<http://mp.weixin.qq.com/s/OlFKpcnBOgcPZmjvdzCCiA>

## 架构师之路2016年精选50篇

原创 2017-02-11  58沈剑 [架构师之路](http://mp.weixin.qq.com/s/OlFKpcnBOgcPZmjvdzCCiA" \l "#)

2016精选索引，**点击标题阅读**相关文章。

**【方法论】**

《**[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959391&idx=1&sn=fb28fd5e5f0895ddb167406d8a735548&scene=21" \l "wechat_redirect" \t "_blank)**》

《[分布式ID生成器](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "wechat_redirect" \t "_blank)》

《[互联网架构，如何进行容量设计](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959542&idx=1&sn=2494bbea9a855e0e1c3ccd6d2562a600&scene=21" \l "wechat_redirect" \t "_blank)》

《[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)》

《[单点系统架构的可用性与性能优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959480&idx=1&sn=337bd74410a6bef616128fd17abd08a8&scene=21" \l "wechat_redirect" \t "_blank)》

《[关于负载均衡的一切](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959585&idx=1&sn=0a9222cbfeb62a662edffafb7f0b43ae&scene=21" \l "wechat_redirect" \t "_blank)》

《[异构服务器负载均衡及过载保护](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959601&idx=1&sn=5684c39676b1f6d9366d9d15a2cdcec3&scene=21" \l "wechat_redirect" \t "_blank)》

《[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)》

《[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)》

《**[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)**》

**【数据库与缓存】**

《**[数据库架构设计的一切](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400465735&idx=1&sn=8d7067de4cc8f73ea5558f07e0a9340e" \l "wechat_redirect" \t "_blank)**》

《[缓存架构细节二三事](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404087915&idx=1&sn=075664193f334874a3fc87fd4f712ebc&scene=21" \l "wechat_redirect" \t "_blank)》

《[数据冗余一致性优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403963671&idx=1&sn=51a2d2fd70212451cd5f22bbe2c6f8d6&scene=21" \l "wechat_redirect" \t "_blank)》

《[缓存与数据库一致性优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404202261&idx=1&sn=1b8254ba5013952923bdc21e0579108e&scene=21" \l "wechat_redirect" \t "_blank)》

《[主从DB与Cache一致性优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404308725&idx=1&sn=1a25ce76dd1956014ceb8a011855268e&scene=21" \l "wechat_redirect" \t "_blank)》

《[DB主库与从库一致性优化4种方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959442&idx=1&sn=feb8ff75385d8031386e120ef3535329&scene=21" \l "wechat_redirect" \t "_blank)》

《[多库多事务一致性优化方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959486&idx=1&sn=cd3e6b27af159f4eabd54f148eec638a&scene=21" \l "wechat_redirect" \t "_blank)》

《[mysql并行复制优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959461&idx=1&sn=c4c9317f55075541be00d59039b88470&scene=21" \l "wechat_redirect" \t "_blank)》

《[互联网公司为何不使用mysql分区表](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959604&idx=1&sn=9f3e21f5ff99abc5a16038a77f31843e&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)》

《[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)》

《[续集：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)》

《[瞬间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)**》

**【服务化与为服务】**

《**[互联网架构为什么要做服务化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959519&idx=1&sn=065074b135fc9cb243abe897261e1a72&scene=21" \l "wechat_redirect" \t "_blank)**》

《[微服务架构究竟多“微”合适](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959530&idx=1&sn=ff84bd74745eee1577e7dfef8ce66bbe&scene=21" \l "wechat_redirect" \t "_blank)》

《[要想微服务，先搞定RPC框架](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959553&idx=1&sn=c1084e91875721c5f6baf544450afa38&scene=21" \l "wechat_redirect" \t "_blank)》

《[RPC-client序列化原理与细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959558&idx=1&sn=610f06c6d62a5c22311d27cf40f758ef&scene=21" \l "wechat_redirect" \t "_blank)》

《[RPC-client异步收发细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959576&idx=1&sn=2be8d3f61effe7118abf920a175da710&scene=21" \l "wechat_redirect" \t "_blank)》

**【消息系统】**

《**[微信为什么不丢消息](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959606&idx=1&sn=f9561231dd33bcd0550b8d0d59d6b876&chksm=bd2d04ea8a5a8dfce90c870279a7f74b7aedd802c2d699dd919d7e40ebe30699381517c2d54b&scene=21" \l "wechat_redirect" \t "_blank)**》

《[微信为什么不丢“离线”消息](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959629&idx=1&sn=053d85a862df6e6c01147a1cf95bdbf2&chksm=bd2d04918a5a8d87a32305fa8ef1603bb0e73e5c4b78afb5a144327841e08840188c4541b4aa&scene=21" \l "wechat_redirect" \t "_blank)》

《[群消息这么复杂，怎么做到不丢不重](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959643&idx=1&sn=844afa6a31770fa587474ecd73c3b3b3&chksm=bd2d04878a5a8d91c5c93ad8e85254185c63eb419457efbedcba54a9b8a9053da1e8980a694a&scene=21" \l "wechat_redirect" \t "_blank)》

《[多点登陆，消息漫游架构随想](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959651&idx=1&sn=959540719a8336bd6f507a324dc7219a&chksm=bd2d04bf8a5a8da9ddfaeb106b61386adeb51c2ad7d4a32589c9a9e113f9341604a490eebb98&scene=21" \l "wechat_redirect" \t "_blank)》

《[QQ状态同步究竟是推还是拉](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959618&idx=1&sn=912a9af6d000c6681dc645e53590729b&chksm=bd2d049e8a5a8d884f6acb35bc5b40edf88127219542c9e3033bb4c2e33854657c315c8a93b7&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)》

《[58到家通用消息平台架构细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "wechat_redirect" \t "_blank)》

《[微信为什么这么省流量](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959676&idx=1&sn=0204f47f3ecad9034d0a3607914d25c3&chksm=bd2d04a08a5a8db61f7d98d811b703847a5c7e04356b70fbd7ae741848eb427ed17d31011ae3&scene=21" \l "wechat_redirect" \t "_blank)》

《[即时通讯协议设计选型](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959701&idx=1&sn=89850966b5e807097f8d273009104c50&chksm=bd2d04498a5a8d5f49e98825accf6fc3d3dd4edd49d92a0012befa992454c82191cbd4efa4be&scene=21" \l "wechat_redirect" \t "_blank)》

《**[http如何像tcp一样实时的收消息](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959605&idx=1&sn=21f25087bef3c3a966ef03b824365621&scene=21" \l "wechat_redirect" \t "_blank)**》

**【架构实践】**

《**[58同城架构演进，流量从0到10亿](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400276397&idx=1&sn=ea044079667b82f6cad58bcb743af7bc&scene=21" \l "wechat_redirect" \t "_blank)**》

《[58同城推荐系统架构设计实践](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959464&idx=1&sn=81c29bf2e237499be07deba826f73c39&scene=21" \l "wechat_redirect" \t "_blank)》

《[58转转从0开始推荐系统实践](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=402978440&idx=1&sn=f0ec24d9f3c81fe49868d12a5128fcc9&scene=21" \l "wechat_redirect" \t "_blank)》

《[58到家O2O快速个性化推荐](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403209303&idx=1&sn=f4fea7390b3a2c31e693189aee44af59&scene=21" \l "wechat_redirect" \t "_blank)》

《[58到家入驻微信钱包的架构优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959423&idx=1&sn=19d27145beab5b0238280e2e3804e41b&scene=21" \l "wechat_redirect" \t "_blank)》

《[58到家快速搭建立体化监控平台之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959433&idx=1&sn=9c9a222c06b2426f3935b04a10f90c0b&scene=21" \l "wechat_redirect" \t "_blank)》

《[高并发下，余额扣减一致性实践](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400556481&idx=1&sn=42bc187b45e4e53aece4b8c1f43e8019&scene=21" \l "wechat_redirect" \t "_blank)》

《[百度是怎么做长文本去重的](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)》

《[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)》

《[快速实现高并发无锁缓存](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959502&idx=1&sn=7a1c7d81a8e030856fb920844dc571c6&scene=21" \l "wechat_redirect" \t "_blank)》

《[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)》

《**[58到家从IDC到云架构平滑迁移实践](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959495&idx=1&sn=0f8789d6803f94e190ba2787389e6d1b&scene=21" \l "wechat_redirect" \t "_blank)**》

**【一分钟系列】**

《[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分钟了解两阶段提交2PC](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400856212&idx=1&sn=b9734da77f2259bdd1fa89700b47eaff&scene=21" \l "wechat_redirect" \t "_blank)》

《[1分钟1副图彻底搞懂join](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400889133&idx=1&sn=3ea1c06fc76e92349a822f41f45b77b5&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分钟实现分布式锁](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959749&idx=1&sn=6727638930905089342662c24f5bba62&chksm=bd2d04198a5a8d0f619f1b9be1ead542f015e6ff87b5aac5a61af94b714459964fedb89e6516&scene=21" \l "wechat_redirect" \t "_blank)》

《[续集：这才是真正的分布式锁](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959755&idx=1&sn=e0875b539895578ce966fdad7f3c0365&chksm=bd2d04178a5a8d01a66d30b5ac8958fc5a6108e6e1d90a265361a7d3b30197bcb51f4433f23b&scene=21" \l "wechat_redirect" \t "_blank)》

**【通用素质】**

《[心态：晋升的为何不是你](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404332602&idx=1&sn=a50ed411c893f6aeadbd66a25d31b94c&scene=21" \l "wechat_redirect" \t "_blank)》

《[鸡汤：你的收入取决于你的努力程度](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959534&idx=1&sn=1f2f3b92332439cd65a3f1e69391f55f&scene=21" \l "wechat_redirect" \t "_blank)》

《[爱女友：“我穿这衣服好看吗”终于破解了](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959561&idx=1&sn=60fcc75435a80280aebe8faffd28fda5&scene=21" \l "wechat_redirect" \t "_blank)》

《[管理：一分钟经理人](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959737&idx=1&sn=5d6f3fd25f6473e77f54269a1b384565&chksm=bd2d04658a5a8d7389fe8cb761dcc494498e5a330fb4b9db1d760d35f7ce8df26fc788227e6d&scene=21" \l "wechat_redirect" \t "_blank)》

《[访谈：架构师到底该不该写代码](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959776&idx=1&sn=10a8bcb1277fdd6bfb9e24b816697038&chksm=bd2d043c8a5a8d2a9b3e6638c855da30e2910b17a7f6d435118fb1fe09a99fb85950d98b4d63&scene=21" \l "wechat_redirect" \t "_blank)》

《**[如何做一场B格满满的技术大会演讲](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959675&idx=1&sn=98c61e127f4b82ca3ba86ff369dfdb3c&chksm=bd2d04a78a5a8db1f5de8c39a1398b8812ff2b9961642f7b813293ade34f602d42ea127b85aa&scene=21" \l "wechat_redirect" \t "_blank)**》

若有收获，帮忙转发。

2017，继续上路，**你的支持是我最大的动力**。

## 秒杀系统架构优化思路

原创 2016-04-06 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959391&idx=1&sn=fb28fd5e5f0895ddb167406d8a735548&scene=21" \l "#)

本文曾在“架构师之路”上发布过，近期支援Qcon-AS大会，在微信群里分享了该话题，故对原文进行重新整理与发布。

### ****一、秒杀业务为什么难做****

1）im系统，例如qq或者微博，每个人都读自己的数据（好友列表、群列表、个人信息）；

2）微博系统，每个人读你关注的人的数据，一个人读多个人的数据；

3）秒杀系统，库存只有一份，所有人会在集中的时间读和写这些数据，多个人读一个数据。

例如：小米手机每周二的秒杀，可能手机只有1万部，但瞬时进入的流量可能是几百几千万。

又例如：12306抢票，票是有限的，库存一份，瞬时流量非常多，都读相同的库存。**读写冲突，锁非常严重，这是秒杀业务难的地方**。那我们怎么优化秒杀业务的架构呢？

### ****二、优化方向****

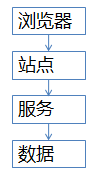
优化方向有两个（今天就讲这两个点）：

（1）**将请求尽量拦截在系统上游**（不要让锁冲突落到数据库上去）。传统秒杀系统之所以挂，请求都压倒了后端数据层，数据读写锁冲突严重，并发高响应慢，几乎所有请求都超时，流量虽大，下单成功的有效流量甚小。以12306为例，一趟火车其实只有2000张票，200w个人来买，基本没有人能买成功，请求有效率为0。

（2）**充分利用缓存**，秒杀买票，这是一个典型的读多写少的应用场景，大部分请求是车次查询，票查询，下单和支付才是写请求。一趟火车其实只有2000张票，200w个人来买，最多2000个人下单成功，其他人都是查询库存，写比例只有0.1%，读比例占99.9%，非常适合使用缓存来优化。好，后续讲讲怎么个“将请求尽量拦截在系统上游”法，以及怎么个“缓存”法，讲讲细节。

### ****三、常见秒杀架构****

常见的站点架构基本是这样的（绝对不画忽悠类的架构图）



（1）**浏览器端**，最上层，会执行到一些JS代码

（2）**站点层**，这一层会访问后端数据，拼html页面返回给浏览器

（3）**服务层**，向上游屏蔽底层数据细节，提供数据访问

（4）**数据层**，最终的库存是存在这里的，mysql是一个典型（当然还有会缓存）

这个图虽然简单，但能形象的说明大流量高并发的秒杀业务架构，大家要记得这一张图。

后面细细解析各个层级怎么优化。

### ****四、各层次优化细节****

**第一层，客户端怎么优化（浏览器层，APP层）**

问大家一个问题，大家都玩过微信的摇一摇抢红包对吧，每次摇一摇，就会往后端发送请求么？回顾我们下单抢票的场景，点击了“查询”按钮之后，系统那个卡呀，进度条涨的慢呀，作为用户，我会不自觉的再去点击“查询”，对么？继续点，继续点，点点点。。。有用么？平白无故的增加了系统负载，一个用户点5次，80%的请求是这么多出来的，怎么整？

（a）**产品层面**，用户点击“查询”或者“购票”后，按钮置灰，禁止用户重复提交请求；

（b）**JS层面**，限制用户在x秒之内只能提交一次请求；

APP层面，可以做类似的事情，虽然你疯狂的在摇微信，其实x秒才向后端发起一次请求。这就是所谓的“将请求尽量拦截在系统上游”，越上游越好，浏览器层，APP层就给拦住，这样就能挡住80%+的请求，这种办法只能拦住普通用户（但99%的用户是普通用户）对于群内的高端程序员是拦不住的。firebug一抓包，http长啥样都知道，js是万万拦不住程序员写for循环，调用http接口的，这部分请求怎么处理？

**第二层，站点层面的请求拦截**

怎么拦截？怎么防止程序员写for循环调用，有去重依据么？ip？cookie-id？…想复杂了，这类业务都需要登录，用uid即可。在**站点层面**，对uid进行请求计数和去重，甚至不需要统一存储计数，直接站点层内存存储（这样计数会不准，但最简单）。一个uid，5秒只准透过1个请求，这样又能拦住99%的for循环请求。

5s只透过一个请求，其余的请求怎么办？缓存，页面缓存，同一个uid，限制访问频度，做页面缓存，x秒内到达站点层的请求，均返回同一页面。同一个item的查询，例如车次，做页面缓存，x秒内到达站点层的请求，均返回同一页面。如此限流，既能保证用户有良好的用户体验（没有返回404）又能保证系统的健壮性（利用页面缓存，把请求拦截在站点层了）。

页面缓存不一定要保证所有站点返回一致的页面，直接放在每个站点的内存也是可以的。优点是简单，坏处是http请求落到不同的站点，返回的车票数据可能不一样，这是站点层的请求拦截与缓存优化。

好，这个方式拦住了写for循环发http请求的程序员，有些高端程序员（黑客）控制了10w个肉鸡，手里有10w个uid，同时发请求（先不考虑实名制的问题，小米抢手机不需要实名制），这下怎么办，站点层按照uid限流拦不住了。

**第三层 服务层来拦截（反正就是不要让请求落到数据库上去）**

服务层怎么拦截？大哥，我是服务层，我清楚的知道小米只有1万部手机，我清楚的知道一列火车只有2000张车票，我透10w个请求去数据库有什么意义呢？没错，**请求队列！**

对于写请求，做请求队列，每次只透有限的写请求去数据层（下订单，支付这样的写业务）

1w部手机，只透1w个下单请求去db

3k张火车票，只透3k个下单请求去db

如果均成功再放下一批，如果库存不够则队列里的写请求全部返回“已售完”。

对于读请求，怎么优化？cache抗，不管是memcached还是redis，单机抗个每秒10w应该都是没什么问题的。如此限流，只有非常少的写请求，和非常少的读缓存mis的请求会透到数据层去，又有99.9%的请求被拦住了。

当然，还有**业务规则上的一些优化**。回想12306所做的，分时分段售票，原来统一10点卖票，现在8点，8点半，9点，...每隔半个小时放出一批：将流量摊匀。

其次，数据粒度的优化：你去购票，对于余票查询这个业务，票剩了58张，还是26张，你真的关注么，其实我们只关心有票和无票？流量大的时候，做一个粗粒度的“有票”“无票”缓存即可。

第三，一些业务逻辑的异步：例如下单业务与 支付业务的分离。这些优化都是结合 业务 来的，我之前分享过一个观点“**一切脱离业务的架构设计都是耍流氓**”架构的优化也要针对业务。

**好了，最后是数据库层**

浏览器拦截了80%，站点层拦截了99.9%并做了页面缓存，服务层又做了写请求队列与数据缓存，每次透到数据库层的请求都是可控的。db基本就没什么压力了，闲庭信步，单机也能扛得住，还是那句话，库存是有限的，小米的产能有限，透这么多请求来数据库没有意义。

全部透到数据库，100w个下单，0个成功，请求有效率0%。透3k个到数据，全部成功，请求有效率100%。

### ****五、总结****

上文应该描述的非常清楚了，没什么总结了，对于秒杀系统，再次重复下我个人经验的两个架构优化思路：

（1）**尽量将请求拦截在系统上游**（越上游越好）；

（2）**读多写少的常用多使用缓存**（缓存抗读压力）；

浏览器和APP：做限速

站点层：按照uid做限速，做页面缓存

服务层：按照业务做写请求队列控制流量，做数据缓存

数据层：闲庭信步

并且：结合业务做优化

### ****六、Q&A****

**问题1、按你的架构，其实压力最大的反而是站点层，假设真实有效的请求数有1000万，不太可能限制请求连接数吧，那么这部分的压力怎么处理？**

答：每秒钟的并发可能没有1kw，假设有1kw，解决方案2个：

（1）站点层是可以通过加机器扩容的，最不济1k台机器来呗。

（2）如果机器不够，抛弃请求，抛弃50%（50%直接返回稍后再试），原则是要保护系统，不能让所有用户都失败。

**问题2、“控制了10w个肉鸡，手里有10w个uid，同时发请求” 这个问题怎么解决哈？**

答：上面说了，服务层写请求队列控制

**问题3：限制访问频次的缓存，是否也可以用于搜索？例如A用户搜索了“手机”，B用户搜索“手机”，优先使用A搜索后生成的缓存页面？**

答：这个是可以的，这个方法也经常用在“动态”运营活动页，例如短时间推送4kw用户app-push运营活动，做页面缓存。

**问题4：如果队列处理失败，如何处理？肉鸡把队列被撑爆了怎么办？**

答：处理失败返回下单失败，让用户再试。队列成本很低，爆了很难吧。最坏的情况下，缓存了若干请求之后，后续请求都直接返回“无票”（队列里已经有100w请求了，都等着，再接受请求也没有意义了）

**问题5：站点层过滤的话，是把uid请求数单独保存到各个站点的内存中么？如果是这样的话，怎么处理多台服务器集群经过负载均衡器将相同用户的响应分布到不同服务器的情况呢？还是说将站点层的过滤放到负载均衡前？**

答：可以放在内存，这样的话看似一台服务器限制了5s一个请求，全局来说（假设有10台机器），其实是限制了5s 10个请求，解决办法：

1）加大限制（这是建议的方案，最简单）

2）在nginx层做7层均衡，让一个uid的请求尽量落到同一个机器上

**问题6：服务层过滤的话，队列是服务层统一的一个队列？还是每个提供服务的服务器各一个队列？如果是统一的一个队列的话，需不需要在各个服务器提交的请求入队列前进行锁控制？**

答：可以不用统一一个队列，这样的话每个服务透过更少量的请求（总票数/服务个数），这样简单。统一一个队列又复杂了。

**问题7：秒杀之后的支付完成，以及未支付取消占位，如何对剩余库存做及时的控制更新？**

答：数据库里一个状态，未支付。如果超过时间，例如45分钟，库存会重新会恢复（大家熟知的“回仓”），给我们抢票的启示是，开动秒杀后，45分钟之后再试试看，说不定又有票哟~

**问题8：不同的用户浏览同一个商品 落在不同的缓存实例显示的库存完全不一样 请问老师怎么做缓存数据一致或者是允许脏读？**

答：目前的架构设计，请求落到不同的站点上，数据可能不一致（页面缓存不一样），这个业务场景能接受。但数据库层面真实数据是没问题的。

**问题9：就算处于业务把优化考虑“3k张火车票，只透3k个下单请求去db”那这3K个订单就不会发生拥堵了吗？**

答：（1）数据库抗3k个写请求还是ok的；（2）可以数据拆分；（3）如果3k扛不住，服务层可以控制透过去的并发数量，根据压测情况来吧，3k只是举例；

**问题10；如果在站点层或者服务层处理后台失败的话，需不需要考虑对这批处理失败的请求做重放？还是就直接丢弃？**

答：别重放了，返回用户查询失败或者下单失败吧，架构设计原则之一是“fail fast”。

**问题11.对于大型系统的秒杀，比如12306，同时进行的秒杀活动很多，如何分流？**

答：垂直拆分

**问题12、额外又想到一个问题。这套流程做成同步还是异步的？如果是同步的话，应该还存在会有响应反馈慢的情况。但如果是异步的话，如何控制能够将响应结果返回正确的请求方？**

答：用户层面肯定是同步的（用户的http请求是夯住的），服务层面可以同步可以异步。

**问题13、秒杀群提问：减库存是在那个阶段减呢？如果是下单锁库存的话，大量恶意用户下单锁库存而不支付如何处理呢？**

答：数据库层面写请求量很低，还好，下单不支付，等时间过完再“回仓”，之前提过了。

 ==【完】==

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

回【晋升】[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)

回【id】[细聊分布式ID生成方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

【小游戏：回大于10的整数，随机返回好文，猜猜怎么实现的】

## 细聊分布式ID生成方法

原创 2016-02-22 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "#)

**一、需求缘起**

几乎所有的业务系统，都有生成一个记录标识的需求，例如：

（1）消息标识：message-id

（2）订单标识：order-id

（3）帖子标识：tiezi-id

这个记录标识往往就是数据库中的**唯一主键**，数据库上会建立聚集索引（cluster index），即在物理存储上以这个字段排序。

这个记录标识上的查询，往往又有分页或者排序的业务需求，例如：

（1）拉取最新的一页消息：selectmessage-id/ order by time/ limit 100

（2）拉取最新的一页订单：selectorder-id/ order by time/ limit 100

（3）拉取最新的一页帖子：selecttiezi-id/ order by time/ limit 100

所以往往要有一个time字段，并且在time字段上建立普通索引（non-cluster index）。

我们都知道普通索引存储的是实际记录的指针，其访问效率会比聚集索引慢，如果记录标识在生成时能够基本按照时间有序，则可以省去这个time字段的索引查询：

select message-id/ (order by message-id)/limit 100

再次强调，能这么做的前提是，message-id的生成基本是**趋势时间递增的**。

这就引出了记录标识生成（也就是上文提到的三个XXX-id）的两大核心需求：

（1）全局唯一

（2）趋势有序

这也是本文要讨论的核心问题：**如何高效生成趋势有序的全局唯一ID。**

**二、常见方法、不足与优化**

**【常见方法一：使用数据库的 auto\_increment 来生成全局唯一递增ID】**

**优点：**

（1）简单，使用数据库已有的功能

（2）能够保证唯一性

（3）能够保证递增性

（4）步长固定

**缺点：**

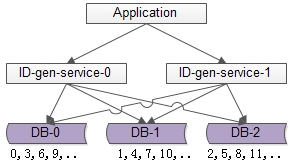
（1）可用性难以保证：数据库常见架构是一主多从+读写分离，生成自增ID是写请求，主库挂了就玩不转了

（2）扩展性差，性能有上限：因为写入是单点，数据库主库的写性能决定ID的生成性能上限，并且难以扩展

**改进方法：**

（1）增加主库，避免写入单点

（2）数据水平切分，保证各主库生成的ID不重复



如上图所述，由1个写库变成3个写库，**每个写库设置不同的auto\_increment初始值，以及相同的增长步长**，以保证每个数据库生成的ID是不同的（上图中库0生成0,3,6,9…，库1生成1,4,7,10，库2生成2,5,8,11…）

改进后的架构保证了可用性，但**缺点**是：

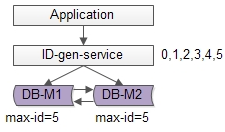
（1）丧失了ID生成的“绝对递增性”：先访问库0生成0,3，再访问库1生成1，可能导致在非常短的时间内，ID生成不是绝对递增的（这个问题不大，我们的目标是趋势递增，不是绝对递增）

（2）数据库的写压力依然很大，每次生成ID都要访问数据库

为了解决上述两个问题，引出了第二个常见的方案

**【常见方法二：单点批量ID生成服务】**

分布式系统之所以难，很重要的原因之一是“没有一个全局时钟，难以保证绝对的时序”，要想保证绝对的时序，还是只能使用单点服务，用本地时钟保证“绝对时序”。数据库写压力大，是因为每次生成ID都访问了数据库，可以使用批量的方式降低数据库写压力。



如上图所述，数据库使用双master保证可用性，数据库中只存储当前ID的最大值，例如0。ID生成服务假设每次批量拉取6个ID，服务访问数据库，将当前ID的最大值修改为5，这样应用访问ID生成服务索要ID，ID生成服务不需要每次访问数据库，就能依次派发0,1,2,3,4,5这些ID了，当ID发完后，再将ID的最大值修改为11，就能再次派发6,7,8,9,10,11这些ID了，于是数据库的压力就降低到原来的1/6了。

**优点**：

（1）保证了ID生成的绝对递增有序

（2）大大的降低了数据库的压力，ID生成可以做到每秒生成几万几十万个

**缺点**：

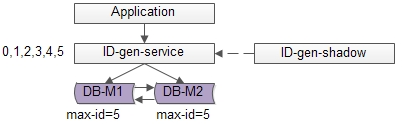
（1）服务仍然是单点

（2）如果服务挂了，服务重启起来之后，继续生成ID可能会不连续，中间出现空洞（服务内存是保存着0,1,2,3,4,5，数据库中max-id是5，分配到3时，服务重启了，下次会从6开始分配，4和5就成了空洞，不过这个问题也不大）

（3）虽然每秒可以生成几万几十万个ID，但毕竟还是有性能上限，无法进行水平扩展

**改进方法**：

单点服务的常用高可用优化方案是“备用服务”，也叫“影子服务”，所以我们能用以下方法优化上述缺点（1）：



如上图，对外提供的服务是主服务，有一个影子服务时刻处于备用状态，当主服务挂了的时候影子服务顶上。这个切换的过程对调用方是透明的，可以自动完成，常用的技术是vip+keepalived，具体就不在这里展开。

**【常见方法三：uuid】**

上述方案来生成ID，虽然性能大增，但由于是单点系统，总还是存在性能上限的。同时，上述两种方案，不管是数据库还是服务来生成ID，业务方Application都需要进行一次远程调用，比较耗时。有没有一种本地生成ID的方法，即高性能，又时延低呢？

uuid是一种常见的方案：string ID =GenUUID();

**优点**：

（1）本地生成ID，不需要进行远程调用，时延低

（2）扩展性好，基本可以认为没有性能上限

**缺点**：

（1）无法保证趋势递增

（2）uuid过长，往往用字符串表示，作为主键建立索引查询效率低，常见优化方案为“转化为两个uint64整数存储”或者“折半存储”（折半后不能保证唯一性）

**【常见方法四：取当前毫秒数】**

uuid是一个本地算法，生成性能高，但无法保证趋势递增，且作为字符串ID检索效率低，有没有一种能保证递增的本地算法呢？

取当前毫秒数是一种常见方案：uint64 ID = GenTimeMS();

**优点**：

（1）本地生成ID，不需要进行远程调用，时延低

（2）生成的ID趋势递增

（3）生成的ID是整数，建立索引后查询效率高

**缺点**：

（1）如果并发量超过1000，会生成重复的ID

我去，这个缺点要了命了，不能保证ID的唯一性。当然，使用微秒可以降低冲突概率，但每秒最多只能生成1000000个ID，再多的话就一定会冲突了，所以使用微秒并不从根本上解决问题。

**【常见方法五：类snowflake算法】**

snowflake是twitter开源的分布式ID生成算法，其**核心思想**是：一个long型的ID，使用其中41bit作为毫秒数，10bit作为机器编号，12bit作为毫秒内序列号。这个算法单机每秒内理论上最多可以生成1000\*(2^12)，也就是400W的ID，完全能满足业务的需求。

借鉴snowflake的思想，结合各公司的业务逻辑和并发量，可以实现自己的分布式ID生成算法。

举例，假设某公司ID生成器服务的需求如下：

（1）单机高峰并发量小于1W，预计未来5年单机高峰并发量小于10W

（2）有2个机房，预计未来5年机房数量小于4个

（3）每个机房机器数小于100台

（4）目前有5个业务线有ID生成需求，预计未来业务线数量小于10个

（5）…

分析过程如下：

（1）高位取从2016年1月1日到现在的毫秒数（假设系统ID生成器服务在这个时间之后上线），假设系统至少运行10年，那至少需要10年\*365天\*24小时\*3600秒\*1000毫秒=320\*10^9，差不多预留39bit给毫秒数

（2）每秒的单机高峰并发量小于10W，即平均每毫秒的单机高峰并发量小于100，差不多预留7bit给每毫秒内序列号

（3）5年内机房数小于4个，预留2bit给机房标识

（4）每个机房小于100台机器，预留7bit给每个机房内的服务器标识

（5）业务线小于10个，预留4bit给业务线标识

http://mmbiz.qpic.cn/mmbiz/YrezxckhYOydMj2Lichnic5csTOdqI2a1mC7utPZR12icoHP8UzEibcVetkkZHxWg3ZNhuItMhiboT2AvBWWKvnDlzA/640?wx_fmt=png&wxfrom=5&wx_lazy=1

这样设计的64bit标识，可以保证：

（1）每个业务线、每个机房、每个机器生成的ID都是不同的

（2）同一个机器，每个毫秒内生成的ID都是不同的

（3）同一个机器，同一个毫秒内，以序列号区区分保证生成的ID是不同的

（4）将毫秒数放在最高位，保证生成的ID是趋势递增的

**缺点**：

（1）由于“没有一个全局时钟”，每台服务器分配的ID是绝对递增的，但从全局看，生成的ID只是趋势递增的（有些服务器的时间早，有些服务器的时间晚）

**最后一个容易忽略的问题**：

生成的ID，例如message-id/ order-id/ tiezi-id，在数据量大时往往需要分库分表，这些ID经常作为取模分库分表的依据，为了分库分表后数据均匀，ID生成往往有“取模随机性”的需求，所以我们通常把每秒内的序列号放在ID的最末位，保证生成的ID是随机的。

又如果，我们在跨毫秒时，序列号总是归0，会使得序列号为0的ID比较多，导致生成的ID取模后不均匀。解决方法是，序列号不是每次都归0，而是归一个0到9的随机数，这个地方。

## 互联网架构，如何进行容量设计？

原创 2016-08-23 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959542&idx=1&sn=2494bbea9a855e0e1c3ccd6d2562a600&scene=21" \l "#)

### ****一、需求缘起****

互联网公司，这样的场景是否似曾相识：

场景一：pm要做一个很大的运营活动，技术老大杀过来，问了两个问题：

（1）机器能抗住么？

（2）如果扛不住，需要加多少台机器？

场景二：系统设计阶段，技术老大杀过来，又问了两个问题：

（1）数据库需要分库么？

（2）如果需要分库，需要分几个库？

技术上来说，这些都是系统容量预估的问题，**容量设计是架构师必备的技能之一**。常见的容量评估包括数据量、并发量、带宽、CPU/MEM/DISK等，今天分享的内容，就以【并发量】为例，看看如何回答好这两个问题。

### ****二、容量评估的步骤与方法****

**【步骤一：评估总访问量】**

如何知道总访问量？对于一个运营活动的访问量评估，或者一个系统上线后PV的评估，有什么好的方法？

答案是：询问业务方，询问运营同学，询问产品同学，看对运营活动或者产品上线后的预期是什么。

举例：58要做一个APP-push的运营活动，计划在30分钟内完成5000w用户的push推送，预计push消息点击率10%，求push落地页系统的总访问量？

回答：5000w\*10% = 500w

**【步骤二：评估平均访问量QPS】**

如何知道平均访问量QPS？

答案是：有了总量，除以总时间即可，如果按照天评估，一天按照4w秒计算。

举例1：push落地页系统30分钟的总访问量是500w，求平均访问量QPS

回答：500w/(30\*60) = 2778，大概3000QPS

举例2：主站首页估计日均pv 8000w，求平均访问QPS

回答：一天按照4w秒算，8000w/4w=2000，大概2000QPS

提问：为什么一天按照4w秒计算？

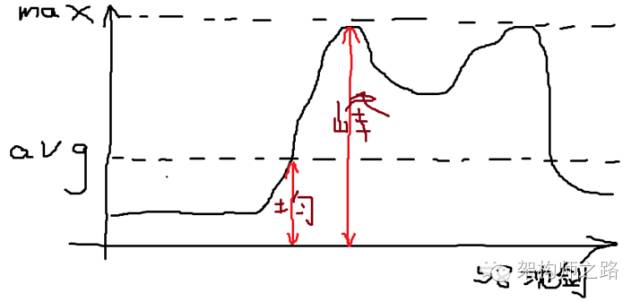
回答：一天共24小时\*60分钟\*60秒=8w秒，一般假设所有请求都发生在白天，所以一般来说一天只按照4w秒评估

**【步骤三：评估高峰QPS】**

系统容量规划时，不能只考虑平均QPS，而是要抗住高峰的QPS，如何知道高峰QPS呢？

答案是：根据业务特性，通过业务访问曲线评估

举例：日均QPS为2000，业务访问趋势图如下图，求峰值QPS预估？



回答：从图中可以看出，峰值QPS大概是均值QPS的2.5倍，日均QPS为2000，于是评估出峰值QPS为5000。

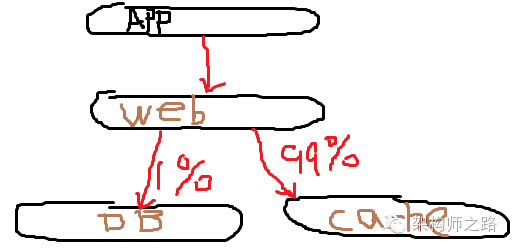
说明：有一些业务例如“秒杀业务”比较难画出业务访问趋势图，这类业务的容量评估不在此列。

**【步骤四：评估系统、单机极限QPS】**

如何评估一个业务，一个服务单机能的极限QPS呢？

答案是：压力测试

在一个服务上线前，一般来说是需要进行压力测试的（很多创业型公司，业务迭代很快的系统可能没有这一步，那就悲剧了），以APP-push运营活动落地页为例（日均QPS2000，峰值QPS5000），这个系统的架构可能是这样的：



1）访问端是APP

2）运营活动H5落地页是一个web站点

3）H5落地页由缓存cache、数据库db中的数据拼装而成

通过压力测试发现，web层是瓶颈，tomcat压测单机只能抗住1200的QPS（一般来说，1%的流量到数据库，数据库500QPS还是能轻松抗住的，cache的话QPS能抗住，需要评估cache的带宽，假设不是瓶颈），我们就得到了web单机极限的QPS是1200。一般来说，线上系统是不会跑满到极限的，打个8折，单机线上允许跑到QPS1000。

**【步骤五：根据线上冗余度回答两个问题】**

好了，上述步骤1-4已经得到了峰值QPS是5000，单机QPS是1000，假设线上部署了2台服务，就能自信自如的回答技术老大提出的问题了：

（1）机器能抗住么？ -> 峰值5000，单机1000，线上2台，扛不住

（2）如果扛不住，需要加多少台机器？ -> 需要额外3台，提前预留1台更好，给4台更稳

除了并发量的容量预估，数据量、带宽、CPU/MEM/DISK等评估亦可遵循类似的步骤。

### ****三、总结****

互联网架构设计如何进行容量评估：

**【步骤一：评估总访问量】** -> 询问业务、产品、运营

**【步骤二：评估平均访问量QPS】**-> 除以时间，一天算4w秒

**【步骤三：评估高峰QPS】** -> 根据业务曲线图来

**【步骤四：评估系统、单机极限QPS】** -> 压测很重要

**【步骤五：根据线上冗余度回答两个问题】** -> 估计冗余度与线上冗余度差值

个人一些经验分享，大伙轻拍，有更好的建议欢迎回复，下篇文章会将好的经验share给更多的同学。

==【完】==

回【无锁】[如何实现超高并发的无锁缓存？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959502&idx=1&sn=7a1c7d81a8e030856fb920844dc571c6&scene=21" \l "wechat_redirect" \t "_blank)

回【机房】[从IDC到云端架构迁移之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959495&idx=1&sn=0f8789d6803f94e190ba2787389e6d1b&scene=21" \l "wechat_redirect" \t "_blank)

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

回【单点】[单点系统架构的可用性与性能优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959480&idx=1&sn=337bd74410a6bef616128fd17abd08a8&scene=21" \l "wechat_redirect" \t "_blank)

回【服务】[互联网架构为什么要做服务化？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959519&idx=1&sn=065074b135fc9cb243abe897261e1a72&scene=21" \l "wechat_redirect" \t "_blank)

【小游戏：回大于10的整数，随机返回好文，猜猜怎么实现的】

## 线程数究竟设多少合理

原创 2016-03-29 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "#)

### ****一、需求缘起****

Web-Server通常有个配置，**最大工作线程数**，后端服务一般也有个配置，工作线程池的**线程数量**，这个线程数的配置不同的业务架构师有不同的经验值，有些业务设置为CPU核数的2倍，有些业务设置为CPU核数的8倍，有些业务设置为CPU核数的32倍。

“工作线程数”的设置依据是什么，到底设置为多少能够最大化CPU性能，是本文要讨论的问题。

### ****二、一些共性认知****

在进行进一步深入讨论之前，先以提问的方式就一些共性认知达成一致。

**提问：工作线程数是不是设置的越大越好？**

回答：肯定不是的

1）一来服务器CPU核数有限，同时并发的线程数是有限的，1核CPU设置10000个工作线程没有意义

2）线程切换是有开销的，如果线程切换过于频繁，反而会使性能降低

**提问：调用sleep()函数的时候，线程是否一直占用CPU？**

回答：不占用，等待时会把CPU让出来，给其他需要CPU资源的线程使用

不止调用sleep()函数，在进行一些阻塞调用，例如网络编程中的**阻塞**accept()【等待客户端连接】和**阻塞**recv()【等待下游回包】也不占用CPU资源

**提问：如果CPU是单核，设置多线程有意义么，能提高并发性能么？**

回答：即使是单核，使用多线程也是有意义的

1）多线程编码可以让我们的服务/代码更加清晰，有些IO线程收发包，有些Worker线程进行任务处理，有些Timeout线程进行超时检测

2）如果有一个任务一直占用CPU资源在进行计算，那么此时增加线程并不能增加并发，例如这样的一个代码

 while(1){ i++; }

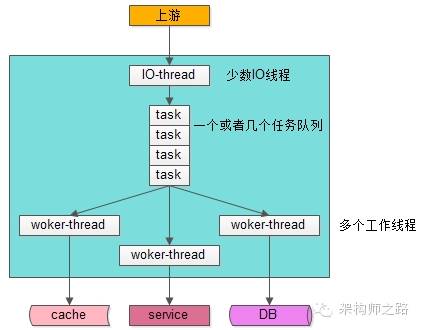
该代码一直不停的占用CPU资源进行计算，会使CPU占用率达到100%

3）通常来说，Worker线程一般不会一直占用CPU进行计算，此时即使CPU是单核，增加Worker线程也能够提高并发，因为这个线程在休息的时候，其他的线程可以继续工作

### ****三、常见服务线程模型****

了解常见的服务线程模型，有助于理解服务并发的原理，一般来说互联网常见的服务线程模型有如下两种

**IO线程与工作线程通过队列解耦类模型**



如上图，大部分Web-Server与服务框架都是使用这样的一种“IO线程与Worker线程通过队列解耦”类线程模型：

1）**有少数几个IO线程**监听上游发过来的请求，并进行收发包（生产者）

2）**有一个或者多个任务队列**，作为IO线程与Worker线程异步解耦的数据传输通道（临界资源）

3）**有多个工作线程**执行正真的任务（消费者）

这个线程模型应用很广，符合大部分场景，这个线程模型的特点是，**工作线程内部是同步阻塞执行任务的**（回想一下tomcat线程中是怎么执行Java程序的，dubbo工作线程中是怎么执行任务的），因此可以通过增加Worker线程数来增加并发能力，今天要讨论的重点是“该模型Worker线程数设置为多少能达到最大的并发”。

**纯异步线程模型**

任何地方都没有阻塞，这种线程模型只需要设置很少的线程数就能够做到很高的吞吐量，Lighttpd有一种单进程单线程模式，并发处理能力很强，就是使用的的这种模型。该模型的缺点是：

1）如果使用单线程模式，难以利用多CPU多核的优势

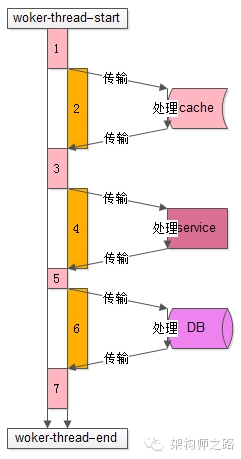
2）程序员更习惯写同步代码，callback的方式对代码的可读性有冲击，对程序员的要求也更高

3）框架更复杂，往往需要server端收发组件，server端队列，client端收发组件，client端队列，上下文管理组件，有限状态机组件，超时管理组件的支持

however，这个模型不是今天讨论的重点。

### ****四、工作线程的工作模式****

了解工作线程的工作模式，对量化分析线程数的设置非常有帮助：



上图是一个典型的工作线程的处理过程，从开始处理start到结束处理end，该任务的处理共有7个步骤：

1）从工作队列里拿出任务，进行一些**本地**初始化计算，例如http协议分析、参数解析、参数校验等

2）**访问cache**拿一些数据

3）拿到cache里的数据后，再进行一些**本地**计算，这些计算和业务逻辑相关

4）通过**RPC调用下游service**再拿一些数据，或者让下游service去处理一些相关的任务

5）RPC调用结束后，再进行一些**本地**计算，怎么计算和业务逻辑相关

6）**访问DB**进行一些数据操作

7）操作完数据库之后做一些收尾工作，同样这些收尾工作也是**本地**计算，和业务逻辑相关

分析整个处理的时间轴，会发现：

1）其中1，3，5，7步骤中【上图中粉色时间轴】，线程进行本地业务逻辑计算时**需要占用CPU**

2）而2，4，6步骤中【上图中橙色时间轴】，访问cache、service、DB过程中线程处于一个等待结果的状态，**不需要占用CPU**，进一步的分解，这个“等待结果”的时间共分为三部分：

2.1）请求在网络上传输到下游的cache、service、DB

2.2）下游cache、service、DB进行任务处理

2.3）cache、service、DB将报文在网络上传回工作线程

### ****五、量化分析并合理设置工作线程数****

最后一起来回答工作线程数设置为多少合理的问题。

通过上面的分析，Worker线程在执行的过程中，有一部计算时间需要占用CPU，另一部分等待时间不需要占用CPU，通过量化分析，例如打日志进行统计，可以统计出整个Worker线程执行过程中这两部分时间的比例，例如：

1）时间轴1，3，5，7【上图中粉色时间轴】的计算执行时间是100ms

2）时间轴2，4，6【上图中橙色时间轴】的等待时间也是100ms

得到的结果是，这个线程计算和等待的时间是1：1，即有50%的时间在计算（占用CPU），50%的时间在等待（不占用CPU）：

1）假设此时是**单核**，则设置为2个工作线程就可以把CPU充分利用起来，让CPU跑到100%

2）假设此时是**N核**，则设置为2N个工作现场就可以把CPU充分利用起来，让CPU跑到N\*100%

结论：

N核服务器，通过执行业务的单线程分析出本地计算时间为x，等待时间为y，则工作线程数（线程池线程数）设置为 N\*(x+y)/x，能让CPU的利用率最大化。

经验：

一般来说，非CPU密集型的业务（加解密、压缩解压缩、搜索排序等业务是CPU密集型的业务），瓶颈都在后端数据库，本地CPU计算的时间很少，所以设置几十或者几百个工作线程也都是可能的。

### ****六、结论****

N核服务器，通过执行业务的单线程分析出本地计算时间为x，等待时间为y，则工作线程数（线程池线程数）设置为 N\*(x+y)/x，能让CPU的利用率最大化。

==【完】==

回【冗余】[冗余表如何保证数据一致](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403963671&idx=1&sn=51a2d2fd70212451cd5f22bbe2c6f8d6&scene=21" \l "wechat_redirect" \t "_blank)

回【缓存】[缓存与数据库架构细节二三事](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404087915&idx=1&sn=075664193f334874a3fc87fd4f712ebc&scene=21" \l "wechat_redirect" \t "_blank)

回【一致】[缓存与数据库一致性](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404202261&idx=1&sn=1b8254ba5013952923bdc21e0579108e&scene=21" \l "wechat_redirect" \t "_blank)

回【淘汰】[双淘汰法保证DB与缓存一致性](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404308725&idx=1&sn=1a25ce76dd1956014ceb8a011855268e&scene=21" \l "wechat_redirect" \t "_blank)

【小游戏：回大于10的整数，随机返回好文，猜猜怎么实现的】

## 单点系统架构的可用性与性能优化

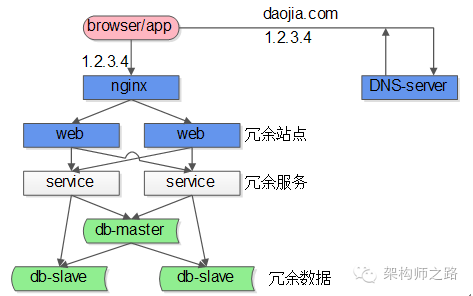
原创 2016-07-08 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959480&idx=1&sn=337bd74410a6bef616128fd17abd08a8&scene=21" \l "#)

### ****一、需求缘起****

明明架构要求高可用，为何系统中还会存在单点？

回答：单点master的设计，会**大大简化系统设计，何况有时候避免不了单点**

在哪些场景中会存在单点？先来看一下一个典型互联网高可用架构。



典型互联网高可用架构：

（1）**客户端层**，这一层是浏览器或者APP，第一步先访问DNS-server，由域名拿到nginx的外网IP

（2）**负载均衡层**，nginx是整个服务端的入口，负责反向代理与负载均衡工作

（3）**站点层**，web-server层，典型的是tomcat或者apache

（4）**服务层**，service层，典型的是dubbo或者thrift等提供RPC调用的后端服务

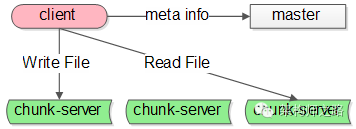
（5）**数据层**，包含cache和db，典型的是主从复制读写分离的db架构

在这个互联网架构中，站点层、服务层、数据库的从库都可以通过冗余的方式来保证高可用，但至少

（1）nginx层是一个潜在的单点

（2）数据库写库master也是一个潜在的单点

再举一个GFS（Google File System）架构的例子。



GFS的系统架构里主要有这么几种角色：

（1）**client**，就是发起文件读写的调用端

（2）**master**，这是一个单点服务，它有全局事业，掌握文件元信息

（3）**chunk-server**，实际存储文件额服务器

这个系统里，master也是一个单点的服务，Map-reduce系统里也有类似的全局协调的master单点角色。

系统架构设计中，像nginx，db-master，gfs-master这样的单点服务，会存在什么问题，有什么方案来优化呢，这是本文要讨论的问题。

### ****二、单点架构存在的问题****

单点系统一般来说存在两个很大的问题：

（1）**非高可用**：既然是单点，master一旦发生故障，服务就会受到影响

（2）**性能瓶颈**：既然是单点，不具备良好的扩展性，服务性能总有一个上限，这个单点的性能上限往往就是整个系统的性能上限

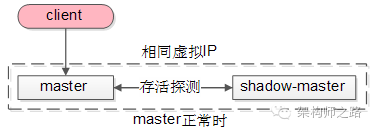
接下来，就看看有什么优化手段可以优化上面提到的两个问题

### ****三、shadow-master解决单点高可用问题****

shadow-master是一种很常见的解决单点高可用问题的技术方案。

**“影子master”，顾名思义，服务正常时，它只是单点master的一个影子，在master出现故障时，shadow-master会自动变成master，继续提供服务。**

shadow-master它能够解决高可用的问题，并且故障的转移是自动的，不需要人工介入，但不足是它使服务资源的利用率降为了50%，**业内经常使用keepalived+vip的方式实现这类单点的高可用**。

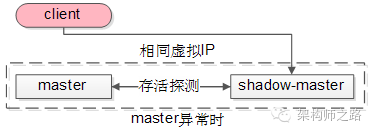


以GFS的master为例，master正常时：

（1）client会连接正常的master，shadow-master不对外提供服务

（2）master与shadow-master之间有一种存活探测机制

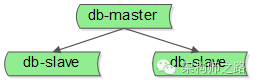
（3）master与shadow-master有相同的虚IP（virtual-IP）



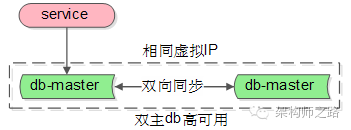
当发现master异常时：

shadow-master会自动顶上成为master，虚IP机制可以保证这个过程对调用方是透明的

除了GFS与MapReduce系统中的主控master，nginx亦可用类似的方式保证高可用，数据库的主库master（主库）亦可用类似的方式来保证高可用，只是细节上有些地方要注意：



传统的一主多从，读写分离的db架构，只能保证读库的高可用，是无法保证写库的高可用的，要想保证写库的高可用，也可以使用上述的shadow-master机制：



（1）两个主库设置**相互同步的双主模式**

（2）平时只有一个主库提供服务，言下之意，shadow-master不会往master同步数据

（3）异常时，虚IP漂移到另一个主库，shadow-master变成主库继续提供服务

需要说明的是，由于数据库的特殊性，数据同步需要时延，如果数据还没有同步完成，流量就切到了shadow-master，可能引起小部分数据的不一致。

### ****四、减少与单点的交互，是存在单点的系统优化的核心方向****

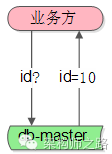
既然知道单点存在性能上限，单点的性能（例如GFS中的master）有可能成为系统的瓶颈，那么，减少与单点的交互，便成了存在单点的系统优化的核心方向。

怎么来减少与单点的交互，这里提两种常见的方法。

**批量写**

批量写是一种常见的提升单点性能的方式。

例如一个利用数据库写单点生成做“ID生成器”的例子：



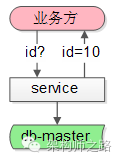
（1）业务方需要ID

（2）利用数据库写单点的auto increament id来生成和返回ID

这是一个很常见的例子，很多公司也就是这么生成ID的，它利用了数据库写单点的特性，方便快捷，无额外开发成本，是一个非常帅气的方案。

潜在的问题是：生成ID的并发上限，取决于单点数据库的写性能上限。

如何提升性能呢？批量写



（1）中间加一个服务，每次从数据库拿出100个id

（2）业务方需要ID

（3）服务直接返回100个id中的1个，100个分配完，再访问数据库

这样一来，每分配100个才会写数据库一次，分配id的性能可以认为提升了100倍。

**客户端缓存**

客户端缓存也是一种降低与单点交互次数，提升系统整体性能的方法。

还是以GFS文件系统为例：



（1）GFS的调用客户端client要访问shenjian.txt，先查询本地缓存，miss了

（2）client访问master问说文件在哪里，master告诉client在chunk3上

（3）client把shenjian.txt存放在chunk3上记录到本地的缓存，然后进行文件的读写操作

（4）未来client要访问文件，从本地缓存中查找到对应的记录，就不用再请求master了，可以直接访问chunk-server。如果文件发生了转移，chunk3返回client说“文件不在我这儿了”，client再访问master，询问文件所在的服务器。

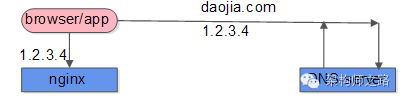
根据经验，这类缓存的命中非常非常高，可能在99.9%以上（因为文件的自动迁移是小概率事件），这样与master的交互次数就降低了1000倍。

### ****五、水平扩展是提升单点系统性能的好方案****

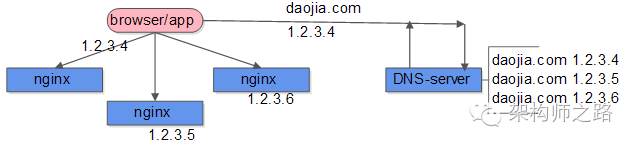
无论怎么批量写，客户端缓存，单点毕竟是单机，还是有性能上限的。

想方设法水平扩展，消除系统单点，理论上才能够无限的提升系统系统。

以nginx为例，如何来进行水平扩展呢？



第一步的DNS解析，只能返回一个nginx外网IP么？答案显然是否定的，**“DNS轮询”技术**支持DNS-server返回不同的nginx外网IP，这样就能实现nginx负载均衡层的水平扩展。



DNS-server部分，一个域名可以配置多个IP，每次DNS解析请求，轮询返回不同的IP，就能实现nginx的水平扩展，扩充负载均衡层的整体性能。

数据库单点写库也是同样的道理，在数据量很大的情况下，可以通过水平拆分，来提升写入性能。

遗憾的是，**并不是所有的业务场景都可以水平拆分**，例如秒杀业务，商品的条数可能不多，数据库的数据量不大，就不能通过水平拆分来提升秒杀系统的整体写性能（总不能一个库100条记录吧？）。

### ****六、总结****

今天的话题就讨论到这里，内容很多，占用大家宝贵的时间深表内疚，估计大部分都记不住，至少记住这几个点吧：

（1）**单点系统存在的问题**：可用性问题，性能瓶颈问题

（2）**shadow-master**是一种常见的解决单点系统可用性问题的方案

（3）**减少与单点的交互**，是存在单点的系统优化的核心方向，常见方法有批量写，客户端缓存

（4）**水平扩展**也是提升单点系统性能的好方案

如果有收获，帮忙随手转发哟。

==【完】==

回【检索】[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)

回【钱包】[入驻微信钱包的技术优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959423&idx=1&sn=19d27145beab5b0238280e2e3804e41b&scene=21" \l "wechat_redirect" \t "_blank)

回【消息】[58到家通用实时消息平台架构细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "wechat_redirect" \t "_blank)

回【秒杀】[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959391&idx=1&sn=fb28fd5e5f0895ddb167406d8a735548&scene=21" \l "wechat_redirect" \t "_blank)

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

回【延时】[mysql并行复制降低主从同步延时的思路与启示](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959461&idx=1&sn=c4c9317f55075541be00d59039b88470&scene=21" \l "wechat_redirect" \t "_blank)

回【推荐】[58同城推荐系统架构设计与实现](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959464&idx=1&sn=81c29bf2e237499be07deba826f73c39&scene=21" \l "wechat_redirect" \t "_blank)

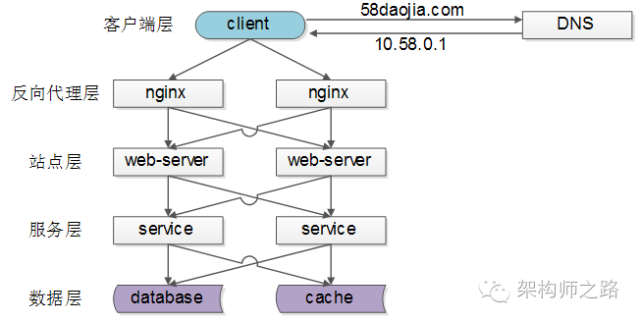
## 一分钟了解负载均衡的一切

原创 2016-09-13 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959585&idx=1&sn=0a9222cbfeb62a662edffafb7f0b43ae&scene=21" \l "#)

**什么是负载均衡**

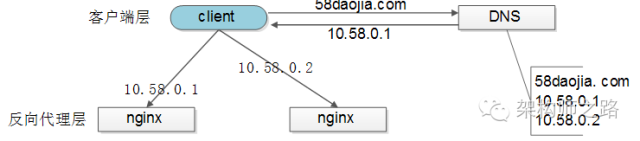
负载均衡（Load Balance）是分布式系统架构设计中必须考虑的因素之一，它通常是指，将请求/数据【均匀】分摊到多个操作单元上执行，负载均衡的关键在于【均匀】。

**常见的负载均衡方案**



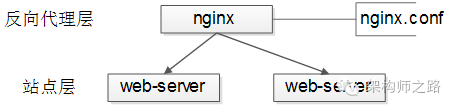
常见互联网分布式架构如上，分为**客户端层、反向代理nginx层、站点层、服务层、数据层**。可以看到，每一个下游都有多个上游调用，只需要做到，每一个上游都均匀访问每一个下游，就能实现“将请求/数据【均匀】分摊到多个操作单元上执行”。

**【客户端层->反向代理层】的负载均衡**



【客户端层】到【反向代理层】的负载均衡，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，保证每个ip的解析概率是相同的。这些ip就是nginx的外网ip，以做到每台nginx的请求分配也是均衡的。

**【反向代理层->站点层】的负载均衡**



【反向代理层】到【站点层】的负载均衡，是通过“nginx”实现的。通过修改nginx.conf，可以实现多种负载均衡策略：

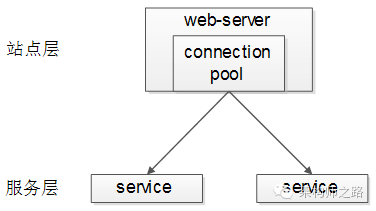
1）**请求轮询**：和DNS轮询类似，请求依次路由到各个web-server

2）**最少连接路由**：哪个web-server的连接少，路由到哪个web-server

3）**ip哈希**：按照访问用户的ip哈希值来路由web-server，只要用户的ip分布是均匀的，请求理论上也是均匀的，ip哈希均衡方法可以做到，同一个用户的请求固定落到同一台web-server上，此策略适合有状态服务，例如session（58沈剑备注：可以这么做，但强烈不建议这么做，站点层无状态是分布式架构设计的基本原则之一，session最好放到数据层存储）

4）…

**【站点层->服务层】的负载均衡**



【站点层】到【服务层】的负载均衡，是通过“服务连接池”实现的。

上游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。

上一篇文章[《RPC-client实现细节》](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959576&idx=1&sn=2be8d3f61effe7118abf920a175da710&scene=21" \l "wechat_redirect" \t "_blank)中有详细的负载均衡、故障转移、超时处理的细节描述，欢迎点击link查阅，此处不再展开。

**【数据层】的负载均衡**

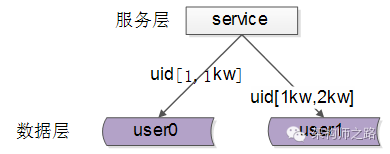
在数据量很大的情况下，由于数据层（db，cache）涉及数据的水平切分，所以数据层的负载均衡更为复杂一些，它分为“数据的均衡”，与“请求的均衡”。

**数据的均衡**是指：水平切分后的每个服务（db，cache），数据量是差不多的。

**请求的均衡**是指：水平切分后的每个服务（db，cache），请求量是差不多的。

业内常见的水平切分方式有这么几种：

一、按照range水平切分



每一个数据服务，存储一定范围的数据，上图为例：

user0服务，存储uid范围1-1kw

user1服务，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

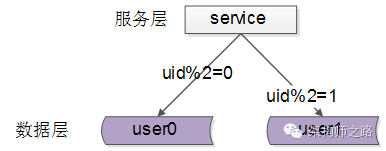
（2）**数据均衡性较好**

（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

二、按照id哈希水平切分



每一个数据服务，存储某个key值hash后的部分数据，上图为例：

user0服务，存储偶数uid数据

user1服务，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

**总结**

负载均衡（Load Balance）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，将请求/数据【均匀】分摊到多个操作单元上执行，负载均衡的关键在于【均匀】。

（1）【客户端层】到【反向代理层】的负载均衡，是通过“DNS轮询”实现的

（2）【反向代理层】到【站点层】的负载均衡，是通过“nginx”实现的

（3）【站点层】到【服务层】的负载均衡，是通过“服务连接池”实现的

（4）【数据层】的负载均衡，要考虑“数据的均衡”与“请求的均衡”两个点，常见的方式有“按照范围水平切分”与“hash水平切分”

谢谢大伙，帮忙转发，欢迎讨论，有问必答。

==【完】==

回【sed】[一分钟sed入门（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403340380&idx=1&sn=9b4ced5a19f60b0cebf6fd582f1d52ae&scene=21" \l "wechat_redirect" \t "_blank)

回【广告】[一分钟读懂互联网广告竞价策略（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403326874&idx=1&sn=ca03344326f35b2971b3ecae2026e8df&scene=21" \l "wechat_redirect" \t "_blank)

回【阵列】[一张“神图”看懂单机/集群/热备/磁盘阵列（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400720056&idx=1&sn=10c796793e27d475603f97b23e137c12&scene=21" \l "wechat_redirect" \t "_blank)

回【awk】[一分钟学awk够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400743254&idx=1&sn=147abc381ccd61e28f52699735c8748e&scene=21" \l "wechat_redirect" \t "_blank)

回【perl】[十分钟学perl够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400764946&idx=1&sn=4efab1953696c2e21071e58a74da9b82&scene=21" \l "wechat_redirect" \t "_blank)

回【mongo】[一分钟了解mongodb（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=201250814&idx=1&sn=fd6d85b727e1229e3324850217748d0f&scene=21" \l "wechat_redirect" \t "_blank)

回【2pc】[一分钟了解两阶段提交2PC（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400856212&idx=1&sn=b9734da77f2259bdd1fa89700b47eaff&scene=21" \l "wechat_redirect" \t "_blank)

回【join】[30秒懂SQL中的join（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400889133&idx=1&sn=3ea1c06fc76e92349a822f41f45b77b5&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

## 如何实施异构服务器的负载均衡及过载保护？

原创 2016-09-20 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959601&idx=1&sn=5684c39676b1f6d9366d9d15a2cdcec3&scene=21" \l "#)

**零、需求缘起**

第一篇文章“[一分钟了解负载均衡](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959585&idx=1&sn=0a9222cbfeb62a662edffafb7f0b43ae&scene=21" \l "wechat_redirect" \t "_blank)”和大家share了互联网架构中反向代理层、站点层、服务层、数据层的常用负载均衡方法。

第二篇文章“[lvs为何不能完全代替DNS轮询](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959595&idx=1&sn=5f0633afd24c547b895f29f6538baa99&scene=21" \l "wechat_redirect" \t "_blank)”和大家share了互联网接入层负载均衡需要解决的问题及架构演进。

在这两篇文章中，都强调了“负载均衡是指，将请求/数据【均匀】分摊到多个操作单元上执行，**负载均衡的关键在于【均匀】**”。

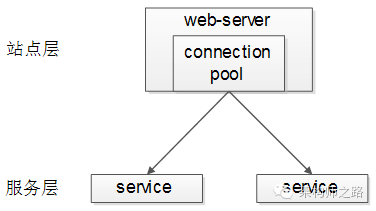
然而，后端的**service有可能部署在硬件条件不同的服务器上**：

1）如果对标最低配的服务器“均匀”分摊负载，高配的服务器的利用率不足；

2）如果对标最高配的服务器“均匀”分摊负载，低配的服务器可能会扛不住；

能否根据异构服务器的处理能力来动态、自适应进行负载均衡及过载保护，是本文要讨论的问题。

**一、service层的负载均衡通常是怎么做的**

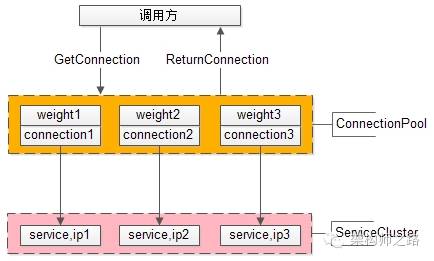


“[一分钟了解负载均衡](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959585&idx=1&sn=0a9222cbfeb62a662edffafb7f0b43ae&scene=21" \l "wechat_redirect" \t "_blank)”中提到，service层的负载均衡，一般是通过service连接池来实现的，调用方连接池会建立与下游服务多个连接，每次请求“随机”获取连接，来保证service访问的均衡性。

“[RPC-client实现细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959576&idx=1&sn=2be8d3f61effe7118abf920a175da710&scene=21" \l "wechat_redirect" \t "_blank)”中提到，负载均衡、故障转移、超时处理等细节也都是通过调用方连接池来实现的。

这个调用方连接池能否实现，根据service的处理能力，动态+自适应的进行负载调度呢？

**二、通过“静态权重”标识service的处理能力**



调用方通过连接池组件访问下游service，通常采用“随机”的方式返回连接，以保证下游service访问的均衡性。

要打破这个随机性，最容易想到的方法，只要为每个下游service设置一个“权重”，代表service的处理能力，来调整访问到每个service的概率，例如：

假设service-ip1，service-ip2，service-ip3的处理能力相同，可以设置weight1=1，weight2=1，weight3=1，这样三个service连接被获取到的概率分别就是1/3，1/3，1/3，能够保证均衡访问。

假设service-ip1的处理能力是service-ip2，service-ip3的处理能力的2倍，可以设置weight1=2，weight2=1，weight3=1，这样三个service连接被获取到的概率分别就是2/4，1/4，1/4，能够保证处理能力强的service分别到等比的流量，不至于资源浪费。

使用nginx做反向代理与负载均衡，就有类似的机制。

这个方案的**优点**是：简单，能够快速的实现异构服务器的负载均衡。

**缺点**也很明显：这个权重是固定的，无法自适应动态调整，而很多时候，服务器的处理能力是很难用一个固定的数值量化。

**三、通过“动态权重”标识service的处理能力**

提问：通过什么来标识一个service的处理能力呢？

回答：其实一个service能不能处理得过来，能不能响应得过来，应该由调用方说了算。调用服务，快速处理了，处理能力跟得上；调用服务，处理超时了，处理能力很有可能跟不上了。

**动态权重设计**

1）用一个动态权重来标识每个service的处理能力，默认初始处理能力相同，即分配给每个service的概率相等；

2）每当service成功处理一个请求，认为service处理能力足够，权重动态+1

3）每当service超时处理一个请求，认为service处理能力可能要跟不上了，权重动态-10（权重下降会更快）

4）为了方便权重的处理，可以把权重的范围限定为[0, 100]，把权重的初始值设为60分

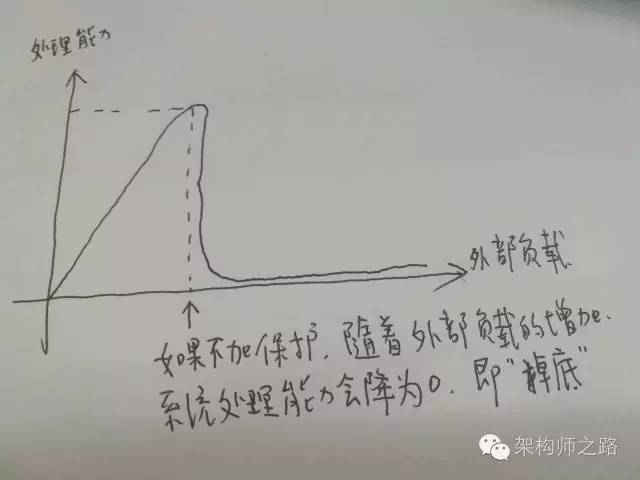
举例说明：

假设service-ip1，service-ip2，service-ip3的动态权重初始值weight1=weight2=weight3=60，刚开始时，请求分配给这3台service的概率分别是60/180，60/180，60/180，即负载是均衡的。

随着时间的推移，处理能力强的service成功处理的请求越来越多，处理能力弱的service偶尔有超时，随着动态权重的增减，权重可能变化成了weight1=100，weight2=60，weight3=40，那么此时，请求分配给这3台service的概率分别是100/200，60/200，40/200，即处理能力强的service会被分配到更多的流量。

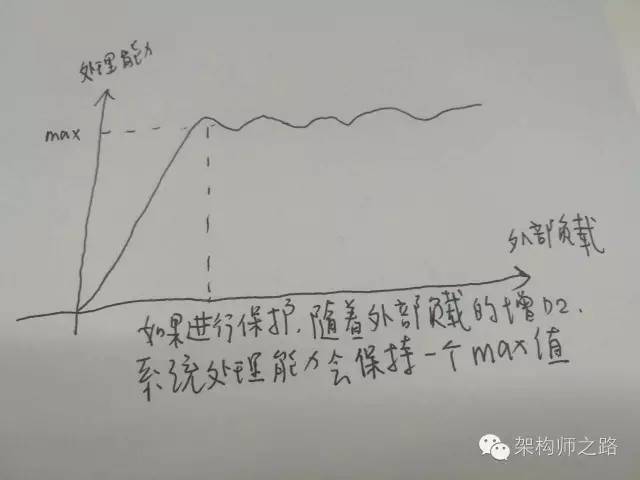
**四、过载保护**

提问：什么是过载保护？



图示：**无**过载保护的负载与处理能力图（会掉底）

回答：互联网软件架构设计中所指的过载保护，是指当系统负载超过一个service的处理能力时，如果service不进行自我保护，可能导致对外呈现处理能力为0，且不能自动恢复的现象。而service的过载保护，是指即使系统负载超过一个service的处理能力，service让能保证对外提供有损的稳定服务。



图示：**有**过载保护的负载与处理能力图（不会掉底）

提问：如何进行过载保护？

回答：最简易的方式，**服务端**设定一个负载阈值，超过这个阈值的请求压过来，全部抛弃。这个方式不是特别优雅。

**五、如何借助“动态权重”来实施过载保护**

动态权重是用来标识每个service的处理能力的一个值，它是RPC-client**客户端**连接池层面的一个东东。服务端处理超时，客户端RPC-client连接池都能够知道，这里只要实施一些策略，就能够对“疑似过载”的服务器进行降压，而不用服务器“抛弃请求”这么粗暴的实施过载保护。

应该实施一些什么样的策略呢，例如：

1）如果某一个service的连接上，连续3个请求都超时，即连续-10分三次，客户端就可以认为，服务器慢慢的要处理不过来了，得给这个service缓一小口气，于是设定策略：接下来的若干时间内，例如1秒（或者接下来的若干个请求），请求不再分配给这个service；

2）如果某一个service的动态权重，降为了0（像连续10个请求超时，中间休息了3次还超时），客户端就可以认为，服务器完全处理不过来了，得给这个service喘一大口气，于是设定策略：接下来的若干时间内，例如1分钟（为什么是1分钟，根据经验，此时service一般在发生fullGC，差不多1分钟能回过神来），请求不再分配给这个service；

3）可以有更复杂的保护策略…

这样的话，不但能借助“动态权重”来实施动态自适应的异构服务器负载均衡，还能在客户端层面更优雅的实施过载保护，在某个下游service快要响应不过来的时候，给其喘息的机会。

需要注意的是：要防止客户端的过载保护引起service的雪崩，如果“整体负载”已经超过了“service集群”的处理能力，怎么转移请求也是处理不过来的，还得通过抛弃请求来实施自我保护。

**六、总结**

1）service的负载均衡、故障转移、超时处理通常是RPC-client连接池层面来实施的

2）异构服务器负载均衡，最简单的方式是静态权重法，缺点是无法自适应动态调整

3）动态权重法，可以动态的根据service的处理能力来分配负载，需要有连接池层面的微小改动

4）过载保护，是在负载过高时，service为了保护自己，保证一定处理能力的一种自救方法

5）动态权重法，还可以用做service的过载保护

谢谢大伙，帮忙转发。

==【完】==

回【sed】[一分钟sed入门（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403340380&idx=1&sn=9b4ced5a19f60b0cebf6fd582f1d52ae&scene=21" \l "wechat_redirect" \t "_blank)

回【广告】[一分钟读懂互联网广告竞价策略（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403326874&idx=1&sn=ca03344326f35b2971b3ecae2026e8df&scene=21" \l "wechat_redirect" \t "_blank)

回【阵列】[一张“神图”看懂单机/集群/热备/磁盘阵列（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400720056&idx=1&sn=10c796793e27d475603f97b23e137c12&scene=21" \l "wechat_redirect" \t "_blank)

回【awk】[一分钟学awk够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400743254&idx=1&sn=147abc381ccd61e28f52699735c8748e&scene=21" \l "wechat_redirect" \t "_blank)

回【perl】[十分钟学perl够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400764946&idx=1&sn=4efab1953696c2e21071e58a74da9b82&scene=21" \l "wechat_redirect" \t "_blank)

回【mongo】[一分钟了解mongodb（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=201250814&idx=1&sn=fd6d85b727e1229e3324850217748d0f&scene=21" \l "wechat_redirect" \t "_blank)

回【2pc】[一分钟了解两阶段提交2PC（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400856212&idx=1&sn=b9734da77f2259bdd1fa89700b47eaff&scene=21" \l "wechat_redirect" \t "_blank)

回【join】[30秒懂SQL中的join（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400889133&idx=1&sn=3ea1c06fc76e92349a822f41f45b77b5&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

## lvs为何不能完全替代DNS轮询

原创 2016-09-18 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959595&idx=1&sn=5f0633afd24c547b895f29f6538baa99&scene=21" \l "#)

上一篇文章“[一分钟了解负载均衡的一切](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959585&idx=1&sn=0a9222cbfeb62a662edffafb7f0b43ae&scene=21" \l "wechat_redirect" \t "_blank)”引起了不少同学的关注，评论中大家争论的比较多的一个技术点是**接入层负载均衡技术**，部分同学持这样的观点：

1）nginx前端加入lvs和keepalived可以替代“DNS轮询”

2）F5能搞定接入层高可用、扩展性、负载均衡，可以替代“DNS轮询”

“DNS轮询”究竟是不是过时的技术，是不是可以被其他方案替代，接入层架构技术演进，是本文将要细致讨论的内容。

**一、问题域**

nginx、lvs、keepalived、f5、DNS轮询，每每提到这些技术，往往讨论的是接入层的这样几个问题：

1）**可用性**：任何一台机器挂了，服务受不受影响

2）**扩展性**：能否通过增加机器，扩充系统的性能

3）**反向代理+负载均衡**：请求是否均匀分摊到后端的操作单元执行

**二、上面那些名词都是干嘛的**

由于每个技术人的背景和知识域不同，上面那些名词缩写（运维的同学再熟悉不过了），还是花1分钟简单说明一下（详细请自行“百度”）：

1）**nginx**：一个高性能的web-server和实施反向代理的软件

2）**lvs**：Linux Virtual Server，使用集群技术，实现在linux操作系统层面的一个高性能、高可用、负载均衡服务器

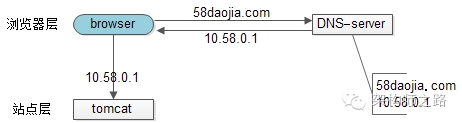
3）**keepalived**：一款用来检测服务状态存活性的软件，常用来做高可用

4）**f5**：一个高性能、高可用、负载均衡的硬件设备（听上去和lvs功能差不多？）

5）**DNS轮询**：通过在DNS-server上对一个域名设置多个ip解析，来扩充web-server性能及实施负载均衡的技术

**三、接入层技术演进**

**【裸奔时代（0）单机架构】**



裸奔时代的架构图如上：

1）浏览器通过DNS-server，域名解析到ip

2）浏览器通过ip访问web-server

**缺点**：

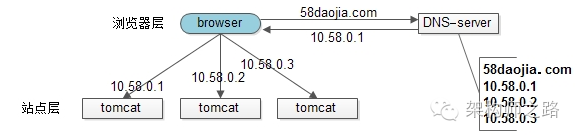
1）非高可用，web-server挂了整个系统就挂了

2）扩展性差，当吞吐量达到web-server上限时，无法扩容

注：单机不涉及负载均衡的问题

**【简易扩容方案（1）DNS轮询】**

假设tomcat的吞吐量是1000次每秒，当系统总吞吐量达到3000时，如何扩容是首先要解决的问题，DNS轮询是一个很容易想到的方案：



此时的架构图如上：

1）多部署几份web-server，1个tomcat抗1000，部署3个tomcat就能抗3000

2）在DNS-server层面，域名每次解析到不同的ip

**优点**：

1）零成本：在DNS-server上多配几个ip即可，功能也不收费

2）部署简单：多部署几个web-server即可，原系统架构不需要做任何改造

3）负载均衡：变成了多机，但负载基本是均衡的

**缺点**：

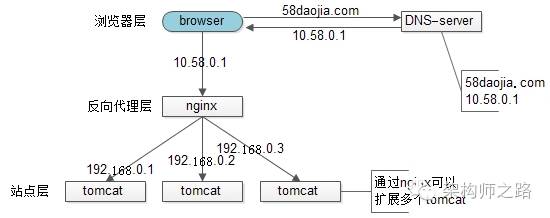
1）非高可用：**DNS-server只负责域名解析ip，这个ip对应的服务是否可用，DNS-server是不保证的**，假设有一个web-server挂了，部分服务会受到影响

2）扩容非实时：DNS解析有一个生效周期

3）暴露了太多的外网ip

**【简易扩容方案（2）nginx】**

tomcat的性能较差，但nginx作为反向代理的性能就强多了，假设线上跑到1w，就比tomcat高了10倍，可以利用这个特性来做扩容：



此时的架构图如上：

1）站点层与浏览器层之间加入了一个反向代理层，利用高性能的nginx来做反向代理

2）nginx将http请求分发给后端多个web-server

**优点**：

1）DNS-server不需要动

2）负载均衡：通过nginx来保证

3）只暴露一个外网ip，nginx->tomcat之间使用内网访问

4）扩容实时：nginx内部可控，随时增加web-server随时实时扩容

5）能够保证站点层的可用性：任何一台tomcat挂了，nginx可以将流量迁移到其他tomcat

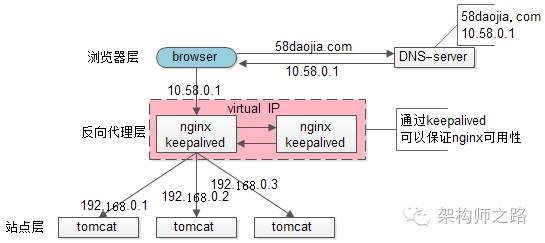
**缺点**：

1）时延增加+架构更复杂了：中间多加了一个反向代理层

2）反向代理层成了单点，非高可用：tomcat挂了不影响服务，**nginx挂了怎么办？**

**【高可用方案（3）keepalived】**

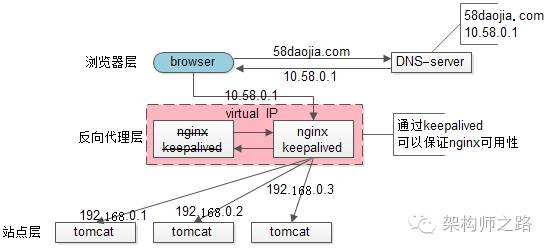
为了解决高可用的问题，keepalived出场了（之前的文章“[使用shadow-master保证系统可用性](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959480&idx=1&sn=337bd74410a6bef616128fd17abd08a8&scene=21" \l "wechat_redirect" \t "_blank)”详细介绍过）：



此时：

1）做两台nginx组成一个集群，分别部署上keepalived，设置成相同的虚IP，保证nginx的高可用

2）当一台nginx挂了，keepalived能够探测到，并将流量自动迁移到另一台nginx上，整个过程对调用方透明



**优点**：

1）解决了高可用的问题

**缺点**：

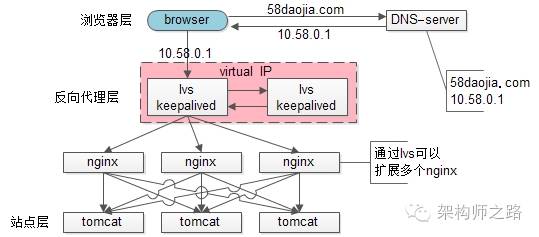
1）资源利用率只有50%

2）nginx仍然是接入单点，**如果接入吞吐量超过的nginx的性能上限怎么办，例如qps达到了50000咧？**

**【scale up扩容方案（4）lvs/f5】**

nginx毕竟是软件，性能比tomcat好，但总有个上限，超出了上限，还是扛不住。

lvs就不一样了，它实施在操作系统层面；f5的性能又更好了，它实施在硬件层面；它们性能比nginx好很多，例如每秒可以抗10w，这样可以利用他们来扩容，常见的架构图如下：



此时：

1）如果通过nginx可以扩展多个tomcat一样，可以通过lvs来扩展多个nginx

2）通过keepalived+VIP的方案可以保证可用性

**99.9999%的公司到这一步基本就能解决接入层高可用、扩展性、负载均衡的问题。**

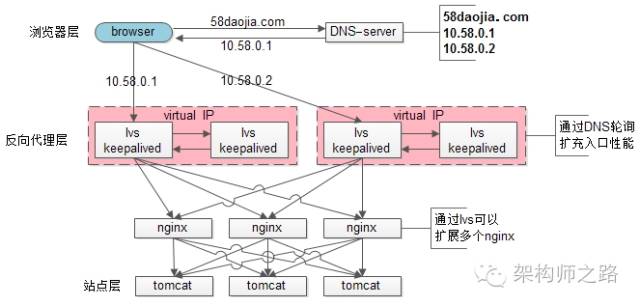
这就完美了嘛？还有潜在问题么？

好吧，不管是使用lvs还是f5，这些都是scale up的方案，根本上，lvs/f5还是会有性能上限，假设每秒能处理10w的请求，一天也只能处理80亿的请求（10w秒吞吐量\*8w秒），那万一**系统的日PV超过80亿怎么办呢？**（好吧，没几个公司要考虑这个问题）

**【scale out扩容方案（5）DNS轮询】**

如之前文章所述，水平扩展，才是解决性能问题的根本方案，能够通过加机器扩充性能的方案才具备最好的扩展性。

facebook，google，baidu的PV是不是超过80亿呢，它们的域名只对应一个ip么，**终点又是起点，还是得通过DNS轮询来进行扩容**：



此时：

1）通过DNS轮询来线性扩展入口lvs层的性能

2）通过keepalived来保证高可用

3）通过lvs来扩展多个nginx

4）通过nginx来做负载均衡，业务七层路由

**四、结论**

聊了这么多，稍微做一个简要的总结：

1）**接入层架构要考虑的问题域为**：高可用、扩展性、反向代理+扩展均衡

2）nginx、keepalived、lvs、f5可以很好的解决**高可用、扩展性、反向代理+扩展均衡**的问题

3）水平扩展scale out是解决扩展性问题的根本方案，**DNS轮询是不能完全被nginx/lvs/f5所替代的**

末了，上一篇文章有同学留言问**58到家采用什么方案**，58到家目前部署在阿里云上，前端购买了SLB服务（可以先粗暴的认为是一个lvs+keepalived的高可用负载均衡服务），后端是nginx+tomcat。

**五、挖坑**

接入层讲了这么多，下一章，准备讲讲服务层“异构服务的负载均”（牛逼的机器应该分配更多的流量，如何做到？）。

希望大家有收获，转发一篇文章只需要3秒钟，求3秒。

==【完】==

回【均衡】[一分钟了解负载均衡的一切](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959585&idx=1&sn=0a9222cbfeb62a662edffafb7f0b43ae&scene=21" \l "wechat_redirect" \t "_blank)

回【容量】[互联网架构，如何进行容量设计？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959542&idx=1&sn=2494bbea9a855e0e1c3ccd6d2562a600&scene=21" \l "wechat_redirect" \t "_blank)

回【单点】[单点系统架构的可用性与性能优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959480&idx=1&sn=337bd74410a6bef616128fd17abd08a8&scene=21" \l "wechat_redirect" \t "_blank)

回【ID】[细聊分布式ID生成方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "wechat_redirect" \t "_blank)

## 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；

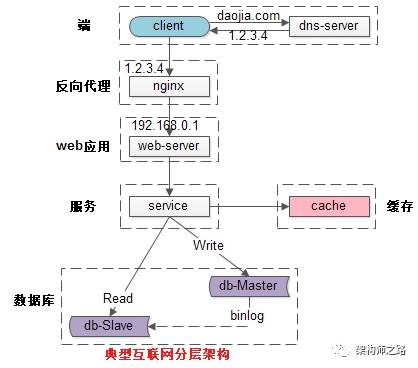
（2）提升单机架构性能，例如：使用Cache来减少IO次数，使用异步来增加单服务吞吐量，使用无锁数据结构来减少响应时间；

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**

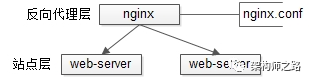
**反向代理层的水平扩展**



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。

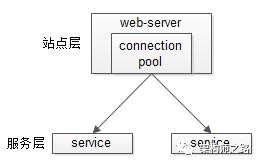
**站点层的水平扩展**



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。

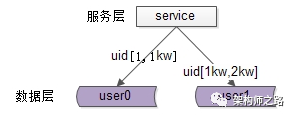
站点层通过RPC-client调用下游的服务层RPC-server时，RPC-client中的连接池会建立与下游服务多个连接，当服务成为瓶颈的时候，只要增加服务器数量，新增服务部署，在RPC-client处建立新的下游服务连接，就能扩展服务层性能，做到理论上的无限高并发。如果需要优雅的进行服务层自动扩容，这里可能需要配置中心里服务自动发现功能的支持。

**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；

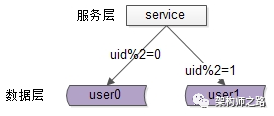
（2）数据均衡性较好；

（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)

## 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**

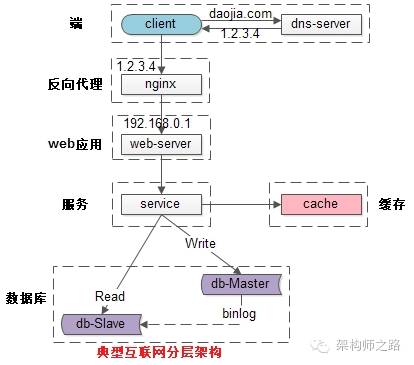
我们都知道，单点是系统高可用的大敌，单点往往是系统高可用最大的风险和敌人，应该尽量在系统设计的过程中避免单点。方法论上，高可用保证的原则是“集群化”，或者叫“冗余”：只有一个单点，挂了服务会受影响；如果有冗余备份，挂了还有其他backup能够顶上。

**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

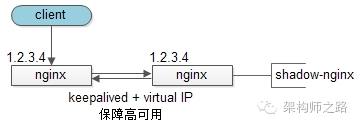
（5）**数据-缓存层**：缓存加速访问存储

（6）**数据-数据库层**：数据库固化数据存储

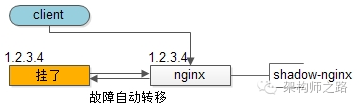
整个系统的高可用，又是通过每一层的**冗余+自动故障转移**来综合实现的。

**四、分层高可用架构实践**

**【客户端层->反向代理层】的高可用**

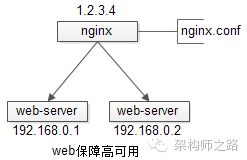


【客户端层】到【反向代理层】的**高可用**，是通过反向代理层的冗余来实现的。以nginx为例：有两台nginx，一台对线上提供服务，另一台冗余以保证高可用，常见的实践是keepalived存活探测，相同virtual IP提供服务。

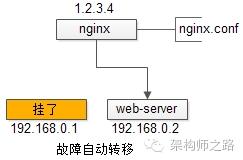


**自动故障转移**：当nginx挂了的时候，keepalived能够探测到，会自动的进行故障转移，将流量自动迁移到shadow-nginx，由于使用的是相同的virtual IP，这个切换过程对调用方是透明的。

**【反向代理层->站点层】的高可用**

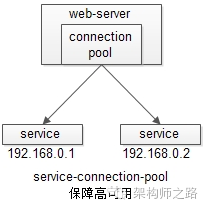


【反向代理层】到【站点层】的**高可用**，是通过站点层的冗余来实现的。假设反向代理层是nginx，nginx.conf里能够配置多个web后端，并且nginx能够探测到多个后端的存活性。

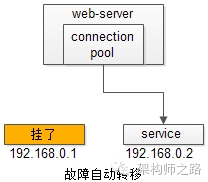


**自动故障转移**：当web-server挂了的时候，nginx能够探测到，会自动的进行故障转移，将流量自动迁移到其他的web-server，整个过程由nginx自动完成，对调用方是透明的。

**【站点层->服务层】的高可用**

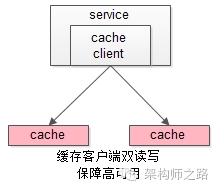


【站点层】到【服务层】的**高可用**，是通过服务层的冗余来实现的。“服务连接池”会建立与下游服务多个连接，每次请求会“随机”选取连接来访问下游服务。



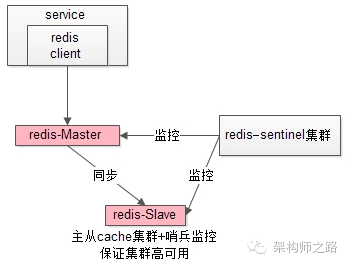
**自动故障转移**：当service挂了的时候，service-connection-pool能够探测到，会自动的进行故障转移，将流量自动迁移到其他的service，整个过程由连接池自动完成，对调用方是透明的（所以说RPC-client中的服务连接池是很重要的基础组件）。

**【服务层>缓存层】的高可用**



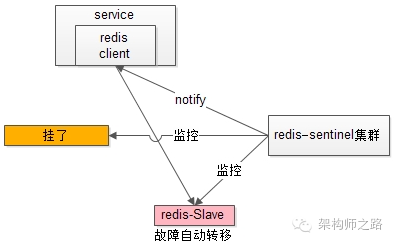
【服务层】到【缓存层】的高可用，是通过缓存数据的冗余来实现的。

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。

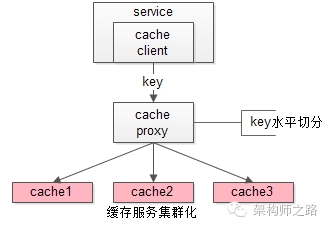
以redis为例，redis天然支持主从同步，redis官方也有sentinel哨兵机制，来做redis的存活性检测。



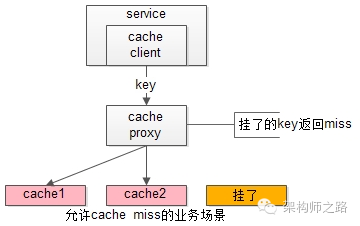
**自动故障转移**：当redis主挂了的时候，sentinel能够探测到，会通知调用方访问新的redis，整个过程由sentinel和redis集群配合完成，对调用方是透明的。

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。

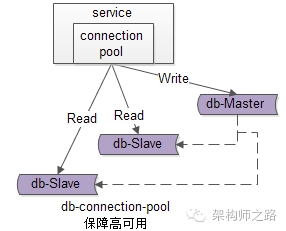


**缓存实例挂了屏蔽**：当有水平切分的实例挂掉时，代理层直接返回cache miss，此时缓存挂掉对调用方也是透明的。key水平切分实例减少，不建议做re-hash，这样容易引发缓存数据的不一致。

**【服务层>数据库层】的高可用**

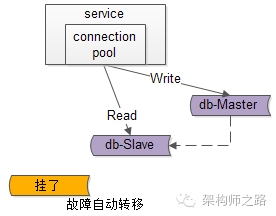
大部分互联网技术，数据库层都用了“主从同步，读写分离”架构，所以数据库层的高可用，又分为“读库高可用”与“写库高可用”两类。

**【服务层>数据库层“读”】的高可用**



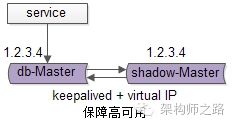
【服务层】到【数据库读】的**高可用**，是通过读库的冗余来实现的。

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。



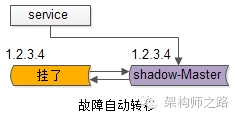
**自动故障转移**：当读库挂了的时候，db-connection-pool能够探测到，会自动的进行故障转移，将流量自动迁移到其他的读库，整个过程由连接池自动完成，对调用方是透明的（所以说DAO中的数据库连接池是很重要的基础组件）。

**【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】==

## 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】**

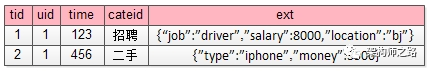
平台型创业型公司，可能有多个品类，例如58同城的招聘房产二手，很多异构数据的存储需求，到底是分还是合，无需纠结：**基础数据基础服务的统一**，无疑是58同城技术路线发展roadmap上最正确的决策之一，把这个方针坚持下来，@老崔 @晓飞 这些高瞻远瞩的先贤功不可没，业务线会有“扩展性”“灵活性”上的微词，后文看看先贤们如何通过一些巧妙的技术方案来解决的。

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？）



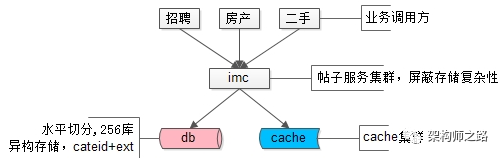
（3）通过ext来存储不同业务线的个性化需求

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

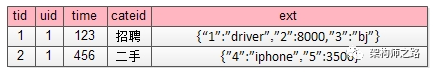
（2）cateid已经不足以描述ext内的内容，品类有层级，深度不确定，ext能否具备自描述性

（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}

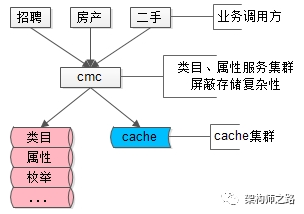
此外，类目属性服务还能记录类目之间的层级关系：

（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。

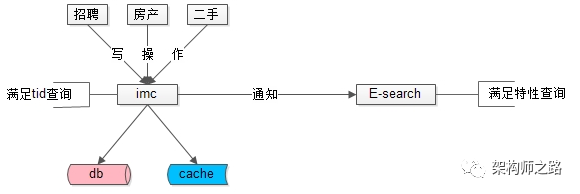
**【统一检索服务】**

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：

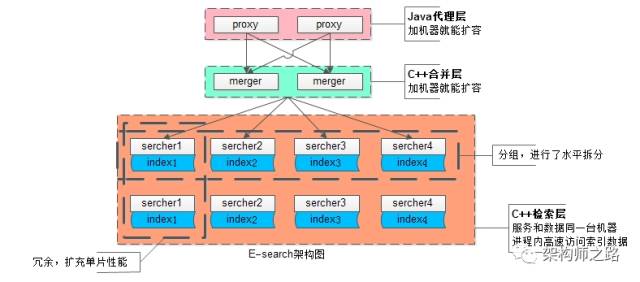
（1）对帖子进行tid正排查询，直接访问帖子服务

（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

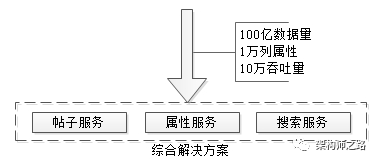
（3.1）为了满足数据容量的扩展性，索引数据进行了水平切分，增加切分份数，就能够无限扩展性能

（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

## 数据库软件架构设计些什么

2015-11-02 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400465735&idx=1&sn=8d7067de4cc8f73ea5558f07e0a9340e" \l "#)

缘起：受@萧田国 萧总邀请，上周五晚上在“高效运维1号群”内分享了《58同城数据库软件架构设计与实践》（这个topic今年在数据库大会上分享过），应组织方要求，发出纪要。

一、基本概念

二、数据库架构设计思路

（1）可用性

（2）读性能

（3）一致性

（4）扩展性

**一、基本概念**

**概念一“单库”**

http://mmbiz.qpic.cn/mmbiz/YrezxckhYOzbRcfS5SIbLMXktibZKEUDiaUiaPaJRzmaNftAkXQu3pVBjZELIMCicrXichu4BqNhZjFS5OHY5aof0dg/640?wx_fmt=png&wxfrom=5&wx_lazy=1

**概念二“分片”**

http://mmbiz.qpic.cn/mmbiz/YrezxckhYOzbRcfS5SIbLMXktibZKEUDia9VKZAyDpXM7T3OlvyMw5TP6kJPjDRr9Um7OTbpcEyeeT89gkDYoYHw/640?wx_fmt=png&wxfrom=5&wx_lazy=1

**分片解决的是“数据量太大”的问题**，也就是通常说的“水平切分”。

一旦引入分片，势必有“数据路由”的概念，哪个数据访问哪个库。

路由规则通常有3种方法：

（1）**范围**：range

优点：简单，容易扩展

缺点：各库压力不均（新号段更活跃）

（2）**哈希**：hash

优点：简单，数据均衡，负载均匀

缺点：迁移麻烦（2库扩3库数据要迁移）

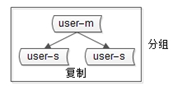
（3）**路由服务**：router-config-server

优点：灵活性强，业务与路由算法解耦

缺点：每次访问数据库前多一次查询

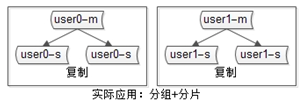
大部分互联网公司采用的方案二：哈希分库，哈希路由

**概念三“分组”**



**分组解决“可用性”问题**，分组通常通过主从复制的方式实现。

**互联网公司数据库实际软件架构是：又分片，又分组（如下图）**



**二、数据库架构设计思路**

数据库软件架构师平时设计些什么东西呢？至少要考虑以下四点：

（1）如何保证数据可用性

（2）如何提高数据库读性能（大部分应用读多写少，读会先成为瓶颈）

（3）如何保证一致性

（4）如何提高扩展性

**2.1如何保证数据的可用性？**

解决可用性问题的思路是=>冗余

如何保证站点的可用性？复制站点，冗余站点

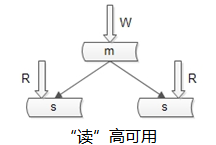
如何保证服务的可用性？复制服务，冗余服务

如何保证数据的可用性？复制数据，冗余数据

数据的冗余，会带来一个副作用=>引发一致性问题（先不说一致性问题，先说可用性）

**如何保证数据库“读”高可用？**

冗余读库

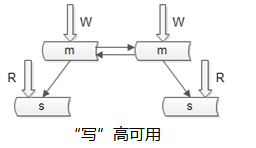


冗余读库带来的副作用？读写有延时，可能不一致

上面这个图是很多互联网公司mysql的架构，写仍然是单点，不能保证写高可用。

**如何保证数据库“写”高可用？**

冗余写库



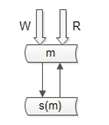
采用双主互备的方式，可以冗余写库

带来的副作用？双写同步，数据可能冲突（例如“自增id”同步冲突）,如何解决同步冲突，有两种常见解决方案：

（1）两个写库使用不同的初始值，相同的步长来增加id：1写库的id为0,2,4,6...；2写库的id为1,3,5,7…

（2）不使用数据的id，业务层自己生成唯一的id，保证数据不冲突

58同城没有使用上述两种架构来做读写的“高可用”，58同城采用的是“双主当主从用”的方式：



仍是双主，但只有一个主提供服务（读+写），另一个主是“shadow-master”，只用来保证高可用，平时不提供服务。

master挂了，shadow-master顶上（vip漂移，对业务层透明，不需要人工介入）

这种方式的**好处**：

1）读写没有延时

2）读写高可用

**不足**：

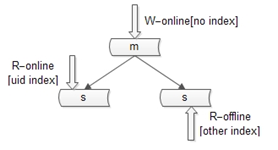
1）不能通过加从库的方式扩展读性能

2）资源利用率为50%，一台冗余主没有提供服务

那如何提高读性能呢？进入第二个话题，如何提供读性能。

**2.2如何扩展读性能？**

提高读性能的方式大致有三种，第一种是**建立索引**。这种方式不展开，要提到的一点是，不同的库可以建立不同的索引。



**写库**不建立索引；

**线上读库**建立线上访问索引，例如uid；

**线下读库**建立线下访问索引，例如time；

第二种扩充读性能的方式是，**增加从库**，这种方法大家用的比较多，但是，存在两个缺点：

（1）从库越多，同步越慢

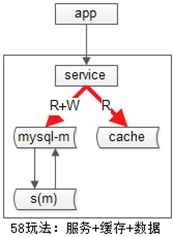
（2）同步越慢，数据不一致窗口越大（不一致后面说，还是先说读性能的提高）

58同城没有采用这种方法提高数据库读性能（没有从库），采用的是**增加缓存**。常见的缓存架构如下：



上游是业务应用，下游是主库，从库（读写分离），缓存。

58同城的玩法是：**服务+数据库+缓存一套**



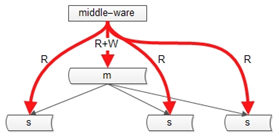
业务层不直接面向db和cache，服务层屏蔽了底层db、cache的复杂性。为什么要引入服务层，今天不展开，58采用了“服务+数据库+缓存一套”的方式提供数据访问，用cache提高读性能。

不管采用主从的方式扩展读性能，还是缓存的方式扩展读性能，数据都要复制多份（主+从，db+cache），一定会引发一致性问题。

**2.3如何保证一致性？**

主从数据库的一致性，通常有两种解决方案：

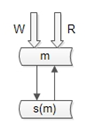
**（1）中间件**



如果某一个key有写操作，在不一致时间窗口内，中间件会将这个key的读操作也路由到主库上。

这个方案的缺点是，数据库中间件的门槛较高（百度，腾讯，阿里，360等一些公司有，当然58也有）

**（2）强制读主**



58的“双主当主从用”的架构，不存在主从不一致的问题。

第二类不一致，是db与缓存间的不一致



常见的缓存架构如上，此时**写操作的顺序是**：

（1）淘汰cache

（2）写数据库

**读操作的顺序是**：

（1）读cache，如果cache hit则返回

（2）如果cache miss，则读从库

（3）读从库后，将数据放回cache

在一些异常时序情况下，有可能从【从库读到旧数据（同步还没有完成），旧数据入cache后】，数据会长期不一致。

**解决办法是“缓存双淘汰”，**写操作时序升级为：

（1）淘汰cache

（2）写数据库

（3）在经验“主从同步延时窗口时间”后，再次发起一个异步淘汰cache的请求

这样，即使有脏数据如cache，一个小的时间窗口之后，脏数据还是会被淘汰。带来的代价是，多引入一次读miss（成本可以忽略）。

除此之外，58同城的最佳实践之一是：建议为所有cache中的item设置一个超时时间。

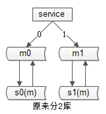
说完一致性，最后一个话题是扩展性。

**2.4如何提高数据库的扩展性？**

原来用hash的方式路由，分为2个库，数据量还是太大，要分为3个库，势必需要进行数据迁移，58同城有一个很帅气的“数据库秒级扩容”方案。

**如何秒级扩容？**

首先，我们不做2库变3库的扩容，我们做2库变4库（库加倍）的扩容（未来4->8->16）



服务+数据库是一套（省去了缓存）

数据库采用“双主”的模式。

**扩容步骤**：

第一步，将一个主库提升

第二步，修改配置，2库变4库（原来MOD2，现在配置修改后MOD4）

扩容完成



原MOD2为偶的部分，现在会MOD4余0或者2

原MOD2为奇的部分，现在会MOD4余1或者3

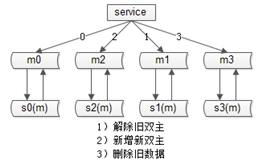
数据不需要迁移，同时，双主互相同步，一遍是余0，一边余2，两边数据同步也不会冲突，秒级完成扩容！

最后，要做一些收尾工作：

（1）将旧的双主同步解除

（2）增加新的双主（双主是保证可用性的，shadow-master平时不提供服务）

（3）删除多余的数据（余0的主，可以将余2的数据删除掉）



这样，秒级别内，我们就完成了2库变4库的扩展。

OK，今天主要分享了58同城，数据库软件架构上：

（1）如何保证数据可用性

（2）如何提高数据库读性能

（3）如何保证数据一致性

（4）如何进行秒级扩容

希望大家有收获，谢谢大家！

===【完】===

## 缓存架构设计细节二三事

原创 2016-03-08 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404087915&idx=1&sn=075664193f334874a3fc87fd4f712ebc&scene=21" \l "#)

本文主要讨论这么几个问题：

（1）“缓存与数据库”需求缘起

（2）“淘汰缓存”还是“更新缓存”

（3）缓存和数据库的操作时序

（4）缓存和数据库架构简析

**一、需求缘起**

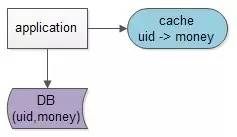
**场景介绍**

缓存是一种提高系统读性能的常见技术，对于读多写少的应用场景，我们经常使用缓存来进行优化。

例如对于用户的余额信息表account(uid, money)，业务上的需求是：

（1）查询用户的余额，SELECT money FROM account WHERE uid=XXX，占99%的请求

（2）更改用户余额，UPDATE account SET money=XXX WHERE uid=XXX，占1%的请求



由于大部分的请求是查询，我们在缓存中建立uid到money的键值对，能够极大降低数据库的压力。

**读操作流程**

有了数据库和缓存两个地方存放数据之后（uid->money），每当需要读取相关数据时（money），操作流程一般是这样的：

（1）读取缓存中是否有相关数据，uid->money

（2）如果缓存中有相关数据money，则返回【这就是所谓的**数据命中**“hit”】

（3）如果缓存中没有相关数据money，则从数据库读取相关数据money【这就是所谓的**数据未命中**“miss”】，放入缓存中uid->money，再返回

缓存的命中率 = 命中缓存请求个数/总缓存访问请求个数 = hit/(hit+miss)

上面举例的余额场景，99%的读，1%的写，这个缓存的命中率是非常高的，会在95%以上。

**那么问题来了**

当数据money发生变化的时候：

（1）是更新缓存中的数据，还是淘汰缓存中的数据呢？

（2）是先操纵数据库中的数据再操纵缓存中的数据，还是先操纵缓存中的数据再操纵数据库中的数据呢？

（3）缓存与数据库的操作，在架构上是否有优化的空间呢？

这是本文关注的三个核心问题。

**二、更新缓存 VS 淘汰缓存**

什么是更新缓存：数据不但写入数据库，还会写入缓存

什么是淘汰缓存：数据只会写入数据库，不会写入缓存，只会把数据淘汰掉

更新缓存的优点：缓存不会增加一次miss，命中率高

淘汰缓存的优点：简单（我去，更新缓存我也觉得很简单呀，楼主你太敷衍了吧）

**那到底是选择更新缓存还是淘汰缓存呢，主要取决于“更新缓存的复杂度”。**

例如，上述场景，只是简单的把余额money设置成一个值，那么：

（1）淘汰缓存的操作为deleteCache(uid)

（2）更新缓存的操作为setCache(uid, money)

更新缓存的代价很小，此时我们应该更倾向于更新缓存，以保证更高的缓存命中率

如果余额是通过很复杂的数据计算得出来的，例如业务上除了账户表account，还有商品表product，折扣表discount

account(uid, money)

product(pid, type, price, pinfo)

discount(type, zhekou)

业务场景是用户买了一个商品product，这个商品的价格是price，这个商品从属于type类商品，type类商品在做促销活动要打折扣zhekou，购买了商品过后，这个余额的计算就复杂了，需要：

（1）先把商品的品类，价格取出来：SELECT type, price FROM product WHERE pid=XXX

（2）再把这个品类的折扣取出来：SELECT zhekou FROM discount WHERE type=XXX

（3）再把原有余额从缓存中查询出来money = getCache(uid)

（4）再把新的余额写入到缓存中去setCache(uid, money-price\*zhekou)

更新缓存的代价很大，此时我们应该更倾向于淘汰缓存。

however，淘汰缓存操作简单，并且带来的副作用只是增加了一次cache miss，建议作为通用的处理方式。

**三、先操作数据库 vs 先操作缓存**

OK，当写操作发生时，假设淘汰缓存作为对缓存通用的处理方式，又面临两种抉择：

（1）先写数据库，再淘汰缓存

（2）先淘汰缓存，再写数据库

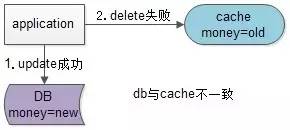
究竟采用哪种时序呢？

还记得在《[冗余表如何保证数据一致性](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403963671&idx=1&sn=51a2d2fd70212451cd5f22bbe2c6f8d6&scene=21" \l "wechat_redirect" \t "_blank)》文章（点击查看）里“究竟先写正表还是先写反表”的结论么？

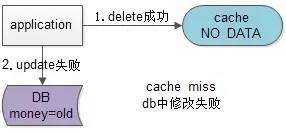
对于一个不能保证事务性的操作，一定涉及“哪个任务先做，哪个任务后做”的问题，解决这个问题的方向是：

**如果出现不一致，谁先做对业务的影响较小，就谁先执行。**

由于写数据库与淘汰缓存不能保证原子性，谁先谁后同样要遵循上述原则。



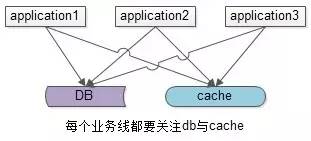
**假设先写数据库，再淘汰缓存**：第一步写数据库操作成功，第二步淘汰缓存失败，则会出现DB中是新数据，Cache中是旧数据，数据不一致。



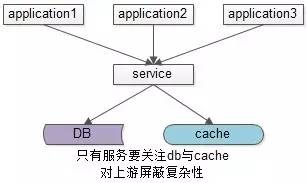
**假设先淘汰缓存，再写数据库**：第一步淘汰缓存成功，第二步写数据库失败，则只会引发一次Cache miss。

**结论：数据和缓存的操作时序，结论是清楚的：先淘汰缓存，再写数据库。**

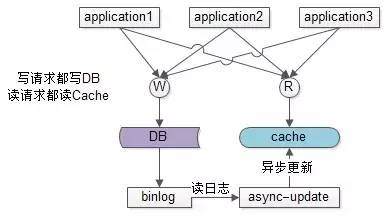
**四、缓存架构优化**



上述缓存架构有一个**缺点**：业务方需要同时关注缓存与DB，有没有进一步的优化空间呢？有两种常见的方案，一种主流方案，一种非主流方案（一家之言，勿拍）。



主流优化方案是**服务化**：加入一个服务层，向上游提供帅气的数据访问接口，向上游屏蔽底层数据存储的细节，这样业务线不需要关注数据是来自于cache还是DB。



非主流方案是**异步缓存更新**：业务线所有的写操作都走数据库，所有的读操作都总缓存，由一个异步的工具来做数据库与缓存之间数据的同步，具体细节是：

（1）要有一个init cache的过程，将需要缓存的数据全量写入cache

（2）如果DB有写操作，异步更新程序读取binlog，更新cache

在（1）和（2）的合作下，cache中有全部的数据，这样：

（a）业务线读cache，一定能够hit（很短的时间内，可能有脏数据），无需关注数据库

（b）业务线写DB，cache中能得到异步更新，无需关注缓存

这样将**大大简化业务线的调用逻辑**，存在的缺点是，如果缓存的数据业务逻辑比较复杂，async-update异步更新的逻辑可能也会比较复杂。

**五、其他未尽事宜**

本文只讨论了缓存架构设计中需要注意的几个细节点，如果数据库架构采用了一主多从，读写分离的架构，在特殊时序下，还很可能引发数据库与缓存的不一致，这个不一致如何优化，后续的文章再讨论吧。

**六、结论强调**

（1）淘汰缓存是一种通用的缓存处理方式

（2）先淘汰缓存，再写数据库的时序是毋庸置疑的

（3）服务化是向业务方屏蔽底层数据库与缓存复杂性的一种通用方式

**七、关于infoQ**

架构师之路并不好走。这需要过硬的技术能力、出色的综合素质、灵活的大脑，更需要丰富的专业知识。

InfoQ是一家坚持原创的深度技术社区，其旗下的QCon、ArchSummit大会也享有盛名。走在架构师之路上的，都应该多关注这个微信号。他们，是前沿技术的传播者。扫描二维码关注InfoQ，回复关键字“女王”有惊喜！（据说有码农界最漂亮的程序媛们的大尺度照片。）

## 细聊冗余表数据一致性（架构师之路）

原创 2016-03-01 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403963671&idx=1&sn=51a2d2fd70212451cd5f22bbe2c6f8d6&scene=21" \l "#)

本文主要讨论四个问题：

（1）为什么会有冗余表的需求

（2）如何实现冗余表

（3）正反冗余表谁先执行

（4）冗余表如何保证数据的一致性

**一、需求缘起**

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

例如订单表，业务上对用户和商家都有订单查询需求：

Order(oid, info\_detail)

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

如果用buyer\_id来分库，seller\_id的查询就需要扫描多库。

如果用seller\_id来分库，buyer\_id的查询就需要扫描多库。

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

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

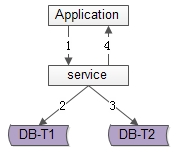
T2(seller\_id, buyer\_id, oid)

同一个数据，冗余两份，一份以buyer\_id来分库，满足买家的查询需求；

一份以seller\_id来分库，满足卖家的查询需求。

**二、冗余表的实现方案**

**【方法一：服务同步写】**



顾名思义，由服务层同步写冗余数据，如上图1-4流程：

（1）业务方调用服务，新增数据

（2）服务先插入T1数据

（3）服务再插入T2数据

（4）服务返回业务方新增数据成功

**优点**：

（1）不复杂，服务层由单次写，变两次写

（2）数据一致性相对较高（因为双写成功才返回）

**缺点**：

（1）请求的处理时间增加（要插入次，时间加倍）

（2）数据仍可能不一致，例如第二步写入T1完成后服务重启，则数据不会写入T2

如果系统对处理时间比较敏感，引出常用的第二种方案

**【方法二：服务异步写】**



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

（1）业务方调用服务，新增数据

（2）服务先插入T1数据

（3）服务向消息总线发送一个异步消息（发出即可，不用等返回，通常很快就能完成）

（4）服务返回业务方新增数据成功

（5）消息总线将消息投递给数据同步中心

（6）数据同步中心插入T2数据

**优点**：

（1）请求处理时间短（只插入1次）

**缺点**：

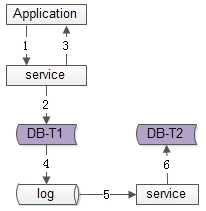
（1）系统的复杂性增加了，多引入了一个组件（消息总线）和一个服务（专用的数据复制服务）

（2）因为返回业务线数据插入成功时，数据还不一定插入到T2中，因此数据有一个不一致时间窗口（这个窗口很短，最终是一致的）

（3）在消息总线丢失消息时，冗余表数据会不一致

如果想解除“数据冗余”对系统的耦合，引出常用的第三种方案

**【方法三：线下异步写】**



数据的双写不再由服务层来完成，而是由线下的一个服务或者任务来完成，如上图1-6流程：

（1）业务方调用服务，新增数据

（2）服务先插入T1数据

（3）服务返回业务方新增数据成功

（4）数据会被写入到数据库的log中

（5）线下服务或者任务读取数据库的log

（6）线下服务或者任务插入T2数据

**优点**：

（1）数据双写与业务完全解耦

（2）请求处理时间短（只插入1次）

**缺点**：

（1）返回业务线数据插入成功时，数据还不一定插入到T2中，因此数据有一个不一致时间窗口（这个窗口很短，最终是一致的）

（2）数据的一致性依赖于线下服务或者任务的可靠性

上述三种方案各有优缺点，但不管哪种方案，都会面临“究竟先写T1还是先写T2”的问题？这该怎么办呢？

**三、究竟先写正表还是反表**

对于一个不能保证事务性的操作，一定涉及“哪个任务先做，哪个任务后做”的问题，解决这个问题的方向是：

**【如果出现不一致】，谁先做对业务的影响较小，就谁先执行。**

以上文的订单生成业务为例，buyer和seller冗余表都需要插入数据：

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

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

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

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

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

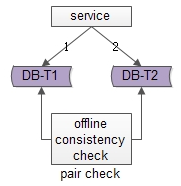
因此，在此例中，应该先插入buyer表T1，再插入seller表T2。

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

**四、如何保证数据的一致性**

从二节和第三节的讨论可以看到，不管哪种方案，因为两步操作不能保证原子性，总有出现数据不一致的可能，那如何解决呢？

**【方法一：线下扫面正反冗余表全部数据】**



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

**优点**：

（1）比较简单，开发代价小

（2）线上服务无需修改，修复工具与线上服务解耦

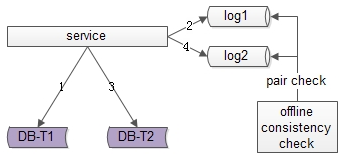
**缺点**：

（1）扫描效率低，会扫描大量的“已经能够保证一致”的数据

（2）由于扫描的数据量大，扫描一轮的时间比较长，即数据如果不一致，不一致的时间窗口比较长

有没有只扫描“可能存在不一致可能性”的数据，而不是每次扫描全部数据，以提高效率的优化方法呢？

**【方法二：线下扫描增量数据】**



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

（1）写入正表T1

（2）第一步成功后，写入日志log1

（3）写入反表T2

（4）第二步成功后，写入日志log2

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

**优点**：

（1）虽比方法一复杂，但仍然是比较简单的

（2）数据扫描效率高，只扫描增量数据

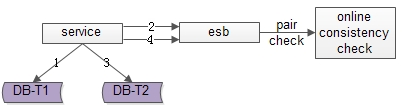
**缺点**：

（1）线上服务略有修改（代价不高，多写了2条日志）

（2）虽然比方法一更实时，但时效性还是不高，不一致窗口取决于扫描的周期

有没有实时检测一致性并进行修复的方法呢？

**【方法三：实时线上“消息对”检测】**



这次不是写日志了，而是向消息总线发送消息，如上图1-4流程所示：

（1）写入正表T1

（2）第一步成功后，发送消息msg1

（3）写入反表T2

（4）第二步成功后，发送消息msg2

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

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

**优点**：

（1）效率高

（2）实时性高

**缺点**：

（1）方案比较复杂，上线引入了消息总线这个组件

（2）线下多了一个订阅总线的检测服务

however，技术方案本身就是一个投入产出比的折衷，可以根据业务对一致性的需求程度决定使用哪一种方法。我这边有过好友数据正反表的业务，使用的就是方法二。

==【完】==

回【58】[58怎么玩数据库架构](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403369608&idx=1&sn=48c56d64c2b5c27fed795219e7ba214a&scene=21" \l "wechat_redirect" \t "_blank)

回【微信】[微信为啥这么省流量](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400163013&idx=1&sn=911cf71925e2ba50d47955a713134acb&scene=21" \l "wechat_redirect" \t "_blank)

回【秒杀】[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=204965266&idx=1&sn=aad227dbcb544394ae1717e6af8b77ae&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

回【id】[细聊分布式ID生成方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "wechat_redirect" \t "_blank)

回【招聘】入职58到家

## 缓存与数据库一致性保证

原创 2016-03-16 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404202261&idx=1&sn=1b8254ba5013952923bdc21e0579108e&scene=21" \l "#)

本文主要讨论这么几个问题：

（1）啥时候数据库和缓存中的数据会不一致

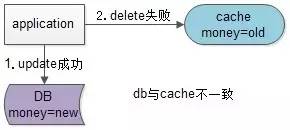
（2）不一致优化思路

（3）如何保证数据库与缓存的一致性

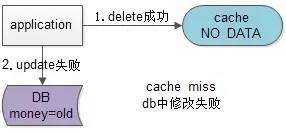
### ****一、需求缘起****

上一篇《[缓存架构设计细节二三事](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404087915&idx=1&sn=075664193f334874a3fc87fd4f712ebc&scene=21" \l "wechat_redirect" \t "_blank)》（点击查看）引起了广泛的讨论，其中有一个结论：当数据发生变化时，“**先淘汰缓存，再修改数据库**”这个点是大家讨论的最多的。

上篇文章得出这个结论的依据是，由于操作缓存与操作数据库不是原子的，非常有可能出现执行失败。



假设先写数据库，再淘汰缓存：第一步写数据库操作成功，第二步淘汰缓存失败，则会出现DB中是新数据，Cache中是旧数据，数据不一致【如上图：db中是新数据，cache中是旧数据】。



假设先淘汰缓存，再写数据库：第一步淘汰缓存成功，第二步写数据库失败，则只会引发一次Cache miss【如上图：cache中无数据，db中是旧数据】。

**结论：先淘汰缓存，再写数据库。**

引发大家热烈讨论的点是“先操作缓存，在写数据库成功之前，如果有读请求发生，可能导致旧数据入缓存，引发数据不一致”，这就是本文要讨论的主题。

### ****二、为什么数据会不一致****

回顾一下上一篇文章中对缓存、数据库进行读写操作的流程。

写流程：

（1）先淘汰cache

（2）再写db

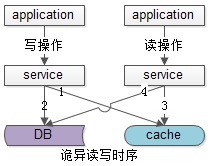
读流程：

（1）先读cache，如果数据命中hit则返回

（2）如果数据未命中miss则读db

（3）将db中读取出来的数据入缓存

什么情况下可能出现缓存和数据库中数据不一致呢？



在分布式环境下，数据的读写都是并发的，上游有多个应用，通过一个服务的多个部署（为了保证可用性，一定是部署多份的），对同一个数据进行读写，在数据库层面并发的读写并不能保证完成顺序，也就是说后发出的读请求很可能先完成（读出脏数据）：

（a）发生了写请求A，A的第一步淘汰了cache（如上图中的1）

（b）A的第二步写数据库，发出修改请求（如上图中的2）

（c）发生了读请求B，B的第一步读取cache，发现cache中是空的（如上图中的步骤3）

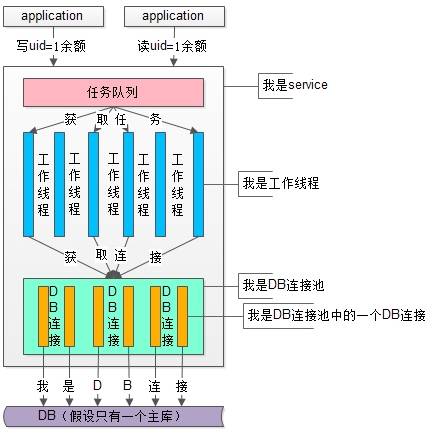
（d）B的第二步读取数据库，发出读取请求，此时**A的第二步写数据还没完成**，读出了一个脏数据放入cache（如上图中的步骤4）

即在数据库层面，后发出的请求4比先发出的请求2先完成了，读出了脏数据，脏数据又入了缓存，缓存与数据库中的数据不一致出现了

### ****三、不一致优化思路****

**能否做到先发出的请求一定先执行完成呢？**常见的思路是“串行化”，今天将和大家一起探讨“串行化”这个点。

先一起细看一下，在一个服务中，并发的多个读写SQL一般是怎么执行的



上图是一个service服务的上下游及服务内部详细展开，细节如下：

（1）service的上游是多个业务应用，上游发起请求对同一个数据并发的进行读写操作，上例中并发进行了一个uid=1的余额修改（写）操作与uid=1的余额查询（读）操作

（2）service的下游是数据库DB，假设只读写一个DB

（3）中间是服务层service，它又分为了这么几个部分

（3.1）最上层是任务队列

（3.2）中间是工作线程，每个工作线程完成实际的工作任务，典型的工作任务是通过数据库连接池读写数据库

（3.3）最下层是数据库连接池，所有的SQL语句都是通过数据库连接池发往数据库去执行的

工作线程的典型工作流是这样的：

void work\_thread\_routine(){

Task t = TaskQueue.pop(); // 获取任务

// 任务逻辑处理，生成sql语句

DBConnection c = CPool.GetDBConnection(); // 从DB连接池获取一个DB连接

c.execSQL(sql); // 通过DB连接执行sql语句

CPool.PutDBConnection(c); // 将DB连接放回DB连接池

}

**提问：任务队列其实已经做了任务串行化的工作，能否保证任务不并发执行？**

答：不行，因为

（1）1个服务有多个工作线程，串行弹出的任务会被并行执行

（2）1个服务有多个数据库连接，每个工作线程获取不同的数据库连接会在DB层面并发执行

**提问：假设服务只部署一份，能否保证任务不并发执行？**

答：不行，原因同上

**提问：假设1个服务只有1条数据库连接，能否保证任务不并发执行？**

答：不行，因为

（1）1个服务只有1条数据库连接，只能保证在一个服务器上的请求在数据库层面是串行执行的

（2）因为服务是分布式部署的，多个服务上的请求在数据库层面仍可能是并发执行的

**提问：假设服务只部署一份，且1个服务只有1条连接，能否保证任务不并发执行？**

答：可以，全局来看请求是串行执行的，吞吐量很低，并且服务无法保证可用性

完了，看似无望了，

1）任务队列不能保证串行化

2）单服务多数据库连接不能保证串行化

3）多服务单数据库连接不能保证串行化

4）单服务单数据库连接可能保证串行化，但吞吐量级低，且不能保证服务的可用性，几乎不可行，那是否还有解？

退一步想，其实不需要让全局的请求串行化，而只需要“让同一个数据的访问能串行化”就行。

在一个服务内，如何做到“让同一个数据的访问串行化”，只需要“让同一个数据的访问通过同一条DB连接执行”就行。

如何做到“让同一个数据的访问通过同一条DB连接执行”，只需要“在DB连接池层面稍微修改，按数据取连接即可”

获取DB连接的CPool.GetDBConnection()【返回任何一个可用DB连接】改为

CPool.GetDBConnection(longid)【返回id取模相关联的DB连接】

这个修改的好处是：

（1）简单，只需要修改DB连接池实现，以及DB连接获取处

（2）连接池的修改不需要关注业务，传入的id是什么含义连接池不关注，直接按照id取模返回DB连接即可

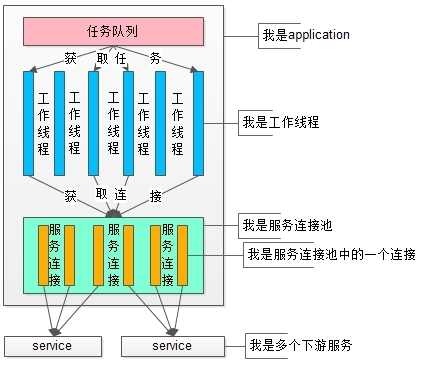
（3）可以适用多种业务场景，取用户数据业务传入user-id取连接，取订单数据业务传入order-id取连接即可

这样的话，就能够保证同一个数据例如uid在数据库层面的执行一定是串行的

稍等稍等，服务可是部署了很多份的，上述方案只能保证同一个数据在一个服务上的访问，在DB层面的执行是串行化的，实际上服务是分布式部署的，在全局范围内的访问仍是并行的，怎么解决呢？能不能做到同一个数据的访问一定落到同一个服务呢？

### ****四、能否做到同一个数据的访问落在同一个服务上？****

上面分析了服务层service的上下游及内部结构，再一起看一下应用层上下游及内部结构



上图是一个业务应用的上下游及服务内部详细展开，细节如下：

（1）业务应用的上游不确定是啥，可能是直接是http请求，可能也是一个服务的上游调用

（2）业务应用的下游是多个服务service

（3）中间是业务应用，它又分为了这么几个部分

（3.1）最上层是任务队列【或许web-server例如tomcat帮你干了这个事情了】

（3.2）中间是工作线程【或许web-server的工作线程或者cgi工作线程帮你干了线程分派这个事情了】，每个工作线程完成实际的业务任务，典型的工作任务是通过服务连接池进行RPC调用

（3.3）最下层是服务连接池，所有的RPC调用都是通过服务连接池往下游服务去发包执行的

工作线程的典型工作流是这样的：

voidwork\_thread\_routine(){

Task t = TaskQueue.pop(); // 获取任务

// 任务逻辑处理，组成一个网络包packet，调用下游RPC接口

ServiceConnection c = CPool.GetServiceConnection(); // 从Service连接池获取一个Service连接

c.Send(packet); // 通过Service连接发送报文执行RPC请求

CPool.PutServiceConnection(c); // 将Service连接放回Service连接池

}

似曾相识吧？没错，只要对服务连接池进行少量改动：

获取Service连接的CPool.GetServiceConnection()【返回任何一个可用Service连接】改为

CPool.GetServiceConnection(longid)【返回id取模相关联的Service连接】

这样的话，就能够保证同一个数据例如uid的请求落到同一个服务Service上。

### ****五、总结****

由于数据库层面的读写并发，引发的数据库与缓存数据不一致的问题（本质是后发生的读请求先返回了），可能通过两个小的改动解决：

（1）修改服务Service连接池，id取模选取服务连接，能够保证同一个数据的读写都落在同一个后端服务上

（2）修改数据库DB连接池，id取模选取DB连接，能够保证同一个数据的读写在数据库层面是串行的

### ****六、遗留问题****

**提问：取模访问服务是否会影响服务的可用性？**

答：不会，当有下游服务挂掉的时候，服务连接池能够检测到连接的可用性，取模时要把不可用的服务连接排除掉。

**提问：取模访问服务与 取模访问DB，是否会影响各连接上请求的负载均衡？**

答：不会，只要数据访问id是均衡的，从全局来看，由id取模获取各连接的概率也是均等的，即负载是均衡的。

**提问：要是数据库的架构做了主从同步，读写分离：写请求写主库，读请求读从库也有可能导致缓存中进入脏数据呀，这种情况怎么解决呢（读写请求根本不落在同一个DB上，并且读写DB有同步时延）？**

答：下一篇文章和大家分享。

## 主从DB与cache一致性

原创 2016-03-24 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404308725&idx=1&sn=1a25ce76dd1956014ceb8a011855268e&scene=21" \l "#)

本文主要讨论这么几个问题：

（1）数据库主从延时为何会导致缓存数据不一致

（2）优化思路与方案

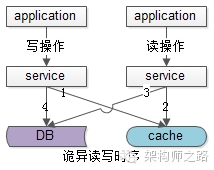
### ****一、需求缘起****

上一篇《缓存架构设计细节二三事》中有一个小优化点，在只有主库时，通过“串行化”的思路可以解决缓存与数据库中数据不一致。引发大家热烈讨论的点是“**在主从同步，读写分离的数据库架构下，有可能出现脏数据入缓存的情况，此时串行化方案不再适用了**”，这就是本文要讨论的主题。

### ****二、为什么数据会不一致****

为什么会读到脏数据，有这么几种情况：

（1）单库情况下，服务层的并发读写，缓存与数据库的操作交叉进行



虽然只有一个DB，在上述诡异异常时序下，也可能脏数据入缓存：

1）请求A发起一个写操作，第一步淘汰了cache，然后这个请求因为各种原因在服务层卡住了（进行大量的业务逻辑计算，例如计算了1秒钟），如上图步骤1

2）请求B发起一个读操作，读cache，cache miss，如上图步骤2

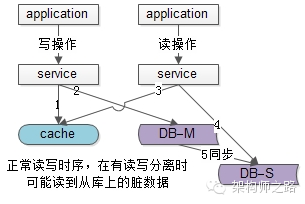
3）请求B继续读DB，读出来一个脏数据，然后脏数据入cache，如上图步骤3

4）请求A卡了很久后终于写数据库了，写入了最新的数据，如上图步骤4

**这种情况虽然少见，但理论上是存在的**， 后发起的请求B在先发起的请求A中间完成了。

（2）主从同步，读写分离的情况下，读从库读到旧数据

在数据库架构做了一主多从，读写分离时，更多的脏数据入缓存是下面这种情况：



1）请求A发起一个写操作，第一步淘汰了cache，如上图步骤1

2）请求A写数据库了，写入了最新的数据，如上图步骤2

3）请求B发起一个读操作，读cache，cache miss，如上图步骤3

4）请求B继续读DB，读的是从库，此时主从同步还没有完成，读出来一个脏数据，然后脏数据入cache，如上图步4

5）最后数据库的主从同步完成了，如上图步骤5

这种情况**请求A和请求B的时序是完全没有问题的，是主动同步的时延**（假设延时1秒钟）中间有读请求读从库读到脏数据导致的不一致。

那怎么来进行优化呢？

### ****三、不一致优化思路****

有同学说“那能不能先操作数据库，再淘汰缓存”，这个是不行的，在[《缓存和数据库先操作谁》](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404087915&idx=1&sn=075664193f334874a3fc87fd4f712ebc&scene=21" \l "wechat_redirect" \t "_blank)的文章中介绍过。

出现不一致的根本原因：

（1）单库情况下，服务层在进行1s的逻辑计算过程中，可能读到旧数据入缓存

（2）主从库+读写分离情况下，在1s钟主从同步延时过程中，可能读到旧数据入缓存

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

答案是可以的。

写请求的步骤由2步升级为3步：

（1）先淘汰缓存

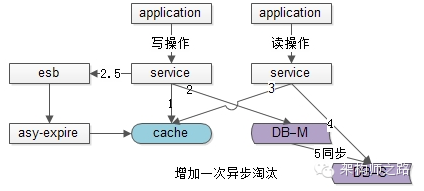
（2）再写数据库（这两步和原来一样）

（3）休眠1秒，再次淘汰缓存

这样的话，1秒内有脏数据如缓存，也会被再次淘汰掉，但带来的**问题**是：

（1）所有的写请求都阻塞了1秒，大大降低了写请求的吞吐量，增长了处理时间，业务上是接受不了的

再次分析，其实第二次淘汰缓存是“为了保证缓存一致”而做的操作，而不是“业务要求”，所以**其实无需等待，用一个异步的timer，或者利用消息总线异步的来做这个事情即可**：



写请求由2步升级为2.5步：

（1）先淘汰缓存

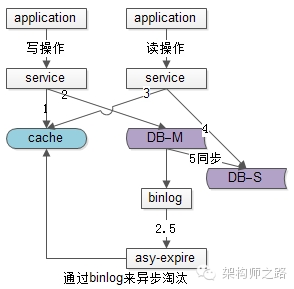
（2）再写数据库（这两步和原来一样）

（2.5）不再休眠1s，而是往消息总线esb发送一个消息，发送完成之后马上就能返回

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

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

上述方案有一个**缺点**，需要业务线的写操作增加一个步骤，**有没有方案对业务线的代码没有任何入侵呢**，是有的，这个方案在[《细聊冗余表数据一致性》](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403963671&idx=1&sn=51a2d2fd70212451cd5f22bbe2c6f8d6&scene=21" \l "wechat_redirect" \t "_blank)中也提到过，通过分析线下的binlog来异步淘汰缓存：



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

**提问：为什么上文总是说1s，这个1s是怎么来的？**

回答：1s只是一个举例，需要根据业务的数据量与并发量，观察主从同步的时延来设定这个值。例如主从同步的时延为200ms，这个异步淘汰cache设置为258ms就是OK的。

### ****四、总结****

在“异常时序”或者“读从库”导致脏数据入缓存时，可以用**二次异步淘汰**的“**缓存双淘汰**”法来解决缓存与数据库中数据不一致的问题，具体实施至少有三种方案：

（1）**timer异步淘汰**（本文没有细讲，本质就是起个线程专门异步二次淘汰缓存）

（2）**总线异步淘汰**

（3）**读binlog异步淘汰**

==【完】==

回【冗余】[细聊冗余表数据一致性](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403963671&idx=1&sn=51a2d2fd70212451cd5f22bbe2c6f8d6&scene=21" \l "wechat_redirect" \t "_blank)

回【一致】[缓存与数据库一致性保证](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404202261&idx=1&sn=1b8254ba5013952923bdc21e0579108e&scene=21" \l "wechat_redirect" \t "_blank)

回【缓存】[缓存架构设计细节二三事](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404087915&idx=1&sn=075664193f334874a3fc87fd4f712ebc&scene=21" \l "wechat_redirect" \t "_blank)

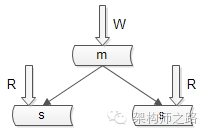
【小游戏：回大于10的整数，随机返回好文】

## DB主从一致性架构优化4种方法

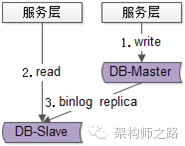
原创 2016-05-18 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959442&idx=1&sn=feb8ff75385d8031386e120ef3535329&scene=21" \l "#)

**需求缘起**

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



这种架构的一个潜在**缺点**是，业务方有可能读取到并不是最新的旧数据：



（1）系统先对DB-master进行了一个写操作，写主库

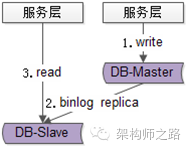
（2）很短的时间内并发进行了一个读操作，读从库，此时主从同步没有完成，故读取到了一个旧数据

（3）主从同步完成

有没有办法解决或者缓解这类**“由于主从延时导致读取到旧数据”**的问题呢，这是本文要集中讨论的问题。

**方案一（半同步复制）**

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



（1）系统先对DB-master进行了一个写操作，写主库

（2）等主从同步完成，写主库的请求才返回

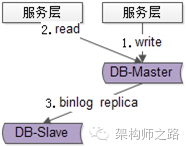
（3）读从库，读到最新的数据（如果读请求先完成，写请求后完成，读取到的是“当时”最新的数据）

**方案优点**：利用数据库原生功能，比较简单

**方案缺点**：主库的写请求时延会增长，吞吐量会降低

**方案二（强制读主库）**

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



**方案优点**：“一致性”上不需要进行系统改造

**方案缺点**：只能通过cache来提升系统的读性能，这里要进行系统改造

**方案三（数据库中间件）**

如果有了数据库中间件，所有的数据库请求都走中间件，这个主从不一致的问题可以这么解决：



（1）所有的读写都走数据库中间件，通常情况下，写请求路由到主库，读请求路由到从库

（2）记录所有路由到写库的key，在经验主从同步时间窗口内（假设是500ms），如果有读请求访问中间件，此时有可能从库还是旧数据，就把这个key上的读请求路由到主库

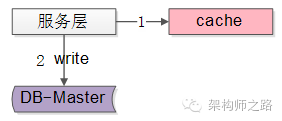
（3）经验主从同步时间过完后，对应key的读请求继续路由到从库

**方案优点**：能保证绝对一致

**方案缺点**：数据库中间件的成本比较高

**方案四（缓存记录写key法）**

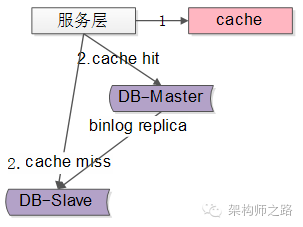
既然数据库中间件的成本比较高，有没有更低成本的方案来记录某一个库的某一个key上发生了写请求呢？很容易想到使用缓存，当写请求发生的时候：



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

（2）修改数据库

而读请求发生的时候：



（1）先到cache里查看，对应库的对应key有没有相关数据

（2）如果cache hit，有相关数据，说明这个key上刚发生过写操作，此时需要将请求路由到主库读最新的数据

（3）如果cache miss，说明这个key上近期没有发生过写操作，此时将请求路由到从库，继续读写分离

**方案优点**：相对数据库中间件，成本较低

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

**总结**

为了解决主从数据库读取旧数据的问题，常用的方案有四种：

（1）半同步复制

（2）强制读主

（3）数据库中间件

（4）缓存记录写key

前3个方案在今年数据库大会（DTCC2016）上share过，相关的材料在网上能下载到。第4个方案是大会现场有其他同学share的一个好方法，感谢这位同学。

==【全文完，希望大伙有收获】==

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

回【秒杀】[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959391&idx=1&sn=fb28fd5e5f0895ddb167406d8a735548&scene=21" \l "wechat_redirect" \t "_blank)

回【消息】[58到家通用实时消息平台架构细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "wechat_redirect" \t "_blank)

回【id】[细聊分布式ID生成方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "wechat_redirect" \t "_blank)

回【监控】[创业公司快速搭建立体化监控](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959433&idx=1&sn=9c9a222c06b2426f3935b04a10f90c0b&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

## 多库多事务降低数据不一致概率

原创 2016-07-21 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959486&idx=1&sn=cd3e6b27af159f4eabd54f148eec638a&scene=21" \l "#)

### ****一、案例缘起****

我们经常使用事务来保证数据库层面数据的ACID特性。

举个栗子，用户下了一个订单，需要修改余额表，订单表，流水表，于是会有类似的伪代码：

start transaction;

         CURDtable t\_account;  any Exception rollback;

         CURDtable t\_order;       any Exceptionrollback;

         CURDtable t\_flow;         any Exceptionrollback;

commit;

如果对余额表，订单表，流水表的SQL操作全部成功，则全部提交，如果任何一个出现问题，则全部回滚，以保证数据的一致性。

互联网的业务特点，数据量较大，并发量较大，经常使用拆库的方式提升系统的性能。**如果进行了拆库，余额、订单、流水可能分布在不同的数据库上，甚至不同的数据库实例上，此时就不能用事务来保证数据的一致性了。**这种情况下如何保证数据的一致性，是今天要讨论的话题。

### ****二、补偿事务****

补偿事务是一种在业务端实施业务逆向操作事务，来保证业务数据一致性的方式。

举个栗子，修改余额表事务为

int Do\_AccountT(uid, money){

start transaction;

         //余额改变money这么多

         CURDtable t\_account with money;       anyException rollback return NO;

commit;

return YES;

}

那么补偿事务可以是：

int Compensate\_AccountT(uid, money){

         //做一个money的反向操作

         returnDo\_AccountT(uid, -1\*money){

}

同理，订单表操作为

Do\_OrderT，新增一个订单

Compensate\_OrderT，删除一个订单

要保重余额与订单的一致性，可能要写这样的代码：

// 执行第一个事务

int flag = Do\_AccountT();

if(flag=YES){

         //第一个事务成功，则执行第二个事务

         flag= Do\_OrderT();

         if(flag=YES){

                  // 第二个事务成功，则成功

                   returnYES;

}

else{

         // 第二个事务失败，执行第一个事务的补偿事务

         Compensate\_AccountT();

}

}

该方案的**不足**是：

（1）不同的业务要写不同的补偿事务，不具备通用性

（2）没有考虑补偿事务的失败

（3）如果业务流程很复杂，if/else会嵌套非常多层

例如，如果上面的例子加上流水表的修改，加上Do\_FlowT和Compensate\_FlowT，可能会变成一个这样的if/else：

// 执行第一个事务

int flag = Do\_AccountT();

if(flag=YES){

         //第一个事务成功，则执行第二个事务

         flag= Do\_OrderT();

         if(flag=YES){

                  // 第二个事务成功，则执行第三个事务

                   flag= Do\_FlowT();

                   if(flag=YES){

                            //第三个事务成功，则成功

                            returnYES;

}

else{

         // 第三个事务失败，则执行第二、第一个事务的补偿事务

         flag =Compensate\_OrderT();

         if … else … // 补偿事务执行失败？

                  flag= Compensate\_AccountT();

                   if … else … // 补偿事务执行失败？

}

}

else{

         // 第二个事务失败，执行第一个事务的补偿事务

         Compensate\_AccountT();

         if … else … // 补偿事务执行失败？

}

}

### ****三、事务拆分分析与后置提交优化****

单库是用这样一个大事务保证一致性：

start transaction;

         CURDtable t\_account;  any Exception rollback;

         CURDtable t\_order;       any Exceptionrollback;

         CURDtable t\_flow;         any Exceptionrollback;

commit;

拆分成了多个库，大事务会变成三个小事务：

start transaction1;

         //第一个库事务执行

         CURDtable t\_account;  any Exception rollback;

         …

// 第一个库事务提交

commit1;

start transaction2;

         //第二个库事务执行

         CURDtable t\_order;       any Exceptionrollback;

         …

// 第二个库事务提交

commit2;

start transaction3;

         //第三个库事务执行

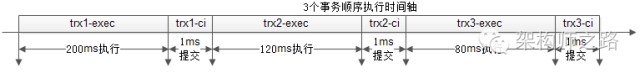
         CURDtable t\_flow;         any Exceptionrollback;

         …

// 第三个库事务提交

commit3;

**一个事务，分成执行与提交两个阶段，执行的时间其实是很长的，而commit的执行其实是很快的**，于是整个执行过程的时间轴如下：



第一个事务执行200ms，提交1ms；

第二个事务执行120ms，提交1ms；

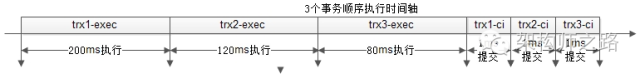
第三个事务执行80ms，提交1ms；

**那在什么时候系统出现问题，会出现不一致呢？**

**回答**：第一个事务成功提交之后，最后一个事务成功提交之前，如果出现问题（例如服务器重启，数据库异常等），都可能导致数据不一致。



如果改变事务执行与提交的时序，变成事务先执行，最后一起提交，情况会变成什么样呢：



第一个事务执行200ms；

第二个事务执行120ms；

第三个事务执行80ms；

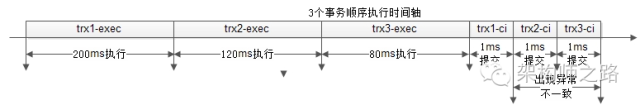
第一个事务执行1ms；

第二个事务执行1ms；

第三个事务执行1ms；

**那在什么时候系统出现问题，会出现不一致呢？**

问题的答案与之前相同：第一个事务成功提交之后，最后一个事务成功提交之前，如果出现问题（例如服务器重启，数据库异常等），都可能导致数据不一致。



**这个变化的意义是什么呢？**

方案一总执行时间是303ms，最后202ms内出现异常都可能导致不一致；

方案二总执行时间也是303ms，但最后2ms内出现异常才会导致不一致；

虽然没有彻底解决数据的一致性问题，但不一致出现的概率大大降低了！

事务提交后置降低了数据不一致的出现概率，**会带来什么副作用呢？**

回答：事务提交时会释放数据库的连接，第一种方案，第一个库事务提交，数据库连接就释放了，后置事务提交的方案，所有库的连接，要等到所有事务执行完才释放。这就意味着，**数据库连接占用的时间增长了，系统整体的吞吐量降低了。**

### ****四、总结****

trx1.exec();

trx1.commit();

trx2.exec();

trx2.commit();

trx3.exec();

trx3.commit();

优化为：

trx1.exec();

trx2.exec();

trx3.exec();

trx1.commit();

trx2.commit();

trx3.commit();

这个小小的改动（改动成本极低），不能彻底解决多库分布式事务数据一致性问题，但能大大降低数据不一致的概率，带来的副作用是数据库连接占用时间会增长，吞吐量会降低。对于一致性与吞吐量的折衷，还需要业务架构师谨慎权衡折衷。

如果有收获，帮忙随手转发哟。

==【完】==

回【检索】[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)

回【钱包】[入驻微信钱包的技术优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959423&idx=1&sn=19d27145beab5b0238280e2e3804e41b&scene=21" \l "wechat_redirect" \t "_blank)

回【消息】[58到家通用实时消息平台架构细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "wechat_redirect" \t "_blank)

回【秒杀】[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959391&idx=1&sn=fb28fd5e5f0895ddb167406d8a735548&scene=21" \l "wechat_redirect" \t "_blank)

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

回【延时】[mysql并行复制降低主从同步延时的思路与启示](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959461&idx=1&sn=c4c9317f55075541be00d59039b88470&scene=21" \l "wechat_redirect" \t "_blank)

回【推荐】[58同城推荐系统架构设计与实现](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959464&idx=1&sn=81c29bf2e237499be07deba826f73c39&scene=21" \l "wechat_redirect" \t "_blank)

回【单点】[单点系统架构的可用性与性能优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959480&idx=1&sn=337bd74410a6bef616128fd17abd08a8&scene=21" \l "wechat_redirect" \t "_blank)

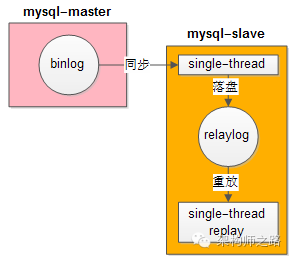
## mysql并行复制降低主从同步延时的思路与启示

原创 2016-06-08 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959461&idx=1&sn=c4c9317f55075541be00d59039b88470&scene=21" \l "#)

### ****一、缘起****

**mysql主从复制，读写分离**是互联网用的非常多的mysql架构，主从复制最令人**诟病**的地方就是，在数据量较大并发量较大的场景下，主从延时会比较严重。

**为什么mysql主从延时这么大？**



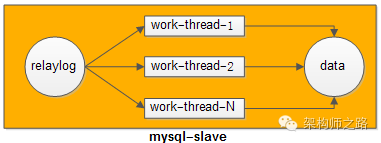
回答：从库使用【单线程】重放relaylog。

**优化思路是什么？**

回答：使用单线程重放relaylog使得同步时间会比较久，导致主从延时很长，优化思路不难想到，可以【多线程并行】重放relaylog来缩短同步时间。

mysql如何“多线程并行”来重放relaylog，是本文要分享的主要内容。

### ****二、如何多线程并行重放relaylog****



通过多个线程来并行重放relaylog是一个很好缩短同步时间的思路，但实施之前要解决这样一个问题：

**如何来分割relaylog，才能够让多个work-thread并行操作数据data时，使得data保证一致性？**

首先，【随机的分配relaylog肯定是不行的】，假设relaylog中有这样三条串行的修改记录：

update account set money=100 where uid=58;

update account set money=150 where uid=58;

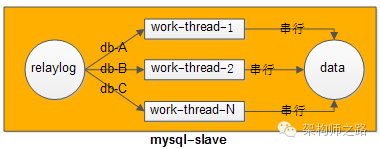
update account set money=200 where uid=58;

**串行执行**：肯定能保证与主库的执行序列一致，最后得到money=200

**随机分配并行执行**：3个工作线程并发执行这3个语句，谁最后执行成功是不确定的，故得到的数据可能与主库不同

好，对于这个问题，可以用什么样的思路来解决呢（大伙怎么想，mysql团队其实也就是这么想的）

**【方法一：相同库上的写操作，用相同的work-thread来重放relaylog；不同库上的写操作，可以用多个work-thread并发来重放relaylog】**



**如何做到呢？**

**回答**：不难，hash(db-name) % thread-num，库名hash之后再模上线程数，就能够做到。

**存在的不足？**

很多公司对mysql的使用是“单库多表”，如果是这样的话，仍然是同一个work-thread在串行执行，还是不能提高relaylog的重放速度。

**优化方案**：将“单库多表”的模式升级为“多库多表”的模式。

其实，数据量大并发量大的互联网业务场景，“多库”模式还具备着其他很多优势，例如：

（1）非常方便的实例扩展：dba很容易将不同的库扩展到不同的实例上

（2）按照业务进行库隔离：业务解耦，进行业务隔离，减少耦合与相互影响

（3）…

对于架构师进行架构设计的启示是：**使用多库的方式设计db架构，能够降低主从同步的延时。**

**新的想法：“单库多表”的场景，还有并行执行优化余地么？**

仔细回顾和思考，即使只有一个库，数据的修改和事务的执行在主库上也是并行操作的，既然在主库上可以并行操作，在从库上为啥就不能并行操作，而要按照库来串行执行呢（表示不服）？

**新的思路：**将主库上同时并行执行的事务，分为一组，编一个号，这些事务在从库上的回放可以并行执行（事务在主库上的执行都进入到prepare阶段，说明事务之间没有冲突，否则就不可能提交），没错，mysql正是这么做的。

**【方法二：基于GTID的并行复制】**

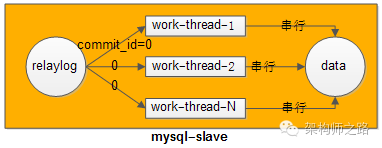
新版的mysql，将组提交的信息存放在GTID中，使用mysqlbinlog工具，可以看到组提交内部的信息：

20160607 23:22 server\_id 58 XXX GTID last\_committed=0 sequence\_numer=1

20160607 23:22 server\_id 58 XXX GTID last\_committed=0 sequence\_numer=2

20160607 23:22 server\_id 58 XXX GTID last\_committed=0 sequence\_numer=3

20160607 23:22 server\_id 58 XXX GTID last\_committed=0 sequence\_numer=4



和原来的日志相比，多了last\_committed和sequence\_number。

last\_committed表示事务提交时，上次事务提交的编号，如果具备相同的last\_committed，说明它们在一个组内，可以并发回放执行。

### ****三、结尾****

从mysql并行复制缩短主从同步时延的思想可以看到，架构的思路是相同的：

（1）**多线程**是一种常见的缩短执行时间的方法

（2）多线程并发分派任务时必须保证**幂等性**：mysql的演进思路，提供了“按照库幂等”，“按照commit\_id幂等”两种方式，思路大伙可以借鉴

另，mysql在并行复制上的逐步优化演进：

**mysql5.5** -> 不支持并行复制，对大伙的启示：升级mysql吧

**mysql5.6** -> 按照库并行复制，对大伙的启示：使用“多库”架构吧

**mysql5.7** -> 按照GTID并行复制

我不是mysql的开发人员，也不是专业的dba，本文仅为一个思路的分享，希望大伙有收获，如果不对也欢迎随时指出。

==【完】==

回【检索】[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)

回【钱包】[入驻微信钱包的技术优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959423&idx=1&sn=19d27145beab5b0238280e2e3804e41b&scene=21" \l "wechat_redirect" \t "_blank)

回【消息】[58到家通用实时消息平台架构细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "wechat_redirect" \t "_blank)

回【秒杀】[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959391&idx=1&sn=fb28fd5e5f0895ddb167406d8a735548&scene=21" \l "wechat_redirect" \t "_blank)

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

## 互联网公司为啥不使用mysql分区表？（一分钟系列）

原创 2016-09-22 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959604&idx=1&sn=9f3e21f5ff99abc5a16038a77f31843e&scene=21" \l "#)

**缘起**：有个朋友问我分区表在58的应用，我回答不出来，在我印象中，百度、58都没有听说有分区表相关的应用，业内进行一些技术交流的时候也更多的是自己分库分表，而不是使用分区表。于是去网上查了一下，并询问了58到家的DBA专家，将自己收到的信息沉淀下来，share给大伙。

**解决什么问题？**

回答：当mysql单表的数据库过大时，数据库的访问速度会下降，“数据量大”问题的常见解决方案是“水平切分”。

**mysql常见的水平切分方式有哪些？**

回答：分库分表，分区表

**什么是mysql的分库分表？**

回答：把一个很大的库（表）的数据分到几个库（表）中，每个库（表）的结构都相同，但他们可能分布在不同的mysql实例，甚至不同的物理机器上，以达到降低单库（表）数据量，提高访问性能的目的。

分库分表往往是业务层实施的，分库分表后，为了满足某些特定业务功能，往往需要rd修改代码。

**什么是mysql的分区表？**

回答：所有数据还在一个表中，但物理存储根据一定的规则放在不同的文件中。这个是mysql支持的功能，业务rd代码无需改动。

**看上去分区表很帅气，为什么大部分互联网还是更多的选择自己分库分表来水平扩展咧？**

回答：

1）分区表，分区键设计不太灵活，如果不走分区键，很容易出现全表锁

2）一旦数据量并发量上来，如果在分区表实施关联，就是一个灾难

3）自己分库分表，自己掌控业务场景与访问模式，可控。分区表，研发写了一个sql，都不确定mysql是怎么玩的，不太可控

4）运维的坑，嘿嘿

5）…

文章很短，一分钟搞定，希望大家有收获，有任何疑问欢迎提出，我不懂的再去问DBA专家。如果大家有分区表的应用，踩了什么坑，亦可回复，我下一篇文章share出来。

埋坑：如何来进行水平切分，分库分表？如果大伙感兴趣，后续和大家聊更多的数据库架构。

想了解什么技术，大家做主。

==【完】==

回【容量】[互联网架构，如何进行容量设计？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959542&idx=1&sn=2494bbea9a855e0e1c3ccd6d2562a600&scene=21" \l "wechat_redirect" \t "_blank)

回【事务】[多库多事务降低数据不一致概率](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959486&idx=1&sn=cd3e6b27af159f4eabd54f148eec638a&scene=21" \l "wechat_redirect" \t "_blank)

回【延时】[mysql并行复制降低主从同步延时的思路与启示](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959461&idx=1&sn=c4c9317f55075541be00d59039b88470&scene=21" \l "wechat_redirect" \t "_blank)

回【主从】[DB主从一致性架构优化4种方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959442&idx=1&sn=feb8ff75385d8031386e120ef3535329&scene=21" \l "wechat_redirect" \t "_blank)

回【一致】[缓存与数据库一致性保证](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404202261&idx=1&sn=1b8254ba5013952923bdc21e0579108e&scene=21" \l "wechat_redirect" \t "_blank)

回【缓存】[缓存架构设计细节二三事](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404087915&idx=1&sn=075664193f334874a3fc87fd4f712ebc&scene=21" \l "wechat_redirect" \t "_blank)

回【冗余】[细聊冗余表数据一致性](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403963671&idx=1&sn=51a2d2fd70212451cd5f22bbe2c6f8d6&scene=21" \l "wechat_redirect" \t "_blank)

回【CAS】[巧用CAS解决数据一致性问题](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400556481&idx=1&sn=42bc187b45e4e53aece4b8c1f43e8019&scene=21" \l "wechat_redirect" \t "_blank)

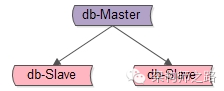
## 即使删了全库，保证半小时恢复

原创 2016-11-23 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959694&idx=1&sn=4ac53b797fca64229901373e793f860a&chksm=bd2d04528a5a8d44618333de549094569f9ec58833ab64d081bc826b9361f587aca2972fc4c6&scene=21" \l "#)

近期一篇《[就这样把根目录删了！！！](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959681&idx=1&sn=40b433673fcdb5705727b7d2c1668875&chksm=bd2d045d8a5a8d4b6a6e46cb849331cdaef830a7281864cc8063ba31905c6513b14c42aaa7cf&scene=21" \l "wechat_redirect" \t "_blank)》引发了广泛的讨论，《[如何防止根目录被删](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959684&idx=1&sn=779fdcd58c1081564e558eff38e575c7&chksm=bd2d04588a5a8d4e030ed3a511f279dbedbb2d0682ab6379b6a68e3466bac0b17f85f4253167&scene=21" \l "wechat_redirect" \t "_blank)》汇总了7种防删方案。还有同学评论中反馈“不小心把库删了”，如何快速恢复删掉的数据库，是今天要讨论的话题。

**【高可用数据库架构】**

一般来说数据库集群会是主从架构：



或者主主架构：

http://mmbiz.qpic.cn/mmbiz_png/YrezxckhYOzTmFaR4BiaLtkmOHtlC0RDD3dQJT1Mia2JN4558TZIClSeld2UqcGZicicjjFwfARhxfC02Zw4EFLDKw/640?wx_fmt=png&wxfrom=5&wx_lazy=1

如果此时主库宕机，可以：

（1）**一个从库顶上，重建集群**

（2）**流量迁移到另一个主库**

来保证数据的安全性与服务的可用性。

但是，如果人为不小心执行了“删全库”操作，命令会同步给其他从（主）库，导致所有库上的数据全部丢失，这下怎么办呢？

可以问问自己，当这种情况发生的时候：

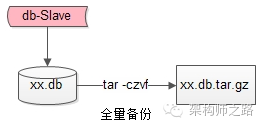
（1）**能不能恢复数据？**（应该没有公司不能）

（2）**多久能够恢复数据？**

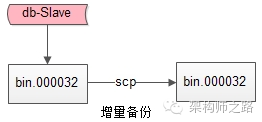
保证数据的安全性是DBA第一要务。

**【全量备份+增量备份】**

常见的数据库安全性策略是：全量备份+增量备份。



**全量备份**：定期（例如一个月）将库文件全量备份



**增量备份**：定期（例如每天）将binlog增量备份

如果不小心误删了全库，可以**这么恢复**：

（1）将最近一次全量备份的全库找到，拷贝回来（文件一般比较大），解压，应用

（2）将最近一次全量备份后，每一天的增量binlog找到，拷贝回来（文件较多），依次重放

（3）将最近一次增量备份后，到执行“删全库”之前的binlog找到，重放

恢复完毕。

为了保证方案的可靠性，建议定期进行恢复演练。

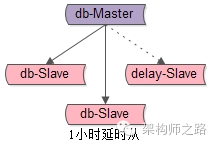
**方案优点**：能够找回数据

**方案缺点**：恢复时间非常长

有没有更优，更快恢复的方案呢？

**【1小时延时从】**

使用1小时延时从库，可大大加速“删全库”恢复时间。



**什么是1小时延时从？**

如图所示，增加一个从库，这个从库不是实时与主库保持同步的，而是每隔1个小时同步一次主库，同步完之后立马断开1小时，这个从库会与主库保持1个小时的数据差距。

当“删全库”事故发生时，只需要：

（1）应用1小时延时从

（2）将1小时延时从最近一次同步时间到，将执行“删全库”之前的binlog找到，重放

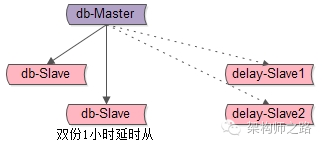
快速恢复完毕。

**方案优点**：能够快速找回数据

**潜在不足**：万一，万一，万一，1小时延时从正在连上主库进行同步的一小段时间内，发生了“删全库”事故，那怎么办咧？

**【双份1小时延时从】**

使用双份1小时延时从库，可以避免上述“万一，万一，万一”的事故发生。



**什么是双份1小时延时从？**

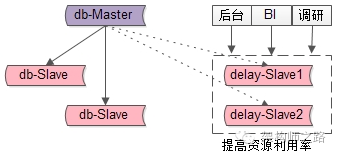
如图所示，两个1小时延时从，他们连主库同步数据的时间“岔开半小时”。

这样，即使一个延时从连上主库进行同步的一小段时间内，发生了“删全库”事故，依然有另一个延时从保有半小时之前的数据，可以实施快速恢复。

**方案优点**：没有万一，都能快速恢复数据

**潜在不足**：资源利用率有点低，为了保证数据的安全性，多了2台延时从，降低了从库利用率

**【提高从库效率】**



1小时延时从也不是完全没有用，对于一些“允许延时”的业务，可以使用1小时延时从，例如：

（1）运营后台，产品后台

（2）BI进行数据同步

（3）研发进行数据抽样，调研

但需要注意的是，毕竟这是从库，只能够提供“只读”服务哟。

**【总结】**

**保证数据的安全性是DBA第一要务**，需要进行：

（1）全量备份+增量备份，并定期进行恢复演练，但该方案恢复时间较久，对系统可用性影响大

（2）1小时延时从，双份1小时延时从能极大加速数据库恢复时间

（3）个人建议1小时延时从足够，后台只读服务可以连1小时延时从，提高资源利用率

大伙还有没有什么好方案？

===【完，有收获？求**帮转**！】===

## 啥，又要为表增加一列属性？

原创 2016-12-14 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959762&idx=1&sn=f7d8d7648416a0157af6ac61a6b555c8&chksm=bd2d040e8a5a8d181ec1baa2e96982991ddfc218bb6838f38da61b651e4b8de5f0bbbd94f814&scene=21" \l "#)

**需求缘起**

产品第一版：用户有用户名、密码、昵称等三个属性,对应表设计：

user(uid, name, passwd, nick)

第二版，产品经理增加了年龄，性别两个属性,表结构可能要变成：

user(uid, name, passwd, nick, age, sex)

假设数据量和并发量比较大，怎么变？

（1）alter table add column？不太可行，锁表时间长

（2）新表+触发器？如果数据量太大，新表不一定装得下，何况触发器对数据库性能的影响比较高

（3）让dba来搞？新表，迁移数据，一致性校验，rename？dba真苦逼

今天分享2个**列扩展性设计**上几个小技巧，只占大伙1分钟（下班太晚的话，只能写一分钟系列=\_=）

**方案一：版本号+通用列**

以上面的用户表为例，假设只有uid和name上有查询需求，表可以设计为

**user(uid, name, version, ext)**

（1）uid和name有查询需求，必须设计为单独的列并建立索引

（2）version是版本号字段，它对ext进行了版本解释

（3）ext采用可扩展的字符串协议载体，承载**被查询**的属性

例如，最开始上线的时候，版本为0，此时只有passwd和nick两个属性，那么数据为：



当产品经理需要扩展属性时，新数据将版本变为1，此时新增了age和sex两个数据，数据变为：

**优点**：

（1）可以随时动态扩展属性

（2）新旧两种数据可以同时存在

（3）迁移数据方便，写个小程序将旧版本ext的改为新版本的ext，并修改version

**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

**改进**：

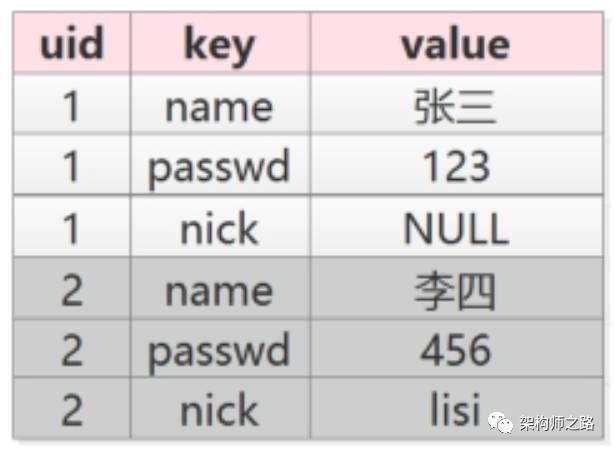
（1）如果ext里的属性有索引需求，可能Nosql的如MongoDB会更适合

**方案二：通过扩展行的方式来扩展属性**

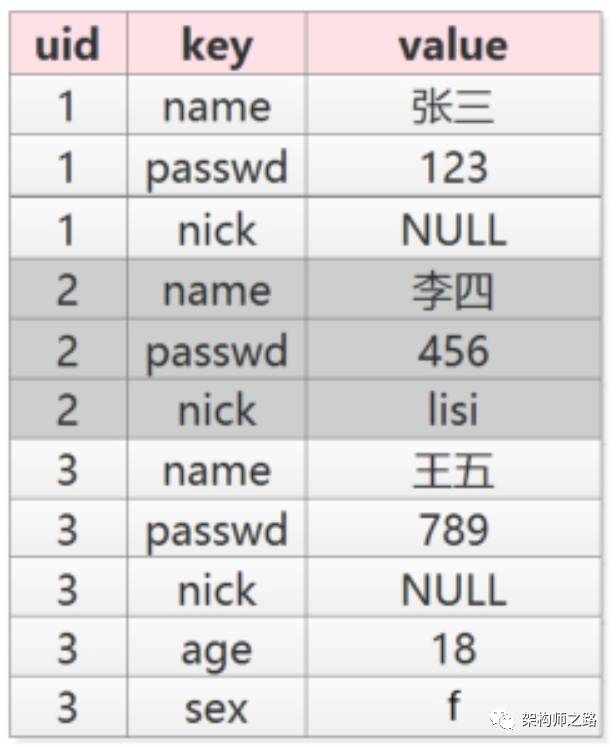
以上面的用户表为例，可以设计为

**user(uid, key, value)**

初期有name, passwd, nick三个属性，那么数据为：



未来扩展了age和sex两个属性，数据变为：

**优点**：

（1）可以随时动态扩展属性

（2）新旧两种数据可以同时存在

（3）迁移数据方便，写个小程序可以将新增的属性加上

（4）各个属性上都可以查询

**不足**：

（1）key值有大量冗余，建议key短一些

（2）本来一条记录很多属性，会变成多条记录，行数会增加很多

**总结**

可以通过“version+ext”或者“key+value”的方式来满足产品新增列的需求，希望没有浪费你这一分钟，有收获就好。

 ==【完】==

欢迎大家提更好的方法，下期总结分享给更多人。

相关阅读：[互联网公司为啥不使用mysql分区表？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959604&idx=1&sn=9f3e21f5ff99abc5a16038a77f31843e&scene=21" \l "wechat_redirect" \t "_blank)

## 这才是真正的表扩展方案

原创 2016-12-15 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959765&idx=1&sn=b9916aa95c320e41035977e0a8098ca6&chksm=bd2d04098a5a8d1f3af38f658c05002151e621170949d2e3bb5b1bceea55c64b0477dba4c647&scene=21" \l "#)

事情变得有意思了，上一篇花1小时撰写的“一分钟”文章，又引起了广泛的讨论，说明相关的技术大家感兴趣，挺好。第一次一篇技术文章的评论量过100，才知道原来“评论精选”还有100上限，甚为欣慰（虽然是以一种自己不愿看到的方式）。

《[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)》的方案颇有争议：

（1）版本号version + 扩展字段ext

（2）用增加列的key+value方式扩充属性

~~有些评论，只能说“~~**~~所谓夏虫，何以语冰~~**~~”（作者要谦和，请删除）。~~因自己时间仓促，有些地方没有交代清楚，对不起大伙，实在抱歉。大部分评论还是在进行技术讨论，故今天再熬夜补充说明一下。

**零、缘起**

讨论问题域：

（1）**数据量大、并发量高场景**，在线数据库属性扩展

（2）数据库**表结构扩展性设计**

**一、哪些方案一定是不行的**

（1）alter table add column

要坚持这个方案的，也不多解释了，大数据高并发情况下，一定不可行

（2）通过增加表的方式扩展，通过外键join来查询

大数据高并发情况下，join性能较差，一定不可行

（3）通过增加表的方式扩展，通过视图来对外

一定不可行。大数据高并发情况下，互联网不怎么使用视图，至少58禁止使用视图

（4）必须遵循“第x范式”的方案

一定不可行。互联网的**主要矛盾之一是吞吐量**，为了保证吞吐量甚至可能牺牲一些事务性和一致性，通过反范式的方式来确保吞吐量的设计是很常见的，例如：冗余数据。互联网的**主要矛盾之二是可用性**，为了保证可用性，常见的技术方案也是数据冗余。在互联网数据库架构设计中，第x范式真的没有这么重要

（5）打产品经理

朋友，这是段子么，这一定不可行

**二、哪些方案可行，但文章未提及**

（1）提前预留一些reserved字段

这个是可以的。但如果预留过多，会造成空间浪费，预留过少，不一定达得到扩展效果。

（2）通过增加表的方式扩展列，上游通过service来屏蔽底层的细节

这个也是可以的。Jeff同学提到的UserExt(uid, newCol1, newCol2)就是这样的方案（但join连表和视图是不行的）

**三、哪些读者没有仔细看文章**

（1）version+ext太弱了，ext不支持索引

回复：属于没有仔细看文章，文章也提了如果有强需求索引可以使用MongoDB，它就是使用的json存储（评论中有不少朋友提到，还有其他数据库支持json检索）

（2）第二种key+value方案不支持索引

回复：uid可以索引

**四、key+value方式使用场景**

服务端，wordpress，EAV，配置，统计项等都经常使用这个方案。

客户端（APP或者PC），保存个人信息也经常使用这个方案。

**今天的重点**

以楼主性格，本不会进行“解释”，上文解释这般，说明这一次，楼主真的认真了。对于技术，认真是好事，认真的男人最可爱（打住，我要吐了）。好了，下面的内容才是今天的重点。

**五、在线表结构变更**

在《[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)》文章的开头，已经说明常见“**新表+触发器+迁移数据+rename**”方案（pt-online-schema-change），这是业内非常成熟的扩展列的方案（以为大伙都熟悉，没有展开讲，只重点讲了两种新方案，这可能是导致被喷得厉害的源头），今天补充说一下。

以**user(uid, name, passwd)**

扩展到**user(uid, name, passwd, age, sex)**为例

基本原理是：

（1）先创建一个扩充字段后的新表user\_new(uid, name, passwd, age, sex)

（2）在原表user上创建三个触发器，对原表user进行的所有insert/delete/update操作，都会对新表user\_new进行相同的操作

（3）分批将原表user中的数据insert到新表user\_new，直至数据迁移完成

（4）删掉触发器，把原表移走（默认是drop掉）

（5）把新表user\_new重命名（rename）成原表user

扩充字段完成。

**优点**：整个过程不需要锁表，可以持续对外提供服务

操作过程中需要**注意**：

（1）变更过程中，最重要的是冲突的处理，一条原则，以触发器的新数据为准，这就要求被迁移的表必须有主键（这个要求基本都满足）

（2）变更过程中，写操作需要建立触发器，所以如果原表已经有很多触发器，方案就不行（互联网大数据高并发的在线业务，一般都禁止使用触发器）

（3）触发器的建立，会影响原表的性能，所以这个操作建议在流量低峰期进行

**pt-online-schema-change**是DBA必备的利器，比较成熟，在互联网公司使用广泛。

楼主非专业的dba，上面的过程有说的不对的地方，欢迎指出。要了解更详细的细节，可以百度一下。有更好的方法，也欢迎讨论，后续会梳理汇总share给更多的朋友。

**六、结束**

欢迎用批判的眼光看问题，欢迎任何友善的技术讨论，不太欢迎“纯属误导”“非常蠢的方案”这样的评论（但我还是会加精选，任何人都有发声的权利）。

借评论中@张九云 朋友的一句话“不要以为自己见过的就是全世界，**任何方案都有使用场景，一切都是tradeoff**”作为今天的结尾，谢谢大家的支持，感谢大家。

==【完】==

如果有收获，帮忙转一下。

## 一分钟掌握数据库垂直拆分

原创 2016-12-20 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959773&idx=1&sn=7e4ad0dcd050f6662dfaf39d9de36f2c&chksm=bd2d04018a5a8d17b92098b4840aac23982e32d179cdd957e4c55011f6a08f6bd31f9ba5cfee&scene=21" \l "#)

**一、缘起**

当数据库的数据量非常大时，**水平切分**和**垂直拆分**是两种常见的降低数据库大小，提升性能的方法。假设有用户表：

user(

uid bigint,

name varchar(16),

pass varchar(16),

age int,

sex tinyint,

flag tinyint,

sign varchar(64),

intro varchar(256)

…);

**水平切分**是指，以某个字段为依据（例如uid），按照一定规则（例如取模），将一个库（表）上的数据拆分到多个库（表）上，以降低单库（表）大小，达到提升性能的目的的方法，水平切分后，各个库（表）的特点是：

（1）每个库（表）的**结构**都一样

（2）每个库（表）的**数据**都不一样，没有交集

（3）所有库（表）的并集是全量数据

**二、什么是垂直拆分**

**垂直拆分**是指，将一个属性较多，一行数据较大的表，将不同的属性拆分到不同的表中，以降低单库（表）大小，达到提升性能的目的的方法，垂直切分后，各个库（表）的特点是：

（1）每个库（表）的**结构**都不一样

（2）一般来说，每个库（表）的**属性**至少有一列交集，一般是主键

（3）所有库（表）的并集是全量数据

还是以上文提到的用户表为例，如果要垂直拆分，可能拆分结果会是这样的：

**user\_base**(

uid bigint,

name varchar(16),

pass varchar(16),

age int,

sex tinyint,

flag tinyint,

…);

**user\_ext**(

uid bigint,

sign varchar(64),

intro varchar(256)

…);

**三、垂直切分的依据是什么**

当一个表属性很多时，如何来进行垂直拆分呢？如果没有特殊情况，拆分依据主要有几点：

（1）将长度较短，访问频率较高的属性尽量放在一个表里，这个表暂且称为**主表**

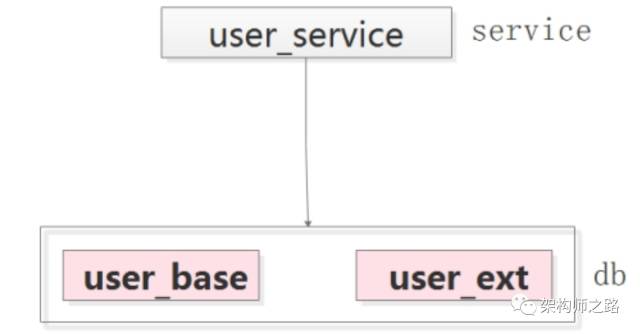
（2）将字段较长，访问频率较低的属性尽量放在一个表里，这个表暂且称为**扩展表**

如果1和2都满足，还可以考虑第三点：

（3）经常一起访问的属性，也可以放在一个表里

优先考虑1和2，第3点不是必须。另，如果实在属性过多，主表和扩展表都可以有多个。

一般来说，**数据量并发量比较大**时，数据库的上层都会有一个**服务层**。需要注意的是，当应用方需要同时访问主表和扩展表中的属性时，服务层**不要使用**join来连表访问，而应该分两次进行查询：



原因是，大数据高并发互联网场景下，一般来说，**吞吐量和扩展性**是主要矛盾：

（1）join更消损耗数据库性能

（2）join会让base表和ext表**耦合在一起**（必须在一个数据库实例上），不利于数据量大时拆分到不同的数据库实例上（机器上）。毕竟减少数据量，提升性能才是垂直拆分的初衷。

**四、为什么要这么这么拆分**

为何要将字段短，访问频率高的属性放到一个表内？为何这么垂直拆分可以提升性能？因为：

（1）数据库有自己的内存buffer，会将磁盘上的数据load到内存buffer里（暂且理解为进程内缓存吧）

（2）内存buffer缓存数据是**以row为单位**的

（3）在内存有限的情况下，在数据库内存buffer里缓存短row，就能缓存更多的数据

（4）在数据库内存buffer里缓存访问频率高的row，就能提升缓存命中率，减少磁盘的访问

举个例子就很好理解了：

假设数据库内存buffer为1G，未拆分的user表1行数据大小为1k，那么只能缓存100w行数据。

如果垂直拆分成user\_base和user\_ext，其中：

（1）user\_base访问频率高（例如uid, name, passwd, 以及一些flag等），一行大小为0.1k

（2）user\_ext访问频率低（例如签名, 个人介绍等），一行大小为0.9k

那边内存buffer就就能缓存近乎1000w行user\_base的记录，访问磁盘的概率会大大降低，数据库访问的时延会大大降低，吞吐量会大大增加。

**五、总结**

（1）**水平拆分**和**垂直拆分**都是降低数据量大小，提升数据库性能的常见手段

（2）流量大，数据量大时，**数据访问要有service层**，并且service层不要通过join来获取主表和扩展表的属性

（3）**垂直拆分的依据**，尽量把长度较短，访问频率较高的属性放在主表里

希望没有浪费你这一分钟，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)

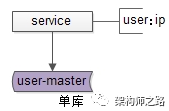
[这才是真正的表扩展方案](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959765&idx=1&sn=b9916aa95c320e41035977e0a8098ca6&chksm=bd2d04098a5a8d1f3af38f658c05002151e621170949d2e3bb5b1bceea55c64b0477dba4c647&scene=21" \l "wechat_redirect" \t "_blank)

## 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 "#)

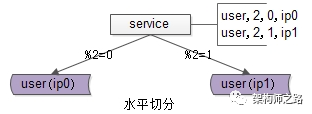
**一、缘起**

（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）。

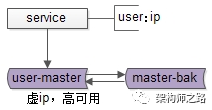
（2）随着数据量的增大，数据要进行**水平切分**，分库后将数据分布到不同的数据库实例（甚至物理机器）上，以达到降低数据量，增强性能的扩容目的：



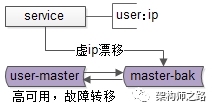
如上图：用户库user分布在两个实例上，ip0和ip1，服务层通过用户标识uid取模的方式进行寻库路由，模2余0的访问ip0上的user库，模2余1的访问ip1上的user库。

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：



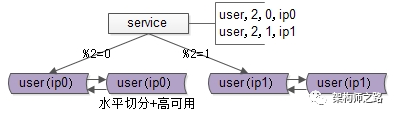
如上图：两个相互同步的主库使用相同的虚ip。



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**：

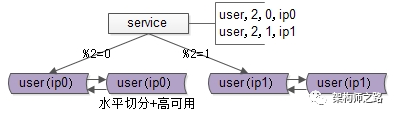
（1）停服务，不高可用

（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

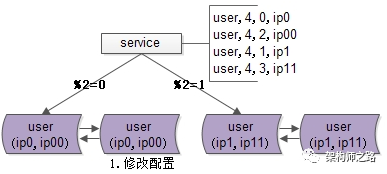
有没有更平滑的方案呢？

**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

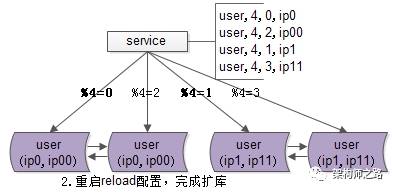
b）修改服务的配置（不管是在配置文件里，还是在配置中心），将2个库的数据库配置，改为4个库的数据库配置，**修改的时候要注意旧库与辛苦的映射关系**：

%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。

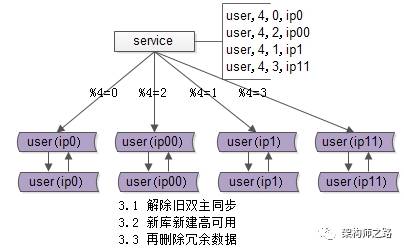
整个过程可以逐步重启，**对服务的正确性和可用性完全没有影响**：

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

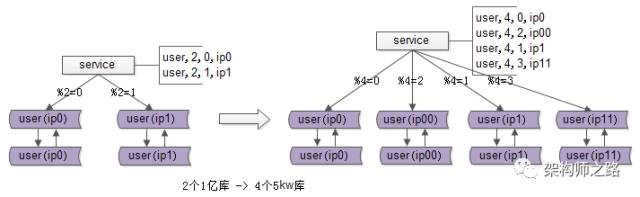
b）解除旧的双主同步，让成对库的数据不再同步增加

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)

## 互联网架构为什么要做服务化？

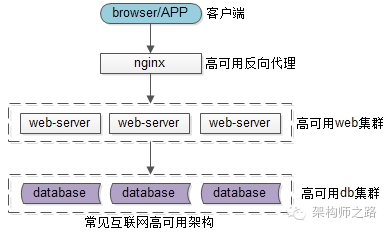
原创 2016-08-10 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959519&idx=1&sn=065074b135fc9cb243abe897261e1a72&scene=21" \l "#)

近期参加一些业界的技术大会，“微服务架构”的话题非常之火，也在一些场合聊过服务化架构实践，最近几期文章期望用通俗易懂的语言聊聊了个人对服务化以及微服务架构的理解，希望能给大伙一些启示。如果有遗漏，也欢迎大家补充。

### ****一、互联网高可用架构，为什么要服务化？****

**【服务化之前高可用架构】**

在服务化之前，互联网的高可用架构大致是这样一个架构：

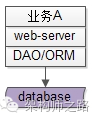


（1）用户端是浏览器browser，APP客户端

（2）后端入口是高可用的nginx集群，用于做反向代理

（3）中间核心是高可用的web-server集群，研发工程师主要编码工作就是在这一层

（4）后端存储是高可用的db集群，数据存储在这一层

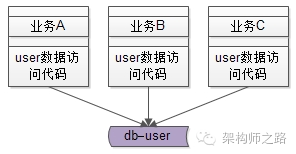


更典型的，web-server层是通过DAO/ORM等技术来访问数据库的。

可以看到，最初都是没有服务层的，此时架构会碰到一些什么痛点呢？

**【架构痛点一：代码到处拷贝】**

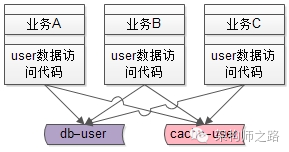
举一个最常见的业务的例子->用户数据的访问，绝大部分公司都有一个数据库存储用户数据，各个业务都有访问用户数据的需求：



在有用户服务之前，各个业务线都是自己通过DAO写SQL访问user库来存取用户数据，这无形中就导致了代码的拷贝。

**【架构痛点二：复杂性扩散】**

随着并发量的越来越高，用户数据的访问数据库成了瓶颈，需要加入缓存来降低数据库的读压力，于是架构中引入了缓存，由于没有统一的服务层，各个业务线都需要关注缓存的引入导致的复杂性：



对于用户数据的写请求，所有业务线都要升级代码：

（1）先淘汰cache

（2）再写数据

对于用户数据的读请求，所有业务线也都要升级代码：

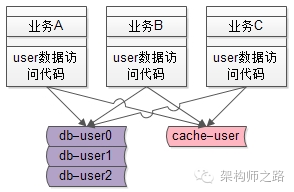
（1）先读cache，命中则返回

（2）没命中则读数据库

（3）再把数据放入cache

这个复杂性是典型的“业务无关”的复杂性，业务方需要被迫升级。

随着数据量的越来越大，数据库需要进行水平拆分，于是架构中又引入了分库分表，由于没有统一的服务层，各个业务线都需要关注分库分表的引入导致的复杂性：



这个复杂性也是典型的“业务无关”的复杂性，业务方需要被迫升级。

包括bug的修改，发现一个bug，多个地方都需要修改。

**【架构痛点三：库的复用与耦合】**

服务化并不是唯一的解决上述两痛点的方法，抽象出统一的“库”是最先容易想到的解决：

（1）代码拷贝

（2）复杂性扩散

的方法。抽象出一个user.so，负责整个用户数据的存取，从而避免代码的拷贝。至于复杂性，也只有user.so这一个地方需要关注了。

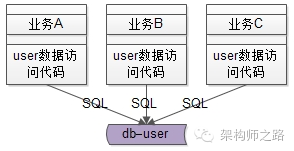
解决了旧的问题，会引入新的问题，库的版本维护与业务线之间代码的耦合：

业务线A将user.so由版本1升级至版本2，如果不兼容业务线B的代码，会导致B业务出现问题；

业务线A如果通知了业务线B升级，则是的业务线B会无故做一些“自身业务无关”的升级，非常郁闷。当然，如果各个业务线都是拷贝了一份代码则不存在这个问题。

**【架构痛点四：SQL质量得不到保障，业务相互影响】**

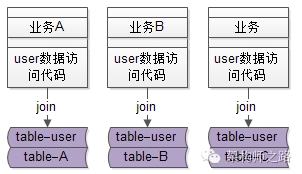
业务线通过DAO访问数据库：



本质上SQL语句还是各个业务线拼装的，资深的工程师写出高质量的SQL没啥问题，经验没有这么丰富的工程师可能会写出一些低效的SQL，假如业务线A写了一个全表扫描的SQL，导致数据库的CPU100%，影响的不只是一个业务线，而是所有的业务线都会受影响。

**【架构痛点五：疯狂的DB耦合】**

业务线不至访问user数据，还会结合自己的业务访问自己的数据：



典型的，通过join数据表来实现各自业务线的一些业务逻辑。

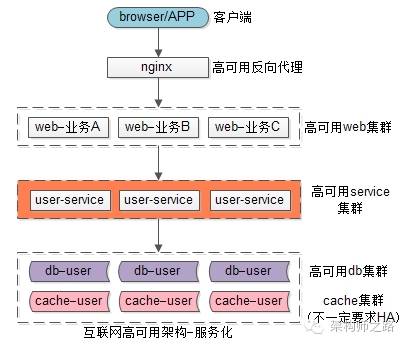
这样的话，业务线A的table-user与table-A耦合在了一起，业务线B的table-user与table-B耦合在了一起，业务线C的table-user与table-C耦合在了一起，结果就是：table-user，table-A，table-B，table-C都耦合在了一起。

随着数据量的越来越大，业务线ABC的数据库是无法垂直拆分开的，必须使用一个大库（疯了，一个大库300多个业务表 =\_=）。

**【架构痛点六：…】**

### ****二、服务化解决什么问题？****

为了解决上面的诸多问题，互联网高可用分层架构演进的过程中，引入了“服务层”。



以上文中的用户业务为例，引入了user-service，对业务线响应所用用户数据的存取。引入服务层有什么好处，解决什么问题呢？

**【好处一：调用方爽】**

有服务层之前：业务方访问用户数据，需要通过DAO拼装SQL访问

有服务层之后：业务方通过RPC访问用户数据，就像调用一个本地函数一样，非常之爽

User = UserService::GetUserById(uid);

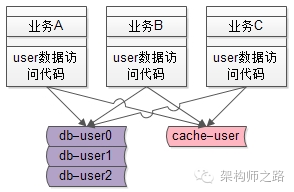
传入一个uid，得到一个User实体，就像调用本地函数一样，不需要关心序列化，网络传输，后端执行，网络传输，范序列化等复杂性。

**【好处二：复用性，防止代码拷贝】**

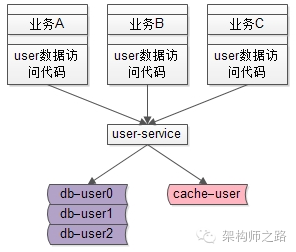
这个不展开叙述，所有user数据的存取，都通过user-service来进行，代码只此一份，不存在拷贝。

升级一处升级，bug修改一处修改。

**【好处三：专注性，屏蔽底层复杂度】**

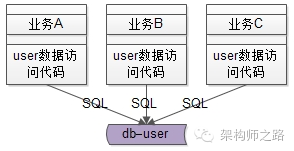


在没有服务层之前，所有业务线都需要关注缓存、分库分表这些细节。

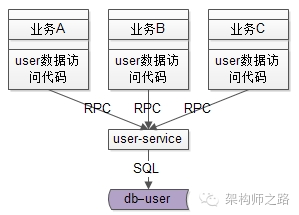


在有了服务层之后，只有服务层需要专注关注底层的复杂性了，向上游屏蔽了细节。

**【好处四：SQL质量得到保障】**

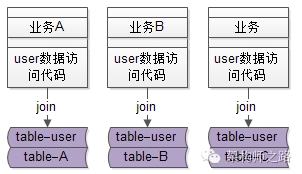


原来是业务向上游直接拼接SQL访问数据库。

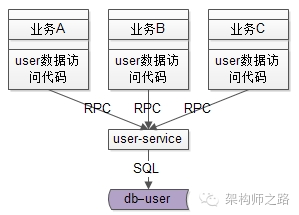


有了服务层之后，所有的SQL都是服务层提供的，业务线不能再为所欲为了。底层服务对于稳定性的要求更好的话，可以由更资深的工程师维护，而不是像原来SQL难以收口，难以控制。

**【好处五：数据库解耦】**



原来各个业务的数据库都混在一个大库里，相互join，难以拆分。



服务化之后，底层的数据库被隔离开了，可以很方便的拆分出来，进行扩容。

**【好处六：提供有限接口，无限性能】**

在服务化之前，各业务线上游想怎么操纵数据库都行，遇到了性能瓶颈，各业务线容易扯皮，相互推诿。

服务化之后，服务只提供有限的通用接口，理论上服务集群能够提供无限性能，性能出现瓶颈，服务层一处集中优化。

**【好处七：…】**

### ****三、其他****

服务化的其他好处，以及带来的问题，**欢迎大家畅所欲言，我下期再来补充**。

下期和大伙聊聊怎么“微”才是“微服务”，以及服务化的常见实践。

帮忙随手转发哟。

==【完】==

回【无锁】[如何实现超高并发的无锁缓存？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959502&idx=1&sn=7a1c7d81a8e030856fb920844dc571c6&scene=21" \l "wechat_redirect" \t "_blank)

回【消息】[58到家通用实时消息平台架构细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "wechat_redirect" \t "_blank)

回【机房】[从IDC到云端架构迁移之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959495&idx=1&sn=0f8789d6803f94e190ba2787389e6d1b&scene=21" \l "wechat_redirect" \t "_blank)

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

回【单点】[单点系统架构的可用性与性能优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959480&idx=1&sn=337bd74410a6bef616128fd17abd08a8&scene=21" \l "wechat_redirect" \t "_blank)

回【事务】[多库多事务降低数据不一致概率](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959486&idx=1&sn=cd3e6b27af159f4eabd54f148eec638a&scene=21" \l "wechat_redirect" \t "_blank)

## 微服务架构多“微”才合适？

原创 2016-08-11 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959530&idx=1&sn=ff84bd74745eee1577e7dfef8ce66bbe&scene=21" \l "#)

## 前情提要：[互联网架构为什么要做服务化？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959519&idx=1&sn=065074b135fc9cb243abe897261e1a72&scene=21" \l "wechat_redirect" \t "_blank)

### ****一、互联网架构为什么要进行服务化-总结****

上一篇和大伙交流了一下，随着数据量、并发量、业务复杂度的增长，互联网架构会出现以下问题：

（1）代码到处拷贝

（2）底层复杂性扩散

（3）基础库（so/jar/dll）耦合

（4）SQL质量得不到保障，业务相互影响

（5）数据库耦合

“服务化”是一个很好的解决上述痛点的方案。

不少评论也提出了不少有建设性的观点，汇总出来分享给大伙：

@田卫 同学提到：

**服务化之后，可能会引发分布式事务的问题**，“没人愿意引入分布式事务，当基于业务水平拆分的时候，要业务专家介入，合理拆分服务化，以后就服务内高内聚，事务可以保证，对于夸服务调用，通过补偿等手段，只要最终一致性就行，毕竟连现在的银行转账都不是强一致性。”

如@田卫所说，分布式事务是业界没有彻底解决的难题，任何架构设计都是一个折衷，吞吐量？时延？一致性？哪个是主要矛盾，优先解决哪个问题。大数据、高并发、业务复杂性是主要矛盾的时候，或许“最终一致性”是一个替代“事务”更好的，或者说业务能够接受的方案。

@侯滇滇 同学提到：

多了一层服务层，**架构实际上是更复杂了**，需要引入一系列机制对服务进行管理，RPC服务化中需要注意：

（1）RPC服务超时，服务调用者应有一些应对策略，比如重发

（2）关键服务例如支付，要注意幂等性，因为重发会导致重复操作

（3）多服务要考虑并发操作，相当单服务的锁机制比如JAVA中的synchronized

@黄明 同学提到：

服务化之后，随着规模的扩大，**一定要考虑“服务治理”**，否则服务之间的依赖关系会乱成麻

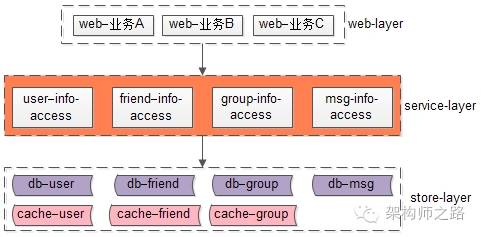
### ****二、互联网微服务架构多“微”才适合****

大家也都认可，随着数据量、流量、业务复杂度的提升，服务化架构是架构演进中的必由之路，今天要讨论的话题是：微服务架构多“微”才合适？

**【粗粒度：一个服务层】**



最粗犷的玩法，所有基础数据的访问，都通过一个service访问，在业务不是特别复杂的时候还好，一旦业务变复杂了，这个service层会变得非常重，成为耦合点之一，**以微信场景为例**，假设有一个通用的服务层来访问基础数据，这个服务层可能是这样的：

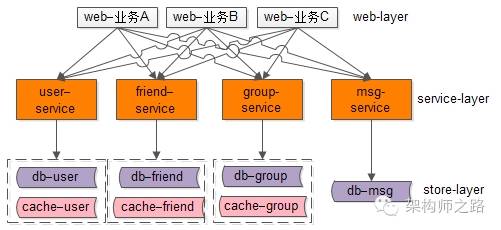


有一个统一的service层，用户信息，好友信息，群组信息，消息信息都通过这个service层来走。

细节：微信单对单消息是一个写多读少的业务，故没有缓存。

**【一个子业务一个service】**

如果所有的信息存储都在一个service里，那么一个地方出bug，就将影响整个业务，所以更合理的做法是在服务层进行细分，架构如何细分？垂直拆分是个好的方案，将子业务一个个拆出来，那么微信的服务化架构或许会变成这个样子：



（1）**用户**相关的子业务有user-service

（2）**好友**相关的子业务有friend-service

（3）**群组**相关的子业务有group-service

（4）**消息**相关的子业务有msg-service

这样的话，一个service出问题也不会影响其他service，同时数据层也按照业务垂直拆分开了。

服务粒度变细之后，出现一个新的问题，业务与服务的连接关系变复杂了，有什么好的优化方案么？



常见的，加入一个高可用服务分发层集群，并在协议设计时加入服务号，可以减少蜘蛛网状的依赖关系：

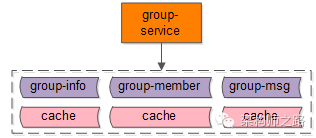
（1）调用方依赖分发层，传入服务号

（2）分发层依赖服务层，通过服务号参数分发

**【一个数据库对应一个service】**

数据访问service最初是从DAO/ORM的数据访问需求过来的，所以有些公司也有一个数据表一个service的玩法。

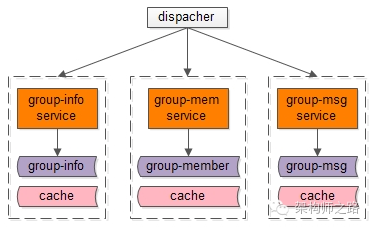
一个子业务对应一个service的玩法是：



（1）服务层，整个群业务是一个service

（2）存储层，实际可能对应了群信息、群成员、群消息等多个数据表

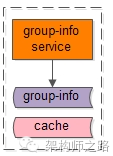
拆分成一个数据表一个service，则架构会变成这样：



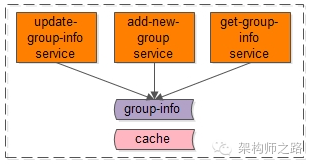
**群信息表，群成员表，群消息表**等各个数据表之间也解耦开了，不会相互影响了。

**【一个接口对应一个service】**

微服务架构中更极端的，甚至一个接口对应一个微服务，这样的话，架构就从：



演化为：



（1）**修改群信息**服务

（2）**增加群信息**服务

（3）**获取群信息**服务

多个服务操纵同一个数据表，使用同一片缓存，每个接口出问题，都不会影响其他接口。

### ****三、粒度粗细的优劣****

上文中谈到的服务化与微服务，不同粒度的服务化各有什么优劣呢？

总的来说，细粒度拆分的**优点**有：

（1）服务都能够独立部署

（2）扩容和缩容方便，有利于提高资源利用率

（3）拆得越细，耦合相对会减小

（4）拆得越细，容错相对会更好，一个服务出问题不影响其他服务

（5）扩展性更好

（6）…

细粒度拆分的**不足**也很明显：

（1）拆得越细，系统越复杂

（2）系统之间的依赖关系也更复杂

（3）运维复杂度提升

（4）监控更加复杂

（5）出问题时定位问题更难

（6）…

关于微服务架构的“粒度”问题，以及各粒度的优劣，大伙有什么好的看法，欢迎补充，建设性的意见将在后续文中和大伙share。

### ****四、结束的话****

聊了许多，有网友问，笔者对待服务化以及微服务粒度的看法，个人觉得，**以“子业务系统”粒度作为微服务的单位是比较合适**的：



末了，讨论完微服务架构的粒度，后续文章和大家聊一聊微服务的最佳实践，需要什么样的框架、组件、技术能够将服务化在较短的时间内开展起来，下周和大伙再聊。

==【完】==

回【无锁】[如何实现超高并发的无锁缓存？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959502&idx=1&sn=7a1c7d81a8e030856fb920844dc571c6&scene=21" \l "wechat_redirect" \t "_blank)

回【消息】[58到家通用实时消息平台架构细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "wechat_redirect" \t "_blank)

回【机房】[从IDC到云端架构迁移之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959495&idx=1&sn=0f8789d6803f94e190ba2787389e6d1b&scene=21" \l "wechat_redirect" \t "_blank)

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

回【单点】[单点系统架构的可用性与性能优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959480&idx=1&sn=337bd74410a6bef616128fd17abd08a8&scene=21" \l "wechat_redirect" \t "_blank)

回【事务】[多库多事务降低数据不一致概率](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959486&idx=1&sn=cd3e6b27af159f4eabd54f148eec638a&scene=21" \l "wechat_redirect" \t "_blank)

回【服务】[互联网架构为什么要做服务化？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959519&idx=1&sn=065074b135fc9cb243abe897261e1a72&scene=21" \l "wechat_redirect" \t "_blank)

## 为什么说要搞定微服务架构，先搞定RPC框架？

原创 2016-08-25 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959553&idx=1&sn=c1084e91875721c5f6baf544450afa38&scene=21" \l "#)

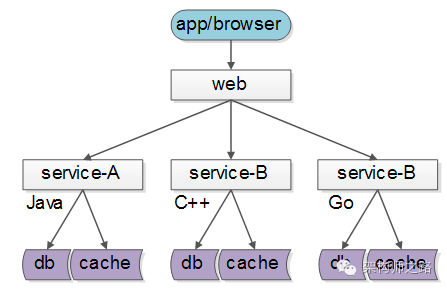
第一章聊了【[“为什么要进行服务化，服务化究竟解决什么问题”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959519&idx=1&sn=065074b135fc9cb243abe897261e1a72&scene=21" \l "wechat_redirect" \t "_blank)】

第二章聊了【[“微服务的服务粒度选型”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959530&idx=1&sn=ff84bd74745eee1577e7dfef8ce66bbe&scene=21" \l "wechat_redirect" \t "_blank)】

今天开始聊一些微服务的实践，第一块，RPC框架的原理及实践，为什么说要搞定微服务架构，先搞定RPC框架呢？

### ****一、需求缘起****

服务化的一个好处就是，不限定服务的提供方使用什么技术选型，能够实现大公司跨团队的技术解耦，如下图：



服务A是欧洲团队提供服务，欧洲团队的技术背景是Java，可以用Java实现服务；

服务B是美洲团队提供服务，可以用C++实现服务；

服务C是中国团队提供服务，可以用Go实现服务；

服务的上游调用方，按照接口、协议即可完成对远端服务的调用。

但实际上，99.9%的公司的团队规模有限，技术团队人数也有限，基本是使用同一套技术体系来调用和提供服务的：



这样的话，**如果没有统一的服务框架，RPC框架**，各个团队的服务提供方就需要各自实现一套序列化、反序列化、网络框架、连接池、收发线程、超时处理、状态机等“业务之外”的重复技术劳动，造成整体的低效。所以，**统一RPC框架**把上述“业务之外”的技术劳动统一处理，是服务化首要解决的问题。

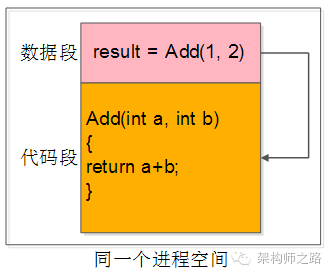
在达成【“使用统一的RPC框架”是正确的道路】这个一致的前提下，本文期望用简单通俗的言语简述一下一个通用RPC框架的技术点与实现。

### ****二、RPC背景与过程****

什么是RPC（Remote Procedure Call Protocol），远程过程调用？

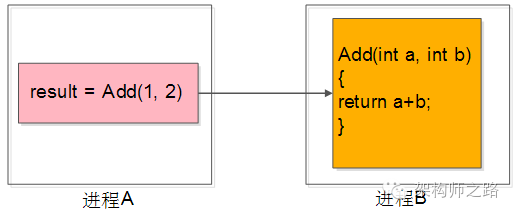
先来看下什么是本地函数调用，当我们写下：

int result = Add(1, 2);



这段代码的时候，我们知道，我们传入了1，2两个入参数，调用了本地代码段中的一个Add函数，得到了result出参。此时，传入数据，传出数据，代码段在同一个进程空间里，这是本地函数调用。

**那有没有办法，我们能够调用一个跨进程（所以叫“远程”，典型的，这个进程部署在另一台服务器上）的函数呢**？



最容易想到的，两个进程约定一个协议格式，使用Socket通信，来传输【入参】【调用哪个函数】【出参】。

假设请求报文协议是一个11字节的字节流：

http://mmbiz.qpic.cn/mmbiz_png/YrezxckhYOw0bibzTnvt7L6gUYIboKjaj5pAeBtd89DtFcBdQGJ5TlCERSaHrfDC4gxiaOVPDJDP3txA7AN2D8nw/640?wx_fmt=png&wxfrom=5&wx_lazy=1

（1）前3个字节填入函数名

（2）中间4个字节填入第一个参数

（3）末尾4个字节填入第二个参数

同时可以设计响应报文协议是一个4字节的字节流：

http://mmbiz.qpic.cn/mmbiz_png/YrezxckhYOw0bibzTnvt7L6gUYIboKjaj5knFI6ahjAMM6k135icQ09RibPxHtibBClHz5LDEcW2cdPIZOJhwLV7Mw/640?wx_fmt=png&wxfrom=5&wx_lazy=1

即处理结果。

调用方的代码可能变为：

request = MakePacket(“add”, 1, 2);

SendRequest\_ToService\_B(request);

response = RecieveRespnse\_FromService\_B();

int result = unMakePacket(respnse);

简单解释一下：

（1）讲传入参数变为字节流

（2）将字节流发给服务B

（3）从服务B接受返回字节流

（4）将返回字节流变为传出参数

服务方的代码可能变为：

request = RecieveRequest();

args/function = unMakePacket(request);

result = Add(1, 2);

response = MakePacket(result);

SendResponse(response);

这个过程也很好理解：

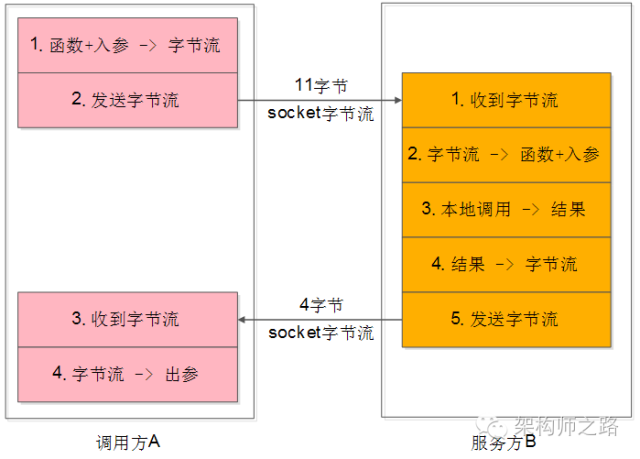
（1）服务端收到字节流

（2）将字节流转为函数名与参数

（3）本地调用函数得到结果

（4）将结果转变为字节流

（5）将字节流发送给调用方



这个过程用一张图描述如上，调用方与服务方的处理步骤都是非常清晰的。**这个过程存在最大的问题是什么呢？**

回答：调用方太麻烦了，每次都要关注很多底层细节

（1）入参到字节流的转化，即序列化应用层协议细节

（2）socket发送，即网络传输协议细节

（3）socket接受

（4）字节流到出参的转化，即反序列化应用层协议细节

**能不能调用层不关注这个细节呢？**

回答：可以，RPC框架就是解决这个问题的，它能够让调用方“像调用本地函数一样调用远端的函数（服务）”。

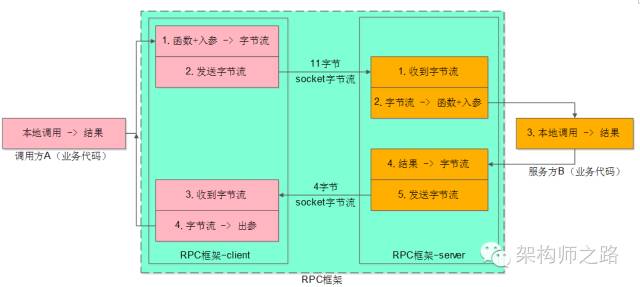
### ****三、RPC框架职责****

通过上面的讨论，**RPC框架要向调用方屏蔽各种复杂性，要向服务提供方也屏蔽各类复杂性**：

（1）调用方感觉就像调用本地函数一样

（2）服务提供方感觉就像实现一个本地函数一样来实现服务

**所以整个RPC框架又分为client部分与server部分，负责把整个非（1）（2）的各类复杂性屏蔽，这些复杂性就是RPC框架的职责。**



再细化一些，client端又包含：序列化、反序列化、连接池管理、负载均衡、故障转移、队列管理，超时管理、异步管理等等等等职责。

server端包含：服务端组件、服务端收发包队列、io线程、工作线程、序列化反序列化、上下文管理器、超时管理、异步回调等等等等职责。

however，因为篇幅有限，这些细节不做深入展开。

### ****四、结论****

（1）RPC框架是架构微服务化的首要基础组件，它能大大降低架构微服务化的成本，提高调用方与服务提供方的研发效率，屏蔽跨进程调用函数（服务）的各类复杂细节

（2）RPC框架的**职责**是：让调用方感觉就像调用本地函数一样调用远端函数、让服务提供方感觉就像实现一个本地函数一样来实现服务

==【完】==

回【演进】[好架构是进化来的，不是设计来的（58架构演进）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400276397&idx=1&sn=ea044079667b82f6cad58bcb743af7bc&scene=21" \l "wechat_redirect" \t "_blank)

回【微信】[微信为啥这么省流量](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400163013&idx=1&sn=911cf71925e2ba50d47955a713134acb&scene=21" \l "wechat_redirect" \t "_blank)

回【id】[细聊分布式ID生成方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "wechat_redirect" \t "_blank)

回【晋升】[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)

回【主从】[DB主从一致性架构优化4种方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959442&idx=1&sn=feb8ff75385d8031386e120ef3535329&scene=21" \l "wechat_redirect" \t "_blank)

## 微服务架构之RPC-client序列化细节

原创 2016-08-26 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959558&idx=1&sn=610f06c6d62a5c22311d27cf40f758ef&scene=21" \l "#)

第一章聊了【[“为什么要进行服务化，服务化究竟解决什么问题”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959519&idx=1&sn=065074b135fc9cb243abe897261e1a72&scene=21" \l "wechat_redirect" \t "_blank)】

第二章聊了【[“微服务的服务粒度选型”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959530&idx=1&sn=ff84bd74745eee1577e7dfef8ce66bbe&scene=21" \l "wechat_redirect" \t "_blank)】

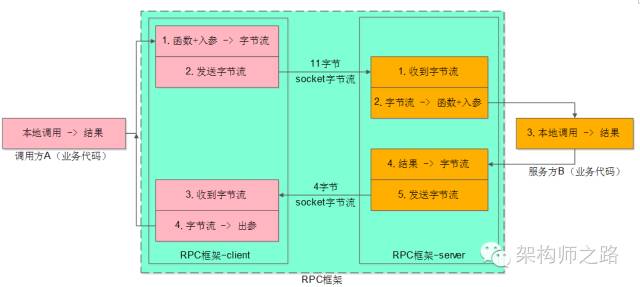
上一篇聊了【[“为什么说要搞定微服务架构，先搞定RPC框架？”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959553&idx=1&sn=c1084e91875721c5f6baf544450afa38&scene=21" \l "wechat_redirect" \t "_blank)】

通过上篇文章的介绍，知道了要实施微服务，首先要搞定RPC框架，RPC框架的职责要向【调用方】和【服务提供方】屏蔽各种复杂性：

（1）让调用方感觉就像调用本地函数一样

（2）让服务提供方感觉就像实现一个本地函数一样来实现服务

整个RPC框架又分为client部分与server部分：



RPC-client的部分流程如上图，要进行序列化反序列化（上图中的1、4），要进行发送字节流与接收字节流（上图中的2、3）。

通过上一篇文章的用户调研：

78%读者 -> 继续聊RPC框架技术细节

14%读者 -> 聊微服务其他实践

7%读者 -> 不聊微服务了，聊最终一致性

那么按照多数读者的意见，今天深入聊RPC的技术细节，本文先讨论RPC-client部分的【序列化反序列化】实施细节（笔者不是这方面的专家，有不对之处，欢迎大家指正，任何具有建设性意见的留言，将在下一章share给更多的小伙伴）。

**一、为什么要进行序列化**

工程师通常使用“对象”来进行数据的操纵：

class User{

         std::Stringuser\_name;

         uint64\_tuser\_id;

         uint32\_tuser\_age;

};

User u = new User(“shenjian”);

u.setUid(123);

u.setAge(35);

但当需要对数据进行存储（固化存储，缓存存储）或者传输（跨进程网络传输）时，“对象”就不这么好用了，往往需要把数据转化成连续空间的二进制字节流，一些典型的场景是：

（1）**数据库索引的磁盘存储**：数据库的索引在内存里是b+树或者hash的格式，但这个格式是不能够直接存储到磁盘上的，所以需要把b+树或者hash转化为连续空间的二进制字节流，才能存储到磁盘上

（2）**缓存的KV存储**：redis/memcache是KV类型的缓存，缓存存储的value必须是连续空间的二进制字节流，而不能够是User对象

（3）**数据的网络传输**：socket发送的数据必须是连续空间的二进制字节流，也不能是对象

所谓**序列化**（Serialization），就是将“对象”形态的数据转化为“连续空间二进制字节流”形态数据的过程，以方便存储与传输。这个过程的逆过程叫做反序列化。

**二、怎么进行序列化**

这是一个非常细节的问题，要是让你来把“对象”转化为字节流，你会怎么做？很容易想到的一个方法是xml（或者json）这类具有**自描述特性**的标记性语言：

<class name=”User”>

<element name=”user\_name” type=”std::String” value=”shenjian” />

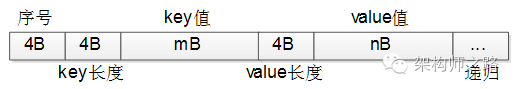
<element name=”user\_id” type=”uint64\_t” value=”123” />

<element name=”user\_age” type=”uint32\_t” value=”35” />

</class>

规定好转换规则，发送方很容易把User类的一个对象序列化为xml，服务方收到xml二进制流之后，也很容易将其范序列化为User对象（特别是语言支持反射的时候，就更easy了）。

第二个方法是自己实现二进制协议来进行序列化，还是以上面的User对象为例，可以设计一个这样的通用协议：



（1）头4个字节表示序号

（2）序号后面的4个字节表示key的长度m

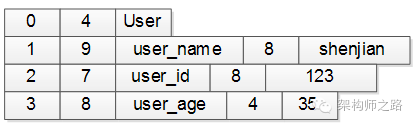
（3）接下来的m个字节表示key的值

（4）接下来的4个字节表示value的长度n

（5）接下来的n个字节表示value的值

（6）像xml一样递归下去，直到描述完整个对象

上面的User对象，用这个协议描述出来可能是这样的：



（1）第一行：序号4个字节（设0表示类名），类名长度4个字节（长度为4），接下来4个字节是类名（”User”），共12字节

（2）第二行：序号4个字节（1表示第一个属性），属性长度4个字节（长度为9），接下来9个字节是属性名（”user\_name”），属性值长度4个字节（长度为8），属性值8个字节（值为”shenjian”），共29字节

（3）第三行：序号4个字节（2表示第二个属性），属性长度4个字节（长度为7），接下来7个字节是属性名（”user\_id”），属性值长度4个字节（长度为8），属性值8个字节（值为123），共27字节

（3）第四行：序号4个字节（3表示第三个属性），属性长度4个字节（长度为8），接下来8个字节是属性名（”user\_name”），属性值长度4个字节（长度为4），属性值4个字节（值为35），共24字节

整个二进制字节流共12+29+27+24=92字节

实际的序列化协议要考虑的细节远比这个多，例如：强类型的语言不仅要还原属性名，属性值，还要还原属性类型；复杂的对象不仅要考虑普通类型，还要考虑对象嵌套类型等。however，序列化的思路都是类似的。

**三、序列化协议要考虑什么因素**

不管使用成熟协议xml/json，还是自定义二进制协议来序列化对象，序列化协议设计时要考虑哪些因素呢？

（1）**解析效率**：这个应该是序列化协议应该首要考虑的因素，像xml/json解析起来比较耗时，需要解析doom树，二进制自定义协议解析起来效率就很高

（2）**压缩率，传输有效性**：同样一个对象，xml/json传输起来有大量的xml标签，信息有效性低，二进制自定义协议占用的空间相对来说就小多了

（3）**扩展性与兼容性**：是否能够方便的增加字段，增加字段后旧版客户端是否需要强制升级，都是需要考虑的问题，xml/json和上面的二进制协议都能够方便的扩展

（4）**可读性与可调试性**：这个很好理解，xml/json的可读性就比二进制协议好很多

（5）**跨语言**：上面的两个协议都是跨语言的，有些序列化协议是与开发语言紧密相关的，例如dubbo的序列化协议就只能支持Java的RPC调用

（6）**通用性**：xml/json非常通用，都有很好的第三方解析库，各个语言解析起来都十分方便，上面自定义的二进制协议虽然能够跨语言，但每个语言都要写一个简易的协议客户端

（7）欢迎大家补充…

**四、业内常见的序列化方式**

（1）**xml/json**：解析效率，压缩率都较差；扩展性、可读性、通用性较好

（2）thrift：没有用过，欢迎大家补充

（3）**protobuf**：Google出品，必属精品，各方面都不错，强烈推荐，属于二进制协议，可读性差了点，但也有类似的to-string协议帮助调试问题

（4）Avro：没有用过，欢迎大家补充

（5）CORBA：没有用过，欢迎大家补充

（6）**mc\_pack**：懂的同学就懂，不懂的就不懂了，09年用过，传说各方面都超越protobuf，懂行的同学可以说一下现状

（7）…

**五、后文预告**

RPC-client的部分，除了要进行序列化反序列化，还要进行发送字节流与接收字节流，下一篇文章会介绍这一部分内容。

RPC-client中数据的发送与接收远比序列化反序列化复杂，其涉及“连接池、负载均衡、故障转移、队列、超时、异步、上下文回调管理”等技术，具体细节，下篇再沟通。

==【完】==

回【sed】[一分钟sed入门（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403340380&idx=1&sn=9b4ced5a19f60b0cebf6fd582f1d52ae&scene=21" \l "wechat_redirect" \t "_blank)

回【广告】[一分钟读懂互联网广告竞价策略（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403326874&idx=1&sn=ca03344326f35b2971b3ecae2026e8df&scene=21" \l "wechat_redirect" \t "_blank)

回【阵列】[一张“神图”看懂单机/集群/热备/磁盘阵列（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400720056&idx=1&sn=10c796793e27d475603f97b23e137c12&scene=21" \l "wechat_redirect" \t "_blank)

回【awk】[一分钟学awk够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400743254&idx=1&sn=147abc381ccd61e28f52699735c8748e&scene=21" \l "wechat_redirect" \t "_blank)

回【perl】[十分钟学perl够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400764946&idx=1&sn=4efab1953696c2e21071e58a74da9b82&scene=21" \l "wechat_redirect" \t "_blank)

回【mongo】[一分钟了解mongodb（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=201250814&idx=1&sn=fd6d85b727e1229e3324850217748d0f&scene=21" \l "wechat_redirect" \t "_blank)

回【2pc】[一分钟了解两阶段提交2PC（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400856212&idx=1&sn=b9734da77f2259bdd1fa89700b47eaff&scene=21" \l "wechat_redirect" \t "_blank)

回【join】[30秒懂SQL中的join（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400889133&idx=1&sn=3ea1c06fc76e92349a822f41f45b77b5&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

## RPC-client异步收发核心细节？

原创 2016-09-12 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959576&idx=1&sn=2be8d3f61effe7118abf920a175da710&scene=21" \l "#)

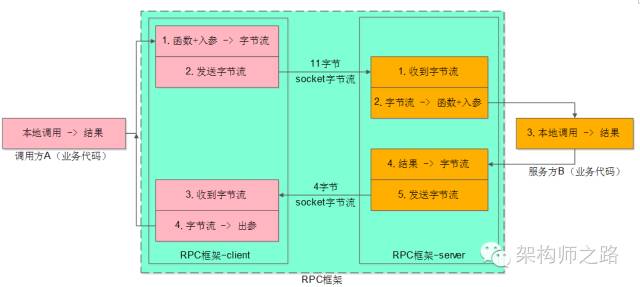
第一章聊了【[“为什么要进行服务化，服务化究竟解决什么问题”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959519&idx=1&sn=065074b135fc9cb243abe897261e1a72&scene=21" \l "wechat_redirect" \t "_blank)】

第二章聊了【[“微服务的服务粒度选型”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959530&idx=1&sn=ff84bd74745eee1577e7dfef8ce66bbe&scene=21" \l "wechat_redirect" \t "_blank)】

第三章聊了【[“为什么说要搞定微服务架构，先搞定RPC框架？”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959553&idx=1&sn=c1084e91875721c5f6baf544450afa38&scene=21" \l "wechat_redirect" \t "_blank)】

上一章聊了【[“微服务架构之RPC-client序列化细节”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959558&idx=1&sn=610f06c6d62a5c22311d27cf40f758ef&scene=21" \l "wechat_redirect" \t "_blank)】

通过上篇文章的介绍，知道了要实施微服务，首先要搞定RPC框架，RPC框架分为客户端部分与服务端部分。



RPC-client的部分又分为：

（1）序列化反序列化的部分（上图中的1、4）

（2）发送字节流与接收字节流的部分（上图中的2、3）

前一篇文章讨论了序列化与范序列化的细节，这一篇文章将讨论发送字节流与接收字节流的部分。

客户端调用又分为同步调用与异步调用

**同步调用**的代码片段为：

Result = Add(Obj1, Obj2);// 得到Result之前处于阻塞状态

**异步调用**的代码片段为：

Add(Obj1, Obj2, callback);// 调用后直接返回，不等结果

处理结果通过回调得到：

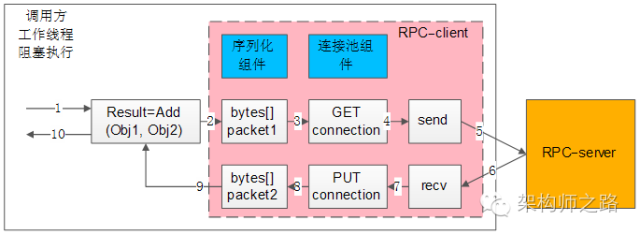
callback(Result){// 得到处理结果后会调用这个回调函数

         …

}

这两个调用方式，RPC-client里，处理方式也不一样，下文逐一叙述。

**RPC-client同步调用**



所谓同步调用，在得到结果之前，一直处于阻塞状态，会一直占用一个工作线程，上图简单的说明了一下组件、交互、流程步骤。

上图中的**左边大框**，就代表了调用方的一个工作线程。

**左边粉色中框**，代表了RPC-client组件。

**右边橙色框**，代表了RPC-server。

**蓝色两个小框**，代表了同步RPC-client两个核心组件，序列化组件与连接池组件。

**白色的流程小框**，以及箭头序号1-10，代表整个工作线程的串行执行步骤：

1）**业务代码发起RPC调用**，Result=Add(Obj1,Obj2)

2）序列化组件，将对象调用序列化成二进制字节流，可理解为一个待发送的包packet1

3）通过连接池组件拿到一个可用的连接connection

4）通过连接connection将包packet1发送给RPC-server

5）发送包在网络传输，发给RPC-server

6）响应包在网络传输，发回给RPC-client

7）通过连接connection从RPC-server收取响应包packet2

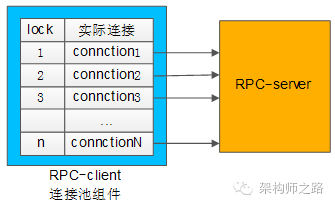
8）通过连接池组件，将conneciont放回连接池

9）序列化组件，将packet2范序列化为Result对象返回给调用方

10）**业务代码获取Result结果**，工作线程继续往下走

RPC框架需要支持负载均衡、故障转移、发送超时，这些特性都是通过连接池组件去实现的。

**连接池组件**



典型连接池组件对外提供的接口为：

int ConnectionPool::init(…);

Connection ConnectionPool::getConnection();

intConnectionPool::putConnection(Connection t);

【INIT】

和下游RPC-server（一般是一个集群），建立N个tcp长连接，即所谓的连接“池”

【getConnection】

从连接“池”中拿一个连接，加锁（置一个标志位），返回给调用方

【putConnection】

将一个分配出去的连接放回连接“池”中，解锁（也是置一个标志位）

**如何实现负载均衡？**

回答：连接池中建立了与一个RPC-server集群的连接，连接池在返回连接的时候，需要具备随机性。

**如何实现故障转移？**

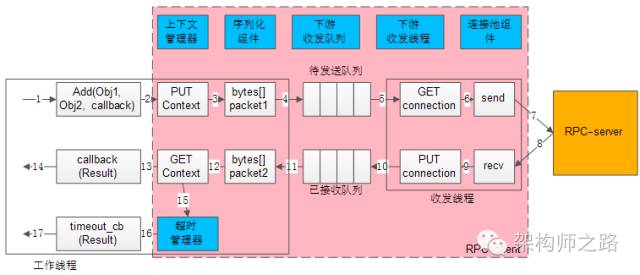
回答：连接池中建立了与一个RPC-server集群的连接，当连接池发现某一个机器的连接异常后，需要将这个机器的连接排除掉，返回正常的连接，在机器恢复后，再将连接加回来。

**如何实现发送超时？**

回答：因为是同步阻塞调用，拿到一个连接后，使用带超时的send/recv即可实现带超时的发送和接收。

总的来说，同步的RPC-client的实现是相对比较容易的，序列化组件、连接池组件配合多工作线程数，就能够实现。还有一个问题，就是【[“工作线程数设置多少最为合适？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)”】，这个问题在之前的文章中讨论过，此处不再深究。

**RPC-client异步回调**



所谓**异步回调**，在得到结果之前，不会处于阻塞状态，理论上任何时间都没有任何线程处于阻塞状态，因此异步回调的模型，理论上只需要很少的工作线程与服务连接就能够达到很高的吞吐量。

上图中**左边的框框**，是少量工作线程（少数几个就行了）进行调用与回调。

**中间粉色的框框**，代表了RPC-client组件。

**右边橙色框**，代表了RPC-server。

**蓝色六个小框**，代表了异步RPC-client六个核心组件：上下文管理器，超时管理器，序列化组件，下游收发队列，下游收发线程，连接池组件。

**白色的流程小框**，以及箭头序号1-17，代表整个工作线程的串行执行步骤：

1）**业务代码发起异步RPC调用**，Add(Obj1,Obj2, callback)

2）上下文管理器，将请求，回调，上下文存储起来

3）序列化组件，将对象调用序列化成二进制字节流，可理解为一个待发送的包packet1

4）下游收发队列，将报文放入“待发送队列”，此时调用返回，不会阻塞工作线程

5）下游收发线程，将报文从“待发送队列”中取出，通过连接池组件拿到一个可用的连接connection

6）通过连接connection将包packet1发送给RPC-server

7）发送包在网络传输，发给RPC-server

8）响应包在网络传输，发回给RPC-client

9）通过连接connection从RPC-server收取响应包packet2

10）下游收发线程，将报文放入“已接受队列”，通过连接池组件，将conneciont放回连接池

11）下游收发队列里，报文被取出，此时回调将要开始，不会阻塞工作线程

12）序列化组件，将packet2范序列化为Result对象

13）上下文管理器，将结果，回调，上下文取出

14）**通过callback回调业务代码，返回Result结果**，工作线程继续往下走

如果请求长时间不返回，处理流程是：

15）上下文管理器，请求长时间没有返回

16）超时管理器拿到超时的上下文

17）**通过timeout\_cb回调业务代码**，工作线程继续往下走

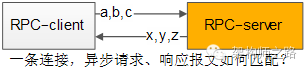
**上下文管理器**

**为什么需要上下文管理器？**

回答：由于请求包的发送，响应包的回调都是异步的，甚至不在同一个工作线程中完成，需要一个组件来记录一个请求的上下文，把请求-响应-回调等一些信息匹配起来。

**如何将请求-响应-回调这些信息匹配起来？**

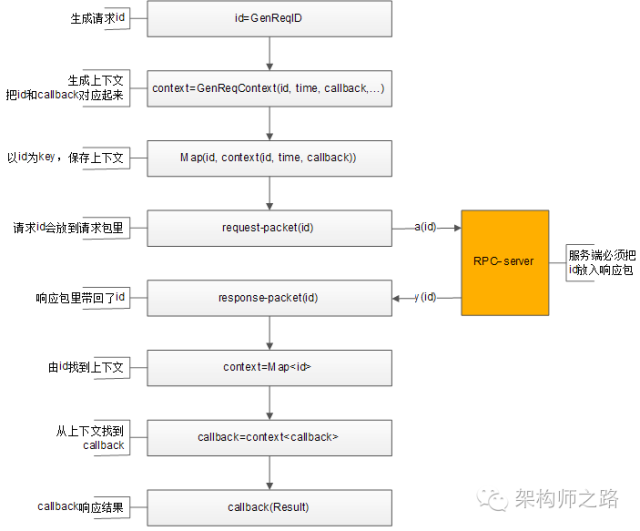
这是一个很有意思的问题，通过一条连接往下游服务发送了a，b，c三个请求包，异步的收到了x，y，z三个响应包：



**（1）怎么知道哪个请求包与哪个响应包对应？**

**（2）怎么知道哪个响应包与哪个回调函数对应？**

回答：这是通过【请求id】来实现请求-响应-回调的串联的。



整个处理流程如上，通过请求id，上下文管理器来对应请求-响应-callback之间的映射关系：

1）生成请求id

2）生成请求上下文context，上下文中包含发送时间time，回调函数callback等信息

3）上下文管理器记录req-id与上下文context的映射关系，

4）将req-id打在请求包里发给RPC-server

5）RPC-server将req-id打在响应包里返回

6）由响应包中的req-id，通过上下文管理器找到原来的上下文context

7）从上下文context中拿到回调函数callback

8）callback将Result带回，推动业务的进一步执行

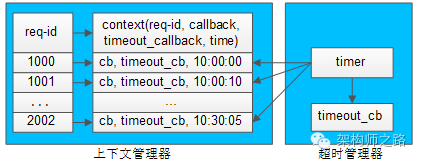
**如何实现负载均衡，故障转移？**

回答：与同步的连接池思路相同。不同在于，同步连接池使用阻塞方式收发，需要与一个服务的一个ip建立多条连接，异步收发，一个服务的一个ip只需要建立少量的连接（例如，一条tcp连接）。

**如何实现超时发送与接收？**

回答：同步阻塞发送，可以直接使用带超时的send/recv来实现，异步非阻塞的nio的网络报文收发，如何实现超时接收呢？（由于连接不会一直等待回包，那如何知晓超时呢？）这时，超时管理器就上场啦。

**超时管理器**



超时管理器，用于实现请求回包超时回调处理。

每一个请求发送给下游RPC-server，会在上下文管理器中保存req-id与上下文的信息，上下文中保存了请求很多相关信息，例如req-id，回包回调，超时回调，发送时间等。

超时管理器启动timer对上下文管理器中的context进行扫描，看上下文中请求发送时间是否过长，如果过长，就不再等待回包，直接超时回调，推动业务流程继续往下走，并将上下文删除掉。

如果超时回调执行后，正常的回包又到达，通过req-id在上下文管理器里找不到上下文，就直接将请求丢弃（因为已经超时处理过了）。

however，异步回调和同步回调相比，除了序列化组件和连接池组件，会多出上下文管理器，超时管理器，下游收发队列，下游收发线程等组件，并且对调用方的调用习惯有影响（同步->回调）。异步回调能提高系统整体的吞吐量，具体使用哪种方式实现RPC-client，可以结合业务场景来选取（对时延敏感的可以选用同步，对吞吐量敏感的可以选用异步）。

末了，通过最近几篇RPC框架细节的文章阅读量来看，貌似大伙对细节不是特别感兴趣，后续文章就不再延续这个系列啦。

==【完】==

回【sed】[一分钟sed入门（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403340380&idx=1&sn=9b4ced5a19f60b0cebf6fd582f1d52ae&scene=21" \l "wechat_redirect" \t "_blank)

回【广告】[一分钟读懂互联网广告竞价策略（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403326874&idx=1&sn=ca03344326f35b2971b3ecae2026e8df&scene=21" \l "wechat_redirect" \t "_blank)

回【阵列】[一张“神图”看懂单机/集群/热备/磁盘阵列（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400720056&idx=1&sn=10c796793e27d475603f97b23e137c12&scene=21" \l "wechat_redirect" \t "_blank)

回【awk】[一分钟学awk够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400743254&idx=1&sn=147abc381ccd61e28f52699735c8748e&scene=21" \l "wechat_redirect" \t "_blank)

回【perl】[十分钟学perl够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400764946&idx=1&sn=4efab1953696c2e21071e58a74da9b82&scene=21" \l "wechat_redirect" \t "_blank)

回【mongo】[一分钟了解mongodb（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=201250814&idx=1&sn=fd6d85b727e1229e3324850217748d0f&scene=21" \l "wechat_redirect" \t "_blank)

回【2pc】[一分钟了解两阶段提交2PC（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400856212&idx=1&sn=b9734da77f2259bdd1fa89700b47eaff&scene=21" \l "wechat_redirect" \t "_blank)

回【join】[30秒懂SQL中的join（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400889133&idx=1&sn=3ea1c06fc76e92349a822f41f45b77b5&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

## 微信为什么不丢消息？

原创 2016-10-09 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959606&idx=1&sn=f9561231dd33bcd0550b8d0d59d6b876&chksm=bd2d04ea8a5a8dfce90c870279a7f74b7aedd802c2d699dd919d7e40ebe30699381517c2d54b&scene=21" \l "#)

上一章和大家分享了《[http如何像tcp一样实时的收消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959605&idx=1&sn=21f25087bef3c3a966ef03b824365621&scene=21" \l "wechat_redirect" \t "_blank)》， 本章来聊一聊即时通讯（Instant Messaging，后简称im）**消息的可靠投递**。

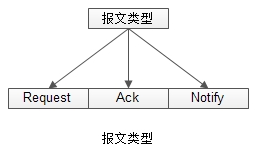
**一、报文类型**

im的客户端与服务器通过发送报文（也就是网络包）来完成消息的传递，报文分为三种

请求报文（request，后简称为为R）

应答报文（acknowledge，后简称为A）

通知报文（notify，后简称为N），这三种报文的解释如下：



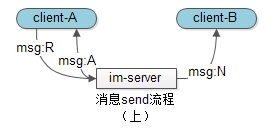
**R**：客户端主动发送给服务器的报文

**A**：服务器被动应答客户端的报文，一个A对应一个R

**N**：服务器主动发送给客户端的报文

**二、普通消息投递流程**

用户A给用户B发送一个“你好”，流程如下：



1）client-A向im-server发送一个消息请求包，即msg:R

2）im-server在成功处理后，回复client-A一个消息响应包，即msg:A

3）如果此时client-B在线，则im-server主动向client-B发送一个消息通知包，即msg:N（当然，如果client-B不在线，则消息会存储离线）

**三、上述消息投递流程出现的问题**

从流程图中容易看到，发送方client-A收到msg:A后，只能说明im-server成功接收到了消息，并不能说明client-B接收到了消息。在若干场景下，可能出现msg:N包丢失，且发送方client-A完全不知道，例如：

1）服务器崩溃，msg:N包未发出

2）网络抖动，msg:N包被网络设备丢弃

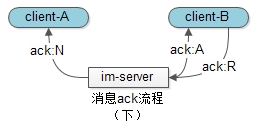
3）client-B崩溃，msg:N包未接收

结论是悲观的：**接收方client-B是否有收到msg:N，发送方client-A完全不可控，那怎么办呢？**

**四、应用层确认+im消息可靠投递的六个报文**

upd是一种不可靠的传输层协议，tcp是一种可靠的传输层协议，tcp是如何做到可靠的？答案是：超时、重传、确认。

要想实现应用层的消息可靠投递，必须加入应用层的确认机制，即：要想让发送方client-A确保接收方client-B收到了消息，必须让接收方client-B给一个消息的确认，这个应用层的确认的流程，与消息的发送流程类似：



4）client-B向im-server发送一个ack请求包，即ack:R

5）im-server在成功处理后，回复client-B一个ack响应包，即ack:A

6）则im-server主动向client-A发送一个ack通知包，即ack:N

至此，发送“你好”的client-A，在收到了ack:N报文后，才能确认client-B真正接收到了“你好”。

会发现，一条消息的发送，分别包含（上）（下）两个半场，即msg的R/A/N三个报文，ack的R/A/N三个报文，一个应用层即时通讯消息的可靠投递，共涉及6个报文，这就是im系统中消息投递的最核心技术。

**五、可靠消息投递存在什么问题**

期望六个报文完成消息的可靠投递，但实际情况，**msg:N，ack:N这两个报文都可能丢失**（原因如第二章所述，可能是服务器奔溃、网络抖动、或者客户端奔溃），此时client-A都收不到期待的ack:N报文，即client-A不能确认client-B是否收到“你好”，但这两个报文的丢失对应的业务影响又大有不同：

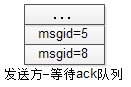
1）msg:N包丢失，业务结果是client-B没有收到消息

2）ack:N包丢失，业务结果是client-B收到了消息，只是client-A不知道而已

那怎么办呢？

**六、消息的超时与重传**

client-A发出了msg:R，收到了msg:A之后，在一个期待的时间内，如果没有收到ack:N，client-A会尝试将msg:R重发。可能client-A同时发出了很多消息，故client-A需要在本地维护一个等待ack队列，并配合timer超时机制，来记录哪些消息没有收到ack:N，以定时重发。



一旦收到了ack:N，说明client-B收到了“你好”消息，对应的消息将从“等待ack队列”中移除。

**七、消息的重传存在什么问题**

第五章提到过，msg:N，ack:N都有可能丢失：

1）msg:N报文丢失，说明client-B之前压根没有收到“你好”报文，超时与重传机制十分有效

2）ack:N报文丢失，说明client-B之前已经收到了“你好”报文（只是client-A不知道而已），超时与重传机制将导致client-B收到重复的消息，那怎么办呢？

**八、消息的去重**

解决方法也很简单，由发送方client-A生成一个消息去重的msgid，保存在“等待ack队列”里，同一条消息使用相同的msgid来重传，供client-B去重，而不影响用户体验。

**九、其他**

1）上述设计理念，由客户端重传，可以保证服务端无状态性（架构设计基本准则）

2）如果client-B不在线，im-server保存了离线消息后，要伪造ack:N发送给client-A

**十、总结**

1）**im系统是通过超时、重传、确认、去重的机制来保证消息的可靠投递，不丢不重**

2）**一个“你好”的发送，包含上半场msg:R/A/N与下半场ack:R/A/N的6个报文**

3）im系统**难以做到系统层面的不丢不重，只能做到业务层面的不丢不重**

末了，微信的消息是不是这么发送的，偶不太清楚，清楚的同学可以说一说。

==【完】==

回【sed】[一分钟sed入门（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403340380&idx=1&sn=9b4ced5a19f60b0cebf6fd582f1d52ae&scene=21" \l "wechat_redirect" \t "_blank)

回【广告】[一分钟读懂互联网广告竞价策略（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403326874&idx=1&sn=ca03344326f35b2971b3ecae2026e8df&scene=21" \l "wechat_redirect" \t "_blank)

回【阵列】[一张“神图”看懂单机/集群/热备/磁盘阵列（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400720056&idx=1&sn=10c796793e27d475603f97b23e137c12&scene=21" \l "wechat_redirect" \t "_blank)

回【awk】[一分钟学awk够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400743254&idx=1&sn=147abc381ccd61e28f52699735c8748e&scene=21" \l "wechat_redirect" \t "_blank)

回【perl】[十分钟学perl够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400764946&idx=1&sn=4efab1953696c2e21071e58a74da9b82&scene=21" \l "wechat_redirect" \t "_blank)

回【mongo】[一分钟了解mongodb（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=201250814&idx=1&sn=fd6d85b727e1229e3324850217748d0f&scene=21" \l "wechat_redirect" \t "_blank)

回【2pc】[一分钟了解两阶段提交2PC（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400856212&idx=1&sn=b9734da77f2259bdd1fa89700b47eaff&scene=21" \l "wechat_redirect" \t "_blank)

回【join】[30秒懂SQL中的join（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400889133&idx=1&sn=3ea1c06fc76e92349a822f41f45b77b5&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

## 微信为啥不丢“离线消息”？

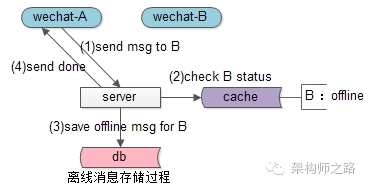
原创 2016-11-02 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959629&idx=1&sn=053d85a862df6e6c01147a1cf95bdbf2&chksm=bd2d04918a5a8d87a32305fa8ef1603bb0e73e5c4b78afb5a144327841e08840188c4541b4aa&scene=21" \l "#)

**需求缘起**

当发送方用户A发送消息给接收方用户B时，如果用户B在线，之前的文章《[微信为啥不丢“在线消息”？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959606&idx=1&sn=f9561231dd33bcd0550b8d0d59d6b876&chksm=bd2d04ea8a5a8dfce90c870279a7f74b7aedd802c2d699dd919d7e40ebe30699381517c2d54b&scene=21" \l "wechat_redirect" \t "_blank)》聊过，可以通过**应用层的确认，发送方的超时重传，接收方的去重保证业务层面消息的不丢不重**。

那如果接收方用户B不在线，系统是如何保证消息的可达性的呢？这是本文要讨论的问题。

**问题：接收方不在线时，消息发送的流程是怎么样的？**



回答：如上图所述，

（1）用户A发送消息给用户B

（2）服务器查看用户B的状态为offline

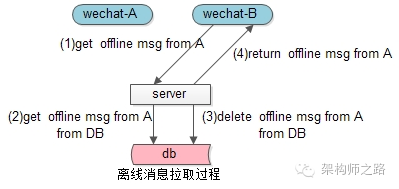
（3）服务器将消息存储到DB中

（4）服务器返回用户A发送成功（对于发送方而言，消息落地DB就认为发送成功）

**问题：离线消息表的设计，拉取离线的过程？**

receiver\_uid, msg\_id, time, sender\_uid,msg\_type, msg\_content …

访问模式：接收方B要拉取发送方A给ta发送的离线消息，只需在receiver\_uid(B), sender\_uid(A)上查询，然后把离线消息删除，再把消息返回B即可。



整体流程如上图所述，

（1）用户B拉取用户A发送给ta的离线消息

（2）服务器从DB中拉取离线消息

（3）服务器从DB中把离线消息删除

（4）服务器返回给用户B想要的离线消息

**问题：上述流程存在的问题？**

回答：如果用户B有很多好友，登陆时客户端需要对所有好友进行离线消息拉取，客户端与服务器交互次数较多

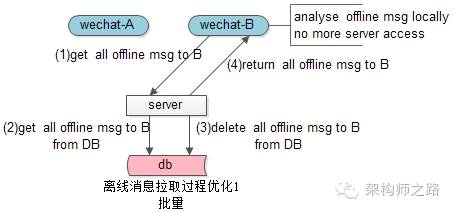
客户端伪代码：

for(all uid in B’s friend-list){      // 登陆时所有好友都要拉取

         get\_offline\_msg(B,uid);   // 与服务器交互

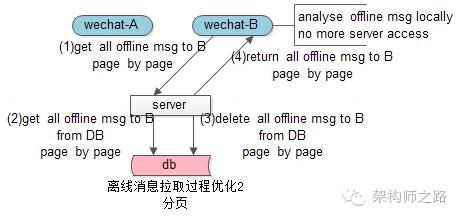
}

优化方案一：先拉取各个好友的离线消息数量，真正用户B进去看离线消息时，才往服务器发送拉取请求（手机端为了节省流量，经常会使用这个按需拉取的优化）



优化方案二：一次性拉取所有好友发送给用户B的离线消息，到**客户端本地**再根据sender\_uid进行计算，这样的话，离校消息表的访问模式就变为->只需要按照receiver\_uid来查询了。登录时与服务器的交互次数降低为了1次。

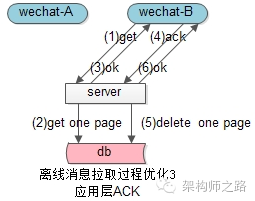
**问题：用户B一次性拉取所有好友发给ta的离线消息，消息量很大时，一个请求包很大，速度慢，容易卡顿怎么办？**



回答：分页拉取，根据业务需求，先拉取最新（或者最旧）的一页消息，再按需一页页拉取。

**问题：如何保证可达性，上述步骤第三步执行完毕之后，第四个步骤离线消息返回给客户端过程中，服务器挂点，路由器丢消息，或者客户端crash了，那离线消息岂不是丢了么（数据库已删除，用户还没收到）？**

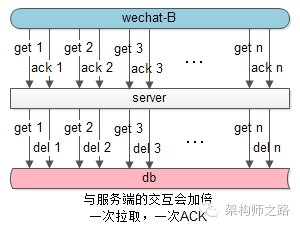
回答：嗯，如果按照上述的1，2，3，4步流程，的确是的，那**如何保证离线消息的可达性？**



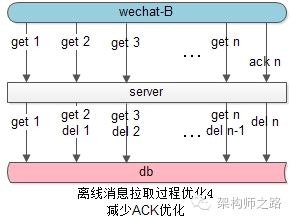
如同在线消息的应用层ACK机制一样，离线消息拉时，不能够直接删除数据库中的离线消息，而必须等应用层的离线消息ACK（说明用户B真的收到离线消息了），才能删除数据库中的离线消息。

**问题：如果用户B拉取了一页离线消息，却在ACK之前crash了，下次登录时会拉取到重复的离线消息么？**

回答：拉取了离线消息却没有ACK，服务器不会删除之前的离线消息，故下次登录时系统层面还会拉取到。但在业务层面，可以根据msg\_id去重。SMC理论：**系统层面无法做到消息不丢不重，业务层面可以做到，对用户无感知**。



**问题：假设有N页离线消息，现在每个离线消息需要一个ACK，那么岂不是客户端与服务器的交互次数又加倍了？有没有优化空间？**



回答：不用每一页消息都ACK，在拉取第二页消息时相当于第一页消息的ACK，此时服务器再删除第一页的离线消息即可，最后一页消息再ACK一次。这样的效果是，不管拉取多少页离线消息，只会多一个ACK请求，与服务器多一次交互。

**总结**

“离线消息”的可达性可能比大家想象的要复杂，常见的优化有：

（1）对于同一个用户B，一次性拉取所有用户发给ta的离线消息，再在客户端本地进行发送方分析，相比按照发送方一个个进行消息拉取，能大大减少服务器交互次数

（2）分页拉取，先拉取计数再按需拉取，是无线端的常见优化

（3）应用层的ACK，应用层的去重，才能保证离线消息的不丢不重

（4）下一页的拉取，同时作为上一页的ACK，能够极大减少与服务器的交互次数

即时通讯系统中，消息的可达性，状态的一致性都是很有意思的话题，关于“群消息”的在线投递与离线拉取还没有介绍过，如果大家感兴趣，后续可以一起探讨：

==【完】==

回【http】[http如何像tcp一样实时的收消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959605&idx=1&sn=21f25087bef3c3a966ef03b824365621&scene=21" \l "wechat_redirect" \t "_blank)

回【可靠】[微信为什么不丢消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959606&idx=1&sn=f9561231dd33bcd0550b8d0d59d6b876&chksm=bd2d04ea8a5a8dfce90c870279a7f74b7aedd802c2d699dd919d7e40ebe30699381517c2d54b&scene=21" \l "wechat_redirect" \t "_blank)

回【状态】[QQ状态同步究竟是推还是拉？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959618&idx=1&sn=912a9af6d000c6681dc645e53590729b&chksm=bd2d049e8a5a8d884f6acb35bc5b40edf88127219542c9e3033bb4c2e33854657c315c8a93b7&scene=21" \l "wechat_redirect" \t "_blank)

回【容量】[互联网架构，如何进行容量设计？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959542&idx=1&sn=2494bbea9a855e0e1c3ccd6d2562a600&scene=21" \l "wechat_redirect" \t "_blank)

## 群消息这么复杂，怎么能做到不丢不重？

原创 2016-11-09 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959643&idx=1&sn=844afa6a31770fa587474ecd73c3b3b3&chksm=bd2d04878a5a8d91c5c93ad8e85254185c63eb419457efbedcba54a9b8a9053da1e8980a694a&scene=21" \l "#)

**【需求缘起】**

之前的文章更多的聊了单对单的消息投递：

《[微信为什么不丢消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959606&idx=1&sn=f9561231dd33bcd0550b8d0d59d6b876&chksm=bd2d04ea8a5a8dfce90c870279a7f74b7aedd802c2d699dd919d7e40ebe30699381517c2d54b&scene=21" \l "wechat_redirect" \t "_blank)》

《[http如何像tcp一样实时的收消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959605&idx=1&sn=21f25087bef3c3a966ef03b824365621&scene=21" \l "wechat_redirect" \t "_blank)》

群聊是多人社交的基本诉求，不管是QQ群，还是微信群，一个群友在群内发了一条消息：

（1）在线的群友能第一时间收到消息

（2）离线的群友能在登陆后收到消息

由于“消息风暴扩散系数”的存在（概念详见《[QQ状态同步究竟是推还是拉？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959618&idx=1&sn=912a9af6d000c6681dc645e53590729b&chksm=bd2d049e8a5a8d884f6acb35bc5b40edf88127219542c9e3033bb4c2e33854657c315c8a93b7&scene=21" \l "wechat_redirect" \t "_blank)》），群消息的复杂度要远高于单对单消息。群消息的**实时性，可达性，离线消息**是今天将要讨论的核心话题。

**【常见的群消息流程】**

开始讲群消息投递流程之前，先介绍两个群业务的核心数据结构：

**群成员表**：用来描述一个群里有多少成员

t\_group\_users(group\_id, user\_id)

**群离线消息表**：用来描述一个群成员的离线消息

t\_offine\_msgs(user\_id, group\_id, sender\_id,time, msg\_id, msg\_detail)

业务场景举例：

（1）一个群中有x,A,B,C,D共5个成员，成员x发了一个消息

（2）成员A与B在线，期望实时收到消息

（3）成员C与D离线，期望未来拉取到离线消息

系统架构简介：

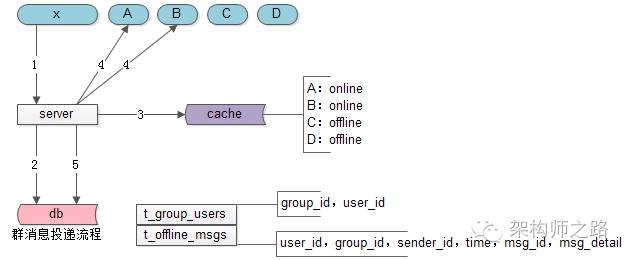
（1）客户端：x,A,B,C,D共5个客户端用户

（2）服务端

         （2.1）所有模块与服务抽象为server

         （2.2）所有用户在线状态抽象存储在高可用cache里

         （2.3）所有数据信息，例如群成员、群离线消息抽象存储在db里



典型群消息投递流程，如图步骤1-4所述：

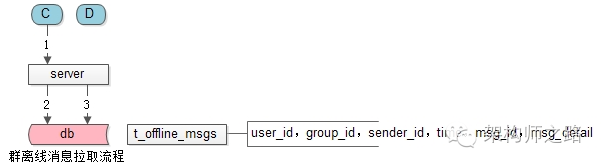
步骤1：群消息发送者x向server发出群消息

步骤2：server去db中查询群中有多少用户(x,A,B,C,D)

步骤3：server去cache中查询这些用户的在线状态

步骤4：对于群中**在线**的用户A与B，群消息server进行实时推送

步骤5：对于群中**离线**的用户C与D，群消息server进行离线存储



典型的群离线消息拉取流程，如图步骤1-3所述：

步骤1：离线消息拉取者C向server拉取群离线消息

步骤2：server从db中拉取离线消息并返回群用户C

步骤3：server从db中删除群用户C的群离线消息

**存在的问题**

上述流程是最容易想，也最容易理解的，存在的问题也最显而易见：对于同一份群消息的内容，多个离线用户存储了很多份。假设群中有200个用户离线，离线消息则冗余了200份，这极大的增加了数据库的存储压力。

**【群消息优化1：减少存储量】**

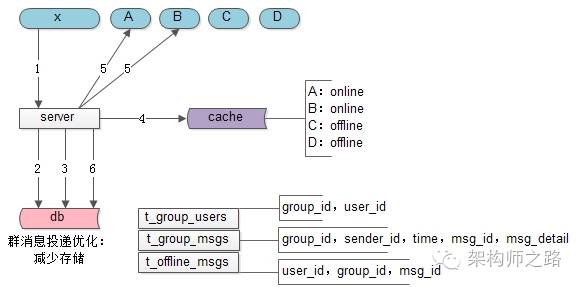
为了减少离线消息的冗余度，增加一个群消息表，用来存储所有群消息的内容，离线消息表只存储用户的群离线消息msg\_id，就能大大的降低数据库的冗余存储量

**群消息表**：用来存储一个群中所有的消息内容

t\_group\_msgs(group\_id, sender\_id, time,msg\_id, msg\_detail)

**群离线消息表**：优化后只存储msg\_id

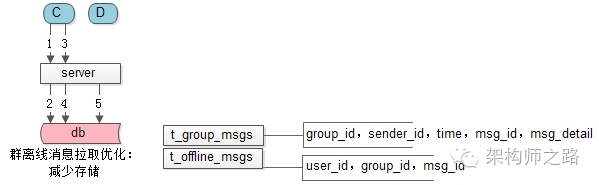
t\_offine\_msgs(user\_id, group\_id, msg\_id)



这样优化后，群在线消息发送就做了一些修改：

步骤3：每次发送在线群消息之前，要先存储群消息的内容

步骤6：每次存储离线消息时，只存储msg\_id，而不用为每个用户存储msg\_detail



拉取离线消息时也做了响应的修改：

步骤1：先拉取所有的离线消息msg\_id

步骤3：再根据msg\_id拉取msg\_detail

步骤5：删除离线msg\_id

**存在的问题**

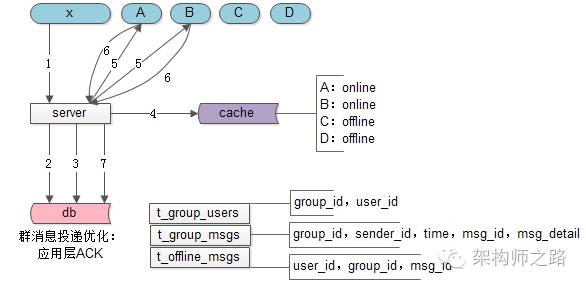
如同单对单消息的发送一样：

（1）在线消息的投递可能出现消息丢失，例如服务器重启，路由器丢包，客户端crash

（2）离线消息的拉取也可能出现消息丢失，原因同上

需要和单对单消息的可靠投递一样，加入应用层的ACK，才能保证群消息一定到达。

**【群消息优化2：应用层ACK】**

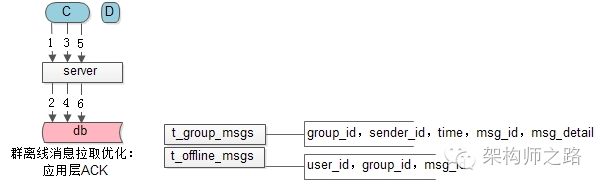


应用层ACK优化后，群在线消息发送又发生了一些变化：

步骤3：在消息msg\_detail存储到群消息表后，不管用户是否在线，都先将msg\_id存储到离线消息表里

步骤6：**在线**的用户A和B收到群消息后，需要增加一个应用层ACK，来标识消息到达

步骤7：在线的用户A和B在应用层ACK后，将他们的离线消息msg\_id删除掉



对应到群**离线**消息的拉取也一样：

步骤1：先拉取msg\_id

步骤3：再拉取msg\_detail

步骤5：最后应用层ACK

步骤6：server收到应用层ACK才能删除离线消息表里的msg\_id

**存在的问题**

（1）如果拉取了消息，却没来得及应用层ACK，会收到重复的消息么？

**回答**：会，可以在客户端去重，对于重复的msg\_id，对用户不展现，从而不影响用户体验

（2）对于离线的每一条消息，虽然只存储了msg\_id，但是每个用户的每一条离线消息都将在数据库中保存一条记录，有没有办法减少离线消息的记录数呢？

**【群消息优化3：离线消息表】**

**离线消息表的优化**

其实，对于一个群用户，在ta登出后的离线期间内，肯定是所有的群消息都没有收到的，完全不用对所有的每一条离线消息存储一个离线msg\_id，而只需要存储最近一条拉取到的离线消息的time（或者msg\_id），下次登录时拉取在那之后的所有群消息即可，而**完全没有必要存储每个人未拉取到的离线消息msg\_id**

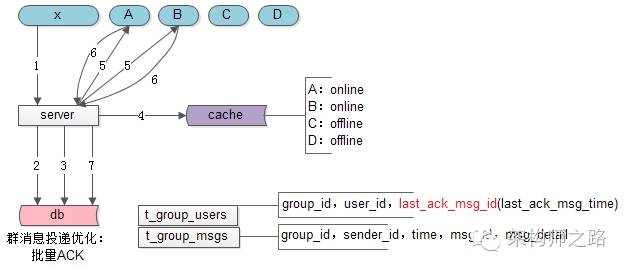
**群成员表**：用来描述一个群里有多少成员，以及每个成员最后一条ack的群消息的msg\_id（或者time）

t\_group\_users(group\_id, user\_id, **last\_ack\_msg\_id(last\_ack\_msg\_time)**)

**群消息表**：用来存储一个群中所有的消息内容，不变

t\_group\_msgs(group\_id, sender\_id, time,msg\_id, msg\_detail)

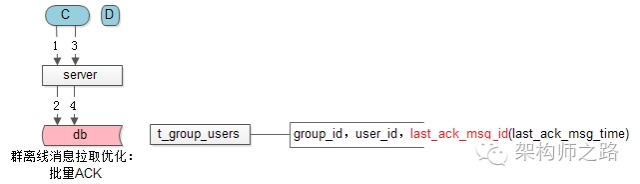
**群离线消息表**：不再需要了



离线消息表优化后，群在线消息的投递流程：

步骤3：在消息msg\_detail存储到群消息表后，不再需要操作离线消息表（优化前需要将msg\_id插入离线消息表）

步骤7：在线的用户A和B在应用层ACK后，将last\_ack\_msg\_id更新即可（优化前需要将msg\_id从离线消息表删除）



群离线消息的拉取流程也类似：

步骤1：拉取离线消息

步骤3：ACK离线消息

步骤4：更新last\_ack\_msg\_id

**存在的问题**

由于“消息风暴扩散系数”的存在，假设1个群有500个用户，“每条”群消息都会变为500个应用层ACK，将对服务器造成巨大的冲击，有没有办法减少ACK请求量呢？

**【群消息优化4：批量ACK】**

由于“消息风暴扩散系数”的存在，如果每条群消息都ACK，会给服务器造成巨大的冲击，为了减少ACK请求量，很容易想到的方法是**批量ACK**。

批量ACK的方式又有两种：

（1）每收到N条群消息ACK一次，这样请求量就降低为原来的1/N了

（2）每隔时间间隔T进行一次群消息ACK，也能达到类似的效果

**新的问题**

批量ACK有可能导致：还没有来得及ACK群消息，用户就退出了，这样下次登录会拉取到重复的离线消息

**解决方案**

msg\_id去重，不对用户展现，保证良好的用户体验

**还可能存在的问题**

群离线消息过多：拉取过慢

**解决方案**：分页拉取（按需拉取）,分页拉取的细节在“微信为啥不丢离线消息”一章中有详细叙述，此处不再展开（详见《[微信为啥不丢“离线消息”？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959629&idx=1&sn=053d85a862df6e6c01147a1cf95bdbf2&chksm=bd2d04918a5a8d87a32305fa8ef1603bb0e73e5c4b78afb5a144327841e08840188c4541b4aa&scene=21" \l "wechat_redirect" \t "_blank)》）。

**【总结】**

群消息还是非常有意思的，**可达性、实时性、离线消息、消息风暴扩散**等等等等，做个总结：

（1）不管是群在线消息，还是群离线消息，应用层的ACK是可达性的保障

（2）群消息只存一份，不用为每个用户存储离线群msg\_id，只需存储一个最近ack的群消息id/time

（3）为了减少消息风暴，可以批量ACK

（4）如果收到重复消息，需要msg\_id去重，让用户无感知

（5）离线消息过多，可以分页拉取（按需拉取）优化

==【完】==

回【http】[http如何像tcp一样实时的收消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959605&idx=1&sn=21f25087bef3c3a966ef03b824365621&scene=21" \l "wechat_redirect" \t "_blank)

回【可靠】[微信为什么不丢消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959606&idx=1&sn=f9561231dd33bcd0550b8d0d59d6b876&chksm=bd2d04ea8a5a8dfce90c870279a7f74b7aedd802c2d699dd919d7e40ebe30699381517c2d54b&scene=21" \l "wechat_redirect" \t "_blank)

回【状态】[QQ状态同步究竟是推还是拉？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959618&idx=1&sn=912a9af6d000c6681dc645e53590729b&chksm=bd2d049e8a5a8d884f6acb35bc5b40edf88127219542c9e3033bb4c2e33854657c315c8a93b7&scene=21" \l "wechat_redirect" \t "_blank)

回【离线】[微信为啥不丢“离线消息”？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959629&idx=1&sn=053d85a862df6e6c01147a1cf95bdbf2&chksm=bd2d04918a5a8d87a32305fa8ef1603bb0e73e5c4b78afb5a144327841e08840188c4541b4aa&scene=21" \l "wechat_redirect" \t "_blank)

## 微信多点登录与QQ消息漫游架构随想

原创 2016-11-10 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959651&idx=1&sn=959540719a8336bd6f507a324dc7219a&chksm=bd2d04bf8a5a8da9ddfaeb106b61386adeb51c2ad7d4a32589c9a9e113f9341604a490eebb98&scene=21" \l "#)

**【需求缘起】**

之前的一些文章简单介绍了

《[“单人消息”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959606&idx=1&sn=f9561231dd33bcd0550b8d0d59d6b876&chksm=bd2d04ea8a5a8dfce90c870279a7f74b7aedd802c2d699dd919d7e40ebe30699381517c2d54b&scene=21" \l "wechat_redirect" \t "_blank)》《[“离线消息”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959629&idx=1&sn=053d85a862df6e6c01147a1cf95bdbf2&chksm=bd2d04918a5a8d87a32305fa8ef1603bb0e73e5c4b78afb5a144327841e08840188c4541b4aa&scene=21" \l "wechat_redirect" \t "_blank)》《[“群消息”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959643&idx=1&sn=844afa6a31770fa587474ecd73c3b3b3&chksm=bd2d04878a5a8d91c5c93ad8e85254185c63eb419457efbedcba54a9b8a9053da1e8980a694a&scene=21" \l "wechat_redirect" \t "_blank)》《[“用户状态”](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959618&idx=1&sn=912a9af6d000c6681dc645e53590729b&chksm=bd2d049e8a5a8d884f6acb35bc5b40edf88127219542c9e3033bb4c2e33854657c315c8a93b7&scene=21" \l "wechat_redirect" \t "_blank)》的一些相关技术（点击上面的link直接阅读），今天来聊一聊“多点登陆”与“消息漫游”。

**提问：什么是多点登录？**

**回答**：以微信为例，可以PC端，phone端同时登录，同时收发消息。

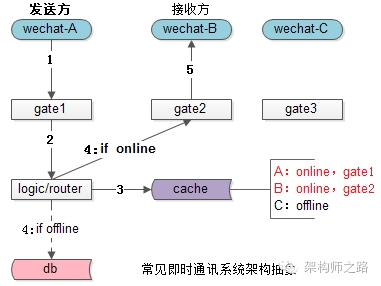
需要注意的是，一个端只能登录一个实例，例如同一个QQ号，在pc1上登录，再到pc2上登录，后者会把前者踢出，pc1会收到通知“你已在别处登录xxoo”。

**提问：什么是消息漫游？**

**回答**：在任何一个终端的任何一个实例登录qq，都能够拉取到所有历史聊天消息，这个就是消息漫游。

微信目前只支持“多点登录”同时收发在线消息，没有实现“消息漫游”，潜台词是：登出手机微信，登录PC微信，聊天，再登录手机微信，是看不到历史消息的。

**【架构回顾】**



整个即时通讯架构可以抽象成这么几层：

（1）**客户端**：例如pc微信，手机qq

（2）服务端：

（2.1）**入口层gate集群**：能够水平扩展，保持与客户端的连接

（2.2）**逻辑层logic、路由层router集群**：高可用可扩展，实现业务逻辑，进行消息的路由

（2.3）**cache**：高可用cache集群，用来存储用户的在线状态，与接入节点（用户具体连接在哪个gate节点）

（2.4）**db**：固化存储消息，群信息，好友关系链等信息

一个**典型的消息投递流程**如上图步骤1-5：

（1）用户A登录在gate1上，发出消息

（2）gate1将消息给logic/router

（3）logic/router查询接收方的在线状态（B在线，C不在线）

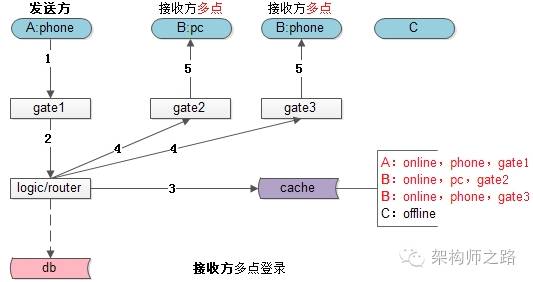
（4）例如接收方C不在线，存储离线

（4）例如接收方B在线，且登录在gate2上，消息投递给gate2

（5）gate2将消息投递给B

当然，单对单消息有一系列应用层超时、重传、确认、去重的机制，这不是本文的重点，不进行展开，细节详见《[微信为啥不丢消息](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959606&idx=1&sn=f9561231dd33bcd0550b8d0d59d6b876&chksm=bd2d04ea8a5a8dfce90c870279a7f74b7aedd802c2d699dd919d7e40ebe30699381517c2d54b&scene=21" \l "wechat_redirect" \t "_blank)》。

**【接收方多点登陆】**



**接收方多点登录**，pc也登录，phone也登录，后一端登录不会将前一端踢出，cache中存储状态与登录点时，不再以user\_id为key，改为以user\_id+终端类型为key即可。

B：online(状态)，gate2(登录点)

改为

B+pc：online(状态)，gate2(登录点)

B+phone：online(状态)，gate3(登录点)

当用户A给用户B发送消息时，取出所有B的登录点，进行消息群发即可（如上图中步骤4与步骤5）。

**【发送方多点登陆】**

有朋友可能要问，**发送方和多点登录有什么关系？**

假设用户A登录了两个点，A1和A2；用户B登录了两个点B1和B2

A(A1发出的)发送消息给B(B1和B2)

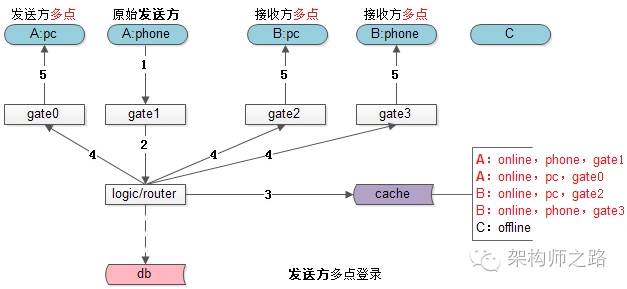
B(B1发出的)发送消息给A(A1和A2)

不就可以了么？

其实不然，A(A1发出的)发送消息给B(B1和B2)，B(B1发出的)发送消息给A(A1和A2)

A2端虽然收到了所有B回复的消息，但消息其实是在A1端发出的，故A2端只知道聊天消息的一半(所有B的回复)，**缺失了聊天的上下文**(所有A1端的发出)

故，**如果发送方也进行了多点登录**，发送出去的任何消息，除了要投递给多点登录的接收方，**还需要投递给多点登录的发送方**。



如上图，发送方A和接收方B都进行了多点登陆，cache中存储的信息为：

A+pc：online(状态)，gate0(登录点)

A+phone：online(状态)，gate1(登录点)

B+pc：online(状态)，gate2(登录点)

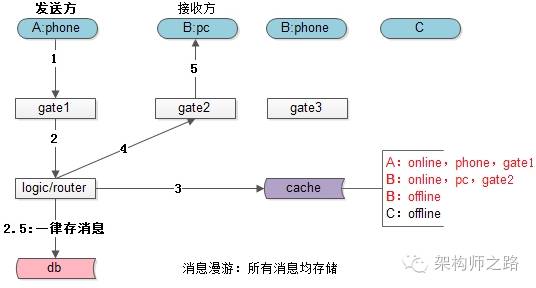
B+phone：online(状态)，gate3(登录点)

当用户A（phone端）给用户B发送消息时，除了要投递给B的所有多点登录端，还需要投递给A多点登陆的其他端（pc端），如上图中步骤4与步骤5。

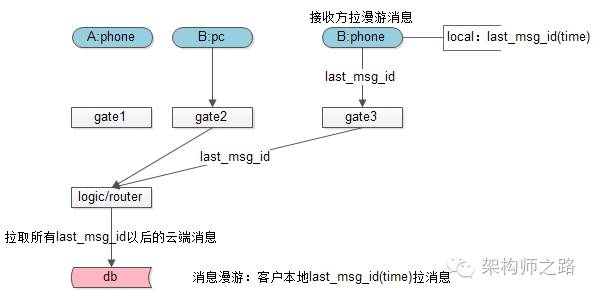
只有这样，才能在所有用户的所有端，恢复与还原双方聊天的上下文。

**【消息漫游】**

如果业务不需要支持“消息漫游”的功能，对于在线消息，如果用户接收到，是不需要存储到数据库的。但如果要支持“换一台机器也能看到历史的聊天消息”，就需要**对所有消息进行存储**了。



消息投递如上图，用户A发送消息给用户B，虽然B在线，仍然要增加一个步骤2.5，在投递之前进行存储，以备B的其他端登陆时，可以拉取到历史消息。



消息拉取如上图，原本不在线的B(phone端)，又重新登录了，ta怎么拉取历史消息？只需要在客户端本地存储一个上一次拉取到的msg\_id(time)，到服务端重新拉取即可。

这里还有个问题，由于服务端存储所有消息成本是非常高的，所以**一般“消息漫游”是有时间（或者消息数）限制**，不能拉取所有所有几年前的历史消息，只能拉取3个月内的云端消息。

**【总结】**

**“多点登录”**是指多个端同时登录一个帐号，同时收发消息，关键点是：

（1）需要在**服务端存储同一个用户多个端的状态与登陆点**

（2）发出消息时，**要对发送方的多端与接收端的多端，都进行消息投递**

**“消息漫游”**是指一个用户在任何端，都可以拉取到历史消息，关键点是：

（1）**所有消息存储在云端**

（2）**每个端本地存储last\_msg\_id**，在登录时可以到云端同步历史消息

（3）云端存储所有消息成本较高，一般会对**历史消息时间(或者条数)进行限制**

==【完】【帮忙转发】==

回【http】[http如何像tcp一样实时的收消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959605&idx=1&sn=21f25087bef3c3a966ef03b824365621&scene=21" \l "wechat_redirect" \t "_blank)

回【可靠】[微信为什么不丢消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959606&idx=1&sn=f9561231dd33bcd0550b8d0d59d6b876&chksm=bd2d04ea8a5a8dfce90c870279a7f74b7aedd802c2d699dd919d7e40ebe30699381517c2d54b&scene=21" \l "wechat_redirect" \t "_blank)

回【状态】[QQ状态同步究竟是推还是拉？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959618&idx=1&sn=912a9af6d000c6681dc645e53590729b&chksm=bd2d049e8a5a8d884f6acb35bc5b40edf88127219542c9e3033bb4c2e33854657c315c8a93b7&scene=21" \l "wechat_redirect" \t "_blank)

回【群】[群消息这么复杂，怎么能做到不丢不重？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959643&idx=1&sn=844afa6a31770fa587474ecd73c3b3b3&chksm=bd2d04878a5a8d91c5c93ad8e85254185c63eb419457efbedcba54a9b8a9053da1e8980a694a&scene=21" \l "wechat_redirect" \t "_blank)

回【离线】[微信为啥不丢“离线消息”？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959629&idx=1&sn=053d85a862df6e6c01147a1cf95bdbf2&chksm=bd2d04918a5a8d87a32305fa8ef1603bb0e73e5c4b78afb5a144327841e08840188c4541b4aa&scene=21" \l "wechat_redirect" \t "_blank)

## QQ状态同步究竟是推还是拉？

原创 2016-10-29 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959618&idx=1&sn=912a9af6d000c6681dc645e53590729b&chksm=bd2d049e8a5a8d884f6acb35bc5b40edf88127219542c9e3033bb4c2e33854657c315c8a93b7&scene=21" \l "#)

前面两篇讲即时通讯核心技术的文章

《[微信为什么不丢消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959606&idx=1&sn=f9561231dd33bcd0550b8d0d59d6b876&chksm=bd2d04ea8a5a8dfce90c870279a7f74b7aedd802c2d699dd919d7e40ebe30699381517c2d54b&scene=21" \l "wechat_redirect" \t "_blank)》

《[http如何像tcp一样实时的收消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959605&idx=1&sn=21f25087bef3c3a966ef03b824365621&scene=21" \l "wechat_redirect" \t "_blank)》

反馈还可以，故继续即时通讯这一个系列吧，今天聊聊即时通讯中的“状态”。

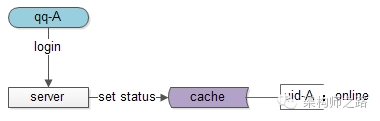
**需求缘起**

“在线状态一致性”（好友在线状态，群友在线状态）是即时通讯领域较难解决的一个技术问题，如何精准实时的获得好友、群友的在线状态，是今天将要探讨的话题。

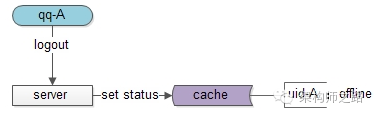
**好友状态一致性**

**问题一：用户uid-A登录时，如何获取自己全部好友的在线状态？**

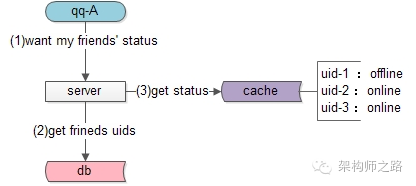
回答：



（1）服务器要存储所有用户的在线状态（往往存储在保证高可用的缓存集群里） -> **保证状态可查**



（2）用户状态实时变更，任何用户登录时，需要将服务端自己的在线状态置为online；任何用户登出时，需要将服务端自己的状态置为offline -> **保证服务端状态存储的一致性与实时性**



（3）uid-A登录时，先去数据库拉取自己的好友列表，再去缓存获取所有好友的在线状态 -> **保证登录时好友状态获取的一致性与实时性**

**问题二：用户uid-A的好友uid-B状态改变时（由登录、登出、隐身等动作触发），uid-A如何知道这一事件？**

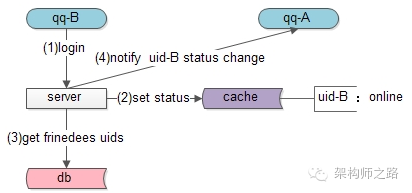
**方案一**：uid-A向服务器轮询**拉取**uid-B（其实是自己的全部好友）的状态，例如每1分钟一次

**缺点**：

（1）如果uid-B的状态改变，uid-A获取不实时，可能有1分钟时延

（2）如果uid-B的状态不改变，uid-A会有大量无效的轮询请求，占用服务器资源

**方案二**：uid-B状态改变时（由登录、登出、隐身等动作触发），服务器不仅在缓存中修改uid-B的状态，还要将这个状体改变的通知**推送**给uid-B的**在线**反向好友（反向好友是指：加了uid-B为好友的人，而不是uid-B的好友，这个细节要注意）



**优点**：

（1）实时

**缺点**：

（2）当在线好友量很大时，任何一个用户状态的改变，会扩散成N个实时通知，这个N叫做“**消息风暴扩散系数**”。

假设一个im系统平均每个用户有200个反向好友，平均有20%的反向好友在线，那么消息风暴扩散系数N=40，这意味着，任何一个状态的变化会变成40个推送请求。

**群友状态一致性**

**问题三：群友状态一致性有什么不同，和好友状态一致性相比复杂在哪里？为什么不能采用实时推送？**

回答：

理论上群友状态也可以通过实时推送的方式实现，以保证实时性。但实际上，群友状态一般都是采用**拉取**的方式获得，因为群友状态“消息风暴扩散系数”N实在太大，全部实时获取系统往往承受不了。

假设平均每个用户加了20个群，平均每个群有200个用户，依然假设20%的用户在线，那么为了保证群友状态的实时性，每个用户登录，就要将自己的状态改变通知发送给20\*200\*20%=800个群友，N=800，意味着，任何一个状态的变化会变成800个推送请求。

XXX系统使用的是群友状态推送，不存在的这样的问题？那很可能是，XXX系统的用户量和活跃度还不够高吧。

**问题四：轮询拉取群友状态也会给服务器带来过大的压力，还有什么优化方式？**

回答：

群友的数据量太大，虽然每个用户平均加入了20个群，但实际上并不会每次登录都进入每一个群。**不采用轮询拉取，而采用按需拉取，延时拉取**的方式，在真正进入一个群时才实时拉取群友的在线状态，是既能满足用户需求（用户感觉是状态是实时、一致的，但其实是进入群才拉取的），又能降低服务器压力。这是一种常见方法。

关于更多按需拉取，延时拉取的讨论，可移步《[微信为啥这么省流量](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400163013&idx=1&sn=911cf71925e2ba50d47955a713134acb&scene=21" \l "wechat_redirect" \t "_blank)》。

**延伸讨论：系统消息/开屏广告的推送与拉取**

**问题五：系统消息/开屏广告一般采用推送还是拉取？**

回答：

不考虑APP端的push（APP端的push，不需要启动APP，不依赖client与server之间的TCP长连接），个人强烈建议系统消息/开屏广告这类消息采用“拉取”的方式，原因是：

（1）这类业务**对消息的实时性往往要求不高**

（2）如果集中推送，“消息风暴扩散系数”过大，容易引发系统抖动；而**拉取的方式，可以摊平这个抖动，用户登录时均匀的发起请求**

（3）如果集中推送，往往不在意用户是否“在线”，往往会造成大量离线垃圾消息；**而拉取的方式，保证只有在线的用户才会收到请求**

（4）…

有不同的建议，欢迎评论讨论。

**总结与建议**

状态的实时性与一致性是一个较难解决的技术问题，不同的业务接受度，不同的数据量并发量在线量，实现方式不同，个人建议的方式是：

（1）**好友状态**，如果对实时性要求较高，可以采用推送的方式同步；如果实时性要求不高，可以采用轮询拉取的方式同步

（2）**群友的状态**，由于消息风暴扩散系数过大，可以采用按需拉取，延时拉取的方式同步

（3）**系统消息/开屏广告**等对实时性要求不高的业务，可以采用拉取的方式获取消息

（4）“**消息风暴扩散系数**”是指一个消息发出时，变成N个消息的扩散系数，这个系数与业务及数据相关，一定程度上它的大小决定了技术采用推送还是拉取

如果大伙有收获，求帮转

今天推一个朋友：江南白衣

大家都是原创技术文章，不容易，希望大家支持

==【完】==

回【服务】[互联网架构为什么要做服务化？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959519&idx=1&sn=065074b135fc9cb243abe897261e1a72&scene=21" \l "wechat_redirect" \t "_blank)

回【粒度】[微服务架构多“微”才合适？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959530&idx=1&sn=ff84bd74745eee1577e7dfef8ce66bbe&scene=21" \l "wechat_redirect" \t "_blank)

回【rpc】[为什么说要搞定微服务架构，先搞定RPC框架？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959553&idx=1&sn=c1084e91875721c5f6baf544450afa38&scene=21" \l "wechat_redirect" \t "_blank)

回【收发】[RPC-client异步收发核心细节？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959576&idx=1&sn=2be8d3f61effe7118abf920a175da710&scene=21" \l "wechat_redirect" \t "_blank)

回【均衡】[一分钟了解负载均衡的一切](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959585&idx=1&sn=0a9222cbfeb62a662edffafb7f0b43ae&scene=21" \l "wechat_redirect" \t "_blank)

回【接入】[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)

## 消息“时序”与“一致性”为何这么难？

原创 2016-11-15 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959666&idx=1&sn=fbdce26e40296d5b30f70915c4b9eb0a&chksm=bd2d04ae8a5a8db868df14904d0a1ddb4eb4d8411442f5529f9760e7c62fca586cfa6f3bd200&scene=21" \l "#)

分布式系统中，很多业务场景都需要考虑消息投递的时序，例如：

（1）单聊消息投递，保证发送方发送顺序与接收方展现顺序一致

（2）群聊消息投递，保证所有接收方展现顺序一致

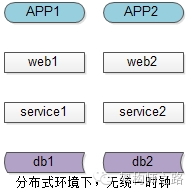
（3）充值支付消息，保证同一个用户发起的请求在服务端执行序列一致

消息时序是分布式系统架构设计中非常难的问题，ta**为什么难**，有什么常见**优化实践**，是本文要讨论的问题。

**一、为什么时序难以保证，消息一致性难？**

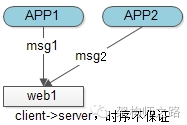
为什么分布式环境下，消息的时序难以保证，这边简要分析了几点原因：

**【时钟不一致】**



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

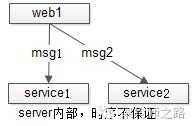
**【多客户端（发送方）】**



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

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

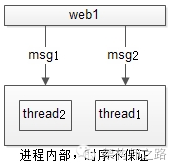
**【服务集群（多接收方）】**



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

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

**【网络传输与多线程】**



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

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

**【怎么保证绝对时序】**

通过上面的分析，假设只有一个发送方，一个接收方，上下游连接只有一条连接池，通过阻塞的方式通讯，难道不能保证先发出的消息msg1先处理么？

回答：可以，但**吞吐量会非常低**，而且单发送方单接收方单连接池的假设不太成立，高并发高可用的架构**不会允许这样的设计出现**。

**二、优化实践**

**【以客户端或者服务端的时序为准】**

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

（1）**邮件展示顺序**，其实是以客户端发送时间为准的，潜台词是，发送方只要将邮件协议里的时间调整为1970年或者2970年，就可以在接收方收到邮件后一直“置顶”或者“置底”

（2）**秒杀活动时间判断**，肯定得以服务器的时间为准，不可能让客户端修改本地时间，就能够提前秒杀

**【服务端能够生成单调递增的id】**

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

**【大部分业务能接受误差不大的趋势递增id】**

消息发送、帖子发布时间、甚至秒杀时间都没有这么精准时序的要求：

（1）同1s内发布的聊天消息时序乱了

（2）同1s内发布的帖子排序不对

（3）用1s内发起的秒杀，由于服务器多台之间时间有误差，落到A服务器的秒杀成功了，落到B服务器的秒杀还没开始，业务上也是可以接受的（用户感知不到）

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

关于绝对递增id，趋势递增id的生成架构，详见文章《[细聊分布式ID生成方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "wechat_redirect" \t "_blank)》，此处不展开。

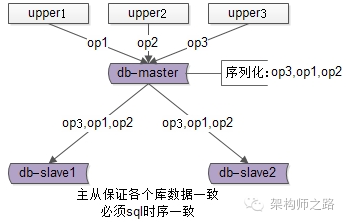
**【利用单点序列化，可以保证多机相同时序】**

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

（1）先在一台机器上序列化操作

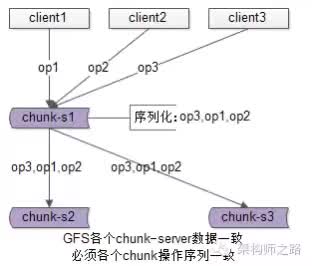
（2）再将操作序列分发到所有的机器，以保证多机的操作序列是一致的，最终数据是一致的

**典型场景一：数据库主从同步**



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

**典型场景二：GFS中文件的一致性**



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

**【单对单聊天，怎么保证发送顺序与接收顺序一致】**

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

回答：

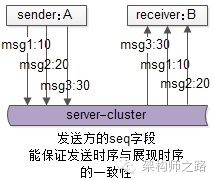
（1）如果利用服务器单点序列化时序，可能出现服务端收到消息的时序为msg3，msg1，msg2，与发出序列不一致

（2）业务上不需要全局消息一致，只需要对于同一个发送方A，ta发给B的消息时序一致就行，常见优化方案，在A往B发出的消息中，加上发送方A本地的一个绝对时序，来表示接收方B的展现时序

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

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

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



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

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

总之，需要一杆标尺来衡量这个时序。

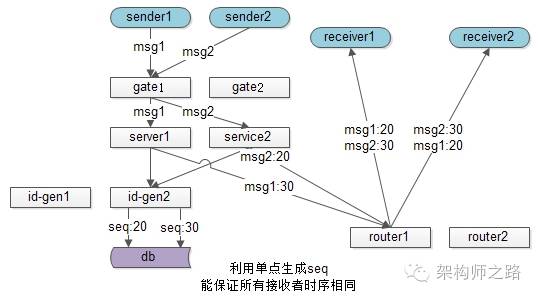
**【群聊消息，怎么保证各接收方收到顺序一致】**

群聊消息的需求，N个群友在一个群里聊，怎么保证所有群友收到的消息显示时序一致？

回答：

（1）不能再利用发送方的seq来保证时序，因为发送方不单点，时间也不一致

（2）可以利用服务器的单点做序列化



此时群聊的发送流程为：

（1）sender1发出msg1，sender2发出msg2

（2）msg1和msg2经过接入集群，服务集群

（3）service层到底层拿一个唯一seq，来确定接收方展示时序

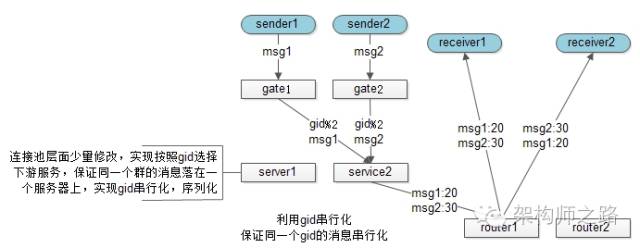
（4）service拿到msg2的seq是20，msg1的seq是30

（5）通过投递服务讲消息给多个群友，群友即使接收到msg1和msg2的时间不同，但可以统一按照seq来展现

这个方法能实现，所有群友的消息展示时序相同。

缺点是，这个生成全局递增序列号的服务很容易成为系统瓶颈，还有没有进一步的**优化方法呢**？

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



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

关于id串行化的细节，可详见《[利用id串行化解决缓存与数据库一致性问题](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404202261&idx=1&sn=1b8254ba5013952923bdc21e0579108e&scene=21" \l "wechat_redirect" \t "_blank)》，此处不展开。

**三、总结**

（1）分布式环境下，**消息的有序性是很难**的，原因多种多样：时钟不一致，多发送方，多接收方，多线程，网络传输不确定性等

（2）**要“有序”，先得有衡量“有序”的标尺**，可以是客户端标尺，可以是服务端标尺

（3）**大部分业务能够接受大范围趋势有序，小范围误差**；绝对有序的业务，可以借助服务器绝对时序的能力

（4）**单点序列化**，是一种常见的保证多机时序统一的方法，典型场景有db主从一致，gfs多文件一致

（5）**单对单聊天**，只需保证发出的时序与接收的时序一致，可以利用客户端seq

（6）**群聊**，只需保证所有接收方消息时序一致，需要利用服务端seq，方法有两种，一种**单点绝对时序**，另一种**id串行化**

==【帮转下】==

回【http】[http如何像tcp一样实时的收消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959605&idx=1&sn=21f25087bef3c3a966ef03b824365621&scene=21" \l "wechat_redirect" \t "_blank)

回【可靠】[微信为什么不丢消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959606&idx=1&sn=f9561231dd33bcd0550b8d0d59d6b876&chksm=bd2d04ea8a5a8dfce90c870279a7f74b7aedd802c2d699dd919d7e40ebe30699381517c2d54b&scene=21" \l "wechat_redirect" \t "_blank)

回【状态】[QQ状态同步究竟是推还是拉？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959618&idx=1&sn=912a9af6d000c6681dc645e53590729b&chksm=bd2d049e8a5a8d884f6acb35bc5b40edf88127219542c9e3033bb4c2e33854657c315c8a93b7&scene=21" \l "wechat_redirect" \t "_blank)

回【群】[群消息这么复杂，怎么能做到不丢不重？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959643&idx=1&sn=844afa6a31770fa587474ecd73c3b3b3&chksm=bd2d04878a5a8d91c5c93ad8e85254185c63eb419457efbedcba54a9b8a9053da1e8980a694a&scene=21" \l "wechat_redirect" \t "_blank)

回【qq】[微信多点登录与QQ消息漫游架构](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959651&idx=1&sn=959540719a8336bd6f507a324dc7219a&chksm=bd2d04bf8a5a8da9ddfaeb106b61386adeb51c2ad7d4a32589c9a9e113f9341604a490eebb98&scene=21" \l "wechat_redirect" \t "_blank)

回【离线】[微信为啥不丢“离线消息”？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959629&idx=1&sn=053d85a862df6e6c01147a1cf95bdbf2&chksm=bd2d04918a5a8d87a32305fa8ef1603bb0e73e5c4b78afb5a144327841e08840188c4541b4aa&scene=21" \l "wechat_redirect" \t "_blank)

## 58到家通用实时消息平台架构细节（Qcon2016）

原创 2016-04-23 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "#)

2016Qcon北京，业务核心架构场，《58到家通用实时消息平台架构细节》。

**一、解决什么问题 + 难点**

解决什么业务问题

（1）**端到云**的实时上报需求：58速运司机端GPS实时上报

（2）**云到端**的实时推送需求：58速运司机订单实时推送

（3）**端到端**的聊天消息需求：用户、商户、客服之间的聊天沟通

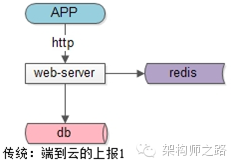
难点：

（1）APP无线环境下消息可达性

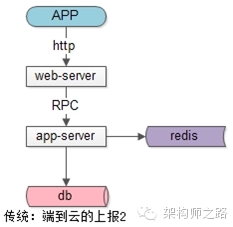
（2）通用性，平台实现尽量与业务解耦

**二、传统解决方案与潜在不足**

**【端到云：http轮询上报GPS消息】**



**方案一**：直接通过业务线web-server写DB



**方案二**：通用web-server层调用业务服务层写DB

**潜在不足**：

（1）http短连接代价高（反复创建与销毁连接）

（2）web-server层吞吐量较低（每秒处理千级别请求）

**【云到端：通过第三方push或者推送服务】**

**方案一**：通过APNs或者米推等第三方推送

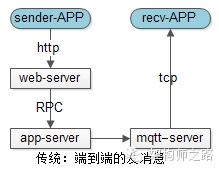
**方案二**：通过自己搭建mqtt服务推送

**潜在不足**：

（1）第三方可达性与实时性无法保证，第三方会进行推送限速

（2）mqtt可用性是个问题

**【端到端：结合上面两种方法实现】**

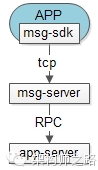


传统方案往往可以通过结合【端到云】与【云到端】来结合解决【端到端】的实时消息推送问题。

**三、通用实时消息平台实现细节**

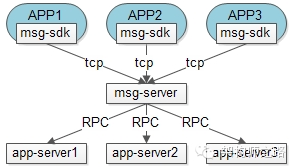
业务的分析与抽象：司机、用户、商家、客服均为**“在线”业务**

**【端到云的优化】**



**传统方案潜在的问题**：http轮询效率不高，web-server性能有限

**优化TIPS**：消息平台使用tcp长连接（如上图）



**潜在的问题**：消息平台与业务线app-server耦合，需要switch case业务线类型来分发投递消息，新增业务线需要新增RPC调用（如上图）

**优化TIPS**：使用消息总线msg-queue解耦（如下图）

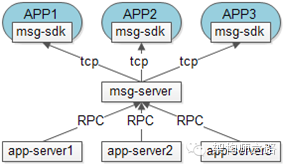


可以看到，使用消息总线后，新增消息发送方，消息平台只需要配置消息类型与消息总线主题的映射关系，新增的app-server消费方订阅新的主题即可，实现消息平台与业务的解耦。

**【云到端的优化】**

**潜在的问题**：可用性问题与第三方限速

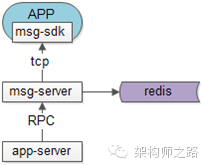
**优化TIPS**：自己提供消息平台集群，提供RPC接口，实现“云到端”的消息通道



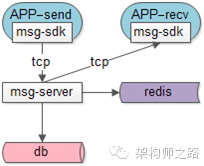
这里要注意的是，“端到云”使用消息总线，是为了业务解耦。“云到端”直接使用RPC接口，也是为了业务解耦，新增消息推送方，消息平台无需改动代码。

**潜在的问题**：不少司机推送订单无回复，抢单率比预期的低

**优化TIPS**：引入状态实时存储，只有“在线”状态的用户才推送消息



**【端到端的优化】**



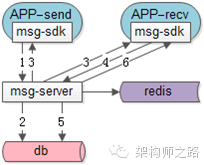
如果业务无关，则直接通过tcp通道投递；如果业务相关，**发送方**先来一个“端到云”的投递（通过mq），业务服务器处理再反向来一个“云到端”的投递（RPC）给**接收方**。

**潜在问题**：如果接收方不在线怎么办

**优化TIPS**：增加DB存储离线消息

**潜在的问题**：无线环境下经常网络不稳（例如进出电梯断网），消息经常丢失

**优化TIPS**：消息平台收到消息先落地数据库，接收方收到后应用层ACK再删除，以保证不丢失



如上图（本文最重要的2张图之一），整个消息投递流程为：

（1）发送发将消息发给消息平台

（2）消息平台先将消息落地DB

（3）消息平台回复发送方消息发送成功（此时和接收方是否接到无关）

（3）与此同时，并行的把消息投递给接收方（如果不在线就存离线了）

（4）接收方应用层ACK表示收到了消息

（5）消息平台将消息删除

（6）告之接收方ACK已经成功处理

可以看到，是使用“应用层ACK来解决消息可达性问题的”

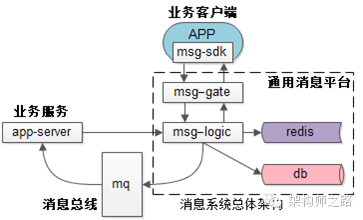
**潜在问题**：发送方没有收到第3步骤中的消息平台回复怎么办？

**优化TIPS**：发送方重发（服务器无状态）

**潜在问题**：接收方收到重发的冗余消息怎么办？

**优化TIPS**：接收方去重（可以做到服务端完全无状态，只需要简单投递消息即可）

**【分层架构说明】**



整个系统的分层架构如上图（本文最重要的2张图之二），整个**消息平台**系统由：

（1）消息平台在APP里的msg-sdk，向APP提供帅气的接口

（2）msg-gate，整个消息平台的tcp接入门户，保持tcp长连接，初步攻防，加解密，压缩解压缩

（3）msg-logic，整个消息平台逻辑处理的部分

（4）redis，高可用redis集群存储用户在线状态online/offline，以及用户在哪一台msg-gate接入（如果在线）

（5）DB，存储离线消息

非消息平台的几个**业务部分**：

（1）APP：业务方APP，可以有多个，通过msg-sdk来接入消息平台

（2）mq：消息平台通过mq来给业务方服务器发“端到云”的消息

（3）app-server：业务方后端，可以有多个，通过mq接收“端到云”的消息，通过RPC发送“云到端”的消息

**【对外提供的接口说明】**

消息平台对业务方提供的接口是**很少很通用**的接口。

**msg-sdk对APP提供**的核心接口有：

（1）login：接入消息平台

（2）logout：登出消息平台

（3）c2s：发送client to server“端到云”的消息

（4）c2c：发送client to client“端到端”的消息

（5）get-offline-msg：拉取离线消息

（6）on-msg-recieved：收到消息的callback回调接口

**消息平台对app-server提供**的核心接口有：

（1）s2c：发送server to client“云到端”的消息

其他业务方不需要关注，是**msg-sdk与消息平台之间的内部接口**有：

（1）keepalive：用于msg-sdk与消息平台的连接保持（对业务方透明）

（2）c2c-ack：用户c2c接口的应用层ack接口（对业务方透明）

**【如何实现跨帐号体系的聊天】**

既然是通用的消息平台，如何实现跨帐号体系的消息发送呢（即如何实现qq与旺旺的聊天）？

**解决方案**：不再使用uid作为整个系统运行的key，而使用domain+uid，或者appid+uid来作为整个系统运行的key

**潜在耦合点**：这样的话，login接口的逻辑处理，消息平台需要switch case (domain或者appid)来进行不同的登录验证，与业务有一定的耦合，不过新增帐号体系的频度很低，远比新增消息类型低

**【协议的扩展性设计】**

APP本质是cs架构，一旦放出去的版本就很难收回来，其兼容系要求远比bs架构难，如何做到新增功能的同时，还能方便的兼容历史旧版APP呢？

**（1）如何方便的增加接口？**

**解决方案**：协议使用定长包头 + 变长包体，使用命令号cmd来扩展新接口【这个变化对业务层是透明的，是msg-sdk与消息平台之间的事情】

**（2）对于同一个接口，能否增加参数，而不影响旧版本的APP？**

**解决方案**：使用可扩展的序列化协议，例如protobuffer【protobuffer这个东西也对业务线透明】

**（3）对于业务方，有很多种类的消息类型，有很多复杂的业务需求，如何保证业务扩展性的同时，又不会增加消息平台的复杂性，并对旧版本APP兼容？**

例如业务线可能有这样的潜在需求：

a）推送一个运营消息

b）推送消息内容支持字体、字号、加粗、颜色

c）推送消息支持图片

d）业务支持“窗口震动”，以及“对方正在输入......”等需求

**解决方案**：使用可扩展的消息体协议（对消息平台透明），例如xml/json来支持可扩展的多样消息类型，并对旧版本APP兼容

<msg>

<type>1</type>

<fond>宋体</font>

<content>hello, world!</content>

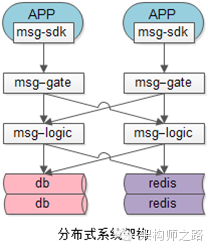
<pic>http://pic.daojia.com/hello.jpg</pic>

</msg>

使用这种消息内容协议，能保证：**扩展性好、旧版本兼容、对消息平台透明**等诸多好处，强烈建议使用

**四、分布式架构细节**

抱歉，主持人提醒时间已到，分布式架构扩展性、负载均衡性、可用性、一致性的问题线下和大家分享，先放一个分布式架构图吧：



<广告>

如果你读到了这里，说明对消息平台还是有点兴趣的，58到家消息平台团队只有一个同学，**奇缺人，欢迎有志之士加入**。

**一些说明**：base北京，核心团队，技术导向，Java方向，和58沈剑一个团队

**加入方式**：（1）直接在文章下面留言（2）在公众号回复“招聘”获取加盟密钥

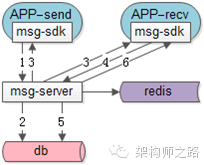
</广告完毕>

**五、总结**

（1）**“端到云”消息投递**：TCP消息通道，消息总线业务解耦

（2）**“云到端”消息投递**：提供RPC接口，引入状态存储

（3）**“端到端”消息投递步骤如下图**：



（4）**“端到端”消息投递技巧**

        a）先存离线消息防丢失

        b）ACK机制保证可达

        c）发送方消息重发

        d）接收方消息去重

（5）**可扩展协议设计**

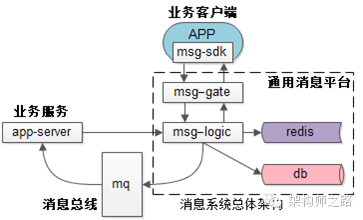
        a）定长包头，变长包体，随时增加接口

        b）可扩展序列化协议，随时变化接口

        c）可扩展消息协议，随时增加类型

（6）**支持跨帐号体系聊天（多个域）**：使用domain（或者appid）+uid作为综合key

（7）**分层架构如下图**



==【完】==

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

回【秒杀】[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959391&idx=1&sn=fb28fd5e5f0895ddb167406d8a735548&scene=21" \l "wechat_redirect" \t "_blank)

回【晋升】[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)

回【id】[细聊分布式ID生成方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

## 微信为啥这么省流量？

原创 2016-11-19 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959676&idx=1&sn=0204f47f3ecad9034d0a3607914d25c3&chksm=bd2d04a08a5a8db61f7d98d811b703847a5c7e04356b70fbd7ae741848eb427ed17d31011ae3&scene=21" \l "#)

**前言**：“客户端上传时间戳”的玩法，你玩过么？一起聊聊时间戳的奇技淫巧！

**缘起**：无线时代，流量敏感。APP在登录后，往往要向服务器同步非常多的数据，很费流量，技术上有没有节省流量的方法呢？这是本文要讨论的问题。

**问题一：APP登录时需要拉取什么数据？**

答：APP登陆时，一般要拉取两类数据，一类是“**id列表型数据**”，一类是“**信息详情型数据**”，以微信为例，需要拉取

（1）好友列表List<user-id>，即所有好友的id（id+name）

（2）群组列表List<group-id>，即所有加入群的id（id+name）

（3）群友列表Map<group-id, List<group-user-id>>，所有群友的id（id+name）

（4）好友详情Map<user-id, User>，所有好友的详情（昵称，备注，标签，地区，相册等）

（5）群组详情Map<group-id, Group>，所有群组的详情（二维码，公告，是否免打扰等）

（6）群友详情Map<group-id, Map<user-id, User>>，所有群友的详情（昵称，备注，标签，地区，相册等）

（7）其他，例如离线消息…

**问题二：能不能在登录的过程中不拉取这些数据，而在登录后拉取？**

答：如果登录时不拉取，登陆后刷好友列表，刷群列表，群成员会很慢。

如果登录时拉取，登陆过程可能会很慢（微信的“大月亮背景”要等多长时间？QQ登录要等30s？）。

为了保证登录后的体验，一般是在登录过程中拉取。

**问题三：能不能直接复用客户端本地的数据？**

答：不能直接复用客户端本地的数据，因为不能确保本地的数据是最新的。

**核心问题：每次登录都需要拉取，太费流量了，有没有优化方法？**

答：常用优化方法有两种

（1）**延迟拉取，按需拉取**

（2）**时间戳**

**问题五：延迟拉取，按需拉取为什么有效？为什么能够减少拉取流量？**

答：用户在使用APP的过程中，有些数据是一定会使用到的，有些数据是不一定会使用到的。对于一定会使用到的数据，登录时拉取可以提升后续用户体验。对于不一定会使用到的数据，登录时拉取可能浪费流量，这些数据如果进行“延迟拉取”，可以节省流量。

**问题六：哪些数据不登录后不一定会使用，可以延迟拉取？**

答：这个问题的答案和业务紧密相关，以微信为例

**一定会使用到的数据**：好友列表（主页面要展示user-name），群组列表（主界面要展示group-name）

**不一定会使用到的数据**：好友详情，群组详情，群友列表，群友详情

故，对于微信，登录时只需要拉取好友列表（id+name）与群组列表（id+name）即可，**而其他数据，等用户真正点击和使用时再拉取即可，这样就可以大大减少拉取流量**。

**问题七：时间戳为什么有效？为什么能够减少拉取流量？**

答：本地数据不能直接使用的原因是，不确定数据是否最新，拉取服务器时间戳与本地时间戳进行比对，如果本地是最新的数据，就能避免重新拉取。id列表数据的变化频度是比较低的（增加id，减少id），时间戳机制非常的有效。

**问题八：加入时间戳机制后，数据拉取流程有什么变化？**

答：假设有100个好友，以好友详情数据的拉取为例，没有时间戳之前，直接向服务器拉取这100个好友的详情数据。

在有了时间戳之后，数据拉取流程变为：

（1）先拉取100个好友的时间戳

（2）客户端将100个好友的时间戳与本地时间戳对比，找出差异，假设有10个好友的信息发生了变化，时间戳改变了

（3）拉取有变化的10个好友的信息

**优点**是：**大大减少了数据传输量**（由拉取100个好友，降低到拉取10个好友）

**缺点**是：**增加了一次网络交互**（原来直接拉取，现在需要分别拉取时间戳与差异数据）

**问题九：使用时间戳的同时，能否降低网络交互次数呢？**

答：可以！

客户端对时间戳的使用，往往采取“客户端拉取时间戳”+“客户端比对时间戳”+“客户端再次拉取差异数据”的方式进行，“时间戳比对”的的CPU计算发生在客户端，其实，这个计算可以转嫁到服务器，步骤为：

（1）客户端**上传100个好友的时间戳**

（2）**“服务端”收到客户端上传的时间戳，与最新时间戳对比**，找出差异，假设有10个好友的信息发生了变化，服务端可以**直接将有差异的10个好友的数据返回**

**优点**是：**客户端减少了一次网络请求**

**缺点**是：比对时间戳差异的**CPU计算由“端”转嫁到了“云”**

**问题十：你怎么知道微信是这么做的？**

答：...不知道，运营需要，故...

但是，“客户端上传时间戳”的方法，我们曾经是这么做的，希望对业界同仁有启示作用。

## 应用层/安全层/传输层如何进行协议选型？

原创 2016-11-26 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959701&idx=1&sn=89850966b5e807097f8d273009104c50&chksm=bd2d04498a5a8d5f49e98825accf6fc3d3dd4edd49d92a0012befa992454c82191cbd4efa4be&scene=21" \l "#)



**系统设计，协议先行。**

大部分技术人没有接触协议的设计细节，更多的是使用已有协议进行应用层的编码，例如：

（1）使用http作为载体，设计get/post/cookie参数

（2）使用dubbo框架，而不用去深究内部的二进制包头包体，以及序列号反序列化的细节

无论如何，了解协议设计的原则，对深入理解系统通信非常有帮助。今天就以即时通讯（后称im）为例，讲讲应用层的协议选型。

**一、im协议的分层设计**

所谓“协议”是双方共同遵守的规则，例如：离婚协议，停战协议。协议有语法、语义、时序三要素。

（1）**语法**：即数据与控制信息的结构或格式

（2）**语义**：即需要发出何种控制信息，完成何种动作以及做出何种响应

（3）**时序**：即事件实现顺序的详细说明

im协议设计分为三层：应用层、安全层、传输层。



分别看下这三层的协议应该如何选型。

**二、im应用层协议设计**

应用层协议选型，常见的有三种：文本协议、二进制协议、流式XML协议。

（1）文本协议

文本协议是指 “贴近人类书面语言表达”的通讯传输协议，典型的协议是http协议，一个http协议大致长成这样：

GET / HTTP/1.1

User-Agent: curl

Host: musicml.net

Accept: \*/\*

文本协议的**特点**是：

a.可读性好，便于调试

b.扩展性也好（通过key:value扩展）

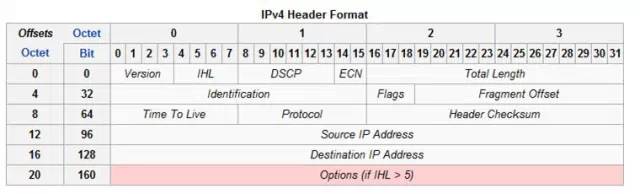
c.解析效率一般（一行一行读入，按照冒号分割，解析key和value）

d.对二进制的支持不好 ，比如语音／视频

im中，msn使用的是文本协议。

（2）二进制协议

二进制协议是指binary协议，典型是ip协议，以下是ip协议的一个图示：



二进制协议一般**定长包头和可扩展变长包体** ，每个字段固定了含义 ，例如IP协议的前4个bit表示协议版本号 （Version）。

二进制协议有这样一些**特点**：

a.可读性差，难于调试

b.扩展性不好 ，如果要扩展字段，旧版协议就不兼容了，所以一般设计时会有一个Version字段

c.解析效率超高（几乎没有解析代价）

对二进制的支持不好 ，比如语音／视频

im中，QQ使用的时二进制协议。

（3）流式XML协议

im的准标准协议xmpp就是使用流式XML，像gtalk，校内通这些im都是基于xmpp的，让我们来看一个xmpp协议的例子：

<message

to=’romeo@example.net’

from=’juliet@example.com’

type=’chat’

xml : lang=’en’>

<body>Wherefore art thou, Romeo?</body>

</message>

从xml标签中大致可以判断这是一个romeo发给juliet的聊天消息。

xmpp协议可以实现跨域的互通。例如gtalk和校内通用户聊天。只要服务端实现了s2s服务（server to server） ，不过现在的im基本没有互通需求 ，所以这个服务基本没有人实现。

Xmpp协议有几个特点：

a.它是准标准协议，可以跨域互通

b.XML的优点，可读性好，扩展性好

c.解析代价超高（dom解析）

d.有效数据传输率超低（大量的标签）

**个人旗帜鲜明的强烈不建议使用xmpp**，特别是无线端im，如果要用，一定要自己做压缩 ，减少网络流量（用过xmpp的同学都清楚，发一个登录包需要多少交互，要浪费多少流量）。

实际的栗子

下面来看一个im协议的实际例子 ，一般常见的做法是：定长二进制包头，可扩展变长包体。

包体可以使用用文本、XML等扩展性好的协议。

包头负责传输和解析效率，与业务无关。包体保证扩展性，与业务相关。

**这是一个实际的16字节im二进制定长包头**

//sizeof(cs\_header)=16

struct cs\_header

{

uint32\_t version;

uint32\_t magic\_num;

uint32\_t cmd;

uint32\_t len;

uint8\_t data[];

}\_\_attribute\_\_((packed));

a.前4个字节是version；

b.接下来的4个字节是个“魔法数字（magic\_num）“，用来保证数据错位或丢包问题，常见的做法是，包头放几个约定好的特殊字符，包尾放几个约定好的特殊字符 约定好，发给你的协议，某几个字节位置，是0x 01020304 ，才是正常报文；

c.接下来是command（命令号），用来区分是keepalive报文、业务报文、密钥交换报文等；

d.len（包体长度），告知服务端要接收多长的包体。

**这是一个实际的可扩展im变长包体**

message CUserLoginReq

{

optional string username = 1;

optional string passwd = 2;

}

message CUserLoginResp

{

optional uint64 uid =1;

}

使用的是google的Protobuf协议，可以看到，登录请求包传入的是用户名与密码，登录响应包返回的是用户的uid。

当然，除了Protobuf，可选择的可扩展包体协议还有xml、json、mcpack（这...）等。

**个人旗帜鲜明的推荐使用Protobuf**，主要有几个原因：

a.现成的解析库种类多，可以生成C++、Java、php等代码

b.自带压缩功能

c.在工业界已广泛应用

d.google制造

**三、im安全层协议设计**

im协议，消息的保密性非常重要 ，谁都不希望自己聊天内容被看到，所以安全层是必不可少的。

1、SSL

证书管理微微复杂，代价有点高。

2、自行加解密

自己来搞加解密，核心在于密钥的生成与管理，密钥管理方式有多种，主要有这么三种：

**（1）固定密钥**

服务端和客户端约定好一个密钥，同时约定好一个加密算法（eg：AES ），每次客户端im在发送前，就用约定好的算法，以及约定好的密钥加密再传输，服务端收到报文后，用约定好的算法，约定好的密钥再解密。这种方式，密钥和算法对程序员都是透明的。

**（2）一人一密钥**

简单说来就是每个人的密钥是固定的，但是每个人之间又不同，其实就是在固定密钥的算法中包含用户的某一特殊属性，比如用户uid、手机号、qq号等。

**（3）动态密钥（一session一密钥）**

动态密钥，一Session一密钥的安全性更高，每次会话前协商密钥。

密钥协商的过程要经过2次非对称密钥的随机生成，1次对称加密密钥的随机生成，具体详情这里不展开，有兴趣的同学可以看下SSL密钥协商额过程。

**四、im传输层协议设计**

可选的协议有TCP和UDP

现在的im传输层基本都是使用TCP，有了epoll等技术后，多连接就不是瓶颈了，单机几十万链接没什么问题。

先聊这么多，希望对大伙进行应用/安全/传输层协议选型有帮助。

## http如何像TCP一样实时的收消息？

原创 2016-09-26 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959605&idx=1&sn=21f25087bef3c3a966ef03b824365621&scene=21" \l "#)

http如何像TCP一样实时的收消息？

**一、webim如何实现消息推送**

webim通常有三种方式实现推送通道：

1）WebSocket

2）FlashSocket

3）http轮询

其中1）和2）是用Tcp长连接实现的，其消息的实时性可以通过tcp保证。

方案3）才算是webim实现消息推送的“正统”方案，用http短连接轮询的方式实现“伪长连接”，既然是轮询，有朋友就对消息的实时性产生了质疑。本文要解答，webim使用http长轮询如何保证消息的绝对实时性。

**二、人们为什么会误解http长轮询不实时**

什么是轮询？我擦，这个该怎么解释咧。

举个栗子，在火车上想上洗手间，挤到洗手间旁，却发现洗手间有人，于是你只能回座位继续等。过了N分钟，又朝洗手间的方向挤过去，却发现洗手间还是有人，又只能回坐等。这么一而再，再而三的每隔N分钟去洗手间查看洗手间是否有蹲位，这就是轮询。

webim用轮询的方式拉取消息会存在什么问题？

webim每隔N分钟，轮询调用 “获取消息”接口，有可能出现消息的延时，某一时刻刚拉取完消息，突然又产生了一条新消息，这条消息就必须等到N分钟之后，再次发起“获取消息”轮询时，才有机会获取到。

减小轮询时间间隔是否能解决消息延时的问题？

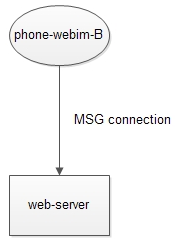
减小轮询时间间隔的确可以缩短延时时间，但也不能保证消息绝对的实时，同时又会产生新的问题，绝大部分的轮询调用，都没有消息返回，造成服务端极大的资源浪费。

很多人基于上述直觉，认为webim使用http长轮询的方式拉取消息，会导致消息有延时，其实，webim的http长轮询根本不是这么玩的。

**三、长轮询实际怎么玩**

消息连接

webim和webserver之间建立一条http连接，专门用作消息通道，这条连接叫http消息连接【见下图】



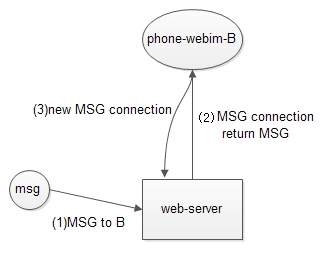
消息连接的4大特性

1）没有消息到达的时候，这个http消息连接将被夯住，不返回，由于http是短连接，这个http消息连接最多被夯住90秒，就会被断开（这是浏览器或者webserver的行为）

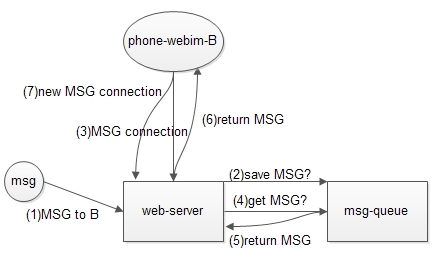
2）在1）的情况下，如果http消息连接被断开，立马再发起一个http消息连接【见下图中的步骤1、2】



3）在1）和2）的配合下，浏览器与webserver之间将永远有一条消息连接在（极限情况下会出现4）），每次收到消息时，这个消息连接就能及时将消息带回浏览器页面，并且在返回后，会立马再发起一个http消息连接【见下图中的步骤1、2、3】



4）如果消息到达时，上一个http消息连接正在返回，没有http消息连接可用（理论上http消息连接的返回是瞬时的，没有连接可用出现的概率极小），则将消息暂存入消息池中，下一个消息连接到达后（上一个消息连接返回后，根据2）和3）会立马返回新的消息连接，无等待时间），将消息带回，并又立刻返回生成新的消息连接【见下图中的步骤1、2、3、4、5、6、7】



上述1-4就能够保证一直有一条http消息连接在，以保证webim消息推送的绝对实时性。

**四、结论**

webim通过http长轮询可以保证消息的绝对实时性。这种实时性的保证**不是通过增加轮询频率来保证的，而是通过夯住http消息连接来保证的**，在大部分时间没有实时消息的情况下，这个http消息连接对于webserver的请求压力是90秒1次，能够大大节省了web服务器资源。

==【完】==

回【sed】[一分钟sed入门（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403340380&idx=1&sn=9b4ced5a19f60b0cebf6fd582f1d52ae&scene=21" \l "wechat_redirect" \t "_blank)

回【广告】[一分钟读懂互联网广告竞价策略（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403326874&idx=1&sn=ca03344326f35b2971b3ecae2026e8df&scene=21" \l "wechat_redirect" \t "_blank)

回【阵列】[一张“神图”看懂单机/集群/热备/磁盘阵列（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400720056&idx=1&sn=10c796793e27d475603f97b23e137c12&scene=21" \l "wechat_redirect" \t "_blank)

回【awk】[一分钟学awk够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400743254&idx=1&sn=147abc381ccd61e28f52699735c8748e&scene=21" \l "wechat_redirect" \t "_blank)

回【perl】[十分钟学perl够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400764946&idx=1&sn=4efab1953696c2e21071e58a74da9b82&scene=21" \l "wechat_redirect" \t "_blank)

回【mongo】[一分钟了解mongodb（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=201250814&idx=1&sn=fd6d85b727e1229e3324850217748d0f&scene=21" \l "wechat_redirect" \t "_blank)

回【2pc】[一分钟了解两阶段提交2PC（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400856212&idx=1&sn=b9734da77f2259bdd1fa89700b47eaff&scene=21" \l "wechat_redirect" \t "_blank)

回【join】[30秒懂SQL中的join（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400889133&idx=1&sn=3ea1c06fc76e92349a822f41f45b77b5&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

## 好架构是进化来的，不是设计来的（58架构演进）

2015-10-27 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400276397&idx=1&sn=ea044079667b82f6cad58bcb743af7bc&scene=21" \l "#)

**好的架构化是进化而来的，不是设计出来的**

----58沈剑

**核心内容**：58同城流量从小到大过程中，架构是如何演进的？遇到了哪些问题？以及如何解决这些问题？

**核心观点**：好的架构不是设计出来的，而是进化而来的。

**如何演进**：站点流量在不同阶段，会遇到不同的问题，找到对应阶段站点架构所面临的主要问题，在不断解决这些问题的过程中，整个系统的架构就不断的演进了。

如何演进，简言之：找到主要矛盾，并解决主要矛盾。

**第一章：建站之初**

建站之初，站点流量非常小，可能低于十万级别。这意味着，平均每秒钟也就几次访问。请求量比较低，数据量比较小，代码量也比较小，几个工程师，很短的时间搭起这样的系统，甚至没有考虑“架构”的问题。

和许多创业公司初期一样，最初58同城的站点架构特点是“ALL-IN-ONE”：



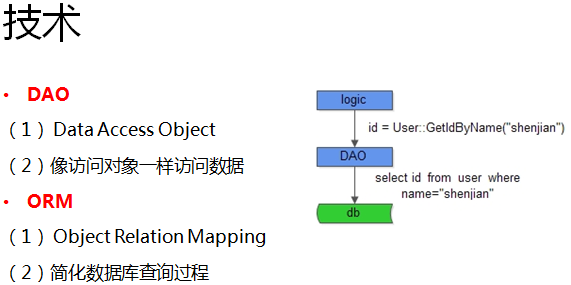
这是一个单机系统，所有的站点、数据库、文件都部署在一台服务器上。工程师每天的核心工作是CURD，浏览器端传过来一些数据，解析GET/POST/COOKIE中传过来的数据，拼装成一些CURD的sql语句访问数据库，数据库返回数据，拼装成页面，返回浏览器。相信很多创业团队的工程师，初期做的也是类似的工作。

58同城最初选择的是微软技术体系这条路：Windows、iis、SQL-Sever、C#

如果重新再来，我们可能会选择LAMP体系。

为什么选择LAMP？

LAMP无须编译，发布快速，功能强大，社区活跃，从前端+后端+数据库访问+业务逻辑处理全部可以搞定，并且开源免费，公司做大了也不会有人上门收钱（不少公司吃过亏）。现在大家如果再创业，强烈建议使用LAMP。



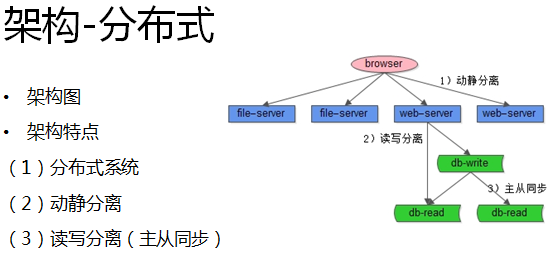
初创阶段，工程师面临的**主要问题**：写CURD的sql语句很容易出错。

我们在这个阶段**引进DAO和ORM，**让工程师们不再直接面对CURD的sql语句，而是面对他们比较擅长的面向对象开发，极大的提高了编码效率，降低了出错率。

**第二章：流量增加，数据库成为瓶颈**

随着流量越来越大，老板不只要求“有一个可以看见的站点”，他希望网站能够正常访问，当然速度快点就更好了。

而此时系统面临问题是：流量的高峰期容易宕机，大量的请求会压到数据库上，数据库成为新的瓶颈，人多并行访问时站点非常卡。这时，我们的机器数量也从一台变成了多台，我们的系统成了所谓的（伪）“**分布式架构**”：



我们使用了一些常见优化手段：

（1）动静分离，动态的页面通过Web-Server访问，静态的文件例如图片就放到单独的文件服务器上；

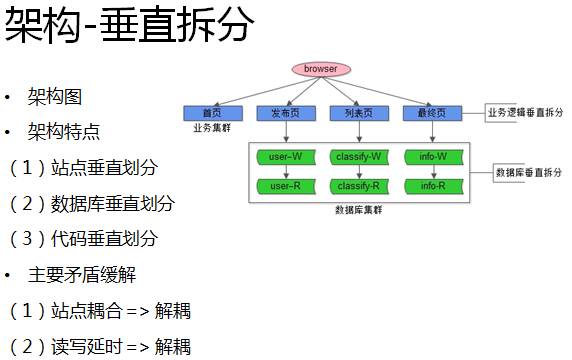
（2）读写分离，将落到数据库上的读写请求分派到不同的数据库服务器上；

互联网绝大部分的业务场景，都是读多写少。对58同城来说，绝大部分用户的需求是访问信息，搜索信息，只有少数的用户发贴。此时读取性能容易成为瓶颈，那么**如何扩展整个站点架构的读性能呢？常用的方法是主从同步，增加从库。**我们原来只有一个读数据库，现在有多个读数据库，就提高了读性能。

在这个阶段，系统的**主要矛盾为“站点耦合+读写延时”**，58同城是如何解决这两个问题的呢？

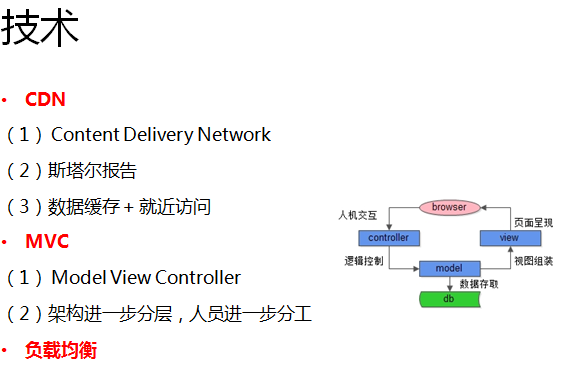
第一个问题是站点耦合。对58同城而言，典型业务场景是：类别聚合的主页，发布信息的发布页，信息聚合的列表页，帖子内容的详细页，原来这些系统都耦合在一个站点中，出现问题的时候，整个系统都会受到影响。

第二个问题是读写延时。数据库做了主从同步和读写分离之后，读写库之间数据的同步有一个延时，数据库数据量越大，从库越多时，延时越明显。对应到业务，有用户发帖子，马上去搜索可能搜索不到（着急的用户会再次发布相同的帖子）。



要解决耦合的问题，最先想到的是针对核心业务做切分，工程师根据业务切分对系统也进行切分：我们将业务垂直拆分成了**首页、发布页、列表页和详情页**。

另外，我们在数据库层面也进行了垂直拆分，将单库数据量降下来，让读写延时得到缓解。



同时，还使用了这些技术来优化系统和提高研发效率：

（1）对动态资源和静态资源进行拆分。对静态资源我们使用了CDN服务，用户就近访问，静态资源的访问速度得到很明显的提升；

（2）除此之外，我们还使用了MVC模式，擅长前端的工程师去做展示层，擅长业务逻辑的工程师就做控制层，擅长数据的工程师就做数据层，专人专用，研发效率和质量又进一步提高。

**第三章：全面转型开源技术体系**

流量越来越大，当流量达到百万甚至千万时，站点面临一个很大的问题就是**性能和成本的折衷**。上文提到58同城最初的技术选型是Windows，我们在这个阶段做了一次脱胎换骨的技术转型，全面转向开源技术：

（1）操作系统转型Linux

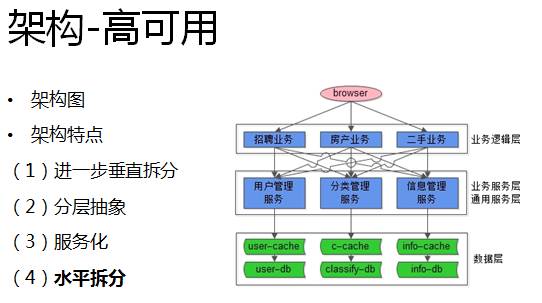
（2）数据库转型Mysql

（3）web服务器转型Tomcat

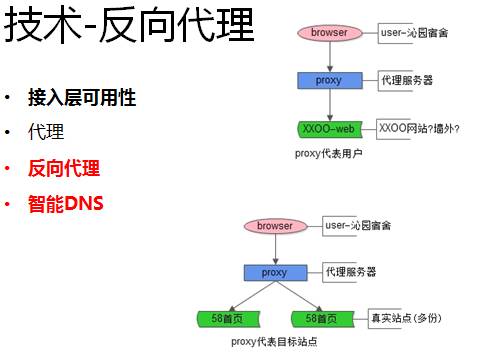
（4）开发语言转向了Java

其实，很多互联网公司在流量从小到大的过程中都经历过类似的转型，例如京东和淘宝。

随着用户量的增加，对站点**可用性要求也越来越高**，机器数也从最开始的几台上升到几百台。那么如何提供保证整个系统的可用性呢？首先，我们在业务层做了进一步的垂直拆分，同时引入了Cache，如下图所示：



在架构上，我们抽象了一个相对独立的服务层，所有数据的访问都通过这个服务层统一来管理，上游业务线就像调用本地函数一样，通过RPC的框架来调用这个服务获取数据，服务层对上游屏蔽底层数据库与缓存的复杂性。



除此之外，为了保证站点的高可用，我们使用了反向代理。

**什么是代理？**代理就是代表用户访问xxoo站点。

**什么是反向代理？**反向代理代表的是58网站，用户不用关注访问是58同城的哪台服务器，由反向代理来代表58同城。58同城通过反向代理，DNS轮询， LVS等技术，来保证接入层的高可用性。

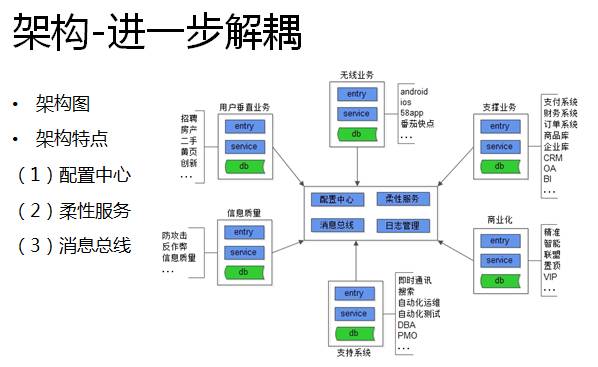
另外，为了保证服务层和数据层的高可用，我们采用了冗余的方法，单点服务不可用，我们就冗余服务，单点数据不可用，我们就冗余数据。

这个阶段58同城进入了一个**业务高速爆发期**，短期内衍生出非常多的业务站点和服务。新增站点、新增服务每次都会做一些重复的事情，例如线程模型，消息队列，参数解析等等，于是，58同城就研发了自己的站点框架和服务框架，现在这两个框架也都已经开源：

（1）站点框架Argo：https://github.com/58code/Argo

（2）服务框架Gaea：https://github.com/58code/Gaea

这个阶段，为了进一步解耦系统，我们引入了配置中心、柔性服务和消息总线。



引入**配置中心**，业务要访问任何一个服务，不需要在本地的配置文件中配置服务的ip list，而只需要访问配置中心。这种方式的扩展性非常好，如果有机器要下线，配置中心会反向通知上游订阅方，而不需要更新本地配置文件。

**柔性服务**是指当流量增加的时候，自动的扩展服务和站点。

**消息总线**也是一种解耦上下游“调用”关系常见的技术手段。

机器越来越多，此时很多系统层面的问题，靠“人肉”已经很难搞定，于是自动化变得越来越重要：自动化回归、自动化测试、自动化运维、自动化监控等等等等。

最后补充一点，这个阶段我们引入了不少智能化产品，比如智能推荐，主动推荐一些相关的数据，以增加58同城的PV；智能广告，通过一些智能的策略，让用户对广告的点击更多，增加同城的收入；智能搜索，在搜索的过程中加入一些智能的策略，提高用户的点击率，以增加58同城的PV。这些智能化产品的背后都由技术驱动。

**第四章、进一步的挑战**

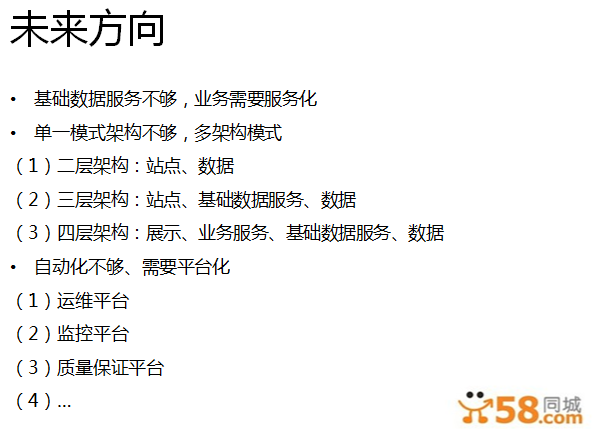
现在，58同城的流量已经达到10亿的量级，架构上我们规划做一些什么样的事情呢，几个方向：

（1）业务服务化

（2）多架构模式

（3）平台化

（4）...



**第五章：小结**

最后做一个简单的总结，网站在不同的阶段遇到的问题不一样，而解决这些问题使用的技术也不一样：

（1）**流量小**的时候，我们要提高开发效率，可以在早期要引入ORM，DAO；

（2）**流量变大**，可以使用动静分离、读写分离、主从同步、垂直拆分、CDN、MVC等方式不断提升网站的性能和研发效率；

（3）面对**更大的流量**时，通过垂直拆分、服务化、反向代理、开发框架（站点/服务）等等手段，可以不断提升高可用（研发效率）；

（4）在面对**上亿级的流量**时，通过配置中心、柔性服务、消息总线、自动化（回归，测试，运维，监控）来迎接新的挑战；

===【完】===

回【同城】58同城mysql实战（上）

回【数据库】58同城数据架构设计（下）

回【秒杀】秒杀系统架构优化思路（火）

回【招聘】入职58到家

回【微信】微信为何这么省流量（火）

## 58同城推荐系统架构设计与实现

原创 2016-06-21 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959464&idx=1&sn=81c29bf2e237499be07deba826f73c39&scene=21" \l "#)

**主题**

58同城推荐系统架构设计与实现

### 一、推荐系统架构介绍

推荐系统是一个微庞大的工程、算法与业务综合的系统，其主要分为三大子系统：

1）**线下**推荐子系统；

2）**线上**推荐子系统；

3）**效果评估**子系统；

后文将重点讨论以上三大子系统的设计与实现。

### 二、线下推荐子系统

线下推荐子系统又主要分为**线下挖掘模块**、**数据管理工具**两大部分。

#### ****线下挖掘模块****



线下挖掘模块，是各类**线下挖掘算法实施的核心**，它读取各种数据源，运用各种算法实施线下数据挖掘，产出初步的挖掘结果，并将挖掘结果以一定格式保存下来。典型的，实施这些挖掘策略的是一些跑在hadoop平台上的job，并行实施策略，并将挖掘结果保存到hadoop上。

#### ****数据管理工具****

数据管理工具，即DataMgrTools，它是一个工具（或者服务），它能够接受一些管理命令，读取某些特定格式的线下数据，将这些数据实时或者周期性的打到线上的redis或者内存中，供线上服务读取。



数据管理工具是一个与业务无关的通用工具，它需要支持多种特定格式数据的上传，因为线下挖掘模块产出的数据可能存储在文件里，HDFS上，数据库里，甚至是特定二进制数据。

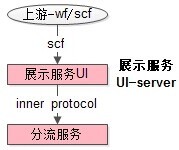
该工具的实现要点是：定义好线下数据格式，线上数据格式，通过上下游API做数据的迁移和转换。

### 三、线上推荐子系统

线上推荐子系统主要分为**展示服务、分流服务、推荐内核、策略module服务**等几个部分。

#### ****展示服务****

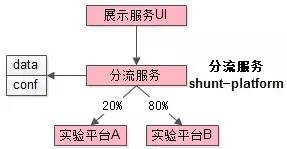
展示服务，或者说是接入服务，它是整个推荐系统线上部分的入口，即整个推荐系统的接入层，它向上游提供接口，供上游业务方调用。



展示服务是无状态的服务（线上子系统各个服务都是无状态的服务），可以任意水平扩展，该服务的实现要点是：定义好通用的接口格式。

#### ****分流服务****

分流服务，它是推荐系统中一个非常有特色也非常重要的一个服务，它的作用是将上游过来的请求，按照不同的策略，以不同的比例，分流到不同的推荐算法实验平台（也就是下游的推荐内核）中去。

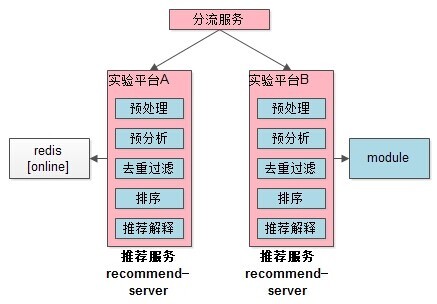


**分流服务如何判断上游过来的一个请求分配到那个推荐算法实验平台呢？**答案是通过策略和配置。从架构图中可以看到，几乎所有的服务都需要读取数据（data）和配置（conf），这些data可能是在线的动态变化的数据（例如：从redis中读取的数据），亦可能是相对静态的数据（例如：城市列表），conf比较好理解，即一些配置（例如：所有请求80%流量必须走A算法实验平台）。通过这些策略和配置，配合请求带过来的参数，分流服务计算出流量分配到哪个实验平台。

该服务的实现要点是：实现通用的支持与或非关系的可配置的分流规则，与下游实验平台定义好通用的接口以实现将流量按需打往不同的实验平台。

#### ****推荐内核****

推荐内核，是各类**线上推荐算法实施的核心**，它其实只是一个通用的实验平台容器，每个推荐服务内部可能跑的是不同类型的推荐算法。



虽然推荐服务中跑着不同的推荐算法，但每个算法的实施步骤都是相同的，都需要经过：

（1）预处理；

（2）预分析；

（3）去重过滤；

（4）排序；

（5）推荐解释；

等五个步骤，每个步骤都可能存在多种不同的算法，不同的模型，各个步骤中的一种算法组合起来，完成一个完整的流程，构成一个“推荐算法实验平台”。

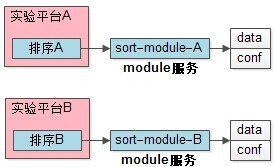
对于上述每个不同步骤中的不同模型，可能需要访问不同的外部module服务，例如：

推荐解释步骤，可能有两个模型，第一个模型在推荐解释阶段可能需要访问“解释-module1-服务”，第二个模型在推荐解释阶段可能需要访问“解释-module2-服务”，这些不同模型访问不同业务的需求，在架构层面都需要支持。

该服务的实现要点是：在一个推荐服务框架中跑多种策略，支持多个算法工程师在一个框架内并行开发/实验多个推荐算法，配合分流服务实现推荐算法实验平台。

#### ****策略服务****

策略服务，又叫策略module服务，它实现了一个个推荐内核下游的推荐module。在推荐内核执行各个推荐步骤时，每个步骤中都可能存在不同的算法/策略，这些算法/和策略可能需要调用一些和策略绑定比较紧密的module服务，它们并不是通用服务，而是相对专有的服务。



例如：排序module服务，需要有一套方便，高效，可扩展的排序服务。

该服务的实现要点是：实现一个通用的服务框架，让算法人员能够快速的生成module服务，并将自己的需求在module中实现，且能够在算法实验平台方便的进行module服务的调用。

### 四、效果评估子系统

效果评估子系统又分为推荐**服务调用端**、**浏览器上报端**、**实施效果分析端**。

#### ****推荐服务调用端****

调用推荐系统接口的58同城业务线，例如招聘业务线。

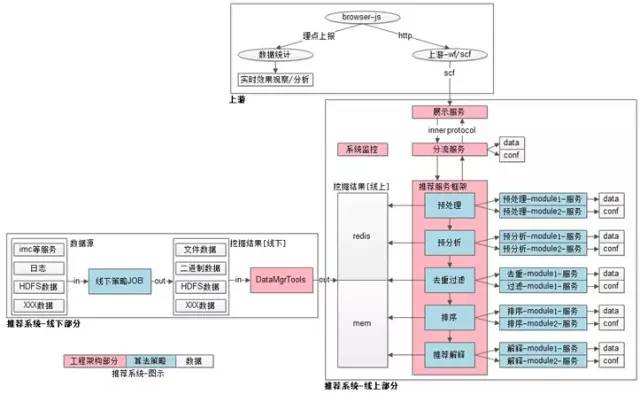
#### ****浏览器上报端****

浏览器端js，调用招聘服务时，能够在页面展现出推荐系统中推荐出来的结果，并且能够知道哪些推荐结果被点击了，且会将这些被展示的与被点击的信息进行上报。

#### ****实时效果分析端****

浏览器js将被展示的推荐结果，与被点击的推荐结果进行上报后，有一个实时效果观察的平台，第一时间得知上线后推荐算法/推荐策略的效果。

### 五、总体架构图



综合前面章节所述，58同城推荐系统总体架构图如上。

推荐系统是一个工程、算法和业务的综合性系统，上线了推荐系统，从此58同城正式进入了智能数据推荐的时代。

**关于-58同城推荐业务**

58同城是一个用户与商户共依的平台，信息的推荐对58同城而言至关重要。以58同城的招聘业务线为例：在招聘用户端，为用户推荐更多很好的相关职位，能够增强用户的体验，也增加了58同城的PV；在招聘商家端，为商户推荐更多更好的相关简历，能够增强商家的体验，促进简历的下载量，从而增加58同城的收入。

推荐业务如此重要，在技术层面，如何设计推荐系统的架构，是本文重点讨论的内容。

==【完】==

回【检索】[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)

回【钱包】[入驻微信钱包的技术优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959423&idx=1&sn=19d27145beab5b0238280e2e3804e41b&scene=21" \l "wechat_redirect" \t "_blank)

回【消息】[58到家通用实时消息平台架构细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "wechat_redirect" \t "_blank)

回【秒杀】[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959391&idx=1&sn=fb28fd5e5f0895ddb167406d8a735548&scene=21" \l "wechat_redirect" \t "_blank)

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

## 从0开始做互联网推荐-以58转转为例

2016-01-04 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=402978440&idx=1&sn=f0ec24d9f3c81fe49868d12a5128fcc9&scene=21" \l "#)

从0开始做互联网推荐【产品+算法+实现】

**一、58转转简介**

**58旗下真实个人闲置物品交易平台**

**二、从0开始设计推荐产品框架**

（1）**首页推荐**：提取用户画像，根据线下提取出的用户年龄、性别、品类偏好等在首页综合推荐宝贝

（2）**宝贝详情页推荐**：买了还买，看了还看类的关联宝贝推荐

（3）**附近推荐**：和首页推荐的差异在于，提高了地理位置的权重，地理位置不仅要包含当前地理位置，还需要包含常见活跃区域，例如家里、公司等

（4）**搜索推荐**：除了关键词全匹配，要考虑同义词、近义词、易错词、拼音等推荐，产品层面，提示“你是不是想找xxoo宝贝”

（5）**召回推荐**：在用户退出系统后，通过RFM模型做优惠券推送或者消息推送做客户挽留与召回

TIPS：什么是RFM模型？

RFM模型：根据用户最近一次购买时间Recency，最近一段时间的购买频度Frequency，最近一段时间的购买金额Monetary，加权得到的一个代表用户成交意愿的一个分值。

**三、从0开始进行推荐策略实现**

**【用户画像】**

根据用户填写的资料、用户历史行为（购买、收藏、喜欢、分享、评论、浏览等行为）、微信背后的用户画像，得到用户的特性画像：

年龄段 -> 推荐母婴、3C用品？

性别 -> 推荐母婴、美容保健用品？

手机型号 -> 推荐手机

活跃时间 -> 在这个时间段推送消息

品类偏好 -> 相关品类推荐

地域 -> 附近推荐

…

**【如何构建画像】**

（1）读取用户安装的应用程序列表构建画像

装有滴滴用户端 -> 没有车

装有滴滴司机端 -> 有车

装有CSDN -> 男性

装有美柚、美颜APP -> 女性

…

（2）用户行为日志

启动日志 -> 获取活跃时段

经纬度 -> 获取活跃地域

购买、收藏、喜欢、分享、评论、浏览-> 获取品类偏好

第三方数据 -> 完善用户画像

**【宝贝画像】**

58转转的宝贝都是非结构化的数据，比较难做统一的宝贝画像，只能细分品类的做宝贝画像，例如手机画像等。

**【如何构建宝贝画像】**

对于58转转来说，要做宝贝画像必须细分类别，可以分词词频统计配合人工review的方式画像，以鞋为例，画像可能为

单鞋

纯牛皮

尺码

适合春秋穿

女鞋

价格及变动

包邮

**【标签化与个性化推荐】**

画像完成之后，如何对用户进行宝贝推荐呢？

（1）给用户和宝贝画像完毕之后，要将每一个用户和每一个宝贝打上标签TAG

（2）统计用户uid所有购买、收藏、喜欢、分享、评论、浏览的所有宝贝ID集合set<bb-id>

（3）统计这些宝贝ID所有对应的TAG，使用加权打分的方式，可以根据频次统计出对各TAG的喜好程度

（4）对于所有宝贝，根据uid对各TAG的喜好程度，使用加权打分的方式，可以统计出对各宝贝的喜好程度

（5）排除已经购买、收藏、喜欢、分享、评论、浏览过的宝贝，其他宝贝按照打分高低推荐即可

（6）搜索推荐需要加上“搜索条件”，附件推荐需要加强“附近权重”

需要注意的是，个性化推荐的准确性，一定程度上依赖于历史行为数据的收集，对于新用户，在缺乏历史行为积累时，可以推荐“热度最高”的宝贝，未来再根据其历史行为，不断增强推荐的准确率。

**【分类预测推荐】**

一个用户对一个宝贝是否进行购买，可以抽象成一个0和1的分类问题，也可以抽象成一个购买概率的数学问题，可以构造分类模型来计算用户对每个宝贝的购买概率，将概率最高的作为推荐的宝贝。

为了实现分类预测推荐，需要：

（1）准备训练数据集，包含用户、宝贝、用户是否购买了宝贝等历史数据，需要注意的是，数据集应当覆盖尽可能多的用户（要包含所有TAG）和宝贝（要包含所有分类及TAG）

（2）构造训练分类模型

（3）根据模型训练的结果，计算每一个用户对每一个宝贝某买的概率

（4）按照概率排序，对宝贝进行推荐

**【协同过滤推荐】**

协同过滤，用过的人都知道，不一定效果最好，但几乎适用于所有的业务场景：当向用户A做协同过滤推荐时，可以先找到和他兴趣相似的用户群体G，然后把G喜欢的、并且A没有点击过的宝贝推荐给A，这就是基于用户的协同过滤。

为了实现系统过滤推荐，需要：

（1）准备训练数据集，根据每个用户对每个宝贝的喜好，构建喜好矩阵（这是一个非常稀疏的矩阵），根据用户对宝贝购买、收藏、喜欢、分享、评论、浏览的行为量化这个喜好

（2）构造系统过滤训练模型

（3）针对每一个用户，根据模型给出其喜好宝贝列表

在做协调过滤推荐时需要注意，较新的宝贝，由于大部分人都没有相关喜好数据，所以使用协同过滤推荐时，新宝贝比较难被推荐上去，这是协同过滤的缺点，需要综合其他推荐策略来解决。

好了，暂时先到这里，希望对58转转，以及其他刚开始做推荐的互联网产品有帮助。

注：本文是58到家推荐负责人@王洪权 在和58转转做技术交流时，@58沈剑 做的纪要。

==【完】==

回【拍卖】互联网广告之拍卖理论

回【广告】一分钟读懂互联网广告竞价策略

回【kkkk】3分钟懂K-means聚类算法（附源码）

回【回归】3分钟懂线性回归预测算法（附源码）

回【宝马】如何在微信刷出宝马广告

回【红包】抢红包统计学（技术贴，知道为啥自己越抢越穷了吧）

## 从0开始做垂直O2O个性化推荐-以58到家美甲为例

2016-01-07 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403209303&idx=1&sn=f4fea7390b3a2c31e693189aee44af59&scene=21" \l "#)

从0开始做垂直O2O个性化推荐

上次以58转转为例，介绍了如何**[从0开始如何做互联网推荐产品](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=402978440&idx=1&sn=f0ec24d9f3c81fe49868d12a5128fcc9&scene=21" \l "wechat_redirect" \t "_blank)**（回复“推荐”阅读），58转转的宝贝为闲置物品，**品类多种多样，要做统一的宝贝画像比较难**，而分类别做宝贝画像成本又非常高，所以更多的是进行用户画像、分类预测推荐、协同过滤推荐等个性化推荐。

有些同学反馈，他们的产品是垂直类的O2O产品，分类单一，**可以简单的实现宝贝画像，这类垂直O2O产品怎么从零开始做个性化推荐呢？**这是本文要讨论的问题

**一、58到家美甲简介**

58到家有三大自营业务“家政”“美甲”和“速运” ，美甲能够实现“足不出户，享品质服务，做美丽女人”，目前提供上门美甲、修复与卸甲、美睫、化妆等服务。

http://bj.daojia.com/liren/

**二、从0开始设计垂直O2O推荐框架**

（1）**列表页推荐**：用户既然进入到了美甲，成交意愿是非常强烈的，首页的推荐至关重要

（2）**宝贝详情页推荐**：买了还买，看了还看类的关联宝贝推荐

（3）**下单成功页推荐**：既然下单了某个甲样，可能会喜欢相近的甲样哟

（4）**召回推荐**：在用户退出系统后，通过RFM模型做优惠券推送或者消息推送做客户挽留与召回

**RFM模型**：根据用户最近一次购买时间Recency，最近一段时间的购买频率Frequency，最近一段时间的购买金额Monetary，加权得到的一个代表用户成交意愿的一个分值。

**三、甲样列表页推荐详细流程**

（1）用户点击进入甲样列表页

（2）画像用户的消费能力

（3）抽取购买、收藏、喜欢、浏览的历史数据

（4）根据历史数据，对所有甲样进行打分，综合一些产品策略，推荐出首屏的4个甲样，例如：



（5）如果用户下单，以被下单的相似甲样做推荐

（6）如果用户跳出，可以根据信用评级、消费等级做优惠券召回推荐

**四、与业务紧密结合的策略规则**

推荐系统并不是一个单纯的算法问题，而是一个与产品、工程架构都相关的综合性问题，不同的业务会有不同的产品策略，这些是在做推荐时需要考虑的，以美甲为例，需要考虑：

（1）排序前2名要推荐**最符合用户消费能力的甲样**（例如“价格小于150”）

（2）被推荐的4个甲样要**覆盖尽可能多的消费区间**（例如“两个甲样价格小于150，两个甲样价格大于150”）

（3）被推荐的4个甲样要**覆盖最火的产品、旧产品、新产品**（例如“1个爆品，2个旧加油，1个新甲样”）

（4）垂直相邻的甲样，颜色不同（为了视觉体验）

（5）水平相邻的甲样，颜色不同（原因同上）

（6）垂直相邻的甲样，款式不同（为了视觉体验，以及产品覆盖度、受众度）

（7）水平相邻的甲样，款式不同（原因同上）

（8）…

**五、如何利用甲样画像与用户购买、收藏、喜欢、浏览的历史数据对所有甲样进行打分？**

**【宝贝画像】**

垂直O2O的相对比较容易做宝贝画像，宝贝品类比较单一（甲样），宝贝的品种也比较少（几千几万种甲样），熟悉业务的人可以对宝贝进行画像（不需要复杂的机器学习方法），以甲样为例，可以抽象出：

款式

颜色

风格

场景

图案

其他

等多个核心属性

**【核心属性赋值，标签化】**

宝贝画像完毕之后，对于每一个核心属性，可以进行赋值，实施标签化

**款式**：纯色，法式，渐变，彩绘，贴饰

**颜色**：红色，粉色，蓝色，白色

**风格**：简约，甜美，复古，可爱

**场景**：派对，旅行，约会，晚宴，夜店

**图案**：卡通，小碎花，动物，桃心，五角星

**【抽取用户历史行为】**

抽取购买、收藏、喜欢、浏览的历史行为数据，得到一些甲样ID集合set<bb-id>

**【查询所有历史行为甲样ID的画像属性，对标签进行频率统计】**

用户U历史行为某买了甲样1：bb-id1，收藏了甲样2：bb-id2

从库中查询出所有甲样的详细属性

bb-id1：彩绘，红色，可爱，夜店，桃心

bb-id2：彩绘，粉色，可爱，夜店，桃心

对标签进行统计

款式：{彩绘：2}

颜色：{红色：1，粉色：1}

风格：{可爱：2}

场景：{夜店：2}

图案：{桃心：2}

**【根据标签统计，量化对标签的喜爱程度】**

例如，标签量化打分公式可以为：score=同类标签出现频率

那么，对于“款式”这个属性，依据上述统计，各标签的打分是：

纯色=0分，法式=0分，渐变=0分，彩绘=1分，晕染=0分，贴饰=0分（假设只有5种款式）

同理，对于“颜色”这个属性，依据上述统计，各标签的打分是：

红色=0.5分，粉色=0.5分，蓝色=0分，白色=0分（假设只有4种颜色）

…

这个打分是一个简单举例，实际上的打分公式会复杂很多（例如购买与收藏贡献的分值不一样）

**【根据上述量化标签，量化用户对每个甲样的喜爱程度】**

例如，对于一个甲样X{纯色，红色，简约，夜店，卡通}，可以计算出用户对它的喜爱分值为

socre-X = 0(纯色) + 0.5(红色) + 0(简约) + 1(夜店) + 0(卡通) = 1.5分

这个打分是一个简单举例，实际上打分公式会复杂很多（例如各个属性的权重是不一样的）

**【对所有甲样计算分值，排序】**

**【从高到底进行甲样推荐】**

推荐的过程中注意，4款甲样要符合第四个大步骤中提到的产品策略（要覆盖各个价格范围，相邻颜色与样式不同等）

**【个性化推荐完成】**

好了，暂时先到这里，上面的思路绝对是能落地的，希望58到家美甲的推荐，对其他刚开始做垂直O2O互联网产品的同学有帮助。

注：本文是58到家推荐负责人@王洪权 做58到家美甲推荐技术交流时，@58沈剑 做的纪要，内容“略”有修改。

==【完】==

回【拍卖】[互联网广告之拍卖理论](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=201042653&idx=1&sn=2b2489baf140668c693033a1b9daaeb8&scene=21" \l "wechat_redirect" \t "_blank)

回【广告】[一分钟读懂互联网广告竞价策略](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=201272878&idx=1&sn=2b48fdffc59408493ad7f6d431611361&scene=21" \l "wechat_redirect" \t "_blank)

回【kkkk】[3分钟懂K-means聚类算法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=202623552&idx=1&sn=cad4495ecc83da74fdc3d8054f78db75&scene=21" \l "wechat_redirect" \t "_blank)（附源码）

回【回归】[3分钟懂线性回归预测算法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=202722882&idx=1&sn=d5395c53d7453222fbd7e06d2e51bc9d&scene=21" \l "wechat_redirect" \t "_blank)（附源码）

回【宝马】[如何在微信刷出宝马广告](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=202849985&idx=1&sn=b921c34be65ebd9f5adfe178dd36ead7&scene=21" \l "wechat_redirect" \t "_blank)

回【红包】[抢红包统计学](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=203616226&idx=1&sn=02e5e7a827ba867849db513a96446fa9&scene=21" \l "wechat_redirect" \t "_blank)（技术贴，知道为啥自己越抢越穷了吧）

回【推荐】[从0开始做互联网推荐](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=402978440&idx=1&sn=f0ec24d9f3c81fe49868d12a5128fcc9&scene=21" \l "wechat_redirect" \t "_blank)

## 58到家入驻微信钱包的技术优化

原创 2016-04-28 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959423&idx=1&sn=19d27145beab5b0238280e2e3804e41b&scene=21" \l "#)

**一、需求缘起**

大伙打开微信钱包，会发现58到家入驻了微信钱包的一级入口（如下图），这个入口流量极大，微信**要求被接入的H5必须能抗住n万的qps**（58到家的系统是偏交易的系统，虽然一天100w订单其实也没多少请求），这是之前的业务系统没有遇到过的，要抗住这个n万的qps的优化思路是怎么样的呢？



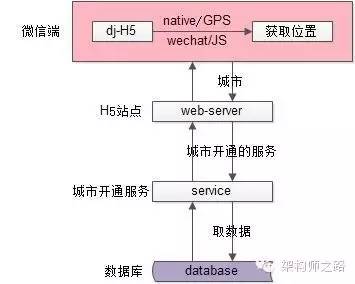
这里做一个思路分享，希望能对业界同仁有启示作用。

**二、业务分析**

在微信钱包里，点击进入58到家，会发现其实是一个**类别落地页，根据不同城市开通的服务类别**，展示不同类别的入口（如下图）。



很容易想到，整个架构与流程是这样滴：



架构分层：

（1）微信钱包端，嵌有到家H5页面

（2）web-server层，生成H5页面

（3）service层，提供“城市开通了哪些核心服务”的接口

（4）数据库层，存储了“城市开通了哪些核心服务”的数据

核心流程：

**步骤一**：微信端通过native的GPS定位或者微信的js-sdk获取用户当前所在城市，并往web-server发送http请求

**步骤二**：web-server收到http请求，调用service层，获取当前城市开通了哪些核心服务的数据，以瓶装返回html

**步骤三**：service收到RPC请求，调用mysql，获取真正的数据

**步骤四**：mysql返回service，service返回web-server，web-server拼装html，返回微信钱包

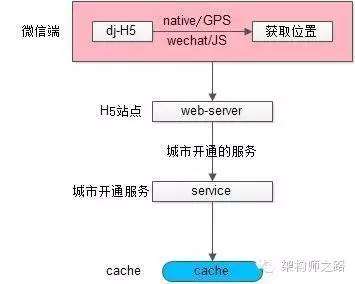
**潜在的问题**：

每秒钟n万的qps数据库扛不住

**三、优化分析**

看到这里，很多读者就笑了，这个场景加个缓存不就搞定了么，好吧，是可以，但本文的重点并不是加一个缓存，还有其他的梗等着你。

**场景分析**：这个“城市开通了哪些核心服务”是一个几乎只读的场景，因为城市要开通新的服务是很低频的，所以一大早就想到了cache的优化（cache要注意高可用），优化后的架构分层如下：

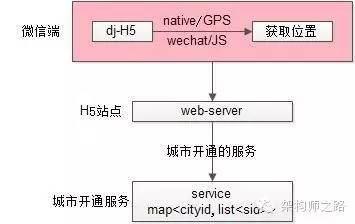


cache存储城市开通的核心服务列表，key value建立的是一个city-id到list<service-id>的映射关系。

几乎100%的请求会命中缓存。

**潜在的问题**：服务与缓存之间的带宽会不会成为瓶颈呢？

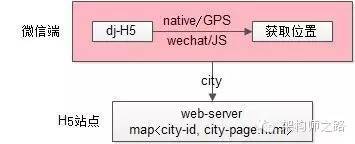
因为几乎是只读的请求，很容易想到将分布式缓存优化为服务内存缓存，优化后的架构分层如下：



**每一个服务内部都有一个map，存储city-id到list<service-id>的映射关系**，而不用通过cache来读取数据。

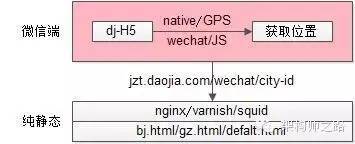
还能不能进一步优化，例如进一步降低网络交互呢？

是可以的，服务层可以做数据的缓存map<cityid, list<sid>>，web-server层可以进一步做页面缓存优化，架构图如下：



每一个站点层，直接做页面缓存，上游传入一个city-id，就直接将提前拼装好的页面返回，得到很高的性能。

有甚者，通过nginx/varnish/squid针对性的做一些“页面静态化的优化”，直接每个city有一个对应的html，能极大的提高吞吐量：



**四、页面静态化的适用场景**

在做站点架构的过程中，**“动静分离”**是一项很常见的优化手段：

（1）对于不变的首页、js、jpg、css，可以用专门的file-server来针对性优化（cdn/nginx/varnish/squid）提供高速访问

（2）对于动态的页面，有专门的tomcat/apache/iis/lighttpd集群来提供动态站点生成

一般来说动态站点时延会大大高于静态站点，应为每次生成动态站点需要访问服务（多次网络传输）、访问数据库（可能有很慢的磁盘io），并且还有大量的计算逻辑，而静态文件则可以直接返回。

**“页面静态化”**是一种将原本需要动态生成的站点提前生成静态站点的优化技术，什么样的业务场景可以进行“页面静态化”优化呢？

解答：**总数据量不大，生成静态页面数量不多的业务，非常适合于“页面静态化”优化**

<广告>

如果你读到了这里，说明对微信平台还是有点兴趣的，58到家**微信平台团队目前**只有一个女生负责（技术圈混，绝不骗人），**奇缺人，欢迎有志之士加入**。

**一些说明**：base北京，核心团队，技术导向，Java方向，和58沈剑一个团队

**加入方式**：（1）直接在文章下面留言（2）在公众号回复“招聘”获取加盟密钥

</广告完毕>

**例如**：到家开通的城市只有几百个，那只需提前生成几百个城市的“静态化页面”即可

**又例如**：一些二手车业务，整个公司可能只有几万量二手车库存，也可以提前生成这几万量二手车的静态页面

**再例如**：像58同城这样的信息模式业务，又几十亿的帖子量，就不太适合于静态化

**五、总结**

**“页面静态化”是一种将原本需要动态生成的站点提前生成静态站点的优化技术，总数据量不大，生成静态页面数量不多的业务，非常适合于“页面静态化”优化**。

==【完】==

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

回【秒杀】[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959391&idx=1&sn=fb28fd5e5f0895ddb167406d8a735548&scene=21" \l "wechat_redirect" \t "_blank)

回【晋升】[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)

回【消息】[58到家通用实时消息平台架构细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "wechat_redirect" \t "_blank)

回【id】[细聊分布式ID生成方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

## 创业公司快速搭建立体化监控之路（WOT2016）

原创 2016-05-05 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959433&idx=1&sn=9c9a222c06b2426f3935b04a10f90c0b&scene=21" \l "#)

本文内容：创业型公司如何快速搭建可扩展，可落地的立体化监控平台

**一、需求缘起**

创业型公司有系统监控么？来看两个case：

**case 1：CXO大群内贴了一张“用户微信投诉”的截图**

（1）CXO大群内贴了一张“用户微信投诉”的截图

（2）技术反馈“正在跟进”

（3）10分钟之后，CXO询问进度，技术反馈“正在解决”

（4）60分钟之后，CXO说怎么还没有解决，技术反馈“正在解决”

实际上，可能还没有找到问题在哪里。

**case 2：用户通过客服反馈功能不可用**

（1）用户反馈到客服，不能下单

（2）客服 -> 产品 -> 测试 -> 技术

（3）技术：站点层 -> 服务层1 -> 服务层2 -> 数据层

可能2个小时过去了，技术还没有定位到问题在哪一层。

存在的问题：技术被动

（1）出了问题成为最后知晓者，用户受影响周期长

（2）查找问题路径长，定位和修复问题时间久，用户受影响周期长

所有系统负责人能快速回答这两个问题么

（1）所负责的系统**现在运行是否正常**？

（2）如果不正常，**问题大致在哪里**？

今天的主题是“创业型公司如何快速解决这两个问题”

**二、解决方案：立体化监控**

**怎么知道系统运行是否正常？**

**回答**：监控

**什么是立体化监控？**

**回答**：多维度监控

**监控维度有哪些？**

回答：（1）机器、操作系统层面

（2）进程、端口层面

（3）日志层面

（4）接口层面

（5）用户层面

**三、创业型公司如何快速实现立体化监控**

**【如可快速实现机器、操作系统级别的监控？】**

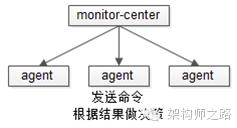
**回答**：zabbix，用过的都说好

**不足**：CPU，LOAD，内存，网络，磁盘异常说明系统一定异常，但**这些参数正常并不能说明系统正常**，例如：进程挂了，端口挂了，通过这些参数就检测不到

**【如何快速实现进程、端口级别的监控？】**

两类实现思路：分发型监控 + 汇总型监控

**分发型监控**



命令由监控中心分发到各个被监控机器的agent上，agent执行监控，实现要点：

（1）监控中心要实现扩展性较强的配置，方便扩展“监控哪个ip上哪个进程或者端口的存活性”

（2）对于进程与端口的监控，甚至无需agent来执行，直接使用带超时的端口连接或者telnet就能快速实现

**汇总型监控**



命令由agent在各台机器上执行，将结果汇总上报到监控中心接口，实现要点：

（1）agent必须能够快速部署到所有的机器

（2）agent如何快速从监控中心获取需要监控的进程和端口，必须要保证扩展性

（3）agent如何快速的执行本地检测，例如：进程监控用ps？端口监控用netstat？

进程与端口监控的**不足**：进程与端口异常说明系统一定异常，但**它们正常并不能说明系统正常**，例如：进程和端口都在，但ERROR日志狂刷

**【如何快速实现日志的监控？】**

两类实现思路：ERROR日志的监控 + 日志关键字监控

这两类实现又有“日志各机器单独监控”与“日志汇总到中心监控”两种方法，暂时不展开。

**ERROR日志监控快速实施要点**

（1）日志分级规范非常重要，需要进行日志按照级别分离，ERROR日志单独拿出来一个文件是最好的

（2）日志切分规范也很重要，建议按照小时切分

（3）1和2的目的，是为了保证扩展性，并减少扫描的日志量，做到了1和2之后，例如用一个crontab，设定一定阈值，每分钟wc -l ERROR文件，超过阈值就可以报警

（4）简易的配置与良好的扩展性，需要支持方面的增加“某一台机器”“某一个路径”“ERROR每分钟超过多少”的报警配置

**日志关键字监控**

和ERROR日志监控的思路是类似的，当日志中出现一些事先设定的关键字（或者出现频率超过一定阈值），例如exception、timeout就报警，这种报警能够报出比ERROR更精准的系统异常

ERROR日志监控与日志关键字监控的**不足**：ERROR日志超过阈值说明系统一定异常，**不超过阈值并不能说明系统正常**，例如：进程死锁，此时并不会刷ERROR日志

**【如何快速实现接口的监控】**

有两种常见的快速实现思路：统一keepalive接口 + 接口处理时间统一上报

**统一keepalive接口快速实施要点**

（1）在站点框架与服务框架层面统一实现一个keepalive接口

（2）监控中心统一调用站点、服务的keepalive接口

（3）简易的配置与良好的扩展性

**接口处理时间统一上报快速实施要点**

（1）在站点框架和服务框架层面统一实现处理时间的收集

（2）由于并发量很大，需要在本地进行初步汇总

（3）或者使用upd上报

（4）时间上报需要异步，不要因为这个而增加业务处理时间

（5）良好的配置与扩展性，监控中心统一配置报警（绝对时间，或者处理时间环比增长报警）

统一keepalive接口与接口处理时间统一上报的**不足**：上报异常说明系统一定异常，**上报正常不能说明系统正常**，例如：某个服务后端的数据库挂了，此时这个服务的keepalive接口返回其实是正常的，接口的处理时间可能会比平常要快很多（原来数据库还要执行一个sql，现在连接都拿不到，立马就返回了）

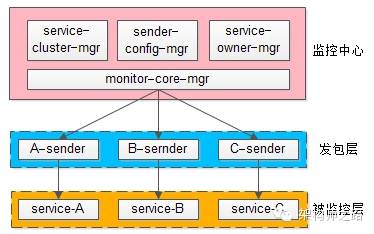
**【到底什么样的监控，才能说明系统是正常的呢？】**

郁闷了，上述多个维度的监控，都不能完全说明系统正常，怎么办？

**回答**：只有站在调用者的角度，对被调用方的可用性可靠性的评判才是最准确的

**思路**：模拟调用方调用站点、服务，来对站点和服务进行监控

通用接口监控分层架构图



如上图所示，实现“模拟调用方对站点和服务进行监控”的分层架构

**被监控层**：被监控的站点和服务，例如A，B，C

**发包层**：模拟站点和服务调用方的发包器，例如A-sender，B-sender，C-sender

**监控中心**：调度发包层对站点和服务进行监控，对结果进行管理，对阈值进行判断与实施报警

监控中心又分为这么几个部分：

（1）**集群管理**：每个被监控服务有哪些ip

（2）**监控项管理**：监控哪个服务、调度频率、防抖动配置、责任人

（3）**责任人管理**：责任人、邮箱、手机号、微信号

（4）**调度中心**：隔多长时间调度每个监控项

（5）**发包层通信**：获取发包层的监控结果与异常信息

监控流程，用伪代码描述吧：

for(每一个监控项里被监控的服务)｛ // 其实是并行执行的，并不是for

         for(这个服务所对应集群里的每个ip)｛

                   调度发包层，对服务进行发包；

                   收集发包层的监控结果与异常信息

                   if(异常次数超过我们设定的阈值){

                        找到服务对应的责任人;

                        异常信息发短信;

                        发邮件;

                        发微信;

}

         ｝

｝

其他实践：

（1）一个服务提供的接口很多，可以选取最核心的接口进行发包监控

（2）写接口可能会对数据产生污染，建议选取读接口进行监控

（3）如果一定要对写接口进行监控，务必插入操作和删除操作要是成对进行的（还是会对业务数据统计产生污染）

（4）发包层的sender程序可以复用接口测试的代码

（5）发包器的结果校验要进行业务校验，例如一个http请求仅仅检查返回码是200是不够的，还要检测返回的html或者json的内容是更准确的

**【什么样的监控，能决定凌晨收到报警而不起床处理呢？】**

**回答**：用户视角的监控

“模拟调用方调用站点、服务，来对站点和服务进行监控”的方法，可以精确的判断有问题的是哪一个ip上的哪一个服务上的哪一个接口，理论上应该是粒度最细的监控了，**为什么还需要用户视角的监控呢？**

**回答**：

（1）架构是做了可用性保证的，一个服务挂了，用户视角的监控没有报警，说明对用户没有影响，如果此时凌晨收到报警，也是不需要马上起床来处理的

（2）用户是在全国各地进行访问的，很有可能某个地域的网络出问题，此时只有在全国布点的用户视角监控才能发现

如何快速的实施用户视角的监控：

（1）复用接入层的接口监控，只是，不对每一个web-server的站点ip实施监控，而是对nginx反向代理层实施监控

（2）引入第三方监控

**四、总结**

创业型公司快速实施立体化多维度监控总结：

（1）机器、操作系统维度监控：zabbix

（2）进程、端口维度监控：分发型监控 + 汇总型监控

（3）错误日志与关键字维度监控

（4）keepalive接口与所有接口统一处理时间统一上报监控

（5）模拟调用方调用站点、服务，来对站点和服务进行监控

**到底什么样的监控，才能说明系统是正常的呢？**

**回答**：只有站在调用者的角度，对被调用方的可用性可靠性的评判才是最准确的

==【全文完，希望大伙有收获】==

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

回【秒杀】[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959391&idx=1&sn=fb28fd5e5f0895ddb167406d8a735548&scene=21" \l "wechat_redirect" \t "_blank)

回【消息】[58到家通用实时消息平台架构细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "wechat_redirect" \t "_blank)

回【id】[细聊分布式ID生成方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

## 巧用CAS解决数据一致性问题

2015-11-05 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400556481&idx=1&sn=42bc187b45e4e53aece4b8c1f43e8019&scene=21" \l "#)

**缘起**：在高并发的分布式环境下，对于数据的查询与修改容易引发一致性问题，本文将分享一种非常简单但有效的优化方法。

**一、业务场景**

业务场景为，购买商品的过程要对余额进行查询与修改，大致的业务流程如下：

（1）从数据库查询用户现有余额 SELECT money FROM t\_yue WHERE uid=$uid，不妨设查询出来的$old\_money=100元

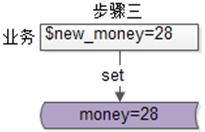


（2）业务层实施业务逻辑，比如购买一个80元的商品，并且打九折

if($old\_money> 80\*0.9) $new\_money=$old\_money-80\*0.9=28

http://mmbiz.qpic.cn/mmbiz/YrezxckhYOwqEGDX16FzribBMJ3okQE4IyibMRTgtommicU61DeN1mia9BSguDFdo8gqoKwI4JbqBEiarwvibbv8NZCQ/640?wx_fmt=png&wxfrom=5&wx_lazy=1

（3）将数据库中的余额进行修改 UPDAtE t\_yue SET money=$new\_money WHERE uid=$uid

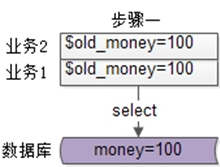


在并发量低的情况下，这个流程没有任何问题，原有金额100元，购买了80元的九折商品（72元），剩余28元。

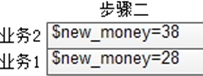
**二、潜在的问题**

在分布式环境中，如果并发量很大，**这种“查询+修改”的业务很容易出现数据不一致**。极限情况下，可能出现这样的异常流程：

（1）业务1和业务2同时查询余额，是100元

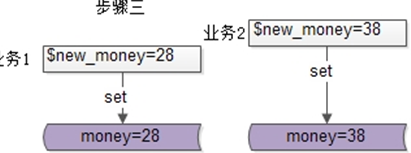


（2）业务1和业务2进行逻辑计算，算出各自业务的余额，假设业务1算出的余额是28元，业务2算出的余额是38元



（3）业务1对数据库中的余额先进行修改，设置成28元。

业务2对数据库中的余额后进行修改，设置成38元。



此时异常出现了，原有金额100元，业务1扣除了72元，业务2扣除了62元，最后剩余38元。

**三、问题原因**

高并发环境下，对同一个数据的**并发读**（两边都读出余额是100）与**并发写**（一个写回28，一个写回38）导致的数据一致性问题。

**四、原因分析**

业务1的写回：原有金额100，这是一个初始状态，写回金额28，理论上只有在原有金额为100的时候才允许写回成功，这一步没问题。

业务2的写回：的原有金额100，这是一个初始状态，写回金额38，理论上只有在原有金额为100的时候才允许写回成功，可实际上，这个时候数据库中的金额已经变为28了，这一步的写操作不应该成功。

**五、简易解决方案**

在set写回的时候，加上初始状态的条件compare，只有初始状态不变时，才允许set写回成功，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。

**六、业务的升级**

业务线使用CAS解决高并发时数据一致性问题，只需要在进行set操作时，compare一下初始值，如果初始值变换，不允许set成功。

对于上文中的业务场景，只需要将“UPDAtEt\_yue SET money=$new\_money WHERE uid=$uid”升级为

“UPDAtE t\_yue SETmoney=$new\_money WHERE uid=$uid **AND money=$old\_money**”即可。

并发操作发生时：

业务1执行 => UPDAtE t\_yue **SET money=28** WHERE uid=$uid **AND money=100**

业务2执行 => UPDAtE t\_yue **SET money=38** WHERE uid=$uid **AND money=100**

**【这两个操作同时进行时，只能有一个执行成功】**。

**七、怎么判断哪个执行成功，哪个执行失败**

set操作，其实无所谓成功或者失败，业务能通过affect rows得知哪个修改没有成功：

执行成功的业务，affect rows为1

执行失败的业务，affect rows为0

**八、总结**

高并发“查询并修改”的场景，可以用CAS（Compare and Set）的方式解决数据一致性问题。对应到业务，即在set的时候，加上初始条件的比对。

===【完】===

## 百度咋做长文本去重（一分钟系列）

原创 2016-01-19 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "#)

缘起：

（1）原创不易，互联网抄袭成风，很多原创内容在网上被抄来抄去，改来改去

（2）百度的网页库非常大，爬虫**如何判断一个新网页是否与网页库中已有的网页重复**呢？

这是本文要讨论的问题（尽量用大家都能立刻明白的语言和示例表述）。

**一、传统签名算法与文本完整性判断**

**问题抛出**：

（1）运维上线一个bin文件，将文件分发到4台线上机器上，如何判断bin文件全部是一致的？

（2）用户A将消息msg发送给用户B，用户B如何判断收到的msg\_t就是用户A发送的msg？

**思路**：

一个字节一个字节的比对两个大文件或者大网页效率低，我们可以用一个签名值（例如md5值）代表一个大文件，签名值相同则认为大文件相同（先不考虑冲突率）

**回答**：

（1）将bin文件取md5，将4台线上机器上的bin文件也取md5，如果5个md5值相同，说明一致

（2）用户A将msg以及消息的md5同时发送给用户B，用户B收到msg\_t后也取md5，得到的值与用户A发送过来的md5值如果相同，则说明msg\_t与msg相同

**结论**：md5是一种签名算法，常用来判断数据的完整性与一致性

**md5设计原则**：两个文本哪怕只有1个bit不同，其md5签名值差别也会非常大，故它只适用于“完整性”check，不适用于“相似性”check。

**新问题抛出**：

有没有一种签名算法，如果文本非常相似，签名值也非常相似呢？

**二、文本相似性的签名算法**

上文提出的问题，可以用局部敏感哈希LSH（Locality Sensitive Hash）解决，局部敏感哈希是一类文本越相似，哈希值越相似的hash算法，有兴趣的同学自行百度，这里分享一下minHash的思路。

**问题的提出**：什么是minHash？

**回答**：minHash是局部敏感哈希的一种，它常用来快速判定集合的相似性，也常用于检测网页的重复性，其思路为，用相同的规则抽取集合中的少部分元素代表整个集合，如果少部分元素的重合度很高，非常可能整个集合的重复度也很高。

**举例**：待判定的集合为A{1, 7, 5, 9, 3, 11, 15, 13}

已有的集合为：

B{10, 8, 2, 4, 6, 0, 1, 16},

C{100, 700, 500, 900, 300, 1100, 1500,1300},

D{1, 3, 2, 4, 6, 5, 8, 7}

假设使用部分元素代替全体集合的规则为：集合内元素进行排序，取值最小的4个（这个过程有信息损失，我们可以认为是一个hash过程）

处理结果为：

A{1, 3, 5, 7}

B{0, 1, 2, 4}      =>     A与B有1个元素相同

C{100, 300, 500, 700}      =>     A与C有0个元素相同

D{1, 2, 3, 4}      =>     A与D有2个元素相同

判断结论：我们认为集合A与集合D是最相似的

这个例子有点2，但基本能说明整体思路，**实际在执行的过程中**：

（1）我们可以使用更多的元素来代表集合，以提高准确性（例如，将上例中的4个元素代表集合升级为8个元素代表集合）

（2）我们可以使用更多的hash函数来代表集合，以提高准确性（例如，上例除了“排序后取值最小的4个元素代表集合”，还可以增加一个哈希函数“排序后取值最大的4个元素代表集合”）

（3）minHash可以量化评判相似度，亦可以评判网页是否重复（一个分类问题），设定相似度阈值，高于阈值为重复，低于阈值为不重复

（4）实际排重过程中，网页库中的哈希值都可以提前计算，只有待判定的集合或者网页的哈希值需要临时计算

**三、minHash与长文本重复度检测有什么关系**

目前看来没什么关系，但如果我们能将每一个长文本用一个集合来表示，就能将长文本的相似度用minHash来解决了。

**问题的提出**：如何将长文本转化为集合？

**回答**：我去，分词不是就可以么

**举例**：待判定的长文本为A{我是58沈剑，我来自58到家}

已有网页库集合为：

B{我是一只来自58的狼}

C{58到家，服务到家}

D{这事和我没关系，我是凑数的}

使用分词将上述文本集合化：

A{我，58，沈剑，来自，到家}

B{我，58，来自，狼}

C{58，服务，到家}

D{事，我，凑数，关系}

判断结论：当当当当，转化为集合后，可以快速判断A与B的相似度最高，当然实际执行过程中，除了分词还得考虑词频，用这种方法对长文本进行相似度检测，准确率非常高（文本越长越准）

**四、还有没有更有效的方法**

使用上述方法进行文本相似度检测，需要进行中文分词，词频统计，哈希值计算，相似度计算，计算量微大。

然而，抄袭成风，一字不改的风气，让技术有了更广阔的优化空间，赞！

怎么优化呢？

**不再进行分词，而是进行“分句”**，用标点符号把长文按照句子分开，使用N个句子集合（例如一篇文章中5条最长的句子作为签名，注意，长句子比短句子更具有区分性）作为文章的签名，在抄袭成风的互联网环境下，此法判断网页的重复度能大大降低工程复杂度，并且准确度也异常的高。

**五、结论**

**在抄袭成风的互联网环境下，采用“分句”的方式，用5条最长的网页内容作为网页的签名，能够极大的降低排重系统复杂度，提高排重准确率，不失为一种好的选择。**

标题只是噱头，百度是不是这么做的我并不知道，知情的同学说一下哈。

如果有收获，帮忙转发哈。

欢迎评论，有问必答。

===【完】===

回【58】[58怎么玩数据库架构](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403369608&idx=1&sn=48c56d64c2b5c27fed795219e7ba214a&scene=21" \l "wechat_redirect" \t "_blank)

回【微信】[微信为啥这么省流量](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400163013&idx=1&sn=911cf71925e2ba50d47955a713134acb&scene=21" \l "wechat_redirect" \t "_blank)

回【秒杀】[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=204965266&idx=1&sn=aad227dbcb544394ae1717e6af8b77ae&scene=21" \l "wechat_redirect" \t "_blank)

## 如何快速实现高并发短文检索

原创 2016-06-02 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959451&idx=1&sn=991d9c3737d7db50a8351d50cdf6419d&scene=21" \l "#)

**一、需求缘起**

某并发量很大，数据量适中的业务线需要实现一个“标题检索”的功能：

（1）并发量较大，每秒20w次

（2）数据量适中，大概200w数据

（3）是否需要分词：是

（4）数据是否实时更新：否

**二、常见潜在解决方案及优劣**

（1）数据库搜索法

具体方法：将标题数据存放在数据库中，使用like来检索

优点：方案简单

缺点：不能实现分词，并发量扛不住

（2）数据库全文检索法

具体方法：将标题数据存放在数据库中，建立全文索引来检索

优点：方案简单

缺点：并发量扛不住

（3）使用开源方案将索引外置

具体方法：搭建lucene，solr，ES等开源外置索引方案

优点：性能比上面两种好

缺点：并发量可能有风险，系统比较重，为一个简单的业务搭建一套这样的系统成本较高

**三、58龙哥的建议**

**问1**：龙哥，58同城第一届编程大赛的题目好像是“黄反词过滤”，你是冠军，当时是用DAT来实现的么？

龙哥：是的

**画外音：什么是DAT？**

普及：DAT是double array trie的缩写，是trie树的一个变体优化数据结构，它在保证trie树检索效率的前提下，能大大减少内存的使用，经常用来解决检索，信息过滤等问题。（具体大伙百度一下“DAT”）

**问2**：上面的业务场景可以使用DAT来实现么？

龙哥：DAT更新数据比较麻烦，不能增量

**问3**：那直接使用trie树可以么？

龙哥：trie树比较占内存

**画外音：什么是trie树？**

普及：trie树，又称单词查找树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计，保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希树高。（来源：百度百科）



例如：上面的trie树就能够表示｛and, as, at, cn, com｝这样5个标题的集合。

**问4**：如果要支持分词，多个分词遍历trie树，还需要合并对吧？

龙哥：没错，每个分词遍历一次trie树，可以得到doc\_id的list，多个分词得到的list合并，就是最终的结果。

**问5**：龙哥，还有什么更好，更轻量级的方案么？

龙哥：用trie树，数据会膨胀文档数\*标题长度这么多，标题越长，文档数越多，内存占用越大。有个一个方案，内存量很小，和标题长度无关，非常帅气。

**问6**：有相关文章么，推荐一篇？

龙哥：可能网上没有，我简单说一下吧，**核心思想**就是“内存hash ＋ ID list”

**索引初始化步骤为**：对所有标题进行分词，以词的hash为key，doc\_id的集合为value

**查询的步骤为**：对查询词进行分词，对分词进行hash，直接查询hash表格，获取doc\_id的list，然后多个词进行合并

=====例子=====

例如：

doc1 : 我爱北京

doc2 : 我爱到家

doc3 : 到家美好

**先标题进行分词**：

doc1 : 我爱北京 -> 我，爱，北京

doc2 : 我爱到家 -> 我，爱，到家

doc3 : 到家美好 -> 到家，美好

**对分词进行hash，建立hash + ID list**：

hash(我) -> {doc1, doc2}

hash(爱) -> {doc1, doc2}

hash(北京) -> {doc1}

hash(到家) -> {doc2, doc3}

hash(美好) -> {doc3}

这样，所有标题的初始化就完毕了，你会发现，数据量和标题的长度没有关系。

**用户输入**“我爱”，**分词**后变为{我，爱}，对各个**分词的hash**进行内存检索

hash(我)->{doc1, doc2}

hash(爱)->{doc1, doc2}

然后**进行合并**，得到最后的查找结果是doc1+doc2。

=====例子END=====

**问7**：这个方法有什么优点呢？

龙哥：存内存操作，能满足很大的并发，时延也很低，占用内存也不大，实现非常简单快速

**问8**：有什么不足呢？和传统搜索有什么区别咧？

龙哥：这是一个快速过度方案，因为索引本身没有落地，还是需要在数据库中存储固化的标题数据，如果不做高可用，数据恢复起来会比较慢。当然做高可用也是很容易的，建立两份一样的hash索引即可。另外，没有做水平切分，但数据量非常非常非常大时，还是要做水平切分改进的。

真实用户反馈：“龙哥指点一二，受益终身”

==【完】==

回【主从】[DB主从一致性架构优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959442&idx=1&sn=feb8ff75385d8031386e120ef3535329&scene=21" \l "wechat_redirect" \t "_blank)

回【钱包】[入驻微信钱包的技术优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959423&idx=1&sn=19d27145beab5b0238280e2e3804e41b&scene=21" \l "wechat_redirect" \t "_blank)

回【消息】[58到家通用实时消息平台架构细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "wechat_redirect" \t "_blank)

回【秒杀】[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959391&idx=1&sn=fb28fd5e5f0895ddb167406d8a735548&scene=21" \l "wechat_redirect" \t "_blank)

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

## 如何实现超高并发的无锁缓存？

原创 2016-07-29 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959502&idx=1&sn=7a1c7d81a8e030856fb920844dc571c6&scene=21" \l "#)

### ****一、需求缘起****

**【业务场景】**

有一类写多读少的业务场景：大部分请求是对数据进行修改，少部分请求对数据进行读取。

例子1：滴滴打车，某**个司机地理位置信息的变化**（可能每几秒钟有一个修改），以及司机地理位置的读取（用户打车的时候查看某个司机的地理位置）。

void **SetDriverInfo**(long driver\_id, DriverInfoi); // 大量请求调用修改司机信息，可能主要是GPS位置的修改

DriverInfo **GetDriverInfo**(long driver\_id);  // 少量请求查询司机信息

例子2：**统计计数的变化**，某个url的访问次数，用户某个行为的反作弊计数（计数值在不停的变）以及读取（只有少数时刻会读取这类数据）。

void **AddCountByType**(long type); // 大量增加某个类型的计数，修改比较频繁

long **GetCountByType**(long type); // 少量返回某个类型的计数

**【底层实现】**

具体到底层的实现，往往是一个Map（本质是一个**定长key，定长value**的缓存结构）来存储司机的信息，或者某个类型的计数。

Map<driver\_id, DriverInfo>

Map<type, count>

**【临界资源】**

这个Map存储了所有信息，当并发读写访问时，它作为临界资源，在读写之前，一般要进行加锁操作，以司机信息存储为例：

void SetDriverInfo(long driver\_id, DriverInfoinfo){

**WriteLock (m\_lock);**

         Map<driver\_id>= info;

**UnWriteLock(m\_lock);**

}

DriverInfo GetDriverInfo(long driver\_id){

         DriverInfo t;

**ReadLock(m\_lock);**

         t= Map<driver\_id>;

**UnReadLock(m\_lock);**

         return t;

}

**【并发锁瓶颈】**

假设滴滴有100w司机同时在线，每个司机没5秒更新一次经纬度状态，那么每秒就有20w次写并发操作。假设滴滴日订单1000w个，平均每秒大概也有300个下单，对应到查询并发量，可能是1000级别的并发读操作。

上述实现方案没有任何问题，但**在并发量很大的时候（每秒20w写，1k读），锁m\_lock会成为潜在瓶颈，**在这类高并发环境下写多读少的业务仓井，如何来进行优化，是本文将要讨论的问题。

### ****二、水平切分+锁粒度优化****

上文中之所以锁冲突严重，是因为所有司机都公用一把锁，锁的粒度太粗（可以认为是一个数据库的“库级别锁”），是否可能进行水平拆分（类似于数据库里的分库），把一个库锁变成多个库锁，来提高并发，降低锁冲突呢？显然是可以的，把1个Map水平切分成多个Map即可：

void SetDriverInfo(long driver\_id, DriverInfoinfo){

         i= driver\_id % N; // 水平拆分成N份，N个Map，N个锁

**WriteLock (m\_lock [i]);  //锁第i把锁**

         Map[i]<driver\_id>= info;  // 操作第i个Map

**UnWriteLock (m\_lock[i]); // 解锁第i把锁**

}

每个Map的并发量（变成了1/N）和数据量都降低（变成了1/N）了，所以理论上，锁冲突会成平方指数降低。

分库之后，仍然是库锁，有没有办法变成数据库层面所谓的“行级锁”呢，难道要把x条记录变成x个Map吗，这显然是不现实的。

### ****三、MAP变Array+最细锁粒度优化****

假设driver\_id是递增生成的，并且缓存的内存比较大，是可以把Map优化成Array，而不是拆分成N个Map，是有可能把锁的粒度细化到最细的（每个记录一个锁）。

void SetDriverInfo(long driver\_id, DriverInfoinfo){

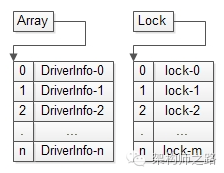
         index= driver\_id;

**WriteLock (m\_lock [index]);  //超级大内存，一条记录一个锁，锁行锁**

         Array[index]= info; //driver\_id就是Array下标

**UnWriteLock (m\_lock[index]); // 解锁行锁**

}



和上一个方案相比，这个方案使得锁冲突降到了最低，但锁资源大增，在数据量非常大的情况下，一般不这么搞。数据量比较小的时候，可以一个元素一个锁的（典型的是连接池，每个连接有一个锁表示连接是否可用）。

上文中提到的另一个例子，用户操作类型计数，操作类型是有限的，即使一个type一个锁，锁的冲突也可能是很高的，还没有方法进一步提高并发呢？

### ****四、把锁去掉，变成无锁缓存****

**【无锁的结果】**

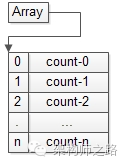
void AddCountByType(long type /\*, int count\*/){

         //不加锁

         Array[type]++; // 计数++

         //Array[type] += count; // 计数增加count

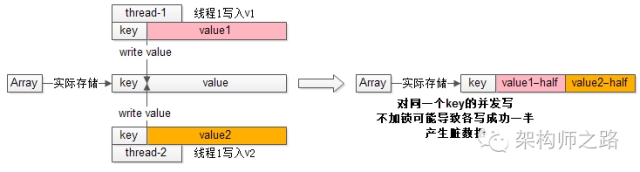
}



如果这个缓存不加锁，当然可以达到最高的并发，但是多线程对缓存中同一块定长数据进行操作时，有可能出现不一致的数据块，这个方案为了提高性能，牺牲了一致性。在读取计数时，获取到了错误的数据，是不能接受的（作为缓存，允许cache miss，却不允许读脏数据）。

**【脏数据是如何产生的】**

这个并发写的脏数据是如何产生的呢，详见下图：



1）线程1对缓存进行操作，对key想要写入value1

2）线程2对缓存进行操作，对key想要写入value2

3）如果不加锁，线程1和线程2对同一个定长区域进行一个并发的写操作，**可能每个线程写成功一半，导致出现脏数据产生**，最终的结果即不是value1也不是value2，而是一个乱七八糟的不符合预期的值value-unexpected。

**【数据完整性问题】**

并发写入的数据分别是value1和value2，读出的数据是value-unexpected，数据的篡改，这本质上是一个数据完整性的问题。**通常如何保证数据的完整性呢？**

**例子1**：运维如何保证，从中控机分发到上线机上的二进制没有被篡改？

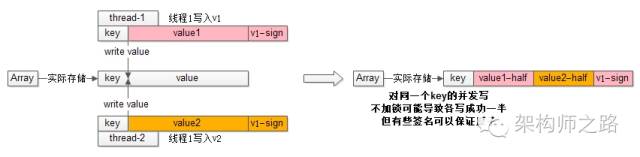
**回答**：md5

**例子2**：即时通讯系统中，如何保证接受方收到的消息，就是发送方发送的消息？

**回答**：发送方除了发送消息本身，还要发送消息的签名，接收方收到消息后要校验签名，以确保消息是完整的，未被篡改。

当当当当 => “签名”是一种常见的保证数据完整性的常见方案。

**【加上签名之后的流程】**



加上签名之后，不但缓存要写入定长value本身，还要写入定长签名（例如16bitCRC校验）：

1）线程1对缓存进行操作，对key想要写入value1，写入签名v1-sign

2）线程2对缓存进行操作，对key想要写入value2，写入签名v2-sign

3）如果不加锁，线程1和线程2对同一个定长区域进行一个并发的写操作，可能每个线程写成功一半，导致出现脏数据产生，最终的结果即不是value1也不是value2，而是一个乱七八糟的不符合预期的值value-unexpected，但签名，一定是v1-sign或者v2-sign中的任意一个

4）数据读取的时候，不但要取出value，还要像消息接收方收到消息一样，校验一下签名，**如果发现签名不一致，缓存则返回NULL，即cache miss**。

当然，对应到司机地理位置，与URL访问计数的case，除了内存缓存之前，肯定需要timer对缓存中的数据定期落盘，写入数据库，如果cache miss，可以从数据库中读取数据。

### ****五、总结****

在【超高并发】，【写多读少】，【定长value】的【业务缓存】场景下：

1）可以通过**水平拆分**来降低锁冲突

2）可以通过**Map转Array**的方式来最小化锁冲突，一条记录一个锁

3）可以**把锁去掉**，最大化并发，但带来的数据完整性的破坏

4）可以通过签名的方式保证数据的完整性，实现**无锁缓存**

帮忙随手转发哟。

==【完】==

回【钱包】[入驻微信钱包的技术优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959423&idx=1&sn=19d27145beab5b0238280e2e3804e41b&scene=21" \l "wechat_redirect" \t "_blank)

回【消息】[58到家通用实时消息平台架构细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "wechat_redirect" \t "_blank)

回【机房】[从IDC到云端架构迁移之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959495&idx=1&sn=0f8789d6803f94e190ba2787389e6d1b&scene=21" \l "wechat_redirect" \t "_blank)

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

回【单点】[单点系统架构的可用性与性能优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959480&idx=1&sn=337bd74410a6bef616128fd17abd08a8&scene=21" \l "wechat_redirect" \t "_blank)

回【事务】[多库多事务降低数据不一致概率](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959486&idx=1&sn=cd3e6b27af159f4eabd54f148eec638a&scene=21" \l "wechat_redirect" \t "_blank)

## “id串行化”到底是怎么实现的？

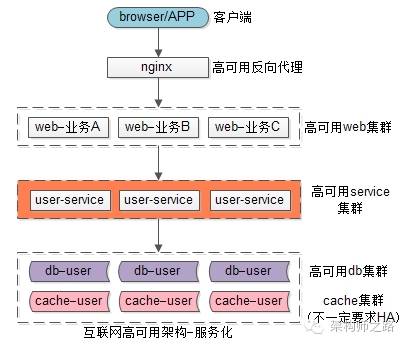
原创 2016-11-17 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959668&idx=1&sn=d4c13071ab619c891efa03ba4e05334e&chksm=bd2d04a88a5a8dbe4ea606c4970be60c5f5c00f3e4cceeb91b26c8975b45678440c516012e79&scene=21" \l "#)

**一、需求缘起**

在上一篇文章《[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)》中，介绍了一种为了保证“所有群友展示的群消息时序都是一致的”所使用的**“id串行化**”的方法：让同一个群gid的所有消息落在同一台服务器上处理。

有朋友就要问了，**如何保证一个群gid的消息落到同一个服务器处理呢**，“**id串行化”具体是怎么实现的呢**，这个问题在年初的一篇文章中描述过，这里再给有疑问的同学解答一下。

**二、互联网高可用常见分层架构**

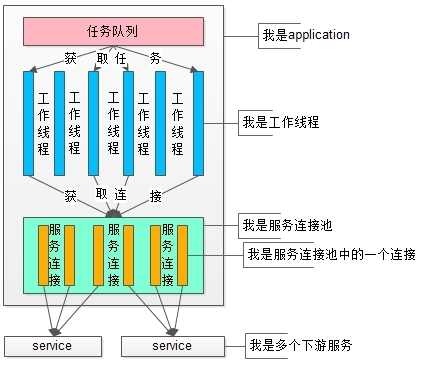


**客户端**，**反向代理层**，**接入层**（此图是http短链接接入，群聊消息的话是tcp长连接接入），**服务层**（处理群消息业务逻辑），**存储层**（缓存cache存储，固化db存储），这是互联网常见的高可用分层架构。

**服务层**的引入至关重要，群消息的投递不能保证落在同一个接入层，但可以保证落在同一个服务层。

**三、服务层上下游细节**

服务化的service一般由RPC-server框架实现，上游应用是多线程程序（站点层http接入应用，或者长连接tcp接入应用）一般通过RPC-client访问service，而RPC-client内部又通过连接池connection-pool访问下游的service（为了保证高可用，是一个service集群）。



如上图：

（1）上游是业务应用（站点层http接入应用，或者长连接tcp接入应用）

（2）下游是service集群

（3）业务应用，它又分为了这么几个部分

（3.1）最上层是任务队列【或许web-server例如tomcat帮你干了这个事情了】

（3.2）中间是工作线程【或许web-server的工作线程或者cgi工作线程帮你干了线程分派这个事情了】，每个工作线程完成实际的业务任务，典型的工作任务是通过服务连接池进行RPC调用

（3.3）最下层是服务连接池，所有的RPC调用都是通过服务连接池往下游服务去发包执行的

工作线程的典型工作流伪代码是这样的：

void work\_thread\_routine(){

Task t = TaskQueue.pop(); // 获取任务

// 任务逻辑处理，组成一个网络包packet，调用下游RPC接口

ServiceConnection c = CPool.GetServiceConnection();

// 从Service连接池**获取一个Service连接**

c.Send(packet); // 通过Service连接发送报文**执行RPC请求**

CPool.PutServiceConnection(c); // **将Service连接放回Service连接池**

}

**如何保证同一个群gid的消息落在同一个service上呢？**

只要对服务连接池进行少量改动：

获取Service连接的CPool.GetServiceConnection()【返回任何一个可用Service连接】**改为**

CPool.GetServiceConnection(long id)【返回id取模相关联的Service连接】

只要传入群gid，就能够保证同一个群的请求获取到同一个连接，从而使请求落到同一个服务Service上。

需要注意的是，连接池不关心传入的long id是什么业务含义：

（1）传入群gid，同gid的请求落在同一个service上

（2）传入用户uid，同uid的请求落在同一个service上

（3）传入任何业务xid，同业务xid的请求落在同一个service上

**四、其他问题**

提问：id串行化访问service，同一个id访问同一个service，当service挂掉时，**是否会影响service的可用性？**

答：不会，当有下游service挂掉的时候，service连接池能够检测到连接的可用性，取模时要把不可用的服务连接排除掉。

提问：取模访问service，**是否会影响各连接上请求的负载均衡？**

答：不会，只要数据访问id是均衡的，从全局来看，由id取模获取各连接的概率也是均等的，即负载是均衡的。

**五、总结**

升级RPC-client内部的连接池，在service连接选取上做微小改动，**就能够实现“id串行化”**，实现不同类型的业务gid/uid等的串行化、序列号需求（这下查找日志就方便了，一个群gid/用户uid的日志只需去一台机器grep啦）。

==【完】==

回【http】[http如何像tcp一样实时的收消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959605&idx=1&sn=21f25087bef3c3a966ef03b824365621&scene=21" \l "wechat_redirect" \t "_blank)

回【可靠】[微信为什么不丢消息？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959606&idx=1&sn=f9561231dd33bcd0550b8d0d59d6b876&chksm=bd2d04ea8a5a8dfce90c870279a7f74b7aedd802c2d699dd919d7e40ebe30699381517c2d54b&scene=21" \l "wechat_redirect" \t "_blank)

回【状态】[QQ状态同步究竟是推还是拉？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959618&idx=1&sn=912a9af6d000c6681dc645e53590729b&chksm=bd2d049e8a5a8d884f6acb35bc5b40edf88127219542c9e3033bb4c2e33854657c315c8a93b7&scene=21" \l "wechat_redirect" \t "_blank)

回【群】[群消息这么复杂，怎么能做到不丢不重？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959643&idx=1&sn=844afa6a31770fa587474ecd73c3b3b3&chksm=bd2d04878a5a8d91c5c93ad8e85254185c63eb419457efbedcba54a9b8a9053da1e8980a694a&scene=21" \l "wechat_redirect" \t "_blank)

回【qq】[微信多点登录与QQ消息漫游架构](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959651&idx=1&sn=959540719a8336bd6f507a324dc7219a&chksm=bd2d04bf8a5a8da9ddfaeb106b61386adeb51c2ad7d4a32589c9a9e113f9341604a490eebb98&scene=21" \l "wechat_redirect" \t "_blank)

回【离线】[微信为啥不丢“离线消息”？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959629&idx=1&sn=053d85a862df6e6c01147a1cf95bdbf2&chksm=bd2d04918a5a8d87a32305fa8ef1603bb0e73e5c4b78afb5a144327841e08840188c4541b4aa&scene=21" \l "wechat_redirect" \t "_blank)

## 从IDC到云端架构迁移之路（GITC2016）

原创 2016-07-22 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959495&idx=1&sn=0f8789d6803f94e190ba2787389e6d1b&scene=21" \l "#)

大家好，很高兴来到GITC2016的舞台，我是来自58到家的沈剑，今天我分享的主题是《58到家从IDC到云端架构迁移之路》。

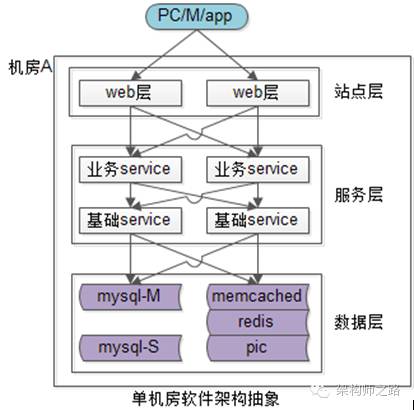
机房迁移是一个很大的动作：

15年在58同城实施过一次（“逐日”项目），几千台物理机，从IDC迁到了腾讯的天津机房，项目做了10个多月，跨所有的部门，与所有的业务都相关；

16年在58到家又实施了一次（“凌云”项目），几百台虚拟机，从IDC迁到阿里云，前后大概一个季度的时间，也是所有技术部门都需要配合的一个大项目。

### ****“单机房架构-全连”****

要说机房迁移，先来看看被迁移的系统是一个什么样的架构。



上图是一个典型的互联网单机房系统架构：

（1）上游是**客户端**，PC浏览器或者APP；

（2）然后是**站点接入层**，为了面对高流量，保证架构的高可用，站点冗余了多份；

（3）接下来是服务层，服务层又分为与业务相关的**业务服务**，以及业务无关的**基础服务**，为了保证高可用，所有服务也冗余了多份；

（4）底层是数据层，数据层又分为**缓存数据**与**数据库**；

至于为什么要做分层架构，不是今天的重点，不做展开讨论，这是一个典型的互联网单机房分层架构：所有的应用、服务、数据是部署在同一个机房，这个架构有的一个关键词，叫做“全连”：

（1）站点层调用业务服务层，业务服务复制了多少份，上层就要连接多少个服务；

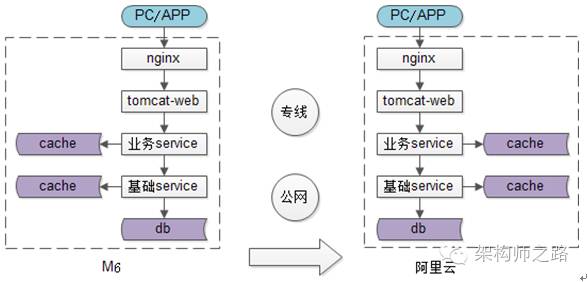
（2）业务服务层调用基础服务层，基础服务复制了多少份，上层就要连多少个服务；

（3）服务层调用数据库，从库冗余了多少份，上层就要连多少个从库；

比如说，站点接入层某一个应用有10台机器，业务服务层某一个服务有8层机器，那肯定是上游的10台会与下游的8台进行一个全相连的。系统架构的**可用性保证**，**负载均衡保证**，是服务的**连接池**去做的。不仅仅接入层连接业务服务层是这样，业务服务层连接基础服务层，服务层连接数据库也都是这样，这就是所谓的“全连”。

### ****“机房迁移的目标是平滑”****

单机房架构的特点是“全连”，那么机房迁移我们是要做一个什么样的事情呢？先看这张图：



之前单机房架构部署在机房A内，迁移之后仍然是单机房架构，只是换了一个B机房，做完这个迁移，有什么好的方案？最容易想到的一个方案，把所有服务在新机房全部搭一套，然后流量切过来了。

当系统有几千台机器，有非常非常多的业务的时候，这是一种“不成功便成仁”的方案。做技术的都知道，设计时要考虑回滚方案，如果只有上线方案而没有回滚方案，这便是一个“不成功便成仁”的方案，根据经验，不成功便成仁的操作结果，往往就“便成仁”了。

最重要的是，全量搭建一套再流量切换，数据层你怎么搭建一套？怎么切？数据层原来都在A机房，B机房还没有全量的数据，是没办法直接切的。要做一个数据同步的方案，最简单的，停两个小时服务，把数据从旧机房导到新机房，数据导完流量再切过去，这是一个数据迁移的简单方案。**这个方案对业务有影响，需要停止服务，这个是无法接受的**，何况像58同城一样有两千多台机器，无限多的数据库实例，无限多的数据表的时候，停服务迁移数据根本是不可能的。

所以，机房迁移的难点，是“平滑”迁移，整个过程不停服务，整体迁移方案的目标是：

**（1）可以分批迁移；**

**（2）随时可以回滚；**

**（3）平滑迁移，不停服务；**

### ****“伪多机房架构-同连”****

如果想要平滑的迁移机房，不停服务，在10个月的逐步迁移过程中，肯定存在一个中间过渡阶段，两边机房都有流量，两边机房都对外提供服务，这就是一个多机房的架构了。

多机房架构是什么样的架构呢？刚刚提到了单机房架构，上层连中层，中层连下层，它是一个全连的架构，能不能直接将单机房的全连架构套用到多机房呢？在另一个机房部署好站点层、服务层、数据层，直接使用“全连”的单机房架构，我们会发现：会有非常多跨机房的连接

（1）站点层连接业务服务层，一半的请求跨机房

（2）业务服务层连接基础服务层，一半的请求跨机房

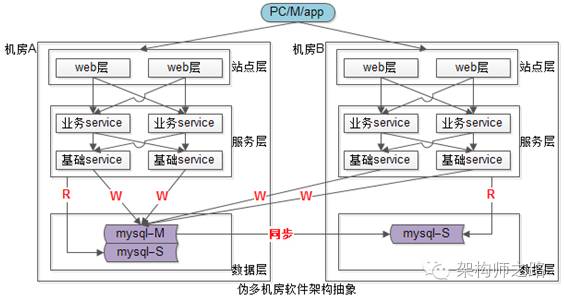
（3）基础服务层连数据层（例如从库），一半的请求跨机房

**大量的跨机房连接会带来什么样的问题呢？**

我们知道，同机房连接，内网的性能损耗几乎可以忽略不计，但是一旦涉及到跨机房的访问，**即使机房和机房之间有专线，访问的时延可能增加到几毫秒**（跟几房间光纤距离有关）。

用户访问一个动态页面，需要用到很多数据，这些数据可能需要10次的业务服务层调用，业务服务层可能又有若干次基础服务层的调用，基础服务层可能又有若干次数据层的调用，假设整个过程中有20次调用，其中有一半调用跨机房，假设机房之间延迟是5毫秒，因为跨机房调用导致的请求迟延就达到了50毫秒，这个是不能接受的。

因此，在多机房架构设计时，要尽量避免跨机房调用（避免跨机房调用做不到，也要做到“最小化”跨机房调用），会使用“同连”的系统架构。



“同连”也很好理解，**在非必须的情况下，优先连接同机房的站点与服务**：

（1）站点层只连接同机房的业务服务层；

（2）业务服务层只连接同机房的基础服务层；

（3）服务层只连接同机房的“读”库；

（4）对于写库，没办法，只有跨机房读“写”库了；

这个方案没有完全避免跨机房调用，但其实它做到了“最小化”跨机房调用，**写主库是需要跨机房的**。但互联网的业务，99%都是读多写少的业务，例如百度的搜索100%是读业务，京东淘宝的电商99%的浏览搜索是读业务，只有下单支付是写业务，58同城99%帖子的列表详情查看是读业务，发布帖子是写业务，写业务比例相对少，只有这一部分请求会跨机房调用。

**迁移机房的过程使用这样一个多机房的架构**，最大的好处就是，除了“配置文件”，整个单机房的架构不需要做任何修改，这个优点是很诱人的，所有的技术部门，所有的业务线，只需要配合在新机房部署应用与服务（数据库是DBA统一部署的），然后使用不同的配置文件（如果有配置中心，这一步都省了），就能实现这个迁移过程，大大简化了迁移步骤。

这个方案当然也有它的**不足**：

（1）跨机房同步数据，会多5毫秒（举个栗子，不要叫真这个数值）延时（主从本来会有延时，这个延时会增大），这个影响的是某一个机房的数据读取；

（2）跨机房写，会多5毫秒延时，这个影响的是某一个机房的数据写入，当然这个写请求比例是很小的；

这个“同连”架构非常适用于做机房迁移，当然也可以用作多机房架构，用作多机房架构时，还有一个缺点：这个架构有“主机房”和“从机房”的区分。

**多机房架构的本意是容机房故障**，这个架构当出现机房故障时，例如一个机房地震了，把入口处流量切到另一个机房就能容错，不过：

（1）挂掉的是不包含数据库主库的从机房，迁移流量后直接容错；

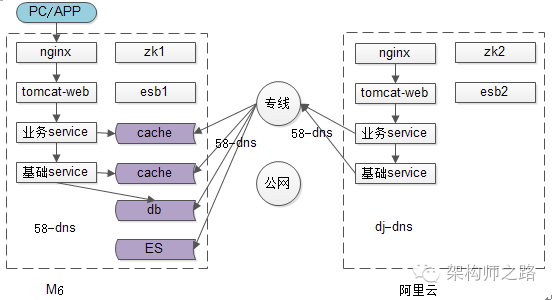
（2）挂掉的是包含数据库主库的主机房，只迁移流量，其实系统整体99%的读请求可以容错，**但1%的写请求其实会受到影响**，此时需要人工介入，将从库变为主库，才能完全容错。这个过程只需要DBA介入，不需要所有业务线上游修改（除非，除非，业务线直接使用的IP连接，这个，我就不说什么了）。

也正是因为这个原因，在机房故障的时候，有一定概率需要少量人工介入，才能容100%的机房故障，因此这个架构才被称为“伪多机房架构”，还不是完全的“多机房多活”架构。

### ****“自顶向下的机房迁移方案”****

话题收回来，机房迁移的过程中，一定存在一个中间过渡阶段，两边机房都有流量，两边机房都对外提供服务的多机房架构。具体到机房的逐步迁移，又是个什么步骤呢？通常有两种方案，一种是自顶向下的迁移，一种是自底向上的迁移，这两种方案在58到家和58同城分别实行过，都是可行的，方案有类似的地方，也有很多细节不一样，因为时间关系展开说一种，在58到家实施过的“自顶向下”的机房迁移方案，整个过程是平滑的，逐步迁移的，可回滚的，对业务无影响的。

**“站点与服务的迁移”**



迁移之前当然要做一些提前准备，新机房要准备就绪，专线要准备就绪，这个是前提。

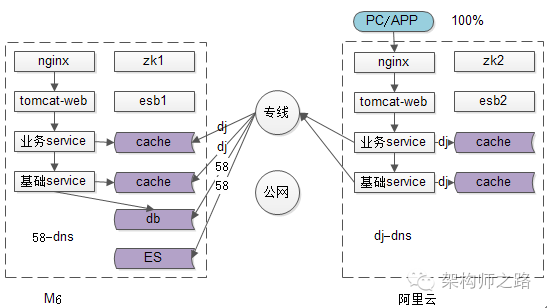
自顶向下的的迁移，最先迁移站点层和服务层：先在新机房，把站点层和服务层搭建好，并做充分的测试（此时数据层比如说缓存和数据库还是在原来的机房）。测试，测试，测试，只要流量没有迁移，在新机房想怎么玩都行，新机房准备的过程中，要注意“同连”，原有机房的配制文件是完全不动的，肯定也是“同连”。

站点层与服务层的迁移，也是一个业务一个业务的逐步迁移的，类似蚂蚁搬家。充分的测试完一个业务的站点层和服务层之后，为了求稳，先切1%的流量到新机房，观察新机房的站点与服务有没有异常，没有问题的话，再5%，10%，20%，50%，100%的逐步放量，直至第一波蚂蚁搬完家。

第一个业务的站点和服务迁移完之后，第二个业务、第三个业务，蚂蚁继续搬家，直至所有的业务把站点层和服务层都全流量的迁移到新机房。

在整个迁移的过程中，任何一个业务，任何时间点发现有问题，可以将流量切回，旧机房的站点、服务、配置都没有动过，依然能提供服务。整个迁移步骤，是比较保险的，有问题随时可以迁回来。

**“缓存的迁移”**



站点层和服务层迁移完之后，接下来我们迁数据层，数据层又分为缓存层和数据库层，先迁缓存。

经过第一步的迁移，所有的入口流量都已经迁到了新的机房（当然旧机房的站点和服务还是不能停，只要旧机房不停，任何时间点出问题，最坏的情况下流量迁回来），接下来迁移缓存，先在新机房要搭建好缓存，缓存的规模和体量与旧机房一样大。

流程上仍然是蚂蚁搬家，按照业务线逐步的迁缓存，使用同连的方式。这个**缓存切换的步骤非常的简单**：运维做一个缓存内网DNS的切换（内网域名不变，IP切到新机房），并杀掉原有缓存连接，业务线不需要做任何修改，只需要配合观察服务。运维杀掉原有缓存连接之后，程序会自动重连，重连上的缓存就是新机房的缓存了，bingo，迁移完毕。

这里要注意几个点：

（1）有些公司缓存没有使用内网域名，而是采用IP直连的话，则需要业务层配合，换新机房IP重启一下即可（如果是IP直连，说明这个架构还有改进的空间哟）；

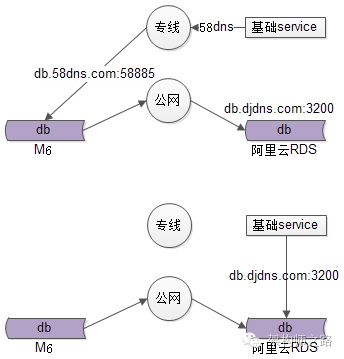
（2）这个操作尽量选在流量低峰期，旧缓存中都是热数据，而新缓存是空数据，如果选在流量高峰期，缓存切换之后，短时间内可能会有大量请求透传到数据库上去，导致数据库压力过大；

（3）这个通用步骤，适用于允许cache miss的业务场景，如果业务对缓存有高可用的要求，不允许cache miss，则需要双写缓存，或者缓存使用主从同步的架构。大部分缓存的业务场景都是允许cache miss的，少数特殊业务使用特殊的方案迁移。

缓存的迁移也是按照业务线，一步步蚂蚁搬家式完成的。在迁移过程中，任何一个业务，任何时间点发现有问题，可以将流量切回原来的缓存。所以迁移的过程中，不仅是站点层和服务层，旧机房的缓存层也是不停服务的，至少保证了流量迁回这个兜底方案。

**“数据库的迁移”**

站点层，服务层，缓存层都迁移完之后，最后是数据库的迁移。



数据库还是在旧机房，其他的缓存，服务，站点都迁移到新机房了，服务通过专线跨机房连数据库。

如何进行数据库迁移呢，首先肯定是在新机房搭建新的数据库，如果是自建的IDC机房，需要自己搭建数据库实例，58到家直接用的是阿里云的RDS。

搭建好数据库之后，接下来进行数据同步，自建机房可以使用数据库MM/MS架构同步，阿里云可以使用DTS同步，DTS同步有一个大坑，只能使用公网进行同步，但问题也不大，只是同步的时间比较长（不知道现能通过专线同步数据了吗？）。

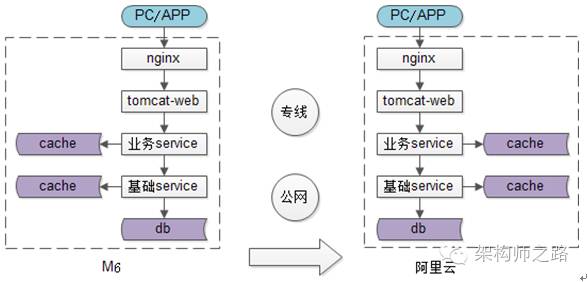
数据库同步完之后，如何进行切换和迁移呢？能不能像缓存的迁移一样，运维改一个数据库内网DNS指向，然后切断数据库连接，让服务重连新的数据库，这样业务服务不需要改动，也不需要重启，这样可以么？

这个方式看上去很不错，但数据库的迁移没有那么理想：

第一，得保证数据库同步完成，才能切流量，但数据同步总是有迟延的，旧机房一直在不停的写如数据，何时才算同步完成呢？

第二，只有域名和端口不发生变化，才能不修改配置完成切换，但如果域名和端口（主要是端口）发生变化，是做不到不修改配置和重启的。举个例子，假设原有数据库实例端口用了5858，很吉利，而阿里云要求你使用3200，就必须改端口重启。

所以，我们最终的迁移方案，是DBA在旧机房的数据库设置一个read only，停止数据的写入，在秒级别，RDS同步完成之后，业务线修改数据库端口，重启连接新机房的数据库，完成数据层的切换。



经过上述站点、服务、缓存、数据库的迁移，我们的平滑机房的目标就这么一步步完成啦。

### ****总结与问答****

四十分钟很短，focus讲了几个点，希望大家有收获。

做个简要的总结：

（1）互联网单机房架构的特点，全连，站点层全连业务服务层，业务服务层全连的基础服务层，基础服务层全连数据库和缓存；

（2）多机房架构的特点，同连，接入层同连服务层，服务层同连缓存和数据库，架构设计上最大程度的减少跨机房的调用；

（3）**自顶向下的机房迁移方案**：先进行站点接入层、业务服务层和基础服务层的迁移，搭建服务，逐步的迁移流量；然后是缓存迁移，搭建缓存，运维修改缓存的内网DNS指向，断开旧连接，重连新缓存，完成迁移；最后数据库的迁移，搭建数据库，数据进行同步，只读，保证数据同步完成之后，修改配置，重启，完成迁移。整个过程分批迁移，一个业务线一个业务线的迁移，一块缓存一块缓存的迁移，一个数据库一个数据库的迁移，任何步骤出现问题是可以回滚的，整个过程不停服务。

**主持人**：讲的很细致，大家有什么问题吗，可以提一些问题，可以举手示意我。

**提问**：做数据迁移的时候，因为您讲的数据中心的都是在同一个老机房，同时又在做同步，我就在想这个数据库的压力是不是特别大。

**沈剑**：非常好的问题，这个地方一方面要考虑压力，更重要的是考虑跨机房的专线，风险最大的是在带宽这一部分，你在第一步迁移完之后，其实所有的缓存，数据库用其实都是跨机房的，都是通过专线去走的，这个专线带宽是需要重点考虑与评估的，数据库的压力其实还好。

**提问**：我想请教一个问题，你这个流量切换的过程中，有测试性的阶段还是直接切过去的。

**沈剑**：在切流量之前，肯定是有测试的，在新机房将服务搭建，在切换流量之前，测试的同学需要进行回归，回归的过程可以提前发现很多问题。逐步的切流量也是为了保证可靠性，我们不是一次性百分之百流量都切过来，先切1%的流量过来，观察服务没有问题，再逐步增大流量切换。

**主持人**：上午先到这里，也欢迎大家关注沈老师的“架构师之路”微信公众号，下午我们准时开始，大家注意好休息时间，谢谢大家。

帮忙随手转发哟。

==【完】==

回【钱包】[入驻微信钱包的技术优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959423&idx=1&sn=19d27145beab5b0238280e2e3804e41b&scene=21" \l "wechat_redirect" \t "_blank)

回【消息】[58到家通用实时消息平台架构细节](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959410&idx=1&sn=b91b5721ca394d15fb391097eddb752d&scene=21" \l "wechat_redirect" \t "_blank)

回【秒杀】[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959391&idx=1&sn=fb28fd5e5f0895ddb167406d8a735548&scene=21" \l "wechat_redirect" \t "_blank)

回【设置】[线程数究竟设多少合理](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404369373&idx=1&sn=cab2ac9ec0edf92e744a2329662b16cd&scene=21" \l "wechat_redirect" \t "_blank)

回【单点】[单点系统架构的可用性与性能优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959480&idx=1&sn=337bd74410a6bef616128fd17abd08a8&scene=21" \l "wechat_redirect" \t "_blank)

回【事务】[多库多事务降低数据不一致概率](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959486&idx=1&sn=cd3e6b27af159f4eabd54f148eec638a&scene=21" \l "wechat_redirect" \t "_blank)

## 一张“神图”看懂单机/集群/热备/磁盘阵列（RAID）

2015-11-10 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400720056&idx=1&sn=10c796793e27d475603f97b23e137c12&scene=21" \l "#)



**单机部署**（stand-alone）：只有一个饮水机提供服务，服务只部署一份

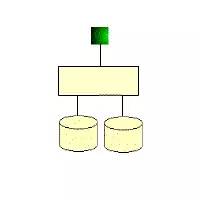
**集群部署**（cluster）：有多个饮水机同时提供服务，服务冗余部署，每个冗余的服务都对外提供服务，一个服务挂掉时依然可用

**热备部署**（hot-swap）：只有一个桶提供服务，另一个桶stand-by，在水用完时自动热替换，服务冗余部署，只有一个主服务对外提供服务，影子服务在主服务挂掉时顶上

磁盘阵列RAID（Redundant Arrays of independent Disks）

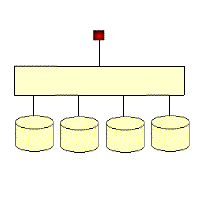


**RAID0**：存储性能高的磁盘阵列，又称striping，它的原理是，将连续的数据分散到不同的磁盘上存储，这些不同的磁盘能同时并行存取数据（速度块）



**RAID1**：安全性高的磁盘阵列，又称mirror，它的原理是，将数据完全复制到另一个磁盘上，磁盘空间利用率只有50%（冗余，数据安全）

**RAID0+1**：RAID0和RAID1的综合方案，这也是**国企**用的比较多的存储方案（速度快，安全性又高，但是很贵）



**RAID5**：RAID0和RAID1的折衷方案，读取速度比较快（不如RAID0，因为多存储了校验位），安全性也很高（可以利用校验位恢复数据），空间利用率也不错（不完全复制，只冗余校验位），这也是**互联网公司**用的比较多的存储方案

===【完】===

## 一分钟学awk够用（产品经理都懂了）

原创 2015-11-11 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400743254&idx=1&sn=147abc381ccd61e28f52699735c8748e&scene=21" \l "#)

1分钟懂awk-技不在深，够用就行

**1.什么是AWK**

（1）Aho、Weinberger、Kernighan三位发明者名字首字母；

（2）一个行文本处理工具；

**2.AWK基本原理**

2.1**原理**：逐行处理文件中的数据

2.2**语法**：

|  |  |
| --- | --- |
|  | awk 'pattern + {action}' |

说明：

（1）单引号''是为了和shell命令区分开；

（2）大括号{}表示一个命令分组；

（3）pattern是一个过滤器，表示命中pattern的行才进行action处理；

（4）action是处理动作；

（5）使用#作为注释；

例子：显示hello.txt中的第3行至第5行

|  |  |
| --- | --- |
|  | cat hello.txt | awk 'NR==3, NR==5{print;}' |

2.3**pattern说明**

pattern参数可以是egrep正则中的一个，正则使用/pattern/

例子：显示hello.txt中，正则匹配hello的行

|  |  |
| --- | --- |
|  | cat hello.txt | awk '/hello/' |

说明：

（1）pattern和action可以只有其一，但不能两者都没有；

（2）默认的action是print；

例子：显示hello.txt中，长度大于100的行号

|  |  |
| --- | --- |
|  | cat hello.txt | awk 'length($0)>80{print NR}' |

**3.内置变量**

FS 分隔符，默认是空格

NR 当前行数，从1开始

NF 当前记录字段个数

$0 当前记录

$1~$n 当前记录第n个字段

例子：显示hello.txt中的第3行至第5行的第一列与最后一列

|  |  |
| --- | --- |
|  | cat hello.txt | awk 'NR==3, NR==5{print $1,$NF}' |

**4.内置函数**

gsub(r,s)：在$0中用s代替r

index(s,t)：返回s中t的第一个位置

length(s)：s的长度

match(s,r)：s是否匹配r

split(s,a,fs)：在fs上将s分成序列a

substr(s,p)：返回s从p开始的子串

**5.操作符**

**5.1运算符**

类似于c，支持+、-、\*、/、%、++、–、+=、-=等诸多操作；

**5.2判断符**

类似于c，支持==、!=、>、=>、~（匹配于）等诸多判断操作；

**6.控制流程**

**6.1.BEGIN和END**

BEGIN和END本质是一个pattern。

BEGIN用于awk程序开始开始前，做一些初始化的工作；

END用于awk程序结束前，做一些收尾的工作。

例子：统计字符个数

|  |  |
| --- | --- |
|  | awk '  BEGIN  {  count=0;  }  {  count+=length($0);  }  END  {  print count;  }' |

**6.2流程控制语句**

（1）if(condition){}else{}

（2）while{}

（3）do{}while(condition);

（4）for(init;condition;step){}

（5）break/continue：如果有END，会执行END中的收尾工作

个流程控制语句用法几乎与c相同。

**7.awk与shell的交互**

（1）awk中使用shell中定义的变量：使用单引号即可；

|  |  |
| --- | --- |
|  | #!/bin/bash  STR="hello"  echo | awk '{  print "'${STR}'";  }' |

（2）awk中使用shell命令：使用双引号，或者system命令；

|  |  |
| --- | --- |
|  | #!/bin/bash  echo hello | awk '{  print $0 | "cat"  }' |

或者

|  |  |
| --- | --- |
|  | #!/bin/bash  echo | awk '{  system("date > date.txt")  }' |

（3）awk中的变量传出至shell：没有什么好方法，老老实实用文件吧；

（4）getline：awk里，从文件中读取变量到awk中

|  |  |
| --- | --- |
|  | #!/bin/bash  echo | awk '{  while(getline < "date.txt")  {  print $0;  }  }' |

**8.结束语**

对不起，楼主欺骗了你，认真看完本文或许不止1分钟。不过，如果你真的认真阅读并超过了1分钟，相信你会有收获。

==【完】==



实时好文，欢迎长按关注“架构师之路”

==好文推荐==

回【join】秒懂SQL中各种join

回【追查】Linux下线上问题常用追查命令

回【微信】微信为何这么省流量（火）

回【秒杀】秒杀系统架构优化思路（火）

回【招聘】入职58到家

回【设计】数据库架构师该设计什么（新）

## 十分钟学perl够用（客服MM都懂了）

原创 2015-11-12 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400764946&idx=1&sn=4efab1953696c2e21071e58a74da9b82&scene=21" \l "#)

**零基础，perl语言，10分钟入门**

**1.Hello,World**

|  |  |
| --- | --- |
| 1  2  3 | #!/usr/bin/perl -w  print ("hello,world!\n");  #print "hello,world!\n"; |

说明：

（1）第一行指定解释器，-w参数表示提示警告（或者使用use strict命令，执行更严格的检查）；

（2）第二行输出hello, world!；

（3）如果习惯c的函数方式，print的参数可以打括号；

（4）第三行是注释，注释以#打头；

（5）如果习惯shell的方式，print的参数可以没有括号；

（6）双引号内可以使用转义字符；

不妨设文件名为helloworld.pm

程序的执行方法为：

（1）perl helloworld.pm

（2）chmod 755 helloworld.pm && ./helloworld.pm

**2.常量**

2.1数字

（1）Perl内部总按照“双精度浮点数”保存数字并执行运算；

（2）0377=>八进制；0xFF=>十六进制；

2.2字符串

（1）单引号表示字符串，不转义；

（2）双引号表示字符串，转义且解释变量；

2.3字符串操作符

（1）拼接操作符：“.”=>拼接字符串；

（2）重复操作符：“x”=>一个字符串重复多次；

|  |  |
| --- | --- |
| 1  2  3 | #!/usr/bin/perl -w  print ("hello,"."world!\n");  print ("hello " x 3); |

输出结果是：

hello,world!

hello hello hello

最后要说明一点，Perl是弱类型语言，字符串和数字会相互转化，这一点和php一样。

**3.变量**

（1）变量以$开头，后接一个标示符；

（2）如何用变量获取用户输入？

使用，它获取用户的输入（一般以换行结束），可以使用chomp去除结尾的换行符。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #!/usr/bin/perl -w  $count = 0;  while($count<10)  {  chomp($input = );  print($input);  $count++;  } |

（3）未定义变量

未定义的变量会赋予undef值，它既不是数字，也不是字符串；

它有可能被当做数字0使用；

使用define函数可以知道一个变量是否被定义；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #!/usr/bin/perl -w  $var = undef;  print($var);  if(defined($var))  {  print("defined!\n");  }  else  {  print("undefined!\n");  }  $var++;  print($var); |

它的输出是：

Use of uninitialized value in print at undef.pm line 3.

undefined!

1

（4）变量的作用域

my和our可以指定变量的作用域

my指定为局部作用域；

our指定为全局作用域（默认为our）；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #!/usr/bin/perl -w  our $g\_one = "global\_one\n";  $g\_two = "global\_two\n";  {  my $local\_one = "local\_one\n";  print($g\_one);  print($g\_two);  print($local\_one);  }  print($g\_one);  print($g\_two);  print($local\_one); |

输出为：

global\_one

global\_two

local\_one

global\_one

global\_two

Use of uninitialized value in print at our\_my.pm line 13.

**4.数组与列表**

4.1数组

和c的数组使用非常类似：

$array[0]=”a0″;

$array[1]=”a1″;

$array[2]=”a2″;

4.2列表

圆括号内的一系列值，构成列表：

(1, 2, 3)

(“hello”, 4)

(“hello”, “world”, “yes”, “no”)

qw(hello world yes no)

(1..10)

说明：

（1）第一行，列表元素为1,2,3；

（2）第二行，列表元素为一个字符串，一个数字；

（3）第三行，列表元素为4个字符串，好多引号和逗号啊；

（4）第四行，wq操作符，用来建立字符串列表，而不用输入这么多引号和逗号，效果同（3）；

（5）范围操作符“..”，表示一个范围，从左至右连续加一。

列表的赋值：

($v1, $v2, $v3) = qw(yes i am);

整个列表的引用，@操作符：

@list = qw(yes i am);

@none = ();

@huge = (1..5);

@stuff = (@list, @none, @huge);

pop和push操作符：

（1）pop弹出列表末端元素；

（2）push向列表末端压入元素；

shift和unshift操作符：

（1）shift移出列表首部元素；

（2）unshift向列表首部压入元素；

列表的输出：

（1）列表输出，只输出列表，元素间不含空格；

（2）列表的字符串化输出，输出列表，元素间加入空格；

（3）foreach控制结果，可以依次取得列表中各个元素

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #!/usr/bin/perl -w  @list = qw(yes i am);  @none = ();  @huge = (1..5);  @stuff = (@list, @none, @huge);  $pop\_last = pop(@stuff);  print($pop\_last);  push(@stuff, "hello");  $shift\_first = shift(@stuff);  print($shift\_first);  unshift(@stuff, "world");  print(@stuff);  print("@stuff");  $element=undef;  foreach $element (@stuff)  {  print("$element!\n");  } |

输出：

5

yes

worldiam1234hello

world i am 1 2 3 4 hello

i!

am!

1!

2!

3!

4!

hello!

4.3默认变量$\_

该使用变量的地方，如果省略变量，则会使用默认变量$\_。

|  |  |
| --- | --- |
| 1  2  3 | #!/usr/bin/perl -w  $\_="hello,world！";  print(); |

输出是：

hello,world！

**5.函数**

5.1函数定义与调用

（1）定义函数的关键字是sub；

（2）函数调用的关键字是&；

（3）可用return显示返回，也可用一个数字隐式返回

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #!/usr/bin/perl  $num=0;  sub sumAdd  {  $num+=1;  print("$num\n");  #return $num; # 显示返回  $num; # 隐式返回  }  &sumAdd;  &sumAdd;  print(&sumAdd); |

执行结果为：

1

2

3

3

5.2函数的参数

（1）调用函数时可直接带参数列表；

（2）函数定义处使用“默认变量”获取参数列表；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #!/usr/bin/perl -w  sub max  {  return ($\_[0]>$\_[1]?$\_[0]:$\_[1]);  }  $big=20;  $small=10;  print(&max($big,$small)); |

输出为：

20

**6.程序输入输出**

上文已经介绍过标准输入，下面介绍其他几种常见的输入输出。

6.1Unix工具输入输出：<>

<>提供类似于Unix工具输入输出的功能，它提供的功能能够很好的和cat/sed/awk/sort/grep等工具结合使用。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #!/usr/bin/perl -w  use strict;  while(<>)  {  chomp();  print("$\_!!!\n");  } |

该脚本的功能，是在输入每行后面加上!!!，它几处使用到了默认变量。

不妨设文件名为diamond.pm

不妨设hello.txt中有三行数据，分别是111,222,333

执行步骤：

（1）chmod 755 diamond.pm

（2）cat hello.txt | ./diamond.pm | cat

输出结果：

111!!!

222!!!

333!!!

6.2格式化输出：printf

|  |  |
| --- | --- |
| 1  2  3  4 | #!/usr/bin/perl -w  $int\_var = 2011;  $str\_var = "hello,world";  printf("%d\n%s\n",$int\_var,$str\_var); |

输出结果为：

2011

hello,world

6.3文件输入输出

Perl保留了6个文件句柄：STDIN/STDOUT/STDERR/DATA/ARGV/ARGVOUT

上述6.1中的程序还能这么执行：

./diamond.pm out.txt

则输出结果会重定向到out.txt中

输入输出到文件中中，需要打开、使用、关闭文件句柄

（1）打开文件句柄：

open LOG, “>>log.txt”;

open CONFIG, ” （2）关闭文件句柄：

close LOG;

close CONFIG;

（3）使用文件句柄：

print LOG (“hello,world!\n”);

print STDERR (“yes i am!\n”);

while()

{

chomp();

…

}

也可以使用select关键字：

print(“to stdout1!”);

select LOG;

print(“to log1″);

print(“to log2″);

select STDOUT;

print(“to stdout2!”);

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #!/usr/bin/perl -w  $input\_file = "hello.txt";  $output\_file = "out.txt";  open INPUT, "<$input\_file"; open OUTPUT, ">>$output\_file";  while(  <input type="text">)  {  chomp();  print OUTPUT ("$\_!!!\n");  }  close OUTPUT;  close INPUT; |

说明：他的功能和之前的diamond.pm是一样的。

**7.哈希hash**

7.1哈希的存取

$key=”am”;

$hash\_one{“yes”} = 3;

$hash\_one{“i”} = 1;

$hash\_one{$key} = 5;

print($hash\_one{“am”});

$value = $hash\_one{“hello”}; # undef

7.2哈希的引用

要引用整个哈希，使用%操作符。

%hash\_one = (“hello”,5,”world”,5);

print ($hash\_one{“hello”});

%hash\_two = %hash\_one;

7.3哈希的松绑

哈希可以转化为键值列表，称为哈希的松绑，转化后不保证键的顺序，但值一定在键的后面。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #!/usr/bin/perl -w  %hash\_one = ("hello",5,"world",5);  $hash\_one{"yes"} = 3;  $hash\_one{"i"} = 1;  $hash\_one{"am"} = 2;  @array\_one = %hash\_one;  print ($hash\_one{"hello"});  print ("@array\_one"); |

输出结果为：

5

yes 3 am 2 hello 5 world 5 i 1

7.4哈希的反转

建立值对应键的反转哈希。

%hash\_reverse = reverse(%hash\_one);

只有在键值一一对应的情况下才凑效，否则会有无法预期的覆盖发生。

7.5哈希的美观赋值

哈希的美观赋值使用=>符号。

%hash\_one = (“hello”,5,”world”,5,”yes”,3,”i”,1,”am”,2);

上面这种赋值方式很容易搞错，特别是键值都是字符串的时候。

%hash\_one = (

“hello” => 5,

“world” => 5,

“yes” => 3,

“i” => 1,

“am” => 2,

);

美观赋值，是不是看起来更美观，更容易区分哈什的键值呢。

7.6哈希的遍历

（1）keys和values函数能返回所有键与值的列表，但列表内顺序不保证。

@k = keys(%hash\_one);

@v = values(%hash\_one);

（2）each函数能一一遍历哈希，返回键值对，非常适合于while等循环；

while(($key, $value) = each(%hash\_one))

{

…

}

示例代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | #!/usr/bin/perl -w  %hash\_one = (  "hello" => 5,  "world" => 5,  "yes" => 3,  "i" => 1,  "am" => 2,  );  @k = keys(%hash\_one);  @v = values(%hash\_one);  print("@k\n");  print("@v\n");  $key = undef;  $value = undef;  while(($key, $value) = each(%hash\_one))  {  print("$key=>$value\n");  } |

输出结果为：

yes am hello world i

3 2 5 5 1

yes=>3

am=>2

hello=>5

world=>5

i=>1

7.7哈希的查询与删除

（1）查询一个键是否存在，使用exists函数；

（2）删除一个键，使用delete函数；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #!/usr/bin/perl -w  %hash\_one=(  "yes" => 3,  "i" => 1,  "am" => 2,  );  delete($hash\_one{"yes"});  if(exists($hash\_one{"yes"}))  {  print($hash\_one{"yes"});  } |

结果什么也不输出。

**8.流程控制**\*（本节可跳过，都是些花哨的用法）

除了各语言常用的if/esle，for，while等流程控制外，Perl还有一些特有的控制语句，更人性化。

（1）unless控制结构

作用效果类似于if not，无效率上提升，只是使表达更自然，代码更容易理解。

（2）until控制结构

作用效果类似于while not

（3）条件修饰

判断条件可以直接写在语句的后面，以增加可读性（habadog注：这是鬼扯）。

print (“$n”) if $n < 0; $i \*= 2 until $i > 1024;

&sumAdd($\_) foreach @num\_list;

（4）裸控制结构

只有一个花括号的结构，往往用来限制作用域，在各语言中都很常见。

{

$a = 1;

…

}

# $a失效了

（5）last控制结构

相当于c中的break，立刻终止循环；

（6）next控制结构

相当于c中的continue，立刻开始下一次循环；

（7）redo控制结构

…独有的，重新开始本次循环；

while(1)

{

# 跳到这里

print (“hello”);

redo;

}

**9.高级特性**

神奇的Perl还有正则、module、文件、字符串、智能匹配、进程管理、线程支持等高级特性，就不在入门手册里介绍了。

如果大伙喜欢，后续发布以上特性的手册。

希望你喜欢上Perl。

===【完】===

## 一分钟sed入门（一分钟系列）

2016-01-13 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403340380&idx=1&sn=9b4ced5a19f60b0cebf6fd582f1d52ae&scene=21" \l "#)

**1.简介**

sed是一种行编辑器，它一次处理一行内容。

**2.sed调用方式**

sed [options] 'command' file(s)

sed [options] -f scriptfile file(s)

第一种直接在命令行中执行，第二种把命令写到了脚本中，二者无本质区别。

示例（1）：打印hello.txt的内容

sed -n p hello.txt

说明：

-n：sed会在处理一行文本前，将待处理的文本打印出来，-n参数关闭了这个功能

p：命令表示打印当前行

hello.txt：待处理的文件

这个指令相当于cat

**3.定址**

告诉sed你期望处理的行，由逗号分隔的两个数字表示，$符号表示最后一行；

当然也可以使用正则来定位期望处理的行。

示例（2）：打印hello.txt的第二行到最后一行

sed -n '2,$'p hello.txt

示例（3）：打印hello.txt中正则匹配"100"的行

sed -n '/100/'p hello.txt

**4.基本命令**

hello.txt的内容为

1 2 3

10 20 30

100 200 300

命令：a\

在匹配行的后面加入一行文本

示例（4）匹配100的行，后面加入一行"new line"

sed '/100/'a\ "new line" hello.txt

输出内容为：

1 2 3

10 20 30

100 200 300

new line

命令：i\

在匹配行的前面加入一行文本

示例（5）匹配100的行，前面加入一行"new line"

sed '/100/'i\ "new line" hello.txt

输出内容为：

1 2 3

10 20 30

new line

100 200 300

命令：c\

将匹配行替换为目的行

示例（5）匹配100的行，替换为"new line"

sed '/100/'c\ "new line" hello.txt

输出内容为：

1 2 3

10 20 30

new line

命令：d

将匹配行删除

示例（5）删除匹配100的行

sed '/100/'d hello.txt

输出内容为：

1 2 3

10 20 30

命令：s

将匹配行替换

详细命令为：s/pattern-to-find/replacement-pattern/g

pattern-to-find：被替换的串

replacement-pattern：替换成这个串

g：全部替换，默认只替换匹配到的第一个

示例（5）讲100替换为hello

sed 's/100/hello/g' hello.txt

输出内容为：

1 2 3

10 20 30

hello 200 300

**5.元字符集**

^：匹配一行的开始

$：匹配一行的结束

.：匹配某个字符

[abc]：匹配指定范围字符

**6.实用命令**

匹配以10开头的行，并替换为yes，并输出

sed -n 's/^10/yes/p' hello.txt

输出内容为：

yes 20 30

yes0 200 300

取出文件中行手的行号与冒号

设hello.txt的内容为

1:#!/bin/sh

2:cat hello.txt

3:exit

sed -n -e 's/^[0-9]\{1,\}://g'p hello.txt

输出结果为：

#!/bin/sh

cat hello.txt

exit

==【全文完】==

回【架构】[58同城推荐系统架构设计与实现](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=201152754&idx=1&sn=b1e05153a298351dfdd917f451e0ef2f&scene=21" \l "wechat_redirect" \t "_blank)

回【twitter】[twitter系统架构分析](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=201845390&idx=1&sn=79723619a5af4fdd8911d504c92cf144&scene=21" \l "wechat_redirect" \t "_blank)

回【12306】[12306系统架构优化](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=201940636&idx=1&sn=32251e8f756cf0cbb352687c20ddecda&scene=21" \l "wechat_redirect" \t "_blank)

回【youtube】[YouTube系统架构](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=202758792&idx=1&sn=3a0d58174f6f65ead4d40aaf6b498cb1&scene=21" \l "wechat_redirect" \t "_blank)

回【同城】[58同城mysql实战](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=204498648&idx=1&sn=849f4cafec3eb4b7fccff884bb81e8ca&scene=21" \l "wechat_redirect" \t "_blank)

回【秒杀】[秒杀系统架构优化思路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=204965266&idx=1&sn=aad227dbcb544394ae1717e6af8b77ae&scene=21" \l "wechat_redirect" \t "_blank)

回【演进】[好架构是进化来的，不是设计来的](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400276397&idx=1&sn=ea044079667b82f6cad58bcb743af7bc&scene=21" \l "wechat_redirect" \t "_blank)

## 一分钟了解两阶段提交2PC（运营MM也懂了）

原创 2015-11-16 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400856212&idx=1&sn=b9734da77f2259bdd1fa89700b47eaff&scene=21" \l "#)

上一期分享了“一分钟了解mongoDB”【回复“mongo”阅读】，本期将分享分布式事务的一种实现方式2PC。

**一、概念**

二阶段提交2PC（Two phase Commit）是指，在分布式系统里，为了保证所有节点在进行事务提交时保持一致性的一种算法。

**二、背景**

在分布式系统里，每个节点都可以知晓自己操作的成功或者失败，却无法知道其他节点操作的成功或失败。

当一个事务跨多个节点时，为了保持事务的原子性与一致性，需要引入一个**协调者**（Coordinator）来统一掌控所有**参与者**（Participant）的操作结果，并指示它们是否要把操作结果进行真正的提交（commit）或者回滚（rollback）。

**三、思路**

2PC顾名思义分为两个阶段，其实施思路可概括为：

（1）投票阶段（voting phase）：参与者将操作结果通知协调者；

（2）提交阶段（commit phase）：收到参与者的通知后，协调者再向参与者发出通知，根据反馈情况决定各参与者是否要提交还是回滚；

**四、缺陷**

算法执行过程中，所有节点都处于阻塞状态，所有节点所持有的资源（例如数据库数据，本地文件等）都处于封锁状态。

典型场景为：

（1）某一个参与者发出通知之前，所有参与者以及协调者都处于阻塞状态；

（2）在协调者发出通知之前，所有参与者都处于阻塞状态；

另外，如有协调者或者某个参与者出现了崩溃，为了避免整个算法处于一个完全阻塞状态，往往需要借助超时机制来将算法继续向前推进，故此时算法的效率比较低。

总的来说，**2PC是一种比较保守的算法**。

**五、举例**

甲乙丙丁四人要组织一个会议，需要确定会议时间，不妨设甲是协调者，乙丙丁是参与者。

投票阶段：

（1）甲发邮件给乙丙丁，周二十点开会是否有时间；

（2）甲回复有时间；

（3）乙回复有时间；

（4）丙迟迟不回复，此时对于这个活动，甲乙丙均处于阻塞状态，算法无法继续进行；

（5）丙回复有时间（或者没有时间）；

提交阶段：

（1）协调者甲将收集到的结果反馈给乙丙丁（什么时候反馈，以及反馈结果如何，在此例中取决与丙的时间与决定）；

（2）乙收到；

（3）丙收到；

（4）丁收到；

**六、结论**

2PC效率很低，分布式事务很难做。

===【完】===

## 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

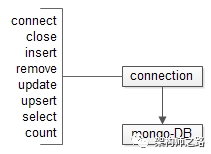
这个连接是什么呢，通过连接怎么调用下游接口？服务/数据库/缓存，**官方会提供不同语言的Driver、Document、DemoCode来教使用方建立连接与调用接口**，以MongoDB的C++官方Driver API为例（伪代码）：

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】==

## 一分钟实现分布式锁

原创 2016-12-12 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959749&idx=1&sn=6727638930905089342662c24f5bba62&chksm=bd2d04198a5a8d0f619f1b9be1ead542f015e6ff87b5aac5a61af94b714459964fedb89e6516&scene=21" \l "#)

**一、缘起**

分布式环境下，多台机器上多个进程对一个数据进行操作，如果不做互斥，就有可能出现“余额扣成负数”，或者“商品超卖”的情况，**如何实现简易分布式锁**，对分布式环境下的临界资源做互斥，是今天将要讨论的话题。

**二、互斥原理**

原理：**多个访问方对同一个资源进行操作，需要进行互斥**，通常是利用一个这些访问方**同时能够访问**到的lock来**实施互斥**的。

例子1：**同一个进程内，多个线程的互斥**，典型的场景是生产者消费者对同一个queue进行操作时的互斥



方案：设定一个所有线程能够访问到的lock实施互斥



步骤：

（1）多个线程同时抢锁

（2）只一个线程抢到，未抢到的阻塞，或下次再来抢

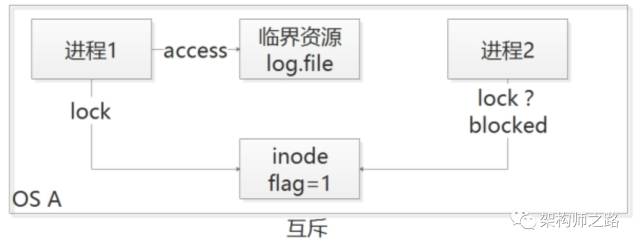
（3）抢到锁的线程操作临界资源

（4）操作完临界资源后释放锁

例子2：**同一个操作系统上，多个进程的互斥**，典型的场景是手机上多个APP对同一个文件进行写入互斥



方案：设定一个所有进程能够访问到的lock实施互斥（例如文件inode，OS帮我们做了）



步骤：

（1）多个进程同时抢锁

（2）只一个进程抢到，未抢到的阻塞，或下次再来抢

（3）抢到锁的进程操作临界资源

（4）操作完临界资源后释放锁

**三、分布式环境下多进程互斥**



分布式环境下，多台机器上多个进程对一个数据进行操作的互斥，例如同一个uid=123要避免同时进行扣款。

根据上面的原理，先找一个多台机器多个进程可以同时访问到的一个lock，例如redis。



步骤：

（1）**多台机器上多个进程对这个锁进行争抢**，例如在缓存上同时进行set key=123操作

（2）**只有一个进程会抢到这个锁**，即只有一个进程对缓存set key=123能够成功，不成功的进程下次再来抢

（3）**抢到锁的进程对余额进行扣减**

（4）**扣减完成之后释放锁**，即对缓存delete key=123

分布式环境下的互斥，搞定。

文章完了，希望大伙对分布式锁原理极其简易实现有个初步的了解，如果有收获，帮忙转发哈，欢迎关注“架构师之路”。

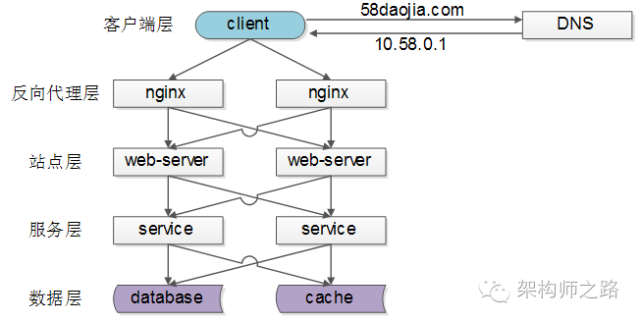
## 一分钟了解负载均衡的一切

原创 2016-09-13 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959585&idx=1&sn=0a9222cbfeb62a662edffafb7f0b43ae&scene=21" \l "#)

**什么是负载均衡**

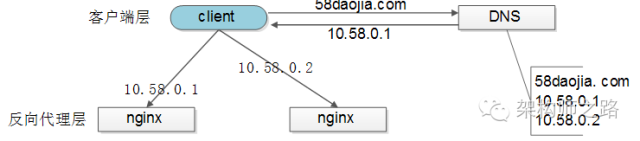
负载均衡（Load Balance）是分布式系统架构设计中必须考虑的因素之一，它通常是指，将请求/数据【均匀】分摊到多个操作单元上执行，负载均衡的关键在于【均匀】。

**常见的负载均衡方案**



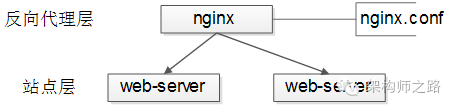
常见互联网分布式架构如上，分为**客户端层、反向代理nginx层、站点层、服务层、数据层**。可以看到，每一个下游都有多个上游调用，只需要做到，每一个上游都均匀访问每一个下游，就能实现“将请求/数据【均匀】分摊到多个操作单元上执行”。

**【客户端层->反向代理层】的负载均衡**



【客户端层】到【反向代理层】的负载均衡，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，保证每个ip的解析概率是相同的。这些ip就是nginx的外网ip，以做到每台nginx的请求分配也是均衡的。

**【反向代理层->站点层】的负载均衡**



【反向代理层】到【站点层】的负载均衡，是通过“nginx”实现的。通过修改nginx.conf，可以实现多种负载均衡策略：

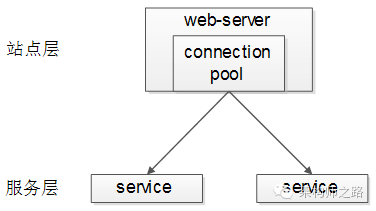
1）**请求轮询**：和DNS轮询类似，请求依次路由到各个web-server

2）**最少连接路由**：哪个web-server的连接少，路由到哪个web-server

3）**ip哈希**：按照访问用户的ip哈希值来路由web-server，只要用户的ip分布是均匀的，请求理论上也是均匀的，ip哈希均衡方法可以做到，同一个用户的请求固定落到同一台web-server上，此策略适合有状态服务，例如session（58沈剑备注：可以这么做，但强烈不建议这么做，站点层无状态是分布式架构设计的基本原则之一，session最好放到数据层存储）

4）…

**【站点层->服务层】的负载均衡**



【站点层】到【服务层】的负载均衡，是通过“服务连接池”实现的。

上游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。

上一篇文章[《RPC-client实现细节》](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959576&idx=1&sn=2be8d3f61effe7118abf920a175da710&scene=21" \l "wechat_redirect" \t "_blank)中有详细的负载均衡、故障转移、超时处理的细节描述，欢迎点击link查阅，此处不再展开。

**【数据层】的负载均衡**

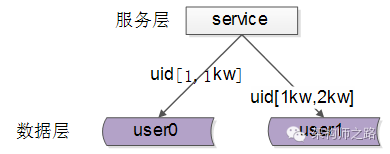
在数据量很大的情况下，由于数据层（db，cache）涉及数据的水平切分，所以数据层的负载均衡更为复杂一些，它分为“数据的均衡”，与“请求的均衡”。

**数据的均衡**是指：水平切分后的每个服务（db，cache），数据量是差不多的。

**请求的均衡**是指：水平切分后的每个服务（db，cache），请求量是差不多的。

业内常见的水平切分方式有这么几种：

一、按照range水平切分



每一个数据服务，存储一定范围的数据，上图为例：

user0服务，存储uid范围1-1kw

user1服务，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

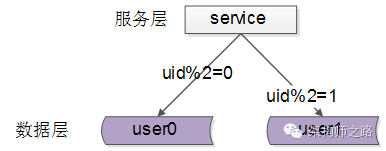
（2）**数据均衡性较好**

（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

二、按照id哈希水平切分



每一个数据服务，存储某个key值hash后的部分数据，上图为例：

user0服务，存储偶数uid数据

user1服务，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

**总结**

负载均衡（Load Balance）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，将请求/数据【均匀】分摊到多个操作单元上执行，负载均衡的关键在于【均匀】。

（1）【客户端层】到【反向代理层】的负载均衡，是通过“DNS轮询”实现的

（2）【反向代理层】到【站点层】的负载均衡，是通过“nginx”实现的

（3）【站点层】到【服务层】的负载均衡，是通过“服务连接池”实现的

（4）【数据层】的负载均衡，要考虑“数据的均衡”与“请求的均衡”两个点，常见的方式有“按照范围水平切分”与“hash水平切分”

谢谢大伙，帮忙转发，欢迎讨论，有问必答。

==【完】==

回【sed】[一分钟sed入门（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403340380&idx=1&sn=9b4ced5a19f60b0cebf6fd582f1d52ae&scene=21" \l "wechat_redirect" \t "_blank)

回【广告】[一分钟读懂互联网广告竞价策略（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403326874&idx=1&sn=ca03344326f35b2971b3ecae2026e8df&scene=21" \l "wechat_redirect" \t "_blank)

回【阵列】[一张“神图”看懂单机/集群/热备/磁盘阵列（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400720056&idx=1&sn=10c796793e27d475603f97b23e137c12&scene=21" \l "wechat_redirect" \t "_blank)

回【awk】[一分钟学awk够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400743254&idx=1&sn=147abc381ccd61e28f52699735c8748e&scene=21" \l "wechat_redirect" \t "_blank)

回【perl】[十分钟学perl够用（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400764946&idx=1&sn=4efab1953696c2e21071e58a74da9b82&scene=21" \l "wechat_redirect" \t "_blank)

回【mongo】[一分钟了解mongodb（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=201250814&idx=1&sn=fd6d85b727e1229e3324850217748d0f&scene=21" \l "wechat_redirect" \t "_blank)

回【2pc】[一分钟了解两阶段提交2PC（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400856212&idx=1&sn=b9734da77f2259bdd1fa89700b47eaff&scene=21" \l "wechat_redirect" \t "_blank)

回【join】[30秒懂SQL中的join（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400889133&idx=1&sn=3ea1c06fc76e92349a822f41f45b77b5&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

## 这才是真正的分布式锁

原创 2016-12-13 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959755&idx=1&sn=e0875b539895578ce966fdad7f3c0365&chksm=bd2d04178a5a8d01a66d30b5ac8958fc5a6108e6e1d90a265361a7d3b30197bcb51f4433f23b&scene=21" \l "#)

昨晚十点下班，回家花了1个小时写了一篇《[一分钟实现分布式锁](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959749&idx=1&sn=6727638930905089342662c24f5bba62&chksm=bd2d04198a5a8d0f619f1b9be1ead542f015e6ff87b5aac5a61af94b714459964fedb89e6516&scene=21" \l "wechat_redirect" \t "_blank)》，引起读者一些反响，有些朋友反馈“setnx算什么方案”，“没有考虑超时”，“为啥不用zookeeper”，有甚者上升到 “质疑58同城的技术水平”，“拉低了架构师的层次”，“适合小学生阅读”。

给58带来负面的影响实在对不起公司，也抱歉耽误部分同学1分钟时间（还好是1分钟系列），不过大部分读者的反馈是正向的，只生气了5分钟。

技术领域，我觉得**了解来龙去脉，了解本质原理，比用什么工具实现更重要**：

（1）进程多线程如何互斥？

（2）一个手机上两个APP访问一个文件如何互斥？

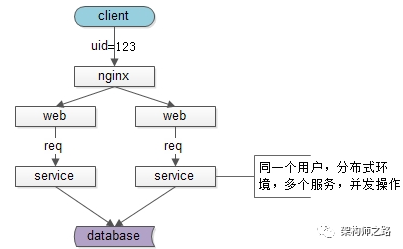
（3）分布式环境下多个服务访问一个资源如何互斥？

归根结底，是利用一个互斥方能够访问的公共资源来实现分布式锁，具体这个公共资源是redis来setnx，还是zookeeper，相反没有这么重要。

言归正传，今天把昨天文章的缘起讲一讲，并通过Google Chubby的论文阅读笔记聊一聊分布式锁。

**一、需求缘起**

58到家APP新上线了导入通讯录好友功能，测试的同学发现，连续点击导入会导入重复数据：



客户端同一个用户同时发出了多个请求，分布式环境下，多台机器上部署的多个service进行了并发操作，故插入了冗余数据。

**解决思路**：同一个用户同时只能有一个导入请求，需要做互斥，最简易的方案，使用setnx快速解决。



（1）同一个用户，多个service进行并发操作，service需要先去抢锁

（2）抢到锁的service，才去数据库操作

具体这个锁用setnx，还是zookeeper都不太重要，**利用一个互斥方能够访问的公共资源来实现分布式锁**，这才是《[一分钟实现分布式锁](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959749&idx=1&sn=6727638930905089342662c24f5bba62&chksm=bd2d04198a5a8d0f619f1b9be1ead542f015e6ff87b5aac5a61af94b714459964fedb89e6516&scene=21" \l "wechat_redirect" \t "_blank)》的重点。

**二、Google Chubby分布式锁阅读笔记**

上一篇文章的评论中，有些朋友提到了zookeeper，会使用不够，借着Google Chubby了解下分布式锁的实现也是有必要的。

早年**Google的四大基础设施**，分别是GFS、MapReduce、BigTable、Chubby，其中**Chubby用于提供分布式的锁服务**。

**1.简介**

Chubby系统提供**粗粒度的分布式锁服务**，Chubby的使用者不需要关注复杂的同步协议，而是通过已经封装好的客户端直接调用Chubby的锁服务，就可以保证数据操作的一致性。

Chubby具有广泛的应用场景，例如：

（1）GFS选主服务器；

（2）BigTable中的表锁；

**2.背景**

Chubby本质上是一个分布式文件系统，存储大量小文件。每个文件就代表一个锁，并且可以保存一些应用层面的小规模数据。用户通过打开、关闭、读取文件来获取共享锁或者独占锁；并通过反向通知机制，向用户发送更新信息。

**3.系统设计**

3.1设计目标

Chubby系统设计的目标基于以下几点：

（1）**粗粒度的锁服务**；

（2）**高可用、高可靠**；

（3）**可直接存储服务信息**，而无需另建服务；

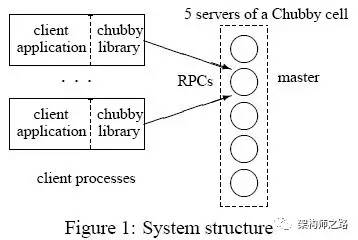
（4）**高扩展性**；

在实现时，使用了以下特性：

（1）缓存机制：客户端缓存，避免频繁访问master；

（2）通知机制：服务器会及时通知客户端服务变化；

3.2整体架构



Chubby架构并不复杂，如上图分为两个重要组件：

（1）Chubby库：客户端通过调用Chubby库，申请锁服务，并获取相关信息，同时通过租约保持与服务器的连接；

（2）Chubby服务器组：一个服务器组一般由五台服务器组成（至少3台），其中一台master，服务维护与客户端的所有通信；其他服务器不断和主服务器通信，获取用户操作。

**4.系统实现**

4.1文件系统

Chubby文件系统类似于简单的unix文件系统，但它不支持文件移动操作与硬连接。文件系统由许多Node组成，每个Node代表一个文件，或者一个目录。文件系统使用Berkeley DB来保存每个Node的数据。文件系统提供的API很少：创建文件系统、文件操作、目录操作等简易操作。

4.2基于ICE的Chubby通信机制

一种基于ICE的RPC异步机制，核心就是异步，部分组件负责发送，部分组件负责接收。

4.3客户端与master的通信

（1）长连接保持连接，连接有效期内，客户端句柄、锁服务、缓存数据均一直有效；

（2）定时双向keep alive；

（3）出错回调是客户端与服务器通信的重点。

下面将说明**正常**、**客户端租约过期**、**主服务器租约过期**、**主服务器出错**等情况。

（1）正常情况

keep alive是周期性发送的一种消息，它有两方面功能：延长租约有效期，携带事件信息告诉客户端更新。正常情况下，租约会由keep alive一直不断延长。

潜在**回调事件**包括：文件内容修改、子节点增删改、master出错等。

（2）客户端租约过期

客户端没有收到master的keep alive，租约随之过期，将会进入一个“危险状态”。由于此时不能确定master是否已经终止，客户端必须主动让cache失效，同时，进入一个寻找新的master的阶段。

这个阶段中，客户端会轮询Chubby Cell中非master的其他服务器节点，当客户端收到一个肯定的答复时，他会向新的master发送keep alive信息，告之自己处于“危险状态”，并和新的master建立session，然后把cache中的handler发送给master刷新。

一段时间后，例如45s，新的session仍然不能建立，客户端立马认为session失效，将其终止。当然这段时间内，不能更改cache信息，以求保证数据的一致性。

（3）master租约过期

master一段时间没有收到客户端的keep alive，则其进入一段等待期，此期间内仍没有响应，则master认为客户端失效。失效后，master会把客户端获得的锁，机器打开的临时文件清理掉，并通知各副本，以保持一致性。

（4）主服务器出错

master出错，需要内部进行重新选举，各副本只响应客户端的读取命令，而忽略其他命令。新上任的master会进行以下几步操作：

a，选择新的编号，不再接受旧master的消息；

b，只处理master位置相关消息，不处理session相关消息；

c，等待处理“危险状态”的客户端keep alive；

d，响应客户端的keep alive，建立新的session，同时拒绝其他session相关操作；同事向客户端返回keep alive，警告客户端master fail-over，客户端必须更新handle和lock；

e，等待客户端的session确认keep alive，或者让session过期；

f，再次响应客户端所有操作；

g，一段时间后，检查是否有临时文件，以及是否存在一些lock没有handle；如果临时文件或者lock没有对应的handle，则清除临时文件，释放lock，当然这些操作都需要保持数据的一致性。

4.4服务器间的一致性操作

这块考虑的问题是：当master收到客户端请求时（主要是写），如何将操作同步，以保证数据的一致性。

（1）节点数目

一般来说，服务器节点数为5，如果临时有节点被拿走，可预期不久的将来就会加进来。

（2）关于复制

服务器接受客户端请求时，master会将请求复制到所有成员，并在消息中添加最新被提交的请求序号。member收到这个请求后，获取master处被提交的请求序号，然后执行这个序列之前的所有请求，并把其记录到内存的日志里。如果请求没有被master接受，就不能执行。

各member会向master发送消息，master收到>=3个以上的消息，才能够进行确认，发送commit给各member，执行请求，并返回客户端。

如果某个member出现暂时的故障，没有收到部分消息也无碍，在收到来自master的新请求后，主动从master处获得已执行的，自己却还没有完成的日志，并进行执行。

最终，所有成员都会获得一致性的数据，并且，在系统正常工作状态中，至少有3个服务器保持一致并且是最新的数据状态。

4.5Chubby系统锁机制

客户端和服务器除了要保存lease对象外，服务器和客户端还需要保存另一张表，用于描述已经加锁的文件及相关信息。由于Chubby系统所使用锁是建议性而非强制性的，这代表着如果有多个锁请求，后达的请求会进入锁等待队列，直到锁被释放。

**5.Chubby使用例子（重点）**

5.1选master

（1）每个server都试图创建/打开同一个文件，并在该文件中记录自己的服务信息，任何时刻都只有一个服务器能够获得该文件的控制权；

（2）首先创建该文件的server成为主，并写入自己的信息；

（3）后续打开该文件的server成为从，并读取主的信息；

5.2进程监控

（1）各个进程都把自己的状态写入指定目录下的临时文件里；

（2）监控进程通过阅读该目录下的文件信息来获得进程状态；

（3）各个进程随时有可能死亡，因此指定目录的数据状态会发生变化；

（4）通过事件机制通知监控进程，读取相关内容，获取最新状态，达到监控目的；

**6.总结**

Google Chubby提供粗粒度锁服务，它的本质是一个松耦合分布式文件系统；开发者不需要关注复杂的同步协议，直接调用库来取得锁服务，并保证了数据的一致性。

最后要说明的是，最终Chubby系统代码共13700多行，其中ice自动生成6400行，手动编写约8000行，这就是Google牛逼的地方：强大的工程能力，快速稳定的实现，然后用来解决各种业务问题。

==【完】==

如果没有时间读完，看看文末“选master”和“进程监控”两个例子，对理解“利用一个互斥方能够访问的公共资源来实现分布式锁”应该有帮助。本篇只是读书笔记，了解更多细节可查阅Google的论文。

## 晋升的为什么不是你

2016-03-26 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=404332602&idx=1&sn=a50ed411c893f6aeadbd66a25d31b94c&scene=21" \l "#)

2011年底的时候，在网上看了一篇文章，《能让你少奋斗10年的工作经验》，其中大部分条目与工作态度相关，有实例，可操作，故有此感慨。

职场纵横，如果下面8条，你也符合部分状态，或许，这就是“晋升的为什么不是你”的答案了。

**一、心灵停留在舒适区是不可原谅的**

状态为：

1）期望舒适，不愿被打扰，不愿被push，不愿被职责，不愿主动关心别人，不愿思考如何提高团队效率；

2）会议上，消极听取领导意见，消极待命，很死的完成交予的任务；

3）不主动接触其他同事，聚会不主动发言，没有做好社交的准备；

把身边的“随意性”赶走，尽早的冲出自己的舒适区域，开始做好和这个社会交流的准备。

**二、“好像、有人会、大概、晚些时候、说不定、应该”不要常挂嘴边**

状态为：

1）“我晚些时候给他”；

2）“应该是明天”；

3）“到时候有人会把东西准备好”；

4）“他好像说是明天”；

这么说的人，**一来想给自己留余地，二来不想给别人造成压迫感**。

这样的答案等于没说，**这样的回答往往暴露其更多的缺点**：

1）之前没有想到这个工作，或者一直在拖延；

2）没有责任心，认为这个不重要；

3）应付上级；

4）不敢说真话；

5）逞能，喜欢答应一些做不到的事；

6）不能独立工作；

**这样的回答，可能让上级更加恼火**：

1）上级的问题没有得到解答，只是起到了提醒你的作用；

2）上级依然要记得提醒你，因为他不知道事情是否真的落实；

3）上级不知道有多少你已经做了的事情中，都是这样落实的（非常致命）；

4）上级往往因为没有得到满意的答案，导致上级自己的计划被推迟或者没有明朗的时间（你把上级的工作计划耽误了）；

一个例子，上级问：你什么时候能把要这个漏洞修好？

乙说：我已经通知他们了，他们大概明天就会来修的。

一天后

上级问：维修公司什么时候回来，你找的是哪家维修公司？

乙说：好像他们说安排不出人来，如果可以的话，今天晚上或者明天下午就能过来。

一天后

上级问：漏洞怎么还没有修好？

乙说：我晚点再问问他们。

上级说：今天下午之前不解决，明天不用来上班了。

**三、不要拖延**

状态为：

1）“决定把事情突击完成”；

2）“工作永远做不完”；

3）“彷徨如何实施的时候，上级看不下去了，上级自己去做了”（危险的信号）；

4）“等看完这一集再做吧”（往往会忘记或者来不及）；

不多说，说做就做是良好的习惯。

如果不知道做，不用想太多太多时间，求助吧。

不要让苦恼和忧虑给你更多的机会蚕食更多的时间。

**四、理论上可行不等于大功告成**

这一点很重要，实施的人开始动手时，往往发现计划可能等于鬼话。

不亲自实践，做计划的人早晚被鄙视。

提高自己办实事的能力，切忌空谈。

一个例子：持续两个小时的会议，理论上看上去很完美，是否考虑到：

开场后调试话筒30分钟；

观众提出尖锐的问题；

探讨的问题展开了；

…

不可能考虑到所有的异常状况，想想按照事先的计划，结果会怎样。

**五、不要让别人等你**

工作中任何时候都不要让别人放下手头的工作来等你。

做协同工作时，关注别人的进度，不要落后。

当大家发现你的工作量完全可以由其他人代替时，团队就可以不需要你了。

**六、细节很重要**

细节是升职的本钱：

1）一个慌忙寻找保险柜钥匙的动作可能让你丧失晋升财务主管的机会；

2）项目谁都能做到60分，但细节决定最终的结果；

3）细节决定成败，就是这么残酷。

**七、不要表现的消极**

1）可能做的事情不是自己的兴趣，但不能懒散，懒得理睬，想办法应付；

2）可能做的事情很机械，但不要表现的闷闷不乐，因为你可能郁闷更久；

3）上级为项目已经够烦恼了，你还想让他看到你的表情？

4）想想你能否在很好的状态下完成以下的工作么：

高速公路收费员：每天对着小窗口递卡片，持续三年；

学校食堂厨师：烧鸡腿，持续一年；

作家：交稿期到了，两个星期没吃早餐了；

门市部销售：坐了一天，每一个人来，和昨天一样；

IT职员：晚上两点下班，第二天还要上班，持续一个月了；

…

想想自己接触这个工作，多长时间才碰到一个难题，然后就开始抱怨。

有没有一个有趣的职业，工作的很开心，还是自己的兴趣。

**八、不要推卸责任**

推卸责任是害怕的条件反射，不要认为别人看不出这一点。

记得我小学里的一件事情。我一次作业没有带来，老师训斥我：你怎么老是作业不带？

我当时说：不是。。。。

当我正要支支吾吾时候，老师说：什么不是？你带来了没有？

我说：没有

老师说：那不就是没有带！什么不是！就是！

很多人面对工作也是这样，上级问责的时候，条件反射的做出推卸动作，然后是一大推无力的辩解和粗糙的借口。

仔细想想，究竟是不是自己的责任，是不是要改进自己处事的方法。

==【完】==

回【ID】[细聊分布式ID生成方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

回【CAS】[巧用CAS解决数据一致性问题](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400556481&idx=1&sn=42bc187b45e4e53aece4b8c1f43e8019&scene=21" \l "wechat_redirect" \t "_blank)

## 你的收入取决于你的努力程度

原创 2016-08-15 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959534&idx=1&sn=1f2f3b92332439cd65a3f1e69391f55f&scene=21" \l "#)

上周和快车司机师傅的一个相遇，一段对话，一点启示。

======

**司机师傅**：你是不是搞IT的？

**我**：你咋知道？

**司机师傅**：一看你背个电脑包，穿着凉鞋短裤，脸上胡子一圈就猜到是写程序的。

**司机师傅**：我之前是做设计的，在广告公司做UI，photoshop用了十多年，闭着眼睛都能将图设计出来，甚是怀念。

**我**：为何不继续做设计了呢？

**司机师傅**：赚的少，又累。14那会最早做滴滴快滴那会，有个朋友说开车一天能赚上千，满多少单月补贴7000，开始还不信，后面试了两天，真的能赚这么多，就辞职和朋友一起开快车了，那个时候，一个月最多能赚4万多。

**我**：那现在呢？

**司机师傅**：后来越补越少了，现在朋友又回去做设计了，我还在开车。

我：那你为啥还继续开车呢？

**司机师傅**：我喜欢开车，开车一个人自由，做设计一帮人在背后指指点点，不痛快。

**司机师傅**：何况，我现在要是每天6点起来，10点收工【我心想，算上中午休息1小时，这一天也得跑15个小时呀】，限行去五环外跑，努点力，一个月税后也能赚2万多，即使现行不跑，也和做设计差不多。

**司机师傅**：其实吧，不只是快车这活，我的那些做设计的朋友，也有赚的多赚的少的，关键看怒不努力。嫌起床早，嫌气温高，嫌抽佣高，哪能赚到钱。不管是做设计，开车，还是其他任何职业，哪有又轻松又赚钱的活，有的话岂不是任何人都去做了，你说是吧？

**我**：确实

**司机师傅**：一定程度上，人的收入取决于ta的努力程度。

======

启示：

（1）**IT男给人的印象：电脑包，凉鞋短裤，胡子一圈 => 需要大家一起改善**

（2）**每一个设计师背后，难道真的都有一群指点江山的神？**

（3）世界上没有又轻松又赚钱的工作

（4）一定程度上，人的收入取决于ta的努力程度

朋友们，听完这个故事，你有啥想法么？

==【完】==

回【ID】[细聊分布式ID生成方法](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403837240&idx=1&sn=ae9f2bf0cc5b0f68f9a2213485313127&scene=21" \l "wechat_redirect" \t "_blank)

回【百度】[百度咋做长文本去重（一分钟系列）](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=403419223&idx=1&sn=7f45d6bf8af2e2e87349570ab93af441&scene=21" \l "wechat_redirect" \t "_blank)

回【CAS】[巧用CAS解决数据一致性问题](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=400556481&idx=1&sn=42bc187b45e4e53aece4b8c1f43e8019&scene=21" \l "wechat_redirect" \t "_blank)

回【晋升】[晋升的为什么不是你](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=201166426&idx=1&sn=a79299ba40b10220f91775170af99b5e&scene=21" \l "wechat_redirect" \t "_blank)

回【粒度】[微服务架构多“微”才合适？](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959530&idx=1&sn=ff84bd74745eee1577e7dfef8ce66bbe&scene=21" \l "wechat_redirect" \t "_blank)

## 一分钟经理人

原创 2016-12-08 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959737&idx=1&sn=5d6f3fd25f6473e77f54269a1b384565&chksm=bd2d04658a5a8d7389fe8cb761dcc494498e5a330fb4b9db1d760d35f7ce8df26fc788227e6d&scene=21" \l "#)

**零、缘起**

近期公司再做管理者培训，偶老大推荐了一本薄薄的《一分钟经理人》，斯宾塞.约翰逊，花了1小时读完有感，沉淀一篇阅读笔记，故有此文。

**一、前言**

常见经理人有两类：

（1）**“铁腕”经理人**：过于关注结果，能让组织获得成功，团队成员却离他而去

**特点**：保证权威性，现实，精明的安排，关注结果

（2）**“好人”经理人**：过于关注人，拿不到好结果，团队成员觉得他是好人

**特点**：民主，支持下属的大部分决定，人情味

有没有“高效”经理人，即能拿到结果，又能建设好极具战斗力的团队，是原书要讨论的内容。

**二、一分钟目标**

大多数组织里，询问经理某个员工在做什么，与询问员工本人在做什么，得到的答案往往不同，对工作目标与工作内容理解不一致，是最常见的问题。

一分钟目标是指，**每个组织的目标极其评价标准不宜超过250个字，以便一分钟之内能够读完这个目标**。清楚目标之后，一切就清楚了，所有人可以根据目标来定期检查工作进度。

目标设定讲究82法则，80%工作成效来自20%的目标实现，关键目标一般3-6个为宜。如果是项目，可以针对项目设定一分钟目标。

一分钟目标避免出现“出其不意”，每个人一开始就知道想要得到什么样的结果。明确的告之团队的目标，以及对员工的要求，别让他们痛苦的去挣扎，别在他们没有达到预期的时候，对他们进行恶意的攻击“Now I get you, you SOB”。

**三、一分钟称赞**

坦诚的对员工的工作给出期望或评价，让人更容易把工作做好。例如：“我希望你能成为团队骨干，同时从工作中得到乐趣”。

好的经理会通过观察员工，**通过发现员工做对了什么，帮助他们充分发挥潜能**。很多经理总在为员工挑错，而高效的管理者会在员工刚进入一个团队时，对于他们做得好的事情，**及时**给与**简短的肯定**：新人很容易知道，在团队中哪些是被鼓励的，具体的示例很容易让人知道肯定是真心的，而不是泛泛而谈“你今年表现很不错”。

只要**在初期**，接手一个新团队，进入一个新职位，负责一个新项目时，让员工保持做正确的事情，保持一个好的节奏，未来的良好工作习惯会成为自然。

**四、一分钟批评**

对业务熟练的**老员工**，一般能够很好的设定目标，保持良好的工作习惯，并实施自我激励。此时，当员工**犯错时，经理一定要及时且明确的指出，具体错在哪里，并告诉员工这件事情给经理的感受**，失望？难过？生气？

这里要注意几点：第一，错误的指出要及时；第二，错误的指出要明确，和表扬一样，绝不能泛泛而谈“你做事很粗心”，不能针对人，而是针对人的行为；第三，一定要小范围正式的沟通；第四，坦诚的一分钟批评。

一分钟的批评不会持续很长时间，但员工绝不会忘记它，一般也不会再犯同样的错误。

**五、总结**

作为管理者，让所有团队成员设定一个明确的**一分钟能够读完的目标**，新员工对于做的好的给与**一分钟及时称赞**，老员工对于做的不好的给与**一分钟及时批评**。

**目标引发行为，结果巩固行为**。团队里的一切都会往好的方向迈进，这，就是**一分钟经理人**。

**六、楼主感触**

不少公司只用很少的时间花在员工的培养上，既然我们都承认，事情是由人完成的，结果是由团队取得的，为什么不对人的培养与发展进行投入呢？**对团队最好的投资，就是把时间花在人的身上**。我始终认为，帮助员工成长和提高是管理者最重要的职责，当然，完成既定目标，协作队友给队友赋能也是非常重要。事在人为，团队在，没有什么是不能成的。

## 架构师到底该不该写代码

原创 2016-12-21 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959776&idx=1&sn=10a8bcb1277fdd6bfb9e24b816697038&chksm=bd2d043c8a5a8d2a9b3e6638c855da30e2910b17a7f6d435118fb1fe09a99fb85950d98b4d63&scene=21" \l "#)

周末InfoQ-StuQ直播，主持和听众提问的简版实录，快消时代，精简到1分钟可以读完（原文有10000字）。

**提问：沈老师是从什么时候开始写文章的？**

我从大学开始有写文章的习惯，最开始主要记录学习上和生活上的一些东西。毕业加入百度之后，在百度空间总结一些学习到的技术的东西，后来百度空间好像转型做交友平台了，于是搭建了自己的博客，在博客上写了一两年。最近当然就是在公众号“架构师之路”上写，梳理和总结自己日常工作中学习到的一些技术，业务上和架构上遇到的一些问题，分享给大家。

**提问：网上有个很有争议的问题“架构师需要写代码吗？”，您对此怎么看？**

我认为架构师应该写代码。

首先，业务是肯定需要深入去了解的，我比较反对一个公司成立一个所谓的架构师部门，拥有公司所有的架构师资源。我的建议是每个业务线团队都需要有架构师。架构师一定要深入了解业务的特点，针对业务的特点去设计系统架构。

我一直有一个观点“任何脱离业务的架构都是耍流氓”。肯定没有一个一成不变的架构方案，适用所有的业务场景。

其次，是要贴近系统，所以得看代码，写代码。即使完全没有时间去写代码的话，至少详细设计的每一个细节架构师都需要清楚，每一个流程、接口参数、数据库设计都要非常清楚。详细设计尽量详细到组内的任何一个工程师拿到详细设计都可以去做实现。CodeReview也非常重要，保证代码至少是有两个人看过，而且它的实现逻辑和详细设计是一致的。

我对架构师的建议是：有时间的话，亲自去写核心代码，如果没有时间的话，要把关详细设计并安排资深工程师去做CodeReview。

**提问：当前互联网技术更新非常快，您认为架构师对此应该持什么态度？**

首先对于新技术，需要去关注，但我的观点是“应用到线上，一定要慎重”。去看、去学、去研究是一个技术人员必须做的，但是学习新技术与把它应用到线上生产环境是两回事。

我负责58到家的一些后端架构，实施一些通用的技术平台，比如说线上的监控、数据的统一收集等，如果技术体系统一，综合成本会非常小。

再拿存储来举例，存储的软件和技术有很多，mysql，sql-server， mongodb等，统一用一个非常重要，一定不能是哪个团队想用什么就用什么。

我的建议是：对新技术我们一定要去学习，但应用到线上一定要慎重。

**提问：大家觉得架构师的知识宽度是很广的，那会不会有什么都懂、什么都不精这样一种现象存在？**

首先什么都懂是绝对不可能的，什么都精也是绝对不可能的，但是架构师也不能哪一块都不精。虽然业务不一样，但是架构设计上肯定会有通用的地方。我原来做过几百万同时在线的即时通讯系统，它肯定有架构领域内通用的东西，比如接入、数据、可用性、扩展性、一致性等，所以这些经验对我后面做推荐系统的设计，支付系统的设计肯定会有帮助。

其实架构师对于知识的宽度和深度都是有要求的，像现在网上有一种说法说架构师的能力是π型人才，除了技术宽度，还要有两条腿：一条是专业能力，还有一条是通用能力，比如表达、沟通、解决问题、管理、创新等。

**提问：有很多立志于成为架构师的人不知道如何开始？沈老师能不能给一些比较具体的建议？**

我认为架构师之路分为三个阶段：

第一个阶段是打基本功的阶段。对应我自己的话就是职业生涯的前三年，语言、数据结构、算法、设计模式、研发工具、调试工具等，基本功没打好，其他的一切都是空谈。

第二个阶段是业务的积累或叫技术深度的积累。对于我来说，则是业务深入，即前五年在即时通讯领域的打拼。业务的深度决定了进入一家公司的时候，你的身价，一个公司要解决某个业务问题，就必须有针对性的招相关的人才，如果你可以解决这个业务领域内的大部分问题，这就是你的核心竞争力。

第三个阶段，π型人才的另外一条腿，即通用素质这一块，就是你的**执行力、责任心、推动能力、沟通表达能力、项目管理能力**，这些会让别人觉得你是靠谱的。在技术能力大家都差不多的情况下，一个事情为什么交给你来做，大家有没有想过？因为公司觉得你是靠谱的，靠谱这个评价很高。

**提问：对一个架构来说，因为没有完美的架构，它一定会有一些缺陷，那好的架构有一个什么样的标准吗？**

架构是为业务服务的，能够满足业务的需求并且对它的扩展性多考虑一步，我觉得这样的架构就是合适的。

我曾经被问到“58同城从05年发展到现在，架构迭代了很多版，如果回到05年重新做架构设计，58的架构会不会是现在的样子”，答案是一定不会跟今天一个样子，一定还是和05年时候一个样子。

**提问：58的技术氛围是怎么建立起来的？**

第一个指导人机制很重要，就是任何一个研发一定会有一个高职阶的人带，有任何技术上的问题一定是有人可以交流和解答的。

第二个我觉得很重要的是技术评审，技术评审是一个很好的契机让大家沟通交流和讨论技术上的问题。

第三个是分享机制，每个团队内部定期组织技术分享，让大家沟通交流。包括我也每周会花时间和团队的同学做一些技术的交流和沟通。

**提问：PHP是世界上最好的语言吗？**

技术的同学在讨论的时候要避免讨论两个问题，一个是哪种语言是世界上最好的语言，第二个要避免讨论的是Vim好还是Emacs好。

**总结**

**（1） 架构师需要写代码吗？**

有时间的话，亲自去写核心代码，如果没有时间的话，要把关详细设计并安排资深工程师去做CodeReview

**（2）对于新技术，持什么样的态度？**

需要去学习，但应用到线上一定要慎重

**（3）对架构师的能力要求？**

π型人才，除了技术宽度，还要有两条腿：一条是专业能力，还有一条是通用能力

**（4）架构师三个阶段？**

打基本功，业务沉淀，通用素质进阶

**（5）好的架构的标准？**

能够满足业务的需求并且对它的扩展性多考虑一步

**（6）技术氛围怎么培养？**

指导人机制，技术评审，技术分享

最后给有志于成为架构师的同学一个建议：多学习、多交流、多沟通。

## 如何做一场B格满满的技术大会演讲

原创 2016-11-18 58沈剑 [架构师之路](http://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651959675&idx=1&sn=98c61e127f4b82ca3ba86ff369dfdb3c&chksm=bd2d04a78a5a8db1f5de8c39a1398b8812ff2b9961642f7b813293ade34f602d42ea127b85aa&scene=21" \l "#)

什么样的演讲和呈现最受听众欢迎，内容干货？逻辑清晰？~~长相帅气~~？

偶尔被邀作为speaker参加一些圈内的技术大会进行演讲。这里我分享下自己的经验，如何做一场B格满满的技术大会演讲，希望给做汇报、总结、述职的技术小伙伴一些小启示。

**【一、了解听众的诉求】**

如同架构设计一样，了解听众诉求永远是第一步的，先看下各类演讲类型听众的诉求：

（1）**给客户进行讲解**：听众想了解产品，打消疑虑

（2）**总结、述职、汇报**：听众想了解工作成果，潜在困难，未来规划

（3）**技术大会**：听众想学习架构知识，借鉴经验解决工作中的问题

例如，top100summit技术峰会，听众主要的目的就是去**学习兄弟公司成功的案例经验**（产品、研发、测试、流程、管理等），那么演讲嘉宾可以在ppt中**针对性的准备这些内容**：

（1）真实案例

（2）碰到的问题，踩到的坑

（3）各种解决方案优缺点，及解决方案的迭代演进

（4）最佳实践

个人建议，尽量**少一些放之四海皆准的原则理论，多一些血肉充实的生动案例**。

**【二、了解到紧张是正常的】**

世界上最令人恐惧的事情，调研结果， “死亡”屈居第二，“当众讲话”高居榜首，应当认识到：

（1）紧张是正常的，紧张说明这个事情你内心重视，紧张能够帮助你发挥的更好

（2）反正我站在台上的**前几分钟是有点害怕的**，但一旦进入主题就好了，讲的内容是我擅长的，我熟悉的，何惧之有？**除非你在讲别人写的ppt**，这样就惧怕露出破绽

（3）根据个人的经验，观众眼中演讲者的表现，会比演讲者自我感觉的表现更好。“漏了一个要点”只有演讲者自己内心知道，听众是不会察觉的，所以“**紧张只要不被听众察觉，就不是紧张**”

（4）根据个人的经验，微笑和热情能让人更加自信，能够一定程度上消除紧张感，有一次大会分享完，举办方跟听众调研哪个老师讲的不错，听众反馈“都不错，有个笑呵呵的老师感觉挺好”，而其实我内心慌的不行

**【三、好的开场是成功的一半】**

不管怎样，被推上台了，上台后第一段开场留给观众的第一印象非常重要，**这么开场相对比较安全**：

（1）我是谁：公司、姓名必须、职位可选（回想起大学第一次班会自我介绍，balabala讲了一堆，最后忘了介绍自己叫什么，囧）

（2）我为什么有资格讲这个topic：这个部分可选，但讲好了可以增强说服力

（3）大概需要多长时间：给观众一个预期

（4）主要内容是什么：不解释

（5）对听众有什么帮助：这个部分非常重要，1-4吹了若干牛逼之后，听众关注的是这个topic和他有什么关系，能够收获什么，能够解决什么问题

举个栗子，top100summit技术峰会我准备这么开场：

大家好，我是来自58到家的架构师沈剑（公司，职位，姓名），我在百度、58同城工作有过X年的架构经验（吹个牛逼，增强信服力），接下来的50分钟里（时间预期），会和大家介绍58到家调用链跟踪系统架构设计与实现细节（主要内容），通过本次分享，希望大家能够了解到调用链跟踪系统能够解决什么问题，架构细节是怎么样的，以及创业型公司如何快速落地和实现调用链跟踪系统（和听众有什么关系，同时介绍内容）

**【四、内容结构】**

开完场，进入最核心的内容部分，内容结构怎么样组织会让人觉得比较有逻辑性而不会显得无头无绪，个人觉得“问题，[坑-方案-优化方案]，总结”这类**“总分总”的结构是比较安全的**，经验如下：

（1）第一部分：问题缘起，为什么需要，是干嘛的，解决什么问题

（2）第二部分，若干个迭代：

         2.1）矛盾冲突点1

         2.2）有什么传统解决方案

         2.3）传统方案的优缺点，及递进方案，我们的最佳实践是什么

（3）第三部分：总结

“架构师之路”公众号的很多文章，主要也遵循这个内容结构。

再举个栗子，top100summit技术峰会我的内容结构准备这么设计：

（1）**总**：调用链跟踪系统解决什么问题：能够快速发现系统中的性能瓶颈，快速找到不合理调用，可视化查看系统之间的依赖关系，系统出问题时迅速定位问题在哪里

（2）**分**：

矛盾冲突1：如何将一个请求在系统中的轨迹串起来，解决方案1， 2，各自优缺点，58到家的实践

矛盾冲突2：如何描述调用的深度与广度，解决方案1，2，各自优缺点，58到家的实践

矛盾冲突3：如何收集数据可视化展现，解决方案1，2，各自优缺点，58到家的实践

矛盾冲突4：创业型公司在人力有限的情况下，如何快速实现，要改哪些地方成本最小，怎么做好扩展性，58到家的实践

（3）**总**：总结58到家的一些实践

**【五、内容呈现】**

内容呈现我做的也不好，ppt内容呈现应该尽量做到：

（1）“清晰、简洁、达意”

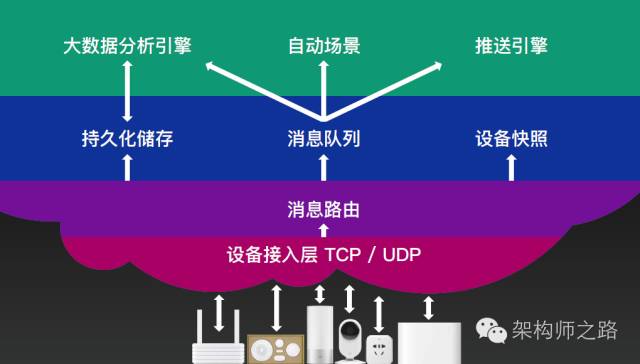
（2）减少大段文字，用架构图、流程图说话，用表格数字对比说话

但自己总不知不觉的，往ppt里添加了很多文字。

这两页是15年某技术大会上**小米**的ppt呈现

【正面case】

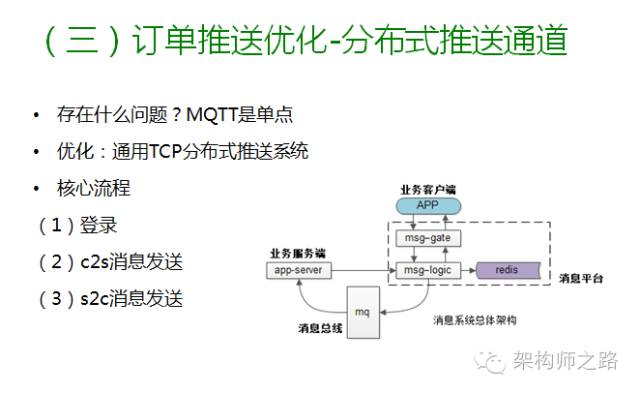




接下来的两页是**我在同一个大会上**的材料

【负面case】





可以体会下差距，什么是专业的，什么是业余的（这还是我自己感觉做的比较好的材料呢=\_=），作为一个技术人，这方面优化的空间的确很大。

**六、如何把控好整体节奏**

内容框架也搞定了，内容呈现也搞定了，有些讲师会有疑问：

（1）“讲的内容过多，时间不够用怎么办”

（2）“很快就把准备的内容讲完了，尴尬，怎么办”

如何把控演讲的整体节奏，我的经验是：

（1）**人在紧张的情况下，语速会加快，演讲往往会比自己预计的时间早结束**

启示一：放慢语速，慢语速会让听众觉得稳重，并且也给了自己思考的时间

启示二：准备比实际需要时间多一点的内容，以防讲太快提前结束尴尬，要知道：讲太快硬要临时想内容撑场面，远比将太慢了快速过掉一些内容容易得多

（2）一定一定一定一定**要规划好，每一页ppt要讲什么内容，哪些是要点，要讲多少分钟**。多人技术大会有一个好处，中途不会有听众打断你，提问环节会统一放在结束，所以提前规划好的节奏，一般不会被打乱

（3）一定一定一定一定**要提前演练**，规划了一页讲5分钟，话匣子一打开，不演练的话实际与计划往往不符合，40分钟演讲自我介绍讲15分钟的场我也见过，讲high了真的收不住，所以一定要实际演练。

大家能看到雷军、罗永浩、柴静在台上举手投足、谈笑风生、镇定自若的样子，每一个眼神、手势、步伐殊不知都经过了几十上百遍演练。

我们只看到别人**牛逼的表面**，却忽略他们**苦逼冰山之下的部分**，人最绝望的状态莫过于，比你牛逼的比你更刻苦。

我去，这鸡汤灌得我自己都要感动了，节奏把握总之一句话，没有临场打断的技术演讲相对比较好控场：**多准备一些内容，放慢一些语速，做好规划，做好演练**。

**【七、如何做好收尾】**

节奏把握住，演讲要进入尾声。一场40-50分钟的演讲，涉及到的架构、流程、方案等技术细节非常多，根据经验，**第二天还能记得10%的听众少之又少**，听众记住的这10%是什么，除了开场灿烂的微笑，大部分就是收尾“反复强调”的总结。

内容不在多，听众有收获就达到目的了，总结上可以反复强调结论，强调实践，如心理学中“近因效应”所述：人对演讲末尾部分的印象最为深刻，记忆也最为深刻。能否在总结处让听众记住你期望ta记住的2-3个点，以达到分享的目的，收尾至关重要。

**【八、如何回答好提问】**

好了，整个内容讲完了，进入提问环节，提问环节也是部分讲师比较头疼的，“万一碰上不会的问题怎么办”，我的经验是：

（1）**首先不要和提问者起冲突**，特别是针对“你讲的我完全不赞同”这类问题，表示“这是自己公司的实践，方案有很多，各有优缺点”之后，可以马上转入“下一个问题”

（2）刻意刁难的技术人一般比较少，更多的情况是，提的问题与话题相关，但自己不100%确定答案，一般可以**将问题技巧性的转化为自己熟悉的问题**，“这位朋友要问的是不是XXX这样一个问题”，而“XXX”问题是自己擅长的，然后顺畅解答

（3）当然，“术”乃技巧，一般实事求是，即使不擅长的问题，说自己不确定，讲讲自己的思路，听众一般也不会苛责

末了，对于提问，还有**两个大招**：

（1）第一个大招，多准备一点内容，把提问时间耗完，是可以跳过提问环节的（举办方肯定不好硬生生打断你，说时间到了，进入提问环节）

（2）第二个大招，是碰到尴尬的问题，可以抛出“这个问题，是个很好的问题，但几句话可能讲不清楚，感兴趣的话，我们线下交流”，**呵呵**。

**【九、总结】**

了解听众 -> 紧张是正常的 -> 做好开头 -> 规划好内容结构 -> 做好内容呈现 -> 把握好演讲节奏 -> 做好收尾 -> 回答好提问。

祝大伙今后演讲**谈客户必成，晋升必过，技术分享爆棚**，希望大伙有收获。

花3秒帮忙转一下哈。