# 如何构建 timeline

分类：数据库　标签：*Redis*

虽然我没有考证过，但至少从 Twitter 开始，timeline 这个词出现在互联网上也快 10 年了。这些年来，很多互联网产品依靠着各式各样的 timelines，吸引了无数的用户。
然而这篇文章并不是研究 timeline 的历史，而只是探索其实现方式而已。

虽然有着各种不同的 timelines，但它们大概都有这 3 个共同的特点：

1. 每个人的 timeline 都是个人独有的。
2. 由一系列按时间顺序排列的 items（条目）组成。
3. Timeline 中的条目有多种不同的来源。

其第一个特点正是它存在的价值：每个用户独有的 timeline，才是他最关心的 timeline。而这也意味着存储 timeline 的开销，是和用户量成正比的。

而第二个特点也就意味着它具备时效性，用户并不关心较老的条目，因此 timeline 的长度并不需要随时间无限增长。对大部分的互联网产品的 timeline 而言，几百条的 items 已经够用了(新浪微博目前是 450 条，其 API 只能获取 150 条；Twitter API 可以获取 3200 条)；另一种策略是按照时间截断（Google Reader 只展示一个月内的未读条目）。有限的条目量，更方便估算存储 feed 的开销，并且能带来更好的性能。
另外，因为用户只关心较新或者未读的条目，所以 timeline 的读写比例大致为 1:1。当活跃用户较少时，写操作甚至可能多于读操作。

它的第三个特点则是它复杂的原因，也导致了它的实现分成了两类模型：

- Pull（拉）模型：计算 timeline 时，从各个来源获取最新的条目，然后聚合起来。
- Push（推）模型：条目产生时，推送到其关注者的 timeline 中。

拉模型实现起来很容易，存储开销也少（不需要为每个用户单独保存一份数据），但因为获取时要实时计算，因此读性能比较差。
推模型的存储开销则比较大，而且在关注者较多时，推送的开销也很大，不过读性能很好，并且更容易实现实时更新和消息推送。
当然也可以把二者结合起来，对来源进行分类，那些关注者较多的来源用拉模型来获取，其余仍推到关注者的 timeline。只要这些来源不多（通常它们还被缓存了），对读性能的影响也并不算大，但极大减少了推的开销。
出于实现的简洁性和效率的考虑，最常用的还是推模型。这也就意味着开发者其实是很讨厌「大 V」的。无奈限制被关注者人数是很不合理的，因此只能通过限制关注人数来限制推的频率了。加上更新太频繁也会让用户疲倦，于是大多数互联网产品都限制了关注人数上限（一般为数千）。

分析完这三个特点，就可以开始设计了。
下面以新浪微博为例，看看如果用 MySQL InnoDB 来存储，应该怎么实现拉模型。

先建张 item 表：

```
CREATE TABLE `item` (
    `id` BIGINT UNSIGNED NOT NULL,
    `poster_id` BIGINT UNSIGNED NOT NULL,
    `content` VARCHAR(255),
    `status` TINYINT UNSIGNED NOT NULL,
    `created_at` INT UNSIGNED NOT NULL,
    PRIMARY KEY (`id`),
    KEY `poster_id__created_at` (`poster_id`, `created_at`)
) ENGINE=InnoDB;
```

额外的信息（例如用的什么客户端，带的图片等）就不列出了。发布微博时，只要往这个表插入一条数据即可。

再建张 friendship 表：

```
CREATE TABLE `friendship` (
    `id` BIGINT UNSIGNED NOT NULL,
```

```
    `follower_id` BIGINT UNSIGNED NOT NULL,
    `followee_id` BIGINT UNSIGNED NOT NULL,
    `created_at` INT UNSIGNED NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY (`follower_id`, `followee_id`),
    KEY `follower_id__created_at` (`follower_id`, `created_at`),
    KEY `followee_id__created_at` (`followee_id`, `created_at`)
) ENGINE=InnoDB;
```

获取 timeline 时，先读下 friendship 表中关注的用户：

```
SELECT followee_id FROM friendship where follower_id = :user_id;
```

然后再从 item 表中获取数据：

```
SELECT * FROM item where poster_id IN (:followee_ids) AND status = :public ORDER BY created_at DESC
LIMIT 0, 50;
```

简直简单得令人发指。

不过缺点也蛮明显的，假如关注了 1000 个用户，第二条语句将查询 1000 次，然后把结果合并起来，再进行排序。有常识的开发者都知道这样做一定慢到不行，除非没什么用户。
而如果要改进，似乎也只能用缓存了。Friendship 表中的数据可以按 follower_id 缓存，于是获取关注的用户变成了 O(1) 的操作；Item 表中的数据可以按 poster_id 缓存，于是获取 timeline 中的数据就变成了 O(N) 的操作（如果限制了最多关注 1000 人，则 N 最大为 1000）；最后在内存中进行合并和排序。现在 MySQL 基本没有查询压力了，只要应用服务器足够多，这仍然是个可用的方案。
考虑到 timeline 中只有较新的数据才经常被读取，所以缓存的量可以进行一些缩减。
而随着用户量的增长，item 和 friendship 表都需要进行分表。一个可能的方案是把 item 表按 poster_id 分表，同时也把近期的 items 按时间分表（当缓存失效时，可以加快查询）；friendship 表则可能需要分别对 follower_id 和 followee_id 分表，多存一份冗余的数据。

接着来实现推模型，仍然用 MySQL 来存储。

在拉模型的基础上，再建一张 timeline 表：

```
CREATE TABLE `timeline` (
    `id` BIGINT UNSIGNED NOT NULL,
    `user_id` BIGINT UNSIGNED NOT NULL,
    `item_id` BIGINT UNSIGNED NOT NULL,
    `created_at` INT UNSIGNED NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `user_id__item_id` (`user_id`, `item_id`),
    KEY `user_id__created_at` (`user_id`, `created_at`)
) ENGINE=InnoDB;
```

当用户发布了一条微博后，从 friendship 中获取所有的关注者，然后批量给他们塞入这条微博即可。而这个操作其实允许延迟和失败，所以只要塞到任务队列里即可。
而获取 timeline 则可以直接按 user_id 查询了：

```
SELECT item_id FROM timeline WHERE user_id = :user_id ORDER BY created_at DESC LIMIT 0, 50;
```

再根据 item_id，从 item 表获取详情即可。
随着时间的增长，timeline 表也需要分表，似乎没什么理由不按 user_id 分表。
另一个问题是限制 timeline 的长度。一个可行的方案是插入数据后，查询一下第 450 条的 created_at 或 id，然后删除小于该值的数据：

```
SELECT created_at FROM timeline WHERE user_id = :user_id ORDER BY created_at DESC LIMIT 450, 1;
DELETE FROM timeline WHERE user_id = :user_id AND created_at < :created_at;
```

不过这开销似乎太大了。
考虑到每次都只插入一条，所以如果每次都进行删除的话，最多只需要删除一条。因此可以用一个计数器记录是否达到了上限，然后再删除最老的一条：

```
DELETE FROM timeline WHERE user_id = :user_id ORDER BY created_at LIMIT 1;
```

相较于拉模型，推模型还需要多做两件事：

1. Item 被发布者删除或隐藏时，需要从他自己和所有关注者的 timeline 中删除。
2. 取消关注时，需要把关注者的 timeline 中所有发自该被关注者的 items 删除。

第一件事其实不做也可以，展示时再判断一下 status 即可。如果非要删除的话，可以找到所有关注者，进行该删除操作：

```
DELETE FROM timeline WHERE user_id IN (:user_ids) AND item_id = :item_id;
```

或者给 item_id 增加一条索引，然后找到所有相关的表，都进行该删除操作：

```
DELETE FROM timeline WHERE item_id = :item_id;
```

第二件事看上去并不太好做，如果可以忍受的话，遍历 timeline 中的所有 items 也是可行的，毕竟一个用户的 timeline 长度是有限的，只是一个 O(N) 的操作而已。
更好的做法是给 timeline 表增加一个冗余字段 poster_id，并增加一条 (user_id, poster_id) 的索引，就能这样删除了：

```
DELETE FROM timeline WHERE user_id = :user_id AND poster_id = :poster_id;
```

除了 MySQL，我接触得最多的是 Redis。而如果不考虑成本的话，用 Redis 来实现 timeline 是个非常好方案。

事实上，完全用 Redis 实现也没必要，毕竟 NoSQL 指的是「Not only SQL」。所以拉模型的那两张表可以仍然放在 MySQL 里，只用 Redis 来存储 timeline 即可。
因为附带了时间属性，所以数据结构方面几乎没有选择，用 sorted set 就对了：

```
key: timeline:user_id
member: item_id
score: created_at
```

插入：

```
ZADD timeline:user_id created_at item_id
```

获取第一页：

```
ZREVRANGE timeline:user_id 0 50 WITHSCORES
```

根据前一页拿到的 score，获取下一页：

```
ZREVRANGEBYSCORE timeline:user_id :created_at 0 WITHSCORES LIMIT 0 50
```

限制 timeline 长度：

```
ZREMRANGEBYRANK timeline:user_id 450 -1
```

删除条目：

```
ZREM timeline:user_id :item_id
```

取关比较麻烦，需要遍历 timeline，再查 MySQL 获取 poster_id 相符的 item，然后再删除。当然也有优化的办法，留到后面再介绍。
除了取关以外，简单程度简直堪比拉模型。
随着用户量的增长，需要按 user_id 做分片。

不过 Redis 的 sorted set 占用的内存很大，存储海量用户是比较昂贵的。要节约成本的话，还是用硬盘数据库比较合适。
而在众多的 NoSQL 数据库中，大部分只是 K/V 型的，没有 sorted set 这种极适合用于展示 timeline 的数据结构。
在剩下的选项中，Cassandra 可能是比较不错的选择。它的写性能比较好，适合读写比例比较接近 1:1 的场景。不足之处是查询上的限制很多，而且对事务的支持很弱。

仍然照老样子，Cassandra 只存 timeline 的数据（其实存 friendship 也很合适的，不过因为几乎是无痛迁移，没什么好介绍的）。
最容易想到的方式是把 user_id 作为 partition key，再和 item_id 一起组合成主键：

```
CREATE TABLE timeline (
    user_id BIGINT,
    item_id BIGINT,
    created_at timestamp,
    PRIMARY KEY ((user_id), item_id)
);
```

插入条目：

```
INSERT INTO timeline (user_id, item_id, created_at) VALUES(:user_id, :item_id, :created_at);
```

获取 feed：

```
SELECT item_id FROM timeline where user_id = :user_id LIMIT 50;
```

这里遇到了问题，没有办法按时间排序和按时间分页获取。

而 Cassandra 要实现排序的话，必须把要排序的字段放在 clustering columns 的最前面：

```
CREATE TABLE timeline (
    user_id bigint,
    item_id bigint,
    created_at timestamp,
    PRIMARY KEY ((user_id), created_at, item_id)
) WITH CLUSTERING ORDER BY (created_at DESC, item_id DESC);
```

这样获取数据时，就自动排好序了，而且也能按时间分页了：

```
SELECT item_id FROM timeline where user_id = :user_id AND created_at < :created_at LIMIT 50;
```

不过由于时间也成为了主键，(user_id, item_id) 的组合就不能保证唯一了。如果想查询某个用户的 timeline 中是否包含某个 item，这样是不行的：

```
SELECT * FROM timeline where user_id = :user_id AND item_id = :item_id;
```

原因是没有提供 created_at，而它在主键中的顺序是比 item_id 更靠前的。

要解决这个问题，就只能给 item_id 创建一条索引了：

```
CREATE INDEX ON timeline (item_id);
```

如果查询时都提供了 partition key（即 user_id）和 indexed column（即 item_id），那么性能应该不会有太大影响。
但即使如此，由于并发的存在，也没法保证 timeline 中不会出现重复的 (user_id, item_id) 组合，因此只好在读取时过滤掉重复的，并异步删除重复条目。

删除时需要给出所有的主键：

```
DELETE FROM timeline WHERE user_id = :user_id AND created_at = :created_at AND item_id = :item_id;
```

并且不能用不等于条件，例如 created_at < :created_at，这也就意味着删除老条目时，需要先把它们的主键都查出来。

取关也遇到了和 Redis 相同的问题。Cassandra 没法创建联合索引，所以不能像 MySQL 一样解决这个问题。
目前为止，还是先忽略这个问题，反正遍历一下也是可以解决的。

接下来面对一个严重的问题：世界上有那么多种 timelines，为什么我这给出的实现像是通用的？
于是我们来做点个性化的东西。新浪微博这些年对 timeline 做了些改动，例如：

- 把同一篇微博的多次转发合并在了一起，而不是让它出现多次。举例来说就是：用户 A 发布了微博 1，我关注的 B、C 和 D 都转发了这条微博，我的 timeline 里应该只出现一次微博 1，并且显示 B、C 和 D 都转发过。
- 在 timeline 中插推广的条目、用户和广告。

前一个需求用拉模型来实现的话，开销会非常大，所以就只考虑推模型了。后一个需求如果要固定位置，可以用拉模型；否则仍然可以用推模型；或者每天第一次访问首页时采用拉模型，然后再塞到 timeline 里。

选定了推模型后，再来看看这两个需求的共同点：都是针对 item 的变化。前者是来源相同的可以被合并，后者是类型从微博扩充到了包含用户和广告。于是很明显的，item 表需要重构了。
比较简单的做法是把存储微博的表改名为 status（新浪微博的 API 里就叫这个）；item 表则增加一个 type 字段，并引用 status 表的微博，或者是用户和广告；timeline 表则似乎不需要什么变化。然而这样设计的话，item 表就需要存储很多 status 表里的冗余字段，而且多了一层关系，会比较影响性能。
考虑到 timeline 表和 item 表都只是一些很简单的关系，所以把它们合在一个表里也是可行的：

```
CREATE TABLE `status` (
    `id` BIGINT UNSIGNED NOT NULL,
```

```
    `origin_id` BIGINT UNSIGNED,
    `content` VARCHAR(255),
    `status` TINYINT UNSIGNED NOT NULL,
    `created_at` INT UNSIGNED NOT NULL,
    PRIMARY KEY (`id`),
    KEY `poster_id__created_at` (`poster_id`, `created_at`),
    KEY `origin_id` (`origin_id`)
) ENGINE=InnoDB;

CREATE TABLE `timeline` (
    `id` BIGINT UNSIGNED NOT NULL,
    `type` TINYINT UNSIGNED NOT NULL,
    `user_id` BIGINT UNSIGNED NOT NULL,
    `item_id` BIGINT UNSIGNED NOT NULL,
    `sources` BLOB,
    `created_at` INT UNSIGNED NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `user_id__item_id` (`user_id`, `item_id`),
    KEY `user_id__created_at` (`user_id`, `created_at`)
) ENGINE=InnoDB;
```

首先是发微博，如果是转发的话，得补上 origin_id。而在塞入 timeline 时，type 用来区分是微博、用户或者广告，item_id 则是 status（或者 user 等）的 id。而在转发时，item_id 设为 origin_id，sources 中填充来源的数组，即 [(B, reposted_at_b), (C, reposted_at_c), (D, reposted_at_d)]。
由于 MySQL 不支持在一行内存储数组，所以这里就把 sources 设为 BLOB 类型了，可以随意用一种编码方式存储，比如 JSON。坏处是不能按 sources 来查询了，因此在取关时需要遍历了。如果再建一张 source 表，也能解决这个问题，但也会增加复杂度和降低性能。也许换成 PostgreSQL 这种支持数组的关系型数据库是更好的选择。
而在删除来源时，如果来源都空了，这个条目也应该清掉，所以原发的微博也需要填充 sources，以便判断。

再来看看 Redis 版需要怎么修改。
很显然的是，推模型是通过提前计算，来降低读取时的开销。也正是因此，MySQL 版本的实现把 sources 额外存储了一份，也就不需要在读取时进行合并了。
于是仿造 MySQL 版的实现，用 hash 来存储 sources：

```
key: sources:user_id
field: item_id
value: encoded_sources
```

然而 type 字段却没多余的地方放了。一种办法是把 timeline 的 member 改成 type:item_id 的形式，这会导致从整型变成字符串，极大增加了内存占用；另一种办法是把 item_id 的最高位作为 type，这使得最大可支持的元素数目会少于 $2^{63}$ / 10，并且可支持的类型不能超过 10（也可以把 10 改成 8，就没有浪费了，但运算会复杂点，可读性也不好）。
此外，由于引入了第 2 个 key，所以插入和删除时都需要用事务了，这使得实现变得稍微复杂些了。
之前所遇到的取关问题却有了新的解决方案，即遍历用户的 sources 记录，删除被取关的用户即可。虽然实现和 MySQL 版相同，但开销会小很多，所以可以接受了。

然后看看 Cassandra，它的解决方案似乎最简单，给 timeline 表增加两个字段即可：

```
CREATE TYPE source (
    user_id bigint,
    reposted_at timestamp
);

CREATE TABLE timeline (
    user_id bigint,
    type tinyint,
    item_id bigint,
    created_at timestamp,
    sources list<frozen <source>>,
    PRIMARY KEY ((user_id), created_at, type, item_id)
) WITH CLUSTERING ORDER BY (created_at DESC);
```

但使用起来并不省心，例如 sources 字段并不支持查询，所以取关时还是得遍历。
而且和之前相比，多了 sources 字段的更新操作，因此并发时更容易遇到问题了。

增加来源：

```
UPDATE timeline SET sources = sources + [(:from_user_id, :reposted_at)] WHERE user_id = :user_id AND
created_at = :created_at AND type = :STATUS AND item_id = :item_id;
```

这个语句也可能导致插入了一行新记录，而且并没有返回值来区分。当有并发的更新、插入和删除存在时，可能会出现重复的 (user_id, type, item_id) 组合，所以更新前后，最好都检查一下是否有重复。

删除来源：

```
UPDATE timeline SET sources = sources - [(:from_user_id, :reposted_at)] WHERE user_id = :user_id AND
created_at = :created_at AND type = :STATUS AND item_id = :item_id;
```

如果想在删除来源后更新时间，则需要删除后再插入一条，因为 created_at 属于主键的一部分，不能被修改。

删除空来源的行：

```
DELETE FROM timeline WHERE user_id = :user_id AND created_at = :created_at AND type = :STATUS AND
item_id = :item_id IF sources = [];
```

很遗憾这条命令不能和删除来源组成一个批处理，因为它们没有执行的先后顺序，而第二条会在 sources 不为空时失败，导致第一条也被回滚。

由于 Cassandra 的诸多限制，导致用它来实现 timeline 其实是有很多坑的。也许比较好的方式是写的时候不做过多的检查，读的时候才进行检查，并且异步修复有问题的数据（短期内的读可以不用再次检查）。毕竟 timeline 里的数据即使出现了一些问题，用户也基本不会发现。

最后，如果还有精力的话，可以做下冷热分离，把活跃用户的数据放 Redis，不活跃用户的数据放 Cassandra。
由于数据都是按 user_id 组织的，所以迁移还算好实现。但在迁移过程中，并发的写操作可能会造成不一致，要解决这个问题可能会影响性能，所以并不算很好的解决办法。
另一种思路是不管冷热都放 Cassandra，但热数据同时也放在 Redis 里。这样只有数据加载到 Redis 时可能会有不一致的问题，但这种错误基本可以忽略，并且至少 Cassandra 中是好的，而且不影响读性能，只是稍微浪费了点。