点信息，其中维护了一个排序队列（按事件的seqno递增排序），里面存放的元素为HamalEventHeader，即HamalEvent中除去数据部分的元数据内容。每个事件在放入EventGraph的同时会将其头部信息封装成HamalEventHeader放入ApplyPoint中，而在Worker执行完该事件后由EventGraphManager将其从EventGraph中删除的同时会将其对应的HamalEventHeader从ApplyPoint中删除。同时将当前排序队列中头部事件（即seqno最小的事件）记为一个记录点事件，这是由于并行复制导致seqno较大的事件可能先于seqno较小事件被完成，而我们关心的记录点意味着所有seqno比该事件小的事件都应该被同步了。通过这种方式保证该队列头部元素即为记录点，尽管该记录点之后的某些事件也可能已经被同步了，但由于我们导入目标库执行时可以保证幂等性，所以重复同步并不会带来数据不一致或其他冲突问题。

## 5. 扩展性设计

在增量Applier的设计中，考虑到其扩展性，相关依赖都进行了扩展性设计。

1. 依赖图管理

依赖图的相关操作都作为一个公共模块提供，用户不需要修改。在构建依赖图时，用户可以根据源端数据库的特点，构建需要的依赖图，定制自己的并行复制策略，阀值等信息。既保证了模块的稳定性，又保证了灵活性。

1. 任务进度管理

任务进度的更新，作为一个公共模块，放在了所有Applier实现的基类中，用户不需要了解其实现，只需要调用获取进度的接口，就可以获取进度。

1. 任务执行器Worker

任务执行器Worker处理导入到目标源的工作，有任务时放入到线程池中执行，没有任务退回到Worker队列等待被启动，最大限度的节约了系统的线程资源。其中重试，终止，启动，修改都提供了统一的接口。

有了以上扩展性的设计，当系统需要支持新增目标源时，可以非常容易的实现一个Applier，并保持并行复制的性能。