<http://mp.weixin.qq.com/s/8RM6U8UqWTDp29DB_hiYVQ>

**架构师之路2017半年精选40篇**

原创 2017-06-24  58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960243&idx=1&sn=c5d0de35c6d87e08370eef4179ff66bb&chksm=bd2d066f8a5a8f79a3478591e44054a1838e6aa8a35e544a5eb42b38cb274de6501a6b3b77e5&mpshare=1&scene=23&srcid=0904rIpHuSoU35kdA35BJ5wU" \l "#)

2017上半年精选索引，点击标题阅读。

**【特别推荐】**

**《[架构师之路2016年精选66篇](http://mp.weixin.qq.com/s/OlFKpcnBOgcPZmjvdzCCiA" \t "_blank)》**

发起一个活动，一起携手见证“架构师之路”的第一篇10w+，还差3000阅读，邀您一起转发。

**【通用设计与方法论】**

**《[单KEY业务，数据库水平切分架构实践](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960212&idx=1&sn=ab4c52ab0309f7380f7e0207fa357128&chksm=bd2d06488a5a8f5e3b7c9de0cc5936818bd9a6ed4058679ae8d819175e0693c6fbd9cdea0c87&scene=21" \l "wechat_redirect" \t "_blank)》**

《[架构设计中常见“反向依赖”与解耦方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960049&idx=1&sn=2787e73be4ea31f52213a2b934a16e09&chksm=bd2d072d8a5a8e3b71cef58f313743a6db6f83889748deb7f8f866817511df98adc5e03db39c&scene=21" \l "wechat_redirect" \t "_blank)》

《[互联网架构如何实现“高可用”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959728&idx=1&sn=933227840ec8cdc35d3a33ae3fe97ec5&chksm=bd2d046c8a5a8d7a13551124af36bedf68f7a6e31f6f32828678d2adb108b86b7e08c678f22f&scene=21" \l "wechat_redirect" \t "_blank)》

《[互联网架构如何实现“高并发”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959830&idx=1&sn=ce1c5a58caed227d7dfdbc16d6e1cea4&chksm=bd2d07ca8a5a8edc45cc45c4787cc72cf4c8b96fb43d2840c7ccd44978036a7d39a03dd578b5&scene=21" \l "wechat_redirect" \t "_blank)》

**《[典型数据库架构设计与实践](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960158&idx=1&sn=019e9dca6e074f62478b2562044cf8de&chksm=bd2d06828a5a8f945e1e6c2aa4702f48b7857ac9a79acf867496f25b28c86e2f28bed19b38e4&scene=21" \l "wechat_redirect" \t "_blank)》**

**【典型架构实践】**

**《[计数系统架构设计一次搞定](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960185&idx=1&sn=0acd4a563c8c9684373fd05c116c4441&chksm=bd2d06a58a5a8fb3281acd95a7d9494161e75dcd27e95fdd526fce2b031ba79c4153bf255cee&scene=21" \l "wechat_redirect" \t "_blank)》**

**《[session一致性架构设计实践](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960128&idx=1&sn=8e0e409b10ab9db549432af461385314&chksm=bd2d069c8a5a8f8ab5cdee602d4062bbdbb25da290668515d36682afa854e374d2a5ff02004b&scene=21" \l "wechat_redirect" \t "_blank)》**

《[TCP接入层的负载均衡、高可用、扩展性架构设计](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960086&idx=1&sn=70bbe7165ecddc7896767f4503a927fe&chksm=bd2d06ca8a5a8fdc67fcacb169583f53a968fdf623a20395926059d44e6abae6b2e56ff1f9f9&scene=21" \l "wechat_redirect" \t "_blank)》

**《[配置中心架构设计演进](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960061&idx=1&sn=3747338a91e85fc33f43d9c1bb98ab10&chksm=bd2d07218a5a8e37ad2e5078736d2449a6ba1e0d75bb18d09990a5f3dcc3af2b7ff414158adb&scene=21" \l "wechat_redirect" \t "_blank)》**

《[跨公网调用的大坑与架构优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960105&idx=1&sn=0069f2264e227e86a63ee50a4899e0a7&chksm=bd2d06f58a5a8fe33271ed3a378932ad023af7a9004d7fdb44e30a53d53928f984f293a41256&scene=21" \l "wechat_redirect" \t "_blank)》

**《[DNS在架构设计中的巧用](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960118&idx=1&sn=767e038cb4378be1c88dd42569a9264f&chksm=bd2d06ea8a5a8ffc7c69c71c2153b10565ecf72b6957c5667fbf9099c9f5478111132e967a35&scene=21" \l "wechat_redirect" \t "_blank)》**

《[互联网智能广告系统简易流程与架构](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960148&idx=1&sn=7c8a2d789fb20981355e49c3defe0229&chksm=bd2d06888a5a8f9e5ef5cf6cfa415d06800f5468148a0ba7a802e646aeec9610823ec0fc6d5d&scene=21" \l "wechat_redirect" \t "_blank)》

**【数据库】**

**《[100亿数据量1万属性10万并发数据库架构设计](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959855&idx=1&sn=f33abe8ec598c273f29cebb9365ece59&chksm=bd2d07f38a5a8ee58a944507a134e1da1efc3ac9c4d1c4cff261137cd986e51f5fe7cee9de15&scene=21" \l "wechat_redirect" \t "_blank)》**

《[数据库秒级平滑扩容架构方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959883&idx=1&sn=e7df8510c7096a5b069e0f12eaaca010&chksm=bd2d07978a5a8e815c2ae41b16b6b4c579923502fb919008a22bb108a1e920109f25387f8903&scene=21" \l "wechat_redirect" \t "_blank)》

**《[58到家数据库30条军规](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959906&idx=1&sn=2cbdc66cfb5b53cf4327a1e0d18d9b4a&chksm=bd2d07be8a5a8ea86dc3c04eced3f411ee5ec207f73d317245e1fefea1628feb037ad71531bc&scene=21" \l "wechat_redirect" \t "_blank)》**

《[再议数据库军规](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959910&idx=1&sn=6b6853b70dbbe6d689a12a4a60b84d8b&chksm=bd2d07ba8a5a8eac6783bac951dba345d865d875538755fe665a5daaf142efe670e2c02b7c71&scene=21" \l "wechat_redirect" \t "_blank)》

**《[业界难题-“夸库分页”的四种方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959942&idx=1&sn=e9d3fe111b8a1d44335f798bbb6b9eea&chksm=bd2d075a8a5a8e4cad985b847778aa83056e22931767bb835132c04571b66d5434020fd4147f&scene=21" \l "wechat_redirect" \t "_blank)》**

《[100亿数据平滑迁移却不影响服务](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959992&idx=1&sn=eb2fbd7d7922db42a593c304e50a65b7&chksm=bd2d07648a5a8e72d489022ec6006274d7e43ab48449b255d5661658c2af8e9221977a9609ed&scene=21" \l "wechat_redirect" \t "_blank)》

《[MySQL-proxy数据库中间件架构](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960133&idx=1&sn=07acfeac5306f92f81dfccf4c880da4f&chksm=bd2d06998a5a8f8fc8c3bbb2c2ea01825ebe6576a35aa4d9789774dba0f7964baf4edf584f1b&scene=21" \l "wechat_redirect" \t "_blank)》

《[用uid分库，那uname上的查询怎么办](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960032&idx=1&sn=1f818d415842d544dc33ab78e8adffea&chksm=bd2d073c8a5a8e2a38e7534d57ca0cad613893b9c3b578a220c4beef5d7495c5a1569cf4768e&scene=21" \l "wechat_redirect" \t "_blank)》

**【一分钟系列】**

《[1分钟了解四层/七层反向代理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960131&idx=1&sn=a3bbcbe03f9e12d32ba751ce6ffae067&chksm=bd2d069f8a5a8f895fed39cad842f6f5a390bb18493f964b910270128f19f0b8af1d1f30b5c7&scene=21" \l "wechat_redirect" \t "_blank)》

**《[1分钟写好连接池](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959821&idx=1&sn=4ede084b05ce81a9a5ddb87ec62434bb&chksm=bd2d07d18a5a8ec7726619dbb9f1e99df8239ebd07d5f01d748e01c08dd543f0a434945301c6&scene=21" \l "wechat_redirect" \t "_blank)》**

《[1分钟理解负载](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959912&idx=1&sn=ebe3320f91f11df7e1d0f54ec639f31c&chksm=bd2d07b48a5a8ea2356749aecef4d7d1a9e9ca75e02b51da255393396165e50a0a6aa2ea80b5&scene=21" \l "wechat_redirect" \t "_blank)》

《[1分钟理解领导者-跟随者线程模型](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959914&idx=1&sn=3600fc162970b39afe25d65bee10cd2e&chksm=bd2d07b68a5a8ea03c2b1372345991b74aa412b7fd96cc82e9ed0eb38e8dd481bf5d8b7b631c&scene=21" \l "wechat_redirect" \t "_blank)》

**【一致性问题】**

《[库存扣多了，到底怎么整](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960197&idx=1&sn=2e5c17d521772d28d39f31af5d22b34a&chksm=bd2d06598a5a8f4f9de2da89ba8fab711823935442fc632b65d461c923852485ff392c987568&scene=21" \l "wechat_redirect" \t "_blank)》

《[再议库存扣减多种方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960200&idx=1&sn=fdec629caceee07b3946b6c338b8ceb7&chksm=bd2d06548a5a8f424dd32be960222edf5cecc3c1e5a8fcbb4cfff35e7da6787e702861131597&scene=21" \l "wechat_redirect" \t "_blank)》

**《[浅谈CAS在分布式ID生成方案上的应用](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960223&idx=1&sn=121716347174eedd6636b1c2c0b96047&chksm=bd2d06438a5a8f55d35add7dc99940c280264fa7caeacb21f7dc79884d00c050d0367a745f87&scene=21" \l "wechat_redirect" \t "_blank)》**

《[CAS下ABA问题及优化方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960240&idx=1&sn=18c4ce7c3baf6705940847c1c9bf49e3&chksm=bd2d066c8a5a8f7a27acb5603611b5cbcf6396a8a5e5d06571dcd41deaa64e615b097e84fef3&scene=21" \l "wechat_redirect" \t "_blank)》

**【搜索架构】**

**《[深入浅出搜索引擎架构、方案与细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959895&idx=1&sn=de25ce2544c088ff9be0b93fd3ea4d15&chksm=bd2d078b8a5a8e9d5ae4339a683d3f980ff2994f3c10c4081c7bab7f0d77f37521de95e974bf&scene=21" \l "wechat_redirect" \t "_blank)》**

《[如何迅猛的实现搜索需求](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959917&idx=1&sn=8faeae7419a756b0c355af2b30c255df&chksm=bd2d07b18a5a8ea75f16f7e98ea897c7e7f47a0441c64bdaef8445a2100e0bdd2a7de99786c0&scene=21" \l "wechat_redirect" \t "_blank)》

《[58同城如何检索到1秒前发布的帖子](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959949&idx=1&sn=83f78cf6293714bd1fd97a11ff7c2c35&chksm=bd2d07518a5a8e47e6fce9fc03cddec1d8a43f2b4ac67cfbbf73a55143593da8a132da7a0815&scene=21" \l "wechat_redirect" \t "_blank)》

**【消息总线架构】**

**《[到底什么时候该使用MQ](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960012&idx=1&sn=c6af5c79ecead98daa4d742e5ad20ce5&chksm=bd2d07108a5a8e0624ae6ad95001c4efe09d7ba695f2ddb672064805d771f3f84bee8123b8a6&scene=21" \l "wechat_redirect" \t "_blank)》**

《[MQ如何实现消息幂等](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960002&idx=1&sn=c0775231bccf002c3178eabe43f1cdcb&chksm=bd2d071e8a5a8e08c3a5287247ea41dee6b2621e6ffafbf909ec1e8a866b7c816eeeea227246&scene=21" \l "wechat_redirect" \t "_blank)》

《[MQ如何快速实现削峰填谷](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960021&idx=1&sn=4bbe275c249a70ab20e36959fc01d4e0&chksm=bd2d07098a5a8e1fd9b505778b551002ab59c35953fa3deaaddc79e3f21bcea5ff48076b4a89&scene=21" \l "wechat_redirect" \t "_blank)》

《[定时任务如何高效触发](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959957&idx=1&sn=a82bb7e8203b20b2a0cb5fc95b7936a5&chksm=bd2d07498a5a8e5f9f8e7b5aeaa5bd8585a0ee4bf470956e7fd0a2b36d132eb46553265f4eaf&scene=21" \l "wechat_redirect" \t "_blank)》

《[快速实现“延迟消息”功能](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959961&idx=1&sn=afec02c8dc6db9445ce40821b5336736&chksm=bd2d07458a5a8e5314560620c240b1c4cf3bbf801fc0ab524bd5e8aa8b8ef036cf755d7eb0f6&scene=21" \l "wechat_redirect" \t "_blank)》

《[MQ如何实现消息必达](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959966&idx=1&sn=068a2866dcc49335d613d75c4a5d1b17&chksm=bd2d07428a5a8e54162ad8ea8e1e9302dfaeb664cecc453bd16a5f299820755bd2e1e0e17b60&scene=21" \l "wechat_redirect" \t "_blank)》

**【通用】**

**《[如何精准理解leader布置的任务](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960035&idx=1&sn=e0a14e8539b001e4d6d149164987235b&chksm=bd2d073f8a5a8e290202fcdfac077744eab84e6549e0608698ae5e1d3ed221ee97ca21f42e6f&scene=21" \l "wechat_redirect" \t "_blank)》**

《[如何快速精准的和leader沟通](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960073&idx=1&sn=3ac3bb78bb643d73e9e68d713d6eceeb&chksm=bd2d06d58a5a8fc378e7cde71808c09dea9b4d5462deec0238a3d6f0ade62976a3d2bd0faafb&scene=21" \l "wechat_redirect" \t "_blank)》

《[迟到是会议最大的敌人](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960065&idx=1&sn=94cfa99d6244d68a6d1c4ca8aeb723d7&chksm=bd2d06dd8a5a8fcb406d89e8817892948e67d7bb8adca5a5004ae46dde40099e12a39754be9b&scene=21" \l "wechat_redirect" \t "_blank)》

《[运维苦，运维说说心底话](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960076&idx=1&sn=bf57176ba98c6e8c376ee6705405c925&chksm=bd2d06d08a5a8fc64ef401a40c24ae92cfab43c25a60c1f95e1ab6d72ca770c8f825b0a03c23&scene=21" \l "wechat_redirect" \t "_blank)》

《[职场中的选择与拒绝](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960095&idx=1&sn=227b8e3ee3027fcf2180fa12f771995e&chksm=bd2d06c38a5a8fd58610bd771b18ddf257e14dc2248101b249ae59432ed257d4bbfd02f2efaf&scene=21" \l "wechat_redirect" \t "_blank)》

《[罗振宇送给新员工的四句话](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960162&idx=1&sn=0a4814e76ae98f7b4597205bc9b4402c&chksm=bd2d06be8a5a8fa8ea9cac53144d780160c853329af461b6ffa2d8002f93425c394140d1edb3&scene=21" \l "wechat_redirect" \t "_blank)》

若有收获，帮忙转发。

下半年，继续上路，**你的支持是我最大的动力**。

## 单KEY业务，数据库水平切分架构实践

原创 2017-06-16 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960212&idx=1&sn=ab4c52ab0309f7380f7e0207fa357128&chksm=bd2d06488a5a8f5e3b7c9de0cc5936818bd9a6ed4058679ae8d819175e0693c6fbd9cdea0c87&scene=21" \l "#)

提醒，本文较长，可提前收藏/转发。

本文将以“用户中心”为例，介绍“单KEY”类业务，随着数据量的逐步增大，数据库性能显著降低，数据库水平切分相关的架构实践：

* 如何来实施水平切分
* 水平切分后常见的问题
* 典型问题的优化思路及实践

**一、用户中心**

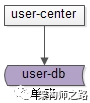
用户中心是一个非常常见的业务，主要提供用户注册、登录、信息查询与修改的服务，其核心元数据为：

User(uid, login\_name, passwd, sex, age, nickname, …)

其中：

* uid为用户ID，主键
* login\_name, passwd, sex, age, nickname, …等用户属性

数据库设计上，一般来说在业务**初期**，单库单表就能够搞定这个需求，典型的架构设计为：

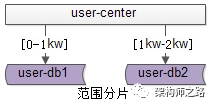


* user-center：用户中心服务，对调用者提供友好的RPC接口
* user-db：对用户进行数据存储

**二、用户中心水平切分方法**

当**数据量越来越大时**，需要对数据库进行水平切分，常见的水平切分算法有**“范围法”和“哈希法”**。

**范围法**，以用户中心的业务主键uid为划分依据，将数据水平切分到两个数据库实例上去：



* user-db1：存储0到1千万的uid数据
* user-db2：存储1到2千万的uid数据

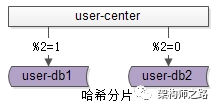
范围法的**优点**是：

* 切分策略简单，根据uid，按照范围，user- center很快能够定位到数据在哪个库上
* 扩容简单，如果容量不够，只要增加user-db3即可

范围法的**不足**是：

* uid必须要满足递增的特性
* 数据量不均，新增的user-db3，在初期的数据会比较少
* 请求量不均，一般来说，新注册的用户活跃度会比较高，故user-db2往往会比user-db1负载要高，导致服务器利用率不平衡

**哈希法**，也是以用户中心的业务主键uid为划分依据，将数据水平切分到两个数据库实例上去：



* user-db1：存储uid取模得1的uid数据
* user-db2：存储uid取模得0的uid数据

哈希法的**优点**是：

* 切分策略简单，根据uid，按照hash，user-center很快能够定位到数据在哪个库上
* 数据量均衡，只要uid是均匀的，数据在各个库上的分布一定是均衡的
* 请求量均衡，只要uid是均匀的，负载在各个库上的分布一定是均衡的

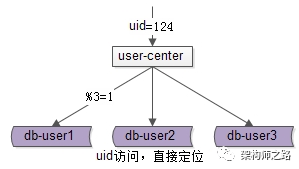
哈希法的**不足**是：

* 扩容麻烦，如果容量不够，要增加一个库，重新hash可能会导致数据迁移，如何平滑的进行数据迁移，是一个需要解决的问题

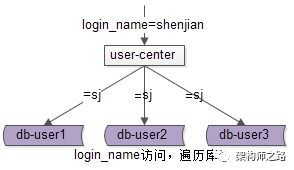
**三、用户中心水平切分后带来的问题**

使用uid来进行水平切分之后，整个用户中心的业务访问会遇到什么问题呢？

对于uid属性上的查询可以直接路由到库，假设访问uid=124的数据，取模后能够直接定位db-user1：



对于非uid属性上的查询，例如login\_name属性上的查询，就悲剧了：



假设访问login\_name=shenjian的数据，由于不知道数据落在哪个库上，往往需要遍历所有库，当分库数量多起来，性能会显著降低。

如何解决分库后，非uid属性上的查询问题，是后文要重点讨论的内容。

**四、用户中心非uid属性查询需求分析**

任何脱离业务的架构设计都是耍流氓，在进行架构讨论之前，先来对业务进行简要分析，看非uid属性上有哪些查询需求。

根据楼主这些年的架构经验，用户中心非uid属性上经常有两类业务需求：

（1）**用户侧，前台访问**，最典型的有两类需求

用户登录：通过login\_name/phone/email查询用户的实体，1%请求属于这种类型

用户信息查询：登录之后，通过uid来查询用户的实例，99%请求属这种类型

用户侧的查询基本上是单条记录的查询，访问量较大，服务需要高可用，并且对一致性的要求较高。

（2）**运营侧，后台访问**，根据产品、运营需求，访问模式各异，按照年龄、性别、头像、登陆时间、注册时间来进行查询。

运营侧的查询基本上是批量分页的查询，由于是内部系统，访问量很低，对可用性的要求不高，对一致性的要求也没这么严格。

这两类不同的业务需求，应该使用什么样的架构方案来解决呢？

**五、用户中心水平切分架构思路**

用户中心在数据量较大的情况下，使用uid进行水平切分，对于非uid属性上的查询需求，架构设计的核心思路为：

* 针对**用户侧**，应该采用“建立非uid属性到uid的映射关系”的架构方案
* 针对**运营侧**，应该采用“前台与后台分离”的架构方案

**六、用户中心-用户侧最佳实践**

**【索引表法】**

**思路**：uid能直接定位到库，login\_name不能直接定位到库，如果通过login\_name能查询到uid，问题解决

**解决方案**：

* 建立一个索引表记录login\_name->uid的映射关系
* 用login\_name来访问时，先通过索引表查询到uid，再定位相应的库
* 索引表属性较少，可以容纳非常多数据，一般不需要分库
* 如果数据量过大，可以通过login\_name来分库

潜在**不足**：多一次数据库查询，性能下降一倍

**【缓存映射法】**

**思路**：访问索引表性能较低，把映射关系放在缓存里性能更佳

**解决方案**：

* login\_name查询先到cache中查询uid，再根据uid定位数据库
* 假设cache miss，采用扫全库法获取login\_name对应的uid，放入cache
* login\_name到uid的映射关系不会变化，映射关系一旦放入缓存，不会更改，无需淘汰，缓存命中率超高
* 如果数据量过大，可以通过login\_name进行cache水平切分

潜在**不足**：多一次cache查询

**【login\_name生成uid】**

**思路**：不进行远程查询，由login\_name直接得到uid

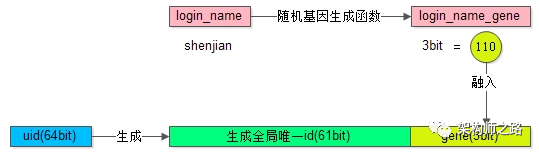
**解决方案**：

* 在用户注册时，设计函数login\_name生成uid，uid=f(login\_name)，按uid分库插入数据
* 用login\_name来访问时，先通过函数计算出uid，即uid=f(login\_name)再来一遍，由uid路由到对应库

潜在**不足**：该函数设计需要非常讲究技巧，有uid生成冲突风险

**【login\_name基因融入uid】**

**思路**：不能用login\_name生成uid，可以从login\_name抽取“基因”，融入uid中



假设分8库，采用uid%8路由，潜台词是，uid的最后3个bit决定这条数据落在哪个库上，这3个bit就是所谓的“基因”。

**解决方案**：

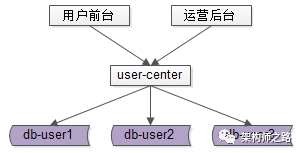
* 在用户注册时，设计函数login\_name生成3bit基因，login\_name\_gene=f(login\_name)，如上图粉色部分
* 同时，生成61bit的全局唯一id，作为用户的标识，如上图绿色部分
* 接着把3bit的login\_name\_gene也作为uid的一部分，如上图屎黄色部分
* 生成64bit的uid，由id和login\_name\_gene拼装而成，并按照uid分库插入数据
* 用login\_name来访问时，先通过函数由login\_name再次复原3bit基因，login\_name\_gene=f(login\_name)，通过login\_name\_gene%8直接定位到库

**七、用户中心-运营侧最佳实践**

前台**用户侧**，业务需求基本都是单行记录的访问，只要建立非uid属性 login\_name / phone / email 到uid的映射关系，就能解决问题。

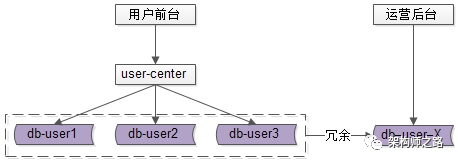
后台**运营侧**，业务需求各异，基本是批量分页的访问，这类访问计算量较大，返回数据量较大，比较消耗数据库性能。

如果此时前台业务和后台业务公用一批服务和一个数据库，有可能导致，由于后台的“少数几个请求”的“批量查询”的“低效”访问，导致数据库的cpu偶尔瞬时100%，影响前台正常用户的访问（例如，登录超时）。



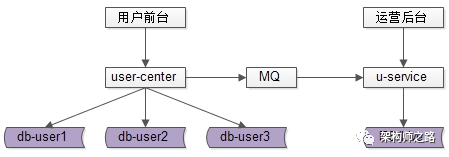
而且，为了满足后台业务各类“奇形怪状”的需求，往往会在数据库上建立各种索引，这些索引占用大量内存，会使得用户侧前台业务uid/login\_name上的查询性能与写入性能大幅度降低，处理时间增长。

对于这一类业务，应该采用“前台与后台分离”的架构方案：



用户侧前台业务需求架构依然不变，产品运营侧后台业务需求则**抽取独立的web / service / db 来支持**，解除系统之间的耦合，对于“业务复杂”“并发量低”“无需高可用”“能接受一定延时”的后台业务：

* 可以去掉service层，在运营后台web层通过dao直接访问db
* 不需要反向代理，不需要集群冗余
* 不需要访问实时库，可以通过MQ或者线下异步同步数据
* 在数据库非常大的情况下，可以使用更契合大量数据允许接受更高延时的“索引外置”或者“HIVE”的设计方案



**八、总结**

将以“用户中心”为典型的“单KEY”类业务，水平切分的架构点，本文做了这样一些介绍。

**水平切分方式**：

* 范围法
* 哈希法

**水平切分后碰到的问题**：

* 通过uid属性查询能直接定位到库，通过非uid属性查询不能定位到库

**非uid属性查询的典型业务**：

* 用户侧，前台访问，单条记录的查询，访问量较大，服务需要高可用，并且对一致性的要求较高
* 运营侧，后台访问，根据产品、运营需求，访问模式各异，基本上是批量分页的查询，由于是内部系统，访问量很低，对可用性的要求不高，对一致性的要求也没这么严格

**这两类业务的架构设计思路**：

* 针对用户侧，应该采用“建立非uid属性到uid的映射关系”的架构方案
* 针对运营侧，应该采用“前台与后台分离”的架构方案

**用户前台侧，“建立非uid属性到uid的映射关系”最佳实践**：

* 索引表法：数据库中记录login\_name->uid的映射关系
* 缓存映射法：缓存中记录login\_name->uid的映射关系
* login\_name生成uid
* login\_name基因融入uid

**运营后台侧，“前台与后台分离”最佳实践**：

* 前台、后台系统web/service/db分离解耦，避免后台低效查询引发前台查询抖动
* 可以采用数据冗余的设计方式
* 可以采用“外置索引”（例如ES搜索系统）或者“大数据处理”（例如HIVE）来满足后台变态的查询需求

其他类型业务的水平切分架构方案，未来和大家聊。

希望大伙有收获。

相关推荐：

[100亿数据平滑数据迁移,不影响服务](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959992&idx=1&sn=eb2fbd7d7922db42a593c304e50a65b7&chksm=bd2d07648a5a8e72d489022ec6006274d7e43ab48449b255d5661658c2af8e9221977a9609ed&scene=21" \l "wechat_redirect" \t "_blank)

[数据库秒级平滑扩容架构方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959883&idx=1&sn=e7df8510c7096a5b069e0f12eaaca010&chksm=bd2d07978a5a8e815c2ae41b16b6b4c579923502fb919008a22bb108a1e920109f25387f8903&scene=21" \l "wechat_redirect" \t "_blank)

## 换IP的是你，凭啥重启的却是我？

原创 2017-04-19 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960049&idx=1&sn=2787e73be4ea31f52213a2b934a16e09&chksm=bd2d072d8a5a8e3b71cef58f313743a6db6f83889748deb7f8f866817511df98adc5e03db39c&scene=21" \l "#)

**一、缘起**

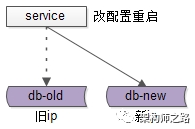
很多公司，技术经常遇到这样的场景：

1）硬件升级，要换一台高配机器

2）网络重新规划，若干服务器要调整机架

3）服务器当机，要重新部署恢复服务

…



更具体的，如上图：数据库换了一个ip，此时往往连接此数据库的上游需要修改配置重启，如果数据库有很多上游调用方，改配置重启的调用方会很多，每次换ip的成本往往很高，成为大家共性的痛点。

由A的调整（数据库换ip），配合修改和调整的却是BCDE（改配置重启），BCDE内心非常的郁闷：明明换ip的是你，凭什么配合重启的却是我？

根本上，这是一个“架构耦合”的问题，是一个**架构设计上“反向依赖”的问题**，本文将讨论的是架构设计中常见的“反向依赖”的设计，以及对应的优化方案，希望对大伙有所启示。

**二、如何寻找不合理“反向依赖”**

**方法论**：

变动方是A，配合方却是BCDE

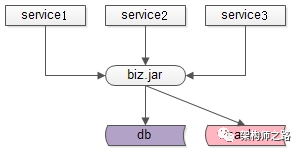
（或者说需求方是A，改动方确是BCDE）

想想“换IP的是你，配合重启的却是我”更好理解。

如果系统中经常出现了这类情况，就是“反向依赖”的特征，往往架构上有优化的空间。

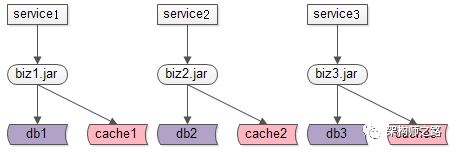
**三、常见的“反向依赖”与优化方案**

**【case1：公共库导致耦合】**



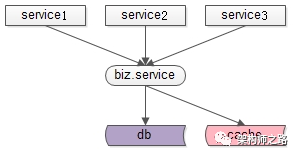
三个服务s1/s2/s3，通过一个公共的库biz.jar来实现一段业务逻辑，s1/s2/s3其实间接通过biz.jar耦合在了一起，一个业务s1修改一块公共的代码，导致影响其他业务s2/s3，架构上是不合理的。

**优化方案1：业务垂直拆分**



如果biz.jar中实现的逻辑“业务特性”很强，可以拆分为biz1.jar/biz2.jar/biz3.jar，来对s1/s2/s3进行解耦。这样的话，任何业务的改动，影响范围只是自己，不会影响其他人。

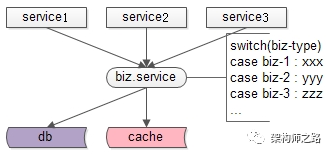
**优化方案2：服务化**



如果biz.jar中实现的逻辑“业务共性”很强，可以将biz.jar优化为biz.service服务，来对s1/s2/s3进行解耦。服务化之后，兼容性能更好的通过接口自动化回归测试来保证。

基础服务的抽象，本身是一种共性聚焦，是系统解耦常见的方案。

**【case2：服务化不彻底导致耦合】**



服务化是解决“业务共性”组件库导致系统耦合的常见方案之一，但如果服务化不彻底，service本身也容易成为业务耦合点。

典型的服务化不彻底导致的业务耦合的特征是，共性服务中，包含大量“根据不同业务，执行不同个性分支”的代码。

switch (biz-type)

case biz-1 : exec1

case biz-2 : exec2

case biz-3 : exec3

…

在这种架构下，biz-1/biz-2/biz-3有个性的业务需求，可能导致修改代码的是共性的biz-service，使其成为研发瓶颈，架构上也是不合理的。

**优化方案：业务特性代码上浮，业务共性代码下沉，彻底解耦**

把swithc case中业务特性代码放到业务层实现，这样biz-1/biz-2/biz-3有个性的业务需求，升级的是自己的业务系统。

**【case3：notify的不合理实现导致的耦合】**

《[究竟什么时候该使用MQ](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960012&idx=1&sn=c6af5c79ecead98daa4d742e5ad20ce5&chksm=bd2d07108a5a8e0624ae6ad95001c4efe09d7ba695f2ddb672064805d771f3f84bee8123b8a6&scene=21" \l "wechat_redirect" \t "_blank)》一文中有一类业务场景，消息发送方不关注消息接收方的执行结果，如果采用调用的方式来实现通知，会导消息发送方和消息接收方耦合。

如何新增消息接收方biz-4，会发现修改代码的是消息发送方，新增一个对biz-4的调用，极不合理。

**优化方案：通过MQ实现解耦**

消息发送方upper将消息发布给MQ，消息接收方从MQ去订阅，任何新增对消息的消费，upper都不需要修改代码。

**【case4：配置中的ip导致上下游耦合】**

即“缘起”中举的例子，下游服务换ip，可能导致多个服务调用方修改配置重启。上下游间接的通过ip这个配置耦合在了一起，架构不合理。

**优化方案：通过内网域名而不是ip来进行下游连接**

如果在配置中使用内网域名来进行下游连接，当下游服务或者数据库更换ip时，只需要**运维层面将内网域名指向新的ip，然后统一切断原有旧的连接**，连接就能够自动切换到新的ip上来。这个过程不需要所有上游配合，非常帅气，强烈推荐！

**【case5：下游扩容导致上下游耦合】**

这次不是换换ip这么简单了，下游服务提供方原来是集群（ip1/ip2/ip3，当然，上游配置的是内网域名），现在集群要扩容为（ip1/ip2/ip3/ip4/ip5），如果没有特殊的架构设计，上游往往需要修改配置，新增扩容后的节点，再重启，导致上下游耦合。

这类case，大伙有什么好的方案解耦么？

技术细节，且听下回分解。

**四、总结**

**如何发现系统架构中不合理的“反向依赖”设计？**

回答：

（1）变动方是A，配合方却是BCDE

（2）需求方是A，改动方确是BCDE

想想“换IP的是你，配合重启的却是我”，此时往往架构上可以进行解耦优化。

**常见反向依赖及优化方案？**

（1）公共库导致耦合

**优化一**：如果公共库是业务特性代码，进行公共库**垂直拆分**

**优化二**：如果公共库是业务共性代码，进行**服务化**下沉抽象

（2）服务化不彻底导致耦合

**特征**：服务中包含大量“根据不同业务，执行不同个性分支”的代码

**优化方案**：个性代码放到业务层实现，**将服务化更彻底更纯粹**

（3）notify的不合理实现导致的耦合

**特征**：调用方不关注执行结果，以调用的方式去实现通知，新增订阅者，修改代码的是发布者

**优化方案**：**通过MQ解耦**

（4）配置中的ip导致上下游耦合

**特征**：多个上游需要修改配置重启

**优化方案**：**使用内网域名替代内网ip**，通过“修改DNS指向，统一切断旧连接”的方式来上游无感切换

（5）下游扩容导致上下游耦合

特性：多个上游需要修改配置重启

换IP的是你，凭啥配合重启的却是我。

大伙碰到过类似的“反向依赖”耦合么？欢迎补充。

==【完】==

推荐阅读：

[到底什么时候该使用MQ？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960012&idx=1&sn=c6af5c79ecead98daa4d742e5ad20ce5&chksm=bd2d07108a5a8e0624ae6ad95001c4efe09d7ba695f2ddb672064805d771f3f84bee8123b8a6&scene=21" \l "wechat_redirect" \t "_blank)

[100亿数据平滑数据迁移,不影响服务](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959992&idx=1&sn=eb2fbd7d7922db42a593c304e50a65b7&chksm=bd2d07648a5a8e72d489022ec6006274d7e43ab48449b255d5661658c2af8e9221977a9609ed&scene=21" \l "wechat_redirect" \t "_blank)

[架构师之路16年精选50篇](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959886&idx=1&sn=03e45a5014053607eff5e55ed2c660d7&chksm=bd2d07928a5a8e8454d395e176fa9d346682abfe9dfbf3244f1dead83ee4508aa25121f9b811&scene=21" \l "wechat_redirect" \t "_blank)

## 究竟啥才是互联网架构“高可用”

原创 2016-12-05 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959728&idx=1&sn=933227840ec8cdc35d3a33ae3fe97ec5&chksm=bd2d046c8a5a8d7a13551124af36bedf68f7a6e31f6f32828678d2adb108b86b7e08c678f22f&scene=21" \l "#)

**一、什么是高可用**

**高可用HA（**High Availability）是分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计减少系统不能提供服务的时间。

假设系统一直能够提供服务，我们说系统的可用性是100%。

如果系统每运行100个时间单位，会有1个时间单位无法提供服务，我们说系统的可用性是99%。

很多公司的高可用目标是4个9，也就是99.99%，这就意味着，系统的年停机时间为8.76个小时。

百度的搜索首页，是业内公认高可用保障非常出色的系统，甚至人们会通过www.baidu.com 能不能访问来判断“网络的连通性”，百度高可用的服务让人留下啦“网络通畅，百度就能访问”，“百度打不开，应该是网络连不上”的印象，这其实是对百度HA最高的褒奖。

**二、如何保障系统的高可用**

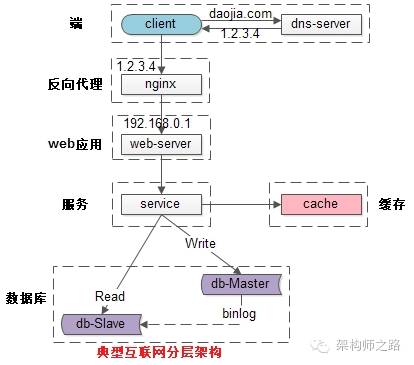
我们都知道，单点是系统高可用的大敌，单点往往是系统高可用最大的风险和敌人，应该尽量在系统设计的过程中避免单点。方法论上，高可用保证的原则是“集群化”，或者叫“冗余”：只有一个单点，挂了服务会受影响；如果有冗余备份，挂了还有其他backup能够顶上。

**保证系统高可用，架构设计的核心准则是：冗余。**

有了冗余之后，还不够，每次出现故障需要人工介入恢复势必会增加系统的不可服务实践。所以，又往往是通过“自动故障转移”来实现系统的高可用。

接下来我们看下典型互联网架构中，如何通过**冗余+自动故障转移**来保证系统的高可用特性。

**三、常见的互联网分层架构**



常见互联网分布式架构如上，分为：

（1）**客户端层**：典型调用方是浏览器browser或者手机应用APP

（2）**反向代理层**：系统入口，反向代理

（3）**站点应用层**：实现核心应用逻辑，返回html或者json

（4）**服务层**：如果实现了服务化，就有这一层

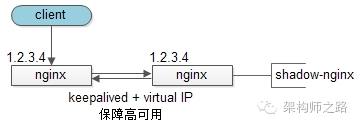
（5）**数据-缓存层**：缓存加速访问存储

（6）**数据-数据库层**：数据库固化数据存储

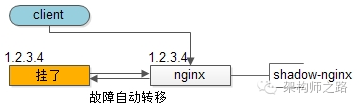
整个系统的高可用，又是通过每一层的**冗余+自动故障转移**来综合实现的。

**四、分层高可用架构实践**

**【客户端层->反向代理层】的高可用**

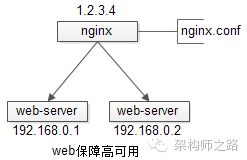


【客户端层】到【反向代理层】的**高可用**，是通过反向代理层的冗余来实现的。以nginx为例：有两台nginx，一台对线上提供服务，另一台冗余以保证高可用，常见的实践是keepalived存活探测，相同virtual IP提供服务。

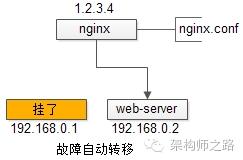


**自动故障转移**：当nginx挂了的时候，keepalived能够探测到，会自动的进行故障转移，将流量自动迁移到shadow-nginx，由于使用的是相同的virtual IP，这个切换过程对调用方是透明的。

**【反向代理层->站点层】的高可用**

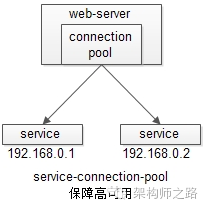


【反向代理层】到【站点层】的**高可用**，是通过站点层的冗余来实现的。假设反向代理层是nginx，nginx.conf里能够配置多个web后端，并且nginx能够探测到多个后端的存活性。

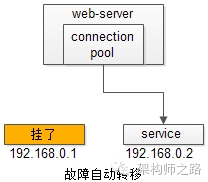


**自动故障转移**：当web-server挂了的时候，nginx能够探测到，会自动的进行故障转移，将流量自动迁移到其他的web-server，整个过程由nginx自动完成，对调用方是透明的。

**【站点层->服务层】的高可用**

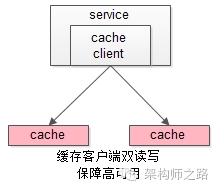


【站点层】到【服务层】的**高可用**，是通过服务层的冗余来实现的。“服务连接池”会建立与下游服务多个连接，每次请求会“随机”选取连接来访问下游服务。



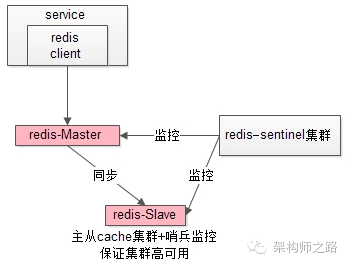
**自动故障转移**：当service挂了的时候，service-connection-pool能够探测到，会自动的进行故障转移，将流量自动迁移到其他的service，整个过程由连接池自动完成，对调用方是透明的（所以说RPC-client中的服务连接池是很重要的基础组件）。

**【服务层>缓存层】的高可用**



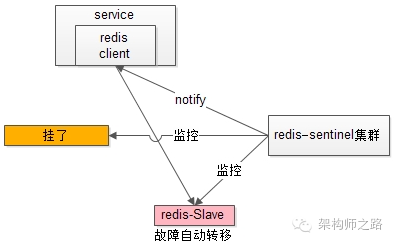
【服务层】到【缓存层】的高可用，是通过缓存数据的冗余来实现的。

缓存层的数据冗余又有几种方式：第一种是利用客户端的封装，service对cache进行双读或者双写。



缓存层也可以通过支持主从同步的缓存集群来解决缓存层的**高可用**问题。

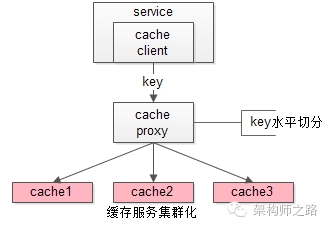
以redis为例，redis天然支持主从同步，redis官方也有sentinel哨兵机制，来做redis的存活性检测。



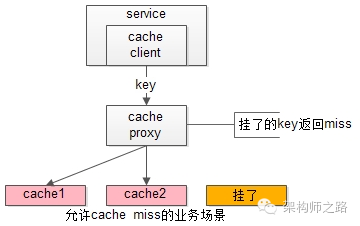
**自动故障转移**：当redis主挂了的时候，sentinel能够探测到，会通知调用方访问新的redis，整个过程由sentinel和redis集群配合完成，对调用方是透明的。

说完缓存的高可用，这里要多说一句，业务对缓存并不一定有“高可用”要求，更多的对缓存的使用场景，是用来“加速数据访问”：把一部分数据放到缓存里，如果缓存挂了或者缓存没有命中，是可以去后端的数据库中再取数据的。

这类允许“cache miss”的业务场景，缓存架构的建议是：



将kv缓存封装成服务集群，上游设置一个代理（代理可以用集群冗余的方式保证高可用），代理的后端根据缓存访问的key水平切分成若干个实例，每个实例的访问并不做高可用。

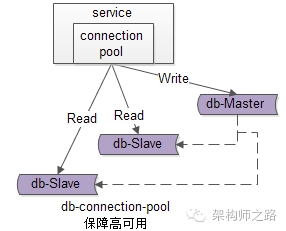


**缓存实例挂了屏蔽**：当有水平切分的实例挂掉时，代理层直接返回cache miss，此时缓存挂掉对调用方也是透明的。key水平切分实例减少，不建议做re-hash，这样容易引发缓存数据的不一致。

**【服务层>数据库层】的高可用**

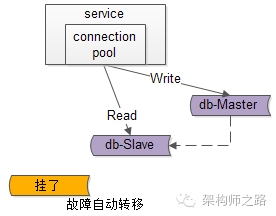
大部分互联网技术，数据库层都用了“主从同步，读写分离”架构，所以数据库层的高可用，又分为“读库高可用”与“写库高可用”两类。

**【服务层>数据库层“读”】的高可用**



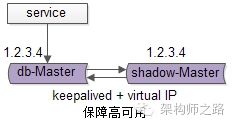
【服务层】到【数据库读】的**高可用**，是通过读库的冗余来实现的。

既然冗余了读库，一般来说就至少有2个从库，“数据库连接池”会建立与读库多个连接，每次请求会路由到这些读库。



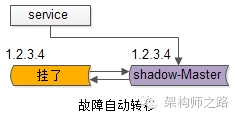
**自动故障转移**：当读库挂了的时候，db-connection-pool能够探测到，会自动的进行故障转移，将流量自动迁移到其他的读库，整个过程由连接池自动完成，对调用方是透明的（所以说DAO中的数据库连接池是很重要的基础组件）。

**【服务层>数据库层“写”】的高可用**



【服务层】到【数据库写】的**高可用**，是通过写库的冗余来实现的。

以mysql为例，可以设置两个mysql双主同步，一台对线上提供服务，另一台冗余以保证高可用，常见的实践是keepalived存活探测，相同virtual IP提供服务。



**自动故障转移**：当写库挂了的时候，keepalived能够探测到，会自动的进行故障转移，将流量自动迁移到shadow-db-master，由于使用的是相同的virtual IP，这个切换过程对调用方是透明的。

**五、总结**

高可用HA（High Availability）是分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计减少系统不能提供服务的时间。

方法论上，高可用是通过**冗余+自动故障转移**来实现的。

整个互联网分层系统架构的高可用，又是通过每一层的**冗余+自动故障转移**来综合实现的，具体的：

（1）【客户端层】到【反向代理层】的高可用，是通过反向代理层的冗余实现的，常见实践是keepalived + virtual IP自动故障转移

（2）【反向代理层】到【站点层】的高可用，是通过站点层的冗余实现的，常见实践是nginx与web-server之间的存活性探测与自动故障转移

（3）【站点层】到【服务层】的高可用，是通过服务层的冗余实现的，常见实践是通过service-connection-pool来保证自动故障转移

（4）【服务层】到【缓存层】的高可用，是通过缓存数据的冗余实现的，常见实践是缓存客户端双读双写，或者利用缓存集群的主从数据同步与sentinel保活与自动故障转移；更多的业务场景，对缓存没有高可用要求，可以使用缓存服务化来对调用方屏蔽底层复杂性

（5）【服务层】到【数据库“读”】的高可用，是通过读库的冗余实现的，常见实践是通过db-connection-pool来保证自动故障转移

（6）【服务层】到【数据库“写”】的高可用，是通过写库的冗余实现的，常见实践是keepalived + virtual IP自动故障转移

末了，希望文章的思路是清晰的，希望大家对高可用的概念和实践有个系统的认识，感谢大家。

==【完】==

## 究竟啥才是互联网架构“高并发”

原创 2017-01-11 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959830&idx=1&sn=ce1c5a58caed227d7dfdbc16d6e1cea4&chksm=bd2d07ca8a5a8edc45cc45c4787cc72cf4c8b96fb43d2840c7ccd44978036a7d39a03dd578b5&scene=21" \l "#)

**一、什么是高并发**

**高并发（**High Concurrency）是互联网分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计保证系统能够同时并行处理很多请求。

高并发相关常用的一些指标有响应时间（Response Time），吞吐量（Throughput），每秒查询率QPS（Query Per Second），并发用户数等。

**响应时间**：系统对请求做出响应的时间。例如系统处理一个HTTP请求需要200ms，这个200ms就是系统的响应时间。

**吞吐量**：单位时间内处理的请求数量。

**QPS**：每秒响应请求数。在互联网领域，这个指标和吞吐量区分的没有这么明显。

**并发用户数**：同时承载正常使用系统功能的用户数量。例如一个即时通讯系统，同时在线量一定程度上代表了系统的并发用户数。

**二、如何提升系统的并发能力**

互联网分布式架构设计，提高系统并发能力的方式，方法论上主要有两种：垂直扩展（Scale Up）与水平扩展（Scale Out）。

**垂直扩展**：提升单机处理能力。垂直扩展的方式又有两种：

（1）增强单机硬件性能，例如：增加CPU核数如32核，升级更好的网卡如万兆，升级更好的硬盘如SSD，扩充硬盘容量如2T，扩充系统内存如128G；

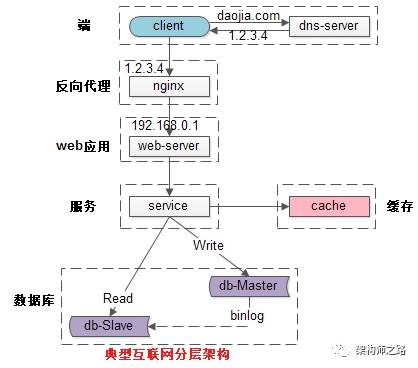
（2）提升单机架构性能，例如：使用Cache来减少IO次数，使用异步来增加单服务吞吐量，使用无锁数据结构来减少响应时间；

在互联网业务发展非常迅猛的早期，如果预算不是问题，强烈建议使用“增强单机硬件性能”的方式提升系统并发能力，因为这个阶段，公司的战略往往是发展业务抢时间，而“增强单机硬件性能”往往是最快的方法。

不管是提升单机硬件性能，还是提升单机架构性能，都有一个致命的不足：单机性能总是有极限的。所以互联网分布式架构设计高并发终极解决方案还是水平扩展。

**水平扩展**：只要增加服务器数量，就能线性扩充系统性能。水平扩展对系统架构设计是有要求的，如何在架构各层进行可水平扩展的设计，以及互联网公司架构各层常见的水平扩展实践，是本文重点讨论的内容。

**三、常见的互联网分层架构**



常见互联网分布式架构如上，分为：

（1）**客户端层**：典型调用方是浏览器browser或者手机应用APP

（2）**反向代理层**：系统入口，反向代理

（3）**站点应用层**：实现核心应用逻辑，返回html或者json

（4）**服务层**：如果实现了服务化，就有这一层

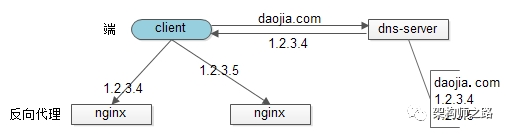
（5）**数据-缓存层**：缓存加速访问存储

（6）**数据-数据库层**：数据库固化数据存储

整个系统各层次的水平扩展，又分别是如何实施的呢？

**四、分层水平扩展架构实践**

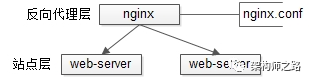
**反向代理层的水平扩展**



反向代理层的水平扩展，是通过“DNS轮询”实现的：dns-server对于一个域名配置了多个解析ip，每次DNS解析请求来访问dns-server，会轮询返回这些ip。

当nginx成为瓶颈的时候，只要增加服务器数量，新增nginx服务的部署，增加一个外网ip，就能扩展反向代理层的性能，做到理论上的无限高并发。

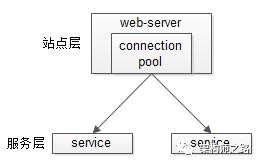
**站点层的水平扩展**



站点层的水平扩展，是通过“nginx”实现的。通过修改nginx.conf，可以设置多个web后端。

当web后端成为瓶颈的时候，只要增加服务器数量，新增web服务的部署，在nginx配置中配置上新的web后端，就能扩展站点层的性能，做到理论上的无限高并发。

**服务层的水平扩展**



服务层的水平扩展，是通过“服务连接池”实现的。

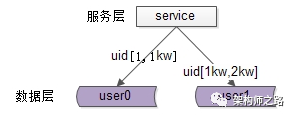
站点层通过RPC-client调用下游的服务层RPC-server时，RPC-client中的连接池会建立与下游服务多个连接，当服务成为瓶颈的时候，只要增加服务器数量，新增服务部署，在RPC-client处建立新的下游服务连接，就能扩展服务层性能，做到理论上的无限高并发。如果需要优雅的进行服务层自动扩容，这里可能需要配置中心里服务自动发现功能的支持。

**数据层的水平扩展**

在数据量很大的情况下，数据层（缓存，数据库）涉及数据的水平扩展，将原本存储在一台服务器上的数据（缓存，数据库）水平拆分到不同服务器上去，以达到扩充系统性能的目的。

互联网数据层常见的水平拆分方式有这么几种，以数据库为例：

**按照范围水平拆分**



每一个数据服务，存储一定范围的数据，上图为例：

user0库，存储uid范围1-1kw

user1库，存储uid范围1kw-2kw

这个方案的好处是：

（1）规则简单，service只需判断一下uid范围就能路由到对应的存储服务；

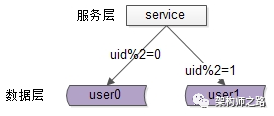
（2）数据均衡性较好；

（3）比较容易扩展，可以随时加一个uid[2kw,3kw]的数据服务；

不足是：

（1）      请求的负载不一定均衡，一般来说，新注册的用户会比老用户更活跃，大range的服务请求压力会更大；

**按照哈希水平拆分**



每一个数据库，存储某个key值hash后的部分数据，上图为例：

user0库，存储偶数uid数据

user1库，存储奇数uid数据

这个方案的好处是：

（1）规则简单，service只需对uid进行hash能路由到对应的存储服务；

（2）数据均衡性较好；

（3）请求均匀性较好；

不足是：

（1）不容易扩展，扩展一个数据服务，hash方法改变时候，可能需要进行数据迁移；

这里需要注意的是，通过水平拆分来扩充系统性能，与主从同步读写分离来扩充数据库性能的方式有本质的不同。

通过水平拆分扩展数据库性能：

（1）每个服务器上存储的数据量是总量的1/n，所以单机的性能也会有提升；

（2）n个服务器上的数据没有交集，那个服务器上数据的并集是数据的全集；

（3）数据水平拆分到了n个服务器上，理论上读性能扩充了n倍，写性能也扩充了n倍（其实远不止n倍，因为单机的数据量变为了原来的1/n）；

通过主从同步读写分离扩展数据库性能：

（1）每个服务器上存储的数据量是和总量相同；

（2）n个服务器上的数据都一样，都是全集；

（3）理论上读性能扩充了n倍，写仍然是单点，写性能不变；

缓存层的水平拆分和数据库层的水平拆分类似，也是以范围拆分和哈希拆分的方式居多，就不再展开。

**五、总结**

**高并发（**High Concurrency）是互联网分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计保证系统能够同时并行处理很多请求。

提高系统并发能力的方式，方法论上主要有两种：垂直扩展（Scale Up）与水平扩展（Scale Out）。前者垂直扩展可以通过提升单机硬件性能，或者提升单机架构性能，来提高并发性，但单机性能总是有极限的，互联网分布式架构设计高并发终极解决方案还是后者：水平扩展。

互联网分层架构中，各层次水平扩展的实践又有所不同：

（1）反向代理层可以通过“DNS轮询”的方式来进行水平扩展；

（2）站点层可以通过nginx来进行水平扩展；

（3）服务层可以通过服务连接池来进行水平扩展；

（4）数据库可以按照数据范围，或者数据哈希的方式来进行水平扩展；

各层实施水平扩展后，能够通过增加服务器数量的方式来提升系统的性能，做到理论上的性能无限。

末了，希望文章的思路是清晰的，希望大家对高并发的概念和实践有个系统的认识，结合上一篇《[究竟啥才是互联网架构“高可用”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959728&idx=1&sn=933227840ec8cdc35d3a33ae3fe97ec5&chksm=bd2d046c8a5a8d7a13551124af36bedf68f7a6e31f6f32828678d2adb108b86b7e08c678f22f&scene=21" \l "wechat_redirect" \t "_blank)》的分享互联网分布式架构是不是逐步的不再神秘啦？

==【完】==

相关阅读：[究竟啥才是互联网架构“高可用”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959728&idx=1&sn=933227840ec8cdc35d3a33ae3fe97ec5&chksm=bd2d046c8a5a8d7a13551124af36bedf68f7a6e31f6f32828678d2adb108b86b7e08c678f22f&scene=21" \l "wechat_redirect" \t "_blank)

## 典型数据库架构设计与实践

原创 2017-05-31 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960158&idx=1&sn=019e9dca6e074f62478b2562044cf8de&chksm=bd2d06828a5a8f945e1e6c2aa4702f48b7857ac9a79acf867496f25b28c86e2f28bed19b38e4&scene=21" \l "#)

本文，将介绍数据库架构设计中的一些基本概念，常见问题以及对应解决方案，为了便于读者理解，将以“用户中心”数据库为例，讲解数据库架构设计的常见玩法。

**一、用户中心**

**用户中心**是一个常见业务，主要提供用户注册、登录、信息查询与修改的服务，其核心元数据为：

User(uid, uname, passwd, sex, age,nickname, …)

其中：

* uid为用户ID，主键
* uname, passwd, sex, age, nickname, …等为用户的属性

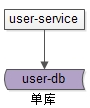
数据库设计上，一般来说在业务初期，单库单表就能够搞定这个需求。

**二、图示说明**

为了方便大家理解，后文图片说明较多，其中：

* “灰色”方框，表示service，服务
* “紫色”圆框，标识master，主库
* “粉色”圆框，表示slave，从库

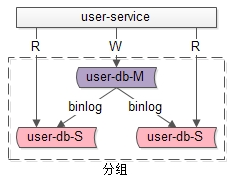
**三、单库架构**



最常见的架构设计如上：

* user-service：用户中心服务，对调用者提供友好的RPC接口
* user-db：一个库进行数据存储

**四、分组架构**



**什么是分组？**

**答**：分组架构是最常见的一主多从，主从同步，读写分离数据库架构：

* user-service：依旧是用户中心服务
* user-db-M(master)：主库，提供数据库写服务
* user-db-S(slave)：从库，提供数据库读服务

主和从构成的数据库集群称为“组”。

**分组有什么特点？**

**答**：同一个组里的数据库集群：

* 主从之间通过binlog进行数据同步
* 多个实例数据库结构完全相同
* 多个实例存储的数据也完全相同，本质上是将数据进行复制

**分组架构究竟解决什么问题？**

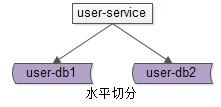
**答**：大部分互联网业务读多写少，**数据库的读往往最先成为性能瓶颈**，如果希望：

* 线性提升数据库读性能
* 通过消除读写锁冲突提升数据库写性能
* 通过冗余从库实现数据的“读高可用”

此时可以使用分组架构，需要注意的是，分组架构中，数据库的主库依然是写单点。

一句话总结，**分组解决的是“数据库读写高并发量高”问题**，所实施的架构设计。

**五、分片架构**



**什么是分片？**

**答**：分片架构是大伙常说的水平切分(sharding)数据库架构：

* user-service：依旧是用户中心服务
* user-db1：水平切分成2份中的第一份
* user-db2：水平切分成2份中的第二份

分片后，多个数据库实例也会构成一个数据库集群。

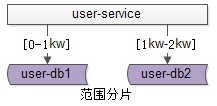
**水平切分，到底是分库还是分表？**

**答**：强烈建议分库，而不是分表，因为：

* 分表依然公用一个数据库文件，仍然有磁盘IO的竞争
* 分库能够很容易的将数据迁移到不同数据库实例，甚至数据库机器上，扩展性更好

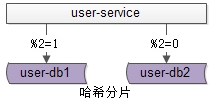
**水平切分，用什么算法？**

**答**：常见的水平切分算法有“范围法”和“哈希法”：



**范围法**如上图：以用户中心的业务主键uid为划分依据，将数据水平切分到两个数据库实例上去：

* user-db1：存储0到1千万的uid数据
* user-db2：存储0到2千万的uid数据



**哈希法**如上图：也是以用户中心的业务主键uid为划分依据，将数据水平切分到两个数据库实例上去：

* user-db1：存储uid取模得1的uid数据
* user-db2：存储uid取模得0的uid数据

这两种方法在互联网都有使用，其中哈希法使用较为广泛。

**分片有什么特点？**

**答**：同一个分片里的数据库集群：

* 多个实例之间本身不直接产生联系，不像主从间有binlog同步
* 多个实例数据库结构，也完全相同
* 多个实例存储的数据之间没有交集，所有实例间数据并集构成全局数据

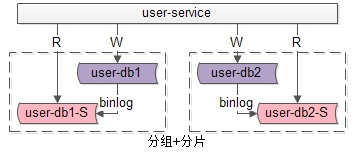
**分片架构究竟解决什么问题？**

**答**：大部分互联网业务数据量很大，单库容量容易成为瓶颈，此时通过分片可以：

* 线性提升数据库写性能，需要注意的是，分组架构是不能线性提升数据库写性能的
* 降低单库数据容量

一句话总结，**分片解决的是“数据库数据量大”问题**，所实施的架构设计。

**六、分组+分片架构**

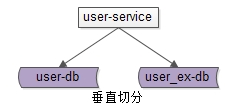


如果业务**读写并发量很高，数据量也很大**，通常需要实施分组+分片的数据库架构：

* 通过分片来降低单库的数据量，线性提升数据库的写性能
* 通过分组来线性提升数据库的读性能，保证读库的高可用

**七、垂直切分**

除了水平切分，垂直切分也是一类常见的数据库架构设计，垂直切分一般和业务结合比较紧密。



还是以用户中心为例，可以这么进行垂直切分：

User(uid, uname, passwd, sex, age, …)

User\_EX(uid, intro, sign, …)

* 垂直切分开的表，主键都是uid
* 登录名，密码，性别，年龄等属性放在一个垂直表（库）里
* 自我介绍，个人签名等属性放在另一个垂直表（库）里

**如何进行垂直切分？**

**答**：根据业务对数据进行垂直切分时，一般要考虑属性的“长度”和“访问频度”两个因素：

* 长度较短，访问频率较高的放在一起
* 长度较长，访问频度较低的放在一起

这是因为，数据库会以行(row)为单位，将数load到内存(buffer)里，在内存容量有限的情况下，长度短且访问频度高的属性，内存能够load更多的数据，命中率会更高，磁盘IO会减少，数据库的性能会提升。

**垂直切分有什么特点？**

**答**：垂直切分和水平切有相似的地方，又不太相同：

* 多个实例之间也不直接产生联系，即没有binlog同步
* 多个实例数据库结构，都不一样
* 多个实例存储的数据之间至少有一列交集，一般来说是业务主键，所有实例间数据并集构成全局数据

**垂直切分解决什么问题？**

**答**：垂直切分即可以降低单库的数据量，还可以降低磁盘IO从而提升吞吐量，但它与业务结合比较紧密，并不是所有业务都能够进行垂直切分的。

**八、总结**

文章较长，希望至少记住这么几点：

* 业务初期用单库
* 读压力大，读高可用，用分组
* 数据量大，写线性扩容，用分片
* 属性短，访问频度高的属性，垂直拆分到一起

希望大伙有收获。

==

相关文章：

《[100亿数据量1万属性10万并发数据库架构设计](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959855&idx=1&sn=f33abe8ec598c273f29cebb9365ece59&chksm=bd2d07f38a5a8ee58a944507a134e1da1efc3ac9c4d1c4cff261137cd986e51f5fe7cee9de15&scene=21" \l "wechat_redirect" \t "_blank)》

《[数据库秒级平滑扩容架构方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959883&idx=1&sn=e7df8510c7096a5b069e0f12eaaca010&chksm=bd2d07978a5a8e815c2ae41b16b6b4c579923502fb919008a22bb108a1e920109f25387f8903&scene=21" \l "wechat_redirect" \t "_blank)》

《[58到家数据库30条军规](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959906&idx=1&sn=2cbdc66cfb5b53cf4327a1e0d18d9b4a&chksm=bd2d07be8a5a8ea86dc3c04eced3f411ee5ec207f73d317245e1fefea1628feb037ad71531bc&scene=21" \l "wechat_redirect" \t "_blank)》

《[业界难题-“夸库分页”的四种方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959942&idx=1&sn=e9d3fe111b8a1d44335f798bbb6b9eea&chksm=bd2d075a8a5a8e4cad985b847778aa83056e22931767bb835132c04571b66d5434020fd4147f&scene=21" \l "wechat_redirect" \t "_blank)》

《[100亿数据平滑迁移却不影响服务](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959992&idx=1&sn=eb2fbd7d7922db42a593c304e50a65b7&chksm=bd2d07648a5a8e72d489022ec6006274d7e43ab48449b255d5661658c2af8e9221977a9609ed&scene=21" \l "wechat_redirect" \t "_blank)》

## 计数系统架构实践一次搞定

原创 2017-06-08 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960185&idx=1&sn=0acd4a563c8c9684373fd05c116c4441&chksm=bd2d06a58a5a8fb3281acd95a7d9494161e75dcd27e95fdd526fce2b031ba79c4153bf255cee&scene=21" \l "#)

提醒，本文较长，可提前收藏/转发。

**一、需求缘起**

很多业务都有“计数”需求，以微博为例：



微博首页的**个人中心**部分，有三个重要的计数：

* 关注了多少人的计数
* 粉丝的计数
* 发布博文的计数

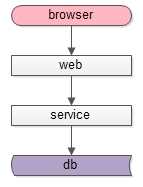


微博首页的**博文消息主体**部分，也有有很多计数，分别是一条博文的：

* 转发计数
* 评论计数
* 点赞计数
* 甚至是浏览计数

在业务复杂，计数扩展频繁，数据量大，并发量大的情况下，**计数系统的架构演进与实践**，是本文将要讨论的问题。

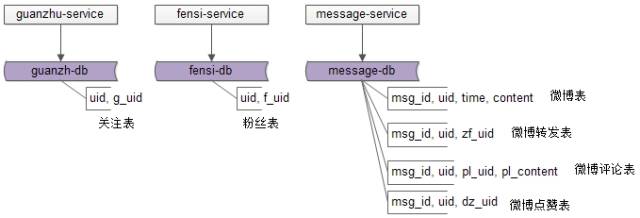
**二、业务分析与计数初步实现**



典型的互联网架构，常常分为这么几层：

* 调用层：处于端上的browser或者APP
* 站点层：拼装html或者json返回的web-server层
* 服务层：提供RPC调用接口的service层
* 数据层：提供固化数据存储的db，以及加速存储的cache

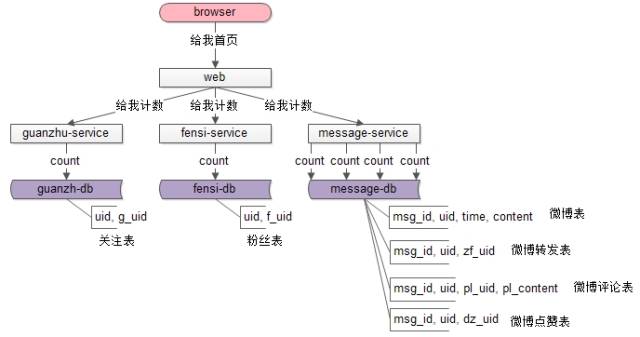
针对“缘起”里微博计数的例子，主要涉及“关注”业务，“粉丝”业务，“微博消息”业务，一般来说，会有相应的db存储相关数据，相应的service提供相关业务的RPC接口：



* 关注服务：提供关注数据的增删查改RPC接口
* 粉丝服务：提供粉丝数据的增删查改RPC接口
* 消息服务：提供微博消息数据的增删查改RPC接口，消息业务相对比较复杂，涉及微博消息、转发、评论、点赞等数据的存储

对关注、粉丝、微博业务进行了初步解析，**那首页的计数需求应该如何满足呢？**

很容易想到，关注服务+粉丝服务+消息服务均提供相应接口，就能拿到相关计数数据。

例如，个人中心首页，需要展现博文数量这个计数，web层访问message-service的count接口，这个接口执行：

select count(\*) from t\_msg where uid = XXX



同理，也很容易拿到关注，粉丝的这些计数。

这个方案叫做**“count”计数法**，在数据量并发量不大的情况下，最容易想到且最经常使用的就是这种方法，但随着数据量的上升，并发量的上升，这个方法的弊端将逐步展现。

例如，微博首页有很多条微博消息，每条消息有若干计数，此时计数的拉取就成了一个庞大的工程：



整个拉取计数的伪代码如下：

list<msg\_id> = getHomePageMsg(uid);// 获取首页所有消息

for( msg\_id in list<msg\_id>){ // 对每一条消息

         getReadCount(msg\_id);  // 阅读计数

         getForwordCount(msg\_id); // 转发计数

         getCommentCount(msg\_id); // 评论计数

         getPraiseCount(msg\_id); // 赞计数

}

其中：

* 每一个微博消息的若干个计数，都对应4个后端服务访问
* 每一个访问，对应一条count的数据库访问（count要了老命了）

其效率之低，资源消耗之大，处理时间之长，可想而知。

**“count”计数法**方案，可以总结为：

* 多条消息多次查询，for循环进行
* 一条消息多次查询，多个计数的查询
* 一次查询一个count，每个计数都是一个count语句

那如何进行优化呢？

**三、计数外置的架构设计**

计数是一个通用的需求，有没有可能，这个计数的需求实现在一个通用的系统里，而不是由关注服务、粉丝服务、微博服务来分别来提供相应的功能呢（否则扩展性极差）？

**这样需要实现一个通用的计数服务。**

通过分析，上述微博的业务可以抽象成两类：

* **用户(uid)维度**的计数：用户的关注计数，粉丝计数，发布的微博计数
* **微博消息(msg\_id)维度**的计数：消息转发计数，评论计数，点赞计数

于是可以抽象出两个表，针对这两个维度来进行计数的存储：

**t\_user\_count** (uid, gz\_count, fs\_count, wb\_count);

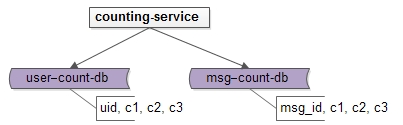
**t\_msg\_count** (msg\_id, forword\_count, comment\_count, praise\_count);

甚至可以更为抽象，一个表搞定所有计数：

t\_count(id, type, c1, c2, c3, …)

通过type来判断，id究竟是uid还是msg\_id，但并不建议这么做。

存储抽象完，再抽象出一个计数服务对这些数据进行管理，提供友善的RPC接口：



这样，在查询一条微博消息的若干个计数的时候，不用进行多次数据库count操作，而会转变为一条数据的多个属性的查询：

for(msg\_id in list<msg\_id>) {

select forword\_count, comment\_count, praise\_count

    from t\_msg\_count

    where msg\_id=$msg\_id;

}

甚至，可以将微博首页所有消息的计数，转变为一条IN语句（不用多次查询了）的批量查询：

select \* from t\_msg\_count

    where msg\_id IN

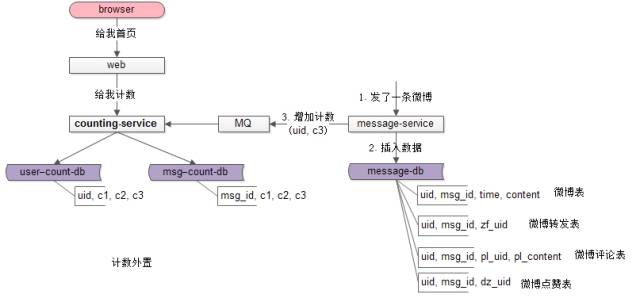
    ($msg\_id1, $msg\_id2, $msg\_id3, …);

**IN查询可以命中msg\_id聚集索引**，效率很高。

方案非常帅气，接下来，问题转化为：**当有微博被转发、评论、点赞的时候，计数服务如何同步的进行计数的变更呢？**

如果让业务服务来调用计数服务，势必会导致业务系统与计数系统耦合。

之前的文章介绍过，对于不关心下游结果的业务，可以使用MQ来解耦（具体请查阅《[到底什么时候该使用MQ？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960012&idx=1&sn=c6af5c79ecead98daa4d742e5ad20ce5&chksm=bd2d07108a5a8e0624ae6ad95001c4efe09d7ba695f2ddb672064805d771f3f84bee8123b8a6&scene=21" \l "wechat_redirect" \t "_blank)》），在业务发生变化的时候，向MQ发送一条异步消息，通知计数系统计数发生了变化即可：



如上图：

* 用户新发布了一条微博
* msg-service向MQ发送一条消息
* counting-service从MQ接收消息
* counting-service变更这个uid发布微博消息计数

这个方案称为**“计数外置”**，可以总结为：

* 通过counting-service单独保存计数
* MQ同步计数的变更
* 多条消息的多个计数，一个批量IN查询完成

**计数外置**，**本质是数据的冗余**，架构设计上，数据冗余必将引发数据的一致性问题，需要有机制来保证计数系统里的数据与业务系统里的数据一致，常见的方法有：

* 对于一致性要求比较高的业务，要有定期check并fix的机制，例如关注计数，粉丝计数，微博消息计数等
* 对于一致性要求比较低的业务，即使有数据不一致，业务可以接受，例如微博浏览数，微博转发数等

**四、计数外置缓存优化**

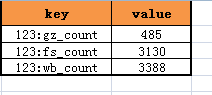
**计数外置很大程度上解决了计数存取的性能问题，但是否还有优化空间呢？**

像关注计数，粉丝计数，微博消息计数，变化的频率很低，查询的频率很高，这类读多些少的业务场景，非常适合使用**缓存**来进行查询优化，减少数据库的查询次数，降低数据库的压力。

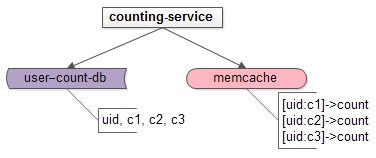
但是，缓存是kv结构的，无法像数据库一样，设置成t\_uid\_count(uid, c1, c2, c3)这样的schema，**如何来对kv进行设计呢？**

缓存kv结构的value是计数，看来只能在key上做设计，很容易想到，可以使用uid:type来做key，存储对应type的计数。

对于uid=123的用户，其关注计数，粉丝计数，微博消息计数的缓存就可以设计为：



此时对应的counting-service架构变为：



如此这般，多个uid的多个计数，又可能会变为多次缓存的访问：

for(uid in list<uid>) {

 memcache::get($uid:c1, $uid:c2, $uid:c3);

}

这个**“计数外置缓存优化”方案**，可以总结为：

* 使用缓存来保存读多写少的计数（其实写多读少，一致性要求不高的计数，也可以先用缓存保存，然后定期刷到数据库中，以降低数据库的读写压力）
* 使用id:type的方式作为缓存的key，使用count来作为缓存的value
* 多次读取缓存来查询多个uid的计数

**五、缓存批量读取优化**

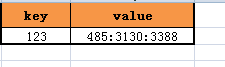
缓存的使用能够极大降低数据库的压力，**但多次缓存交互依旧存在优化空间，有没有办法进一步优化呢？**

当当当当！

不要陷入思维定式，谁说value一定只能是一个计数，难道不能多个计数存储在一个value中么？

缓存kv结构的key是uid，value可以是多个计数同时存储。

对于uid=123的用户，其关注计数，粉丝计数，微博消息计数的缓存就可以设计为：



这样多个用户，多个计数的查询就可以一次搞定：

memcache::get($uid1, $uid2, $uid3, …);

然后对获取的value进行分析，得到关注计数，粉丝计数，微博计数。

如果计数value能够事先预估一个范围，甚至可以用一个整数的不同bit来存储多个计数，用整数的与或非计算提高效率。

这个**“计数外置缓存批量优化”方案**，可以总结为：

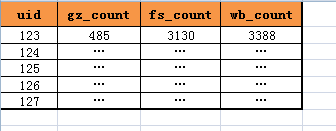
* 使用id作为key，使用同一个id的多个计数的拼接作为value
* 多个id的多个计数查询，一次搞定

**六、计数扩展性优化**

考虑完效率，架构设计上还需要考虑**扩展性**，如果uid除了关注计数，粉丝计数，微博计数，还要**增加一个计数，这时系统需要做什么变更呢？**

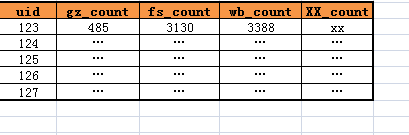
之前的数据库结构是：

t\_user\_count(uid, gz\_count, fs\_count, wb\_count)



这种设计，通过列来进行计数的存储，如果增加一个XX计数，数据库的表结构要变更为：

t\_user\_count(uid, gz\_count, fs\_count, wb\_count, XX\_count)

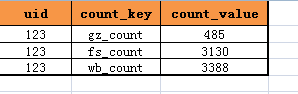


在数据量很大的情况下，**频繁的变更数据库schema的结构显然是不可取的**，有没有扩展性更好的方式呢？

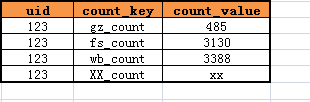
当当当当！

不要陷入思维定式，谁说只能通过扩展列来扩展属性，通过扩展行来扩展属性，在“架构师之路”的系列文章里也不是第一次出现了（具体请查阅《[啥，又要为表增加一列属性？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959762&idx=1&sn=f7d8d7648416a0157af6ac61a6b555c8&chksm=bd2d040e8a5a8d181ec1baa2e96982991ddfc218bb6838f38da61b651e4b8de5f0bbbd94f814&scene=21" \l "wechat_redirect" \t "_blank)》《[这才是真正的表扩展方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959765&idx=1&sn=b9916aa95c320e41035977e0a8098ca6&chksm=bd2d04098a5a8d1f3af38f658c05002151e621170949d2e3bb5b1bceea55c64b0477dba4c647&scene=21" \l "wechat_redirect" \t "_blank)》《[100亿数据1万属性数据架构设计](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959855&idx=1&sn=f33abe8ec598c273f29cebb9365ece59&chksm=bd2d07f38a5a8ee58a944507a134e1da1efc3ac9c4d1c4cff261137cd986e51f5fe7cee9de15&scene=21" \l "wechat_redirect" \t "_blank)》），完全可以这样设计表结构：

t\_user\_count(uid, count\_key, count\_value)



如果需要新增一个计数XX\_count，只需要增加一行即可，而不需要变更表结构：

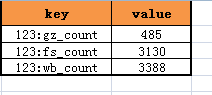


**七、总结**

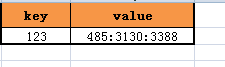
小小的计数，在数据量大，并发量大的时候，其架构实践思路为：

* **计数外置**：由“count计数法”升级为“计数外置法”
* 读多写少，甚至写多但一致性要求不高的计数，需要进行**缓存优化**，降低数据库压力
* **缓存kv设计优化**，可以由[key:type]->[count]，优化为[key]->[c1:c2:c3]

即：

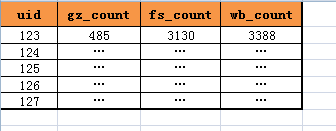


优化为：

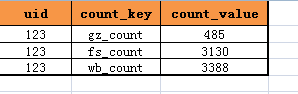


* **数据库扩展性优化**，可以由列扩展优化为行扩展

即：



优化为：



计数系统架构先聊到这里，希望大家有收获。

===【完】===

相关阅读：

[到底什么时候该使用MQ？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960012&idx=1&sn=c6af5c79ecead98daa4d742e5ad20ce5&chksm=bd2d07108a5a8e0624ae6ad95001c4efe09d7ba695f2ddb672064805d771f3f84bee8123b8a6&scene=21" \l "wechat_redirect" \t "_blank)

[啥，又要为表增加一列属性？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959762&idx=1&sn=f7d8d7648416a0157af6ac61a6b555c8&chksm=bd2d040e8a5a8d181ec1baa2e96982991ddfc218bb6838f38da61b651e4b8de5f0bbbd94f814&scene=21" \l "wechat_redirect" \t "_blank)

[这才是真正的表扩展方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959765&idx=1&sn=b9916aa95c320e41035977e0a8098ca6&chksm=bd2d04098a5a8d1f3af38f658c05002151e621170949d2e3bb5b1bceea55c64b0477dba4c647&scene=21" \l "wechat_redirect" \t "_blank)

[100亿数据1万属性数据架构设计](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959855&idx=1&sn=f33abe8ec598c273f29cebb9365ece59&chksm=bd2d07f38a5a8ee58a944507a134e1da1efc3ac9c4d1c4cff261137cd986e51f5fe7cee9de15&scene=21" \l "wechat_redirect" \t "_blank)

## session一致性架构设计实践

原创 2017-05-18 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960128&idx=1&sn=8e0e409b10ab9db549432af461385314&chksm=bd2d069c8a5a8f8ab5cdee602d4062bbdbb25da290668515d36682afa854e374d2a5ff02004b&scene=21" \l "#)

**一、缘起**

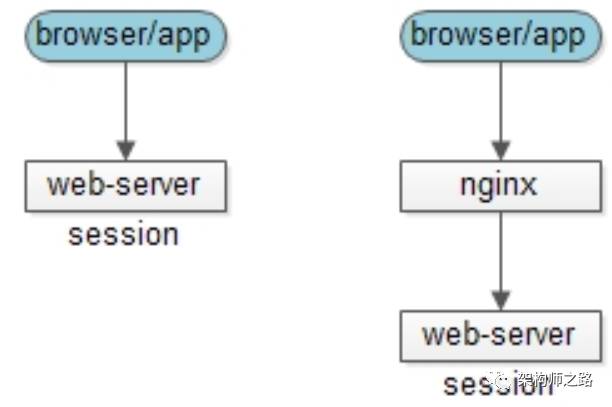
**什么是session？**

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。

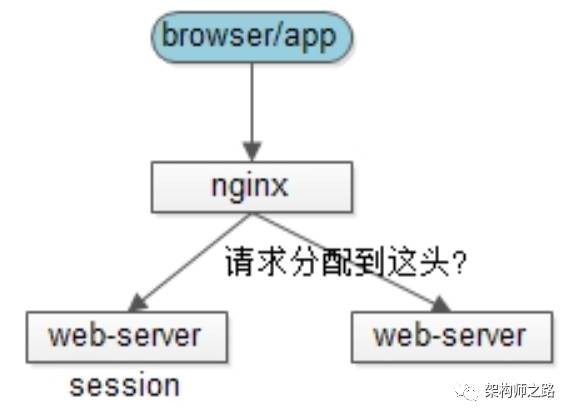
**什么是session一致性问题？**

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（废话，因为只有一台）。

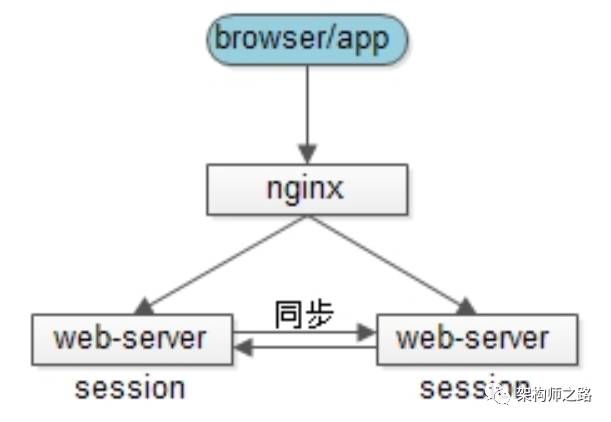
此时的web-server是无法保证高可用的，采用“冗余+故障转移”的多台web-server来保证高可用时，每次http短连接请求就不一定能路由到正确的session了。



如上图，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。

在web-server高可用时，如何保证session路由的一致性，是今天将要讨论的问题。

**二、session同步法**



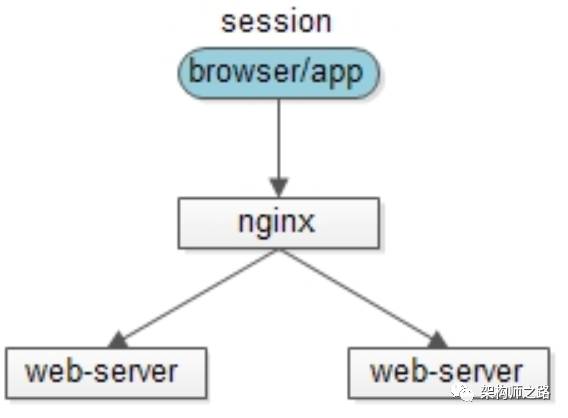
**思路**：多个web-server之间相互同步session，这样每个web-server之间都包含全部的session

**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

**三、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

**优点**：服务端不需要存储

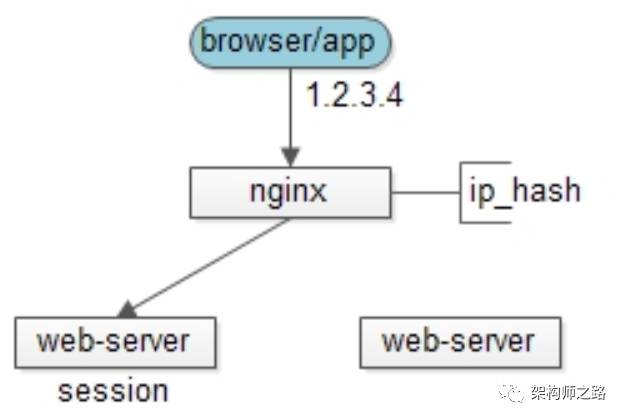
**缺点**：

* 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

“端存储”的方案虽然不常用，但确实是一种思路。

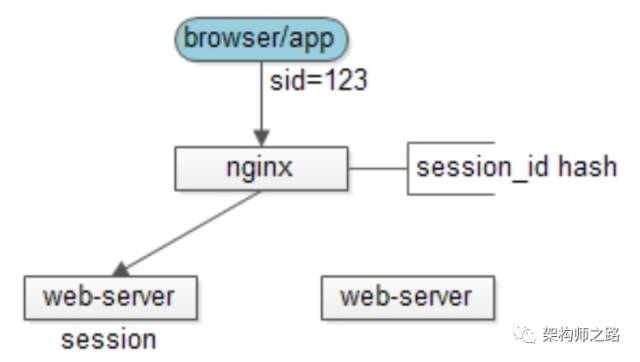
**三、反向代理hash一致性**

**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上呢？



**方案一：四层代理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



**方案二：七层代理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）

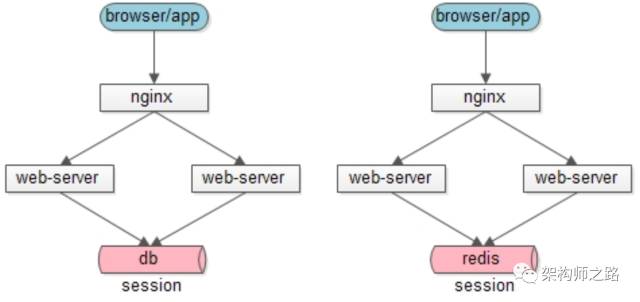
**不足**：

* 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。

对于四层hash还是七层hash，个人推荐前者：**让专业的软件做专业的事情**，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（例如，有时候多机房多活需要按照业务属性路由到不同机房的web-server）。

**四、后端统一存储**



**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

对于db存储还是cache，个人推荐后者：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。

**五、总结**

**保证session一致性**的架构设计常见方法：

* session同步法：多台web-server相互同步数据
* 客户端存储法：一个用户只存储自己的数据
* 反向代理hash一致性：四层hash和七层hash都可以做，保证一个用户的请求落在一台web-server上
* 后端统一存储：web-server重启和扩容，session也不会丢失

对于方案3和方案4，个人建议推荐后者：

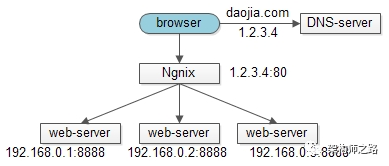
* **web层、service层无状态是大规模分布式系统设计原则之一**，session属于状态，不宜放在web层
* **让专业的软件做专业的事情**，web-server存session？还是让cache去做这样的事情吧

希望大伙有收获，思路比结果重要，帮转哟。

## TCP接入层的负载均衡、高可用、扩展性架构

原创 2017-05-07 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960086&idx=1&sn=70bbe7165ecddc7896767f4503a927fe&chksm=bd2d06ca8a5a8fdc67fcacb169583f53a968fdf623a20395926059d44e6abae6b2e56ff1f9f9&scene=21" \l "#)

**一、web-server的负载均衡**



互联网架构中，web-server接入一般使用nginx来做反向代理，实施负载均衡。整个架构分三层：

* **上游调用层**，一般是browser或者APP
* **中间反向代理层**，nginx
* **下游真实接入集群**，web-server，常见web-server的有tomcat，apache

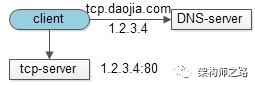
整个访问过程为：

* browser向daojia.com发起请求
* DNS服务器将daojia.com解析为外网IP(1.2.3.4)
* browser通过外网IP(1.2.3.4)访问nginx
* nginx实施负载均衡策略，常见策略有轮询，随机，IP-hash等
* nginx将请求转发给内网IP(192.168.0.1)的web-server

由于http短连接，以及web应用无状态的特性，理论上**任何一个http请求落在任意一台web-server都应该得到正常处理**（如果必须落在一台，说明架构不合理，不能水平扩展）。

问题来了，tcp是有状态的连接，客户端和服务端一旦建立连接，**一个client发起的请求必须落在同一台tcp-server上**，此时如何做负载均衡，如何保证水平扩展呢？

**二、单机法tcp-server**



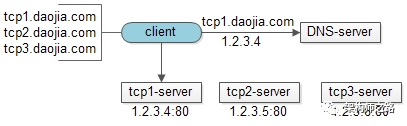
单个tcp-server显然是可以保证请求一致性：

* client向tcp.daojia.com发起tcp请求
* DNS服务器将tcp.daojia.com解析为外网IP(1.2.3.4)
* client通过外网IP(1.2.3.4)向tcp-server发起请求

**方案的缺点？**

无法保证高可用。

**三、集群法tcp-server**



通过搭建tcp-server集群来保证高可用，**客户端来实现负载均衡**：

* client内配置有tcp1/tcp2/tcp3.daojia.com三个tcp-server的外网IP
* 客户端通过“随机”的方式选择tcp-server，假设选择到的是tcp1.daojia.com
* 通过DNS解析tcp1.daojia.com
* 通过外网IP连接真实的tcp-server

**如何保证高可用呢？**

如果client发现某个tcp-server连接不上，则选择另一个。

**潜在的缺点？**

每次连接前，需要多实施一次DNS访问：

* 难以预防DNS劫持
* 多一次DNS访问意味着更长的连接时间，这个不足在手机端更为明显

**如何解决DNS的问题？**

直接将IP配置在客户端，可以解决上述两个问题，很多公司也就是这么做的（俗称“**IP直通车**”）。

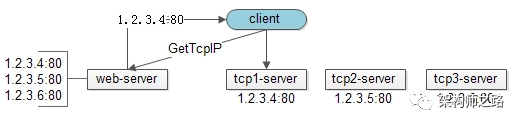
**“IP直通车”有什么新问题？**

将IP写死在客户端，在客户端实施负载均衡，扩展性很差：

* 如果原有IP发生变化，客户端得不到实时通知
* 如果新增IP，即tcp-sever扩容，客户端也得不到实时通知
* 如果负载均衡策略变化，需要升级客户端

**四、服务端实施负载均衡**

只有将复杂的策略下沉到服务端，才能根本上解决扩展性的问题。



增加一个http接口，将客户端的“IP配置”与“均衡策略”放到服务端是一个不错的方案：

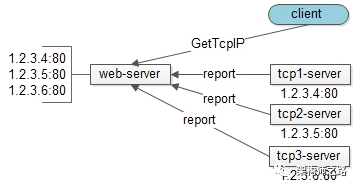
* client每次访问tcp-server前，先调用一个**新增的get-tcp-ip接口**，对于client而言，这个http接口只返回一个tcp-server的IP
* 这个http接口，实现的是原client的IP均衡策略
* 拿到tcp-server的IP后，和原来一样向tcp-server发起TCP长连接

这样的话，扩展性问题就解决了：

* 如果原有IP发生变化，只需要修改get-tcp-ip接口的配置
* 如果新增IP，也是修改get-tcp-ip接口的配置
* 如果负载均衡策略变化，需要升级客户端

然而，**新的问题**又产生了，如果所有IP放在客户端，当有一个IP挂掉的时候，client可以再换一个IP连接，保证可用性，而get-tcp-ip接口只是维护静态的tcp-server集群IP，对于这些IP对应的tcp-server是否可用，是完全不知情的，怎么办呢？

**五、tcp-server状态上报**

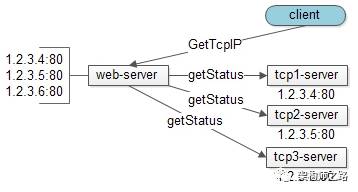


get-tcp-ip接口怎么知道tcp-server集群中各台服务器是否可用呢，tcp-server主动上报是一个潜在方案，如果某一个tcp-server挂了，则会终止上报，对于停止上报状态的tcp-server，get-tcp-ip接口，将不返回给client相应的tcp-server的外网IP。

**该设计的存在的问题？**

诚然，状态上报解决了tcp-server高可用的问题，但这个设计犯了一个“反向依赖”的耦合小错误：使得tcp-server要依赖于一个与本身业务无关的web-server。

**六、tcp-server状态拉取**



**更优的方案**是：web-server通过“拉”的方式获取各个tcp-server的状态，而不是tcp-server通过“推”的方式上报自己的状态。

这样的话，每个tcp-server都独立与解耦，只需专注于资深的tcp业务功能即可。

**高可用、负载均衡、扩展性**等任务由get-tcp-ip的web-server专注来执行。

多说一句，将负载均衡实现在服务端，还有一个**好处**，可以实现异构tcp-server的负载均衡，以及过载保护：

* **静态实施**：web-server下的多个tcp-server的IP可以配置负载权重，根据tcp-server的机器配置分配负载（nginx也有类似的功能）
* **动态实施**：web-server可以根据“拉”回来的tcp-server的状态，动态分配负载，并在tcp-server性能极具下降时实施过载保护

**七、总结**

**web-server如何实施负载均衡？**

利用nginx反向代理来轮询、随机、ip-hash。

**tcp-server怎么快速保证请求一致性？**

单机。

**如何保证高可用？**

客户配置多个tcp-server的域名。

**如何防止DNS劫持，以及加速？**

IP直通车，客户端配置多个tcp-server的IP。

**如何保证扩展性？**

服务端提供get-tcp-ip接口，向client屏屏蔽负载均衡策略，并实施便捷扩容。

**如何保证高可用？**

tcp-server“推”状态给get-tcp-ip接口，

or

get-tcp-ip接口“拉”tcp-server状态。

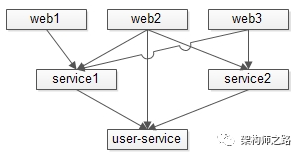
细节重要，思路比细节更重要

## “配置”也有架构演进？看完深有痛感

2017-04-20 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960061&idx=1&sn=3747338a91e85fc33f43d9c1bb98ab10&chksm=bd2d07218a5a8e37ad2e5078736d2449a6ba1e0d75bb18d09990a5f3dcc3af2b7ff414158adb&scene=21" \l "#)

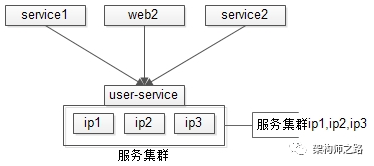
**一、缘起**

随着互联网业务的越来越复杂，用户量与流量越来越大，“服务化分层”是架构演进的必由之路。



如上图：站点应用会调用服务，上游服务调用底层服务，**依赖关系会变得非常复杂**。

对于同一个服务，它有多个上游调用。为了保证高可用，一个底层服务往往是若干个节点形成一个集群提供服务。

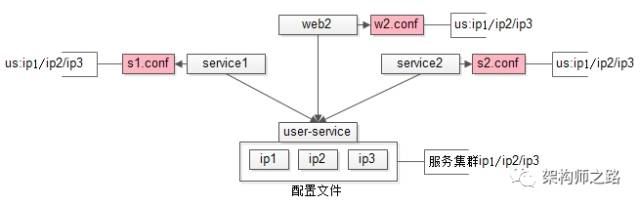


如上图：用户中心服务user-service有三个节点，ip1/ip2/ip3对上游提供服务，任何一个节点当机，都不影响服务的可用性。

那么**问题来了**，当服务集群增减节点的时候，是否存在“反向依赖”，是否“耦合”，是否**上游调用方需要修改配置重启**，是否能做到上游无感知，即“配置的架构变迁”，是今天需要讨论的问题。

**二、配置私藏**

“配置私藏”是配置文件架构的最初级阶段，上游调用下游，每个上游都有一个专属的私有配置文件，记录被调用下游的每个节点配置信息。



如上图：

1）用户中心user-service有ip1/ip2/ip3三个节点

2）service1调用了用户中心，它有一个专属配置文件s1.conf，里面配置了us的集群是ip1/ip2/ip3

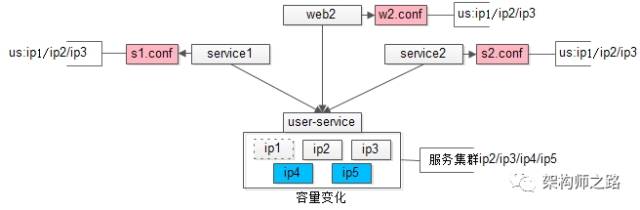
3）service2也调用了用户中心，同理有个配置文件s2.conf，记录了us集群是ip1/ip2/ip3

4）web2也调用了用户中心，同理w2.conf，配置了us集群是ip1/ip2/ip3

是不是很熟悉？

没错，绝大部分公司，初期都是这么玩的。

**配置私藏架构的缺点是什么呢？**

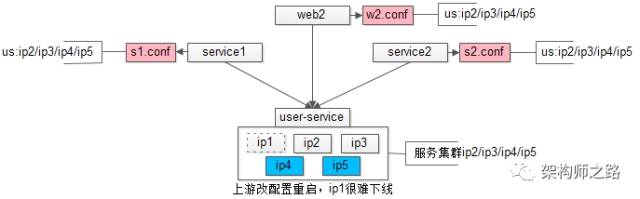


来看一个容量变化的需求：

1）运维检测出ip1节点的硬盘性能下降，通知研发未来要将ip1节点下线

2）由于5月8日要做大促运营活动，未来流量会激增，研发准备增加两个节点ip4和ip5

**此时要怎么做呢？**



需要用户中心的负责人通知所有上游调用者，**修改“私藏”的配置，并重启上游**，连接到新的集群上去。在ip1上没有流量之后，**通知运维将ip1节点下线**，以完成整个缩容扩容过程。

大伙是这么做的么？当业务复杂度较高，研发人数较多，服务依赖关系较复杂的时候，就没这么简单了。

**问题一**：调用方很痛，**容量变化的是你，凭啥修改配置重启的是我**？这是一个典型的“反向依赖”架构设计，上下游通过配置耦合，值得优化（特别是上层服务，ta依赖的服务很多的时候，可能每周都有类似的配合重启需求）。

**问题二**：服务方很痛，**ta不知道有多少个上游调用了自己**（特别是底层基础服务，像用户中心这种，调用ta的上游很多），往往只能通过以下方式来定位上游：

         a）**群里吼**

         b）**发邮件询问**

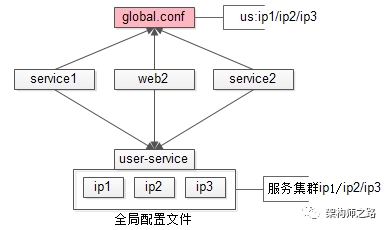
         c）**通过连接找到ip，通过ip问运维，找到机器负责人，再通过机器负责人找到对应调用服务**

（似曾相识的请转发=\_=）

不管哪种方式，都很有可能遗漏，导致ip1一直有流量难以下线，ip4/ip5的流量难以均匀迁移过来。该如何优化呢？

**三、全局配置**

架构的升级并不是一步到位的，先来用最低的成本来解决上述“修改配置重启”的问题一。



“全局配置”法：对于通用的服务，建立全局配置文件，消除配置私藏：

1）**运维层面制定规范**，新建全局配置文件，例如/opt/globalconf/global.conf，如果配置较多，注意做好配置的垂直拆分

2）**对于服务方**，如果是通用的服务，集群信息配置在global.conf里

3）**对于调用方**，调用方禁止配置私藏，必须从global.conf里读取通用下游配置

这么做的**好处**：

1）如果下游**容量变化**，只需要修改一处配置global.conf，而不需要各个上游修改

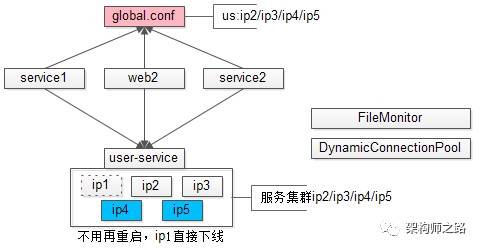
2）**调用方下一次重启的时候**，自动迁移到扩容后的集群上来了

3）**修改成本非常小**，读取配置文件目录变了

**不足**：

如果调用方一直不重启，就没有办法将流量迁移到新集群上去了

**有没有方面实现自动流量迁移呢？**



答案是肯定的，只需要实现**两个并不复杂的组件**，就能实现调用方的流量自动迁移：

1）**文件监控组件FileMonitor**

作用是监控文件的变化，起一个timer，定期监控文件的ModifyTime或者md5就能轻松实现，当文件变化后，实施回调。

2）**动态连接池组件DynamicConnectionPool**

“连接池组件”是RPC-client中的一个子组件，用来维护与多个RPC-server节点之间的连接。所谓**“动态连接池”**，是指连接池中的连接可以动态增加和减少（用锁来互斥或者线程安全的数据结构很容易实现）。

这两个组件完成后：

1）一旦全局配置文件变化，文件监控组件实施回调

2）如果动态连接池组件发现配置中减少了一些节点，就动态的将对应连接销毁，如果增加了一些节点，就动态建立连接，自动完成下游节点的增容与缩容。

**四、配置中心**

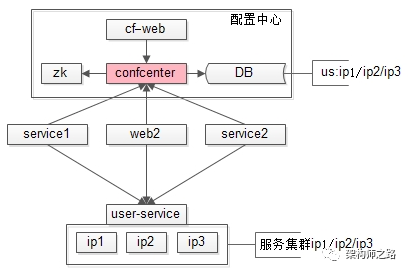
**全局配置文件**是一个能够快速落地的，解决“修改配置重启”问题的方案，但它仍然解决不了，服务提供方“不知道有多少个上游调用了自己”这个问题。

如果不知道多少上游调用了自己，

“按照调用方限流”

“绘制全局架构依赖图”

等需求便**难以实现**，怎么办，可以**采用“配置中心”来解决**。

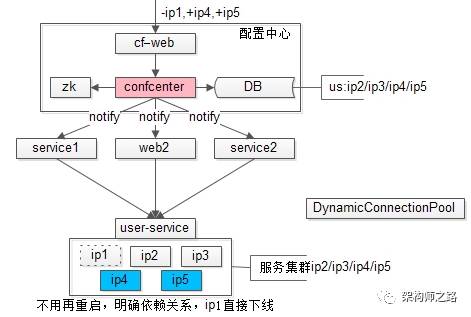


对比“全局配置”与“配置中心”的架构图，会发现配置由静态的文件 升级为 动态的服务：

1）整个**配置中心子系统**由zk、conf-center服务，DB配置存储与，conf-web配置后台组成

2）所有**下游服务的配置**，通过后台设置在配置中心里

3）所有**上游需要拉取配置**，需要去配置中心注册，拉取下游服务配置信息（ip1/ip2/ip3）



**当下游服务需要扩容缩容时**：

4）conf-web配置后台进行设置，新增ip4/ip5，减少ip1

5）conf-center服务将变更的配置推送给已经注册关注相关配置的调用方

6）结合动态连接池组件，完成**自动的扩容与缩容**

配置中心的**好处**：

1）调用方不需要再重启

2）服务方从配置中心中很清楚的知道上游依赖关系，从而实施按照调用方限流

3）很容易从配置中心得到全局架构依赖关系

**痛点一、痛点二同时解决**。

**不足**：系统复杂度相对较高，对配置中心的可靠性要求较高，一处挂全局挂。

**五、总结**

**解决什么问题？**

配置导致系统耦合，架构反向依赖。

**什么痛点？**

**上游痛：**扩容的是下游，改配置重启的是上游

**下游痛：**不知道谁依赖于自己

**配置架构如何演进？**

一、配置私藏

二、全局配置文件

三、配置中心

大伙的配置架构进化到第几个步骤啦？欢迎留言。

若有收获，帮忙转发。

==【完】==

推荐阅读：

[换IP的是你，凭啥重启的却是我？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960049&idx=1&sn=2787e73be4ea31f52213a2b934a16e09&chksm=bd2d072d8a5a8e3b71cef58f313743a6db6f83889748deb7f8f866817511df98adc5e03db39c&scene=21" \l "wechat_redirect" \t "_blank)

## 跨公网调用的大坑与架构优化方案

原创 2017-05-13 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960105&idx=1&sn=0069f2264e227e86a63ee50a4899e0a7&chksm=bd2d06f58a5a8fe33271ed3a378932ad023af7a9004d7fdb44e30a53d53928f984f293a41256&scene=21" \l "#)

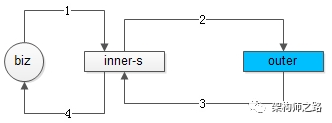
第三方接口挂掉，我们的服务会受影响么？

**一、缘起与大坑**

很多时候，业务需要跨公网调用一个第三方服务提供的接口，为了避免每个调用方都依赖于第三方服务，往往会抽象一个服务：

* 解除调用方与第三方接口的耦合
* 当第三方的接口变动时，只有服务需要修改，而不是所有调用方均修改

此时**接口调用流程是什么样的呢？**



如上图1-4所述：

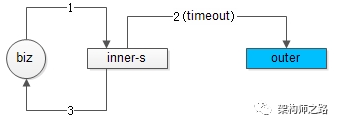
（1）业务调用方调用内部service

（2）内部service跨公网调用第三方接口

（3）第三方接口返回结果给内部service

（4）内部service返回结果给业务调用方

**这个过程存在什么潜在的大坑呢？**



内部服务可能对上游业务提供了很多服务接口，当有一个接口跨公网第三方调用超时时，可能导致所有接口都不可用，即使大部分接口不依赖于跨公网第三方调用。

**为什么会出现这种情况呢？**

内部服务对业务方提供的N个接口，会共用服务容器内的工作线程（假设有100个工作线程）。

假设这N个接口的某个接口跨公网依赖于第三方的接口，发生了网络抖动，或者接口超时（不妨设超时时间为5秒）。

潜台词是，这个工作线程会被占用5秒钟，然后超时返回业务调用方。

假设这个请求的吞吐量为20qps，言下之意，很短的时间内，所有的100个工作线程都会被卡在这个第三方超时等待上，而其他N-1个原本没有问题的接口，也得不到工作线程处理。

**潜在优化方案？**

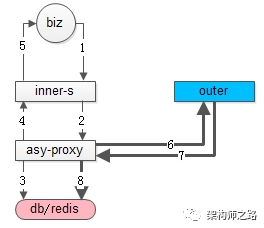
* 增大工作线程数（不根本解决问题）
* 降低超时时间（不根本解决问题）
* 垂直拆分，N个接口拆分成若干个服务，使得在出问题时，被牵连的接口尽可能少（依旧不根本解决问题，难道一个服务只提供一个接口吗？）

跨公网调用的稳定性优化，是本文要讨论的问题。

**二、异步代理法**

**业务场景**：通过OpenID实时获取微信用户基本信息

**解决方案**：增加一个代理，向服务屏蔽究竟是“本地实时”还是“异步远程”去获取返回结果



**本地实时流程**如上图1-5：

（1）业务调用方调用内部service

（2）内部service调用异步代理service

（3）异步代理service通过OpenID在本地拿取数据

（4）异步代理service将数据返回内部service

（5）内部service返回结果给业务调用方

**异步远程流程**如上图6-8粗箭头的部分：

（6）异步代理service定期跨公网调用微信服务

（7）微信服务返回数据

（8）刷新本地数据

**优点**：公网抖动，第三方接口超时，不影响内部接口调用

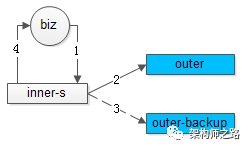
**不足**：本地返回的不是最新数据（很多业务可以接受数据延时）

有时候，内部service和异步代理service可以合成一个service

**三、第三方接口备份与切换法**

**业务场景**：调用第三方短信网关，或者电子合同等

**解决方案**：同时使用（或者备份）多个第三方服务



**流程**如上图1-4：

（1）业务调用方调用内部service

（2）内部service调用第一个三方接口

（3）超时后，调用第二个备份服务，未来都直接调用备份服务，直到超时的服务恢复

（4）内部service返回结果给业务调用方

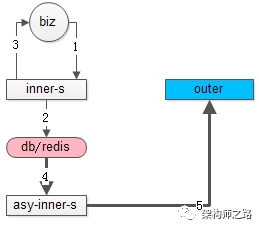
**优点**：公网抖动，第三方接口超时，不影响内部接口调用（初期少数几个请求会超时）

**不足**：不是所有公网调用都能够像短息网关，电子合同服务一样有备份接口的，像微信、支付宝等就只此一家

**四、异步调用法**

**业务场景**：本地结果，同步第三方服务，例如用户在58到家平台下单，58到家平台需要通知平台商家为用户提供服务

**解决方案**：本地调用成功就返回成功，异步调用第三方接口同步数据（和异步代理有微小差别）



**本地流程**如上图1-3：

（1）业务调用方调用内部service

（2）内部service写本地数据

（3）内部service返回结果给业务调用方成功

**异步流程**如上图4-5粗箭头的部分：

（4）异步service定期将本地数据取出（或者通知也行，实时性好）

（5）异步调用第三方接口同步数据

**优点**：公网抖动，第三方接口超时，不影响内部接口调用

**不足**：不是所有业务场景都可以异步同步数据

**五、总结**

跨公网调用第三方，**可能存在的问题**：

* 公网抖动，第三方服务不稳定，影响自身服务
* 一个接口超时，占住工作线程，影响其他接口

降低影响的**优化方案**：

* 增大工作线程数
* 降低超时时间
* 服务垂直拆分

业务需求决定技术方案，结合业务的解决方案：

* **业务能接受旧数据**：读取本地数据，异步代理定期更新数据
* **有多个第三方服务提供商**：多个第三方互备
* **向第三方同步数据**：本地写成功就算成功，异步向第三方同步数据

希望第三方的服务挂掉，不再影响大家的服务。

这个锅，我们不背。

===【完】===



相关推荐：

[TCP接入层的负载均衡、高可用、扩展性架构](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960086&idx=1&sn=70bbe7165ecddc7896767f4503a927fe&chksm=bd2d06ca8a5a8fdc67fcacb169583f53a968fdf623a20395926059d44e6abae6b2e56ff1f9f9&scene=21" \l "wechat_redirect" \t "_blank)

[配置文件架构迭代与演进](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960061&idx=1&sn=3747338a91e85fc33f43d9c1bb98ab10&chksm=bd2d07218a5a8e37ad2e5078736d2449a6ba1e0d75bb18d09990a5f3dcc3af2b7ff414158adb&scene=21" \l "wechat_redirect" \t "_blank)

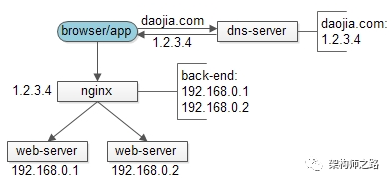
若有收获， 帮忙转发哟

## DNS在架构设计中的巧用

原创 2017-05-17 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960118&idx=1&sn=767e038cb4378be1c88dd42569a9264f&chksm=bd2d06ea8a5a8ffc7c69c71c2153b10565ecf72b6957c5667fbf9099c9f5478111132e967a35&scene=21" \l "#)

**一、缘起**

**一个http请求从客户端到服务端，整个执行流程是怎么样的呢？**



一个典型流程如上：

(1)客户端通过域名daojia.com请求dns-server

(2)dns-server返回域名对应的外网ip(1.2.3.4)

(3)客户端访问外网ip(1.2.3.4)向反向代理nginx

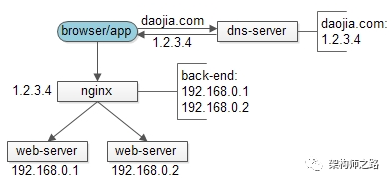
(4)反向代理nginx配置了多个后端web-server服务内网ip(192.168.0.1/192.168.0.2)

(5)请求最终落到某一个web-server进行处理

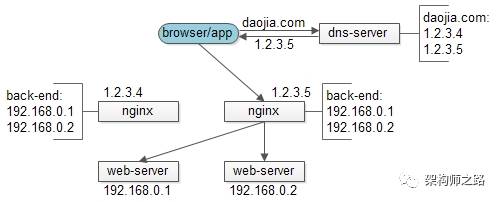
其中，第一个步骤域名daojia.com到外网ip(1.2.3.4)的转换，发生在整个服务端外部，服务端不可控。

架构设计时，能够巧用dns做一些什么事情呢，是本文要讨论的问题。

**二、反向代理水平扩展**

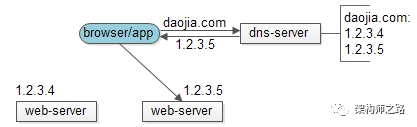


典型的互联网架构中，可以通过增加web-server来扩充web层的性能，但**反向代理nginx仍是整个系统的唯一入口**，如果系统吞吐超过nginx的性能极限，难以扩容，此时就需要dns-server来配合水平扩展。



具体做法是：在dns-server对于同一个域名可以配置多个nginx的外网ip，每次dns解析请求，轮询返回不同的ip，这样就能实现nginx的水平扩展，这个方法叫“**dns轮询**”。

**三、web-server负载均衡**



既然“dns轮询”可以将同一个域名的流量均匀分配到不同的nginx，那么也可以利用它来做web-server的负载均衡：

(1)架构中去掉nginx层

(2)将多个web-server的内网ip直接改为外网ip

(3)在dns-server将域名对应的外网ip进行轮询解析

**和nginx相比，dns来实施负载均衡有什么优缺点呢？**

**优点**：

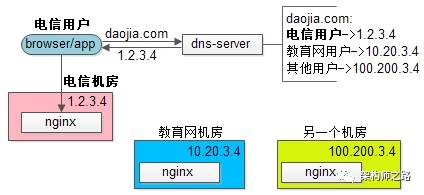
* 利用第三方dns实施，服务端架构不用动
* 少了一层网络请求

**不足**：

* dns只具备解析功能，不能保证对应外网ip的可用性（即使能够做80口的探测，实时性肯定也是比nginx差很多的），而nginx做反向代理时，与web-server之间有保活探测机制，当web-server挂掉时，能够自动迁移流量
* 当web-server需要扩容时，通过dns扩容生效时间长，而nginx是服务端完全自己可控的部分，web-server扩容更实时更方便

因为上面两个原因，**架构上很少取消反向代理层**，而直接使用dns来实施负载均衡。

**四、用户就近访问**



如文章“缘起”中所述，http请求的第一个步骤域名到外网ip的转换，发生在整个服务端外部，服务端不可控，那么如果要实施“根据客户端ip来分配最近的服务器机房访问”，就只能在dns-server上做了：

(1)电信用户想要访问某一个服务器资源

(2)浏览器向dns-server发起服务器域名解析请求

(3)dns-server识别出访问者是电信用户

(4)dns-server将电信机房的nginx外网ip返回给访问者

(5)访问者就近访问

根据用户ip来返回最近的服务器ip，称为“**智能dns**”，**cdn**以及**多机房多活**中最常用。

**五、总结**

架构设计中，dns有它独特的功能和作用：

* **dns轮询**，水平扩展反向代理层
* 去掉反向代理层，**利用dns实施负载均衡**
* **智能dns**，根据用户ip来就近访问服务器

其他相关技术点：

[lvs为何不能完全替代DNS轮询](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959595&idx=1&sn=5f0633afd24c547b895f29f6538baa99&scene=21" \l "wechat_redirect" \t "_blank)

[反向代理层的高可用架构设计](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959728&idx=1&sn=933227840ec8cdc35d3a33ae3fe97ec5&chksm=bd2d046c8a5a8d7a13551124af36bedf68f7a6e31f6f32828678d2adb108b86b7e08c678f22f&scene=21" \l "wechat_redirect" \t "_blank)

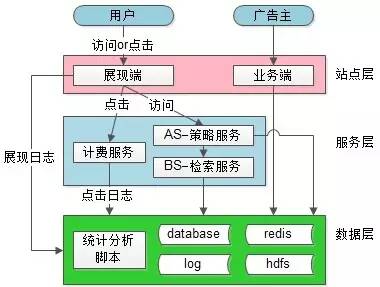
[免dns解析，ip直通车技术](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960086&idx=1&sn=70bbe7165ecddc7896767f4503a927fe&chksm=bd2d06ca8a5a8fdc67fcacb169583f53a968fdf623a20395926059d44e6abae6b2e56ff1f9f9&scene=21" \l "wechat_redirect" \t "_blank)

## 互联网智能广告系统简易流程与架构

原创 2017-05-26 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960148&idx=1&sn=7c8a2d789fb20981355e49c3defe0229&chksm=bd2d06888a5a8f9e5ef5cf6cfa415d06800f5468148a0ba7a802e646aeec9610823ec0fc6d5d&scene=21" \l "#)

很多朋友估计没有做过这一块，争取最简洁的语言描述清楚。

**一、业务简述**



从业务上看    整个智能广告系统，主要分为：

1）**业务端**：广告主的广告后台

2）**展现端**：用户实际访问的页面

业务端，**广告主**主要有两类行为：

1）**广告设置行为**：例如设置投放计划，设置地域，类别，关键字，竞价等

2）**效果查看行为**：例如广告展示次数是多少，广告点击次数是多少等

展现端，**用户**主要也有两类行为：

1）**站点浏览行为**：用户浏览实际的信息，此时广告系统决定出广告主的什么广告

2）**广告点击行为**：此时广告系统会对广告主进行扣费

**二、业务流程**

下面通过一个的例子，让业务流程更直观。

步骤一：**广告主**在业务端投递广告

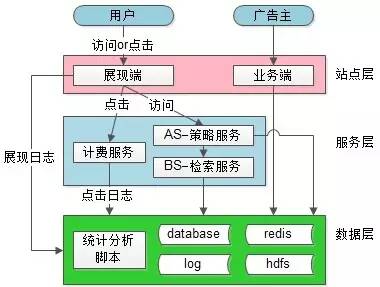
广告主登录业务端后台，进行设置：

* 今日投放**地域**是“北京-上地”
* 投放**类别**是“租房”
* 定向**人群**为“女”，“30岁以下”
* 需要推广的广告**内容**是他发布的一条“房屋出租”的帖子
* **竞价**设置的是0.2元
* 单日**预算**是20元

这些数据，当然通过业务端存储到了数据层，即数据库和缓存里。

步骤二：**用户**来到了网站，进入了“北京-上地-租房”类别，广告初筛实施

用户产生了平台浏览行为，网站除了展示自然内容，还要展示广告内容。被展现的广告不能太离谱，太离谱用户也不会点击。



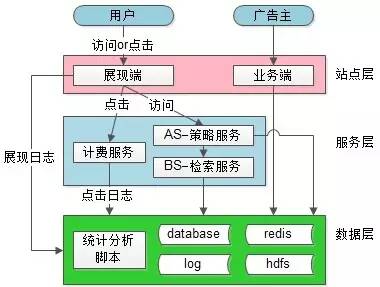
合适的广告，必须符合“**语义相关性**”，即**基础检索**属性（广告属性）必须符合（**广告能否满足用户的需求**，满足了点击率才高），这个工作是通过BS-basic search检索服务完成的。

BS从数据层检索到“北京-上地-租房”的广告帖子。

步骤三：用户属性与广告主属性匹配，广告精筛实施

步骤二中，基础属性初筛了以后，要进行**更深层次的策略筛选**（**用户能否满足广告的需求**），此例中，广告主的精准需求为：

* 用户**性别**为“女”
* 用户**年龄**为“30岁以下”
* 用户**访问IP**是“北京”



系统将初筛出来的M条广告和用户属性进行匹配筛选，又过滤掉了一部分，最后剩余N条待定广告，这些广告既满足用户的需求（初筛），这些用户也满足广告主的需求（精筛），后者是在AS-advanced search策略服务完成的。

步骤四：综合排序，并返回Top X的广告

经过步骤2和步骤3的初筛和精筛之后，待选的N条广告既能满足用户当前的需求，用户亦能满足广告主的筛选需求，但实际情况是，广告位只有3个，怎么办呢？就需要我们对N条广告进行综合打分排序（**满足平台的需求**，广告平台要多赚钱嘛）。

**打分排序的依据是什么呢？**

有人说按照竞价排序bid，出价高的打分高（这是大家对百度最大的误解，百度是cpc收费）

有人说按照CTR点击率排序，CTR高的点的人多（百度的kpi指标可不是pv）

出价高，但没人点击，广告平台没有收益；点击率高，但出价低，广告平台还是没有收益。**最终应该按照广告的出价与CTR的乘积作为综合打分排序的依据，bid\*CTR**。

既然bid\*CTR是所有广告综合打分的依据，且出价bid又是广告主事先设定好的，那么实际上，广告排序问题的核心又转向了广告CTR的预测，**CTR预测**是推荐系统、广告系统、搜索系统里非常重要的一部分，是一个工程，算法，业务三方结合的问题，本文就不展开讨论了。

无论如何，N条广告，根据bid\*预估CTR进行综合打分排序后，返回了打分最高的3个广告（广告位只有3个）。

有些系统没有第二步骤用户属性过滤，而是将用户属性因素考虑到综合排序中。

步骤五：展现端展示了广告，用户点击了广告

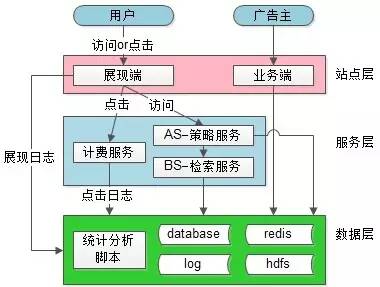
展示了广告后，展现端js会上报广告**展示日志**，有部分用户点击了广告，服务端会记录**点击日志**，这些日志可以作为广告算法实施的数据源，同时，他们经过统计分析之后，会被展示给广告主，让他们能够看到自己广告的展示信息，点击信息。

这些日志（一般会实施AB测），也是算法效果好坏评估的重要依据，根据效果逐步优化改进算法。

步骤六：对广告主进行扣费

用户既然点击了广告，平台就要对投放广告的广告主进行**扣费**了，扣费前当然要经过反作弊系统的过滤（主要是恶意点击），扣费后信息会实时反映到数据层，费用扣光后，广告就要从数据层下线。

**三、系统综述**



聊完业务流程，再来看系统架构，任何脱离业务的架构设计都是耍流氓。

从系统分层架构上看，智能广告系统分为三层：

* **站点层**：用户和广告主直接面向的网站站点
* **服务层**：为了实现智能广告的业务逻辑，提供的通用服务，此处又主要分为**四大类服务**：

策略服务BS：实施广告策略，综合排序

检索服务AS：语义相关性检索

计费服务：用户点击广告时进行扣费

反作弊服务：不是每次点击都扣费，要经过反作弊，去除恶意点击（相对独立，未在架构图中画出）

* **数据层**：用户数据，广告数据，竞价数据，日志数据等等等等

**四、总结**

智能广告系统的业务流程与系统架构：

1）**广告主投放与设置广告**

2）**用户访问平台，展现合适广告**

    通过**广告属性**，进行“语义相关性”初筛，通过BS完成

    通过**用户属性**，出价信息，点击率预测信息，进行综合打分排序筛选，通过AS完成

3）**记录展现日志，点击日志，进行扣费**

广告是展现，是一个：

* 广告满足用户需求（初筛）
* 用户满足广告需求（精筛）
* 平台利益最大化（bid\*CTR综合排序）

的过程

**广告的排序**不是由出价(bid)决定的，而是由出价(bid)\*点击率(ctr)决定的。

**点击率(ctr)**是一个未来将要发生的行为，智能广告系统的**核心与难点**是点击率预测。

==【完】==

文章说清楚了么，希望这一分钟没有浪费。

若有收获，帮转哟。

## 100亿数据1万属性数据架构设计

原创 2017-01-18 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959855&idx=1&sn=f33abe8ec598c273f29cebb9365ece59&chksm=bd2d07f38a5a8ee58a944507a134e1da1efc3ac9c4d1c4cff261137cd986e51f5fe7cee9de15&scene=21" \l "#)

一分钟系列之《[啥，又要为表增加一列属性？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959762&idx=1&sn=f7d8d7648416a0157af6ac61a6b555c8&chksm=bd2d040e8a5a8d181ec1baa2e96982991ddfc218bb6838f38da61b651e4b8de5f0bbbd94f814&scene=21" \l "wechat_redirect" \t "_blank)》分享了两种数据库属性扩展思路，被喷得厉害。第二天补充了一篇《[这才是真正的表扩展方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959765&idx=1&sn=b9916aa95c320e41035977e0a8098ca6&chksm=bd2d04098a5a8d1f3af38f658c05002151e621170949d2e3bb5b1bceea55c64b0477dba4c647&scene=21" \l "wechat_redirect" \t "_blank)》，分享了互联网大数据高并发情况下，数据库属性扩容的成熟工具及思路。

对于version + ext方案，还是有很多朋友质疑“线上不可能这么用”。本篇将讲述一下58同城最核心的数据“帖子”的架构实现技术细节，说明不仅不是“不可能这么用”，而是大数据，可变属性，高吞吐场景下的“常用手段”。

**一、背景描述及业务介绍**

**问：什么是数据库扩展的version + ext方案？**

使用ext来承载不同业务需求的个性化属性，使用version来标识ext里各个字段的含义。



例如上述user表：

verion=0表示ext里是passwd/nick

version=1表示ext里是passwd/nick/age/sex

**优点？**

（1）可以随时动态扩展属性，扩展性好

（2）新旧两种数据可以同时存在，兼容性好

**不足？**

（1）ext里的字段无法建立索引

（2）ext里的key值有大量冗余，建议key短一些

**问：什么是58同城最核心的数据？**

58同城是一个信息平台，有很多垂直品类：招聘、房产、二手物品、二手车、黄页等等，每个品类又有很多子品类，不管哪个品类，最核心的数据都是“帖子信息”（业务像一个大论坛？）。

**问：帖子信息有什么特点？**

大家去58同城的首页上看看就知道了：

（1）每个品类的属性千差万别，招聘帖子和二手帖子属性完全不同，二手手机和二手家电的属性又完全不同，目前恐怕有近万个属性

（2）帖子量很大，100亿级别

（3）每个属性上都有查询需求（各组合属性上都可能有组合查询需求），招聘要查职位/经验/薪酬范围，二手手机要查颜色/价格/型号，二手要查冰箱/洗衣机/空调

（4）查询量很大，每秒几10万级别

如何解决**100亿数据量，1万属性，多属性组合查询，10万并发查询**的技术难题，是今天要讨论的内容。

**二、最容易想到的方案**

每个公司的发展都是一个从小到大的过程，撇开并发量和数据量不谈，先看看

（1）如何实现属性扩展性需求

（2）多属性组合查询需求

最开始，可能只有一个**招聘品类**，那帖子表可能是这么设计的：

tiezi(tid,uid, c1, c2, c3)

那如何满足各属性之间的组合查询需求呢？

最容易想到的是通过组合索引：

index\_1(c1,c2) index\_2(c2, c3) index\_3(c1, c3)

随着业务的发展，又新增了一个**房产类别**，新增了若干属性，新增了若干组合查询，于是帖子表变成了：

tiezi(tid,uid, c1, c2, c3, c10, c11, c12, c13)

其中c1,c2,c3是招聘类别属性，c10,c11,c12,c13是房产类别属性，这两块属性一般没有组合查询需求

但为了满足房产类别的查询需求，又要建立了若干组合索引（不敢想有多少个索引能覆盖所有两属性查询，三属性查询）

是不是发现玩不下去了？

**三、友商的玩法**

新增属性是一种扩展方式，新增表也是一种方式，有友商是这么玩的，按照业务进行垂直拆分：

tiezi\_zhaopin(tid,uid, c1, c2, c3)

tiezi\_fangchan(tid,uid, c10, c11, c12, c13)

这些表，这些服务维护在不同的部门，不同的研发同学手里，看上去各业务线灵活性强，这恰恰是悲剧的开始：

（1）tid如何规范？

（2）属性如何规范？

（3）按照uid来查询怎么办（查询自己发布的所有帖子）？

（4）按照时间来查询怎么办（最新发布的帖子）？

（5）跨品类查询怎么办（例如首页搜索框）？

（6）技术范围的扩散，有的用mongo存储，有的用mysql存储，有的自研存储

（7）重复开发了不少组件

（8）维护成本过高

（9）…

想想看，电商的商品表，不可能一个类目一个表的。

**四、58同城的玩法**

**【统一帖子中心服务】**

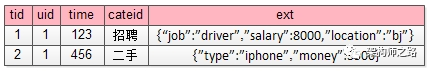
平台型创业型公司，可能有多个品类，例如58同城的招聘房产二手，很多异构数据的存储需求，到底是分还是合，无需纠结：**基础数据基础服务的统一**，无疑是58同城技术路线发展roadmap上最正确的决策之一，把这个方针坚持下来，@老崔 @晓飞 这些高瞻远瞩的先贤功不可没，业务线会有“扩展性”“灵活性”上的微词，后文看看先贤们如何通过一些巧妙的技术方案来解决的。

如何将不同品类，异构的数据统一存储起来，采用的就是类似version+ext的方式：

tiezi(tid,uid, time, title, cate, subcate, xxid, ext)

（1）一些通用的字段抽取出来单独存储

（2）通过cate, subcate, xxid等来定义ext是何种含义（和version有点像？）



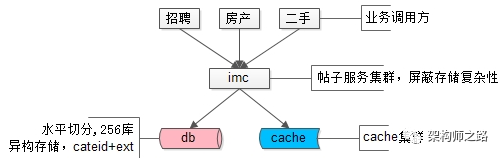
（3）通过ext来存储不同业务线的个性化需求

例如招聘的帖子：

ext : {“job”:”driver”,”salary”:8000,”location”:”bj”}

而二手的帖子：

ext : {”type”:”iphone”,”money”:3500}



58同城最核心的帖子数据，100亿的数据量，分256库，异构数据mysql存储，上层架了一个服务，使用memcache做缓存，就是这样一个简单的架构，一直坚持这这么多年。上层的这个服务，就是**58同城最核心的统一服务IMC（Imformation Management Center）**，注意这个最核心，是没有之一。

解决了海量异构数据的存储问题，遇到的**新问题**是：

（1）每条记录ext内key都需要重复存储，占据了大量的空间，能否压缩存储

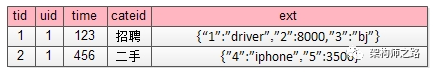
（2）cateid已经不足以描述ext内的内容，品类有层级，深度不确定，ext能否具备自描述性

（3）随时可以增加属性，保证扩展性

**【统一类目属性服务】**

每个业务有多少属性，这些属性是什么含义，值的约束等揉不到帖子服务里，怎么办呢？

58同城的先贤们抽象出一个统一的类目、属性服务，单独来管理这些信息，而帖子库ext字段里json的key，统一由数字来表示，减少存储空间。



如上图所示，json里的key不再是”salary” ”location” ”money” 这样的长字符串了，取而代之的是数字1,2,3,4，这些数字是什么含义，属于哪个子分类，值的校验约束，统一都存储在类目、属性服务里。



这个表里对帖子中心服务里ext字段里的数字key进行了解释：

1代表job，属于招聘品类下100子品类，其value必须是一个小于32的[a-z]字符

4代表type，属于二手品类下200子品类，其value必须是一个short

这样就对原来帖子表ext里的

ext : {“1”:”driver”,”2”:8000,”3”:”bj”}

ext : {”4”:”iphone”,”5”:3500}

key和value都做了统一约束。

除此之外，如果ext里某个key的value不是正则校验的值，而是枚举值时，需要有一个对值进行限定的枚举表来进行校验：



这个枚举校验，说明key=4的属性（对应属性表里二手，手机类型字段），其值不只是要进行“short类型”校验，而是value必须是固定的枚举值。

ext : {”4”:”iphone”,”5”:3500}这个ext就是不合法的（key=4的value=iphone不合法），合法的应该为

ext : {”4”:”5”,”5”:3500}

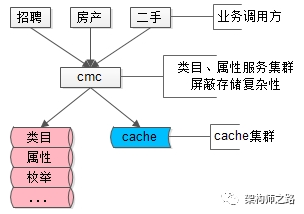
此外，类目属性服务还能记录类目之间的层级关系：

（1）一级类目是招聘、房产、二手…

（2）二手下有二级类目二手家具、二手手机…

（3）二手手机下有三级类目二手iphone，二手小米，二手三星…

（4）…



协助解释58同城最核心的帖子数据，描述品类层级关系，保证各类目属性扩展性，保证各属性值合理性校验，就是**58同城另一个统一的核心服务CMC（Category Management Center）**。

多提一句，类目、属性服务像不像电商系统里的SKU扩展服务？

（1）品类层级关系，对应电商里的类别层级体系

（2）属性扩展，对应电商里各类别商品SKU的属性

（3）枚举值校验，对应属性的枚举值，例如颜色：红，黄，蓝

解决了key压缩，key描述，key扩展，value校验，品类层级的问题，**还有这样的一个问题**没有解决：每个品类下帖子的属性各不相同，查询需求各不相同，如何解决100亿数据量，1万属性的查询需求，是58同城面临的新问题。

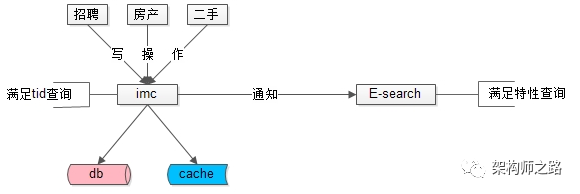
**【统一检索服务】**

数据量很大的时候，不同属性上的查询需求，不可能通过组合索引来满足所有查询需求，怎么办呢？

58同城的先贤们，从一早就确定了“外置索引，统一检索服务”的技术路线：

（1）数据库提供“帖子id”的正排查询需求

（2）所有非“帖子id”的个性化检索需求，统一走外置索引



元数据与索引数据的操作遵循：

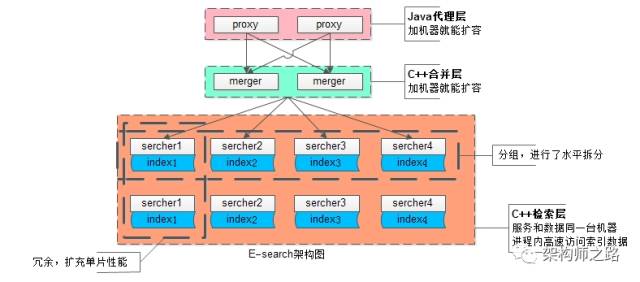
（1）对帖子进行tid正排查询，直接访问帖子服务

（2）对帖子进行修改，帖子服务通知检索服务，同时对索引进行修改

（3）对帖子进行复杂查询，通过检索服务满足需求

这个扛起58同城80%终端请求（不管来自PC还是APP，不管是主页、城市页、分类页、列表页、详情页，很可能这个请求最终会是一个检索请求）的服务，就是**58同城另一个统一的核心服务E-search**，这个搜索引擎的每一行代码都来自58同城@老崔 @老龚 等先贤们，目前系统维护者，就是“架构师之路”里屡次提到的@龙神 。

对于这个服务的架构，简单展开说明一下：



为应对100亿级别数据量、几十万级别的吞吐量，业务线各种复杂的复杂检索查询，扩展性是设计重点：

（1）**统一的Java代理层集群**，其无状态性能够保证增加机器就能扩充系统性能

（2）**统一的合并层C服务集群**，其无状态性也能够保证增加机器就能扩充系统性能

（3）**搜索内核检索层C服务集群**，服务和索引数据部署在同一台机器上，服务启动时可以加载索引数据到内存，请求访问时从内存中load数据，访问速度很快

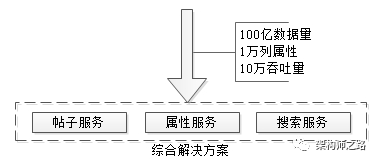
（3.1）为了满足数据容量的扩展性，索引数据进行了水平切分，增加切分份数，就能够无限扩展性能

（3.2）为了满足一份数据的性能扩展性，同一份数据进行了冗余，理论上做到增加机器就无限扩展性能

系统时延，100亿级别帖子检索，包含请求分合，拉链求交集，从merger层均可以做到10ms返回。

58同城的帖子业务，一致性不是主要矛盾，E-search会定期全量重建索引，以保证即使数据不一致，也不会持续很长的时间。

**五、总结**



文章写了很长，最后做一个简单总结，面对**100亿数据量，1万列属性，10万吞吐量的业务需求**，58同城的经验，是采用了元数据服务、属性服务、搜索服务来解决的。

再回到文首version + ext的方案，希望朋友有新的收获和感触，帮转哈。

==【完】==

相关阅读：

《[啥，又要为表增加一列属性？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959762&idx=1&sn=f7d8d7648416a0157af6ac61a6b555c8&chksm=bd2d040e8a5a8d181ec1baa2e96982991ddfc218bb6838f38da61b651e4b8de5f0bbbd94f814&scene=21" \l "wechat_redirect" \t "_blank)》 <- 相关

《[这才是真正的表扩展方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959765&idx=1&sn=b9916aa95c320e41035977e0a8098ca6&chksm=bd2d04098a5a8d1f3af38f658c05002151e621170949d2e3bb5b1bceea55c64b0477dba4c647&scene=21" \l "wechat_redirect" \t "_blank)》 <- 相关

《[究竟啥才是互联网架构“高可用”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959728&idx=1&sn=933227840ec8cdc35d3a33ae3fe97ec5&chksm=bd2d046c8a5a8d7a13551124af36bedf68f7a6e31f6f32828678d2adb108b86b7e08c678f22f&scene=21" \l "wechat_redirect" \t "_blank)》 <- 火

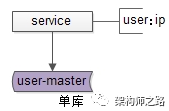
《[究竟啥才是互联网架构“高并发”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959830&idx=1&sn=ce1c5a58caed227d7dfdbc16d6e1cea4&chksm=bd2d07ca8a5a8edc45cc45c4787cc72cf4c8b96fb43d2840c7ccd44978036a7d39a03dd578b5&scene=21" \l "wechat_redirect" \t "_blank)》 <- 火

## 数据库秒级平滑扩容架构方案

原创 2017-02-09 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959883&idx=1&sn=e7df8510c7096a5b069e0f12eaaca010&chksm=bd2d07978a5a8e815c2ae41b16b6b4c579923502fb919008a22bb108a1e920109f25387f8903&scene=21" \l "#)

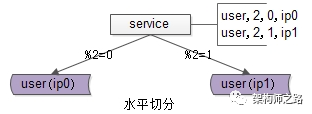
**一、缘起**

（1）并发量大，流量大的互联网架构，一般来说，数据库上层都有一个**服务层**，服务层记录了“业务库名”与“数据库实例”的映射关系，通过**数据库连接池**向数据库路由sql语句以执行：



如上图：服务层配置用户库user对应的数据库实例物理位置为ip（其实是一个内网域名）。

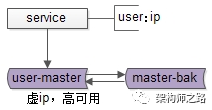
（2）随着数据量的增大，数据要进行**水平切分**，分库后将数据分布到不同的数据库实例（甚至物理机器）上，以达到降低数据量，增强性能的扩容目的：



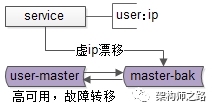
如上图：用户库user分布在两个实例上，ip0和ip1，服务层通过用户标识uid取模的方式进行寻库路由，模2余0的访问ip0上的user库，模2余1的访问ip1上的user库。

关于数据库水平切分，垂直切分的更多细节，详见《[一分钟掌握数据库垂直拆分](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959773&idx=1&sn=7e4ad0dcd050f6662dfaf39d9de36f2c&chksm=bd2d04018a5a8d17b92098b4840aac23982e32d179cdd957e4c55011f6a08f6bd31f9ba5cfee&scene=21" \l "wechat_redirect" \t "_blank)》 。

（3）互联网架构需要保证**数据库高可用**，常见的一种方式，使用双主同步+keepalived+虚ip的方式保证数据库的可用性：



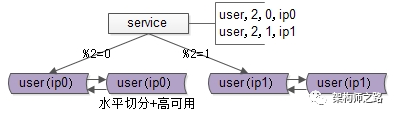
如上图：两个相互同步的主库使用相同的虚ip。



如上图：当主库挂掉的时候，虚ip自动漂移到另一个主库，**整个过程对调用方透明**，通过这种方式保证数据库的高可用。

关于高可用的更多细节，详见《[究竟啥才是互联网架构“高可用”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959728&idx=1&sn=933227840ec8cdc35d3a33ae3fe97ec5&chksm=bd2d046c8a5a8d7a13551124af36bedf68f7a6e31f6f32828678d2adb108b86b7e08c678f22f&scene=21" \l "wechat_redirect" \t "_blank)》。

（4）综合上文的（2）和（3），线上实际的架构，既有水平切分，又有高可用保证，所以实际的数据库架构是这样的：



**提问**：如果数据量持续增大，分2个库性能扛不住了，该怎么办呢？

**回答**：继续水平拆分，拆成更多的库，降低单库数据量，增加库主库实例（机器）数量，提高性能。

**最终问题抛出**：分成x个库后，随着数据量的增加，要增加到y个库，数据库扩容的过程中，能否平滑，持续对外提供服务，保证服务的可用性，是本文要讨论的问题。

**二、停服务方案**

在讨论平滑方案之前，先简要说明下“x库拆y库”停服务的方案：

（1）站点挂一个公告“为了为广大用户提供更好的服务，本站点/游戏将在今晚00:00-2:00之间升级，届时将不能登录，用户周知”

（2）停服务

（3）新建y个库，做好高可用

（4）数据迁移，重新分布，写一个数据迁移程序，从x个库里导入到y个库里，路由规则由%x升级为%y

（5）修改服务配置，原来x行配置升级为y行

（6）重启服务，连接新库重新对外提供服务

整个过程中，**最耗时的是第四步数据迁移**。

**回滚方案**：

如果数据迁移失败，或者迁移后测试失败，则将配置改回x库，恢复服务，改天再挂公告。

**方案优点**：简单

**方案缺点**：

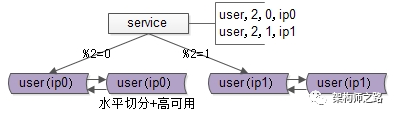
（1）停服务，不高可用

（2）技术同学压力大，所有工作要在规定时间内做完，根据经验，压力越大约容易出错（这一点很致命）

（3）如果有问题第一时间没检查出来，启动了服务，运行一段时间后再发现有问题，难以回滚，需要回档，可能会丢失一部分数据

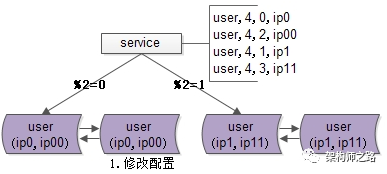
有没有更平滑的方案呢？

**三、秒级、平滑、帅气方案**



再次看一眼扩容前的架构，分两个库，假设每个库1亿数据量，如何**平滑扩容，增加实例数，降低单库数据量**呢？三个简单步骤搞定。

**（1）修改配置**



主要修改两处：

a）数据库实例所在的机器做双虚ip，原来%2=0的库是虚ip0，现在增加一个虚ip00，%2=1的另一个库同理

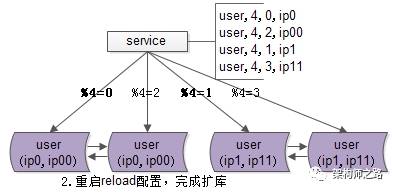
b）修改服务的配置（不管是在配置文件里，还是在配置中心），将2个库的数据库配置，改为4个库的数据库配置，**修改的时候要注意旧库与辛苦的映射关系**：

%2=0的库，会变为%4=0与%4=2；

%2=1的部分，会变为%4=1与%4=3；

这样修改是为了保证，拆分后依然能够路由到正确的数据。

**（2）reload配置，实例扩容**



服务层reload配置，reload可能是这么几种方式：

a）比较原始的，重启服务，读新的配置文件

b）高级一点的，配置中心给服务发信号，重读配置文件，重新初始化数据库连接池

不管哪种方式，reload之后，数据库的**实例扩容就完成了**，原来是2个数据库实例提供服务，现在变为4个数据库实例提供服务，这个过程一般可以在秒级完成。

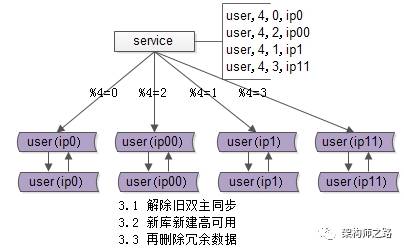
整个过程可以逐步重启，**对服务的正确性和可用性完全没有影响**：

a）即使%2寻库和%4寻库同时存在，也不影响数据的正确性，因为此时仍然是双主数据同步的

b）服务reload之前是不对外提供服务的，冗余的服务能够保证高可用

完成了实例的扩展，会发现每个数据库的数据量依然没有下降，所以第三个步骤还要做一些收尾工作。

**（3）收尾工作，数据收缩**



有这些一些**收尾工作**：

a）把双虚ip修改回单虚ip

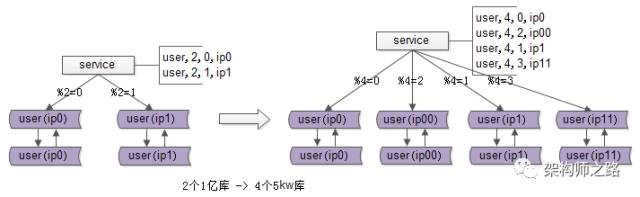
b）解除旧的双主同步，让成对库的数据不再同步增加

c）增加新的双主同步，保证高可用

d）删除掉冗余数据，例如：ip0里%4=2的数据全部干掉，只为%4=0的数据提供服务啦

这样下来，**每个库的数据量就降为原来的一半**，**数据收缩完成**。

**四、总结**



该帅气方案能够实现n库扩2n库的秒级、平滑扩容，增加数据库服务能力，降低单库一半的数据量，其核心原理是：**成倍扩容，避免数据迁移**。

**迁移步骤**：

（1）修改配置

（2）reload配置，**实例扩容**完成

（3）删除冗余数据等收尾工作，**数据量收缩**完成

希望大伙有收获，好文值得**转发**。

==【完】==

相关阅读：

[一分钟掌握数据库垂直拆分](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959773&idx=1&sn=7e4ad0dcd050f6662dfaf39d9de36f2c&chksm=bd2d04018a5a8d17b92098b4840aac23982e32d179cdd957e4c55011f6a08f6bd31f9ba5cfee&scene=21" \l "wechat_redirect" \t "_blank)

[究竟啥才是互联网架构“高可用”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959728&idx=1&sn=933227840ec8cdc35d3a33ae3fe97ec5&chksm=bd2d046c8a5a8d7a13551124af36bedf68f7a6e31f6f32828678d2adb108b86b7e08c678f22f&scene=21" \l "wechat_redirect" \t "_blank)

[究竟啥才是互联网架构“高并发”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959830&idx=1&sn=ce1c5a58caed227d7dfdbc16d6e1cea4&chksm=bd2d07ca8a5a8edc45cc45c4787cc72cf4c8b96fb43d2840c7ccd44978036a7d39a03dd578b5&scene=21" \l "wechat_redirect" \t "_blank)

[100亿数据1万属性数据架构设计](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959855&idx=1&sn=f33abe8ec598c273f29cebb9365ece59&chksm=bd2d07f38a5a8ee58a944507a134e1da1efc3ac9c4d1c4cff261137cd986e51f5fe7cee9de15&scene=21" \l "wechat_redirect" \t "_blank)

## 58到家数据库30条军规解读

原创 2017-02-15 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959906&idx=1&sn=2cbdc66cfb5b53cf4327a1e0d18d9b4a&chksm=bd2d07be8a5a8ea86dc3c04eced3f411ee5ec207f73d317245e1fefea1628feb037ad71531bc&scene=21" \l "#)

**军规适用场景**：并发量大、数据量大的互联网业务

**军规**：介绍内容

**解读**：讲解原因，解读比军规更重要

**一、基础规范**

**（1）必须使用InnoDB存储引擎**

解读：支持事务、行级锁、并发性能更好、CPU及内存缓存页优化使得资源利用率更高

**（2）必须使用UTF8字符集**

解读：万国码，无需转码，无乱码风险，节省空间

**（3）数据表、数据字段必须加入中文注释**

解读：N年后谁tm知道这个r1,r2,r3字段是干嘛的

**（4）禁止使用存储过程、视图、触发器、Event**

解读：高并发大数据的互联网业务，架构设计思路是“解放数据库CPU，将计算转移到服务层”，并发量大的情况下，这些功能很可能将数据库拖死，业务逻辑放到服务层具备更好的扩展性，能够轻易实现“增机器就加性能”。数据库擅长存储与索引，CPU计算还是上移吧

**（5）禁止存储大文件或者大照片**

解读：为何要让数据库做它不擅长的事情？大文件和照片存储在文件系统，数据库里存URI多好

**二、命名规范**

**（6）只允许使用内网域名，而不是ip连接数据库**

**（7）线上环境、开发环境、测试环境数据库内网域名遵循命名规范**

业务名称：xxx

线上环境：dj.xxx.db

开发环境：dj.xxx.rdb

测试环境：dj.xxx.tdb

**从库**在名称后加-s标识，**备库**在名称后加-ss标识

线上从库：dj.xxx-s.db

线上备库：dj.xxx-sss.db

**（8）库名、表名、字段名：小写，下划线风格，不超过32个字符，必须见名知意，禁止拼音英文混用**

**（9）表名t\_xxx，非唯一索引名idx\_xxx，唯一索引名uniq\_xxx**

**三、表设计规范**

**（10）单实例表数目必须小于500**

**（11）单表列数目必须小于30**

**（12）表必须有主键，例如自增主键**

解读：

a）主键递增，数据行写入可以提高插入性能，可以避免page分裂，减少表碎片提升空间和内存的使用

b）主键要选择较短的数据类型， Innodb引擎普通索引都会保存主键的值，较短的数据类型可以有效的减少索引的磁盘空间，提高索引的缓存效率

c） 无主键的表删除，在row模式的主从架构，会导致备库夯住

**（13）禁止使用外键，如果有外键完整性约束，需要应用程序控制**

解读：外键会导致表与表之间耦合，update与delete操作都会涉及相关联的表，十分影响sql 的性能，甚至会造成死锁。高并发情况下容易造成数据库性能，大数据高并发业务场景数据库使用以性能优先

**四、字段设计规范**

**（14）必须把字段定义为NOT NULL并且提供默认值**

解读：

a）null的列使索引/索引统计/值比较都更加复杂，对MySQL来说更难优化

b）null 这种类型MySQL内部需要进行特殊处理，增加数据库处理记录的复杂性；同等条件下，表中有较多空字段的时候，数据库的处理性能会降低很多

c）null值需要更多的存储空，无论是表还是索引中每行中的null的列都需要额外的空间来标识

d）对null 的处理时候，只能采用is null或is not null，而不能采用=、in、<、<>、!=、not in这些操作符号。如：where name!=’shenjian’，如果存在name为null值的记录，查询结果就不会包含name为null值的记录

**（15）禁止使用TEXT、BLOB类型**

解读：会浪费更多的磁盘和内存空间，非必要的大量的大字段查询会淘汰掉热数据，导致内存命中率急剧降低，影响数据库性能

**（16）禁止使用小数存储货币**

解读：使用整数吧，小数容易导致钱对不上

**（17）必须使用varchar(20)存储手机号**

解读：

a）涉及到区号或者国家代号，可能出现+-()

b）手机号会去做数学运算么？

c）varchar可以支持模糊查询，例如：like“138%”

**（18）禁止使用ENUM，可使用TINYINT代替**

解读：

a）增加新的ENUM值要做DDL操作

b）ENUM的内部实际存储就是整数，你以为自己定义的是字符串？

**五、索引设计规范**

**（19）单表索引建议控制在5个以内**

**（20）单索引字段数不允许超过5个**

解读：字段超过5个时，实际已经起不到有效过滤数据的作用了

**（21）禁止在更新十分频繁、区分度不高的属性上建立索引**

解读：

a）更新会变更B+树，更新频繁的字段建立索引会大大降低数据库性能

b）“性别”这种区分度不大的属性，建立索引是没有什么意义的，不能有效过滤数据，性能与全表扫描类似

**（22）建立组合索引，必须把区分度高的字段放在前面**

解读：能够更加有效的过滤数据

**六、SQL使用规范**

**（23）禁止使用SELECT \*，只获取必要的字段，需要显示说明列属性**

解读：

a）读取不需要的列会增加CPU、IO、NET消耗

b）不能有效的利用覆盖索引

c）使用SELECT \*容易在增加或者删除字段后出现程序BUG

**（24）禁止使用INSERT INTO t\_xxx VALUES(xxx)，必须显示指定插入的列属性**

解读：容易在增加或者删除字段后出现程序BUG

**（25）禁止使用属性隐式转换**

解读：SELECT uid FROM t\_user WHERE phone=13812345678 会导致全表扫描，而不能命中phone索引，猜猜为什么？（这个线上问题不止出现过一次）

**（26）禁止在WHERE条件的属性上使用函数或者表达式**

解读：SELECT uid FROM t\_user WHERE from\_unixtime(day)>='2017-02-15' 会导致全表扫描

正确的写法是：SELECT uid FROM t\_user WHERE day>= unix\_timestamp('2017-02-15 00:00:00')

**（27）禁止负向查询，以及%开头的模糊查询**

解读：

a）负向查询条件：NOT、!=、<>、!<、!>、NOT IN、NOT LIKE等，会导致全表扫描

b）%开头的模糊查询，会导致全表扫描

**（28）禁止大表使用JOIN查询，禁止大表使用子查询**

解读：会产生临时表，消耗较多内存与CPU，极大影响数据库性能

**（29）禁止使用OR条件，必须改为IN查询**

解读：旧版本Mysql的OR查询是不能命中索引的，即使能命中索引，为何要让数据库耗费更多的CPU帮助实施查询优化呢？

**（30）应用程序必须捕获SQL异常，并有相应处理**

总结：大数据量高并发的互联网业务，极大影响数据库性能的都不让用，不让用哟。

==【完】==

自转到58到家，负责过较长一段时间DBA部门，搭建团队，讨论流程，讨论规范，讨论自动化，尝试平台化，故希望沉淀一些东西。

今天先写到这里，希望大伙有收获，帮转哟。

## 再议数据库军规

原创 2017-02-17 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959910&idx=1&sn=6b6853b70dbbe6d689a12a4a60b84d8b&chksm=bd2d07ba8a5a8eac6783bac951dba345d865d875538755fe665a5daaf142efe670e2c02b7c71&scene=21" \l "#)

上一篇《[58到家数据库30条军规解读](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959906&idx=1&sn=2cbdc66cfb5b53cf4327a1e0d18d9b4a&chksm=bd2d07be8a5a8ea86dc3c04eced3f411ee5ec207f73d317245e1fefea1628feb037ad71531bc&scene=21" \l "wechat_redirect" \t "_blank)》引发了广泛的讨论，某些军规部分同学有疑惑，补充一文说明。

**军规：必须使用UTF8字符集**

和DBA负责人确认后，**纠正为“新库默认使用utf8mb4字符集”。**

这点感谢网友的提醒，utf8mb4是utf8的超集，emoji表情以及部分不常见汉字在utf8下会表现为乱码，故需要升级至utf8mb4。

默认使用这个字符集的原因是：“标准，万国码，无需转码，无乱码风险”，并不“节省空间”。

一个潜在坑：阿里云上RDS服务如果要从utf8升级为utf8mb4，需要重启实例，所以58到家并没有把所有的数据库升级成这个字符集，而是“新库默认使用utf8mb4字符集”。

自搭的Mysql可以完成在线转换，而不需要重启数据库实例。

**军规：数据表、数据字段必须加入中文注释**

这一点应该没有疑问。

不过也有朋友提出，加入注释会方便黑客，建议“注释写在文档里，文档和数据库同步更新”。这个建议根据经验来说是不太靠谱的：

（1）不能怕bug就不写代码，怕黑客就不写注释，对吧？

（2）文档同步更新也不太现实，还是把注释写好，代码可读性做好更可行，互联网公司的文档管理？呆过互联网公司的同学估计都清楚。

**军规：禁止使用存储过程、视图、触发器、Event**

**军规：禁止使用外键，如果有外键完整性约束，需要应用程序控制**

**军规：禁止大表使用JOIN查询，禁止大表使用子查询**

很多网友提出，这些军规不合理，完全做到不可能。

如原文所述，58到家数据库30条军规的背景是“并发量大、数据量大的互联网业务”，这类业务架构设计的重点往往是吞吐量，性能优先（和钱相关的少部分业务是一致性优先），**对数据库性能影响较大的数据库特性较少使用**。这类场景的架构方向是“解放数据库CPU，把复杂逻辑计算放到服务层”，服务层具备更好的扩展性，容易实现“增机器就扩充性能”，数据库擅长存储与索引，勿让数据库背负过重的任务。

关于这个点，再有较真的柳岩小编就不回复了哈，任何事情都没有百分之百，但58到家的数据库使用确实没有存储过程、视图、触发器、外键、用户自定义函数，针对业务特性设计架构，等单库吞吐量到了几千上万，就明白这些军规的重要性啦。

**军规：只允许使用内网域名，而不是ip连接数据库**

这一点应该也没有疑问。

不只是数据库，缓存（memcache、redis）的连接，服务（service）的连接都必须使用内网域名，机器迁移/平滑升级/运维管理…太多太多的好处，如果朋友你还是采用ip直连的，赶紧升级到内网域名吧。

**军规：禁止使用小数存储国币**

有朋友问存储前乘以100，取出后除以100是否可行，个人建议“尽量少的使用除法”。

曾经踩过这样的坑，100元分3天摊销，每天摊销100/3元，结果得到3个33.33。后来实施对账系统，始终有几分钱对不齐，郁闷了很久（不是几分钱的事，是业务方质疑的眼神让研发很不爽），最后发现是除法惹的祸。

解决方案：使用“分”作为单位，这样数据库里就是整数了。

**案例：SELECT uid FROM t\_user WHERE phone=13812345678 会导致全表扫描，而不能命中phone索引**

这个坑大家没踩过么？

phone是varchar类型，SQL语句带入的是整形，故不会命中索引，加个引号就好了：

SELECT uid FROM t\_user WHERE phone=’13812345678’

**军规：禁止使用负向查询NOT、!=、<>、!<、!>、NOT IN、NOT LIKE等，会导致全表扫描**

此军规争议比较大，部分网友反馈不这么做很多业务实现不了，稍微解释一下：

一般来说，WHERE过滤条件不会只带这么一个“负向查询条件”，还会有其他过滤条件，举个例子：查询沈剑已完成订单之外的订单（好拗口）：

SELECT oid FROM t\_order WHERE uid=123 AND status != 1;

订单表5000w数据，但uid=123就会迅速的将数据量过滤到很少的级别（uid建立了索引），此时再接上一个负向的查询条件就无所谓了，扫描的行数本身就会很少。

但如果要查询所有已完成订单之外的订单：

SELECT oid FROM t\_order WHERE status != 1;

这就挂了，立马CPU100%，status索引会失效，负向查询导致全表扫描。

末了，除了《[58到家数据库30条军规解读](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959906&idx=1&sn=2cbdc66cfb5b53cf4327a1e0d18d9b4a&chksm=bd2d07be8a5a8ea86dc3c04eced3f411ee5ec207f73d317245e1fefea1628feb037ad71531bc&scene=21" \l "wechat_redirect" \t "_blank)》中提到的**基础规范、命名规范、表设计规范、字段设计规范、索引设计规范、SQL使用规范**，还有一个**行为规范**的军规：

（31）禁止使用应用程序配置文件内的帐号手工访问线上数据库

（32）**禁止非DBA对线上数据库进行写操作**，修改线上数据需要提交工单，由DBA执行，提交的SQL语句必须经过测试

（33）分配非DBA以只读帐号，必须通过VPN+跳板机访问授权的从库

（34）开发、测试、线上环境隔离

为什么要制定行为规范的军规呢，大伙的公司是不是有这样的情况：

任何研发、测试都有连接线上数据库的帐号？

是不是经常有这类误操作？

（1）本来只想update一条记录，where条件搞错，update了全部的记录

（2）本来只想delete几行记录，结果删多了，四下无人，再insert回去

（3）以为drop的是测试库，结果把线上库drop掉了

（4）以为操作的是分库x，结果SecureCRT开窗口太多，操作成了分库y

（5）写错配置文件，压力测试压到线上库了，生成了N多脏数据

…

无数的事情，结果就是打电话给DBA，让他们帮忙擦屁股。

…

所谓的“业务灵活性”都是扯淡，为什么要有行为规范？不让你带刀，不是限制你，而是保护你的安全。要相信DBA是专业的，让专业的人干专业的事情。别把DBA看做你的对立面，多和他们沟通业务场景，沟通请求读写比，沟通访问模式，他们真的能帮助到你，这是我带DBA团队的一些感触。

**谁都可能删除全库，能找回数据的，真的只有DBA。**

==【完】==

相关文章：[58到家数据库30条军规解读](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959906&idx=1&sn=2cbdc66cfb5b53cf4327a1e0d18d9b4a&chksm=bd2d07be8a5a8ea86dc3c04eced3f411ee5ec207f73d317245e1fefea1628feb037ad71531bc&scene=21" \l "wechat_redirect" \t "_blank)

如果RD心怀内疚，如果DBA深有同感，帮转哟。

## 业界难题-“跨库分页”的四种方案

原创 2017-02-27 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959942&idx=1&sn=e9d3fe111b8a1d44335f798bbb6b9eea&chksm=bd2d075a8a5a8e4cad985b847778aa83056e22931767bb835132c04571b66d5434020fd4147f&scene=21" \l "#)

**一、需求缘起**

**分页需求**

互联网很多业务都有分页拉取数据的需求，例如：

（1）微信消息过多时，拉取第N页消息

（2）京东下单过多时，拉取第N页订单

（3）浏览58同城，查看第N页帖子

这些业务场景对应的消息表，订单表，帖子表分页拉取需求有这样一些特点：

（1）有一个业务主键id, 例如msg\_id, order\_id, tiezi\_id

（2）分页排序是按照非业务主键id来排序的，业务中经常按照时间time来排序order by

在数据量不大时，**可以通过在排序字段time上建立索引，利用SQL提供的offset/limit功能就能满足分页查询需求**：

select \* from t\_msg order by time offset 200 limit 100

select \* from t\_order order by time offset 200 limit 100

select \* from t\_tiezi order by time offset 200 limit 100

此处假设一页数据为100条，均拉取第3页数据。

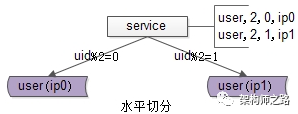
**分库需求**

高并发大流量的互联网架构，一般通过服务层来访问数据库，随着数据量的增大，数据库需要进行水平切分，分库后将数据分布到不同的数据库实例（甚至物理机器）上，以达到降低数据量，增加实例数的扩容目的。

一旦涉及分库，逃不开**“分库依据”patition key**的概念，使用哪一个字段来水平切分数据库呢：大部分的业务场景，会使用业务主键id。

确定了分库依据patition key后，接下来要确定的是**分库算法**：大部分的业务场景，会使用业务主键id取模的算法来分库，这样即能够保证每个库的数据分布是均匀的，又能够保证每个库的请求分布是均匀的，实在是简单实现负载均衡的好方法，此法在互联网架构中应用颇多。

举一个更具体的例子：



用户库user，水平切分后变为两个库，分库依据patition key是uid，分库算法是uid取模：uid%2余0的数据会落到db0，uid%2余1的数据会落到db1。

**问题的提出**

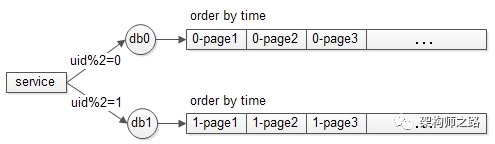
仍然是上述用户库的例子，如果业务要查询“最近注册的第3页用户”，该如何实现呢？单库上，可以

select \* from t\_user order by time offset 200 limit 100

变成两个库后，分库依据是uid，排序依据是time，数据库层失去了time排序的全局视野，数据分布在两个库上，此时该怎么办呢？

如何满足“跨越多个水平切分数据库，且分库依据与排序依据为不同属性，并需要进行分页”的查询需求，实现 select \* from T order by time offset X limit Y的跨库分页SQL，是本文将要讨论的技术问题。

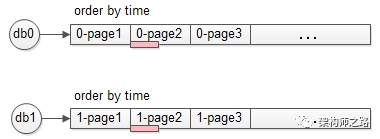
**二、全局视野法**



如上图所述，服务层通过uid取模将数据分布到两个库上去之后，每个数据库都失去了全局视野，数据按照time局部排序之后，不管哪个分库的第3页数据，都不一定是全局排序的第3页数据。

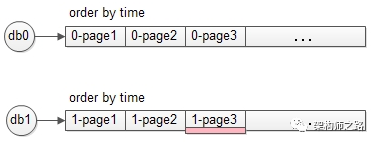
那到底哪些数据才是全局排序的第3页数据呢，暂且分三种情况讨论。

（1）**极端情况**，两个库的数据完全一样



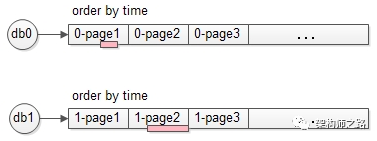
如果两个库的数据完全相同，只需要每个库offset一半，再取半页，就是最终想要的数据（如上图中粉色部分数据）。

（2）**极端情况**，结果数据来自一个库



也可能两个库的数据分布及其不均衡，例如db0的所有数据的time都大于db1的所有数据的time，则可能出现：一个库的第3页数据，就是全局排序后的第3页数据（如上图中粉色部分数据）。

（3）**一般情况**，每个库数据各包含一部分



正常情况下，全局排序的第3页数据，每个库都会包含一部分（如上图中粉色部分数据）。

由于不清楚到底是哪种情况，所以必须每个库都返回3页数据，所得到的6页数据在服务层进行内存排序，得到数据全局视野，再取第3页数据，便能够得到想要的全局分页数据。

再总结一下这个方案的步骤：

（1）将order by time offset X limit Y，改写成order by time offset 0 limit X+Y

（2）服务层将改写后的SQL语句发往各个分库：即例子中的各取3页数据

（3）假设共分为N个库，服务层将得到N\*(X+Y)条数据：即例子中的6页数据

（4）服务层对得到的N\*(X+Y)条数据进行内存排序，内存排序后再取偏移量X后的Y条记录，就是全局视野所需的一页数据

**方案优点**：通过服务层修改SQL语句，扩大数据召回量，能够得到全局视野，业务无损，精准返回所需数据。

**方案缺点**（显而易见）：

（1）每个分库需要返回更多的数据，增大了网络传输量（耗网络）；

（2）除了数据库按照time进行排序，服务层还需要进行二次排序，增大了服务层的计算量（耗CPU）；

（3）最致命的，这个算法随着页码的增大，性能会急剧下降，这是因为SQL改写后每个分库要返回X+Y行数据：返回第3页，offset中的X=200；假如要返回第100页，offset中的X=9900，即每个分库要返回100页数据，数据量和排序量都将大增，性能平方级下降。

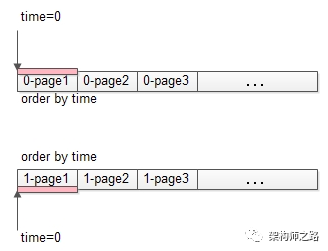
**三、业务折衷法**

“全局视野法”虽然性能较差，但其业务无损，数据精准，不失为一种方案，有没有性能更优的方案呢？

“**任何脱离业务的架构设计都是耍流氓**”，技术方案需要折衷，在技术难度较大的情况下，业务需求的折衷能够极大的简化技术方案。

**业务折衷一：禁止跳页查询**

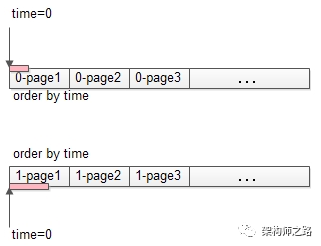
在数据量很大，翻页数很多的时候，很多产品并不提供“直接跳到指定页面”的功能，而只提供“下一页”的功能，这一个小小的业务折衷，就能极大的降低技术方案的复杂度。



如上图，不够跳页，那么第一次只能够查第一页：

（1）将查询order by time offset 0 limit 100，改写成order by time where time>0 limit 100

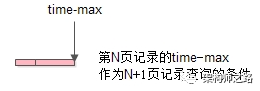
（2）上述改写和offset 0 limit 100的效果相同，都是每个分库返回了一页数据（上图中粉色部分）；



（3）服务层得到2页数据，内存排序，取出前100条数据，作为最终的第一页数据，这个全局的第一页数据，一般来说每个分库都包含一部分数据（如上图粉色部分）；

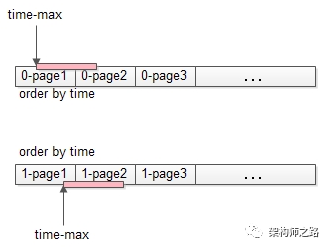
咦，这个方案也需要服务器内存排序，岂不是和“全局视野法”一样么？第一页数据的拉取确实一样，但每一次“下一页”拉取的方案就不一样了。

点击“下一页”时，需要拉取第二页数据，在第一页数据的基础之上，能够找到第一页数据time的最大值：

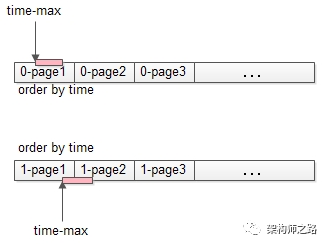


这个上一页记录的time\_max，会作为第二页数据拉取的查询条件：

（1）将查询order by time offset 100 limit 100，改写成order by time where time>$time\_max limit 100



（2）这下不是返回2页数据了（“全局视野法，会改写成offset 0 limit 200”），每个分库还是返回一页数据（如上图中粉色部分）；



（3）服务层得到2页数据，内存排序，取出前100条数据，作为最终的第2页数据，这个全局的第2页数据，一般来说也是每个分库都包含一部分数据（如上图粉色部分）；

如此往复，查询全局视野第100页数据时，不是将查询条件改写为offset 0 limit 9900+100（**返回100页数据**），而是改写为time>$time\_max99 limit 100（**仍返回一页数据**），以保证数据的传输量和排序的数据量不会随着不断翻页而导致性能下降。

**业务折衷二：允许数据精度损失**

“全局视野法”能够返回业务无损的精确数据，在查询页数较大，例如第100页时，会有性能问题，此时业务上是否能够接受，返回的100页不是精准的数据，而允许有一些数据偏差呢？

**数据库分库-数据均衡原理**

使用patition key进行分库，在数据量较大，数据分布足够随机的情况下，各分库所有非patition key属性，在各个分库上的数据分布，统计概率情况是一致的。

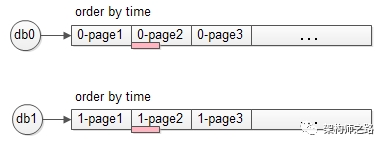
例如，在uid随机的情况下，使用uid取模分两库，db0和db1：

（1）**性别**属性，如果db0库上的男性用户占比70%，则db1上男性用户占比也应为70%

（2）**年龄**属性，如果db0库上18-28岁少女用户比例占比15%，则db1上少女用户比例也应为15%

（3）**时间**属性，如果db0库上每天10:00之前登录的用户占比为20%，则db1上应该是相同的统计规律

…



利用这一原理，要查询全局100页数据，offset 9900 limit 100改写为offset 4950 limit 50，每个分库偏移4950（一半），获取50条数据（半页），得到的数据集的并集，基本能够认为，是全局数据的offset 9900 limit 100的数据，当然，这一页数据的精度，并不是精准的。

根据实际业务经验，用户都要查询第100页网页、帖子、邮件的数据了，这一页数据的精准性损失，业务上往往是可以接受的，但此时技术方案的复杂度便大大降低了，既不需要返回更多的数据，也不需要进行服务内存排序了。

**四、终极武器-二次查询法**

有没有一种技术方案，即能够满足业务的精确需要，无需业务折衷，又高性能的方法呢？这就是接下来要介绍的终极武器：“二次查询法”。

为了方便举例，假设一页只有5条数据，查询第200页的SQL语句为select \* from T order by time offset 1000 limit 5;

**步骤一：查询改写**

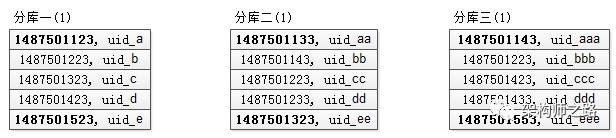
将select \* from T order by time offset 1000 limit 5

改写为select \* from T order by time offset 500 limit 5

并投递给所有的分库，注意，这个offset的500，来自于全局offset的总偏移量1000，除以水平切分数据库个数2。

如果是3个分库，则可以改写为select \* from T order by time offset 333 limit 5

假设这三个分库返回的数据(time, uid)如下：



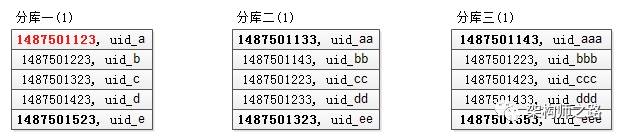
可以看到，每个分库都是返回的按照time排序的一页数据。

**步骤二：找到所返回3页全部数据的最小值**

第一个库，5条数据的time最小值是1487501123

第二个库，5条数据的time最小值是1487501133

第三个库，5条数据的time最小值是1487501143



故，三页数据中，time最小值来自第一个库，time\_min=1487501123，这个过程只需要比较各个分库第一条数据，时间复杂度很低

**步骤三：查询二次改写**

第一次改写的SQL语句是select \* from T order by time offset 333 limit 5

第二次要改写成一个between语句，between的起点是time\_min，between的终点是原来每个分库各自返回数据的最大值：

第一个分库，第一次返回数据的最大值是1487501523

所以查询改写为select \* from T order by time where time between time\_min and 1487501523

第二个分库，第一次返回数据的最大值是1487501323

所以查询改写为select \* from T order by time where time between time\_min and 1487501323

第三个分库，第一次返回数据的最大值是1487501553

所以查询改写为select \* from T order by time where time between time\_min and 1487501553

相对第一次查询，第二次查询条件放宽了，故第二次查询会返回比第一次查询结果集更多的数据，假设这三个分库返回的数据(time, uid)如下：



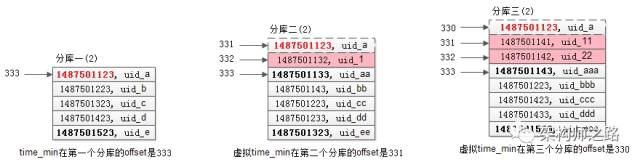
可以看到：

由于time\_min来自原来的分库一，所以分库一的返回结果集和第一次查询相同（所以其实这次访问是可以省略的）；

分库二的结果集，比第一次多返回了1条数据，头部的1条记录（time最小的记录）是新的（上图中粉色记录）；

分库三的结果集，比第一次多返回了2条数据，头部的2条记录（time最小的2条记录）是新的（上图中粉色记录）；

**步骤四：在每个结果集中虚拟一个time\_min记录，找到time\_min在全局的offset**



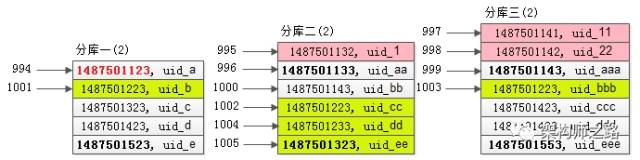
在第一个库中，time\_min在第一个库的offset是333

在第二个库中，(1487501133, uid\_aa)的offset是333（根据第一次查询条件得出的），故虚拟time\_min在第二个库的offset是331

在第三个库中，(1487501143, uid\_aaa)的offset是333（根据第一次查询条件得出的），故虚拟time\_min在第三个库的offset是330

综上，time\_min在全局的offset是333+331+330=994

**步骤五：既然得到了time\_min在全局的offset，就相当于有了全局视野，根据第二次的结果集，就能够得到全局offset 1000 limit 5的记录**



第二次查询在各个分库返回的结果集是有序的，又知道了time\_min在全局的offset是994，一路排下来，容易知道全局offset 1000 limit 5的一页记录（上图中黄色记录）。

是不是非常巧妙？这种方法的**优点**是：可以精确的返回业务所需数据，每次返回的数据量都非常小，不会随着翻页增加数据的返回量。

**不足是**：需要进行两次数据库查询。

**五、总结**

今天介绍了解决“跨N库分页”这一难题的四种方法：

**方法一：全局视野法**

（1）将order by time offset X limit Y，改写成order by time offset 0 limit X+Y

（2）服务层对得到的N\*(X+Y)条数据进行内存排序，内存排序后再取偏移量X后的Y条记录

这种方法随着翻页的进行，性能越来越低。

**方法二：业务折衷法-禁止跳页查询**

（1）用正常的方法取得第一页数据，并得到第一页记录的time\_max

（2）每次翻页，将order by time offset X limit Y，改写成order by time where time>$time\_max limit Y

以保证每次只返回一页数据，性能为常量。

**方法三：业务折衷法-允许模糊数据**

（1）将order by time offset X limit Y，改写成order by time offset X/N limit Y/N

**方法四：二次查询法**

（1）将order by time offset X limit Y，改写成order by time offset X/N limit Y

（2）找到最小值time\_min

（3）between二次查询，order by time between $time\_min and $time\_i\_max

（4）设置虚拟time\_min，找到time\_min在各个分库的offset，从而得到time\_min在全局的offset

（5）得到了time\_min在全局的offset，自然得到了全局的offset X limit Y

==【完，希望大家有收获】==

相关推荐：

[究竟啥才是互联网架构“高可用”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959728&idx=1&sn=933227840ec8cdc35d3a33ae3fe97ec5&chksm=bd2d046c8a5a8d7a13551124af36bedf68f7a6e31f6f32828678d2adb108b86b7e08c678f22f&scene=21" \l "wechat_redirect" \t "_blank)

[究竟啥才是互联网架构“高并发”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959830&idx=1&sn=ce1c5a58caed227d7dfdbc16d6e1cea4&chksm=bd2d07ca8a5a8edc45cc45c4787cc72cf4c8b96fb43d2840c7ccd44978036a7d39a03dd578b5&scene=21" \l "wechat_redirect" \t "_blank)

[100亿数据1万属性数据架构设计](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959855&idx=1&sn=f33abe8ec598c273f29cebb9365ece59&chksm=bd2d07f38a5a8ee58a944507a134e1da1efc3ac9c4d1c4cff261137cd986e51f5fe7cee9de15&scene=21" \l "wechat_redirect" \t "_blank)

[数据库秒级平滑扩容架构方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959883&idx=1&sn=e7df8510c7096a5b069e0f12eaaca010&chksm=bd2d07978a5a8e815c2ae41b16b6b4c579923502fb919008a22bb108a1e920109f25387f8903&scene=21" \l "wechat_redirect" \t "_blank)

[58到家数据库30条军规解读](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959906&idx=1&sn=2cbdc66cfb5b53cf4327a1e0d18d9b4a&chksm=bd2d07be8a5a8ea86dc3c04eced3f411ee5ec207f73d317245e1fefea1628feb037ad71531bc&scene=21" \l "wechat_redirect" \t "_blank)

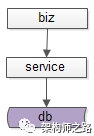
若有收获，帮转哟

## 100亿数据平滑数据迁移,不影响服务

原创 2017-03-23 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959992&idx=1&sn=eb2fbd7d7922db42a593c304e50a65b7&chksm=bd2d07648a5a8e72d489022ec6006274d7e43ab48449b255d5661658c2af8e9221977a9609ed&scene=21" \l "#)

**一、问题的提出**

互联网有很多“数据量较大，并发量较大，业务复杂度较高”的业务场景，其典型系统分层架构如下：



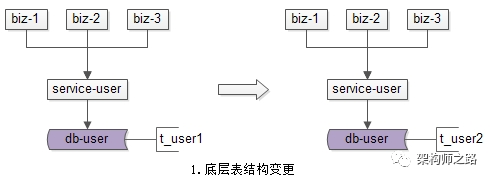
（1）上游是**业务层biz**，实现个性化的业务逻辑

（2）中游是**服务层service**，封装数据访问

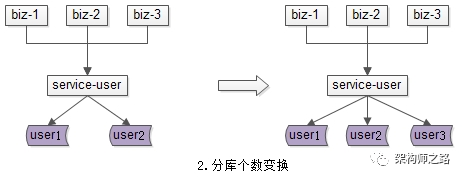
（3）下游是**数据层db**，存储固化的业务数据

服务化分层架构的好处是，服务层屏蔽下游数据层的复杂性，例如缓存、分库分表、存储引擎等存储细节不需要向调用方暴露，而只向上游提供方便的RPC访问接口，当有一些数据层变化的时候，所有的调用方也不需要升级，只需要服务层升级即可。

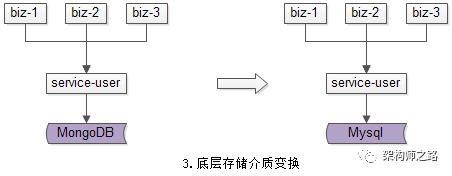
互联网架构，很多时候面临着这样一些需求：



**需求1->底层表结构变更**：数据量非常大的情况下，数据表增加了一些属性，删除了一些属性，修改了一些属性。



**需求2->分库个数变换**：由于数据量的持续增加，底层分库个数非成倍增加。

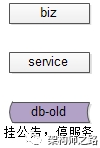


**需求3->底层存储介质变换**：底层存储引擎由一个数据库换为另一个数据库。

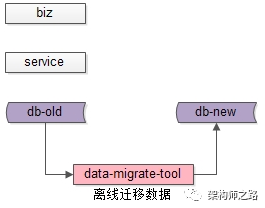
种种需求，都需要进行数据迁移，如何平滑迁移数据，迁移过程不停机，保证系统持续服务，是文本将要讨论的问题。

**二、停机方案**

在讨论平滑迁移数据方案之前，先看下不平滑的停机数据迁移方案，主要分三个步骤。



**步骤一**：**挂**一个类似“为了给广大用户提供更好的服务，服务器会在凌晨0:00-0:400进行停机维护”的**公告**，并在对应时段进行**停机**，这个时段系统没有流量进入。

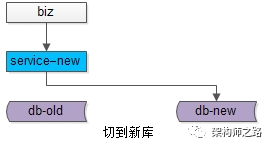


**步骤二**：停机后，研发一个**离线**的数据迁移工具，进行**数据迁移**。针对第一节的三类需求，会分别开发不同的数据迁移工具。

（1）底层表结构变更需求：开发旧表导新表的工具

（2）分库个数变换需求：开发2库导3库的工具

（3）底层存储介质变换需求：开发Mongo导Mysql工具



**步骤三**：**恢复服务**，并将流量切到新库，不同的需求，可能会涉及不同服务升级。

（1）底层表结构变更需求：服务要升级到访问新表

（2）分库个数变换需求：服务不需要升级，只需要改寻库路由配置

（3）底层存储介质变换需求：服务升级到访问新的存储介质

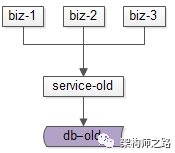
总的来说，停机方案是相对直观和简单的，但对服务的可用性有影响，许多游戏公司的服务器升级，游戏分区与合区，可能会采用类似的方案。

除了影响服务的可用性，这个方案还有一个缺点，就是必须在指定时间完成升级，这个对研发、测试、运维同学来说，**压力会非常大**，一旦出现问题例如数据不一致，必须在规定时间内解决，否则只能回滚。根据经验，**人压力越大越容易出错**，这个缺点一定程度上是致命的。

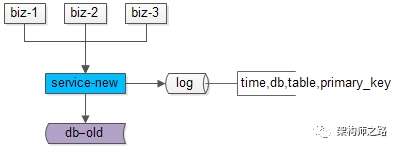
无论如何，停机方案并不是今天要讨论的重点，接下来看一下常见的平滑数据迁移方案。

**三、平滑迁移-追日志法**

平滑迁移方案一，追日志法，这个方案主要分为五个步骤。



数据迁移前，上游业务应用通过旧的服务访问旧的数据。



**步骤一**：服务进行升级，记录“对旧库上的数据修改”的日志（这里的修改，为数据的insert, delete, update），这个日志不需要记录详细数据，主要记录：

（1）被修改的库

（2）被修改的表

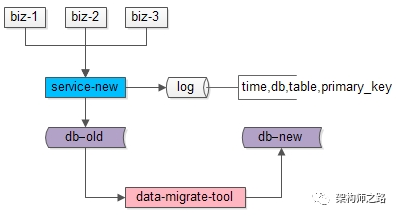
（3）被修改的唯一主键

具体新增了什么行，修改后的数据格式是什么，不需要详细记录。这样的好处是，不管业务细节如何变化，日志的格式是固定的，这样能保证方案的通用性。

这个服务升级风险较小：

（1）写接口是少数接口，改动点较少

（2）升级只是增加了一些日志，对业务功能没有任何影响



**步骤二**：研发一个数据迁移工具，进行数据迁移。这个数据迁移工具和离线迁移工具一样，把旧库中的数据转移到新库中来。

这个小工具的风险较小：

（1）整个过程依然是旧库对线上提供服务

（2）小工具的复杂度较低

（3）任何时间发现问题，都可以把新库中的数据干掉重来

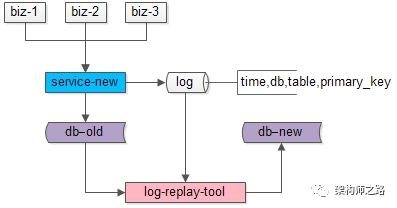
（4）可以限速慢慢迁移，技术同学没有时间压力

**数据迁移完成之后，就能够切到新库提供服务了么？**

答案是否定的，在数据迁移的过程中，旧库依然对线上提供着服务，库中的数据随时可能变化，这个变化并没有反映到新库中来，于是旧库和新库的数据并不一致，所以不能直接切库，需要将数据追平。

哪些数据发生了变化呢？

步骤一中日志里记录的不就是么？



**步骤三**：研发一个读取日志并迁移数据的小工具，要把步骤二迁移数据过程中产生的差异数据追平。这个小工具需要做的是：

（1）读取日志，得到哪个库、哪个表、哪个主键发生了变化

（2）把旧库中对应主键的记录读取出来

（3）把新库中对应主键的记录替换掉

无论如何，原则是数据以旧库为准。

这个小工具的风险也很小：

（1）整个过程依然是旧库对线上提供服务

（2）小工具的复杂度较低

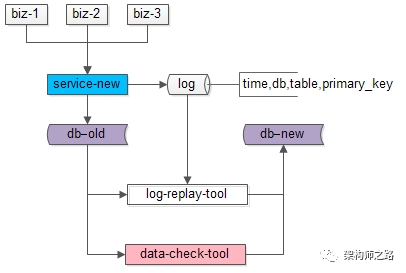
（3）任何时间发现问题，大不了从步骤二开始重来

（4）可以限速慢慢重放日志，技术同学没有时间压力

**日志重放之后，就能够切到新库提供服务了么？**

答案依然是否定的，在日志重放的过程中，旧库中又可能有数据发生了变化，导致数据不一致，所以还是不能切库，需要进一步读取日志，追平记录。可以看到，重放日志追平数据的程序是一个while(1)的程序，新库与旧库中的数据追平也会是一个“无限逼近”的过程。

**什么时候数据会完全一致呢？**



**步骤四**：在持续重放日志，追平数据的过程中，研发一个数据校验的小工具，将旧库和新库中的数据进行比对，直到数据完全一致。

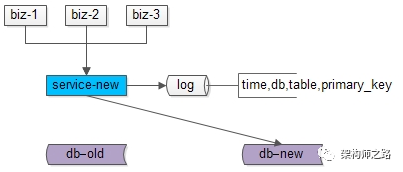
这个小工具的风险依旧很小：

（1）整个过程依然是旧库对线上提供服务

（2）小工具的复杂度较低

（3）任何时间发现问题，大不了从步骤二开始重来

（4）可以限速慢慢比对数据，技术同学没有时间压力



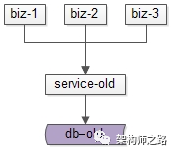
**步骤五**：在数据比对完全一致之后，将流量迁移到新库，新库提供服务，完成迁移。

如果步骤四数据一直是99.9%的一致，不能完全一致，也是正常的，可以做一个秒级的旧库readonly，等日志重放程序完全追上数据后，再进行切库切流量。

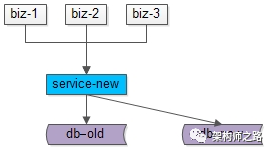
至此，升级完毕，整个过程能够持续对线上提供服务，不影响服务的可用性。

**四、平滑迁移-双写法**

平滑迁移方案二，双写法，这个方案主要分为四个步骤。



数据迁移前，上游业务应用通过旧的服务访问旧的数据。



**步骤一**：服务进行升级，对“对旧库上的数据修改”（这里的修改，为数据的insert, delete, update），在新库上进行相同的修改操作，这就是所谓的“双写”，主要修改操作包括：

（1）旧库与新库的同时insert

（2）旧库与新库的同时delete

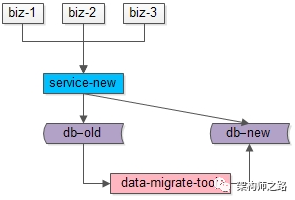
（3）旧库与新库的同时update

由于新库中此时是没有数据的，所以双写旧库与新库中的affect rows可能不一样，不过这完全不影响业务功能，只要不切库，依然是旧库提供业务服务。

这个服务升级风险较小：

（1）写接口是少数接口，改动点较少

（2）新库的写操作执行成功与否，对业务功能没有任何影响



**步骤二**：研发一个数据迁移工具，进行数据迁移。这个数据迁移工具在本文中已经出现第三次了，把旧库中的数据转移到新库中来。

这个小工具的风险较小：

（1）整个过程依然是旧库对线上提供服务

（2）小工具的复杂度较低

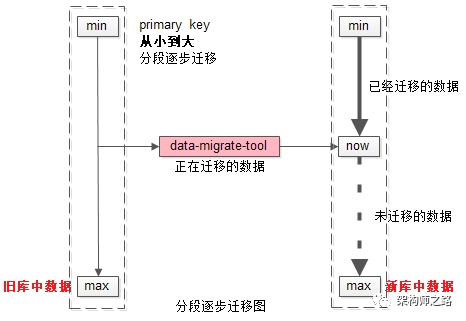
（3）任何时间发现问题，都可以把新库中的数据干掉重来

（4）可以限速慢慢迁移，技术同学没有时间压力

**数据迁移完成之后，就能够切到新库提供服务了么？**

答案是肯定的，因为前置步骤进行了双写，所以理论上数据迁移完之后，新库与旧库的数据应该完全一致。

由于迁移数据的过程中，旧库新库双写操作在同时进行，**怎么证明数据迁移完成之后数据就完全一致了呢？**



如上图所示：

（1）左侧是旧库中的数据，右侧是新库中的数据

（2）按照primary key从min到max的顺序，分段，限速进行数据的迁移，假设已经迁移到now这个数据段

数据迁移过程中的修改操作分别讨论：

（1）假设迁移过程中进行了一个**双insert操作**，旧库新库都插入了数据，数据一致性没有被破坏

（2）假设迁移过程中进行了一个**双delete操作**，这又分为两种情况

         （2.1）假设这delete的数据属于[min,now]范围，即已经完成迁移，则旧库新库都删除了数据，数据一致性没有被破坏

         （2.2）假设这delete的数据属于[now,max]范围，即未完成迁移，则旧库中删除操作的affect rows为1，新库中删除操作的affect rows为0，但是数据迁移工具在后续数据迁移中，并不会将这条旧库中被删除的数据迁移到新库中，所以数据一致性仍没有被破坏

（3）假设迁移过程中进行了一个**双update操作**，可以认为update操作是一个delete加一个insert操作的复合操作，所以数据仍然是一致的

除非除非除非，在一种非常非常非常极限的情况下：

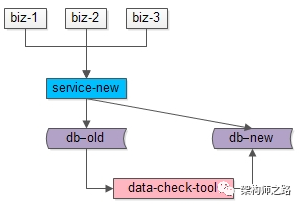
（1）date-migrate-tool**刚好**从旧库中将某一条数据X取出

（2）在X插入到新库中之前，旧库与新库中**刚好**对X进行了双delete操作

（3）date-migrate-tool再将X插入到新库中

这样，会出现新库比旧库多出一条数据X。

但无论如何，为了保证数据的一致性，切库之前，还是需要进行数据校验的。



**步骤三**：在数据迁移完成之后，需要使用数据校验的小工具，将旧库和新库中的数据进行比对，完全一致则符合预期，如果出现步骤二中的极限不一致情况，则以旧库中的数据为准。

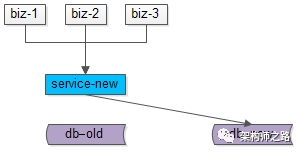
这个小工具的风险依旧很小：

（1）整个过程依然是旧库对线上提供服务

（2）小工具的复杂度较低

（3）任何时间发现问题，大不了从步骤二开始重来

（4）可以限速慢慢比对数据，技术同学没有时间压力



**步骤四**：数据完全一致之后，将流量切到新库，完成平滑数据迁移。

至此，升级完毕，整个过程能够持续对线上提供服务，不影响服务的可用性。

**五、总结**

针对互联网很多“数据量较大，并发量较大，业务复杂度较高”的业务场景，在

（1）底层表结构变更

（2）分库个数变换

（3）底层存储介质变换

的众多需求下，需要进行数据迁移，完成**“平滑迁移数据，迁移过程不停机，保证系统持续服务”**有两种常见的解决方案。

**追日志法**，五个步骤：

（1）**服务进行升级**，记录“对旧库上的数据修改”的日志

（2）研发一个数据迁移**小工具**，进行数据迁移

（3）研发一个读取日志**小工具**，追平数据差异

（4）研发一个数据比对**小工具**，校验数据一致性

（5）流量切到新库，完成平滑迁移

**双写法**，四个步骤：

（1）**服务进行升级**，记录“对旧库上的数据修改”进行新库的双写

（2）研发一个数据迁移**小工具**，进行数据迁移

（3）研发一个数据比对**小工具**，校验数据一致性

（4）流量切到新库，完成平滑迁移

文章比较长，希望大家有收获。

帮转哟。

==【完】==

推荐阅读：

[业界难题-“跨库分页”的四种方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959942&idx=1&sn=e9d3fe111b8a1d44335f798bbb6b9eea&chksm=bd2d075a8a5a8e4cad985b847778aa83056e22931767bb835132c04571b66d5434020fd4147f&scene=21" \l "wechat_redirect" \t "_blank)

[数据库秒级平滑扩容架构方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959883&idx=1&sn=e7df8510c7096a5b069e0f12eaaca010&chksm=bd2d07978a5a8e815c2ae41b16b6b4c579923502fb919008a22bb108a1e920109f25387f8903&scene=21" \l "wechat_redirect" \t "_blank)

## mysql-proxy数据库中间件架构

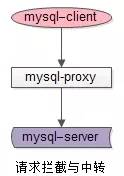
原创 2017-05-20 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960133&idx=1&sn=07acfeac5306f92f81dfccf4c880da4f&chksm=bd2d06998a5a8f8fc8c3bbb2c2ea01825ebe6576a35aa4d9789774dba0f7964baf4edf584f1b&scene=21" \l "#)

### ****一、mysql-proxy简介****

mysql-proxy是mysql官方提供的mysql中间件**服务**，上游可接入若干个mysql-client，后端可连接若干个mysql-server。

它**使用mysql协议**，任何使用mysql-client的上游无需修改任何代码，即可迁移至mysql-proxy上。

mysql-proxy最基本的用法，就是作为一个请求拦截，请求中转的中间层：



进一步的，mysql-proxy可以分析与修改请求。**拦截查询和修改结果，需要通过编写Lua脚本来完成**。

mysql-proxy允许用户指定Lua脚本对请求进行拦截，对请求进行分析与修改，它还允许用户指定Lua脚本对服务器的返回结果进行修改，加入一些结果集或者去除一些结果集均可。

所以说，根本上，mysql-proxy是一个官方提供的框架，具备良好的扩展性，可以用来完成：

* sql拦截与修改
* 性能分析与监控
* 读写分离
* 请求路由
* ...

这个框架提供了6个hook点，能够让用户能够动态的介入到client与server中的通讯中去。

### ****二、mysql-proxy架构与原理****

如“简介”中所述，mysql-proxy向用户提供了6个hook点，让用户实现Lua脚本来完成各种功能，这些hook点是以函数的形式提供的，用户可以实现这些函数，在不同事件、不同操作发生时，做我们期望的事情。

**connect\_server()**

mysql-client向proxy发起连接时，proxy会调用这个函数。用户可以实现该函数，来做一些负载均衡的事情，例如选择将要连向那个mysql-server。假设有多个mysql-server后端，而用户又没有实现这个函数，proxy默认采用轮询（round-robin）策略。

**read\_handshake()**

mysql-server向proxy返回“初始握手信息”时，proxy会调用这个函数。用户可以实现这个函数，来做更多的权限验证工作。

**read\_auth()**

mysql-client向proxy发送认证报文（user\_name, password,database）时，proxy会调用这个函数。

**read\_auth\_result()**

mysql-server向proxy返回认证结果时，proxy会调用这个函数。

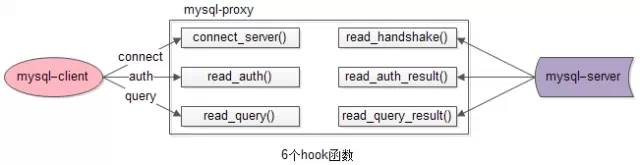
**read\_query()**

认证完成后，mysql-client每次经过proxy向mysql-server发送query报文时，proxy会调用这个函数。用户如果要拦截请求，就可以模拟mysql-server直接返回了，当然用户亦可以实现各种策略，修改请求，路由请求等各种不同的业务逻辑。

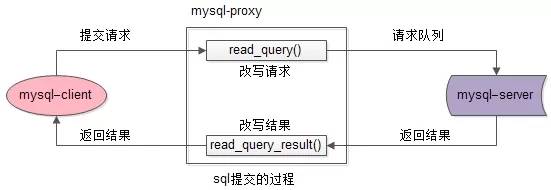
**read\_query\_result()**

认证完成后，mysql-server每次经过proxy向mysql-client返回query结果时，proxy会调用这个函数。需要注意，如果用户没有显示实现read\_query()函数，则read\_query\_result()函数是不会被调用的。用户可以在此处实现各种合并策略，或者对结果集进行修改。

下图是一个各hook函数的触发架构图，箭头方向表示触发时机：



可以发现，最重要的两个函数其实是read\_query()和read\_query\_result()，各种sql的改写与结果集的改写逻辑，都是在这两个函数中实现的，更细节的query过程如下图：



**三、mysql-proxy典型应用**

**案例一： sql时间统计分析**

假设mysql-client提交的原sql为：

XYZ;

proxy可以在read\_query()里将其改写为：

SELECT NOW();

XYZ;

SELECT NOW();

这样在返回结果集时，就可以在应用层对sql时间进行记录，以方便统计分析。

**案例二：sql性能统计分析**

假设mysql-client提交的原sql为：

XYZ;

proxy可以在read\_query()里将其改写为：

XYZ;

EXPLAIN XYZ;

这样在返回结果集时，就可以在应用层对sql性能进行记录，以方便统计分析。

**需要强调的是**，这两个案例，由于proxy在read\_query()时对sql进行了改写，故在read\_query\_result()时，mysql-server其实返回了比原请求更多的信息，proxy一定要将多余的信息去掉，再返回mysql-client。多说一句，可以加入一个唯一ID，来对请求sql和返回结果进行配对。

**案例三：读写分离**

mysql-proxy启动时，通过参数即可配置后端mysql-server是主server还是read-only，无需修改任何代码：

shell> mysql-proxy \

--proxy-backend-addresses=10.0.1.2:3306 \

--proxy-read-only-backend-addresses=10.0.1.3:3306

注意，这里的两个mysql-server为**主从架构**。

**案例四：性能水平扩展**

mysql-proxy启动时，通过参数配置多个后端，即可实现性能的水平扩展，无需修改任何代码：

shell> mysql-proxy \

--proxy-backend-addresses=10.0.1.2:3306 \

--proxy-backend-addresses=10.0.1.3:3306

注意，这里的两个mysql-server为**主主架构**，如果不做特殊修改，负载均衡策略为round-robin。

**四、mysql-proxy其他问题**

**提问：Lua脚本引入的额外开销有多大？**

**官网回答**：Lua很快，对于大部分应用来说，额外开销很小，原始包（raw packet）开销大概在400微秒左右。

**楼主**：这，，，我不太相信。

**提问：mysql-proxy和mysql-server可以部署在一台机器上么？**

**官网回答**：proxy单独部署也可以，和mysql部署在同一台机器上也可以。相比mysql而言，proxy不怎么占CPU和内存，其性能损耗可以忽略不计。

**楼主**：这，，，性能损耗可以忽略，这我也不太信。

**提问：proxy可以处理SSL连接么？proxy不会获取和保存我的明文密码吧？**

**官网回答**：作为中间人，不能处理加密信息。不会获取密码，也获取不到。mysql协议不允许密码以明文传输，传输的都是加密后的密文。

**提问：在Lua脚本里可以使用LuaSocket，连缓存，连其他服务么？**

**官网回答**：理论上可以。但是，大哥，你确定要这样做么，强烈不建议这样。

==【完】==

相关推荐：

[用uid分库，uname上的查询怎么办？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960032&idx=1&sn=1f818d415842d544dc33ab78e8adffea&chksm=bd2d073c8a5a8e2a38e7534d57ca0cad613893b9c3b578a220c4beef5d7495c5a1569cf4768e&scene=21" \l "wechat_redirect" \t "_blank)

[100亿数据平滑数据迁移,不影响服务](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959992&idx=1&sn=eb2fbd7d7922db42a593c304e50a65b7&chksm=bd2d07648a5a8e72d489022ec6006274d7e43ab48449b255d5661658c2af8e9221977a9609ed&scene=21" \l "wechat_redirect" \t "_blank)

[业界难题-“跨库分页”的四种方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959942&idx=1&sn=e9d3fe111b8a1d44335f798bbb6b9eea&chksm=bd2d075a8a5a8e4cad985b847778aa83056e22931767bb835132c04571b66d5434020fd4147f&scene=21" \l "wechat_redirect" \t "_blank)

[58到家数据库30条军规解读](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959906&idx=1&sn=2cbdc66cfb5b53cf4327a1e0d18d9b4a&chksm=bd2d07be8a5a8ea86dc3c04eced3f411ee5ec207f73d317245e1fefea1628feb037ad71531bc&scene=21" \l "wechat_redirect" \t "_blank)

## 用uid分库，uname上的查询怎么办？

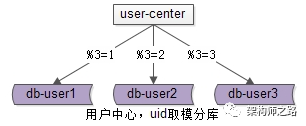
原创 2017-04-14 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960032&idx=1&sn=1f818d415842d544dc33ab78e8adffea&chksm=bd2d073c8a5a8e2a38e7534d57ca0cad613893b9c3b578a220c4beef5d7495c5a1569cf4768e&scene=21" \l "#)

1分钟系列

**【缘起】**

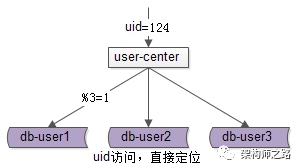
**用户中心**是几乎每一个公司必备的基础服务，用户注册、登录、信息查询与修改都离不开用户中心。

当数据量越来越大时，需要多用户中心进行水平切分。最常见的**水平切分方式**，按照uid取模分库：



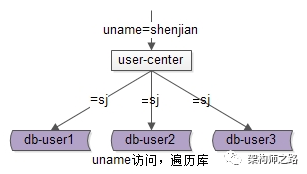
通过uid取模，将数据分布到多个数据库实例上去，**提高服务实例个数，降低单库数据量**，以达到扩容的目的。

水平切分之后：



uid属性上的查询可以直接路由到库，如上图，假设访问uid=124的数据，取模后能够直接定位db-user1。

对于uname上的查询，就不能这么幸运了：



uname上的查询，如上图，假设访问uname=shenjian的数据，由于不知道数据落在哪个库上，往往需要遍历所有库**【扫全库法】**，当分库数量多起来，性能会显著降低。

用uid分库，如何高效实现上的查询，是本文将要讨论的问题。

**【索引表法】**

**思路**：uid能直接定位到库，uname不能直接定位到库，如果通过uname能查询到uid，问题解决

**解决方案**：

1）建立一个索引表记录uname->uid的映射关系

2）用uname来访问时，先通过索引表查询到uid，再定位相应的库

3）索引表属性较少，可以容纳非常多数据，一般不需要分库

4）如果数据量过大，可以通过uname来分库

**潜在不足**：多一次数据库查询，性能下降一倍

**【缓存映射法】**

**思路**：访问索引表性能较低，把映射关系放在缓存里性能更佳

**解决方案**：

1）uname查询先到cache中查询uid，再根据uid定位数据库

2）假设cache miss，采用扫全库法获取uname对应的uid，放入cache

3）uname到uid的映射关系不会变化，映射关系一旦放入缓存，不会更改，无需淘汰，缓存命中率超高

4）如果数据量过大，可以通过name进行cache水平切分

**潜在不足**：多一次cache查询

**【uname生成uid】**

**思路**：不进行远程查询，由uname直接得到uid

**解决方案**：

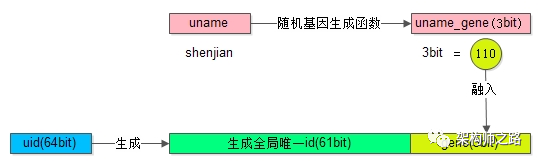
1）在用户注册时，设计函数uname生成uid，uid=f(uname)，按uid分库插入数据

2）用uname来访问时，先通过函数计算出uid，即uid=f(uname)再来一遍，由uid路由到对应库

**潜在不足**：该函数设计需要非常讲究技巧，有uid生成冲突风险

**【uname基因融入uid】**

**思路**：不能用uname生成uid，可以从uname抽取“基因”，融入uid中



假设分8库，采用uid%8路由，潜台词是，uid的最后3个bit决定这条数据落在哪个库上，这3个bit就是所谓的“基因”。

**解决方案**：

1）在用户注册时，设计函数uname生成3bit基因，uname\_gene=f(uname)，如上图粉色部分

2）同时，生成61bit的全局唯一id，作为用户的标识，如上图绿色部分

3）接着把3bit的uname\_gene也作为uid的一部分，如上图屎黄色部分

4）生成64bit的uid，由id和uname\_gene拼装而成，并按照uid分库插入数据

5）用uname来访问时，先通过函数由uname再次复原3bit基因，uname\_gene=f(uname)，通过uname\_gene%8直接定位到库

**【总结】**

**业务场景**：用户中心，数据量大，通过uid分库后，通过uname路由不到库

**解决方案**：

1）扫全库法：遍历所有库

2）索引表法：数据库中记录uname->uid的映射关系

3）缓存映射法：缓存中记录uname->uid的映射关系

4）uname生成uid

5）uname基因融入uid

==【完】==

推荐阅读：

[一分钟掌握数据库垂直拆分](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959773&idx=1&sn=7e4ad0dcd050f6662dfaf39d9de36f2c&chksm=bd2d04018a5a8d17b92098b4840aac23982e32d179cdd957e4c55011f6a08f6bd31f9ba5cfee&scene=21" \l "wechat_redirect" \t "_blank)

[数据库秒级平滑扩容架构方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959883&idx=1&sn=e7df8510c7096a5b069e0f12eaaca010&chksm=bd2d07978a5a8e815c2ae41b16b6b4c579923502fb919008a22bb108a1e920109f25387f8903&scene=21" \l "wechat_redirect" \t "_blank)

[58到家数据库30条军规解读](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959906&idx=1&sn=2cbdc66cfb5b53cf4327a1e0d18d9b4a&chksm=bd2d07be8a5a8ea86dc3c04eced3f411ee5ec207f73d317245e1fefea1628feb037ad71531bc&scene=21" \l "wechat_redirect" \t "_blank)

## 一分钟了解四层/七层反向代理

原创 2017-05-19 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960131&idx=1&sn=a3bbcbe03f9e12d32ba751ce6ffae067&chksm=bd2d069f8a5a8f895fed39cad842f6f5a390bb18493f964b910270128f19f0b8af1d1f30b5c7&scene=21" \l "#)

上一篇文章《[session一致性架构设计实践](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960128&idx=1&sn=8e0e409b10ab9db549432af461385314&chksm=bd2d069c8a5a8f8ab5cdee602d4062bbdbb25da290668515d36682afa854e374d2a5ff02004b&scene=21" \l "wechat_redirect" \t "_blank)》，对于其中的第三种“反向代理hash法”，不少同学留言问：

* 什么是四层反向代理hash
* 什么是七层反向代理hash
* 中间还有三层那里去了
* ...

今天花几分钟简单和大家解释一下。

场景：**访问**用户通过proxy请求**被访问**的真实服务器

路径：用户 -> proxy -> real-server

**什么是代理？**

回答：[proxy]代表[访问用户]，此时proxy是**代理**。

例如：

在家访问xxoo网站，不希望xxoo网站trace到我们的真实ip，于是就找一个proxy，通过proxy来访问，此时proxy代表用户，**网站以为proxy的ip就是用户的ip**。

**什么是反向代理？**

回答：[proxy]代表[被访问的服务器]，此时proxy是**反向代理**。

例如：

web-server希望对用户屏蔽高可用、屏蔽web-server扩展、web-server内网ip等细节，于是就找了一个proxy隔在中间，此时proxy代表web-server集群，**用户以为proxy的ip就是被访问web-server的ip**（web-server是集群，具体访问了哪个web-server，用户不知道），由于web-server集群有多台，此时反向代理服务器要具备**负载均衡**的功能。

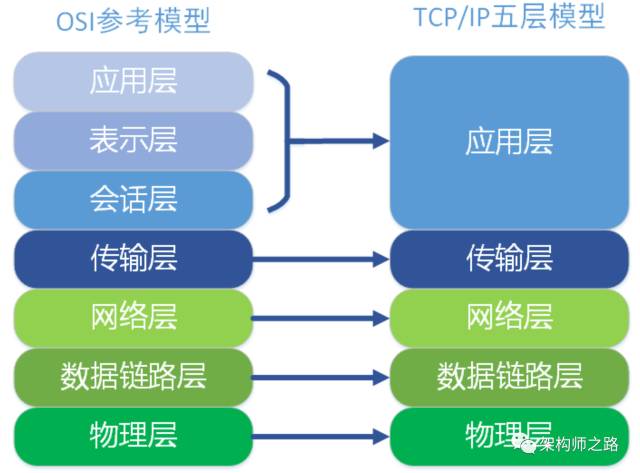
**一般怎么做反向代理，负载均衡？**

回答：nginx/apache，lvs，F5

**什么是四层（转发/交换），什么是七层（转发/交换）？**

回答：这个是来源于OSI七层模型

大学“计算机网络”课程，之前都是用这个七层模型，新版教程用TCP/IP五层模型，这两个模型之间有一个对应关系如下：



可以看到，**四层**是指传输层，**七层**是指应用层。

更具体的，对应到nginx反向代理hash：

* **四层**：根据用户ip+port来做hash
* **七层**：根据http协议中的某些属性来做hash

**为什么中间少了几层？**

回答：OSI应用层、表示层、会话层合并到TCP/IP的应用层啦。

**上面有四层，七层，那有没有二层，三层呢？**

回答：有

* 二层：根据**数据链路层**MAC地址完成数据交换
* 三层：根据**网络层**IP地址完成数据交换

希望解答了大伙之前的一些疑问，希望这一分钟没有浪费，如果有描述不准确的地方，欢迎指正。

==【完】==

相关推荐：

[session一致性架构设计实践](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960128&idx=1&sn=8e0e409b10ab9db549432af461385314&chksm=bd2d069c8a5a8f8ab5cdee602d4062bbdbb25da290668515d36682afa854e374d2a5ff02004b&scene=21" \l "wechat_redirect" \t "_blank)

[DNS在架构设计中的巧用](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960118&idx=1&sn=767e038cb4378be1c88dd42569a9264f&chksm=bd2d06ea8a5a8ffc7c69c71c2153b10565ecf72b6957c5667fbf9099c9f5478111132e967a35&scene=21" \l "wechat_redirect" \t "_blank)

[跨公网调用的大坑与架构优化方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960105&idx=1&sn=0069f2264e227e86a63ee50a4899e0a7&chksm=bd2d06f58a5a8fe33271ed3a378932ad023af7a9004d7fdb44e30a53d53928f984f293a41256&scene=21" \l "wechat_redirect" \t "_blank)

## 连接池原来这么简单（一分钟系列）

原创 2017-01-05 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959821&idx=1&sn=4ede084b05ce81a9a5ddb87ec62434bb&chksm=bd2d07d18a5a8ec7726619dbb9f1e99df8239ebd07d5f01d748e01c08dd543f0a434945301c6&scene=21" \l "#)

应网友要求，写一写连接池实现细节。

**一、如何通过连接访问下游**

工程架构中有很多访问下游的需求，下游包括但不限于**服务/数据库/缓存**，其通讯步骤是为：

（1）与下游建立一个连接

（2）通过这个连接，收发请求

（3）交互结束，关闭连接，释放资源

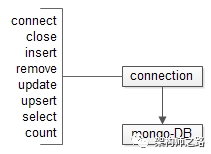
这个连接是什么呢，通过连接怎么调用下游接口？服务/数据库/缓存，**官方会提供不同语言的Driver、Document、DemoCode来教使用方建立连接与调用接口**，以MongoDB的C++官方Driver API为例（伪代码）：

DBClientConnection\* c = new DBClientConnection();

c->connect(“127.0.0.1:8888”);

c->insert(“db.s”, BSON(”shenjian”));

c->close();



这个DBClientConnection就是一个与MongoDB的连接，官方Driver通过它提供了若干API，让用户可以对MongoDB进行连接，增删查改，关闭的操作，从而实现不同的业务逻辑。

**二、为什么需要连接池**

当并发量很低的时候，上述伪代码没有任何问题，但**当服务单机QPS达到几百、几千的时候**，建立连接connect和销毁连接close就会成为瓶颈，此时该如何优化？

结论也很简单，服务启动的时候，先建立好若干连接Array[DBClientConnection]，当有请求过来的时候，从Array中取出一个，执行下游操作，执行完再放回，从而避免反复的建立和销毁连接，以提升性能。

而**这个对Array[DBClientConnection]进行维护的数据结构，就是连接池**。有了连接池之后，数据库操作的伪代码变为：

DBClientConnection\* c = ConnectionPool::GetConnection();

c->insert(“db.s”, BSON(”shenjian”));

ConnectionPool::FreeConnection(c);

**三、连接池核心接口与实现**

通过上面的讨论，可以看到连接池ConnectionPool主要有三个核心接口：

（1）**Init**：初始化好Array[DBClientConnection]，这个接口只在服务启动时调用一次

（2）**GetConnection**：请求每次需要访问数据库时，不是connect一个连接，而是通过连接池的这个接口来拿

（3）**FreeConnection**：请求每次访问完数据库时，不是close一个连接，而是把这个连接放回连接池

连接池核心数据结构：

（1）**连接数组**Array DBClientConnection [N]

（2）**互斥锁数组**Array lock[N]

连接池核心接口实现：

Init(){

 for i = 1 to N {

  Array DBClientConnection [i] = new();

  Array DBClientConnection [i]->connect();

  Array lock[i] = 0;

 }

}

**说明**：把所有连接和互斥锁初始化

GetConnection()

 for i = 1 to N {

  if(Array lock[i] == 0){

   Array lock[i] = 1;

   return Array DBClientConnection[i];

   }

 }

}

**说明**：找一个可用的连接，锁住，并返回连接

FreeConnection(c)

 for i = 1 to N {

 if(Array DBClientConnection [i] == c){

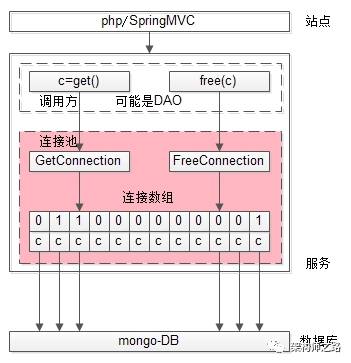
   Array lock[i] = 0;

   }

  }

}

**说明**：找到连接，把锁释放



可以发现，简单的连接池管理并不是很复杂，基本原理即如上所述。

**四、未尽事宜**

上述伪代码忽略了一些细节，在实现连接池中是需要考虑的：

（1）如果连接全部被占用，是返回失败，还是让上游等待

（2）需要实施连接可用性检测

（3）为了让调用方更友好，可能还需要包装一层DAO层，让“连接”这个东西对调用方都是黑盒的

（4）通过freeArray，connectionMap可以让取连接和放回连接都达到O(1)时间复杂度

（5）可以通过hash实现id串行化

（6）负载均衡、故障转移、服务自动扩容都可以在这一层实现

希望这一分钟大家有收获。

==【完】==

相关阅读：

[“id串行化”到底是怎么实现的？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959668&idx=1&sn=d4c13071ab619c891efa03ba4e05334e&chksm=bd2d04a88a5a8dbe4ea606c4970be60c5f5c00f3e4cceeb91b26c8975b45678440c516012e79&scene=21" \l "wechat_redirect" \t "_blank)

[消息“时序”与“一致性”为何这么难？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959666&idx=1&sn=fbdce26e40296d5b30f70915c4b9eb0a&chksm=bd2d04ae8a5a8db868df14904d0a1ddb4eb4d8411442f5529f9760e7c62fca586cfa6f3bd200&scene=21" \l "wechat_redirect" \t "_blank)

## 一分钟理解负载LoadAverage

原创 2017-02-20 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959912&idx=1&sn=ebe3320f91f11df7e1d0f54ec639f31c&chksm=bd2d07b48a5a8ea2356749aecef4d7d1a9e9ca75e02b51da255393396165e50a0a6aa2ea80b5&scene=21" \l "#)

**一、什么是Load Average？**

系统负载（System Load）是系统CPU繁忙程度的度量，即有多少进程在等待被CPU调度（进程等待队列的长度）。

平均负载（Load Average）是一段时间内系统的平均负载，这个一段时间一般取1分钟、5分钟、15分钟。

**二、如何查看Load？**

**top，uptime，w**等命令都可以查看系统负载：

[shenjian@dev02 ~]$ uptime

13:53:39 up 10 days, 2:15, 1 user, load average: 1.5, 2.5, 5.5

如上所示，dev02机器**1分钟平均负载，5分钟平均负载，15分钟平均负载**分别是1.5、2.5、5.5

**三、Load的数值是什么含义？**

把**CPU**比喻成一条（单核）**马路**，**进程任务**比喻成马路上跑着的**汽车**，**Load**则表示**马路的繁忙程度**。

Load小于1：不堵车，汽车在马路上跑得游刃有余：

http://mmbiz.qpic.cn/mmbiz/YrezxckhYOyUwPsibCnqOe4SSBdyZeSA3aPLSKPdMKnPiasOgES7uEJRIFJ0nUFxQ2sIcTZ4Y6B5zbxEVrdvE2aQ/640?wxfrom=5&wx_lazy=1

[Load<1，单核]

Load等于1：马路已无额外的资源跑更多的汽车了：

http://mmbiz.qpic.cn/mmbiz/YrezxckhYOyUwPsibCnqOe4SSBdyZeSA3PDxBbiaic4JRqXLQuB2OC5uq3UPJNabMJpMxVJOnqD0mjTia58LQaboJQ/640?wxfrom=5&wx_lazy=1

[Load==1，单核]

Load大于1：汽车都堵着等待进入马路：

http://mmbiz.qpic.cn/mmbiz/YrezxckhYOyUwPsibCnqOe4SSBdyZeSA3HnHkVjDWUicfI5qT6rbBl3jWzmOzrBDRVcUkkHBx0TtayUR5GLbDXsg/640?wxfrom=5&wx_lazy=1

[Load>1，单核]

如果有两个CPU，则表示有两条马路，此时即使Load大于1也不代表有汽车在等待：

http://mmbiz.qpic.cn/mmbiz/YrezxckhYOyUwPsibCnqOe4SSBdyZeSA3BmqCyvoMqYkKI54VMfT8DkvFEAgangheXNLSrq66JGxXbujVlZxKibg/640?wxfrom=5&wx_lazy=1

[Load==2，双核，没有等待]

**四、什么样的Load值得警惕（单核）？**

Load < 0.7时：系统很闲，马路上没什么车，要考虑多部署一些服务

0.7 < Load < 1时：系统状态不错，马路可以轻松应对

Load == 1时：系统马上要处理不多来了，赶紧找一下原因

Load > 5时：马路已经非常繁忙了，进入马路的每辆汽车都要无法很快的运行

**五、不同Load值说明什么问题？**

结合具体情况具体分析：

1）1分钟Load>5，5分钟Load<1，15分钟Load<1：短期内繁忙，中长期空闲，初步判断是一个**“抖动”或者是“拥塞前兆”**

2）1分钟Load>5，5分钟Load>1，15分钟Load<1：短期内繁忙，中期内紧张，很可能是一个**“拥塞的开始”**

3）1分钟Load>5，5分钟Load>5，15分钟Load>5：短中长期都繁忙，系统**“正在拥塞”**

4）1分钟Load<1，5分钟Load>1，15分钟Load>5：短期内空闲，中长期繁忙，不用紧张，系统**“拥塞正在好转”**

**六、Load总结**

http://mmbiz.qpic.cn/mmbiz/YrezxckhYOyUwPsibCnqOe4SSBdyZeSA3aPLSKPdMKnPiasOgES7uEJRIFJ0nUFxQ2sIcTZ4Y6B5zbxEVrdvE2aQ/640?wxfrom=5&wx_lazy=1[Load<1，单核]

http://mmbiz.qpic.cn/mmbiz/YrezxckhYOyUwPsibCnqOe4SSBdyZeSA3PDxBbiaic4JRqXLQuB2OC5uq3UPJNabMJpMxVJOnqD0mjTia58LQaboJQ/640?wxfrom=5&wx_lazy=1[Load==1，单核]

http://mmbiz.qpic.cn/mmbiz/YrezxckhYOyUwPsibCnqOe4SSBdyZeSA3HnHkVjDWUicfI5qT6rbBl3jWzmOzrBDRVcUkkHBx0TtayUR5GLbDXsg/640?wxfrom=5&wx_lazy=1[Load>1，单核]

http://mmbiz.qpic.cn/mmbiz/YrezxckhYOyUwPsibCnqOe4SSBdyZeSA3BmqCyvoMqYkKI54VMfT8DkvFEAgangheXNLSrq66JGxXbujVlZxKibg/640?wxfrom=5&wx_lazy=1[Load==2，双核]

希望上面一幅图对大家理解Load Average有帮助，赶快uptime一下，看一下自己系统的负载吧。

==【完】==

相关文章：

[1分钟1副图看懂单机/集群/热备/磁盘阵列](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400720056&idx=1&sn=10c796793e27d475603f97b23e137c12&scene=21" \l "wechat_redirect" \t "_blank)

[1分钟awk够用，收藏后备用](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400743254&idx=1&sn=147abc381ccd61e28f52699735c8748e&scene=21" \l "wechat_redirect" \t "_blank)

[1分钟perl够用，收藏后备用](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400764946&idx=1&sn=4efab1953696c2e21071e58a74da9b82&scene=21" \l "wechat_redirect" \t "_blank)

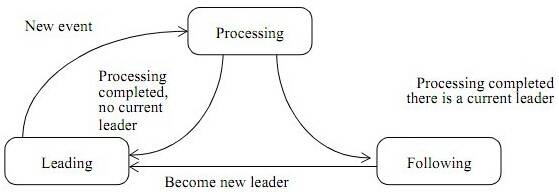
[1分钟sed够用，收藏后备用](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403340380&idx=1&sn=9b4ced5a19f60b0cebf6fd582f1d52ae&scene=21" \l "wechat_redirect" \t "_blank)

[1分钟写好连接池](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959821&idx=1&sn=4ede084b05ce81a9a5ddb87ec62434bb&chksm=bd2d07d18a5a8ec7726619dbb9f1e99df8239ebd07d5f01d748e01c08dd543f0a434945301c6&scene=21" \l "wechat_redirect" \t "_blank)

帮忙**转发**一下吧。

## 1分钟了解Leader-Follower线程模型

原创 2017-02-21 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959914&idx=1&sn=3600fc162970b39afe25d65bee10cd2e&chksm=bd2d07b68a5a8ea03c2b1372345991b74aa412b7fd96cc82e9ed0eb38e8dd481bf5d8b7b631c&scene=21" \l "#)



上图就是L/F多线程模型的状态变迁图，共6个关键点：

（1）线程有3种状态：领导leading，处理processing，追随following

（2）假设共N个线程，其中只有1个leading线程（等待任务），x个processing线程（处理），余下有N-1-x个following线程（空闲）

（3）有**一把锁**，谁抢到就是leading

（4）事件/任务来到时，leading线程会对其进行处理，从而转化为processing状态，**处理完成之后，又转变为following**

（5）丢失leading后，following会尝试抢锁，抢到则变为leading，否则保持following

（6）following不干事，就是抢锁，力图成为leading

优点：**不需要消息队列**

适用场景：**线程能够很快的完成工作任务**

有人说“并发量大时，L/F的锁容易成为系统瓶颈，需要引入一个消息队列解决。”

此观点不对，一个消息队列，其仍是临界资源，仍需要一把锁来保证互斥，只是锁竞争从leading移到了消息队列上，此时消息队列仅仅只能起到消息缓冲的作用。

**根本解决方案是降低锁粒度**（例如多个队列）。

F-L线程模型，可以考虑使用哟？

==【完】==

相关文章：

[1分钟1副图看懂单机/集群/热备/磁盘阵列](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400720056&idx=1&sn=10c796793e27d475603f97b23e137c12&scene=21" \l "wechat_redirect" \t "_blank)

[1分钟awk够用，收藏后备用](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400743254&idx=1&sn=147abc381ccd61e28f52699735c8748e&scene=21" \l "wechat_redirect" \t "_blank)

[1分钟perl够用，收藏后备用](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400764946&idx=1&sn=4efab1953696c2e21071e58a74da9b82&scene=21" \l "wechat_redirect" \t "_blank)

[1分钟sed够用，收藏后备用](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403340380&idx=1&sn=9b4ced5a19f60b0cebf6fd582f1d52ae&scene=21" \l "wechat_redirect" \t "_blank)

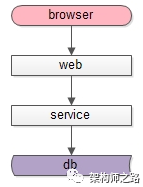
[1分钟理解负载LoadAverage](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959912&idx=1&sn=ebe3320f91f11df7e1d0f54ec639f31c&chksm=bd2d07b48a5a8ea2356749aecef4d7d1a9e9ca75e02b51da255393396165e50a0a6aa2ea80b5&scene=21" \l "wechat_redirect" \t "_blank)

[1分钟写好连接池](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959821&idx=1&sn=4ede084b05ce81a9a5ddb87ec62434bb&chksm=bd2d07d18a5a8ec7726619dbb9f1e99df8239ebd07d5f01d748e01c08dd543f0a434945301c6&scene=21" \l "wechat_redirect" \t "_blank)

帮忙**转发**一下吧。

## 库存扣多了，到底怎么整

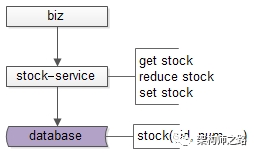
原创 2017-06-14 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960197&idx=1&sn=2e5c17d521772d28d39f31af5d22b34a&chksm=bd2d06598a5a8f4f9de2da89ba8fab711823935442fc632b65d461c923852485ff392c987568&scene=21" \l "#)



业务复杂、数据量大、并发量大的业务场景下，典型的互联网架构，一般会分为这么几层：

* 调用层，一般是处于端上的browser或者APP
* 站点层，一般是拼装html或者json返回的web-server层
* 服务层，一般是提供RPC调用接口的service层
* 数据层，提供固化数据存储的db

对于库存业务，一般有个库存服务，提供库存的**查询、扣减、设置**等RPC接口：



* **库存查询**，stock-service本质上执行的是

select num from stock where sid=$sid

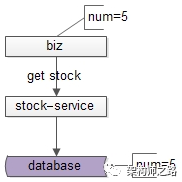
* **库存扣减**，stock-service本质上执行的是

update stock set num=num-$reduce where sid=$sid

* **库存设置**，stock-service本质上执行的是

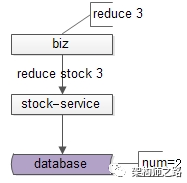
update stock set num=$num\_new where sid=$sid

用户下单前，一般会对库存进行查询，有足够的存量才允许扣减：



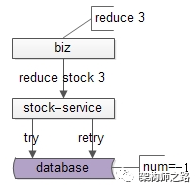
如上图所示，通过查询接口，得到库存是5。

用户下单时，接着会对库存进行扣减：



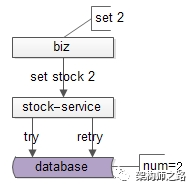
如上图所示，购买3单位的商品，通过扣减接口，最终得到库存是2。

希望设计往往有**容错机制**，例如“重试”，如果通过扣减接口来修改库存，在重试时，可能会得到错误的数据，导致重复扣减：



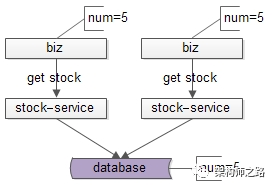
如上图所示，如果数据库层面有重试容错机制，可能导致一次扣减执行两次，最终得到一个负数的错误库存。

重试导致错误的**根本原因**，是因为“扣减”操作是一个**非幂等**的操作，不能够重复执行，改成设置操作则不会有这个问题：



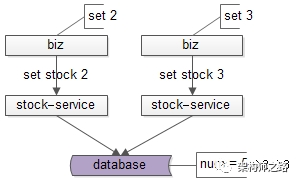
如上图所示，同样是购买3单位的商品，通过设置库存操作，即使有重试容错机制，也不会得到错误的库存，设置库存是一个**幂等**操作。

在并发量很大的情况下，还会有其他的问题：



如上图所示，两个并发的操作，查询库存，都得到了库存是5。

接下来用户发生了**并发**的购买动作（秒杀类业务特别容易出现）：



如上图所示：

* 用户1购买了3个库存，于是库存要设置为2
* 用户2购买了2个库存，于是库存要设置为3
* 这两个设置库存的接口并发执行，**库存会先变成2，再变成3，导致数据不一致**（实际卖出了5件商品，但库存只扣减了2，最后一次设置库存会覆盖和掩盖前一次并发操作）

其**根本原因**是，设置操作发生的时候，没有检查库存与查询出来的库存有没有变化，理论上：

* 库存为5时，用户1的库存设置才能成功
* 库存为5时，用户2的库存设置才能成功

实际执行的时候：

* 库存为5，用户1的set stock 2确实应该成功
* 库存变为2了，用户2的set stock 3应该失败掉

升级修改很容易，将库存设置接口，stock-service上执行的：

update stock set num=$y where sid=$sid

升级为：

update stock set num=$num\_new where sid=$sid and num=$num\_old

这正是大家常说的“Compare And Set”（CAS），是一种常见的降低读写锁冲突，保证数据一致性的方法。

**总结**

在业务复杂，数据量大，并发量大的情况下，库存扣减容易引发数据的不一致，常见的优化方案有两个：

* 调用“设置库存”接口，能够保证数据的**幂等性**
* 在实现“**设置库存**”接口时，需要加上原有库存的比较，才允许设置成功，能解决高并发下库存扣减的一致性问题

希望大伙有收获。

## 库存扣减还有这么多方案？

原创 2017-06-15 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960200&idx=1&sn=fdec629caceee07b3946b6c338b8ceb7&chksm=bd2d06548a5a8f424dd32be960222edf5cecc3c1e5a8fcbb4cfff35e7da6787e702861131597&scene=21" \l "#)

昨天一篇《[库存扣多了，到底怎么整](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960197&idx=1&sn=2e5c17d521772d28d39f31af5d22b34a&chksm=bd2d06598a5a8f4f9de2da89ba8fab711823935442fc632b65d461c923852485ff392c987568&scene=21" \l "wechat_redirect" \t "_blank)》，核心观点是：

* 用“设置库存”替代“扣减库存”，以保证**幂等性**
* 使用CAS乐观锁，在“设置库存”时加上原始库存的比对，**避免数据不一致**

文章非常多朋友留言发表观点，“架构师之路”能引发不少同学思考，甚是欣慰。

原以为两个核心观点应该是没有疑义的，结果很多朋友说方案不好，今天交流下部分回复的方案，个人的一些看法。

**留言一**

是否能使用

update stock set num=num-$count where sid=$sid and stock>=$count;

的方式扣减库存？

**回答**：这个方案无法保证幂等性，有可能出现重复扣减。

**留言二**

把库存放到reids里，利用redis的事务性来扣减库存。

**分析**：

**redis是如何实现事务操作的？**

本质也是乐观锁。

在redis客户端执行：

$num = GET key

$num = $num - $count

SET key $num

在并发量大的时候，会遇到和《[库存扣多了，到底怎么整](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960197&idx=1&sn=2e5c17d521772d28d39f31af5d22b34a&chksm=bd2d06598a5a8f4f9de2da89ba8fab711823935442fc632b65d461c923852485ff392c987568&scene=21" \l "wechat_redirect" \t "_blank)》文章中一样的并发一致性问题。

redis的WATCH和EXEC可以提供类似事务的机制：

* WATCH观察key是否被改动
* 如果提交时key被改动，EXEC将返回null，表示事务失败

上面保证一致性的库存扣减可能类似于这样执行：

WATCH key

$num = GET key

$num = $num - $count

MULTI

SET key $num

EXEC

在WATCH之后，EXEC执行之前，如果key的值发生变化，则EXEC会失败。

redis的WATCH为何能够保证事务性，本质上，它使用的就是乐观锁CAS机制。

大部分情况下，redis不同的客户端会访问不同的key，所以WATCH碰撞的概率会比较小，在秒杀的业务场景，即使使用WATCH，调用侧仍然需要重试。

在CAS机制这一点上，redis和mysql相比没有额外的优势。

redis的**性能之所以高**，还是redis内存访问与mysql数据落盘的差异导致的。内存访问的不足是，数据具备“易失性”，如果重启，可能导致数据的丢失。当然redis也可以固化数据，难道每次都刷盘？redis真心没法当作mysql用。

最后，redis用单线程来避免物理锁，但mysql多线程也有多线程并发的优势。

**回答**：可以使用redis的事务性扣减库存，但在CAS机制上比mysql没有优势，高性能是因为其内存存储的原因，带来的副作用是数据有丢失风险，具体怎么用，还得结合业务折衷（任何脱离业务的架构设计都是耍流氓）。

**柳岩三**

支持幂等能否使用客户端token，业务流水？

能否使用时间戳，版本号来保证一致性？

**回答**：可以。

**留言四**

能否使用队列，在数据库侧串行执行，降低锁冲突？

**回答**：可以。

**留言五**

能否使用事务？

**回答**：容易死锁，吞吐量很低，不建议。

**留言六**

能否使用分布式锁解决，例如setnx, mc, zookeeper？

**回答**：可以，但吞吐量真的高么。

**留言六**

文章重点讲了幂等性和一致性，没有深入展开讲高吞吐，利用缓存抗读请求，利用水平扩展增加性能是提升吞吐量的根本方案。

**回复**：很中肯。

留言1-6代表了评论的多个观点，由于时间有限，《[库存扣多了，到底怎么整](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960197&idx=1&sn=2e5c17d521772d28d39f31af5d22b34a&chksm=bd2d06598a5a8f4f9de2da89ba8fab711823935442fc632b65d461c923852485ff392c987568&scene=21" \l "wechat_redirect" \t "_blank)》许多地方没有讲清楚，大伙见谅。

如果本文让你对redis的事务操作有个重新认识，帮忙点赞和转发哟。

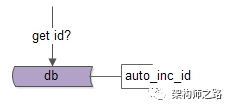
## 浅谈CAS在分布式ID生成方案上的应用

原创 2017-06-17 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960223&idx=1&sn=121716347174eedd6636b1c2c0b96047&chksm=bd2d06438a5a8f55d35add7dc99940c280264fa7caeacb21f7dc79884d00c050d0367a745f87&scene=21" \l "#)

近几篇文章聊CAS被骂得较多，今天还是聊CAS，谈谈CAS在一种“分布式ID生成方案”上的应用。

所谓“分布式ID生成方案”，是指在分布式环境下，生成全局唯一ID的方法。

可以**利用DB自增键(auto inc id)来生成全局唯一ID**，插入一条记录，生成一个ID：



这个方案利用了数据库的单点特性，其优点为：

* 无需写额外代码
* 全局唯一
* **绝对**递增
* 递增ID的步长确定

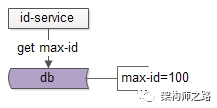
其不足为：

* 需要做数据库HA，保证生成ID的高可用
* 数据库中记录数较多
* **生成ID的性能，取决于数据库插入性能**

优化方案为：

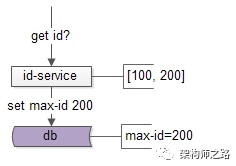
* 利用双主保证高可用
* 定期删除数据
* **增加一层服务，采用批量生成的方式降低数据库的写压力，提升整体性能**

增加服务后，DB中只需保存当前最大的ID即可，在服务启动初始化的过程中，首先拉取当前的max-id：



select max\_id from T;

然后批量获取一批ID，放到id-servcie内存里，并将max-id写回数据库：



update T set max\_id=200;

这样，id-service就拿到了[100, 200]这**一批ID**，上游在获取ID时，不用每次都插入数据库，而是分配完100个ID后，再修改max-id的值，这样分配ID的整体性能就增加了100倍。

这个方案的优点：

* 数据库只保存一条记录
* 性能极大增强

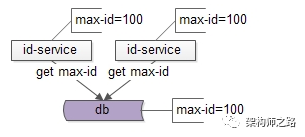
其不足为：

* 如果id-service重启，可能内存会有一段已经申请的ID没有分配出去，导致**ID空洞**，当然，这不是一个严重的问题
* 服务没有做HA，无法保证高可用

优化方案为：

* 冗余服务，做集群保证高可用

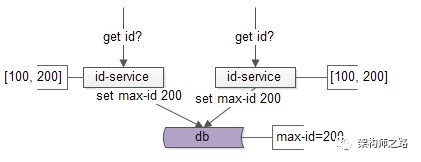
冗余了服务后，多个服务在启动过程中，进行ID批量申请时，可能由于并发导致数据不一致：



select max\_id from T;

如上图所示，两个id-service在启动的过程中，同时拿到了max-id为100。

两个id-service同时对数据库的max-id进行写回：



update T set max\_id=200;

写回max-id成功后，这**两个id-service都以为自己拿到了[100,200]这一批ID**，导致集群会生成重复的ID。

问题发生的原因，是**并发写回时，没有对max-id的初始值进行比对**：

id-service1写回max-id=200成功的条件是，max-id必须等于100

id-service2写回max-id=200成功的条件是，max-id也必须等于100

id-service1写回时，max-id是100，理应写回成功

id-service2写回时，max-id已经被改成了200，不应该写回成功

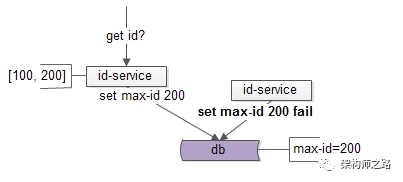
只要实施CAS乐观锁，在写回时对max-id的初始条件进行比对，就能避免数据的不一致，写回SQL由：

update T set max\_id=200;

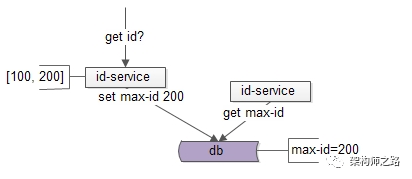
升级为：

update T set max\_id=200 where max\_id=100;

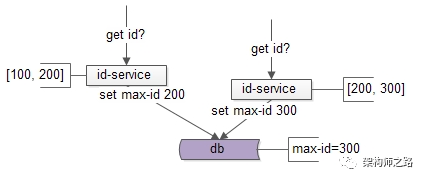
这样，id-service2写回时，就会失败：



失败后，id-service2要再次查询max-id：



此时max-id已经变为200，于是id-service2获取到了[200, 300]这一批ID，并将max-id=300写回：



update t set max\_id=300 where max\_id=200;

写回成功。

这种方案的好处是：

* 能够通过水平扩展的方式，达到分布式ID生成服务的无限性能
* 使用CAS简洁的保证不会生成重复的ID

其不足为：

* 由于有多个service，生成的ID **不是绝对递增**的，而是**趋势递增**的

本文介绍了CAS在分布式ID生成方案上的一种应用，更多的分布式ID生成方案，请参考《[细聊分布式ID生成器架构](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "wechat_redirect" \t "_blank)》。

==【完】==

看了《[库存扣多了，到底怎么整](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960197&idx=1&sn=2e5c17d521772d28d39f31af5d22b34a&chksm=bd2d06598a5a8f4f9de2da89ba8fab711823935442fc632b65d461c923852485ff392c987568&scene=21" \l "wechat_redirect" \t "_blank)》与《[库存扣减多种方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960200&idx=1&sn=fdec629caceee07b3946b6c338b8ceb7&chksm=bd2d06548a5a8f424dd32be960222edf5cecc3c1e5a8fcbb4cfff35e7da6787e702861131597&scene=21" \l "wechat_redirect" \t "_blank)》两篇文章的评论，还挺受挫的，也挺委屈的。描述了A方案和B方案，也从来没说C方案和D方案不行，好友网友反问“为什么不用C”“为什么不用D”“误导”“取关”。

只想说，时间有限，经验有限，2个小时的时间，真的只能聊1-2个技术点。

最后想起一个段子：

code review 100行代码，总能挑出100个毛病

code review 10000行代码，却只是说，代码写得不错

末了，如《[库存扣多了，到底怎么整](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960197&idx=1&sn=2e5c17d521772d28d39f31af5d22b34a&chksm=bd2d06598a5a8f4f9de2da89ba8fab711823935442fc632b65d461c923852485ff392c987568&scene=21" \l "wechat_redirect" \t "_blank)》与《[库存扣减多种方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960200&idx=1&sn=fdec629caceee07b3946b6c338b8ceb7&chksm=bd2d06548a5a8f424dd32be960222edf5cecc3c1e5a8fcbb4cfff35e7da6787e702861131597&scene=21" \l "wechat_redirect" \t "_blank)》两篇文章中很多读者的评论所述，CAS方法会遇到ABA问题，下一篇文章聊聊ABA问题，以及优化方案。

本文有收获的话，帮转下，给作者一些鼓励，谢谢。

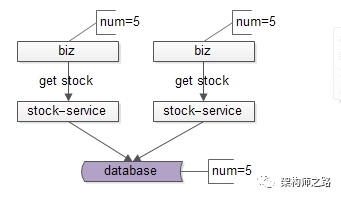
## CAS下ABA问题及优化方案

原创 2017-06-21 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960240&idx=1&sn=18c4ce7c3baf6705940847c1c9bf49e3&chksm=bd2d066c8a5a8f7a27acb5603611b5cbcf6396a8a5e5d06571dcd41deaa64e615b097e84fef3&scene=21" \l "#)

**一、并发业务场景**

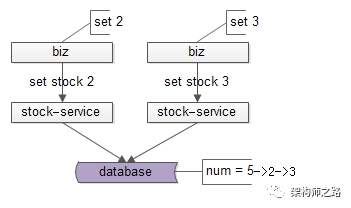
库存业务，stock(sid, num)，其中：

* sid为库存id
* num为库存值



如上图所示，两个并发的查询库存操作，同时从数据库都得到了库存是5。

接下来用户发生了并发的库存扣减动作：



如上图所示：

* 用户1购买了3个库存，于是库存要设置为2
* 用户2购买了2个库存，于是库存要设置为3

这两个设置库存的接口并发执行，库存会先变成2，再变成3，导致数据不一致（实际卖出了5件商品，但库存只扣减了2，最后一次设置库存会覆盖和掩盖前一次并发操作）

**二、不一致原因分析**

出现数据不一致的根本原因，是设置操作发生的时候，没有检查库存与查询出来的库存有没有变化，理论上：

* 仅库存为5的时候，用户1的库存设置2才能成功
* 仅库存为5的时候，用户2的库存设置3才能成功

实际执行的时候：

* 库存为5，用户1的set stock 2确实应该成功
* 库存变为2了，用户2的set stock 3应该失败掉

**三、CAS优化**

大家常说的“Compare And Set”（CAS），是一种常见的降低读写锁冲突，保证数据一致性的乐观锁机制。

针对上述库存扣减的例子，CAS升级很容易，将库存设置接口执行的SQL：

update stock set num=$num\_new where sid=$sid

升级为：

update stock set num=$num\_new where sid=$sid **and num=$num\_old**

即可。

**四、什么是ABA问题**

CAS乐观锁机制确实能够提升吞吐，并保证一致性，但在极端情况下可能会出现ABA问题。

**什么是ABA问题？**

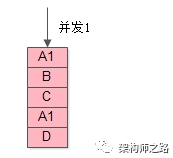
考虑如下操作：

* 并发1（上）：获取出数据的初始值是A，后续计划实施CAS乐观锁，期望数据仍是A的时候，修改才能成功
* 并发2：将数据修改成B
* 并发3：将数据修改回A
* 并发1（下）：CAS乐观锁，检测发现初始值还是A，进行数据修改

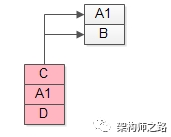
上述并发环境下，并发1在修改数据时，虽然还是A，但已经不是初始条件的A了，中间发生了A变B，B又变A的变化，**此A已经非彼A**，数据却成功修改，可能导致错误，这就是CAS引发的所谓的ABA问题。

库存操作，出现ABA问题并不会对业务产生影响。

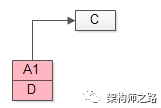
再看一个堆栈操作的例子：



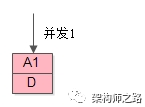
并发1（上）：读取栈顶的元素为“A1”



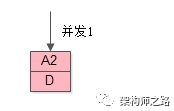
并发2：进行了2次出栈



并发3：又进行了1次出栈



并发1（下）：实施CAS乐观锁，发现栈顶还是“A1”，于是修改为A2



此时会出现**系统错误**，因为此“A1”非彼“A1”

**五、ABA问题的优化**

ABA问题导致的原因，是CAS过程中只简单进行了“值”的校验，再有些情况下，“值”相同不会引入错误的业务逻辑（例如库存），有些情况下，“值”虽然相同，却已经不是原来的数据了。

**优化方向**：CAS不能只比对“值”，还必须确保的是原来的数据，才能修改成功。

**常见实践**：“版本号”的比对，一个数据一个版本，版本变化，即使值相同，也不应该修改成功。

库存的并发读写例子，引入版本号的具体实践如下：

（1）库存表由

stock(sid, num)

升级为

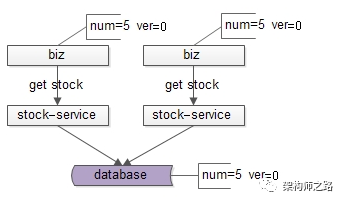
stock(sid, num, **version**)

（2）查询库存时同时查询版本号

select num from stock where sid=$sid

升级为

select num, **version** from stock where sid=$sid



假设有并发操作，都会将版本号查询出来

（3）设置库存时，必须版本号相同，并且版本号要修改

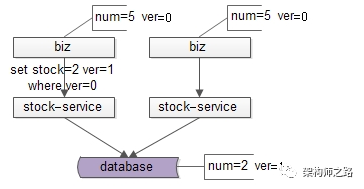
旧版本**“值”比对**CAS

update stock set num=$num\_new where sid=$sid and **num=$num\_old**

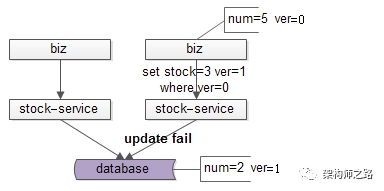
升级为**“版本号”比对**CAS

update stock set num=$num\_new, **version=$version\_new**

 where sid=$sid and **version=$version\_old**



此时假设有并发操作，第一个操作，比对版本号成功，于是把库存和版本号都进行了修改。



同时存在的第二个并发操作，比对版本号发生了变化，也是库存应该修改失败。

**六、总结**

* select&set业务场景，在并发时会出现一致性问题
* 基于“值”的CAS乐观锁，可能导致ABA问题
* CAS乐观锁，必须保证修改时的“此数据”就是“彼数据”，应该由“值”比对，优化为“版本号”比对

相关文章：

[库存扣多了，到底怎么整 | 架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960197&idx=1&sn=2e5c17d521772d28d39f31af5d22b34a&chksm=bd2d06598a5a8f4f9de2da89ba8fab711823935442fc632b65d461c923852485ff392c987568&scene=21" \l "wechat_redirect" \t "_blank)

[库存扣减还有这么多方案？ | 架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960200&idx=1&sn=fdec629caceee07b3946b6c338b8ceb7&chksm=bd2d06548a5a8f424dd32be960222edf5cecc3c1e5a8fcbb4cfff35e7da6787e702861131597&scene=21" \l "wechat_redirect" \t "_blank)

[浅谈CAS在分布式ID生成方案上的应用 | 架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960223&idx=1&sn=121716347174eedd6636b1c2c0b96047&chksm=bd2d06438a5a8f55d35add7dc99940c280264fa7caeacb21f7dc79884d00c050d0367a745f87&scene=21" \l "wechat_redirect" \t "_blank)

如果了解到了ABA问题，帮忙转发哈。

## 深入浅出搜索架构引擎、方案与细节（上）

原创 2017-02-14 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959895&idx=1&sn=de25ce2544c088ff9be0b93fd3ea4d15&chksm=bd2d078b8a5a8e9d5ae4339a683d3f980ff2994f3c10c4081c7bab7f0d77f37521de95e974bf&scene=21" \l "#)

**一、缘起**

《[100亿数据1万属性数据架构设计](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959855&idx=1&sn=f33abe8ec598c273f29cebb9365ece59&chksm=bd2d07f38a5a8ee58a944507a134e1da1efc3ac9c4d1c4cff261137cd986e51f5fe7cee9de15&scene=21" \l "wechat_redirect" \t "_blank)》文章发布后，不少朋友对58同城自研搜索引擎E-search比较感兴趣，故专门撰文体系化的聊聊搜索引擎，从宏观到细节，希望把逻辑关系讲清楚，内容比较多，分上下两期。

主要内容如下，本篇（上）会重点介绍前三章：

（1）**全网搜索引擎架构与流程**

（2）**站内搜索引擎架构与流程**

（3）**搜索原理、流程与核心数据结构**

（4）流量数据量由小到大，搜索方案与架构变迁

（5）数据量、并发量、策略扩展性及架构方案

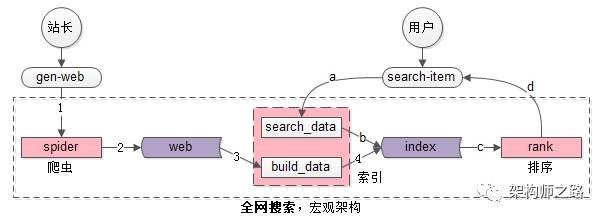
（6）实时搜索引擎核心技术

可能99%的同学不实施搜索引擎，但本文一定对你有帮助。

**二、全网搜索引擎架构与流程**

全网搜索的宏观架构长啥样？

全网搜索的宏观流程是怎么样的？



全网搜索引擎的宏观架构如上图，核心子系统主要分为三部分（粉色部分）：

（1）**spider爬虫系统**

（2）**search&index建立索引与查询索引系统**，这个系统又主要分为两部分：

一部分用于生成索引数据build\_index

一部分用于查询索引数据search\_index

（3）**rank打分排序系统**

核心数据主要分为两部分（紫色部分）：

（1）web网页库

（2）index索引数据

全网搜索引擎的业务特点决定了，这是一个“写入”和“检索”完全分离的系统：

**【写入】**

**系统组成**：由spider与search&index两个系统完成

**输入**：站长们生成的互联网网页

**输出**：正排倒排索引数据

**流程**：如架构图中的1，2，3，4

（1）spider把互联网网页抓过来

（2）spider把互联网网页存储到网页库中（这个对存储的要求很高，要存储几乎整个“万维网”的镜像）

（3）build\_index从网页库中读取数据，完成分词

（4）build\_index生成倒排索引

**【检索】**

**系统组成**：由search&index与rank两个系统完成

**输入**：用户的搜索词

**输出**：排好序的第一页检索结果

**流程**：如架构图中的a，b，c，d

（a）search\_index获得用户的搜索词，完成分词

（b）search\_index查询倒排索引，获得“字符匹配”网页，这是初筛的结果

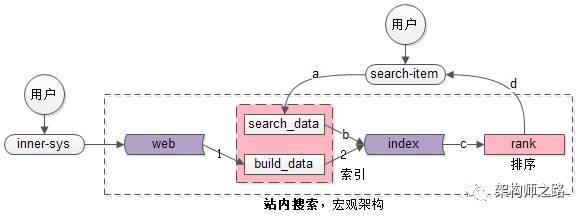
（c）rank对初筛的结果进行打分排序

（d）rank对排序后的第一页结果返回

**三、站内搜索引擎架构与流程**

做全网搜索的公司毕竟是少数，绝大部分公司要实现的其实只是一个站内搜索，站内搜索引擎的宏观架构和全网搜索引擎的宏观架构有什么异同？

以58同城100亿帖子的搜索为例，站内搜索系统架构长啥样？站内搜索流程是怎么样的？



站内搜索引擎的宏观架构如上图，与全网搜索引擎的宏观架构相比，差异只有写入的地方：

（1）**全网搜索需要spider要被动去抓取数据**

（2）**站内搜索是内部系统生成的数据**，例如“发布系统”会将生成的帖子主动推给build\_data系统

看似“很小”的差异，架构实现上**难度**却差很多：全网搜索如何“实时”发现“全量”的网页是非常困难的，而站内搜索容易实时得到全部数据。

对于spider、search&index、rank三个系统：

（1）spider和search&index是相对工程的系统

（2）rank是和业务、策略紧密、算法相关的系统，搜索体验的差异主要在此，而业务、策略的优化是需要时间积累的，这里的启示是：

a）Google的体验比Baidu好，根本在于前者rank牛逼

b）国内互联网公司（例如360）短时间要搞一个体验超越Baidu的搜索引擎，是很难的，真心需要时间的积累

**四、搜索原理与核心数据结构**

什么是正排索引？

什么是倒排索引？

搜索的过程是什么样的？

会用到哪些算法与数据结构？

前面的内容太宏观，为了照顾大部分没有做过搜索引擎的同学，数据结构与算法部分从正排索引、倒排索引一点点开始。

**提问：什么是正排索引（forward index）？**

**回答**：由key查询实体的过程，是正排索引。

用户表：t\_user(uid, name, passwd, age, sex)，由uid查询整行的过程，就是正排索引查询。

网页库：t\_web\_page(url, page\_content)，由url查询整个网页的过程，也是正排索引查询。

网页内容分词后，page\_content会对应一个**分词后的集合list<item>**。

简易的，正排索引可以理解为Map<url, list<item>>，能够由网页快速（时间复杂度O(1)）找到内容的一个数据结构。

**提问：什么是倒排索引（inverted index）？**

**回答**：由item查询key的过程，是倒排索引。

对于网页搜索，倒排索引可以理解为Map<item, list<url>>，能够由查询词快速（时间复杂度O(1)）找到包含这个查询词的网页的数据结构。

举个例子，假设有3个网页：

url1 -> “我爱北京”

url2 -> “我爱到家”

url3 -> “到家美好”

这是一个**正排索引**Map<url, page\_content>。

分词之后：

url1 -> {我，爱，北京}

url2 -> {我，爱，到家}

url3 -> {到家，美好}

这是一个**分词后的正排索引**Map<url, list<item>>。

**分词后倒排索引**：

我 -> {url1, url2}

爱 -> {url1, url2}

北京 -> {url1}

到家 -> {url2, url3}

美好 -> {url3}

由检索词item快速找到包含这个查询词的网页Map<item, list<url>>就是倒排索引。

正排索引和倒排索引是spider和build\_index系统提前建立好的数据结构，**为什么要使用这两种数据结构**，是因为它能够快速的实现“用户网页检索”需求**（业务需求决定架构实现）**。

**提问：搜索的过程是什么样的？**

假设搜索词是“我爱”，用户会得到什么网页呢？

（1）分词，“我爱”会分词为{我，爱}，时间复杂度为O(1)

（2）每个分词后的item，从倒排索引查询包含这个item的网页list<url>，时间复杂度也是O(1)：

我 -> {url1, url2}

爱 -> {url1, url2}

（3）求list<url>的交集，就是符合所有查询词的结果网页，对于这个例子，{url1, url2}就是最终的查询结果

看似到这里就结束了，其实不然，分词和倒排查询时间复杂度都是O(1)，整个搜索的时间复杂度取决于“求list<url>的交集”，**问题转化为了求两个集合交集**。

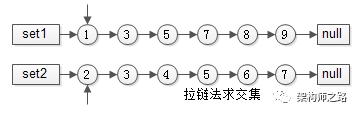
字符型的url不利于存储与计算，一般来说每个url会有一个数值型的url\_id来标识，后文为了方便描述，list<url>统一用list<url\_id>替代。

**list1和list2，求交集怎么求？**

**方案一**：for \* for，土办法，时间复杂度O(n\*n)

每个搜索词命中的网页是很多的，O(n\*n)的复杂度是明显不能接受的。倒排索引是在创建之初可以进行排序预处理，问题转化**成两个有序的list求交集**，就方便多了。

**方案二**：有序list求交集，拉链法



有序集合1{1,3,5,7,8,9}

有序集合2{2,3,4,5,6,7}

两个指针指向首元素，比较元素的大小：

（1）如果相同，放入结果集，随意移动一个指针

（2）否则，移动值较小的一个指针，直到队尾

这种方法的好处是：

（1）集合中的元素最多被比较一次，时间复杂度为O(n)

（2）多个有序集合可以同时进行，这适用于多个分词的item求url\_id交集

这个方法就像一条拉链的两边齿轮，一一比对就像拉链，故称为**拉链法**

**方案三**：分桶并行优化

数据量大时，**url\_id分桶水平切分+并行运算是一种常见的优化方法**，如果能将list1<url\_id>和list2<url\_id>分成若干个桶区间，每个区间利用多线程并行求交集，各个线程结果集的并集，作为最终的结果集，能够大大的减少执行时间。

举例：

有序集合1{1,3,5,7,8,9, 10,30,50,70,80,90}

有序集合2{2,3,4,5,6,7, 20,30,40,50,60,70}

求交集，先进行分桶拆分：

桶1的范围为[1, 9]

桶2的范围为[10, 100]

桶3的范围为[101, max\_int]

于是：

集合1就拆分成

集合a{1,3,5,7,8,9}

集合b{10,30,50,70,80,90}

集合c{}

集合2就拆分成

集合d{2,3,4,5,6,7}

集合e{20,30,40,50,60,70}

集合e{}

每个桶内的数据量大大降低了，并且每个桶内没有重复元素，可以利用多线程并行计算：

桶1内的集合a和集合d的交集是x{3,5,7}

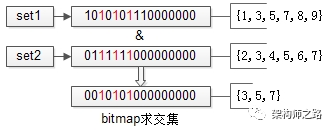
桶2内的集合b和集合e的交集是y{30, 50, 70}

桶3内的集合c和集合d的交集是z{}

最终，集合1和集合2的交集，是x与y与z的并集，即集合{3,5,7,30,50,70}

**方案四**：bitmap再次优化

数据进行了水平分桶拆分之后，**每个桶内的数据一定处于一个范围之内**，如果集合符合这个特点，**就可以使用bitmap来表示集合**：



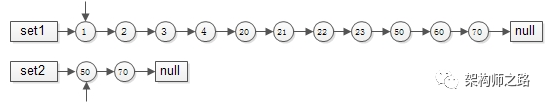
如上图，假设set1{1,3,5,7,8,9}和set2{2,3,4,5,6,7}的所有元素都在桶值[1, 16]的范围之内，可以用16个bit来描述这两个集合，原集合中的元素x，在这个16bitmap中的第x个bit为1，此时两个bitmap求交集，只需要将两个bitmap进行“与”操作，结果集bitmap的3，5，7位是1，表明原集合的交集为{3,5,7}

水平分桶，bitmap优化之后，能极大提高求交集的效率，但时间复杂度仍旧是O(n)

bitmap需要大量连续空间，占用内存较大

**方案五**：跳表skiplist

有序链表集合求交集，跳表是最常用的数据结构，它可以将有序集合求交集的复杂度由O(n)降至O(log(n))

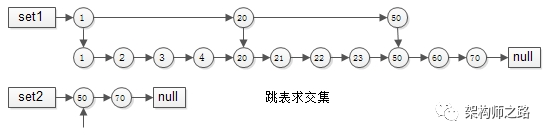


集合1{1,2,3,4,20,21,22,23,50,60,70}

集合2{50,70}

要求交集，如果用拉链法，会发现1,2,3,4,20,21,22,23都要被无效遍历一次，每个元素都要被比对，时间复杂度为O(n)，能不能每次比对“跳过一些元素”呢？

跳表就出现了：



集合1{1,2,3,4,20,21,22,23,50,60,70}建立跳表时，一级只有{1,20,50}三个元素，二级与普通链表相同

集合2{50,70}由于元素较少，只建立了一级普通链表

如此这般，在实施“拉链”求交集的过程中，set1的指针能够由1跳到20再跳到50，**中间能够跳过很多元素，无需进行一一比对**，跳表求交集的时间复杂度近似O(log(n))，这是搜索引擎中常见的算法。

**五、总结**

文字很多，有宏观，有细节，对于大部分不是专门研究搜索引擎的同学，记住以下几点即可：

（1）**全网搜索引擎系统**由spider， search&index， rank三个子系统构成

（2）**站内搜索引擎与全网搜索引擎的差异**在于，少了一个spider子系统

（3）spider和search&index系统是两个工程系统，rank系统的优化却需要长时间的调优和积累

（4）**正排索引（forward index）**是由网页url\_id快速找到分词后网页内容list<item>的过程

（5）**倒排索引（inverted index）**是由分词item快速寻找包含这个分词的网页list<url\_id>的过程

（6）用户**检索的过程**，是先分词，再找到每个item对应的list<url\_id>，最后进行集合求交集的过程

（7）**有序集合求交集的方法**有

         a）二重for循环法，时间复杂度O(n\*n)

         b）拉链法，时间复杂度O(n)

         c）水平分桶，多线程并行

         d）bitmap，大大提高运算并行度，时间复杂度O(n)

         e）跳表，时间复杂度为O(log(n))

**六、下章预告**

a）流量数据量由小到大，搜索方案与架构变迁-> 这个应该很有用，很多处于不同发展阶段的互联网公司都在做搜索系统，58同城经历过流量从0到10亿，数据量从0到100亿，搜索架构也不断演化着

b）数据量、并发量、策略扩展性及架构方案

c）实时搜索引擎核心技术 -> **站长发布1个新网页，Google如何做到15分钟后检索出来**

==【（上）完】==

相关文章：

[如何快速实现高并发短文检索](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959451&idx=1&sn=991d9c3737d7db50a8351d50cdf6419d&scene=21" \l "wechat_redirect" \t "_blank)

[100亿数据1万属性数据架构设计](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959855&idx=1&sn=f33abe8ec598c273f29cebb9365ece59&chksm=bd2d07f38a5a8ee58a944507a134e1da1efc3ac9c4d1c4cff261137cd986e51f5fe7cee9de15&scene=21" \l "wechat_redirect" \t "_blank)

## 就是这么迅猛的实现搜索需求

原创 2017-02-23 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959917&idx=1&sn=8faeae7419a756b0c355af2b30c255df&chksm=bd2d07b18a5a8ea75f16f7e98ea897c7e7f47a0441c64bdaef8445a2100e0bdd2a7de99786c0&scene=21" \l "#)

**一、缘起**

《[深入浅出搜索架构（上篇）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959895&idx=1&sn=de25ce2544c088ff9be0b93fd3ea4d15&chksm=bd2d078b8a5a8e9d5ae4339a683d3f980ff2994f3c10c4081c7bab7f0d77f37521de95e974bf&scene=21" \l "wechat_redirect" \t "_blank)》详细介绍了：

（1）全网搜索引擎架构与流程

（2）站内搜索引擎架构与流程

（3）搜索原理与核心数据结构

本文重点介绍：

（4）流量数据量由小到大，常见搜索方案与架构变迁

（5）数据量、并发量、扩展性方案

只要业务有检索需求，本文一定对你有帮助。

**二、检索需求的满足与架构演进**

任何互联网需求，或多或少有检索需求，还是以58同城的帖子业务场景为例，帖子的标题，帖子的内容有很强的用户检索需求，在业务、流量、并发量逐步递增的**各个阶段，应该如何实现检索需求呢**？

**原始阶段-LIKE**

数据在数据库中可能是这么存储的：

t\_tiezi(tid, title, content)

满足标题、内容的检索需求可以通过LIKE实现：

select tid from t\_tiezi where content like ‘%天通苑%’

能够快速满足业务需求，**存在的问题**也显而易见：

（1）效率低，每次需要全表扫描，计算量大，并发高时cpu容易100%

（2）不支持分词

**初级阶段-全文索引**

如何快速提高效率，支持分词，并对原有系统架构影响尽可能小呢，第一时间想到的是建立全文索引：

alter table t\_tiezi add fulltext(title,content)

使用match和against实现索引字段上的查询需求。

全文索引能够快速实现业务上分词的需求，并且快速提升性能（分词后倒排，至少不要全表扫描了），但也存在**一些问题**：

（1）只适用于MyISAM

（2）由于全文索引利用的是数据库特性，搜索需求和普通CURD需求耦合在数据库中：检索需求并发大时，可能影响CURD的请求；CURD并发大时，检索会非常的慢；

（3）数据量达到百万级别，性能还是会显著降低，查询返回时间很长，业务难以接受

（4）比较难水平扩展

**中级阶段-开源外置索引**

为了解决全文索的局限性，当数据量增加到大几百万，千万级别时，就要考虑外置索引了。外置索引的**核心思路**是：索引数据与原始数据分离，前者满足搜索需求，后者满足CURD需求，通过一定的机制（双写，通知，定期重建）来保证数据的一致性。

原始数据可以继续使用Mysql来存储，外置索引如何实施？Solr，Lucene，ES都是常见的开源方案。

**楼主强烈推荐ES**（ElasticSearch），原因是**Lucene虽好，但始终有一些不足**：

（1）Lucene只是一个库，潜台词是，需要自己做服务，自己实现高可用/可扩展/负载均衡等复杂特性

（2）Lucene只支持Java，如果要支持其他语言，还是得自己做服务

（3）Lucene不友好，这是很致命的，非常复杂，使用者往往需要深入了解搜索的知识来理解它的工作原理，为了屏蔽其复杂性，一个办法是自己做服务

…

…

为了改善Lucene的各项不足，解决方案都是“封装一个接口友好的服务，屏蔽底层复杂性”，于是有了ES：

（1）ES是一个以Lucene为内核来实现搜索功能，提供REStful接口的服务

（2）ES能够支持很大数据量的信息存储，支持很高并发的搜索请求

（3）ES支持集群，向使用者屏蔽高可用/可扩展/负载均衡等复杂特性

目前58到家使用ES作为核心，实现了自己的搜索服务平台，能够通过在平台上简单的配置，实现业务方的搜索需求。

搜索服务数据量最大的“接口耗时数据收集”需求，数据量大概在7亿左右；并发量最大的“经纬度，地理位置搜索”需求，线上平均并发量大概在600左右，压测数据并发量在6000左右。

**结论**：ES完全能满足10亿数据量，5k吞吐量的常见搜索业务需求，强烈推荐。

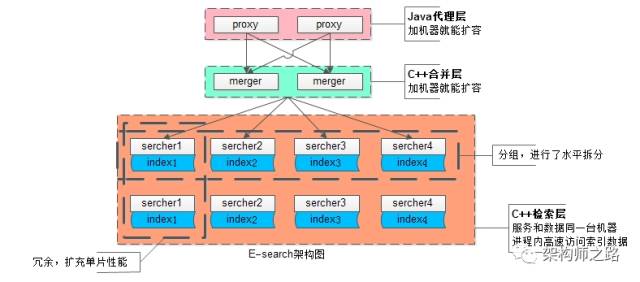
**高级阶段-自研搜索引擎**

当数据量进一步增加，达到**10亿、100亿数据量**；并发量也进一步增加，达到每秒**10万吞吐**；业务个性也逐步增加的时候，就需要自研搜索引擎了，定制化实现搜索内核了。

**三、数据量、并发量、扩展性方案**

到了定制化自研搜索引擎的阶段，超大数据量、超高并发量为设计重点，为了达到“无限容量、无限并发”的需求，架构设计需要重点考虑“扩展性”，力争做到：**增加机器就能扩容（数据量+并发量）**。

58同城的自研搜索引擎E-search初步架构图如下：



（1）上层**proxy（粉色）是接入集群**，为对外门户，接受搜索请求，其无状态性能够保证增加机器就能扩充proxy集群性能

（2）中层**merger（浅蓝色）是逻辑集群**，主要用于实现搜索合并，以及打分排序，业务相关的rank就在这一层实现，其无状态性也能够保证增加机器就能扩充merger集群性能

（3）底层**searcher（暗红色大框）是检索集群**，服务和索引数据部署在同一台机器上，服务启动时可以加载索引数据到内存，请求访问时从内存中load数据，访问速度很快

（3.1）为了满足数据容量的扩展性，索引数据进行了水平切分，增加切分份数，就能够无限扩展性能，如上图searcher分为了4组

（3.2）为了满足一份数据的性能扩展性，同一份数据进行了冗余，理论上做到增加机器就无限扩展性能，如上图每组searcher又冗余了2份

如此设计，真正做到做到**增加机器就能承载更多的数据量，响应更高的并发量**。

**三、总结**

为了满足搜索业务的需求，随着数据量和并发量的增长，搜索架构一般会经历这么几个阶段：

（1）原始阶段-**LIKE**

（2）初级阶段-**全文索引**

（3）中级阶段-**开源外置索引**

（4）高级阶段-**自研搜索引擎**

你的搜索架构到了哪一个阶段？数据量、并发量、好的经验欢迎分享？

欢迎留言，有问必答。

如果有收获，欢迎帮转。

**四、下章预告**

实时搜索引擎核心技术，站长发布1个新网页，Google如何做到15分钟后检索出来。

==【（中）完】==

相关文章：

[如何快速实现高并发短文检索](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959451&idx=1&sn=991d9c3737d7db50a8351d50cdf6419d&scene=21" \l "wechat_redirect" \t "_blank)

[深入浅出搜索引擎架构、方案、细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959895&idx=1&sn=de25ce2544c088ff9be0b93fd3ea4d15&chksm=bd2d078b8a5a8e9d5ae4339a683d3f980ff2994f3c10c4081c7bab7f0d77f37521de95e974bf&scene=21" \l "wechat_redirect" \t "_blank)

## 百度如何能实时检索到15分钟前新生成的网页？

原创 2017-03-01 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959949&idx=1&sn=83f78cf6293714bd1fd97a11ff7c2c35&chksm=bd2d07518a5a8e47e6fce9fc03cddec1d8a43f2b4ac67cfbbf73a55143593da8a132da7a0815&scene=21" \l "#)

**一、缘起**

《[深入浅出搜索架构（上篇）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959895&idx=1&sn=de25ce2544c088ff9be0b93fd3ea4d15&chksm=bd2d078b8a5a8e9d5ae4339a683d3f980ff2994f3c10c4081c7bab7f0d77f37521de95e974bf&scene=21" \l "wechat_redirect" \t "_blank)》详细介绍了前三章：

（1）全网搜索引擎架构与流程

（2）站内搜索引擎架构与流程

（3）搜索原理与核心数据结构

《[深入浅出搜索架构（中篇）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959917&idx=1&sn=8faeae7419a756b0c355af2b30c255df&chksm=bd2d07b18a5a8ea75f16f7e98ea897c7e7f47a0441c64bdaef8445a2100e0bdd2a7de99786c0&scene=21" \l "wechat_redirect" \t "_blank)》介绍了：

（4）流量数据量由小到大，常见搜索方案与架构变迁

（5）数据量、并发量、扩展性架构方案

本篇将讨论：

（6）百度为何能实时检索出15分钟之前新出的新闻？58同城为何能实时检索出1秒钟之前发布的帖子？**搜索引擎的实时性架构**，是本文将要讨论的问题。

**二、实时搜索引擎架构**

大数据量、高并发量情况下的搜索引擎为了保证实时性，架构设计上的两个要点：

（1）索引分级

（2）dump&merge

**索引分级**

《[深入浅出搜索架构（上篇）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959895&idx=1&sn=de25ce2544c088ff9be0b93fd3ea4d15&chksm=bd2d078b8a5a8e9d5ae4339a683d3f980ff2994f3c10c4081c7bab7f0d77f37521de95e974bf&scene=21" \l "wechat_redirect" \t "_blank)》介绍了搜索引擎的底层原理，在数据量非常大的情况下，为了保证倒排索引的高效检索效率，任何对数据的更新，并不会实时修改索引，一旦产生碎片，会大大降低检索效率。

既然索引数据不能实时修改，如何保证最新的网页能够被索引到呢？

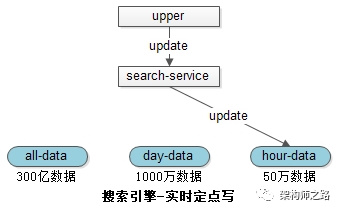
索引分为**全量库、日增量库、小时增量库**。

如下图所述：

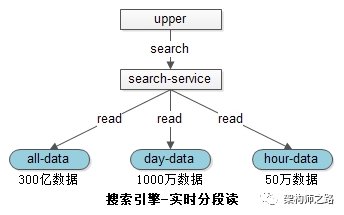
（1）300亿数据在全量索引库中

（2）1000万1天内修改过的数据在天库中

（3）50万1小时内修改过的数据在小时库中



当有**修改**请求发生时，只会操作最低级别的索引，例如小时库。



当有**查询**请求发生时，会同时查询各个级别的索引，将结果合并，得到最新的数据：

（1）全量库是紧密存储的索引，无碎片，速度快

（2）天库是紧密存储，速度快

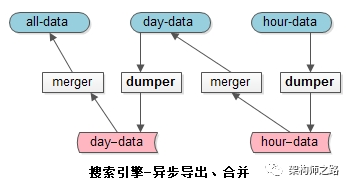
（3）小时库数据量小，速度也快

数据的写入和读取都是实时的，所以58同城能够检索到1秒钟之前发布的帖子，即使全量库有300亿的数据。

新的问题来了：小时库数据何时反映到天库中，天库中的数据何时反映到全量库中呢？

**dump&merge**

这是由两个异步的工具完成的：



**dumper**：将在线的数据导出

**merger**：将离线的数据合并到高一级别的索引中去

小时库，一小时一次，合并到天库中去；

天库，一天一次，合并到全量库中去；

这样就保证了小时库和天库的数据量都不会特别大；

如果数据量和并发量更大，还能增加星期库，月库来缓冲。

**三、总结**

超大数据量，超高并发量，实时搜索引擎的两个架构要点：

（1）索引分级

（2）dump&merge

如《[深入浅出搜索架构（上篇）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959895&idx=1&sn=de25ce2544c088ff9be0b93fd3ea4d15&chksm=bd2d078b8a5a8e9d5ae4339a683d3f980ff2994f3c10c4081c7bab7f0d77f37521de95e974bf&scene=21" \l "wechat_redirect" \t "_blank)》中所述，全网搜索引擎分为Spider, Search&Index, Rank三个部分。本文描述的是Search&Index如何实时修改和检索，Spider子系统如何能实时找到全网新生成的网页，又是另外一个问题，未来撰文讲述。

希望大家有收获，帮转哟。

==【完】==

相关文章：

[如何快速实现高并发短文检索](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959451&idx=1&sn=991d9c3737d7db50a8351d50cdf6419d&scene=21" \l "wechat_redirect" \t "_blank)

[深入浅出搜索引擎架构、方案、细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959895&idx=1&sn=de25ce2544c088ff9be0b93fd3ea4d15&chksm=bd2d078b8a5a8e9d5ae4339a683d3f980ff2994f3c10c4081c7bab7f0d77f37521de95e974bf&scene=21" \l "wechat_redirect" \t "_blank)

[就是这么迅猛的实现搜索需求](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959917&idx=1&sn=8faeae7419a756b0c355af2b30c255df&chksm=bd2d07b18a5a8ea75f16f7e98ea897c7e7f47a0441c64bdaef8445a2100e0bdd2a7de99786c0&scene=21" \l "wechat_redirect" \t "_blank)

## 到底什么时候该使用MQ？

原创 2017-04-05 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960012&idx=1&sn=c6af5c79ecead98daa4d742e5ad20ce5&chksm=bd2d07108a5a8e0624ae6ad95001c4efe09d7ba695f2ddb672064805d771f3f84bee8123b8a6&scene=21" \l "#)

**一、缘起**

一切脱离业务的架构设计与新技术引入都是耍流氓。

引入一个技术之前，首先应该解答的问题是，这个技术解决什么问题。

就像微服务分层架构之前，应该首先回答，为什么要引入微服务，微服务究竟解决什么问题（详见《[互联网架构为什么要做微服务？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959519&idx=1&sn=065074b135fc9cb243abe897261e1a72&scene=21" \l "wechat_redirect" \t "_blank)》）。

最近分享了几篇MQ相关的文章：

《[MQ如何实现延时消息](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959961&idx=1&sn=afec02c8dc6db9445ce40821b5336736&chksm=bd2d07458a5a8e5314560620c240b1c4cf3bbf801fc0ab524bd5e8aa8b8ef036cf755d7eb0f6&scene=21" \l "wechat_redirect" \t "_blank)》

《[MQ如何实现消息必达](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959966&idx=1&sn=068a2866dcc49335d613d75c4a5d1b17&chksm=bd2d07428a5a8e54162ad8ea8e1e9302dfaeb664cecc453bd16a5f299820755bd2e1e0e17b60&scene=21" \l "wechat_redirect" \t "_blank)》

《[MQ如何实现幂等性](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960002&idx=1&sn=c0775231bccf002c3178eabe43f1cdcb&chksm=bd2d071e8a5a8e08c3a5287247ea41dee6b2621e6ffafbf909ec1e8a866b7c816eeeea227246&scene=21" \l "wechat_redirect" \t "_blank)》

不少网友询问，究竟什么时候使用MQ，MQ究竟适合什么场景，故有了此文。

**二、MQ是干嘛的**

消息总线（Message Queue），后文称MQ，是一种跨进程的通信机制，用于上下游传递消息。



在互联网架构中，MQ是一种非常常见的上下游“逻辑解耦+物理解耦”的消息通信服务。

使用了MQ之后，消息发送上游只需要依赖MQ，逻辑上和物理上都不用依赖其他服务。

**三、什么时候不使用消息总线**



既然MQ是互联网分层架构中的解耦利器，那所有通讯都使用MQ岂不是很好？**这是一个严重的误区**，调用与被调用的关系，是无法被MQ取代的。

MQ的**不足**是：

1）系统更复杂，多了一个MQ组件

2）消息传递路径更长，延时会增加

3）消息可靠性和重复性互为矛盾，消息不丢不重难以同时保证

4）上游无法知道下游的执行结果，这一点是很致命的

举个**栗子**：用户登录场景，登录页面调用passport服务，passport服务的执行结果直接影响登录结果，此处的“登录页面”与“passport服务”就必须使用调用关系，而不能使用MQ通信。

无论如何，记住这个**结论**：**调用方实时依赖执行结果的业务场景，请使用调用，而不是MQ**。

**四、什么时候使用MQ**

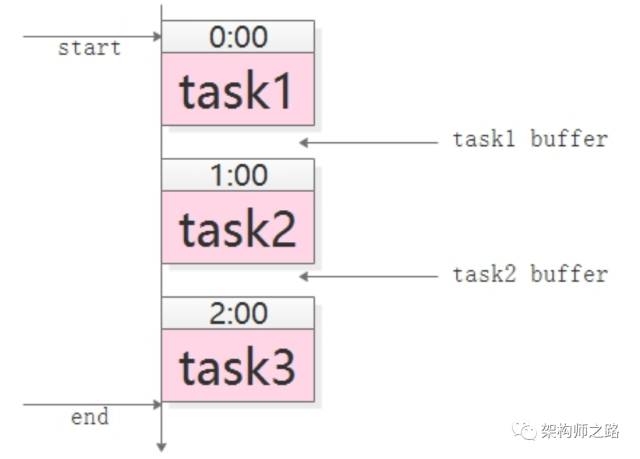
**【典型场景一：数据驱动的任务依赖】**

 什么是任务依赖，举个**栗子**，互联网公司经常在凌晨进行一些数据统计任务，这些任务之间有一定的依赖关系，比如：

1）task3需要使用task2的输出作为输入

2）task2需要使用task1的输出作为输入

这样的话，tast1, task2, task3之间就有任务依赖关系，必须task1先执行，再task2执行，载task3执行。



对于这类需求，**常见的实现方式**是，使用cron人工排执行时间表：

1）task1，0:00执行，经验执行时间为50分钟

2）task2，1:00执行（为task1预留10分钟buffer），经验执行时间也是50分钟

3）task3，2:00执行（为task2预留10分钟buffer）

这种方法的**坏处**是：

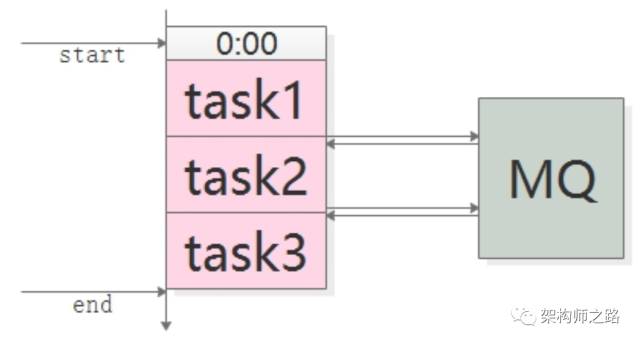
1）如果有一个任务执行时间超过了预留buffer的时间，将会得到错误的结果，因为后置任务不清楚前置任务是否执行成功，此时要手动重跑任务，还有可能要调整排班表

2）总任务的执行时间很长，总是要预留很多buffer，如果前置任务提前完成，后置任务不会提前开始

3）如果一个任务被多个任务依赖，这个任务将会称为关键路径，排班表很难体现依赖关系，容易出错

4）如果有一个任务的执行时间要调整，将会有多个任务的执行时间要调整

无论如何，采用“cron排班表”的方法，各任务耦合，**谁用过谁痛谁知道**（采用此法的请评论留言）



优化方案是，采用MQ解耦：

1）task1准时开始，结束后发一个“task1 done”的消息

2）task2订阅“task1 done”的消息，收到消息后第一时间启动执行，结束后发一个“task2 done”的消息

3）task3同理

采用MQ的**优点**是：

1）不需要预留buffer，上游任务执行完，下游任务总会在第一时间被执行

2）依赖多个任务，被多个任务依赖都很好处理，只需要订阅相关消息即可

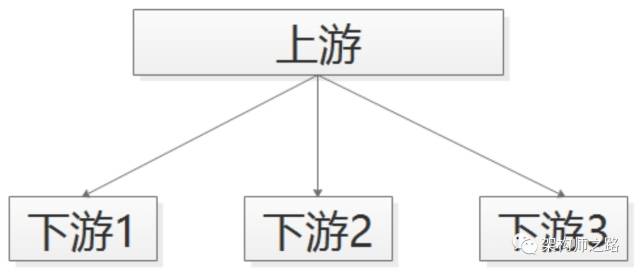
3）有任务执行时间变化，下游任务都不需要调整执行时间

需要**特别说明**的是，MQ只用来传递上游任务执行完成的消息，并不用于传递真正的输入输出数据。

**【典型场景二：上游不关心执行结果】**

上游需要关注执行结果时要用“调用”，上游不关注执行结果时，就可以使用MQ了。

举个**栗子**，58同城的很多下游需要关注“用户发布帖子”这个事件，比如招聘用户发布帖子后，招聘业务要奖励58豆，房产用户发布帖子后，房产业务要送2个置顶，二手用户发布帖子后，二手业务要修改用户统计数据。



对于这类需求，**常见的实现方式**是，使用调用关系：

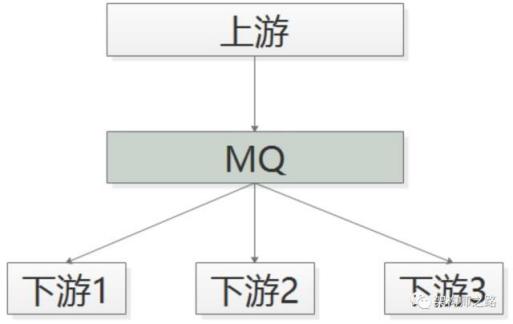
帖子发布服务执行完成之后，调用下游招聘业务、房产业务、二手业务，来完成消息的通知，但事实上，这个通知是否正常正确的执行，帖子发布服务根本不关注。

这种方法的**坏处**是：

1）帖子发布流程的执行时间增加了

2）下游服务当机，可能导致帖子发布服务受影响，上下游逻辑+物理依赖严重

3）每当增加一个需要知道“帖子发布成功”信息的下游，修改代码的是帖子发布服务，这一点是最恶心的，属于架构设计中典型的依赖倒转，**谁用过谁痛谁知道**（采用此法的请评论留言）



优化方案是，采用MQ解耦：

1）帖子发布成功后，向MQ发一个消息

2）哪个下游关注“帖子发布成功”的消息，主动去MQ订阅

采用MQ的**优点**是：

1）上游执行时间短

2）上下游逻辑+物理解耦，除了与MQ有物理连接，模块之间都不相互依赖

3）新增一个下游消息关注方，上游不需要修改任何代码

**典型场景三：上游关注执行结果，但执行时间很长**

 有时候上游需要关注执行结果，但执行结果时间很长（典型的是调用离线处理，或者跨公网调用），也经常使用**回调网关+MQ**来解耦。

举个**栗子**，微信支付，跨公网调用微信的接口，执行时间会比较长，但调用方又非常关注执行结果，此时一般怎么玩呢？



一般采用“回调网关+MQ”方案来解耦：

1）调用方直接跨公网调用微信接口

2）微信返回调用成功，此时并不代表返回成功

3）微信执行完成后，回调统一网关

4）网关将返回结果通知MQ

5）请求方收到结果通知

这里**需要注意**的是，不应该由回调网关来调用上游来通知结果，如果是这样的话，每次新增调用方，回调网关都需要修改代码，仍然会反向依赖，使用**回调网关+MQ**的方案，新增任何对微信支付的调用，都不需要修改代码啦。

**五、总结**

**MQ是一个互联网架构中常见的解耦利器。**

**什么时候不使用MQ？**

上游实时关注执行结果

**什么时候使用MQ？**

1）数据驱动的任务依赖

2）上游不关心多下游执行结果

3）异步返回执行时间长

==【完】==

相关阅读：

《[MQ如何实现延时消息](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959961&idx=1&sn=afec02c8dc6db9445ce40821b5336736&chksm=bd2d07458a5a8e5314560620c240b1c4cf3bbf801fc0ab524bd5e8aa8b8ef036cf755d7eb0f6&scene=21" \l "wechat_redirect" \t "_blank)》

《[MQ如何实现消息必达](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959966&idx=1&sn=068a2866dcc49335d613d75c4a5d1b17&chksm=bd2d07428a5a8e54162ad8ea8e1e9302dfaeb664cecc453bd16a5f299820755bd2e1e0e17b60&scene=21" \l "wechat_redirect" \t "_blank)》

《[MQ如何实现幂等性](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960002&idx=1&sn=c0775231bccf002c3178eabe43f1cdcb&chksm=bd2d071e8a5a8e08c3a5287247ea41dee6b2621e6ffafbf909ec1e8a866b7c816eeeea227246&scene=21" \l "wechat_redirect" \t "_blank)》

栗子不少，有收获确实可以帮转一手。

## 消息总线真的能保证幂等？

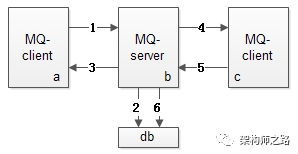
原创 2017-03-31 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960002&idx=1&sn=c0775231bccf002c3178eabe43f1cdcb&chksm=bd2d071e8a5a8e08c3a5287247ea41dee6b2621e6ffafbf909ec1e8a866b7c816eeeea227246&scene=21" \l "#)

**一、缘起**

如《[消息总线消息必达](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959966&idx=1&sn=068a2866dcc49335d613d75c4a5d1b17&chksm=bd2d07428a5a8e54162ad8ea8e1e9302dfaeb664cecc453bd16a5f299820755bd2e1e0e17b60&scene=21" \l "wechat_redirect" \t "_blank)》所述，MQ消息必达，架构上有两个核心设计点：

（1）消息落地

（2）消息超时、重传、确认



再次回顾消息总线核心架构，它由**发送端、服务端、固化存储、接收端**四大部分组成。

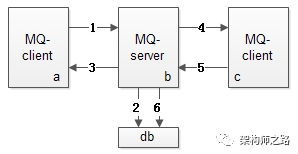
为保证消息的可达性，超时、重传、确认机制可能导致消息总线、或者业务方**收到重复的消息**，从而对业务产生影响。

举个栗子：

购买会员卡，上游支付系统负责给用户扣款，下游系统负责给用户发卡，通过MQ异步通知。不管是上半场的ACK丢失，导致MQ收到重复的消息，还是下半场ACK丢失，导致购卡系统收到重复的购卡通知，都可能出现，上游扣了一次钱，下游发了多张卡。

**消息总线的幂等性设计**至关重要，是本文将要讨论的重点。

**二、上半场的幂等性设计**



MQ消息发送上半场，即上图中的1-3

1，发送端MQ-client将消息发给服务端MQ-server

2，服务端MQ-server将消息落地

3，服务端MQ-server回ACK给发送端MQ-client

如果3丢失，发送端MQ-client超时后会重发消息，可能导致服务端MQ-server收到重复消息。

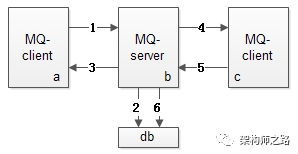
此时重发是MQ-client发起的，消息的处理是MQ-server，为了避免步骤2落地重复的消息，对每条消息，**MQ系统内部必须生成一个inner-msg-id**，作为去重和幂等的依据，这个内部消息ID的特性是：

（1）全局唯一

（2）MQ生成，具备业务无关性，对消息发送方和消息接收方屏蔽

**有了这个inner-msg-id，就能保证上半场重发，也只有1条消息落到MQ-server的DB中，实现上半场幂等。**

**三、下半场的幂等性设计**



MQ消息发送下半场，即上图中的4-6

4，服务端MQ-server将消息发给接收端MQ-client

5，接收端MQ-client回ACK给服务端

6，服务端MQ-server将落地消息删除

需要强调的是，接收端MQ-client回ACK给服务端MQ-server，是消息消费业务方的主动调用行为，不能由MQ-client自动发起，因为MQ系统不知道消费方什么时候真正消费成功。

如果5丢失，服务端MQ-server超时后会重发消息，可能导致MQ-client收到重复的消息。

此时重发是MQ-server发起的，消息的处理是消息消费业务方，消息重发势必导致业务方重复消费（上例中的一次付款，重复发卡），为了保证业务幂等性**，业务消息体中，必须有一个biz-id**，作为去重和幂等的依据，这个业务ID的特性是：

（1）对于同一个业务场景，全局唯一

（2）由业务消息发送方生成，业务相关，对MQ透明

（3）由业务消息消费方负责判重，以保证幂等

最常见的业务ID有：支付ID，订单ID，帖子ID等。

具体到支付购卡场景，发送方必须将支付ID放到消息体中，消费方必须对同一个支付ID进行判重，保证购卡的幂等。

**有了这个业务ID，才能够保证下半场消息消费业务方即使收到重复消息，也只有1条消息被消费，保证了幂等。**

**三、总结**

MQ为了保证消息必达，消息上下半场均可能发送重复消息，如何保证消息的幂等性呢？

**上半场**

MQ-client生成inner-msg-id，保证上半场幂等。

这个ID全局唯一，业务无关，由MQ保证。

**下半场**

业务**发送方**带入biz-id，业务**接收方去重**保证幂等。

这个ID对单业务唯一，业务相关，对MQ透明。

结论：幂等性，不仅对MQ有要求，对业务上下游也有要求。

==【完】==

相关阅读：

[“延迟消息”如何实现](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959961&idx=1&sn=afec02c8dc6db9445ce40821b5336736&chksm=bd2d07458a5a8e5314560620c240b1c4cf3bbf801fc0ab524bd5e8aa8b8ef036cf755d7eb0f6&scene=21" \l "wechat_redirect" \t "_blank)

[消息总线能否实现消息必达](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959966&idx=1&sn=068a2866dcc49335d613d75c4a5d1b17&chksm=bd2d07428a5a8e54162ad8ea8e1e9302dfaeb664cecc453bd16a5f299820755bd2e1e0e17b60&scene=21" \l "wechat_redirect" \t "_blank)

## 58到家MQ如何快速实现流量削峰填谷

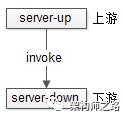
原创 2017-04-12 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960021&idx=1&sn=4bbe275c249a70ab20e36959fc01d4e0&chksm=bd2d07098a5a8e1fd9b505778b551002ab59c35953fa3deaaddc79e3f21bcea5ff48076b4a89&scene=21" \l "#)

**问：为什么会有本文？**

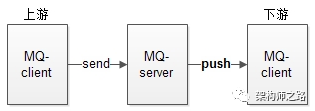
答：上一篇文章《[到底什么时候该使用MQ？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960012&idx=1&sn=c6af5c79ecead98daa4d742e5ad20ce5&chksm=bd2d07108a5a8e0624ae6ad95001c4efe09d7ba695f2ddb672064805d771f3f84bee8123b8a6&scene=21" \l "wechat_redirect" \t "_blank)》引起了广泛的讨论，有朋友回复说，MQ的还有一个**典型应用场景**是缓冲流量，削峰填谷，本文将简单介绍下，MQ要实现什么细节，才能缓冲流量，削峰填谷。

**问：站点与服务，服务与服务上下游之间，一般如何通讯？**

答：有两种常见的方式



一种是“直接调用”，通过RPC框架，上游直接调用下游。



在某些业务场景之下（具体哪些业务场景，见《[到底什么时候该使用MQ？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960012&idx=1&sn=c6af5c79ecead98daa4d742e5ad20ce5&chksm=bd2d07108a5a8e0624ae6ad95001c4efe09d7ba695f2ddb672064805d771f3f84bee8123b8a6&scene=21" \l "wechat_redirect" \t "_blank)》），可以采用“MQ推送”，上游将消息发给MQ，MQ将消息推送给下游。

**问：为什么会有流量冲击？**

答：不管采用“直接调用”还是“MQ推送”，都有一个**缺点**，下游消息接收方无法控制到达自己的流量，如果调用方不限速，很有可能把下游压垮。

举个**栗子**，秒杀业务：

**上游**发起下单操作

**下游**完成秒杀业务逻辑（库存检查，库存冻结，余额检查，余额冻结，订单生成，余额扣减，库存扣减，生成流水，余额解冻，库存解冻）

上游下单业务简单，每秒发起了10000个请求，下游秒杀业务复杂，每秒只能处理2000个请求，很有可能上游不限速的下单，导致下游系统被压垮，引发雪崩。

为了避免雪崩，**常见的优化方案**有两种：

1）业务上游队列缓冲，限速发送

2）业务下游队列缓冲，限速执行

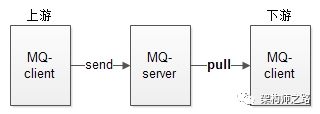
不管哪种方案，都会引入业务的复杂性，有“缓冲流量”需求的系统都需要加入类似的机制（具体怎么保证消息可达，见《[消息总线能否实现消息必达？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959966&idx=1&sn=068a2866dcc49335d613d75c4a5d1b17&chksm=bd2d07428a5a8e54162ad8ea8e1e9302dfaeb664cecc453bd16a5f299820755bd2e1e0e17b60&scene=21" \l "wechat_redirect" \t "_blank)》），正所谓“通用痛点统一解决”，需要一个通用的机制解决这个问题。

**问：如何缓冲流量？**

答：明明中间有了MQ，并且MQ有消息落地的机制，为何不能利用MQ来做缓冲呢？显然是可以的。

**问：MQ怎么改能缓冲流量？**

答：由MQ-server推模式，升级为MQ-client拉模式。



MQ-client根据自己的处理能力，每隔一定时间，或者每次拉取若干条消息，实施**流控**，达到保护自身的效果。并且这是MQ提供的通用功能，无需上下游修改代码。

**问：如果上游发送流量过大，MQ提供拉模式确实可以起到下游自我保护的作用，会不会导致消息在MQ中堆积？**

答：下游MQ-client拉取消息，消息接收方能够批量获取消息，需要下游消息接收方进行优化，方能够提升整体吞吐量，例如：批量写。

**结论**

1）MQ-client提供拉模式，定时或者批量拉取，可以起到削平流量，下游自我保护的作用（MQ需要做的）

2）要想提升整体吞吐量，需要下游优化，例如批量处理等方式（消息接收方需要做的）

58到家架构优化具备**整体性**，需要通用服务和业务方一起优化升级。

==【完】==

相关阅读：

《[到底什么时候使用MQ](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960012&idx=1&sn=c6af5c79ecead98daa4d742e5ad20ce5&chksm=bd2d07108a5a8e0624ae6ad95001c4efe09d7ba695f2ddb672064805d771f3f84bee8123b8a6&scene=21" \l "wechat_redirect" \t "_blank)》

《[MQ如何实现延时消息](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959961&idx=1&sn=afec02c8dc6db9445ce40821b5336736&chksm=bd2d07458a5a8e5314560620c240b1c4cf3bbf801fc0ab524bd5e8aa8b8ef036cf755d7eb0f6&scene=21" \l "wechat_redirect" \t "_blank)》

《[MQ如何实现消息必达](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959966&idx=1&sn=068a2866dcc49335d613d75c4a5d1b17&chksm=bd2d07428a5a8e54162ad8ea8e1e9302dfaeb664cecc453bd16a5f299820755bd2e1e0e17b60&scene=21" \l "wechat_redirect" \t "_blank)》

《[MQ如何实现幂等性](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960002&idx=1&sn=c0775231bccf002c3178eabe43f1cdcb&chksm=bd2d071e8a5a8e08c3a5287247ea41dee6b2621e6ffafbf909ec1e8a866b7c816eeeea227246&scene=21" \l "wechat_redirect" \t "_blank)》

## 10w定时任务，如何高效触发超时

原创 2017-03-09 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959957&idx=1&sn=a82bb7e8203b20b2a0cb5fc95b7936a5&chksm=bd2d07498a5a8e5f9f8e7b5aeaa5bd8585a0ee4bf470956e7fd0a2b36d132eb46553265f4eaf&scene=21" \l "#)

**一、缘起**

很多时候，业务有定时任务或者定时超时的需求，当任务量很大时，可能需要维护大量的timer，或者进行低效的扫描。

**例如**：58到家APP实时消息通道系统，对每个用户会维护一个APP到服务器的TCP连接，用来实时收发消息，对这个TCP连接，有这样一个需求：“如果连续30s没有请求包（例如登录，消息，keepalive包），服务端就要将这个用户的状态置为离线”。

其中，单机TCP同时在线量约在10w级别，keepalive请求包大概30s一次，吞吐量约在3000qps。

**一般来说怎么实现这类需求呢？**

**“轮询扫描法”**

1）用一个Map<uid, last\_packet\_time>来记录每一个uid最近一次请求时间last\_packet\_time

2）当某个用户uid有请求包来到，实时更新这个Map

3）启动一个timer，当Map中不为空时，轮询扫描这个Map，看每个uid的last\_packet\_time是否超过30s，如果超过则进行超时处理

**“多timer触发法”**

1）用一个Map<uid, last\_packet\_time>来记录每一个uid最近一次请求时间last\_packet\_time

2）当某个用户uid有请求包来到，实时更新这个Map，并同时对这个uid请求包启动一个timer，30s之后触发

3）每个uid请求包对应的timer触发后，看Map中，查看这个uid的last\_packet\_time是否超过30s，如果超过则进行超时处理

**方案一**：只启动一个timer，但需要轮询，效率较低

**方案二**：不需要轮询，但每个请求包要启动一个timer，比较耗资源

特别在同时在线量很大时，很容易CPU100%，**如何高效维护和触发大量的定时/超时任务，是本文要讨论的问题**。

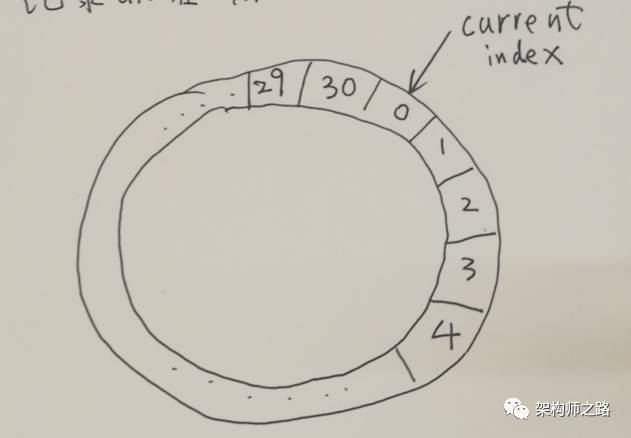
**二、环形队列法**

废话不多说，三个重要的数据结构：

1）30s超时，就创建一个index从0到30的**环形队列**（本质是个数组）

2）环上每一个slot是一个**Set<uid>，任务集合**

3）同时还有一个**Map<uid, index>**，记录uid落在环上的哪个slot里



**同时**：

1）启动一个timer，每隔1s，在上述环形队列中移动一格，0->1->2->3…->29->30->0…

2）有一个Current Index指针来标识刚检测过的slot

**当有某用户uid有请求包到达时**：

1）从Map结构中，查找出这个uid存储在哪一个slot里

2）从这个slot的Set结构中，删除这个uid

3）将uid重新加入到新的slot中，具体是哪一个slot呢 => Current Index指针所指向的**上一个**slot，因为这个slot，会被timer在30s之后扫描到

（4）更新Map，这个uid对应slot的index值

**哪些元素会被超时掉呢？**

Current Index每秒种移动一个slot，这个slot对应的Set<uid>中所有uid都应该被集体超时！如果最近30s有请求包来到，一定被放到Current Index的前一个slot了，Current Index所在的slot对应Set中所有元素，都是最近30s没有请求包来到的。

所以，当没有超时时，Current Index扫到的每一个slot的Set中应该都没有元素。

**优势**：

（1）只需要1个timer

（2）timer每1s只需要一次触发，消耗CPU很低

（3）批量超时，Current Index扫到的slot，Set中所有元素都应该被超时掉

**三、总结**

这个环形队列法是一个通用的方法，Set和Map中可以是任何task，本文的uid是一个最简单的举例。

**HashedWheelTimer**也是类似的原理，有兴趣的同学可以百度一下这个数据结构，Netty中的一个工具类，希望大家有收获，帮忙转发一下哈。

==【完】==

相关推荐：

[究竟啥才是互联网架构“高可用”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959728&idx=1&sn=933227840ec8cdc35d3a33ae3fe97ec5&chksm=bd2d046c8a5a8d7a13551124af36bedf68f7a6e31f6f32828678d2adb108b86b7e08c678f22f&scene=21" \l "wechat_redirect" \t "_blank)

[100亿数据1万属性数据架构设计](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959855&idx=1&sn=f33abe8ec598c273f29cebb9365ece59&chksm=bd2d07f38a5a8ee58a944507a134e1da1efc3ac9c4d1c4cff261137cd986e51f5fe7cee9de15&scene=21" \l "wechat_redirect" \t "_blank)

[数据库秒级平滑扩容架构方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959883&idx=1&sn=e7df8510c7096a5b069e0f12eaaca010&chksm=bd2d07978a5a8e815c2ae41b16b6b4c579923502fb919008a22bb108a1e920109f25387f8903&scene=21" \l "wechat_redirect" \t "_blank)

[58到家数据库30条军规](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959906&idx=1&sn=2cbdc66cfb5b53cf4327a1e0d18d9b4a&chksm=bd2d07be8a5a8ea86dc3c04eced3f411ee5ec207f73d317245e1fefea1628feb037ad71531bc&scene=21" \l "wechat_redirect" \t "_blank)

[业界难题-“跨库分页”的四种方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959942&idx=1&sn=e9d3fe111b8a1d44335f798bbb6b9eea&chksm=bd2d075a8a5a8e4cad985b847778aa83056e22931767bb835132c04571b66d5434020fd4147f&scene=21" \l "wechat_redirect" \t "_blank)

## 1分钟实现“延迟消息”功能

原创 2017-03-15 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959961&idx=1&sn=afec02c8dc6db9445ce40821b5336736&chksm=bd2d07458a5a8e5314560620c240b1c4cf3bbf801fc0ab524bd5e8aa8b8ef036cf755d7eb0f6&scene=21" \l "#)

**一、缘起**

很多时候，业务有“在一段时间之后，完成一个工作任务”的需求。

例如：滴滴打车订单完成后，如果用户一直不评价，48小时后会将自动评价为5星。

一般来说怎么实现这类“48小时后自动评价为5星”需求呢？

**常见方案**：启动一个cron定时任务，每小时跑一次，将完成时间超过48小时的订单取出，置为5星，并把评价状态置为已评价。

假设订单表的结构为：t\_order(oid, finish\_time, stars, status, …)，更具体的，定时任务每隔一个小时会这么做一次：

select oid from t\_order where finish\_time > 48hours and status=0;

update t\_order set stars=5 and status=1 where oid in[…];

如果数据量很大，需要分页查询，分页update，这将会是一个for循环。

方案的不足：

（1）**轮询效率比较低**

（2）每次扫库，已经被执行过记录，仍然会被扫描（只是不会出现在结果集中），有**重复计算**的嫌疑

（3）**时效性不够好**，如果每小时轮询一次，最差的情况下，时间误差会达到1小时

（4）如果通过增加cron轮询频率来减少（3）中的时间误差，（1）中轮询低效和（2）中重复计算的问题会进一步凸显

如何利用“延时消息”，对于每个任务只触发一次，保证效率的同时保证实时性，是今天要讨论的问题。

**二、高效延时消息设计与实现**

高效延时消息，包含两个重要的数据结构：

（1）环形队列，例如可以创建一个包含3600个slot的**环形队列**（本质是个数组）

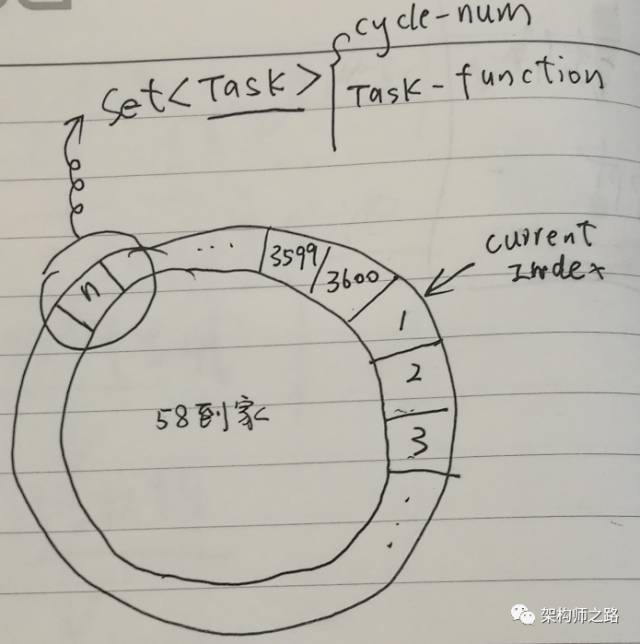
（2）任务集合，环上每一个slot是一个**Set<Task>**

同时，启动一个timer，这个timer每隔1s，在上述环形队列中移动一格，有一个Current Index指针来标识正在检测的slot。

Task结构中有两个很重要的属性：

（1）Cycle-Num：当Current Index第几圈扫描到这个Slot时，执行任务

（2）Task-Function：需要执行的任务指针



假设当前Current Index指向第一格，当有延时消息到达之后，例如希望3610秒之后，触发一个延时消息任务，只需：

（1）**计算这个Task应该放在哪一个slot**，现在指向1，3610秒之后，应该是第11格，所以这个Task应该放在第11个slot的Set<Task>中

（2）**计算这个Task的Cycle-Num**，由于环形队列是3600格（每秒移动一格，正好1小时），这个任务是3610秒后执行，所以应该绕3610/3600=1圈之后再执行，于是Cycle-Num=1

Current Index不停的移动，每秒移动到一个新slot，这个slot中对应的Set<Task>，每个Task看Cycle-Num是不是0：

（1）如果不是0，说明还需要多移动几圈，将Cycle-Num减1

（2）如果是0，说明马上要执行这个Task了，取出Task-Funciton执行（可以用单独的线程来执行Task），并把这个Task从Set<Task>中删除

使用了“延时消息”方案之后，“订单48小时后关闭评价”的需求，只需将在订单关闭时，触发一个48小时之后的延时消息即可：

（1）无需再轮询全部订单，效率高

（2）一个订单，任务只执行一次

（3）时效性好，精确到秒（控制timer移动频率可以控制精度）

**三、总结**

环形队列是一个实现“延时消息”的好方法，开源的MQ好像都不支持延迟消息，不妨自己实现一个简易的“延时消息队列”，能解决很多业务问题，并减少很多低效扫库的cron任务。

另外，关于MQ的可达性、幂等性未来撰文另述。

如果对文章和配图满意的话，帮忙转发一下哈。

==【完】==

相关阅读：

[10w定时任务，如何高效触发超时](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959957&idx=1&sn=a82bb7e8203b20b2a0cb5fc95b7936a5&chksm=bd2d07498a5a8e5f9f8e7b5aeaa5bd8585a0ee4bf470956e7fd0a2b36d132eb46553265f4eaf&scene=21" \l "wechat_redirect" \t "_blank)

## 消息总线能否实现消息必达？

原创 2017-03-18 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959966&idx=1&sn=068a2866dcc49335d613d75c4a5d1b17&chksm=bd2d07428a5a8e54162ad8ea8e1e9302dfaeb664cecc453bd16a5f299820755bd2e1e0e17b60&scene=21" \l "#)

**一、缘起**

上周讨论了两期环形队列的业务应用：

《[高效定时任务的触发](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959957&idx=1&sn=a82bb7e8203b20b2a0cb5fc95b7936a5&chksm=bd2d07498a5a8e5f9f8e7b5aeaa5bd8585a0ee4bf470956e7fd0a2b36d132eb46553265f4eaf&scene=21" \l "wechat_redirect" \t "_blank)》

《[延迟消息的快速实现](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959961&idx=1&sn=afec02c8dc6db9445ce40821b5336736&chksm=bd2d07458a5a8e5314560620c240b1c4cf3bbf801fc0ab524bd5e8aa8b8ef036cf755d7eb0f6&scene=21" \l "wechat_redirect" \t "_blank)》

两期的均有大量读者提问：

任务、延迟消息都放在内存里，万一重启了怎么办？

能否保证消息必达？

今天就简单聊聊消息队列（MsgQueue）的消息必达性架构与流程。

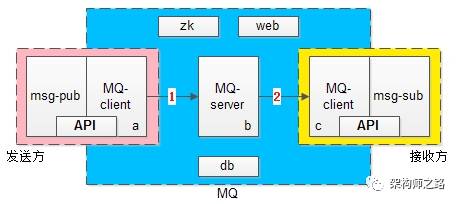
**二、架构方向**

MQ要想尽量消息必达，架构上有两个核心设计点：

（1）**消息落地**

（2）**消息超时、重传、确认**

**三、MQ核心架构**



上图是一个MQ的核心架构图，基本可以分为三大块：

（1）**发送方** -> 左侧粉色部分

（2）**MQ核心集群** -> 中间蓝色部分

（3）**接收方** -> 右侧黄色部分

**粉色发送方**又由两部分构成：**业务调用方**与**MQ-client-sender**

其中后者向前者提供了两个核心API：

SendMsg(bytes[] msg)

SendCallback()

**蓝色MQ核心集群**又分为四个部分：**MQ-server，zk，db，管理后台web**

**黄色接收方**也由两部分构成：**业务接收方**与**MQ-client-receiver**

其中后者向前者提供了两个核心API：

RecvCallback(bytes[] msg)

SendAck()

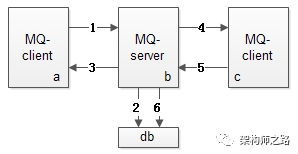
MQ是一个系统间解耦的利器，它能够很好的解除发布订阅者之间的耦合，它将上下游的消息投递解耦成两个部分，如上述架构图中的1箭头和2箭头：

（1）发送方将消息投递给MQ，**上半场**

（2）MQ将消息投递给接收方，**下半场**

**四、MQ消息可靠投递核心流程**

MQ既然将消息投递拆成了上下半场，为了保证消息的可靠投递，上下半场都必须尽量保证消息必达。



MQ消息投递**上半场**，MQ-client-sender到MQ-server流程见上图1-3：

（1）MQ-client将消息发送给MQ-server（此时业务方调用的是API：SendMsg）

（2）MQ-server将消息落地，落地后即为发送成功

（3）MQ-server将应答发送给MQ-client（此时回调业务方是API：SendCallback）

MQ消息投递**下半场**，MQ-server到MQ-client-receiver流程见上图4-6：

（1）MQ-server将消息发送给MQ-client（此时回调业务方是API：RecvCallback）

（2）MQ-client回复应答给MQ-server（此时业务方主动调用API：SendAck）

（3）MQ-server收到ack，将之前已经落地的消息删除，完成消息的可靠投递

**如果消息丢了怎么办？**

MQ消息投递的上下半场，都可以出现消息丢失，为了降低消息丢失的概率，MQ需要进行超时和重传。

**上半场的超时与重传**

MQ上半场的1或者2或者3如果丢失或者超时，MQ-client-sender内的timer会重发消息，直到期望收到3，如果重传N次后还未收到，则SendCallback回调发送失败，需要注意的是，这个过程中MQ-server可能会收到同一条消息的多次重发。

**下半场的超时与重传**

MQ下半场的4或者5或者6如果丢失或者超时，MQ-server内的timer会重发消息，直到收到5并且成功执行6，这个过程可能会重发很多次消息，一般采用指数退避的策略，先隔x秒重发，2x秒重发，4x秒重发，以此类推，需要注意的是，这个过程中MQ-client-receiver也可能会收到同一条消息的多次重发。

MQ-client与MQ-server如何进行消息去重，如何进行架构幂等性设计，下一次撰文另述，此处暂且认为为了保证消息必达，可能收到重复的消息。

**五、总结**

消息总线是系统之间的解耦利器，但切勿滥用，未来也会撰文细究MQ的使用场景，消息总线为了尽量保证消息必达，架构设计方向为：

（1）**消息收到先落地**

（2）**消息超时、重传、确认保证消息必达**

有问题随时沟通交流，后续讲消息去重、幂等性设计、何时该使用MQ。

==【完】==

相关阅读：

[架构师之路16年精选66篇](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959886&idx=1&sn=03e45a5014053607eff5e55ed2c660d7&chksm=bd2d07928a5a8e8454d395e176fa9d346682abfe9dfbf3244f1dead83ee4508aa25121f9b811&scene=21" \l "wechat_redirect" \t "_blank)

## 如何精确理解leader布置的任务

原创 2017-04-17 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960035&idx=1&sn=e0a14e8539b001e4d6d149164987235b&chksm=bd2d073f8a5a8e290202fcdfac077744eab84e6549e0608698ae5e1d3ed221ee97ca21f42e6f&scene=21" \l "#)

**【缘起】**

和一个同学交代了一个很重要的事情，结果执行的结果并不是自己想要的，微微生气之余，简单的聊聊“如何精确的理解leader布置的任务”。

**【员工角度的潜在困惑】**

1）leader讲了很多，脑子记不住

2）带了纸和笔，却不知道记什么

3）记了很多，却不知道“自己所理解leader想要的”和“leader真实想要的”是否一致

4）不敢问，心想先做出来结果再说，如果不是leader想要的，leader届时肯定会指正（其实已经晚了，可能最佳时间已经过了，不好的印象可能已经留下了）

这些困惑大伙会不会有，或者还有哪些困惑？

**【职场不会有人和你说】**

**如何精准理解leader布置的任务？**

不只为留下一个“靠谱”的印象，更重要的是自身“职业性”的提升，个人的经验，分三步走：

**一，快速反馈**

不是快速响应，不是停下手头的事情，直接杀去leader工位听leader布置任务。

是让leader知道，已经收到相关的信息：

“收到，马上过来”

“收到，紧急么，正在处理一个线上问题，不紧急的话30分钟之后过来”

一般来说，leader询问和布置的工作是leader直接关注的，重要性会很高，紧急程度不确定的话可以直接向leader询问，停下手中高优的工作，直接杀奔去leader工位，也是不可取的。

千万不要因为手头有很重要的工作而不反馈，leader不清楚有没有收到信息，而需要反复提醒，这一点是很危险的。

**二，倾听记录**

要去leader工位，听leader开始布置任务了，**是否需要带纸和笔？**

回答：需要，不确定leader布置的任务有多复杂，万一脑子记不下呢，准备多一点肯定不会错。

带了纸和笔，**要记哪些东西？**

回答：工作任务安排，无非记录5W2H2R

5W-> why, who, when, where, what: 为什么要做，希望谁，在什么时间，什么地方，完成什么事情

2H-> how, how much: 希望怎么做，做到什么程度

2R-> resource, result: 有什么资源支持，希望获得什么结果

**三，复述确认**

不敢问，是一个大问题，自己心中的疑问，一定要向leader提问，5W2H2R中未包含全面的要素，也可以向leader提问。

当然，有一些要素leader不一定提供，例如how，怎么做这项工作，这是后续需要去解决的。

“自己所理解leader想要的”和“leader真实想要的”是否一致，复述和确认就显得很重要，将自己记录的和理解的复述一边，和leader进行确认：

“因为why的原因，需要who在when时间where地点（通过how方法）完成what事情（做到how much程度），有resource资源支持，期望达到result结果，是这样么？”

如果leader反馈了理解上的分歧，需要进一步消除分歧，最终达成共识。更正式的，可以将记录的要点发出mail，一来正式，二来备忘，三来如果需要相关方支持起到通知的作用。

**【总结】**

精确理解leader布置的任务，**三个步骤**：**快速反馈、倾听记录、复述确认**。

职场这些小事，或许不会有人和我们说。

不知道大伙有没有类似的经验或者困惑？

或者大伙觉得哪些点是leader需要做好的？

==【完】==

相关推荐：

[晋升的为什么不是你](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404332602&idx=1&sn=a50ed411c893f6aeadbd66a25d31b94c&scene=21" \l "wechat_redirect" \t "_blank)

## 如何快速精确的和leader沟通

原创 2017-04-29 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960073&idx=1&sn=3ac3bb78bb643d73e9e68d713d6eceeb&chksm=bd2d06d58a5a8fc378e7cde71808c09dea9b4d5462deec0238a3d6f0ade62976a3d2bd0faafb&scene=21" \l "#)

**【缘起】**

一个同学找我讨论个事情，沟通了一会还是不确定要表达什么，希望我配合什么。结合自己的经验，简单的聊聊“如何**快速**精准的和leader沟通一件事”。

**【员工角度的潜在困惑，以及我作为leader内心的真实想法】**

疑问一：leader时间很宝贵，会不会占用太多时间，导致影响leader工作节奏？

**内心独白**：

事情确实不少，但**我存在的价值不就是帮助大家解决问题么**，期望你能尽快将问题描述清楚，我会综合手头要处理事情的重要紧急程度，判断是“现场沟通”，还是“一会找你”，或者“某个时间点找我”。

疑问二：leader所说的“一会找你”是敷衍么？

**内心独白**：

兄弟，你想多了，“一会找你”就是字面的意思。

一般来说，当手头有更重要的事情时，我会说“一会找你”。如无意外，这个事情将被写进schedule，稍后插空完成。极端情况下，如果没有找你，那是我真的忘了，请在即时通讯工具上，友善小窗提醒我，我一定会响应，并表达歉意。

疑问三：leader为何会说“某个时间点来找我”？

**内心独白**：

一般比较正式，需要20分钟以上的沟通和讨论，会单独预留固定的时间和你一起商量把事情解决。

千万不要迟到，迟到可能打乱我的工作节奏。

其他困惑？欢迎讨论。

**【职场不会有人和你说】**

如何和leader快速+精准的沟通一件事？

**一，表明期望**

用最简明的几句话说清楚希望leader配合什么事情，第一步往往是很多人做的不好的，经常的开头是这样的：

“早上，我们和PM及业务方对XXX事情进行了讨论，业务方认为YYY，产品认为ZZZ，但我们认为这样会导致WWW…”

这类**事实陈述式开头**，不利于leader了解需要沟通的主题，以及需要配合的事项，这样leader可能难以判断事情的紧急优先程度。

如果是需要**征得leader同意**，可以这样开头：

“XXX某个项目的排期，需要你确认一下，这个项目是关于YYY…”

如果是**请求leader帮助**，可以这样开头：

“XXX运维工单，需要你通过一下，这个运维工单是关于YYY…”

如果是**工作汇报，或者是交流讨论**，可能需要较长的时间，不宜莽撞的临时抽空当面交流，**最好只是约一个时间**：

“XXX项目的设计评审，不知道你什么时候有空，这个项目是关于YYY…”

无论如何，快速说明来意，第一时间让leader清楚需要配合什么，以及事情的紧急优先程度是什么样的，而不是一上来就陈述细节。

**二，来龙去脉**

表面来意之后，需要进一步说明更多事情的细节，对来龙去脉做一个简单的描述，这个过程“逻辑性”非常重要。

**事实性描述**，可以采用先后时序：

“先xxx，然后yyy，最后zzz”

也可以采用分角色立场描述：

“pm认为xxx，qa觉得，我们认为zzz”

**决策性描述**，一定要说明理由：

“建议采用xxx方案，因为xxx原因一，二，三，所以xxx方案我认为是最佳选择”

这种“论点，论据，强调论点”的表达方式，逻辑性强，重点突出，强烈推荐。

**三，复述确认**

对沟通目的，来龙去脉做了简要称述，最后对期望leader要配合的事情做复述和确认。这个过程，最重要的是**“强烈的行动导向”**，为leader做出的决策和行动提供强有力的信心（例如进一步计划）：

“好的，如果确认使用方案XXX，后续我们将YYY”

“如果确认明天下午XXX时间有空，后续我们会YYY”

“既然同意了XXX排期，后续我们则YYY”

这个最终的确认非常重要，**所有的沟通都是为了明确最终的这个结果**，最怕误以为双方达成了一致，其实非常含糊。

**【总结】**

“快速”“精确”和leader完成沟通，三个步骤：

（1）**表明期望**：简洁，明确

（2）**来龙去脉**：逻辑性强

（3）**复述确认**：行动导向

职场这些小事，或许不会有人和我们说，不知道大伙有没有类似的经验或者困惑？欢迎分享。

==【完】==

推荐文章：

[不为迟到找借口](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960065&idx=1&sn=94cfa99d6244d68a6d1c4ca8aeb723d7&chksm=bd2d06dd8a5a8fcb406d89e8817892948e67d7bb8adca5a5004ae46dde40099e12a39754be9b&scene=21" \l "wechat_redirect" \t "_blank)

[如何精确理解leader布置的任务](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960035&idx=1&sn=e0a14e8539b001e4d6d149164987235b&chksm=bd2d073f8a5a8e290202fcdfac077744eab84e6549e0608698ae5e1d3ed221ee97ca21f42e6f&scene=21" \l "wechat_redirect" \t "_blank)

## 不为迟到找借口

原创 2017-04-22 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960065&idx=1&sn=94cfa99d6244d68a6d1c4ca8aeb723d7&chksm=bd2d06dd8a5a8fcb406d89e8817892948e67d7bb8adca5a5004ae46dde40099e12a39754be9b&scene=21" \l "#)

工作中有非常多的会议，而“迟到”是会议的大敌，常见的情景是：

==现象==

1）约定10:00会议

2）结果10:00人没有来齐

3）等到10:15人来齐，会议才正式开始

==发展==

4）按时与会的同学受到伤害，准时与会反而要等待

5）大家都是聪明人，按时与会的同学下次也10:15分钟到场，反正准时到也开始不了

6）下次约10:00的会议，所有同学都默认10:15分才能正式开始

==恶化==

7）下次10:15人仍没有到齐

8）等到10:30人才到齐，会议才正式开始

9）…递归，迟到时间越来越长

**常见应对方法一**：不管人是否到齐，会议准时开始

**结果**：有时候迟到的人是核心与会人员，行不通

**常见应对方法二**：迟到者必须说明合理原因

**结果**：迟到者一定有“合理原因”

**常见应对方案三**：对迟到者实施惩罚，例如迟到1分钟罚10元

**结果**：无法监督，难以实施

不知道大伙的公司有没有遇到这类现象：遵守规则的人吃亏，最后风气越来越差，劣币驱逐良币，不知道大伙的公司是怎么避免会议迟到现象的？

**58到家的实践**：任何提前24小时预约的会议，迟到者请未迟到者喝咖啡，星巴克。

实施后效果甚好，迟到者越来越少，看对大伙是否有帮助。守时不是对员工的高要求，而是职场的基本素质。

**不为迟到找借口，只为咖啡找星巴克。**

共勉。

==【完】==

相关推荐：

[如何精确理解leader布置的任务](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960035&idx=1&sn=e0a14e8539b001e4d6d149164987235b&chksm=bd2d073f8a5a8e290202fcdfac077744eab84e6549e0608698ae5e1d3ed221ee97ca21f42e6f&scene=21" \l "wechat_redirect" \t "_blank)

[晋升的为什么不是你](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404332602&idx=1&sn=a50ed411c893f6aeadbd66a25d31b94c&scene=21" \l "wechat_redirect" \t "_blank)

还有一个“厕所抽烟”的难题，至今没有解决，有什么好的方案么？

## 运维说给研发测试的心底话

原创 2017-05-02 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960076&idx=1&sn=bf57176ba98c6e8c376ee6705405c925&chksm=bd2d06d08a5a8fc64ef401a40c24ae92cfab43c25a60c1f95e1ab6d72ca770c8f825b0a03c23&scene=21" \l "#)

互联网，讲究快速迭代，快速上线，敏捷开发。

有些**固定上线时间**的项目，可能因为技术方案变化，导致测试时间压缩，最终导致上线出问题，而由运维来背锅。

为保住KPI，运维有很多心里话想和研发测试说一说：

（1）“**敏捷开发，频繁交付**”的KPI，真不是增加运维人手就能解决的，需要自动化回归的支持，需要自动化上线的支持

（2）“**上线失败，快速回滚**”的KPI，真不是增加运维人手就能解决的，需要回滚方案的支持，而回滚方案真的测试过么

（3）“**快速扩容，快速响应**”的KPI，真不是增加运维人手就能解决的，需要架构设计的支持（很多系统无法水平扩展，来了机器，无法扩容），需要快速部署的支持，需要服务发现的支持（所有上游修改配置重启肯定是不行的），需要压力测试和容量评估的支持

（4）“**系统高可用**”的KPI，真不是增加运维人手就能解决的，需要优雅降级的支持，需要架构设计的支持，如何评判系统是否高可用？这个简单，关掉线上任何一台机器试试，看用户服务是否受影响，如果受影响，研发哥哥们拜托了

（5）“**快速故障报警**”的KPI，真不是增加运维人手就能解决的，需要监控系统的支持（操作系统和运维层面的监控，我们可以实施，但错误日志、接口、业务的监控呢？），另外报警短信能少一点么，过度报警会让人变得“麻木不仁”的

（6）“**快速故障定位**”的KPI，真不是增加运维人手就能解决的，需要数据量化健康信息的支持（58到家的守望者平台做的还是不错的），需要快速诊断的支持（58到家的调用链跟踪系统做的还是不错的）

（7）“**快速故障恢复**”的KPI，真不是增加运维人手就能解决的，需要故障转移的支持，相信我们，故障发生时，如果运维人员不知道怎么抉择，且又必须做出抉择，这时的抉择往往是错的（我们能做的，是重启），我们也不想凌晨打给你们，但希望你们能实现自动化方案

（8）“**内审合规**”的KPI，真不是增加运维人手就能解决的，在资源允许的情况下，请不要手动删除任何资源，数据是很重要的资源。访问控制和权限申请的流程，真的不是限制大家，相反，哪一次数据的误删除，不是我们加班来恢复的？宝宝心里苦呀

我们的KPI都掌握在大家的手里，技术一家人，希望研发测试的同学理解。

大家还有什么苦水？一起说一说。

这个问题，大家也自己问一问，关掉线上任何一台机器，用户服务受影响么？

==【完】==

相关推荐：

[换IP的是你，凭啥重启的却是我？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960049&idx=1&sn=2787e73be4ea31f52213a2b934a16e09&chksm=bd2d072d8a5a8e3b71cef58f313743a6db6f83889748deb7f8f866817511df98adc5e03db39c&scene=21" \l "wechat_redirect" \t "_blank)

[58到家数据库30条军规解读](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959906&idx=1&sn=2cbdc66cfb5b53cf4327a1e0d18d9b4a&chksm=bd2d07be8a5a8ea86dc3c04eced3f411ee5ec207f73d317245e1fefea1628feb037ad71531bc&scene=21" \l "wechat_redirect" \t "_blank)

[再议数据库军规](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959910&idx=1&sn=6b6853b70dbbe6d689a12a4a60b84d8b&chksm=bd2d07ba8a5a8eac6783bac951dba345d865d875538755fe665a5daaf142efe670e2c02b7c71&scene=21" \l "wechat_redirect" \t "_blank)

[即使删了全库，保证半小时恢复](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959694&idx=1&sn=4ac53b797fca64229901373e793f860a&chksm=bd2d04528a5a8d44618333de549094569f9ec58833ab64d081bc826b9361f587aca2972fc4c6&scene=21" \l "wechat_redirect" \t "_blank)

## 职场中的选择与拒绝

原创 2017-05-11 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960095&idx=1&sn=227b8e3ee3027fcf2180fa12f771995e&chksm=bd2d06c38a5a8fd58610bd771b18ddf257e14dc2248101b249ae59432ed257d4bbfd02f2efaf&scene=21" \l "#)

**一**

毕业时找工作，大学生SK面临着几个选择，阿朗贝尔实验室，移动通讯，百度后端。

要是你，你选择哪个，拒绝哪个？

最终，SK选择了后者，在百度，他掌握了**职业初期的技术基础技能**。

**二**

百度即时通讯组解散（SK还挺悲剧的），工程师SK面临着几个选择，转岗去无线组做后端开发，还是转岗去大搜组做策略？

要是你，你选择哪个，拒绝哪个？

最终，SK选择了后者，虽然不能再做即时通讯，既然来了百度，他就要学习百度最核心的技术。

**三**

转岗还不到半年，同城要组建即时通讯团队，高级工程师SK是要继续留在百度大搜，还是换地儿接着做即时通讯？

要是你，你选择哪个，拒绝哪个？

最终，SK选择了后者，在新公司的即使通讯团队，他提升了即时通讯架构的**技术核心竞争力**。

**四**

两年后即时通讯组发展相对成熟，系统相对稳定，架构师SK带的队伍趋于平稳，需求过来，按节奏响应即可，如果在线量不到千万，或许架构不再需要大的升级，怎么办？困惑ing，困惑ing。继续做即时通讯或许个人提升较慢，还是不再带队，归零心态，换个业务试一试？

要是你，你选择哪个，拒绝哪个？

最终，SK选择了后者，做支付/做摊销/做数据库中间件/做推荐系统/做APP原生化，在各个大项目的洗礼中，他提升了的**业务知识与架构技能的广度**。

**五**

又过了两年，集团组织架构调整，高级架构师SK有机会去向往已久但相对成熟的架构部，还是去新事业部负责一个刚成立的4人小组？

要是你，你选择哪个，拒绝哪个？

最终，SK选择了后者，做二手，做心宠，做优品，做转转架构，很短的时间，团队成型并能够转起来，他提升了**团队的管理能力**。

**六**

2015年，二手/心宠/优品20人团队雏形已备，转转架构设计已定，SK又有只身一人到到家负责大后端的机会。

要是你，你选择哪个，拒绝哪个？

最终，SK选择了后者，从开始招大后端的第一个人开始，到今天。

**节奏**

这是一个看上去平淡的故事，互联网的技术人，SK简单的总结的节奏是：

* 1-2年，打基本功非常重要
* 3-4年，一定要有一技之长及核心竞争力
* 5-6年，可以成为多面手，或者可以成为小组长
* 7-8年，可以成为专家，或者可以成为有经验的管理者

天赋异禀，或者有好的机会，可以走的更快。

走慢了，看是自己太浮躁，还是哪里出了问题。

**启示**

每一次选择和拒绝，SK还领悟到：

* 工作的开心比什么都重要，不开心多和leader聊一聊
* 技术的提升/业务知识的学习很重要，这些才是核心竞争力
* 公平的环境很重要，要相信，做出了成绩老大一定能看得见，一定会有相应的回报，如果不是这样，早点换一家公司
* 技术氛围很重要
* 如果团队志同道合，**坚持走下来**，往往收获是最大的

共勉！

===【完】===

## 罗振宇送给新员工的四句话

原创 2017-06-04 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960162&idx=1&sn=0a4814e76ae98f7b4597205bc9b4402c&chksm=bd2d06be8a5a8fa8ea9cac53144d780160c853329af461b6ffa2d8002f93425c394140d1edb3&scene=21" \l "#)

看了最新的一期奇葩说《在职场被冤枉，到底要不要澄清》，原来觉得罗振宇是个奸商，这次看来要转粉了，以自己写架构文章的方式记录下来其一些观点，分享给大家吧。

**职场第一大忌是什么？**

**答**：遇到问题，搁置或绕路

**职场如何消解自己的负面情绪？**

**答**：职场的负面情绪一般是因为某个人，**解决方案**是找ta沟通，沟通的结果有两个：

* 问题解决了
* 没解决，但自己解脱了

**职场最没有前途的是什么人？**

**答**：“反馈黑洞”，即不沟通，不同步信息，这样的人越能干，效果越负向。

在职场，没有沟通能力，其他能力都归零。

**职场中遇到谣言，澄清不了，就不澄清了么？**

**答**：澄清是为了说服谁吗？这个逻辑无异于：

* 读一本书，就是为了有用
* 投一笔资，就是为了盈利
* 做一件事情，就是为了有回报

成人的世界难道是这个逻辑？

**成人世界的逻辑**应该是“去下注大概率成功的事情”：

* 高考成功的概率比不读书去做生意成功的概率更高，所以应该去参加高考
* 投资风口中迅猛发展的企业，财富增值的概率高，于是投资这些企业
* 读书比刷朋友圈更有可能有收获，于是就读书

讲事实不能说服说有人，但事实就不重要么？

故，澄清是必要的。

**职场中最重要的能力是什么？**

**答**：表达能力。

传统社会最重要的资产是财富和权力，未来社会最重要的资产是影响力。

影响力如何构成：

* 写作
* 演讲

这个社会，有啥事是不能够通过一顿撸串解决的呢？如果有，那就两顿；

人在职场，有啥事是不能够通过一次沟通解决的呢？如果有，那就两次。

**职场中怎么样的态度才能够“如鱼得水”？**

**答**：诚恳

诚恳的沟通，让自己包裹在事实当中，最稳妥。

没有任何道路通向真诚，真诚是通向一切的道路。

总结，在职场中混，罗振宇送给新员工的四句话：

* **目标清晰，想方法和行动**
* **做大概率会赢的事情**
* **沟通和表达是最重要的**
* **找不到态度，就回到真诚**