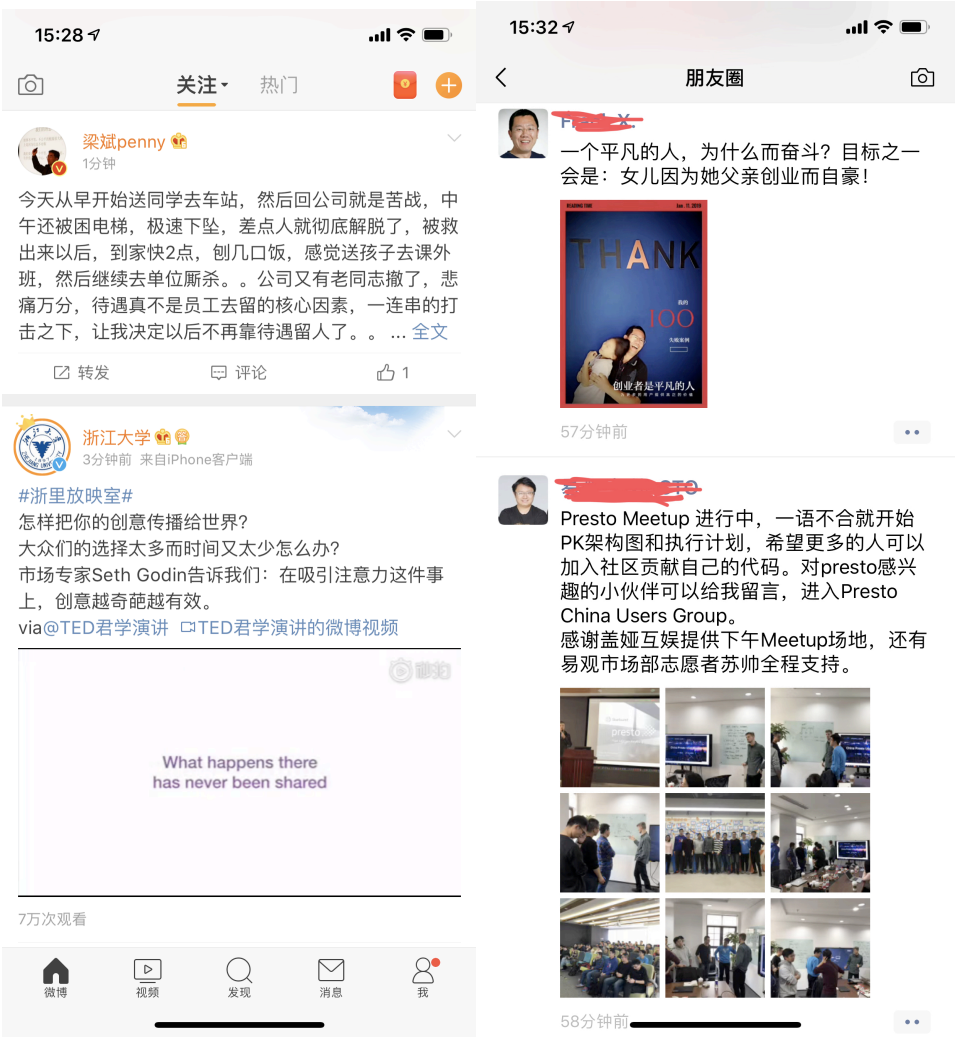
推送系统总体设计&详细设计

目 录

1 背景

推送系统用于展现好友动态，服务于众多的产品线（类似于微信朋友圈、新浪微博 feed 流）。



2 名词解释

推送系统：

好友动态：

拉链：

logic：

di：

3 设计目标

3.1 实现的功能

推送系统提供以下功能：

- 提供动态的添加功能(add)；
- 提供动态的修改功能(upd)；
- 提供动态的删除功能(del)；
- 提供屏蔽发送动态功能，用户可以设置不对其他好友(用户)展现某类动态；
- 提供屏蔽动态类型的功能；
- 提供发送动态的频度控制功能(可以按照分产品线进行发送频度的控制)；
- 提供可方便扩展动态类型；
- 提供对不同类型的动态，提供一定的隔离度；
- 当前，动态仅仅针对个人，但是，在产品中，存在着这对非个人的动态。例如话题、团队、群组等；
- 对于动态存在着权限控制的问题，可以设置不同的级别，例如全公开和完全私有；
- 支持直接查询一个或者多个产品线动态的功能；
- 支持直接查询一个用户动态的功能；
- 支持查询屏蔽用户某类动态的功能；
- 支持动态的聚类展现；
- 提供 HTTP 访问接口。

3.2 设计的性能指标

推送系统是一个高速稳定的系统，它主要定位于动态的提交以及动态的浏览相关操作。因此在整个性能方面，我们涉及包括如下的几个部分。

- 提交：

针对“动态添加、修改和删除”这类操作，要求提交吞吐量分别可以达到 10000s-1，用户看到的延迟为 1s 以内。

- 查询：

推送系统对查询的性能要求极高，可以通过全内存加冗余的方式解决其高性能，其全局的访问量要求可以达到 50000s-1。

4 系统环境

4.1 假设及与其它系统联系

基于相关通用性的考虑，推送系统会分为两个层次，底层为通用的核心数据层，而上层则为可定制的对外服务层，维护一些各个产品线可能不同的需求和功能

从需求来看，用户的查询列表，主要为好友列表，所以这次将好友列表查询作为一个定制集成在推送系统外层。

4.2 相关软件及硬件

推送系统运行于传统的 64 位 Linux 环境中，要求 16 核机器，Linux 内核 2.6，至少 32GByte 内存。

4.3 系统限制

对于用户的批量查询 ,基于性能的考虑 ,展现的数据可能只会是有选择的选取更新最频繁的部分用户(如 2000 个)。

由于现有数据量不大，所以计划，仍然保留 90 天的数据，有利于其他小数据量产品线获得更好的展现效果；

在默认情况下，同一个用户相同种类的动态，仅仅展现 5 个，以减少动态之间挤掉的情况；

默认情况下，一个用户一个产品线，保留 100 条最新动态；

对于动态聚类展现，聚类后的动态，有一个上限(5 条)，其他满足聚类需求的元素，被丢弃；

默认情况下，一个用户最多展现 100 条动态，如果使用聚类，一天中的 5 条算作一条；

动态聚类仅仅提供一种聚类方式以及聚类仅仅提供一层聚类，对于更多层次的聚类，可以此层聚类为基础，在已经聚类好的基础上，上层应用进行小范围内的自定义聚类；

对于单个动态，对其 other 扩展字段内容大小有限制(65KB)；

一个用户最多展现 100 条(一个 APP 一天的 5 条作为一条)动态。对单个产品线也是 100 条，底层的存储不限于 100 条，只是在浏览端和 Cache 端进行了控制，用以提高性能。

对外服务最多也只提供 100 条动态。

4.4 数据规模估计

5 设计思路及折衷

5.1 推？ Or 拉？

推还是拉是推送系统讨论的焦点。推送系统的推，指的是用户的行为都及时的推送到需要展现这个用户信息的组去，查询的时候，直接获取这个组的数据就可以了。而推送系统的拉，指的是用户的行为都被记录在这个用户本身的行为之中，在查询的时候，需要确定这个组需要展现那些用户，再做一定的处理。

推和拉两种方式，在实现上，会有很大的不同，对用户的最后展现效果，也会各有优劣，以好友动态为例。

两种方案对比

1. 系统的性能
2. 可扩展性
3. 安全性

新浪微博等好友动态，普通浏览者可见的，这点和其他有些 SNS 网站有较大的差别。所以在黄反处理上有更大的力度要求，不能像某些 SNS 网站那样，不做处理。

5.2 设计思路折中

推送系统仍然采用“拉“的模型，用户的动态行为都被记录在这个用户本身的动态行为之中，在查询的时候，需要确定这个组需要展现那些用户，再做一定的处理。。

5.3 其他

对于推送系统的定位是，争取作为整个数据中心，那么必然要求照顾到各个产品线的需求以及展现。推送系统的一个主要问题，便是希望能够均匀的展现用户动态，尽量保证不会因为某种类型的动态冲掉了整个产品线的动态，甚至是一个用户所有的动态。因此决定对每种 APP 每天(时间跨度可调)只保留 5 条动态，就能够有效的避免对其他应用的冲击。

6 系统设计

6.1 基本介绍

将整个系统划分为两层，核心层尽量不涉及到定制化的需求，保证最大化的通用性；而外层则用来处理可能来自其他产品线的耦合，实现定制化的功能。

直接和好友系统做一定的绑定，有利于其他产品做简单的调用，避免了各个产品线重复的开发功能过于相似的接入层，提高推送系统的通用性和复用性。

6.2 系统架构图及说明

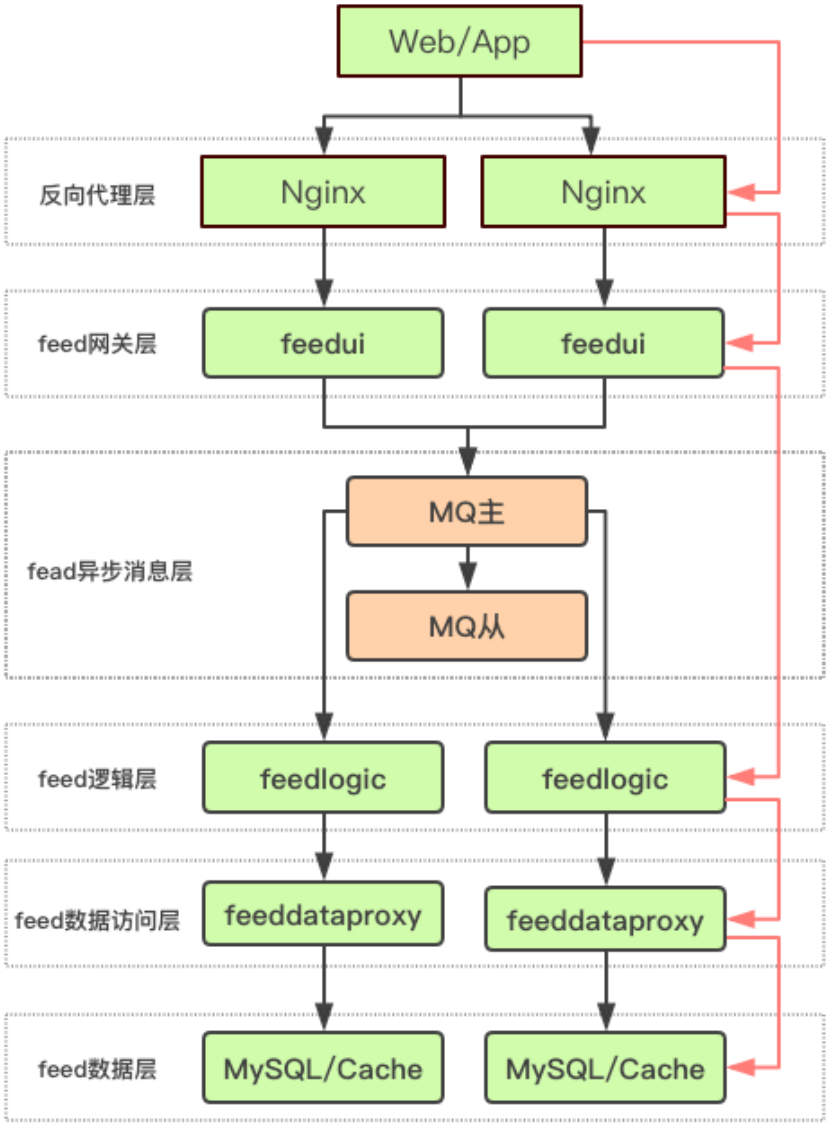


图 6-1 推送系统整体架构图

整个推送系统的设计，采用了接入层 feedui，异步消息分发 feeadMQ，业务逻辑层，数据访问层，数据存储

层这样的层次进行架构。

整个推送系统的分层，分为如下几个组成部分:

- feedui

feedui 实现了推送系统对外的所有数据接口。

- feedMQ:

feedui 对推送系统的所有更新接口(添加、更改和删除)提交到 feedMQ 进行异步处理。

- 查询部分:

查询部分的功能，向分布式 cache 进行直接查询，对查询到的动态进行排序、聚类(grpid 的产生)、组装结果，返回给查询端；对于未在 cache 中命中的查询，考虑到查询性能，我们并不会再次查询数据库，而是把未命中的查询记录下来，提交到 feedMQ 中，由异步更新模块(feedlogic)查询数据库填充到 cache 中，保证下次查询命中。

- feedMQ

可以使用 RockectMQ。

feedui 把更新命令提交到 feedMQ。

- feedsvr

这个模块，实现了所有的业务逻辑操作。

这些操作包括：

a) 单条动态添加操作；

b) 按照{product, subtype}删除、修改操作；

c) 按照{{srctype, srcid}[], {product,subtype}[]}的分页查询。

- feeddataproxy

这个模块，实现了所有的原子 CRUD 操作。

在设计上，推送系统采用数据全内存和 Cache 冗余机制，提供高性能服务。

在整个系统的设计上，对外形成如下接口层：

feedui 对外形成的 HTTP 数据访问接口层

6.3 系统流程图及说明

6.4 设计

6.4.1 设计思想

设计思想其实是一个元数据和内容数据分离的设计思想。其思想就是，将需要进行逻辑组织的数据和内容数据进行拆分，减少数据组织过程中，需要读取的部分。在现有的推送系统中，如果在一个比较宽泛的程度来说，现有的推送中，logic 和 DI 实际上使用了数据库的表拆分进行实现的。

6.4.2 对于多产品线动态的保护

为了保护相对弱势的产品线的动态不被强势的产品线的动态所冲掉，需要使用一种方式将弱势的产品线中的动态进行保护。由于实际上产品的动态在 DI 上是不被会被抹去的，对于冲掉的含义，仅限于 logic 部分。

为了将多个产品线的动态进行相互保护，最简单的办法是为每个产品线维护一个 logic 的拉链。在现阶段，产品线之间的活跃用户的相互重叠非常小，如果每个产品线分别维护一个拉链，会比较麻烦。所以 logic 拉链只需要一个即可。

6.5 数据库的组织方式、分表

6.5.1 数据库分表策略设计方案一

在现在的推送设计中，有两种表结构，meta 和 feed.

在 meta 表中仅仅记录了一些分表的时间信息，比较简单，而主要的动态数据全部存储在 feed 结构的表中。

对 feed 表结构，使用了时间和用户 id 两种方式进行分表，拆分成 feed_[0-5]_[0-5]这样的 6*6 个表。大概形成这样的一种结构：

在时间上，用“week%6”进行分表，这样做的目的除了可以有效地减少单个表的规模，另外主要是为了对数据进行有效的清理。在每周的固定时间，对一个时间分区上的表进行“DROP TABLE”和“CREATE TABLE”操作，这样可以对历史数据进行高效干净的处理。

在 uid 上，根据“uid%6”进行分表，则主要是考虑控制表规模，提高表的查询效率。

存在问题：

但是在实际的工作中，发现的问题却是分表太多，导致一次前端的查询导致了在多个数据库中进行查询，实

际的情况却是导致了查询开销过大。

另外，由于现在有的数据的组织发生了一些变化，所以对分表策略有新的要求。

6.5.2 数据库分表策略设计方案二

在设计中有两类数据表，di 数据表，logic 数据表。

logic 数据的访问和更新方式都是使用 uid 为 key，拉出一条拉链。在这样的拉链上进行数据的操作。logic 数据的基本元素相对较小，希望能够对于一个用户的 logic 数据的查询，不需要在多个表中进行查询。

每个 logic 单元中，维护了一个动态的 id，利用这个 id 向 di 数据表进行完整动态的获取。那么在查询的时候，对于 di 数据的获取方式，是根据一个 id 进行的。

除了添加和查询这样的按照用户 id，动态 id 的查询，为了进行删除，修改，将删除/修改的结果直接用{uid，product, subtype, eid，groupid}的方式在 logic 进行删除/修改。

另外，在 logic 表中，存储的是定长的数据，因此数据库本身就能够对其进行高效的回收，而对于 DI 中的数据，由于数据变长，没有办法进行有效的回收，必须考虑将新数据和老数据进行隔离。

此外，DI 分表后要能够方便根据动态 id 直接取出整个动态的完整数据。这点在分表策略上需要考虑这方面的问题，也就是，根据动态 ID 能够很容易的找出动态所在的 DI 子表。

因此准备使用如下的设计模式：

1)对于 logic 数据，采用的分表策略为，根据 uid 取模进行分表,对于 logic 数据库不进行删除操作。

2)对于 DI 数据的数据库分表，准备使用根据动态 id 进行水平分表，这样，根据一个动态 id，做一个除法，就可以很容易的知道此动态所在的子表，加速查询。

由于需要查询的动态仅仅保存在最新的一些 DI 子表中，所以可以定期的清除掉一些老的 DI 表。通过这样的方式实现系统的扩展。

在 logic 的工作中，所有的数据都是使用 eid 进行的，包括动态的过期控制，因此需要记录一些辅助的数据，维护一些时间点上的 eid。

这样分表的主要目的就是为了让 logic 和 DI 的单张表的规模不会随着数据规模的扩张而变大。

这样设计另外的一个好处就是今后 PM 进行策略制定的时候，已经不再会有在“一个星期”的限制了，而是可以在任何天数上制定聚合等策略了。

6.6 核心数据设计

6.6.1 一般动态模型

在整体的设计上，我们在对“动态”的组织方面，仍然按照用户为中心。在此基础上，还须根据用户的查询模式，在查询的时候，还需要进行一些划分。我们对动态数据进行抽象，对其建模如下：

在存储的设计上，准备使用传统的的方式进行消息的存储。

other 字段是扩展字段，以 PB 序列化后的形式进行存储，将来的需求字段都可以填入此 other 字段中。

在设计中，对于删除操作，需要直接进行拉链删除，同时，对于添加操作，也会直接将拉链范围外的元素进行删除操作。

6.6.2 数据操作模型设计

对 logic，采用 srcid 取模进行存储，di 采用对 eid 取除的方式进行存储。

在访问模式上，存在如下几种操作：

1. logic 和 di 的事务性的插入新记录；
2. 对 logic 进行{srctype, srcid, product}或者{srctype, srcid, product, subtype}的拉链拉取；
3. 对 di 进行按照 eid 的存取；
4. 对 logic 进行按照{srctype, srcid, product, subtype, eid}的修改和删除；

6.6.3 其他表考虑

DI 表：KEY(eid)

相关说明：聚类的展现需求，目前有 4 种，

围绕主语进行聚类：如添加好友，展现为 a 添加了 b,c,d,e 为好友，上限为 5 个 id

围绕宾语进行聚类：如参与投票，展现为 a，b，c，d 参与了 xxxx 投票，上限为 10 个 id

不进行聚类：如发表朋友圈， 展现为 a 发表了朋友圈 xxxx；a 发表了朋友圈 yyyy；

多级聚类：对于空间上传图片，展现为 a 在相册 xxx 中上传了图片 a，b，c，d; 需要对 spaceid 以及相册 id 联合进行聚类。

这些功能仅仅使用 groupid 来进行实现。

6.6.4 表空间回收

在我们的设计上，对于 DI 表，由于 90 天之后，我们不会再对 DI 有任何的更新和查询，因此不准备对 DI 数

据做任何的回收工作，待 DI 无效后，我们会根据磁盘状况，定期的进行“DROP TABLE”操作。

对于 logic 表，在对相应的用户的进行更新的时候，会对相应 logic 数据进行一次清理。根据实验，初步观察，InnoDB 引擎，在重复使用 PRIMARY_KEY 的情况下，能够非常高效的对删除的空间进行回收，但是对于不重复使用 PRIMARY_KEY 的情况下，回收将非常低效。

为了避免这样的情况，推送系统使用更加激进的方式，进行 logic 空间的回收，在设计上面，使用如下的方式：在准备向一张 logic 表中插入一个 logic 单元的时候，首先尝试删除过期的两个 logic 单元。通过这种方式，可以保证在整个 logic 空间当中，基本上没有过期的数据。

6.6.5 性能问题的考虑

在旧数据的保留上，综合动态数据规模和动态展现效果，拟保留 90 天的数据量，对于动态需要进行被动修改和删除的时候，如果存在数据规模过大的问题，可以在更新的范围上，做一定的修改，仅仅对一定时间(eid)范围内的动态进行修改和删除支持。这样，级联修改和删除操作就会仅仅集中于少数几张 DI 表上。

在设计上，我们对性能的瓶颈主要在添加的插入上(当然级联修改和删除更加耗时，只是每天量不是太大)，并且在动态插入的过程中会伴随着查询操作，通过引入数据库分片机制，解决添加操作的性能瓶颈。初期上线，打算分四个数据库。

在性能上，大量的查询会威胁到整个推送系统的性能(每天的查询请求上亿次)。为了提供高速稳定的服务，采用动态数据全内存的方式解决海量的查询请求，并提供 cache 的冗余机制解决查询服务的稳定和高可用性。

6.7 数据库与 Cache 的组织方式

数据库使用 mysql 5.7， 存储引擎: InnoDB，编码:UTF8mb4；

数据库名为:feed；

分表策略

在部署上，我们会对数据按照 srcid 进行分片操作，但是一个 feedlogic/feeddataproxy，会负责所有分区的数据处理。

Cache 主要是针对查询进行优化。由于推送系统对于实时性的要求不算太高，对于用户提交更新/删除时所需要的优化，仅仅在数据库索引层面进行，Cache 不做考虑。

现在，用户查询时，对数据库的操作集中在两个方面。

一方面，根据 uid 拉取 logic 数据，另外根据 eid 进行 DI 数据的查询。因此设计的 cache 有两种，一种是

(uid,product)为 key，logic 拉链为 value 进行组织的 logic 的变长 cache；另外一种是 eid 为 key，具体动态内容为 value 进行组织的 DI 的变长 cache。具体设计如下：

所有的 cache，采用 Codis 作为缓存，

6.8 与外部系统的接口

6.8.1 具体的命令处理

feedui 处理来自第三方业务层的请求，请求的命令字有四种: add，update，del 和 query。准备采用 PB 的方式作为提交报文，而用一个定长 Header 作为回复报文，除了 query 请求，回复数据实际上是一个空数据。

后端为了统一，会调整接收的命令字内容（将 uid 调整为 srctype 和 srcid），再转发给后端。

具体提交报文要求的格式如下：

- 添加命令（add）:

- 更改命令(update)：
- 删除命令(del)：
- 查询命令（query）

支持非聚类和聚类方式两种方式的查询请求，现有的查询命令如下：

对应于上面的查询请求，应答数据为:

此外还需要提供一个带有聚类功能的查询，此功能提供给推送以外以及使用。具体的查询参数如下：

对于上面的一个请求，其应答数据格式为：

综上，其实聚类查询和非聚类查询，其请求结构基本一致，只是命令号不同。

综上，其实聚类查询和非聚类查询，其请求结构基本一致，只是命令号不同。