
电商项目高并发缓存实践

大部分面向公众的互联网系统，其并发请求数量与在线用户数量都是正相关的，而 MySQL 能够承担的并发读写量是有一定上限的，当系统的访问量超过一定程度时，纯 MySQL 就很难应付了。

绝大多数互联网系统都是采用 MySQL+Redis 这对经典组合来解决高并发问题的。Redis 作为 MySQL 的前置缓存，可以应对绝大部分查询请求，从而在很大程度上缓解 MySQL 并发请求的压力，但是不能一说到缓存脑海中就只有 Redis，这无论在工作还是面试中都不合适，所以我们先全面了解缓存。

缓存为什么不仅仅是 Redis

缓存的意义

缓存大体可以分为三类：

客户端缓存；服务端缓存；网络中的缓存。

根据规模和部署方式缓存也可以分为：

单体缓存；缓存集群；分布式缓存。

可见，在软件系统中缓存几乎无处不在，所以说缓存为王不是没有原因的。

从浏览器到网络，再到应用服务器，甚至到数据库，通过在各个层面应用缓存技术，整个系统的性能将大幅提高。

例如，缓存离客户端更近（**缓存一定是离用户越近越好**），从缓存请求内容比从源服务器所用时间更少，呈现速度更快，系统就显得更灵敏。缓存数据的重复使用，大大降低了用户的带宽使用，其实也是一种变相的省钱，同时保证了带宽请求在一个低水平上，更容易维护。

所以，使用缓存技术，可以降低系统的响应时间，减少网络传输时间和应用延迟时间，进而提高了系统的吞吐量，增加了系统的并发用户数。利用缓存还可以最小化系统的工作量，使用了缓存就可以不必反复从数据源中查找，缓存所创建或提供的同一条数据更好地利用了系统的资源，并且能够更好的保护后端的系统和服务。

因此，缓存是系统调优时常用且行之有效的手段，无论是操作系统还是应用系统，缓存策略无处不在。

缓存的分类

客户端缓存

客户端缓存相对于其他端的缓存而言，要简单一些，而且通常是和服务端以及网络侧的应用或缓存配合使用的。对于互联网应用而言，也就是通常所说的

BS 架构应用，可以分为页面缓存和浏览器缓存。对于移动互联网应用而言，是指 APP 自身所使用的缓存。

页面缓存

页面缓存有两层含义：一个是页面自身对某些元素或全部元素进行缓存；另一层意思是服务端将静态页面或动态页面的元素进行缓存，然后给客户端使用。这里的页面缓存指的是页面自身的缓存或者离线应用缓存。

页面缓存是将之前渲染的页面保存为文件，当用户再次访问时可以避开网络连接，从而减少负载，提升性能和用户体验。

HTML5 提供的离线应用缓存机制，使得网页应用可以离线使用，这种机制在浏览器上支持度非常广，可以放心地使用该特性来加速页面的访问，比如我们商城系统的前端页面中就使用了 `localStorage`。

浏览器缓存

浏览器缓存是根据一套与服务器约定的规则进行工作的，工作规则很简单：检查以确保副本是最新的，通常只要一次会话。浏览器会在硬盘上专门开辟一个空间来存储资源副本作为缓存。在用户触发“后退”操作或点击一个之前看过的链接的时候，浏览器缓存会很管用。同样，如果访问系统中的同一张图片，该图片可以从浏览器缓存中调出并几乎立即显现出来。

对浏览器而言，HTTP1.0 提供了一些很基本的缓存特性，例如在服务器侧设置 `Expires` 的 HTTP 头来告诉客户端在重新请求文件之前缓存多久是安全的，可以通过 `if-modified-since` 的条件请求来使用缓存，还有 `Cache-Control` 等等。HTTP 1.1 有了较大的增强，缓存系统被形式化了，引入了实体标签 `e-tag` 等。

通过在 HTML 页面的节点中加入 `meta` 标签，可以告诉浏览器当前页面不被缓存，每次访问都需要去服务器拉取。

APP 上的缓存

尽管混合编程(hybrid programming)成为时尚，但整个移动互联网目前还是原生应用(以下简称 APP)的天下。无论大型或小型 APP，灵活的缓存不仅大大减轻了服务器的压力，

而且因为更快速的用户体验而方便了用户。如何把 APP 缓存对于业务组件透明，以及 APP 缓存数据的及时更新，是 APP 缓存能否成功应用起来的关键。APP 可以将内容缓存在内存、文件或本地数据库（例如 SQLite)中

网络缓存

网络中的缓存位于客户端和服务端之间，代理或响应客户端的网络请求，从而对重复的请求返回缓存中的数据资源。同时，接受服务端的请求，更新缓存中的内容。

Web 代理缓存

Web 代理几乎是伴随着互联网诞生的，常用的 Web 代理分为正向代理、反向代理和透明代理。Web 代理缓存是将 Web 代理作为缓存的一种技术。

为了从源服务器取得内容,用户向代理服务器发送一个请求并指定目标服务器,然后代理服务向源服务器转交请求并将获得的内容返回给客户端。一般地,客户端要进行一些特别的设置才能使用正向代理。

反向代理与正向代理相反,对于客户端而言代理服务器就像是源服务器,并且客户端不需要进行设置。客户端向反向代理发送普通请求,接着反向代理将判断向何处转发请求,并将从源服务器获得的内容返回给客户端。

透明代理的意思是客户端根本不需要知道有代理服务器的存在,由代理服务器改变客户端请求的报文字段,并会传送真实的 IP 地址。加密的透明代理属于匿名代理,不用设置就可以使用代理了。透明代理的例子就是时下很多公司使用的行为管理软件。

Web 正向代理代理缓存是指使用正向代理的缓存技术。**Web** 代理缓存的作用跟浏览器的内置缓存类似,只是介于浏览器和互联网之间。

当通过代理服务器进行网络访问时,浏览器不是直接到 **Web** 服务器去取回网页而是向 **Web** 代理发出请求,由代理服务器来取回浏览器所需要的信息并传送给浏览器。而且,**Web** 代理缓存有很大的存储空间,不断将新获取的数据储存在本地的存储器上,如果浏览器所请求的数据在 **Web** 代理的缓存上已经存在而且是最新的,那么就不重新从 **Web** 服务器取数据,而是直接将缓存的数据传送给用户的浏览器,这样就能显著提高浏览速度和效率。对于企业而言,使用 **Web** 代理既可以节省成本,又能提高性能。

对于 **Web** 代理缓存而言,较流行的是 **Squid**,它支持建立复杂的缓存层级结构,拥有详细的日志、高性能缓存以及用户认证支持。**Squid** 同时支持各种插件。

使用 **web** 反向代理服务器和使用正向代理服务器一样,可以拥有缓存的作用,反向代理缓存可以缓存原始资源服务器的资源,而不是每次都要向原始资源服务器请求数据,特别是一些静态的数据,比如图片和文件,很多 **Web** 服务器就具备反向代理的功能,比如大名鼎鼎的 **Nginx**,我们的商城系统的秒杀部分就使用了 **Nginx** 的缓存机制。

边缘缓存

如果反向代理服务器能够做到和用户来自同一个网络,那么用户访问反向代理服务器,就会得到很高质量的响应速度,所以可以将这样的反向代理缓存称为边缘缓存。边缘缓存在网络上位于靠近用户的一侧,可以处理来自不同用户的请求,主要用于向用户提供静态的内容,以减少应用服务器的介入。边缘缓存的一个有名的开源工具就是 **Varnish**。

边缘缓存中典型的商业化服务就是 **CDN** 了,例如 **AWS** 的 **Cloud Front**,我国的 **ChinaCache** 等,现在一般的公有云服务商都提供了 **CDN** 服务。**CDN** 是 **Content Delivery Network** 的简称,即“内容分发网络”的意思。。

服务端缓存

服务端缓存是整个缓存体系中的重头戏,网站的架构演进中服务端缓存是系统性能的重中之重了。

数据库缓存

数据库属于 IO 密集型的应用，主要负责数据的管理及存储。数据库缓存是一类特殊的缓存，是数据库自身的缓存机制。大多数数据库不需要配置就可以快速运行，但并没有为特定的需求进行优化。在数据库调优的时候，缓存优化是一项很重要的工作。

当使用 InnoDB 存储引擎的时候，`innodb_buffer_pool_size` 参数可能是影响性能的最为关键的一个参数了，用来设置用于缓存 InnoDB 索引及数据块的内存区域大小，简单来说，当操作一个 InnoDB 表的时候，返回的所有数据或者查询过程中用到的任何一个索引块，都会在这个内存区域中去查询一遍。

`innodb_buffer_pool_size` 设置了 InnoDB 存储引擎需求最大的一块内存区域的大小，直接关系到 InnoDB 存储引擎的性能，所以如果有足够的内存，尽可能将该参数设置到足够大，将尽可能多的 InnoDB 的索引及数据都放入到该缓存区域中，直至全部。

比如在商城系统的 MySQL 中 `innodb_buffer_pool_size` 被设置为物理内存的一半：

```
[root@192-168-65-184 conf]# free -m
```

	total	used	free	shared	buff/cache	available
Mem:	7820	1619	4819	396	1381	5539
Swap:	8063	0	8063			

```
[mysqld]
innodb_buffer_pool_size = 4294967296
innodb_buffer_pool_instances = 4
join_buffer_size = 33554432
sort_buffer_size = 2097152
read_rnd_buffer_size = 2097152
```

当然，数据库还有很多值得深入研究的地方，需要专业的技能，这就是很多公司专门设有 DBA 角色的原因。

应用级缓存

在系统开发的时候，适当地使用应用级缓存，往往可以取得事半功倍的效果。应用级缓存在这里指的是用来写带有缓存特性的应用框架，或者可用于缓存功能的专用库。

在 Java 语言中，缓存框架更多，例如 Ehcache、Voldemort（Voldemort 是一款基于 Java 开发的分布式键-值缓存系统，像 JBoss 的缓存一样，Voldemort 同样支持多台服务器之间的缓存同步，以增强系统的可靠性和读取性能。Voldemort 相当于是 Amazon Dynamo 的一个开源实现，LinkedIn 用它解决了网站的高扩展性存储问题）等等。

平台级缓存

在很多时候，我们还需要考虑使用平台级缓存。

不论是 Redis 还是 MongoDB，以及 Memcached 都可以作为平台级缓存的重要技术，当然 Redis/Memcached 用的更多一些，MongoDB 更多的时候是做为持久化的 NoSQL 数据库来说使用的。

缓存的数据一致性

工程实践

只要使用到缓存，无论是本地内存做缓存还是使用 `redis` 做缓存，那么就会存在数据同步的问题。

先读缓存数据，缓存数据有，则立即返回结果；如果没有数据，则从数据库读数据，并且把读到的数据同步到缓存里，提供下次读请求返回数据。

这样能有效减轻数据库压力，但是如果修改删除数据，因为缓存无法感知到数据在数据库的修改。这样就会造成数据库中的数据与缓存中数据不一致的问题，那该如何解决呢？

有好几种解决方案，

1. 先更新缓存，再更新数据库
2. 先更新数据库，再更新缓存
3. 先删除缓存，后更新数据库
4. 先更新数据库，后删除缓存

我们一一来看看：

更新缓存类

1、先更新缓存，再更新 DB

这个方案我们一般不考虑。原因是更新缓存成功，更新数据库出现异常了，导致缓存数据与数据库数据完全不一致，而且很难察觉，因为缓存中的数据一直都存在。

2. 先更新 DB，再更新缓存

这个方案也我们一般不考虑，原因跟第一个一样，数据库更新成功了，缓存更新失败，同样会出现数据不一致问题。

删除缓存类

3、先删除缓存，后更新 DB

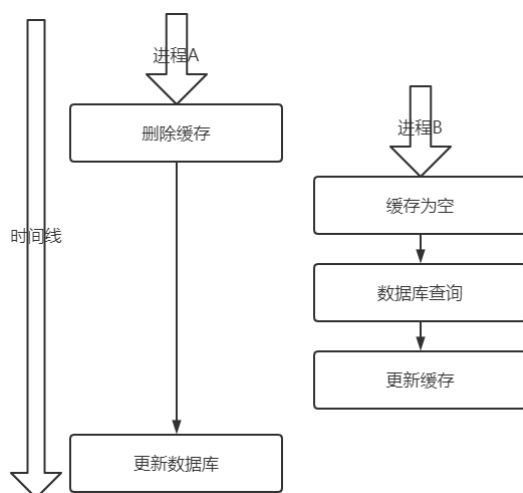
该方案也会出问题，具体出现的原因如下。

此时来了两个请求，请求 A（更新操作） 和请求 B（查询操作）

请求 A 会先删除 `Redis` 中的数据，然后去数据库进行更新操作；

此时请求 B 看到 `Redis` 中的数据是空的，会去数据库中查询该值，补录到 `Redis` 中；

但是此时请求 A 并没有更新成功，或者事务还未提交，请求 B 去数据库查询得到旧值；



那么这时候就会产生数据库和 Redis 数据不一致的问题。因此一般不建议使用这种方式。

如何解决呢？其实最简单的解决办法就是延时双删的策略。就是

- (1) 先淘汰缓存
- (2) 再写数据库
- (3) 休眠 1 秒，再次淘汰缓存

这么做，可以将 1 秒内所造成的缓存脏数据，再次删除。

那么，这个 1 秒怎么确定的，具体该休眠多久呢？

针对上面的情形，自行评估自己的项目的读数据业务逻辑的耗时。然后写数据的休眠时间则在读数据业务逻辑的耗时基础上，加几百 ms 即可。这么做的目的，就是确保读请求结束，写请求可以删除读请求造成的缓存脏数据。

但是上述的保证事务提交完以后再进行删除缓存还有一个问题，就是如果你使用的是 **Mysql 的读写分离的架构**的话，那么其实主从同步之间也会有时间差。

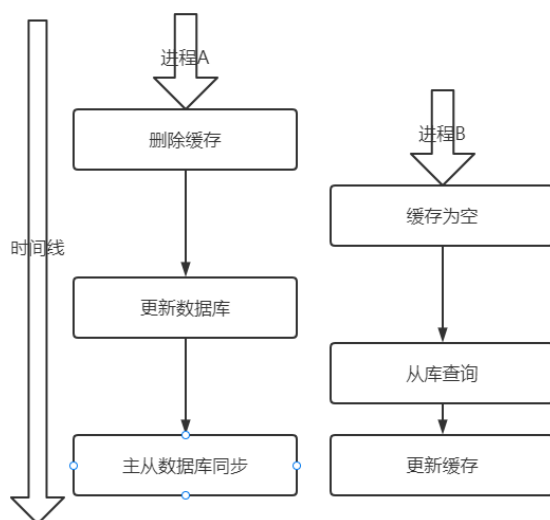
此时来了两个请求，请求 A（更新操作） 和请求 B（查询操作）

请求 A 更新操作，删除了 Redis，

请求主库进行更新操作，主库与从库进行同步数据的操作，

请 B 查询操作，发现 Redis 中没有数据，

去从库中拿去数据，此时同步数据还未完成，拿到的数据是旧数据。



此时的解决办法有两个：

- 1、还是使用双删延时策略。只是，睡眠时间修改为在主从同步的延时时间基础上，加几百 ms。
- 2、就是如果是对 Redis 进行填充数据的查询数据库操作，那么就强制将其指向主库进行查询。

继续深入，采用这种同步淘汰策略，吞吐量降低怎么办？

那就将第二次删除作为异步的。自己起一个线程，异步删除。这样，写的请求就不用沉睡一段时间了，再返回。这么做，加大吞吐量。

不过总的来说，先删除缓存值再更新数据库有可能导致请求因缓存缺失而访问数据库，给数据库带来压力； 2. 业务应用中读取数据库和写缓存的时间有时不好估算，导致延迟双删中的 sleep 时间不好设置。

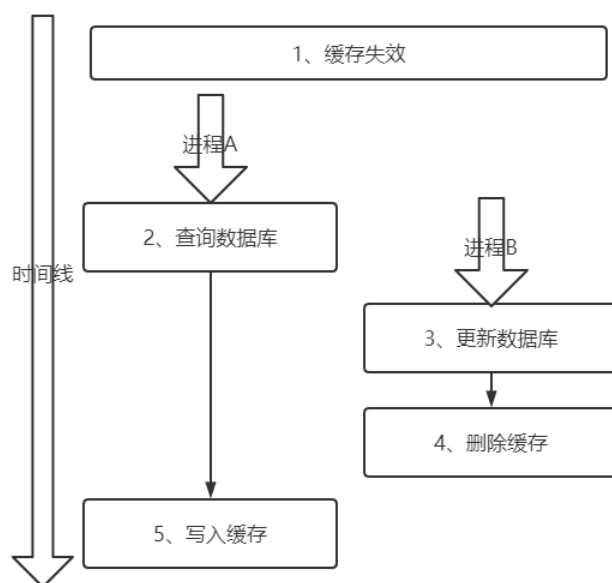
4、先更新 DB，后删除缓存

读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。更新的时候，先更新数据库，然后再删除缓存。

这种情况不存在并发问题么？

依然存在。假设这会有两个请求，一个请求 A 做查询操作，一个请求 B 做更新操作，那么会有如下情形产生

- (1) 缓存刚好失效
- (2) 请求 A 查询数据库，得一个旧值
- (3) 请求 B 将新值写入数据库
- (4) 请求 B 删除缓存
- (5) 请求 A 将查到的旧值写入缓存



如果发生上述情况，确实是会发生脏数据。然而，发生这种情况的概率又有多少呢？

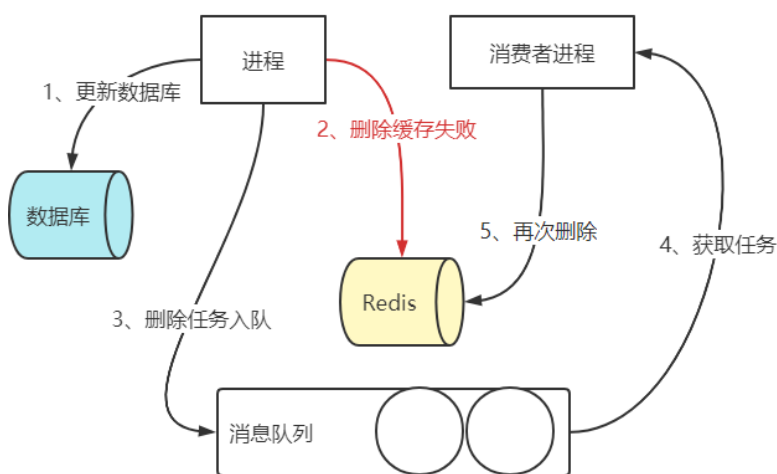
发生上述情况有一个先天性条件，就是步骤（3）的写数据库操作比步骤（2）的读数据库操作耗时更短，才有可能使得步骤（4）先于步骤（5）。可是，大家想想，数据库的读操作的速度远快于写操作的（不然做读写分离干嘛，做读写分离的意义就是因为读操作比较快，耗资源少），因此步骤（3）耗时比步骤（2）更短，这一情形很难出现。一定要解决怎么办？如何解决上述并发问题？

首先，给缓存设有效时间是一种方案。

其次，采用异步延时删除策略。

但是，更新数据库成功了，但是在删除缓存的阶段出错了没有删除成功怎么办？这个问题，在删除缓存类的方案都是存在的，那么此时再读取缓存的时候每次都是错误的数据了。

此时解决方案有两个，一是就是利用消息队列进行删除的补偿。具体的业务逻辑用语言描述如下：

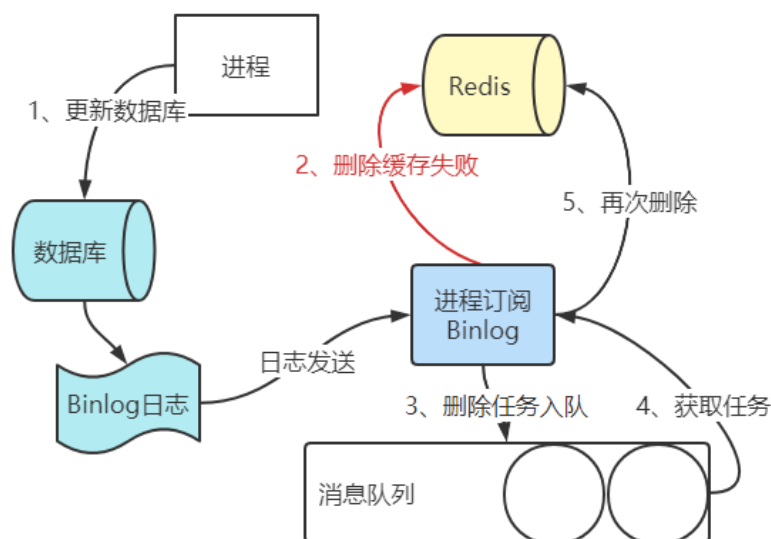


1、请求 A 先对数据库进行更新操作

2、在对 Redis 进行删除操作的时候发现报错，删除失败

- 3、此时将 Redis 的 key 作为消息体发送到消息队列中
- 4、系统接收到消息队列发送的消息后
- 5、再次对 Redis 进行删除操作

但是这个方案会有一个缺点就是会对业务代码造成大量的侵入，深深的耦合在一起，所以这时会有一个优化的方案，我们知道对 Mysql 数据库更新操作后再 binlog 日志中我们都能够找到相应的操作，那么我们可以订阅 Mysql 数据库的 binlog 日志对缓存进行操作。



说到底就是通过数据库的 binlog 来异步淘汰 key，利用工具(canal)将 binlog 日志采集发送到 MQ 中，然后通过 ACK 机制确认处理删除缓存。

先更新 DB，后删除缓存，这种方式，被称为 Cache Aside Pattern，属于缓存更新的设计模式之一。

缓存更新的设计模式

更新缓存的设计模式主要有四种：Cache aside, Read through, Write through, Write behind caching

Cache Aside

这是最常用最常用的 pattern 了。上面已经讲过了，这是标准模式，包括 Facebook 的论文《Scaling Memcache at Facebook》也使用了这个策略。

缓存一致性如果追求强一致，要么串行化，要么使用分布式读写锁，要么通过 2PC 或是 Paxos 协议保证一致性，要么就是拼命的降低并发时脏数据的概率，而一般大厂包括 Facebook 都选择了使用这个降低概率的做法，因为强一致的实现性能往往比较差，而且比较复杂，还要考虑各种容错问题。

Read/Write Through

我们可以看到，在上面的 Cache Aside 套路中，我们的应用代码需要维护两个数据存储，一个是缓存（Cache），一个是数据库（Repository）。所以，应用程序比较啰嗦。而 Read/Write Through 套路是把更新数据库（Repository）的操作由缓存自己代理了，所以，对于应用层来说，就简单很多了。可以理解为，应用认为后端就是一个单一的存储，而存储自己维护自己的 Cache。

Read Through

Read Through 套路就是在查询操作中更新缓存，也就是说，当缓存失效的时候（过期或 LRU 换出），**Cache Aside** 是由调用方负责把数据加载入缓存，而 **Read Through** 则用缓存服务自己来加载，从而对应用方是透明的。

Write Through

Write Through 套路和 **Read Through** 相仿，不过是在更新数据时发生。当有数据更新的时候，如果没有命中缓存，直接更新数据库，然后返回。如果命中了缓存，则更新缓存，然后再由 **Cache** 自己更新数据库。

Write Behind Caching

Write Behind 又叫 **Write Back**。Linux 文件系统的 **Page Cache** 也是同样算法。

Write Back 套路，一句说就是，在更新数据的时候，只更新缓存，不更新数据库，而我们的缓存会异步地批量更新数据库。这个设计的好处就是让数据的 I/O 操作飞快无比，因为异步，**write back** 还可以合并对同一个数据的多次操作，所以性能的提高是相当可观的。

但是，其带来的问题是，数据不是强一致性的，而且可能会丢失（我们知道 Unix/Linux 非正常关机会导致数据丢失，就是因为这个事）。

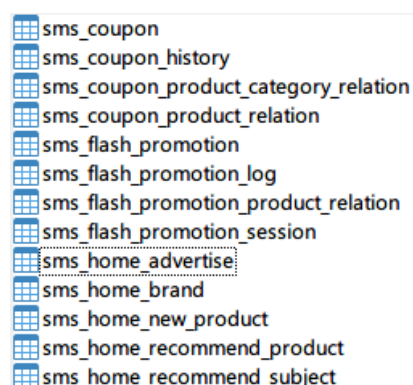
另外，**Write Back** 实现逻辑比较复杂，因为他需要 **track** 有哪数据是被更新了的，需要刷到持久层上。

商城项目的缓存实践

基础实现

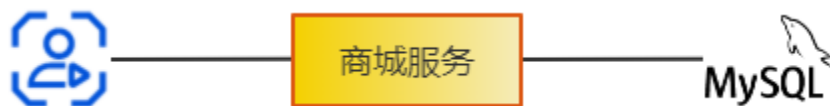
商城项目中有多个地方使用到了缓存来提升性能，最显著的就是 **tulingmall-portal**，商城的首页入口服务。

既然是商城的首页入口，那么必然的它就是整个商城系统访问最多的的一个部分，大体有推荐品牌、人气推荐、新品推荐、轮播广告以及秒杀活动这几个部分，而这几个部分都属于促销信息，数据则存储在



```
sms_coupon
sms_coupon_history
sms_coupon_product_category_relation
sms_coupon_product_relation
sms_flash_promotion
sms_flash_promotion_log
sms_flash_promotion_product_relation
sms_flash_promotion_session
sms_home_advertise
sms_home_brand
sms_home_new_product
sms_home_recommend_product
sms_home_recommend_subject
```

在商城系统的早期（四期及以前）版本，这些数据是直接从数据表中读取然后显示在商城的首页上，访问路径为：



在我们现在的商城系统，做了微服务拆分，上述的数据表全部归属于 tulingmall-promotion 促销管理服务负责访问路径为：



那就意味着不管是早期版本还是微服务版本，不做优化的话，首页其实是无法应对高并发的。

在前面的课程我们说过，如果任何人看到的内容都是一样的（推荐系统要紧密结合大数据，不在本课程的考虑范围），也就是说，对后端服务来说，任何人的查询请求和返回的数据都是一样的。在这种情况下，缓存的命中率非常高，几乎所有的请求都可以命中缓存，很自然，首页上的这些展示很符合这些特性，所以我们想到用缓存来应对首页的高并发。

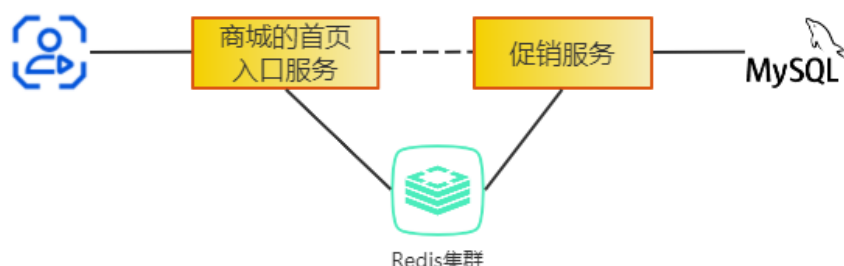
为了保护数据库应对高并发，我们考虑首先将促销信息放入缓存，所以在 tulingmall-promotion 的 HomePromotionServiceImpl 中，这些数据在获取时（以新品推荐为例）：

```
PageHelper.startPage( pageNum: 0, ConstantPromotion.HOME_RECOMMEND_PAGESIZE, orderBy:
SmsHomeNewProductExample example = new SmsHomeNewProductExample();
example.or().andRecommendStatusEqualTo(ConstantPromotion.HOME_PRODUCT_RECOMMEND_YES);
List<Long> newProductIds = smsHomeNewProductMapper.selectProductIdByExample(example);
newProducts = pmsProductClientApi.getProductBatch(newProductIds);
redisOpsExtUtil.putListAllRight(newProductKey, newProducts);
```

可以看到，我们从数据库中获取后，也往 Redis 缓存集群中存放了一份，很自然的，当首页获取新品推荐时，自然就可以 Redis 集群中获得，以下代码在 tulingmall-portal 的 HomeServiceImpl

```
/*从远程(Redis或者对应微服务)获取推荐内容*/
4 usages  2 Mark
public HomeContentResult getFromRemote(){
    List<PmsBrand> recommendBrandList = null;
    List<SmsHomeAdvertise> smsHomeAdvertises = null;
    List<PmsProduct> newProducts = null;
    List<PmsProduct> recommendProducts = null;
    HomeContentResult result = null;
    /*从redis获取*/
    if(promotionRedisKey.isAllowRemoteCache()){
        recommendBrandList = redisOpsUtil.getListAll(promotionRedisKey.getBrandKey
        smsHomeAdvertises = redisOpsUtil.getListAll(promotionRedisKey.getHomeAdver
        newProducts = redisOpsUtil.getListAll(promotionRedisKey.getNewProductKey()
        recommendProducts = redisOpsUtil.getListAll(promotionRedisKey.getRecProduc
    }
    /*redis没有则从微服务中获取*/
    if(CollectionUtil.isEmpty(recommendBrandList)
        ||CollectionUtil.isEmpty(smsHomeAdvertises)
        ||CollectionUtil.isEmpty(newProducts)
        ||CollectionUtil.isEmpty(recommendProducts)) {
        result = promotionFeignApi.content( getType: 0).getData();
```

很自然的，访问路径变为：



通过这种方式，我们以高性能的 Redis 取代 MySQL，并且缩短了整个访问路径，提升了首页服务的性能。

在前面我们说过，缓存一定是离用户越近越好，依据这个原则，首页还有优化的空间，从上面的访问路径可以看到，首页服务需要到 Redis 集群中获得数据用以展示，能不能将缓存的数据再提前呢？于是我们在首页服务内引入了应用级缓存 Caffeine。

TIPS: Caffeine 基于 Google 的 Guava Cache，提供一个性能卓越的本地缓存 (local cache) 实现，也是 SpringBoot 内置的本地缓存实现，有资料表明 Caffeine 性能是 Guava Cache 的 6 倍

```
public class CaffeineCacheConfig {

    @Mark
    @Bean(name = "promotion")
    public Cache<String, HomeContentResult> promotionCache() {
        int rnd = ThreadLocalRandom.current().nextInt( bound: 10);
        return Caffeine.newBuilder()
            // 设置最后一次写入经过固定时间过期
            .expireAfterWrite( duration: 30 + rnd, TimeUnit.MINUTES)
            // 初始的缓存空间大小
            .initialCapacity(20)
            // 缓存的最大条数
            .maximumSize(100)
            .build();
    }
}
```

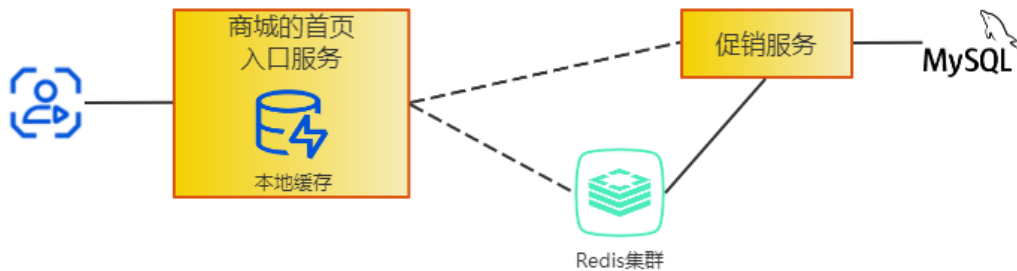
于是在 tulingmall-portal 的 HomeServiceImpl 中，对于查询请求我们先从本地缓存中获取，未获取到再从远程获取

```

public HomeContentResult recommendContent(){
    /*品牌和产品在本地缓存中统一处理，有则视为同有，无则视为同无*/
    final String brandKey = promotionRedisKey.getBrandKey();
    final boolean allowLocalCache = promotionRedisKey.isAllowLocalC
    /*先从本地缓存中获取推荐内容*/
    HomeContentResult result = allowLocalCache ?
        promotionCache.getIfPresent(brandKey) : null;
    if(result == null){
        result = allowLocalCache ?
            promotionCacheBak.getIfPresent(brandKey) : null;
    }
    /*本地缓存中没有*/
    if(result == null){
        log.warn("从本地缓存中获取推荐品牌和商品失败，可能出错或禁用");
        result = getFromRemote();
        if(null != result) {
            promotionCache.put(brandKey, result);
            promotionCacheBak.put(brandKey, result);
        }
    }
}

```

访问路径自然就变为下图所示，并通过这种方式进一步提升了首页的访问性能。



促销数据先从 tulingmall-portal 的本地 Caffeine 缓存获得，如果没有，再从远程获得，获取步骤首先从 Redis 集群获取，Redis 没有，则从 tulingmall-promotion 中获取，tulingmall-promotio 则会从数据库中获取并写入 Redis 集群。此时 tulingmall-portal 获得远程的应答后，将数据写入本地的 Caffeine 缓存。

到了这一步，我们在首页中引入缓存的工作并没有完成，还有几个缓存使用的问题需要解决。

缓存预热

首页服务如果出现重启会导致缓存中的首页数据丢失，导致查询请求直接访问数据库，所以在首页服务中有专门的缓存预热机制，在 tulingmall-portal 启动时从远程获得相关的数据写入本地的 Caffeine 缓存

```

public class PreheatCache implements CommandLineRunner {

    1 usage
    @Autowired
    private HomeService homeService;

    2 Mark
    @Override
    public void run(String... args) throws Exception {
        for(String str : args) {
            log.info("系统启动命令行参数: {}",str);
        }
        homeService.preheatCache();
    }
}

```

```

try {
    if(promotionRedisKey.isAllowLocalCache()){
        final String brandKey = promotionRedisKey.getBrandKey();
        HomeContentResult result = getFromRemote();
        promotionCache.put(brandKey,result);
        promotionCacheBak.put(brandKey,result);
        log.info("promotionCache 数据缓存预热完成");
    }
} catch (Exception e) {
    log.error("promotionCache 数据缓存预热失败:",e);
}

```

在 tulingmall-promotion 中也存在同样的缓存预热机制,负责将数据从 MySQL 数据写入到 Redis 集群中

```

public class PreheatCache implements CommandLineRunner {

    1 usage
    @Autowired
    private HomePromotionService homePromotionService;

    2 Mark
    @Override
    public void run(String... args) throws Exception {
        for(String str : args) {
            log.info("系统启动命令行参数: {}",str);
        }
        homePromotionService.content(ConstantPromotion.HOME_GET_TYPE_ALL);
    }
}

```



```

public HomeContentResult content(int getType) {
    HomeContentResult result = new HomeContentResult();
    if(ConstantPromotion.HOME_GET_TYPE_ALL == getType
        ||ConstantPromotion.HOME_GET_TYPE_BARND == getType){
        //获取推荐品牌
        getRecommendBrand(result);
    }
    if(ConstantPromotion.HOME_GET_TYPE_ALL == getType
        ||ConstantPromotion.HOME_GET_TYPE_NEW == getType){
        getRecommendProducts(result);
    }
    if(ConstantPromotion.HOME_GET_TYPE_ALL == getType
        ||ConstantPromotion.HOME_GET_TYPE_HOT == getType){
        getHotProducts(result);
    }
    if(ConstantPromotion.HOME_GET_TYPE_ALL == getType
        ||ConstantPromotion.HOME_GET_TYPE_AD == getType){
        //获取首页广告
        result.setAdvertiseList(getHomeAdvertiseList());
    }
    return result;
}

```

数据一致性

必然的，推荐品牌、人气推荐、新品推荐、轮播广告以及秒杀活动等促销信息一定存在着变化，当数据库中更新后，就和缓存中的数据不一致了，需要我们更新缓存。

对于 tulingmall-portal 的本地 Caffeine 缓存，我们设置了过期时间 30 分钟

```

return Caffeine.newBuilder()
    // 设置最后一次写入经过固定时间过期
    .expireAfterWrite( duration: 30 + rnd, TimeUnit.MINUTES)
    // 初始的缓存空间大小
    .initialCapacity(20)
    // 缓存的最大条数
    .maximumSize(100)
    .build();

```

并在 RefreshPromotionCache 中以后台任务的形式异步的刷新缓存，每分钟检查一次本地 Caffeine 缓存是否已无效，无效则刷新缓存

```

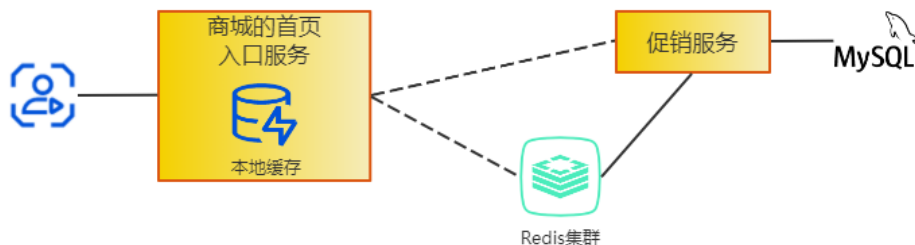
@Async
@Scheduled(initialDelay=5000*60,fixedDelay = 1000*60)
public void refreshCache(){
    if(promotionRedisKey.isAllowLocalCache()){
        log.info("检查本地缓存[promotionCache] 是否需要刷新...");
        final String brandKey = promotionRedisKey.getBrandKey();
        if(null == promotionCache.getIfPresent(brandKey)||null == promotionCacheBak.getIfPresent(brandKey)){
            log.info("本地缓存[promotionCache] 需要刷新");
            HomeContentResult result = homeService.getFromRemote();
            if(null != result){
                if(null == promotionCache.getIfPresent(brandKey)) {
                    promotionCache.put(brandKey,result);
                    log.info("刷新本地缓存[promotionCache] 成功");
                }
                promotionCacheBak.put(brandKey,result);
                log.info("刷新本地缓存[promotionCacheBak] 成功");
            }else{
                log.warn("从远程获得[promotionCache] 数据失败");
            }
        }
    }
}
}

```

而对于 Redis 集群中的数据，则是利用 Canal 监测数据库的更新，然后删除缓存中的对应部分，具体实现在 tulingmall-canal 数据同步程序的 PromotionData 中。Redis 数据的再载入自然由 tulingmall-promotion 负责。

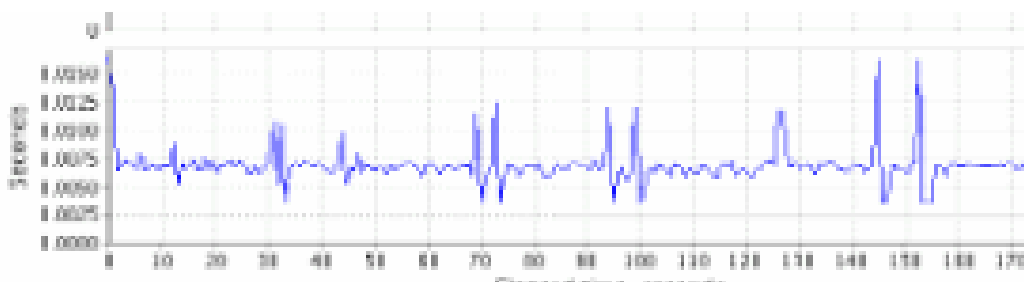
双缓存

从前面的促销数据的获得路径我们知道



促销数据先从 tulingmall-portal 的本地 Caffeine 缓存获得，如果没有，再从远程获得。为了数据的一致性，本地 Caffeine 设置了过期时间，Redis 集群中的数据也会在数据变动后被删除。

这就会造成一种情况，当本地 Caffeine 缓存中已经失效，这个时候就需要去远程获取，最好的情况下可以从 Redis 集群中获得，最坏的情况下需要从数据库中获得。于是在远程获取的情况下，用户的访问链路变长，单次耗时变长，整体会出现类似下图很明显的有规律毛刺现象



如何避免这种情况呢？于是我们引入了双缓存机制：

```
@Bean(name = "promotion")
public Cache<String, HomeContentResult> promotionCache() {
    int rnd = ThreadLocalRandom.current().nextInt( bound: 10);
    return Caffeine.newBuilder()
        // 设置最后一次写入经过固定时间过期
        .expireAfterWrite( duration: 30 + rnd, TimeUnit.MINUTES)
        // 初始的缓存空间大小
        .initialCapacity(20)
        // 缓存的最大条数
        .maximumSize(100)
        .build();
}

/*以双缓存的形式提升首页的访问性能，这个备份缓存其实运行过程中会永不过期
 * 可以作为首页的降级和兜底方案 * */
- Mark
@Bean(name = "promotionBak")
public Cache<String, HomeContentResult> promotionCacheBak() {
    int rnd = ThreadLocalRandom.current().nextInt( bound: 10);
    return Caffeine.newBuilder()
        // 设置最后一次访问经过固定时间过期
        .expireAfterAccess( duration: 41 + rnd, TimeUnit.MINUTES)
        // 初始的缓存空间大小
        .initialCapacity(20)
        // 缓存的最大条数
        .maximumSize(100)
        .build();
}
```

也就是促销数据在本地的缓存我们保存了两份，并且将备份缓存作为降级和兜底方案。

在首页获得促销数据的过程中，我们会在两份本地缓存中获取

```
/*先从本地缓存中获取推荐内容*/
HomeContentResult result = allowLocalCache ?
    promotionCache.getIfPresent(brandKey) : null;
if(result == null){
    result = allowLocalCache ?
        promotionCacheBak.getIfPresent(brandKey) : null;
}
```

只有两份本地缓存都没有，才会到远程获得。

在数据的过期机制上可以看到，我们使用了不同的策略，正式缓存是最后一次写入后经过固定时间过期，备份缓存是设置最后一次访问后经过固定时间过期。这就意味着备份缓存中内容不管是读写后，实际过期时间都会后延，正式缓存中的数据在被读取后，实际过期时间不会后延。

在本地缓存的异步刷新机制上，正式缓存只有无效才会被重新写入，备份缓存无论是否无效都会重新写入，一则可以保证备份缓存中的数据不至于真的永

久无法过期而太旧，二则使备份缓存的过期时间不管用户是否访问首页都可以不断后延。

同时 `tulingmall-portal` 中也提供了专门的缓存管理接口 `CacheManagerController`，方便进行强制本地缓存失效和手动刷新本地缓存。

其他

秒杀的活动信息在首页的是单独处理的，但是处理的思路是一致的，只有细微的地方有所不同，这里不再赘述。

其实首页的这些信息还可以使用固定的图片或者 `Http` 连接作为本地缓存、`Redis` 集群和 `tulingmall-promotion` 微服务均失效的最终的降级和兜底方案，这个可以自行实现。

同时我们系统在缓存的使用上，还有可以改进的空间，比如注册用户的布隆过滤器化以应对缓存穿透，商品信息的缓存化等等。

大厂如何利用 Redis 应对海量和高可用、高并发

用 Redis 构建缓存集群的最佳实践

前面我们说过 `MySQL` 何应对海量数据，应对高并发的方法：

数据量太大导致查询慢的问题该如何解决？

存档历史数据，或者分库分表，这是数据分片。

并发需求太高，系统无法支撑该如何解决？读写分离，这是增加实例数。

数据库宕机该如何解决？增加从节点。主节点宕机的时候用从节点代替主节点。这是主从复制。

这些方法不仅仅是 `MySQL` 所特有的，它们也适用于几乎所有的存储系统。生产系统可用的 `Redis` 缓存集群也是一样的。

`Redis` 从 3.0 版本开始，就提供了官方的集群支持，即 `Redis Cluster`。`Redis Cluster` 集群相较于单个节点的 `Redis`，能保存更多的数据，支持更高的并发，并且可以实现高可用，能够在单个节点故障的情况下，继续提供服务。

与 `MySQL` 分库分表的方式类似，为了能够保存更多的数据，`RedisCluster` 也是通过分片的方式，把数据分布到集群的多个节点上。

复习一下，`Redis Cluster` 是如何实现分片的呢？其引入了一个“槽”（`Slot`）的概念，这个槽就是哈希表中的哈希槽。槽是 `Redis` 分片的基本单位，每个槽里面包含一些键（`Key`）。每个集群的槽数都是固定的 16 384(即 16×1024)个，每个键落在哪个槽中也是固定的，这个算法属于哈希分片算法。

那么，怎么知道哪个槽存放在哪个具体的 `Redis` 节点上的呢？这个映射关系保存在集群的每个 `Redis` 节点上，集群初始化的时候 `Redis` 会自动平均分配这 16 384 个槽，除此之外，还可以通过命令来调整。这个分槽的方法就是分片算法中的查表法。

客户端可以连接集群的任意一个节点来访问集群的数据，当客户端请求一个键的时候，收到请求的那个 Redis 实例首先会通过上面的公式，计算出这个键在哪个槽中，然后再查询槽和节点的映射关系，找到数据所在的真正节点。如果这个节点正好是它自己，那就执行命令返回结果。如果数据不在当前这个节点上，那就向客户端返回一个重定向的命令，告诉客户端，应该去哪个节点上请求这个键的数据，让后客户端会再次连接正确的节点来访问该键。

解决完分片问题之后，Redis Cluster 就可以通过水平扩容来增加集群的存储容量了。当然每次向集群增加节点的时候都需要进行槽的迁移，这个可以自动迁移也可以手动迁移。

分片可以解决 Redis 保存海量数据的问题，并且从客观上提升 Redis 的并发能力和查询性能。但是并不能解决高可用的问题，每个节点只保存了整个集群数据的一个子集，任何一个节点宕机，都会导致这个宕机节点上的那部分数据无法访问。

那么 Redis Cluster 是如何解决高可用问题的呢？增加从节点，做主从复制。Redis Cluster 支持为每个分片增加一个或多个从节点。并且内部通过选举来确定有主节点的存活。

一般来说，Redis Cluster 进行分片之后，每个分片都会承接一部分并发请求，加上 Redis 本身单节点的性能就非常高，所以大部分情况下，不需要再像 MySQL 那样做读写分离来解决高并发的数据问题。默认情况下，集群的读写请求都是由主节点负责的，从节点的作用只是作为一个热备。当然了，Redis Cluster 也支持读写分离，以及在从节点上读取数据，但是需要客户端的支持。

在我们的商城项目中，就使用了 Redis Cluster。

为什么大厂不用 Redis Cluster 构建集群

Redis Cluster 的优点是易于使用。分片、主从复制、弹性扩容这些功能都可以实现自动化，单的部署就可以实现一个大容量、高可靠、高可用的 Redis 集群，并且对于应用来说，其近乎透明。

所以 Redis Cluster 非常适合用于构建中小规模的 Redis 集群，这里的中小规模指的是，大概几个到几十个节点这样规模的 Redis 集群。

不过 Redis Cluster 不太适合构建超大规模集群，主要原因是，它采用了去中心化的设计。Redis Cluster 采用了一种去中心化的流言（Gossip）协议来传播集群配置的变化。

流言协议的优点是去中心化，就像八卦传闻一样，不需要组织，人们会自发传播，通过人传人，八卦信息会大范围扩散出去。因此流言协议的部署和维护非常简单，由于没有中心节点，因此不存在单点故障，任何一个节点出现故障，都不会影响信息在集群中的传播。不过流言协议也有缺点，那就是传播速度比较慢，而且是集群规模越大，传播的速度就越慢。关于这一点也很好理解，在集群规模比较大的情况下，数据不同步的问题会明显放大，而且还带有一定的不确定性，如果出现问题，就会很难排查到问题所在。

这里也有一个与之相关的面试题：为什么槽的范围是 0 ~16383？

redis 的作者在 github 上有个回答:<https://github.com/redis/redis/issues/2576> 说明了原因

Redis 集群中,在握手成功后,节点之间会定期发送 ping/pong 消息,交换数据信息,集群中节点数量越多,消息体内容越大,比如说 10 个节点的状态信息约 1kb。同时 redis 集群内节点,每秒都在发 ping 消息。在这种情况下,一个总节点数为 200 的 Redis 集群,默认情况下,这时 ping/pong 消息占用带宽达到 25M,这还只是槽的范围是 0 ~16383 的情况。

那么如果槽位设计为 65536,光发送心跳信息的信息头可达 8k,发送的心跳包过于庞大,非常浪费带宽。

其次 redis 的集群主节点越多,心跳包的消息体内携带的数据越多。如果节点过 1000 个,也会导致网络拥堵。因此 redis 作者,不建议 redis cluster 节点数量超过 1000 个。

那么,对于节点数在 1000 以内的 redis cluster 集群,16384 个槽位够用了,可以以确保每个 master 有足够的插槽,没有必要拓展到 65536 个。

所以 Redis 作者决定取 16384 个槽,作为一个设计权衡。

如何用 Redis 构建超大规模集群

由于 Redis Cluster 不适合构建成大规模的集群,因此很多大型企业都会选择自主搭建 Redis 集群。虽然不同企业的解决方案会有其自身的特色,但是总体的架构都是大同小异的。

用 Redis 构建超大规模集群可以采用多种方式,比较常用的方法是采用一种基于代理的方式,即在客户端和 Redis 节点之间,增加一层代理服务这个代理服务可以起到如下三个方面的作用。

第一个作用是,负责在客户端和 Redis 节点之间转发请求和响应。客户端只与代理服务打交道,代理收到客户端的请求之后,会转发到对应的 Redis 节点上,节点返回的响应再经由代理转发返回给客户端。

第二个作用是,负责监控集群中所有 Redis 节点的状态,如果发现存在问题节点,就及时进行主从切换。

第三个作用是,维护集群的元数据,这个元数据主要是集群所有节点的主从信息,以及槽和节点的关系映射表。像开源的 Redis 集群方案 twemproxy 和 Codis,采用的都是代理服务这种架构。

代理服务架构最大的优点是对客户端透明,从客户端的视角来看,整个集群就像是一个超大容量的单节点 Redis 一样。除此之外,由于分片算法是受代理服务控制的,因此扩容比较方便,新节点加入集群后,直接修改代理服务中的元数据就可以完成扩容。

不过 Redis 集群代理架构的缺点也很突出,由于增加了一层代理转发,因此每次数据访问的链路变得更长了,这必然会导致一定的性能损失。而且代理服务本身也是集群的单点。当然,我们可以把代理服务也做成一个集群来解决单点问题,那样集群就更复杂了。

用 Redis 构建超大规模集群的另外一种方式是不使用代理服务,只是把代理服务的寻址功能前移到客户端中。客户端在发起请求之前,首先会查询元数据,

这样就可以知道要访问的是哪个分片和哪个节点，然后直连对应的 **Redis** 节点访问数据即可。

当然，客户端不用每次都去查询元数据，因为这个元数据基本上是很少会发生变化的，客户端可以自行缓存元数据，这样访问性能基本上就与单机版的 **Redis** 一样了。如果某个分片的主节点宕机了，就会选举新的主节点，并更新元数据中的信息。对集群的扩容操作也比较简单，除了必须完成数据的迁移工作之外，再更新一下元数据就可以了。

当然元数据服务仍然是一个单点，但是它的数据量不大，访问量也不大，相对来说比较容易实现。利用已有的 **ZooKeeper**、**etcd** 甚至 **MySQL** 都可以被用来实现上述元数据服务。

定制客户端的 **Redis** 集群方案应该是最适合超大规模 **Redis** 集群的方案，在性能、弹性、高可用等几个方面的表现都非常好，缺点是整个架构比较复杂，客户端不能通用，需要开发定制化的 **Redis** 客户端，所以往往只有规模足够大的企业才能负担得起高昂的定制开发成本。

本文档分享地址：

<http://note.youdao.com/noteshare?id=87daa6069abcf85c475d8729c1751464&sub=DEB9E636579341109FF636BE47984A4C>