07 | 分布式锁和原子性：你看到的未读消息提醒是真的吗？

2019-09-11 袁武林

即时消息技术剖析与实战 进入课程 >

讲述：袁武林
时长 13:05　大小 11.98M

你好，我是袁武林。

在前面几节课程中，我着重把即时消息场景中几个核心的特性，进行了较为详细的讲解。在实际用户场景下，除了实时性、可靠性、一致性、安全性这些刚需外，还有很多功能对用户体验的影响也是很大的，比如今天我要讲的“消息未读数”。

消息未读数对用户使用体验影响很大，这是因为“未读数”是一种强提醒方式，它通过 App 角标，或者 App 内部 Tab 的数字标签，来告诉用户收到了新的消息。

对于在多个社交 App 来回切换的重度用户来说，基本上都是靠“未读数”来获取新消息事件，如果“未读数”不准确，会对用户造成不必要的困扰。

比如，我们看到某个 App 有一条“未读消息提醒”，点进去事件却没有，这种情况对于“强迫症患者”实在属于不可接受；或者本来有了新的消息，但未读数错误，导致没有提醒到用户，这种情况可能会导致用户错过一些重要的消息，严重降低用户的使用体验。所以，从这里我们可以看出“消息未读数”在整个消息触达用户路径中的重要地位。

消息和未读不一致的原因

那么在即时消息场景中，究竟会有哪些情况导致消息和未读数出现“不一致”的情况呢？要搞清楚这个问题，我们要先了解两个涉及未读数的概念：“总未读”与“会话未读”。我们分别来看看以下两个概念。

会话未读：当前用户和某一个聊天方的未读消息数。比如用户 A 收到了用户 B 的 2 条消息，这时，对于用户 A 来说，他和用户 B 的会话未读就是“2”，当用户 A 打开和用户 B 的聊天对话页查看这两条消息时，对于用户 A 来说，他和用户 B 的会话未读就变成 0 了。对于群聊或者直播间来说也是一样的逻辑，会话未读的对端只不过是一个群或者一个房间。

总未读：当前用户的所有未读消息数，这个不难理解，总未读其实就是所有会话未读的和。比如用户 A 除了收到用户 B 的 2 条消息，还收到了用户 C 的 3 条消息。那么，对于用户 A 来说，总未读就是“5”。如果用户查看了用户 B 发给他的 2 条消息，这时用户 A 的总未读就变成了“3”。

从上面的概念我们知道，实际上总未读数就是所有会话未读数的总和，那么，在实现上是不是只需要给每个用户维护一套会话未读就可以了呢？

会话未读和总未读单独维护

理论上是可以的。但在很多即时消息的“未读数”实现中，会话未读数和总未读数一般都是单独维护的。

原因在于“总未读”在很多业务场景里会被高频使用，比如每次消息推送需要把总未读带上用于角标未读展示。

另外，有些 App 内会通过定时轮询的方式来同步客户端和服务端的总未读数，比如微博的消息栏总未读不仅包括即时消息相关的消息数，还包括其他一些业务通知的未读数，所以通过消息推送到达后的累加来计算总未读，并不是很准确，而是换了另外一种方式，通过轮询来同步总未读。

对于高频使用的“总未读”，如果每次都通过聚合所有会话未读来获取，用户的互动会话不多的话，性能还可以保证；一旦会话数比较多，由于需要多次从存储获取，容易出现某些会话未读由于超时等原因没取到，导致总未读数计算少了。

而且，多次获取累加的操作在性能上比较容易出现瓶颈。所以，出于以上考虑，总未读数和会话未读数一般是单独维护的。

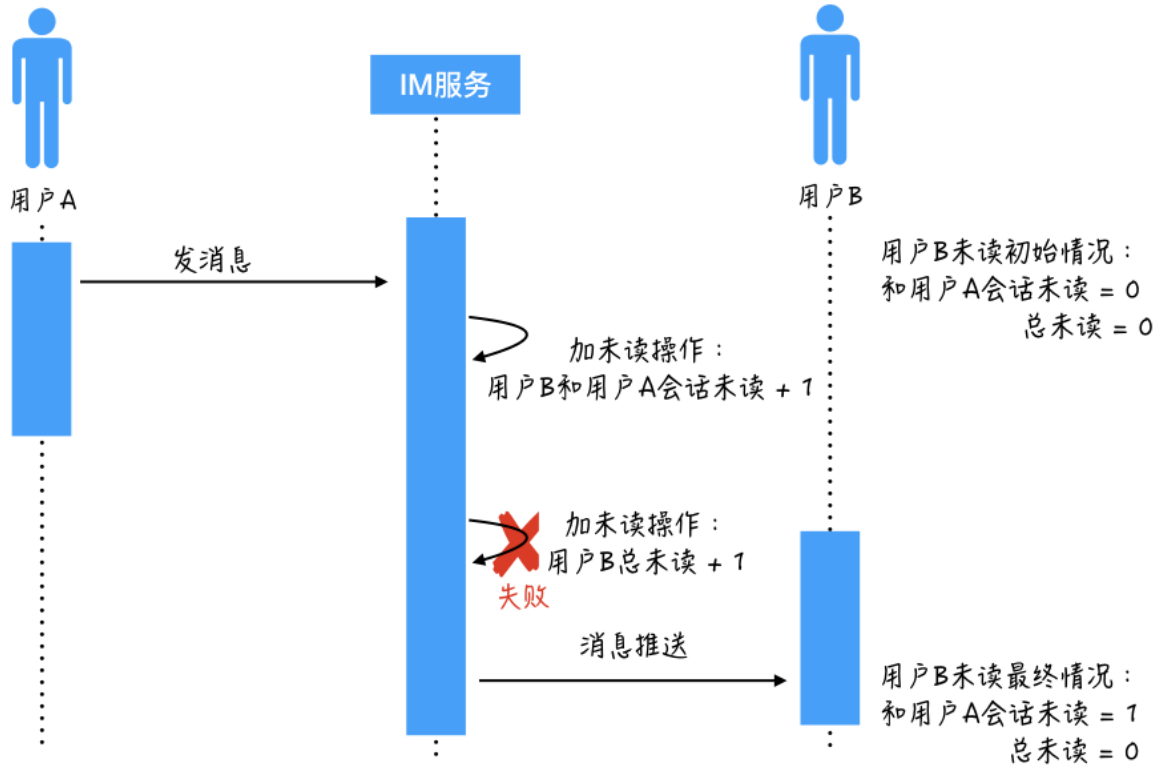
未读数的一致性问题

单独维护总未读和会话未读能解决总未读被“高频”访问的性能问题，但同时也会带来新的问题：未读数的一致性。

未读数一致性是指：**维护的总未读数和会话未读数的总和要保持一致。**如果两个未读数不能保持一致，就会出现“收到新消息，但角标和 App 里的消息栏没有未读提醒”，或者“有未读提醒，点进去找不到是哪个会话有新消息”的情况。

这两种异常情况都是我们不愿意看到的。那么这些异常情况究竟是怎么出现的呢？

我们来看看案例，我们先来看看第一个：



用户 A 给用户 B 发送消息，用户 B 的初始未读状态是：和用户 A 的会话未读是 0，总未读也是 0。

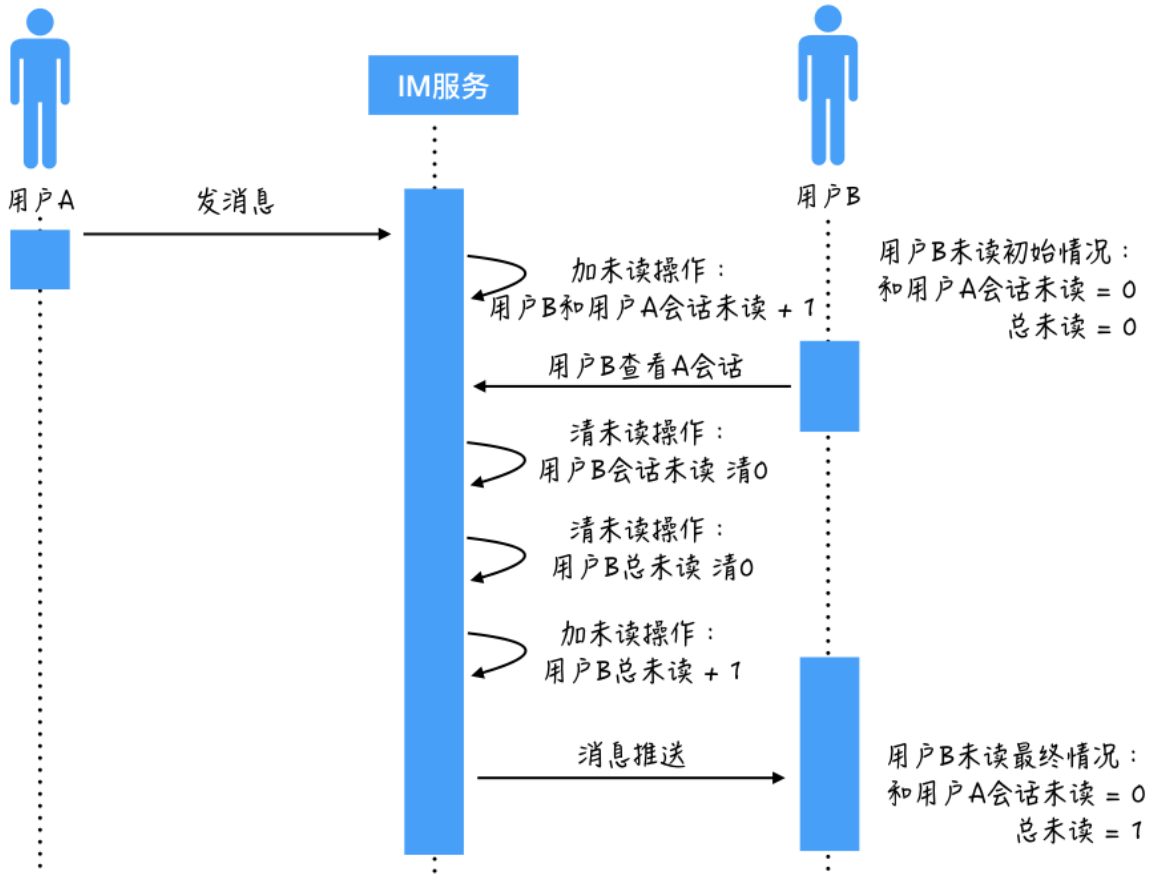
消息到达 IM 服务后，执行加未读操作：先把用户 B 和用户 A 的会话未读加 1，再把用户 B 的总未读加 1。

假设加未读操作第一步成功了，第二步失败。最后 IM 服务把消息推送给用户 B。这个时候用户 B 的未读状态是：和用户 A 的会话未读是 1，总未读是 0。

这样，由于加未读第二步执行失败导致的后果是：用户 B 不知道收到了一条新消息的情况，从而可能漏掉查看这条消息。

那么案例是由于在加未读的第二步“加总未读”的时候出现异常，导致未读和消息不一致的情况。

那么，是不是只要加未读操作都正常执行就没有问题了呢？**接下来，我们再看下第二个案例。**



用户 A 给用户 B 发送消息，用户 B 的初始未读状态是：和用户 A 的会话未读是 0，总未读也是 0。

消息到达 IM 服务后，执行加未读操作：先执行加未读的第一步，把用户 B 和用户 A 的会话未读加 1。

这时执行加未读操作的服务器由于某些原因变慢了，恰好这时用户 B 在 App 上点击查看和用户 A 的聊天会话，从而触发了清未读操作。

执行清未读第一步，把用户 B 和用户 A 的会话未读清 0，然后继续执行清未读第二步，把用户 B 的总未读也清 0。

清未读的操作都执行完之后，执行加未读操作的服务器才继续恢复执行加未读的第二步，把用户 B 的总未读加 1，那么这个时候就出现了两个未读不一致的情况。

导致的后果是：用户 B 退出会话后，看到有一条未读消息，但是点进去却找不到是哪个聊天会话有未读消息。

这里，我来分析一下这两种不一致的案例原因：其实都是因为两个未读的变更不是原子性的，会出现某一个成功另一个失败的情况，也会出现由于并发更新导致操作被覆盖的情况。所以要解决这些问题，需要保证两个未读更新操作的原子性。

保证未读更新的原子性

那么，在分布式场景下，如何保证两个未读的“原子更新”呢？一个比较常见的方案是使用一个分布式锁来解决，每次修改前先加锁，都变更完后再解开。

分布式锁

分布式锁的实现有很多，比如，依赖 DB 的唯一性、约束来通过某一条固定记录的插入成功与否，来判断锁的获取。也可以通过一些分布式缓存来实现，比如 MC 的 add、比如 Redis 的 setNX 等。具体实现机制，我这里就不细讲了，在我们的实战课程中，我们会有相应的代码体现。

不过，要注意的是，分布式锁也存在它自己的问题。由于需要增加一套新的资源访问逻辑，锁的引入会降低吞吐；同时对锁的管理和异常的处理容易出现 Bug，比如需要资源的单点问题、需要考虑宕机情况下如何保证锁最终能释放。

支持事务功能的资源

除了分布式锁外，还可以通过一些支持事务功能的资源，来保证两个未读的更新原子性。

事务提供了一种“将多个命令打包， 然后一次性按顺序地执行”的机制， 并且事务在执行的期间不会主动中断，服务器在执行完事务中的所有命令之后，才会继续处理其他客户端的其他命令。

比如：Redis 通过 MULTI、DISCARD 、EXEC 和 WATCH 四个命令来支持事务操作。

比如每次变更未读前先 watch 要修改的 key，然后事务执行变更会话未读和变更总未读的操作，如果在最终执行事务时被 watch 的两个未读的 key 的值已经被修改过，那么本次事务会失败，业务层还可以继续重试直到事务变更成功。

依托 Redis 这种支持事务功能的资源，如果未读数本身就存在这个资源里，是能比较简单地做到两个未读数“原子变更”的。

但这个方案在性能上还是存在一定的问题，由于 watch 操作实际是一个乐观锁策略，对于未读变更较频繁的场景下（比如一个很火的群里大家发言很频繁），可能需要多次重试才可以最终执行成功，这种情况下执行效率低，性能上也会比较差。

原子化嵌入脚本

那么有没有性能不错还能支持”原子变更“的方案呢？

其实在很多资源的特性中，都支持”原子化的嵌入脚本“来满足业务上对多条记录变更高一致性的需求。Redis 就支持通过嵌入 Lua 脚本来原子化执行多条语句，利用这个特性，我们就可以在 Lua 脚本中实现总未读和会话未读的原子化变更，而且还能实现一些比较复杂的未读变更逻辑。

比如，有的未读数我们不希望一直存在而干扰到用户，如果用户 7 天没有查看清除未读，这个未读可以过期失效，这种业务逻辑就比较方便地使用 Lua 脚本来实现“读时判断过期并清除”。

原子化嵌入脚本不仅可以在实现复杂业务逻辑的基础上，来提供原子化的保障，相对于前面分布式锁和 watch 事务的方案，在执行性能上也更胜一筹。

不过这里要注意的是，由于 Redis 本身是服务端单线程模型，Lua 脚本中尽量不要有远程访问和其他耗时的操作，以免长时间悬挂（Hang）住，导致整个资源不可用。

小结

本节课我们先了解了未读数在即时消息场景中的重要性，然后分析了造成未读数和消息不一致的原因，原因主要在于：“总未读数”和“会话未读数”在大部分业务场景中需要能够独立维护，但两个未读数的变更存在成功率不一致和并发场景下互相覆盖的情况。

接下来我们探讨了几种保证未读数原子化变更的方案，以及深入分析了每种方案各自的优劣，三种方案分别为：

- **分布式锁**，具备较好普适性，但执行效率较差，锁的管理也比较复杂，适用于较小规模的即时消息场景；
- **支持事务功能的资源**，不需要额外的维护锁的资源，实现较为简单，但基于乐观锁的 watch 机制在较高并发场景下失败率较高，执行效率比较容易出现瓶颈；

原子化嵌入脚本，不需要额外的维护锁的资源，高并发场景下性能也较好，嵌入脚本的开发需要一些额外的学习成本。

这一篇我们讲到的内容，简单来看只是消息未读数不一致的场景，但是，如果我们站在宏观视角下，不难看出在分布式场景下，这其实是一个并发更新的问题。

不管是分布式锁、还是支持事务功能的资源，以及我们最后讲到的原子化的嵌入脚本，其实不仅仅可以用来解决未读数的问题，对于通用的分布式场景下涉及的需要保证多个操作的原子性或者事务性的时候，这些都是可以作为可选方案来考虑的。

最后给你留一个思考题：类似 Redis+Lua 的原子化嵌入脚本的方式，是否真的能够做到“万无一失”的变更一致性？比如，执行过程中机器掉电会出现问题吗？

你可以给我留言，我们一起讨论，感谢你的收听，我们下次再见。

上一篇　06 | HttpDNS和TLS：你的消息聊天真的安全吗?

下一篇　08 | 智能心跳机制：解决网络的不确定性

精选留言 (10)

写留言

钢
2019-09-11

原子化嵌入式脚本有例子介绍吗

展开

作者回复: 示例挺多的哈，给一个redis官网的例子：

```
local current
current = redis.call("incr",KEYS[1])
if tonumber(current) == 1 then
    redis.call("expire",KEYS[1],1)
end
```

3

leslie
2019-09-11

Redis不是特别熟悉：其实老师今天的问题和另外一个问题有点类似；既然问题是"执行过程中掉电是否会出现问题"这个极端场景：那么我就用极端场景解释，老师看看是否有理或者可能啊。

我的答案是会：尤其是极端场景下会，虽然概念很小；其实老师今天的问题是李玥老师的消息队列课程中前几天的期中考题，"数据写入PageCache后未做刷盘，那种情况下数...

展开

作者回复: 没关系哈，互相探讨的过程希望大家不要拘谨。正如你所说，redis在执行lua脚本过程中如果发生掉电，是可能会导致两个未读不一致的，因为lua脚本在redis中的执行只能保证多条命令会原子执行，整体执行完成才会同步给从库并写入aof，所以如果执行过程中掉电，会直接导致被中断的后面部分的脚本得不到执行。当然， 实际情况中这种概率非常小。作为兜底的方案，可以在未读变更时如果会话比较少，可以获取一次全量的会话未读来覆盖总未读，从而有机会能得到最终一致。

2

王棕生
2019-09-12

对于老师本节讲述的未读数不一致的问题，我想是否可以通过下面的方法来解决：

1 用户的未读数是在用户离线的时候，其他用户给他发消息的时候产生的，所以，只需要维护用户会话未读数即可；
2 当用户登录的时候，发送一个消息到MQ，由MQ触发维护用户总的未读数的操作，即将用户所有的会话未读数相加后的数值放入总未读数字段中。...

展开

作者回复: 思路是好的哈，不过很多时候不仅仅是用户登录的时候才需要总未读，比如每来一条消息需要进行系统推送时，由于苹果的APNs不支持角标未读的累加，只能每次获取总未读带下去。另外，客户端维护总未读这个也需要考虑比如离线消息太多，需要推拉结合获取时，到达客户端的消息数不一定是真正的未读消息数。

1

A:春哥大魔王
2019-09-11

老师请教个问题，针对于高频修改场景，频繁的一个字段状态变更，为了解决一个操作一次请求的问题可以采用客户端缓存一段时间聊天记录，批量发送，或者服务端分区批量发送以减少网络io或者db压力，但是两者都存在因为crash造成消息丢失的问题，请问这种情况有什么比较好的解决吗🙏

展开

作者回复: buffer缓冲和强一致性本身就是两个比较对立的概念，所以要做到既能缓冲请求频率又保证强一致性是比较困难的。如果可以的话，尽量让不容易宕机的一方来进行buffer缓冲，比如：如果是客户端和服务端都能缓冲，那还是让服务端来缓冲可能比较可靠一些。

1

墙角儿的花
2019-09-11

老师 对于im服务器集群，客户端的socket均布在各个服务器，目标socket不在同一个服务器上时，服务器间需要转发消息，这个场景需要低延迟无需持久化，服务器间用redis的发布订阅，因其走内存较快，即使断电还可以走库。im服务器和入库服务间用其他mq解耦，因为这个环节需要持久化，所以选rocketmq或kafka，但kafka会延迟批量发布消息 所以选rocketmq，这两个环节的mq选型可行吗。

展开

作者回复: 可以的，长连接入层和后面的业务层之间可以通过redis的pub sub来降低消息延迟，消息发送的api层和具体的持久化层出于成本考虑可以通过其他非内存型来实现，kafka由于是顺序的读写，写入和读取的性能有系统的PageCache来加速，所以性能上不会差。不知道你这里说的延迟批量发布消息具体是什么原因呢?

1

romantic_PK
2019-09-11

老师你好，我想请教一个问题，如何实现微信打开聊天窗口后，点击未读数定位到第一条未读消息的位置，请指点迷津，谢谢。

展开

作者回复: 这个应该是客户端逻辑哈，点击未读数实际上是把最新的一条消息id带进去了，端上在已有的消息里查询这条消息就可以了。这也是为什么最近联系人需要带上最新的一条消息了。

小祺
2019-09-11

首先，如果修改“会话未读数“和“总未读数”是放在一个数据库事务中肯定是可以保证原子性的，但是数据库没法满足高并发的需求，所以通常可以使用Redis来解决高并发问题，为了保证Redis多条命令的原子性老师给出了3个方案。
分布式锁：我认为分布式锁只能解决并发问题，因为第一条命令成功第二条命令失败的情况依然可能发生，怎么办呢？只能不断的重试第二条命令吗？...

展开

作者回复: 分布式锁需要能拿到锁就能保证同一时间只有拿到锁的进程才行执行操作，因为会话未读和总未读变更是在一个进程里，所以理论上是可以保证原子性的。但如果像你所说，第二条加未读的命令一直执行失败还是会出现不一致的情况，这种情况一个是重试，另外就是回滚第一个操作。

lua脚本这个可以考虑在脚本中增加一些修复机制，比如会话数比较少的情况下聚合一次未读来覆盖总未读。

卫江
2019-09-11

首先，我认为redis的脚本化提供了类似于事务的功能，只是功能上面更强大，也更便捷。但是同redis的事务一样，对于事务的ACID支持并不完善。老师提的问题，执行过程中掉电，首先这个的执行事务肯定是失败的，即使开启持久化也没有办法修复，同时客户端也会收到断线回复，所以，就可以当做失败处理，而针对于失败，业务可以通过重试来进行

容错，但是感觉这里需要特别的设计，比如针对于某个玩家的消息未读等信息的更新和...

展开

作者回复: 是的，redis对于脚本执行并没有做到真正的事务性，lua脚本在redis中的执行只能保证多条命令会原子执行，整体执行完成才会同步给从库并写入aof，所以如果执行过程中掉电，会直接导致被中断的后面部分的脚本得不到执行。lua脚本中可以增加一些修复机制，比如会话比较少的话就聚合一次会话来覆盖总未读。

bbpatience

2019-09-11

掉电会出现问题，在redis做主从拷贝时，锁信息有可能正好没有同步到从，这些从在切为主时，没有锁信息。可以用zk来解决分布式锁问题，它能保证掉电后再选举成功的节点，一定包含锁信息

作者回复: 好像和提的问题不大对的上哈，redis对于脚本的执行问题在于并不能保证执行过程掉电后从库或者aof能够感知到来用于恢复。

云师兄

2019-09-11

老师解决原子性方案两个是从redis角度提出的，是否实践中就是使用redis存储消息未读数？是考虑未读数的高频读写吗

展开

作者回复: 是的，未读数读取是一个相对高频的场景，特别是总未读。推送时更新角标和app上的消息tab未读轮询都会用到，所以如果是基于服务端来实现的话，推荐使用redis。