谈谈在线表格协同方案

pedrogao | 7月14日创建 | 1 12 0 0

背景

在线协同表格允许多个用户同时在同一个表格中编辑数据和进行其他相关操作，帮助团队成员实时合作、共享信息和协作解决问题。

因为功能强大，所以也带来了很大的复杂性，是目前前端工程化的天花板，当然对后端而言也是一个巨大的挑战，其中最大的挑战就是表格之间的协同。

因为工作原因，有幸参与公司协同表格的调研攻坚项目，踩了很多坑同时也收获很多，同时也沉淀出一份设计文档。

术语

- OT：Operational Transformation 操作转换，解决用户操作冲突的一类算法；
- OP：表格原子操作 Operation；
- 快照：表格完整数据快照；
- OP应用：对表格快照应用OP，得到一个新的快照 S1 = apply(S0, OP)；
- transform：OP转换操作，将有冲突的OP转换为无冲突的OP'；
- OP'：OP 转换后得到 OP'；

OT协同原理

表格A（图1）能够同时被两个用户打开，用户1和用户2分别操作表格A，操作类型为：

- 用户1将表格第1行内容移动到表格第2行（moveRow(0, 1)），如图2；
- 用户2将单元格A1内容更新为'xxxx'（updateCell(0, 0, 'xxxx')），如图3；

	A	B	C
1	1	2	3
2			

图1-原始表格A

	A	B	C
1			
2	1	2	3

图2-表格A-moveRow(0, 1)

	A	B	C
1	xxxx		
2			

图3-表格A-updateCell(0, 0, 'xxxx')

假设用户1、2在不同的客户端同时操作，由于网络通信的不稳定，两个操作达到服务端的先后顺序无法预测，所以最后会产生两个不同的结果：

场景1：moveRow(0, 1)先到，updateCell(0, 0, 'xxxx')后到

	A	B	C
1	1	2	3
2			

	A	B	C
1			
2	1	2	3

	A	B	C
1	xxxx		
2	1	2	3

场景2：updateCell(0, 0, 'xxxx')先到，moveRow(0, 1)后到

	A	B	C
1	1	2	3
2			

	A	B	C
1	xxxx	2	3
2			

	A	B	C
1			
2	xxxx	2	3

因为 moveRow(0, 1) 与 updateCell(0, 0, 'xxxx') 这两个操作存在冲突，所以两个操作达到服务端的先后顺序不同，最后产生的结果就完全不同了。

那么如何解决掉这两个操作的冲突了？容易想到的方案有两个：

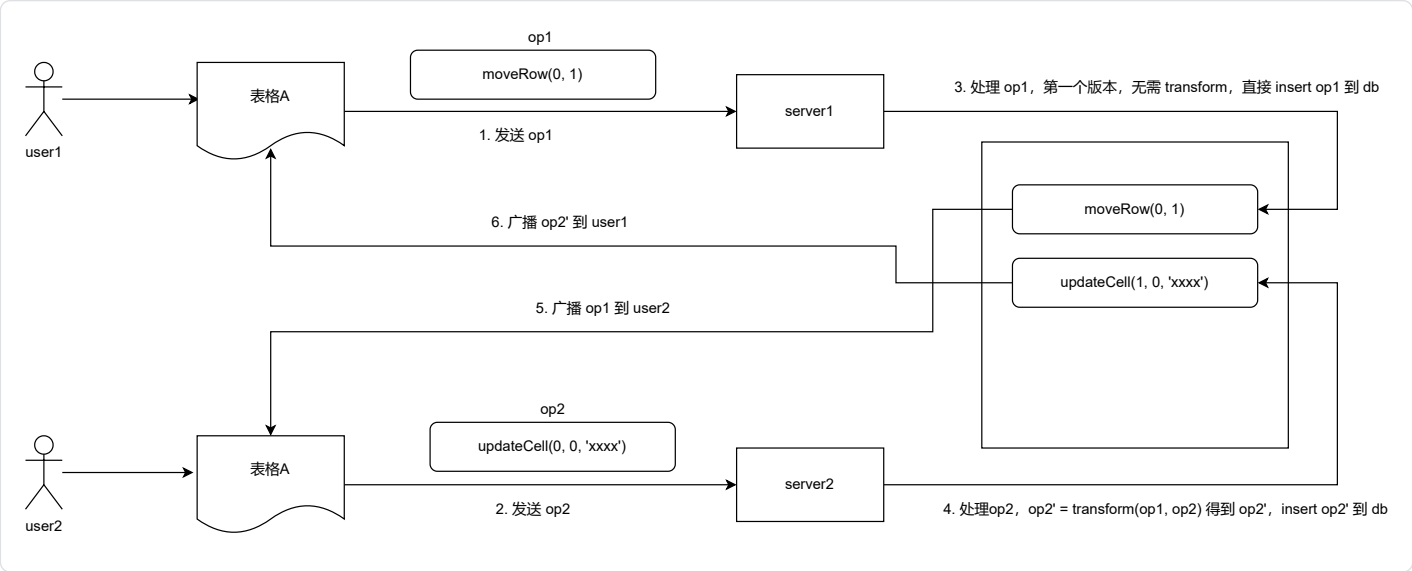
方案	细节	优点	缺点
1. 前端加锁	• 前端操作表格时，对所操作行、列进行加锁；	• 方案简单，后端只需提供加、解锁接口即可；	

	• 此时其它客户端不可对相同行列进行操作； • 操作完毕后提交，再解锁，其它客户端再操作；		• 加、解锁方案简单，但实现有很多 corner case，比如锁超时处理，网页断网后如何处理锁，锁不可解如何处理等等； • 体验差，一个客户端编辑时，其它所有客户端只能等待；
2．后端解决冲突	• 前端允许多个客户端在同一行列进行操作； • 操作到达后端后，服务器解决操作之间的冲突，并再将正确的解冲突操作返回给前端；	• 允许多个客户端同时操作；	• 后端需要解决操作冲突，并将正确解冲突操作广播至前端再处理； • 解冲突算法实现复杂；

对比两个方案，毫无疑问，我们得选择方案2，而这方案2其实就是OT算法协同的雏形。

下面，我们代入OT算法来看看如何解决用户1、2对表格A的操作冲突。

OT下场景1，moveRow(0, 1)先到，updateCell(0, 0, 'xxxx')后到：



1．用户1将表格第1行内容移动到表格第2行，op1= moveRow(0, 1) 到达服务端，由于是第一个op，因此无需 transform，直接将 op1 存储到数据库中，此时服务端表格内容为：

	A	B	C
1			
2	1	2	3

2．用户2将单元格A1内容更新为'xxxx'，op2 = updateCell(0, 0, 'xxxx') 到达服务端，已有 op1，得到 op2' = updateCell(0, 1, 'xxxx') = transform(op1, op2)，将 op2' 存储到数据中，此时服务端表格内容为：

	A	B	C
1			
2	xxxx	2	3

3．最后将 op1，op2'广播到客户端，所有表格内容收敛到终态。

OT下场景2，updateCell(0, 0, 'xxxx')先到，moveRow(0, 1)后到：

1．用户2将单元格A1内容更新为'xxxx'，op1 = updateCell(0, 0, 'xxxx') 到达服务端 ，由于是第一个op，因此无需 transform，直接将 op1 存储到数据库中，此时服务端表格内容为：

	A	B	C
1	xxxx	2	3
2			

2. 用户1将表格第1行内容移动到表格第2行，op2= moveRow(0, 1) 到达服务端，已有 op1，得到 op2' = moveRow(0, 1) = transform(op1, op2)，将 op2' 存储到数据中，此时服务端表格内容为：

	A	B	C
1			
2	xxxx	2	3

3. 最后将 op1，op2'广播到客户端，所有表格内容收敛到终态。

在使用OT后，无论 op1、op2以何种顺序到达服务端，最后得到的表格内容一定是最终一致的，即使表格在 op 应用的过程中内容有区别。

整体设计

设计目标

> 提供一个易用稳定、高可靠、高性能的OT协同平台，支持但不限于表格协同。

功能性诉求：

OP查询、存储：单/批量查询OP，存储OP；

OP转换：转换并发OP，解决OP之间冲突；

快照生成、存储：表格应用OP，生成快照并存储；

OP提交、ACK：客户端提交OP，转化得到OP'后存储并ACK；

OP广播：广播 OP' 到其它客户端，从而达到多端协同效果；

非功能性诉求：

> 在满足核心功能的基础上提供以下非功能性保障

平台化：OT协同标准化，支持多OT类型接入；

易用性：自助OT接入，使用成本远低自建；

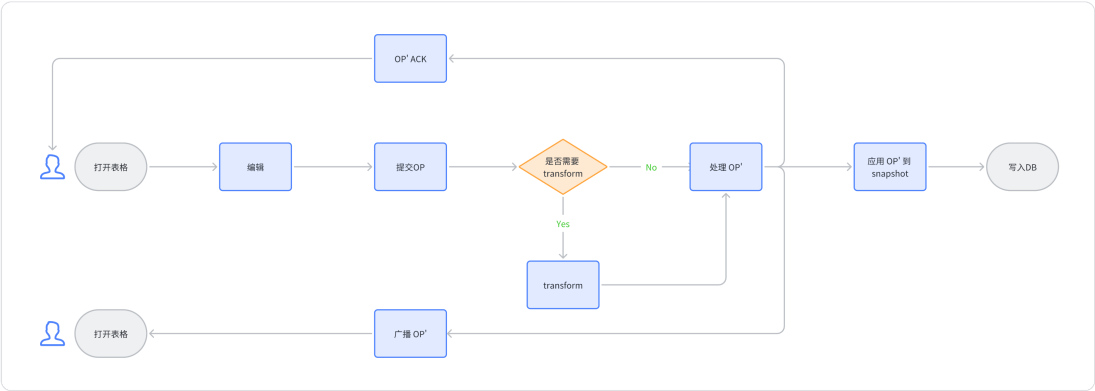
可观测性：OP消息全链路追踪，方便内容丢失排查；

高性能：支持百人同时编辑；

设计思路

> 优秀的方案是一步步思考、推演得来的，我们先设计一版满足所有功能性诉求的基础版，然后再按照非功能性需求来逐渐优化。

基础版



上图描述了OT协同整体流程：

- OP提交：客户端提交OP、等待ACK；
- OP转换：服务端转换并发OP，生成无冲突的OP'；
- OP存储、应用、快照生成：存储客户端提交并转换后的OP'，应用OP到快照，然后存储最新快照；
- OP ACK：服务端收到OP提交，并转换成功后返回客户端ACK；
- OP广播：客户端提交OP转换成功得到OP'后，将OP'广播到其它客户端；

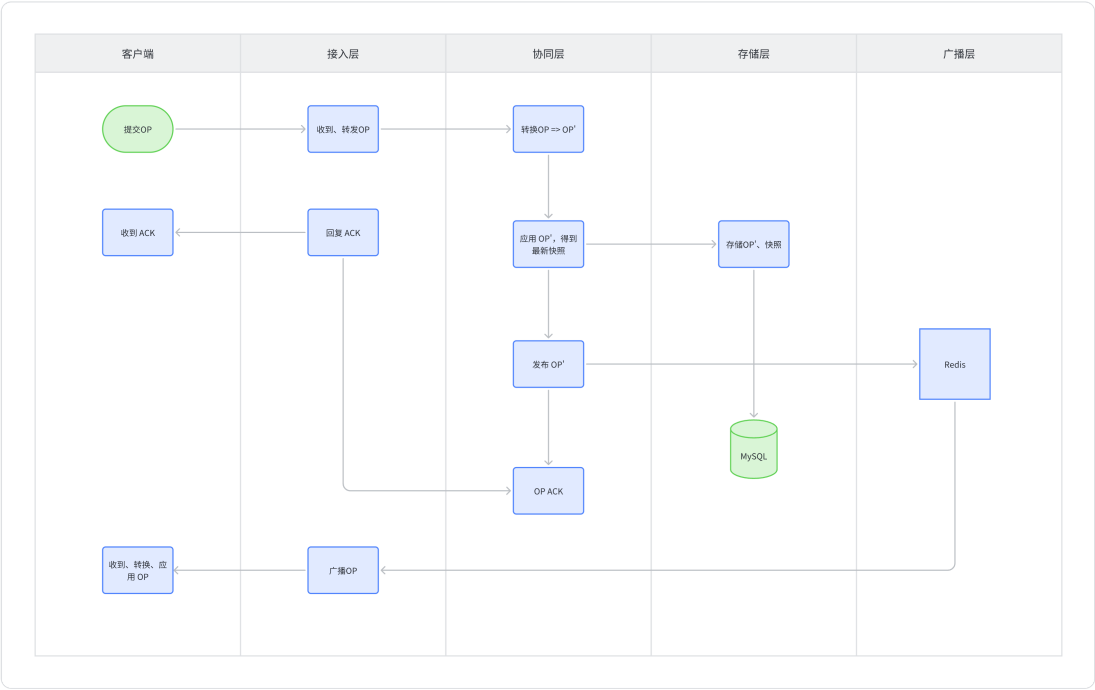
从流程中可以看出，OT协同需要以下核心组件：

- 客户端：监听用户表格操作，产生提交OP、处理OP ACK、接收并应用广播而来的OP；
- 接入层：管理客户端长连接、转发用户提交OP、回复OP ACK、订阅并广播OP；
- 协同层：处理OP、转换OP、发布OP、应用OP；
- 存储层：存储OP、查询OP、存储快照、查询快照；
- 广播层：发布、订阅OP通道；

从核心组件的功能需求上，做出基础设施选型：

1. 由于OP提交、ACK、广播十分频繁，客户端、接入层之间通过**websocket长连接**通信；
2. 客户端需要担当生成、提交、缓存、转换OP的任务；
3. 接入层管理整个websocket长连接生命周期，订阅广播层发布的OP，然后将其发布到客户端；
4. 协同层负责处理、转化OP生成OP'，将其提交到存储持久化，并发布至广播层；
5. 广播层担当OP发布订阅的角色，由于接入层、协同层都是多个节点，因此广播需要跨节点广播，选择 **Redis pub/sub**来实现；
6. 服务端含多层，每层之间通过**RPC**通信；
7. OP、快照持久化选择 **MySQL**；

整理后，协同流程图如下：



协同基础版能够满足一般协同场景下，最核心的部分在于：

- 客户端OP处理：
  - 监听表格事件，提交OP；
  - OP提交后，如果未ACK，则需要将后续产生的OP缓存到本地客户端，如果成功ACK，再提交新的OP；
  - 收到广播OP后，转换本地未提交OP，然后应用到快照；
- 协同层OP转换：
  - 收到OP后，找到并发OP，然后转换得到无冲突的OP'；
  - OP ACK告知客户端OP提交成功；
  - OP 广播通知其它客户端更新表格；

客户端、协同层能够正确处理OP、快照就能让协同正确的走下去，满足基本场景下的多人编辑是没问题的。

优化版

> 👍 优化1：客户端提交OP后，在OP未ACK的情况下，用户继续编辑产生OP，会缓存OP列表到本地客户端；如果ACK后，OP列表一个个提交，然后一个个ACK，OP提交效率低下；如果将本地缓存的多个OP合并(compose)成一个OP然后提交。

- 在ACK慢的情况下，提升客户端整体OP提交效率。

> 👍 优化2：OP、快照查询引入内存、Redis二级缓存提升整体查询吞吐量。

- 提升OP、快照查询吞吐量，避免直接打到数据库；
- OP缓存在内存中，提升整体 transform 效率；

> 👍 优化3：OP提交转发 RPC 使用一致性哈希，让一个表格的所有OP提交、应用在一个协同层节点处理，充分利用已有的OP内存缓存列表；

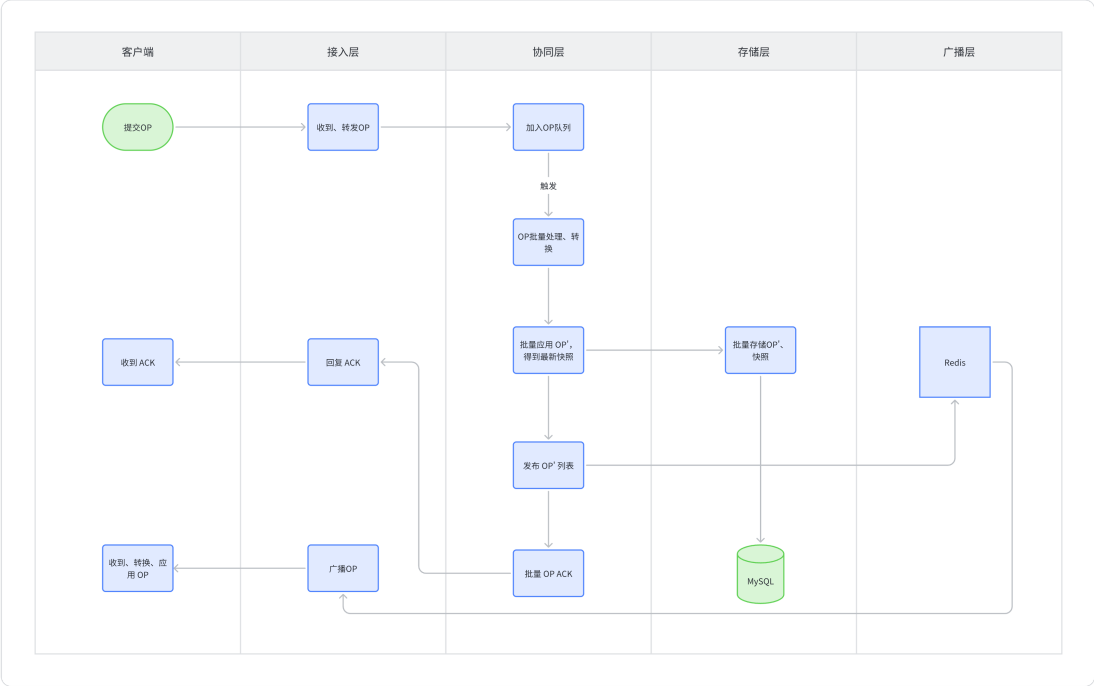
- 一个表格的OP、快照都缓存在一个节点上，避免多个节点缓存造成浪费；
- 多节点滚动发布、节点规模变更时，一致性哈希选择会出现变化，

> 🚫 **优化4：异步ACK，批量OP转换、存储、广播；**

> 优化4需要对后端进行大量改造，理论上是正确的，但实际上没有压测，是否提升吞吐量有待进一步确认。

- OP提交后，不再线性处理完成后立马ACK，而是先加入OP队列中，然后触发 OP 批量 transform；
- 允许客户端并发提交OP，协同层批量处理，提升多个客户端吞吐量；
- OP批量 transform 后，持久化、广播都是批量，效率提升；

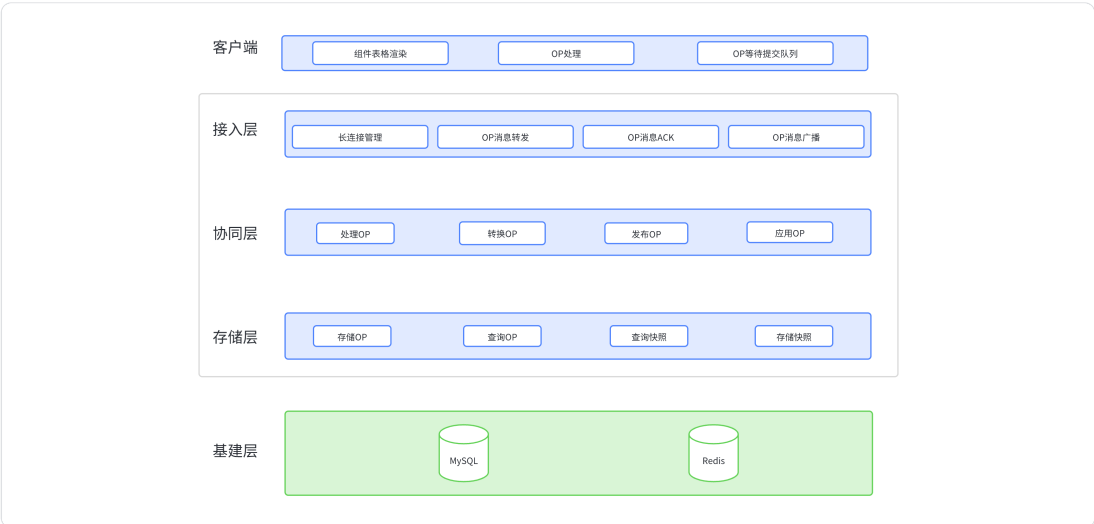
优化4改造后，整体流程如下：



详细设计

分层模型

按照整体设计，分为客户端、服务端、中间件三个大模块。



客户端：

视图层：渲染表格等组件

OP处理：生成OP、提交OP、处理OP及OP等待队列；

服务端3层：

接入层：管理客户端长连接、转发、回复、广播 OP 消息；

协同层：处理、转换、发布、应用OP；

存储层：存储、查询OP和快照；

中间件：

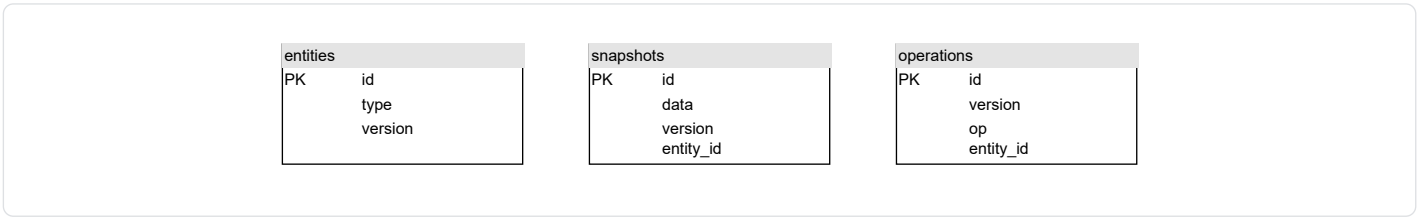
MySQL：快照，OP持久化；

Redis-广播：OP广播；

Redis-缓存：快照，OP查询缓存；

分库分表

由于后续需要支持不限于表格的多种在线协同应用，整体需求量又很大，因此在项目初期就直接定下分库分表。

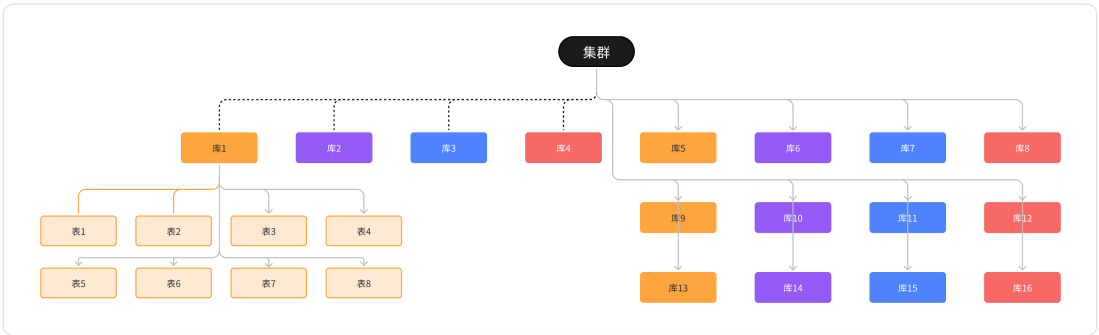


图中给出了核心的三个表：

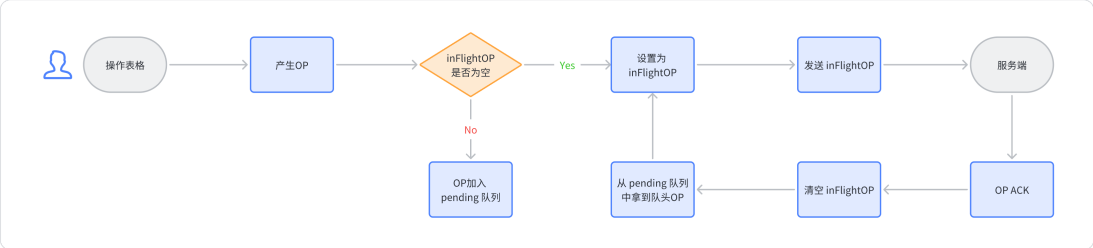
- entities：协同应用
  - type：应用类型，如表格、文档等；
  - version：应用的最新版本，每次OP应用后新增一个版本；
- snapshorts：应用数据快照
  - entity_id：应用ID；
  - data：快照数据
  - version：快照版本
- operations：操作记录
  - entity_id：应用ID；
  - op：操作记录内容
  - version：操作记录版本

三个表按照 entity_id 来哈希拆分，拆分到128张表中，如果分布均匀的话，能够支持十亿级数据，每张表都能保障索引效率和查询性能。

由于表格数据是核心数据，数据库部署为1主2备，本地1备，异地1备；逻辑上虽然拆成了128张表，但整体上部署在一个集群中，8张表1个物理库，共16个库。



客户端OP处理



OP提交、ACK：

1. 操作表格产生OP1，检查 inFightOP 是否为空；
    a. 如果为空，则直接将 OP1 设置为 inFightOP 然后发送；
    b. 如果不为空，则将OP1推入OP等待提交(pending)队列；
2. 发送 inFightOP，等待ACK；
3. 服务端 ACK inFightOP 后，清空当前 inFightOP，检查pending队列；
    a. 如果对队列为空，则无待处理OP，结束；
    b. 否则将pending队列队头OP设置为inFightOP，在进行步骤2；

OP transform/批量 transform

> 😃 transform 的性能和吞吐量直接决定了整体OT协同的性能和吞吐量，是整个协同的关键。

transform原理其实就是上述的OT协同原理，由于OP的版本增长必须是线性的，如下：

线性版本增长有两个好处：

- 保证了OP transform 的正确性；
- 保证了已处理 OP 是绝对无冲突的；

但同时也有一些麻烦：

- OP版本递增必须是串行化的；
- 出现并发OP时，必须得先后transform处理，然后新增版本并持久化；

这样，在服务端的OP列表(主节点)在 transform 上必须是串行化，针对后端协同的优化点可以分为如下两类：

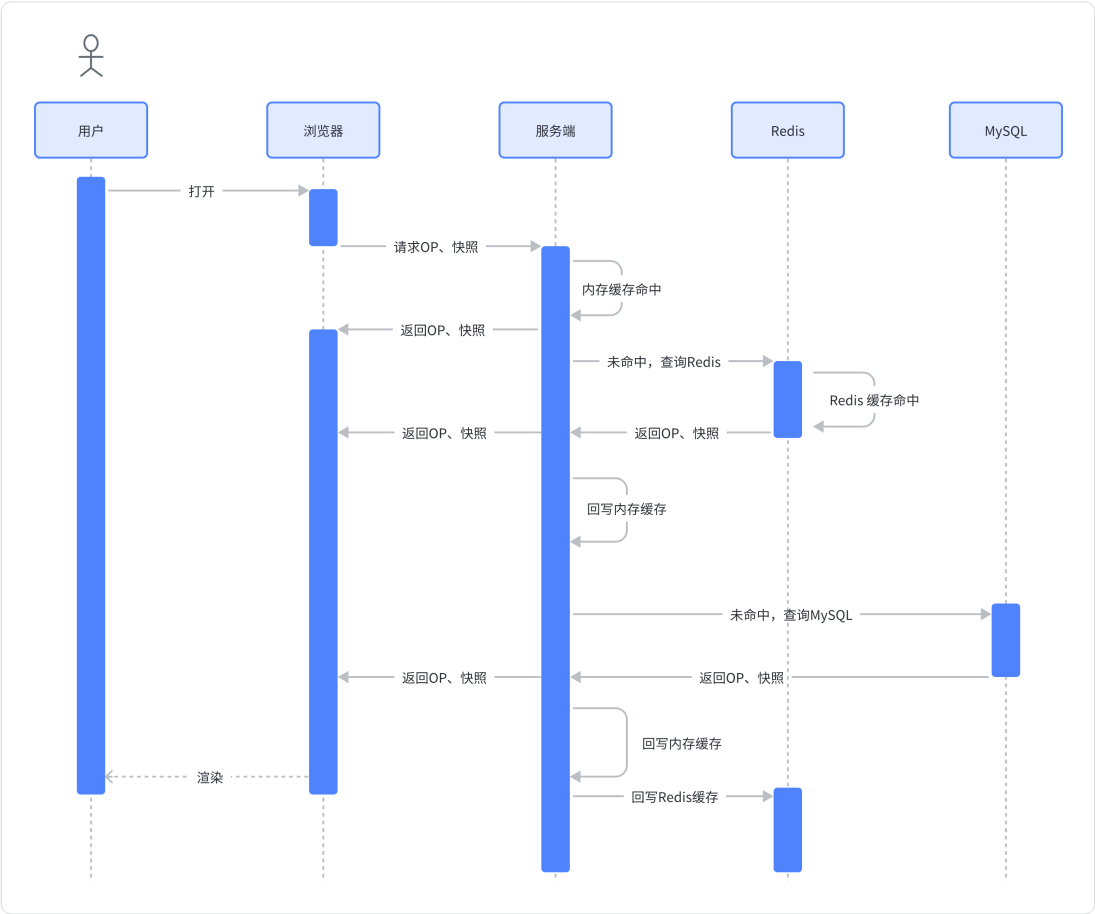
- 提升 transform 效率，基本在几毫秒的水位线上；
- 优化 OP 提交、处理、持久化的性能上；

批量 transform 不是让多个OP 同时 transform，而是批量提交、持久化、广播从而提升整体的吞吐量。

另外 OP 做 transform 时只有相邻的并发OP必须相互等待的，而其它已有的OP是无非等待的，完全可以并行 transform，等到了有OP冲突的时候才需要串行化处理，极致优化下，服务端每秒吞吐量，几乎就是 transform 的每秒吞吐量。

二级缓存

查询OP、快照是一个十分高频的操作，虽然有分库分表来兜底查询性能，但在极端高频的查询情况下，数据库是脆弱的，为了保证数据库的安全性，留更多的空间给OP、快照存储，因此需要二级缓存来帮MySQL来抗住查询压力。如下图所示：



查询优先走内存缓存，内存没有再查询Redis，Redis没有再查询MySQL，查到后再回写到内存和Redis，最大限度的提升查询QPS。

如果快照发生了更新，可以采取延迟双删的方式来删除缓存旧数据，或者采取延迟双刷的方式来刷新缓存。

总结

多学习、多总结、多沉淀，遇到问题多下沉思考。以上就是我的总结。

参考资料

- GitHub - share/sharedb: Realtime database backend based on Operational Transformation (OT)
- Operational-Transformation
- GitHub - ottypes/docs