互联网一致性架构设计

<https://mp.weixin.qq.com/s/Smni4c4MxY2IZVD_h-428Q>

1. session一致性
2. 背景知识

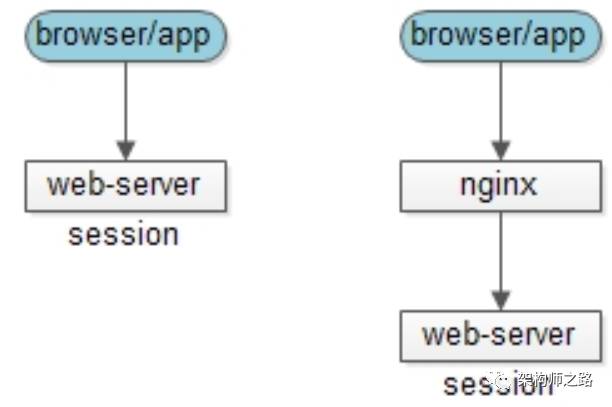
* 什么是session

服务器为每个用户创建一个会话，存储用户的相关信息，以便多次请求能够定位到同一个上下文。

Web开发中，web-server可以自动为同一个浏览器的访问用户自动创建session，提供数据存储功能。最常见的，会把用户的登录信息、用户信息存储在session中，以保持登录状态。

* 什么是session一致性

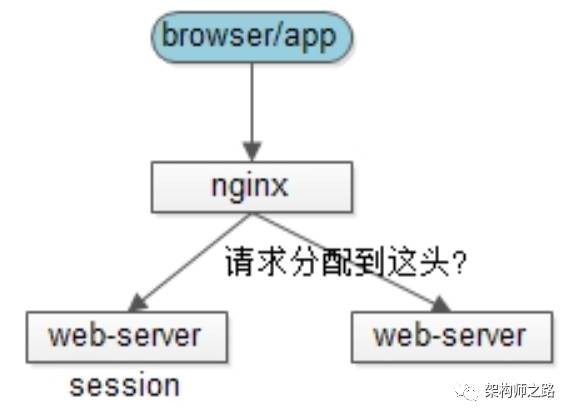
只要用户不重启浏览器，每次http短连接请求，理论上服务端都能定位到session，保持会话。



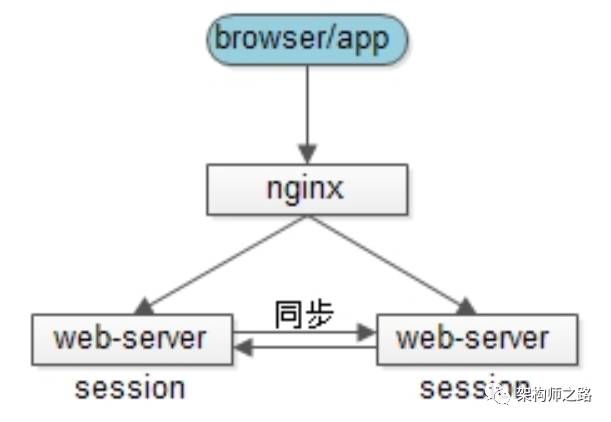
当只有一台web-server提供服务时，每次http短连接请求，都能够正确路由到存储session的对应web-server(因为只有一台0.0)此时的web-server是无法保证高可用的。

当采用“冗余+故障转移”的多台web-server来保证高可用时，每次http短连接请求就不一定能路由到正确的session了。如下图，假设用户包含登录信息的session都记录在第一台web-server上，反向代理如果将请求路由到另一台web-server上，可能就找不到相关信息，而导致用户需要重新登录。

接下来提出几种session一致性的方案



1. 方案一：session同步法



* 思路

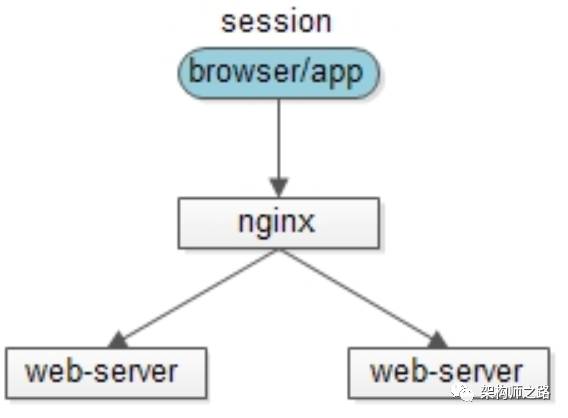
多个web-server之间相互同步session，这样每个web-server之间都包含全部的session

* 优点

web-server支持的功能，应用程序不需要修改代码

* 不足
* session的同步需要数据传输，占内网带宽，有时延
* 所有web-server都包含所有session数据，数据量受内存限制，无法水平扩展
* 有更多web-server时要歇菜

1. 方案二：客户端存储法



* 思路

服务端存储所有用户的session，内存占用较大，可以将session存储到浏览器cookie中，每个端只要存储一个用户的数据了(“端存储”的方案不常用，但也是一种思路。)

* 优点

服务端不需要存储

* 不足
* 每次http请求都携带session，占外网带宽
* 数据存储在端上，并在网络传输，存在泄漏、篡改、窃取等安全隐患
* session存储的数据大小受cookie限制

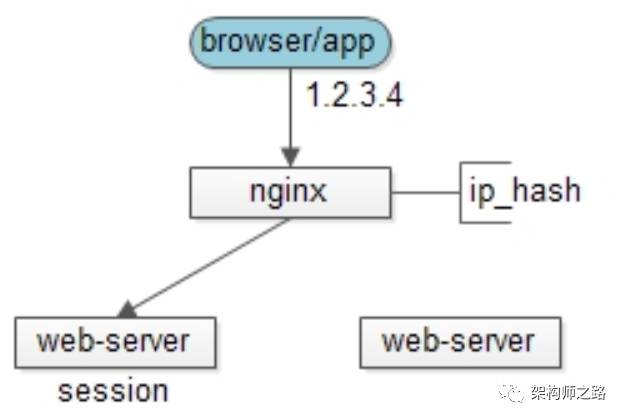
1. 方案三：反向代理Hash一致性

* 思路

web-server为了保证高可用，有多台冗余，反向代理层能不能做一些事情，让同一个用户的请求保证落在一台web-server上。

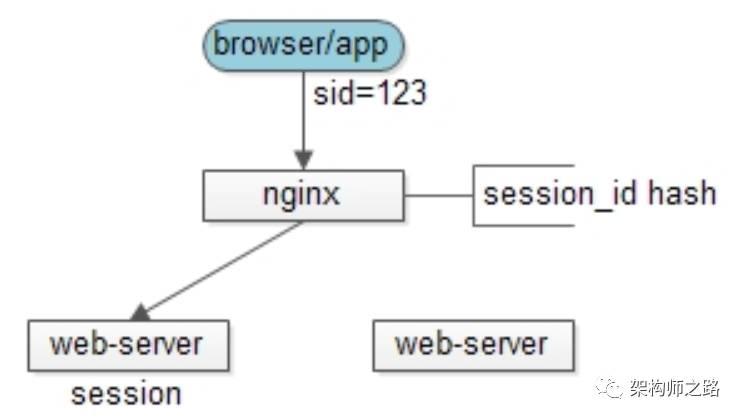
* 四层代理hash

反向代理层使用用户ip来做hash，以保证同一个ip的请求落在同一个web-server上



* 七层代理hash

反向代理使用http协议中的某些业务属性来做hash，例如sid，city\_id，user\_id等，能够更加灵活的实施hash策略，以保证同一个浏览器用户的请求落在同一个web-server上。



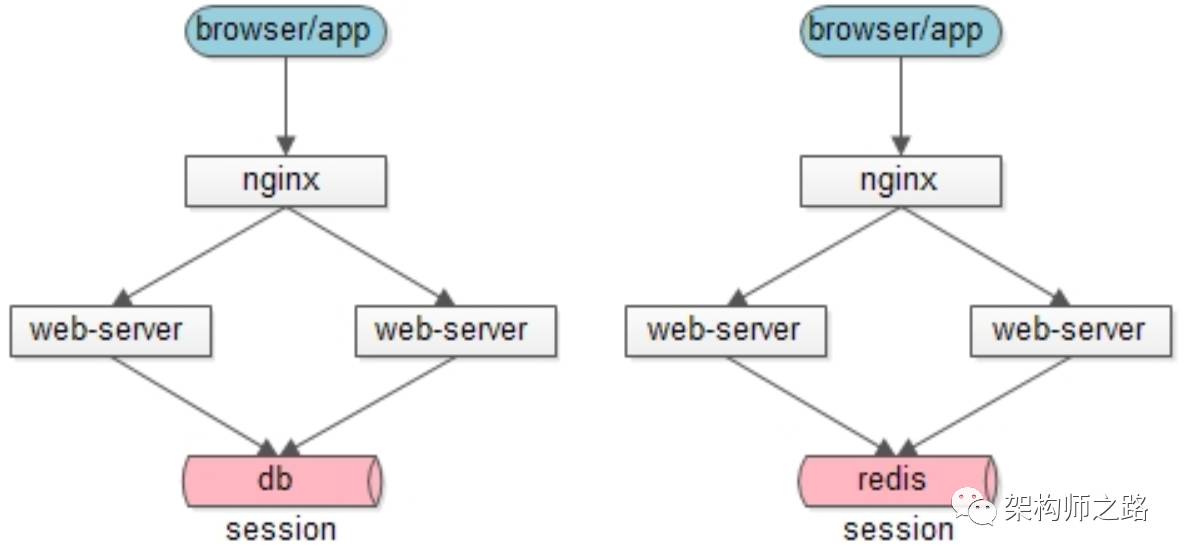
* 优点
* 只需要改nginx配置，不需要修改应用代码
* 负载均衡，只要hash属性是均匀的，多台web-server的负载是均衡的
* 可以支持web-server水平扩展（session同步法是不行的，受内存限制）
* 不足
* 如果web-server重启，一部分session会丢失，产生业务影响，例如部分用户重新登录
* 如果web-server水平扩展，rehash后session重新分布，也会有一部分用户路由不到正确的session

session一般是有有效期的，所有不足中的两点，可以认为等同于部分session失效，一般问题不大。

* 两种思路的比较

推荐四层，反向代理就负责转发，尽量不要引入应用层业务属性，除非不得不这么做

1. 方案四：后端统一储存



* 思路

将session存储在web-server后端的存储层，数据库或者缓存

* 优点
* 没有安全隐患
* 可以水平扩展，数据库/缓存水平切分即可
* web-server重启或者扩容都不会有session丢失
* 不足

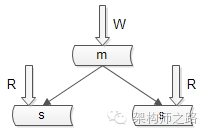
增加了一次网络调用，并且需要修改应用代码

* 探讨

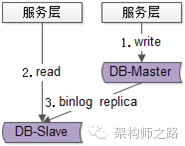
对于db存储和cache，倾向于后者。session读取的频率会很高，数据库压力会比较大。如果有session高可用需求，cache可以做高可用，但大部分情况下session可以丢失，一般也不需要考虑高可用。

1. 数据库主从一致性
2. 问题起源

大部分互联网的业务都是“读多写少”的场景，数据库层面，读性能往往成为瓶颈。如下图：业界通常采用“一主多从，读写分离，冗余多个读库”的数据库架构来提升数据库的读性能。



这种架构的一个潜在缺点是，业务方有可能读取到并不是最新的旧数据：

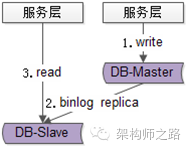


* 系统先对DB-master进行了一个写操作，写主库
* 很短的时间内并发进行了一个读操作，读从库，此时主从同步没有完成，故读取到了一个旧数据
* 主从同步完成

下面提出几个能够解决或者缓解这类“由于主从延时导致读取到旧数据”的问题的方案。

1. 方案一：半同步复制

不一致是因为写完成后，主从同步有一个时间差，假设是500ms，这个时间差有读请求落到从库上产生的。有没有办法做到，等主从同步完成之后，主库上的写请求再返回呢？答案是肯定的，就是大家常说的“半同步复制”semi-sync：



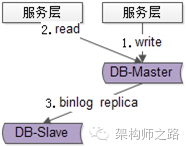
* 步骤
  + 系统先对DB-master进行了一个写操作，写主库
  + 等主从同步完成，写主库的请求才返回
  + 读从库，读到最新的数据（如果读请求先完成，写请求后完成，读取到的是“当时”最新的数据）
* 方案优点

利用数据库原生功能，比较简单

* 方案缺点
* 主库的写请求时延会增长，吞吐量会降低

1. 方案二：强制读主库

如果不使用“增加从库”的方式来增加提升系统的读性能，完全可以读写都落到主库，这样就不会出现不一致了：



* 优点

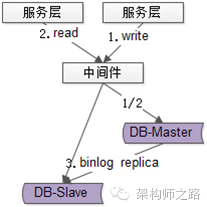
“一致性”上不需要进行系统改造

* 缺点

只能通过cache来提升系统的读性能，这里要进行系统改造

1. 数据库中间件

如果有了数据库中间件，所有的数据库请求都走中间件，可以这么解决：

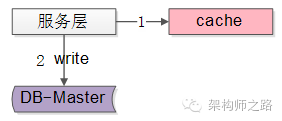


* 步骤
  + 所有的读写都走数据库中间件，通常情况下，写请求路由到主库，读请求路由到从库
  + 记录所有路由到写库的key，在经验主从同步时间窗口内（假设是500ms），如果有读请求访问中间件，此时有可能从库还是旧数据，就把这个key上的读请求路由到主库
  + 经验主从同步时间过完后，对应key的读请求继续路由到从库
* 优点：能保证绝对一致
* 缺点：数据库中间件的成本比较高

1. 缓存记录写key法

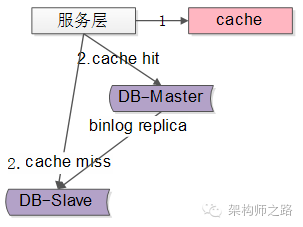
既然数据库中间件的成本比较高，一个更低成本的方案来记录某一个库的某一个key上发生了写请求就是使用缓存

当写请求发生的时候：



* 将某个库上的某个key要发生写操作，记录在cache里，并设置“经验主从同步时间”的cache超时时间，例如500ms
* 修改数据库

而读请求发生的时候：



* 先到cache里查看，对应库的对应key有没有相关数据
* 如果cache hit，有相关数据，说明这个key上刚发生过写操作，此时需要将请求路由到主库读最新的数据
* 如果cache miss，说明这个key上近期没有发生过写操作，此时将请求路由到从库，继续读写分离

优点：相对数据库中间件，成本较低

缺点：为了保证“一致性”，引入了一个cache组件，并且读写数据库时都多了一步cache操作

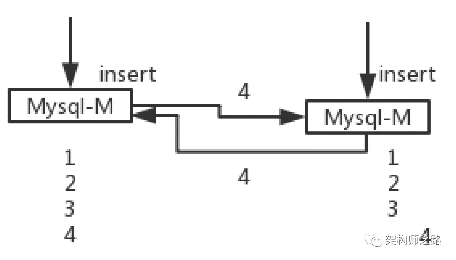
1. 数据库双主一致性
2. 双主保证高可用

MySQL数据库集群常使用一主多从，主从同步，读写分离的方式来扩充数据库的读性能，保证读库的高可用，但此时写库仍然是单点。

在一个MySQL数据库集群中可以设置两个主库，并设置双向同步，以冗余写库的方式来保证写库的高可用。

1. 并发引发不一致

数据冗余会引发数据的一致性问题，因为数据的同步有一个时间差，并发的写入可能导致数据同步失败，引起数据丢失：



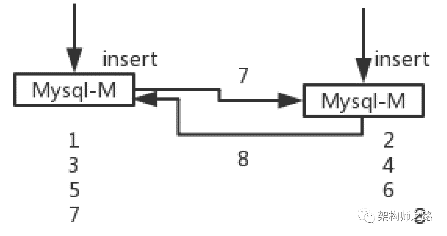
如上图所述，假设主库使用了auto increment来作为自增主键：

* 两个MySQL-master设置双向同步可以用来保证主库的高可用
* 数据库中现存的记录主键是1，2，3
* 主库1插入了一条记录，主键为4，并向主库2同步数据
* 数据同步成功之前，主库2也插入了一条记录，由于数据还没有同步成功，插入记录生成的主键也为4，并向主库1也同步数据
* 主库1和主库2都插入了主键为4的记录，双主同步失败，数据不一致

1. 相同步长免冲突

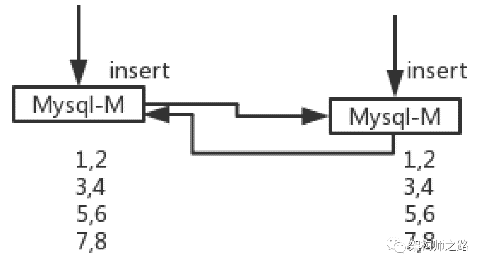
通过一下方式来保证两个主库生成的主键一定不冲突：

* 设置不同的初始值
* 设置相同的增长步长



如上图所示：

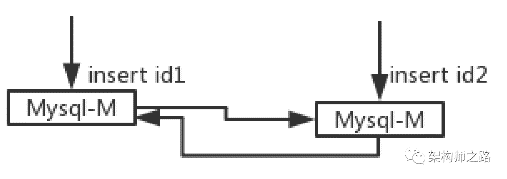
* 两个MySQL-master设置双向同步可以用来保证主库的高可用
* 库1的自增初始值是1，库2的自增初始值是2，增长步长都为2
* 库1中插入数据主键为1/3/5/7，库2中插入数据主键为2/4/6/8，不冲突
* 数据双向同步后，两个主库会包含全部数据



如上图所示，两个主库最终都将包含1/2/3/4/5/6/7/8所有数据，即使有一个主库挂了，另一个主库也能够保证写库的高可用。

1. 上游生成ID避免冲突

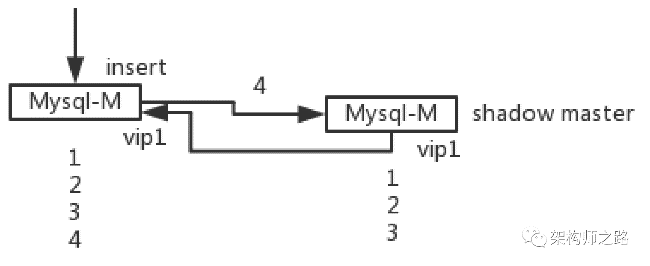
由业务上游，使用统一的ID生成器，来保证ID的生成不冲突：



如上图所示，调用方插入数据时，带入全局唯一ID，而不依赖于数据库的auto increment，也能解决这个问题。(《分布式ID生成器》)

1. 消除双写不治本

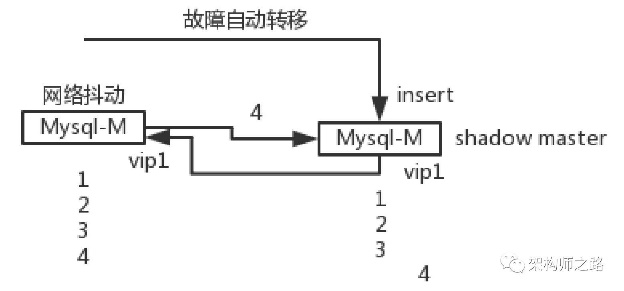
使用auto increment两个主库并发写可能导致数据不一致，只使用一个主库提供服务，另一个主库作为shadow-master，只用来保证高可用，能否避免一致性问题？



如上图所示：

* 两个MySQL-master设置双向同步可以用来保证主库的高可用
* 只有主库1对外提供写入服务
* 两个主库设置相同的虚IP，在主库1挂掉或者网络异常的时候，虚IP自动漂移，shadow master顶上，保证主库的高可用

这个切换由于虚IP没有变化，所以切换过程对调用方是透明的，但在极限的情况下，也可能引发数据的不一致：



如上图所示

* 两个MySQL-master设置双向同步可以用来保证主库的高可用，并设置了相同的虚IP
* 网络抖动前，主库1对上游提供写入服务，插入了一条记录，主键为4，并向shadow master主库2同步数据
* 突然主库1网络异常，keepalived检测出异常后，实施虚IP漂移，主库2开始提供服务
* 在主键4的数据同步成功之前，主库2插入了一条记录，也生成了主键为4的记录，结果导致数据不一致

1. 内网DNS探测

虚IP漂移，双主同步延时导致的数据不一致，本质上，需要在双主同步完数据之后，再实施虚IP偏移，使用内网DNS探测，可以实现shadow master延时高可用：

* 使用内网域名连接数据库，例如：db.58daojia.org
* 主库1和主库2设置双主同步，不使用相同虚IP，而是分别使用ip1和ip2
* 一开始db.58daojia.org指向ip1
* 用一个小脚本轮询探测ip1主库的连通性
* 当ip1主库发生异常时，小脚本delay一个x秒的延时，等待主库2同步完数据之后，再将db.58daojia.org解析到ip2
* 程序以内网域名进行重连，即可自动连接到ip2主库，并保证了数据的一致性

1. 总结

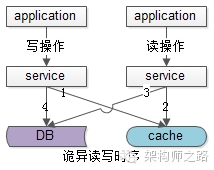
总之，有这些小技巧：

* 双主同步是一种常见的保证写库高可用的方式
* 设置相同步长，不同初始值，可以避免auto increment生成冲突主键
* 不依赖数据库，业务调用方自己生成全局唯一ID是一个好方法
* shadow master保证写库高可用，只有一个写库提供服务，并不能完全保证一致性
* 内网DNS探测，可以实现在主库1出现问题后，延时一个时间，再进行主库切换，以保证数据一致性

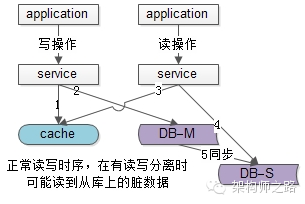
1. 数据库与缓存一致性

——数据库主从延时方案导致与缓存数据不一致

1. 原因
   1. 单库条件下，服务层并发读写，缓存与数据库操作交叉进行，如下图，虽然只有一个DB，但是按照几张图中的诡异读写时序，就会发生脏数据入缓存：



* + 1. 请求A发起一个写操作，第一步淘汰了cache，然后这个请求因为各种原因在服务层卡住了（进行大量的业务逻辑计算，例如计算了1秒钟），如上图步骤1
    2. 请求B发起一个读操作，读cache，cache miss，如上图步骤2
    3. 请求B继续读DB，读出来一个脏数据，然后脏数据入cache，如上图步骤3
    4. 请求A卡了很久后终于写数据库了，写入了最新的数据，如上图步骤4
  1. 主从同步，读写分离的情况下，读从库读到旧数据，在数据库架构做了一主多从，读写分离时，更多的脏数据入缓存是下面这种情况：



* + 1. 请求A发起一个写操作，第一步淘汰了cache，如上图步骤1
    2. 请求A写数据库了，写入了最新的数据，如上图步骤2
    3. 请求B发起一个读操作，读cache，cache miss，如上图步骤3
    4. 请求B继续读DB，读的是从库，此时主从同步还没有完成，读出来一个脏数据，然后脏数据入cache，如上图步4
    5. 最后数据库的主从同步完成了，如上图步骤5

1. 优化思路

综上，不一致的原因主要有以下两个：

* 1. 单库情况下，服务层在进行1s的逻辑计算过程中，可能读到旧数据入缓存
  2. 主从库+读写分离情况下，在1s钟主从同步延时过程中，可能读到旧数据入缓存

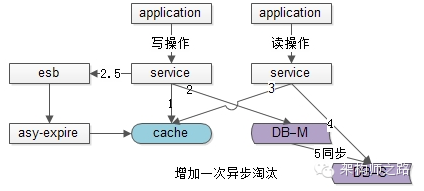
既然旧数据就是在那1s的间隙中入缓存的，可以在写请求完成后，再休眠1s，再次淘汰缓存，就能将这1s内写入的脏数据再次淘汰掉

这样写请求的步骤变为三步：

* 先淘汰缓存
* 再写数据库（这两步和原来一样）
* 休眠1秒，再次淘汰缓存

但是这样也产生了几个问题

* 所有的写请求都阻塞了1秒，大大降低了写请求的吞吐量，增长了处理时间，业务上是接受不了的
  + 解决：其实第二次淘汰缓存是“为了保证缓存一致”而做的操作，而不是“业务要求”，所以其实无需等待，**用一个异步的timer**，或者**利用消息总线**异步的来做这个事情即可：



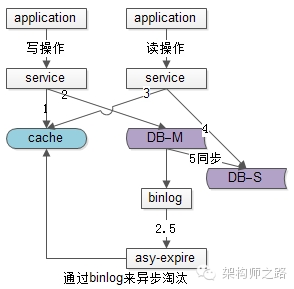
写请求由2步升级为2.5步：

* + - 先淘汰缓存
    - 再写数据库（这两步和原来一样）
    - 不再休眠1s，而是往消息总线esb发送一个消息，发送完成之后马上就能返回

这样的话，写请求的处理时间几乎没有增加，这个方法淘汰了缓存两次，因此被称为“缓存双淘汰”法。这个方法付出的代价是，缓存会增加1次cache miss（代价几乎可以忽略）。

而在下游，有一个异步淘汰缓存的消费者，在接收到消息之后，asy-expire在1s之后淘汰缓存。这样，即使1s内有脏数据入缓存，也有机会再次被淘汰掉。

上述方案有一个缺点，需要业务线的写操作增加一个步骤，通过下面方案可以避免对业务线的代码的入侵呢即**通过分析线下的binlog**来异步淘汰缓存：



业务线的代码就不需要动了，新增一个线下的读binlog的异步淘汰模块，读取到binlog中的数据，异步的淘汰缓存。

* + 总结
    - 异步timer
    - 消息总线
    - binlog

1. 数据冗余一致性
2. 需求源起

互联网很多业务场景的数据量很大，此时数据库架构要进行水平切分，水平切分会有一个patition key，通过patition key的查询能够直接定位到库，但是非patition key上的查询可能就需要扫描多个库了。

例如订单表，业务上对用户和商家都有订单查询需求：

Order(oid, info\_detail)

T(buyer\_id, seller\_id, oid)

如果用buyer\_id来分库，seller\_id的查询就需要扫描多库。

如果用seller\_id来分库，buyer\_id的查询就需要扫描多库。

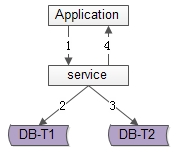
这类需求，为了做到高吞吐量低延时的查询，往往使用“数据冗余”的方式来实现，就是文章标题里说的“冗余表”：

T1(buyer\_id, seller\_id, oid)

T2(seller\_id, buyer\_id, oid)

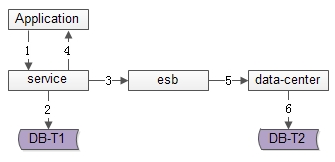
同一个数据，冗余两份，一份以buyer\_id来分库，满足买家的查询需求；一份以seller\_id来分库，满足卖家的查询需求

1. 冗余表实现方案
   1. 方案一：服务同步写



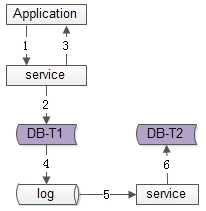
顾名思义，由服务层同步写冗余数据，如上：

* + 业务方调用服务，新增数据
  + 服务先插入T1数据
  + 服务再插入T2数据
  + 服务返回业务方新增数据成功
* 优点
  + 不复杂，服务层由单次写，变两次写
  + 数据一致性相对较高（因为双写成功才返回）
* 缺点
  + 请求的处理时间增加（要插入次，时间加倍）
  + 数据仍可能不一致，例如第二步写入T1完成后服务重启，则数据不会写入T2
  1. 方案二：服务异步写(业务对时间敏感)



数据的双写并不再由服务来完成，服务层异步发出一个消息，通过消息总线发送给一个专门的数据复制服务来写入冗余数据，如上图1-6流程：

* + 业务方调用服务，新增数据
  + 服务先插入T1数据
  + 服务向消息总线发送一个异步消息（发出即可，不用等返回，通常很快就能完成）
  + 服务返回业务方新增数据成功
  + 消息总线将消息投递给数据同步中心
  + 数据同步中心插入T2数据
* 优点
  + 请求处理时间短
* 缺点
  + 系统的复杂性增加了，多引入了一个组件（消息总线）和一个服务（专用的数据复制服务）
  + 因为返回业务线数据插入成功时，数据还不一定插入到T2中，因此数据有一个不一致时间窗口（这个窗口很短，最终是一致的）
  + 在消息总线丢失消息时，冗余表数据会不一致
  1. 方案三：线下异步写(解除“数据冗余”对系统的耦合)



数据的双写不再由服务层来完成，而是由线下的一个服务或者任务来完成，如上图：

* + 业务方调用服务，新增数据
  + 服务先插入T1数据
  + 服务返回业务方新增数据成功
  + 数据会被写入到数据库的log中
  + 线下服务或者任务读取数据库的log
  + 线下服务或者任务插入T2数据
* 优点
  + 数据双写与业务完全解耦
  + 请求处理时间短（只插入1次）
* 缺点
  + 返回业务线数据插入成功时，数据还不一定插入到T2中，因此数据有一个不一致时间窗口（这个窗口很短，最终是一致的）
  + 数据的一致性依赖于线下服务或者任务的可靠性

1. 先写正表还是反表

对于一个不能保证事务性的操作，一但涉及“哪个任务先做，哪个任务后做”的问题，解决这个问题的方向是：【如果出现不一致】，谁先做对业务的影响较小，就谁先执行。

以上文的订单生成业务为例，buyer和seller冗余表都需要插入数据：

T1(buyer\_id, seller\_id, oid)

T2(seller\_id, buyer\_id, oid)

用户下单时，如果“先插入buyer表T1，再插入seller冗余表T2”，当第一步成功、第二步失败时，出现的业务影响是“买家能看到自己的订单，卖家看不到推送的订单”

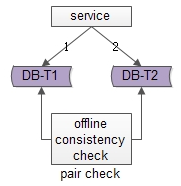
相反，如果“先插入seller表T2，再插入buyer冗余表T1”，当第一步成功、第二步失败时，出现的业务影响是“卖家能看到推送的订单，卖家看不到自己的订单”

由于这个生成订单的动作是买家发起的，买家如果看不到订单，会觉得非常奇怪，并且无法支付以推动订单状态的流转，此时即使卖家看到有人下单也是没有意义的。

因此，在此例中，应该先插入buyer表T1，再插入seller表T2。

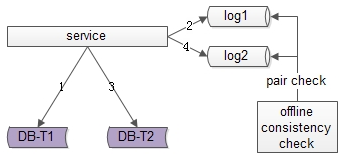
however，记住结论：【如果出现不一致】，谁先做对业务的影响较小，就谁先执行。

1. 如何保证数据一致性
   1. 方法一：线下扫描冗余数据表的全部数据



下启动一个离线的扫描工具，不停的比对正表T1和反表T2，如果发现数据不一致，就进行补偿修复。

* + 优点
    - 比较简单，开发代价小
    - 线上服务无需修改，修复工具与线上服务解耦
  + 缺点
    - 扫描效率低，会扫描大量的“已经能够保证一致”的数据
    - 由于扫描的数据量大，扫描一轮的时间比较长，即数据如果不一致，不一致的时间窗口比较长
  1. 方法二：线下扫描增量数据

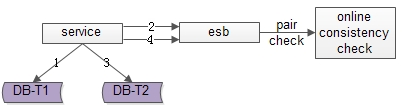


每次只扫描增量的日志数据，就能够极大提高效率，缩短数据不一致的时间窗口，如上图1-4流程所示

* + - 写入正表T1
    - 第一步成功后，写入日志log1
    - 写入反表T2
    - 第二步成功后，写入日志log2

当然，我们还是需要一个离线的扫描工具，不停的比对日志log1和日志log2，如果发现数据不一致，就进行补偿修复。

* + 优点
    - 虽比方法一复杂，但仍然是比较简单的
    - 数据扫描效率高，只扫描增量数据
  + 缺点
    - 线上服务略有修改（代价不高，多写了2条日志）
    - 虽然比方法一更实时，但时效性还是不高，不一致窗口取决于扫描的周期
  1. 方法三：实时线上消息对检测



这次不是写日志了，而是向消息总线发送消息，

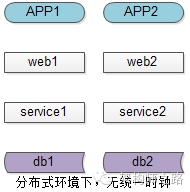
* + - 写入正表T1
    - 第一步成功后，发送消息msg1
    - 写入反表T2
    - 第二步成功后，发送消息msg2

这次不是需要一个周期扫描的离线工具了，而是一个实时订阅消息的服务不停的收消息。

假设正常情况下，msg1和msg2的接收时间应该在3s以内，如果检测服务在收到msg1后没有收到msg2，就尝试检测数据的一致性，不一致时进行补偿修复

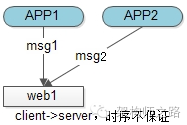
* + 优点
    - 效率高
    - 实时性高
  + 缺点
    - 方案比较复杂，上线引入了消息总线这个组件
    - 线下多了一个订阅总线的检测服务

1. 消息时序一致性
   1. 时序难以保证，消息一致性难的原因分析
      1. 时钟不一致



分布式环境下有多个客户端，web集群，service集群，db集群，他们都分布在不同的机器上，机器之间都是使用的本地时钟，而没有一个所谓的“全局时钟”，所以不能用“本地时间”来完全决定消息的时序。

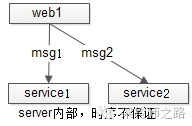
* + 1. 多客户端(发送方)



多服务器不能用“本地时间”进行比较，假设只有一个接收方，能否用接收方本地时间表示时序呢？遗憾的是，由于多个客户端的存在，即使是一台服务器的本地时间，也无法表示“绝对时序”。

如上图，绝对时序上，APP1先发出msg1，APP2后发出msg2，都发往服务器web1，网络传输是不能保证msg1一定先于msg2到达的，所以即使以一台服务器web1的时间为准，也不能精准描述msg1与msg2的绝对时序。

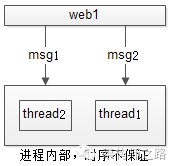
* + 1. 服务集群(多接收方)



多发送方不能保证时序，假设只有一个发送方，能否用发送方的本地时间表示时序呢？遗憾的是，由于多个接收方的存在，无法用发送方的本地时间，表示“绝对时序”。

如上图，绝对时序上，web1先发出msg1，后发出msg2，由于网络传输及多接收方的存在，无法保证msg1先被接收到先被处理，故也无法保证msg1与msg2的处理时序。

* + 1. 网络传输与多线程



多发送方与多接收方都难以保证绝对时序，假设只有单一的发送方与单一的接收方，能否保证消息的绝对时序呢？结论是悲观的，由于网络传输与多线程的存在，仍然不行。

如上图，web1先发出msg1，后发出msg2，即使msg1先到达（网络传输其实还不能保证msg1先到达），由于多线程的存在，也不能保证msg1先被处理完。

* + 1. 保证绝对时序

通过上面的分析，假设只有一个发送方，一个接收方，上下游连接只有一条连接池，通过阻塞的方式通讯，是可以保证先发出的消息msg1先处理么，但是吞吐量会非常低，而且单发送方单接收方单连接池的假设不太成立，高并发高可用的架构不会允许这样的设计出现。

* 1. 优化实践
     1. 以客户端或者服务器端的时序为准

多客户端、多服务端导致“时序”的标准难以界定，需要一个标尺来衡量时序的先后顺序，可以根据业务场景，以客户端或者服务端的时间为准，例如：

* + 邮件展示顺序，其实是以客户端发送时间为准的，潜台词是，发送方只要将邮件协议里的时间调整为1970年或者2970年，就可以在接收方收到邮件后一直“置顶”或者“置底”
  + 秒杀活动时间判断，肯定得以服务器的时间为准，不可能让客户端修改本地时间，就能够提前秒杀
    1. 服务器端生成单调递增的id

这个是毋庸置疑的，不展开讨论，例如利用单点写db的seq/auto\_inc\_id肯定能生成单调递增的id，只是说性能及扩展性会成为潜在瓶颈。对于严格时序的业务场景，可以利用服务器的单调递增id来保证时序。

* + 1. 大部分业务能接受误差不大的趋势递增的id

消息发送、帖子发布时间、甚至秒杀时间都没有这么精准时序的要求：

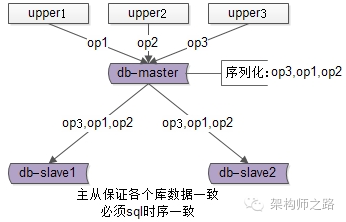
* + 同1s内发布的聊天消息时序乱了
  + 同1s内发布的帖子排序不对
  + 用1s内发起的秒杀，由于服务器多台之间时间有误差，落到A服务器的秒杀成功了，落到B服务器的秒杀还没开始，业务上也是可以接受的（用户感知不到）

所以，大部分业务，长时间趋势递增的时序就能够满足业务需求，非常短时间的时序误差一定程度上能够接受。

* + 1. 利用单点序列化，可以保证多机时序相同

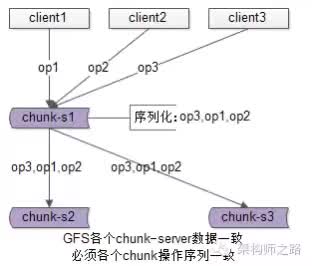
数据为了保证高可用，需要做到进行数据冗余，同一份数据存储在多个地方，怎么保证这些数据的修改消息是一致的呢？利用的就是“单点序列化”：

* + 先在一台机器上序列化操作
  + 再将操作序列分发到所有的机器，以保证多机的操作序列是一致的，最终数据是一致的
    - 1. 典型场景一：数据库主从一致



数据库的主从架构，上游分别发起了op1,op2,op3三个操作，主库master来序列化所有的SQL写操作op3,op1,op2，然后把相同的序列发送给从库slave执行，以保证所有数据库数据的一致性，就是利用“单点序列化”这个思路。

* + - 1. 典型场景二：DFS中文件的一致性



GFS(Google File System)为了保证文件的可用性，一份文件要存储多份，在多个上游对同一个文件进行写操作时，也是由一个主chunk-server先序列化写操作，再将序列化后的操作发送给其他chunk-server，来保证冗余文件的数据一致性的。

* + 1. 单对单聊天，保证发送顺序与接收顺序一致

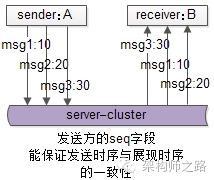
单人聊天的需求，发送方A依次发出了msg1，msg2，msg3三个消息给接收方B，这三条消息能否保证显示时序的一致性（发送与显示的顺序一致）

* + 如果利用服务器单点序列化时序，可能出现服务端收到消息的时序为msg3，msg1，msg2，与发出序列不一致
  + 业务上不需要全局消息一致，只需要对于同一个发送方A，ta发给B的消息时序一致就行，常见优化方案，在A往B发出的消息中，加上发送方A本地的一个绝对时序，来表示接收方B的展现时序

msg1{seq:10, receiver:B,msg:content1 }

msg2{seq:20, receiver:B,msg:content2 }

msg3{seq:30, receiver:B,msg:content3 }

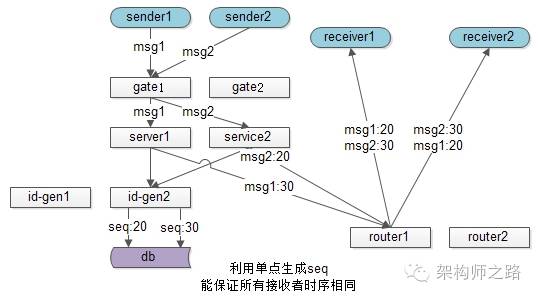


潜在问题：如果接收方B先收到msg3，msg3会先展现，后收到msg1和msg2后，会展现在msg3的前面。

无论如何，是按照接收方收到时序展现，还是按照服务端收到的时序展现，还是按照发送方发送时序展现，是pm需要思考的点，技术上都能够实现（接收方按照发送时序展现是更合理的）。

总之，需要一杆标尺来衡量这个时序。

* + 1. 群聊消息，保证各接收方收到顺序一致
  + 不能再利用发送方的seq来保证时序，因为发送方不单点，时间也不一致
  + 可以利用服务器的单点做序列化

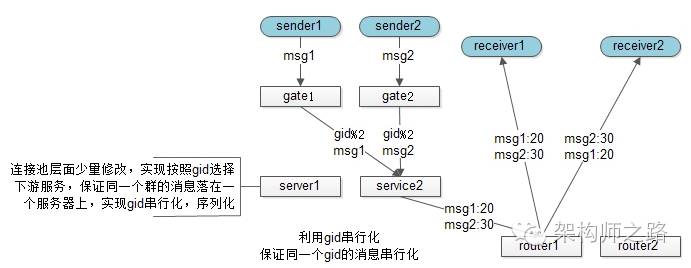


群聊的发送流程为：

* + sender1发出msg1，sender2发出msg2
  + msg1和msg2经过接入集群，服务集群
  + service层到底层拿一个唯一seq，来确定接收方展示时序
  + service拿到msg2的seq是20，msg1的seq是30
  + 通过投递服务将消息给多个群友，群友即使接收到msg1和msg2的时间不同，但可以统一按照seq来展现

这个方法能实现，所有群友的消息展示时序相同。缺点是，这个生成全局递增序列号的服务很容易成为系统瓶颈。

可以这样优化：群消息其实也不用保证全局消息序列有序，而只要保证一个群内的消息有序即可，这样的话，“id串行化”就成了一个很好的思路。



这个方案中，service层不再需要去一个统一的后端拿全局seq，而是在service连接池层面做细小的改造，保证一个群的消息落在同一个service上，这个service就可以用本地seq来序列化同一个群的所有消息，保证所有群友看到消息的时序是相同的。

1. 分布式事务一致性
   1. https://mp.weixin.qq.com/s/FvB-hOBT13SMfZko5iagAg
2. 库存扣减一致性
   1. https://mp.weixin.qq.com/s/waGRvyhab2z8b-BIw9bJeA