高并发，你真的理解透彻了吗?

原创 骆俊武 武哥漫谈IT 2020-06-27

收录于话题
#架构 1922 #高并发 380 #高性能 198 #技术干货 13

高并发，几乎是每个程序员都想拥有的经验。原因很简单：随着流量变大，会遇到各种各样的技术问题，比如接口响应超时、CPU load升高、GC频繁、死锁、大数据量存储等等，这些问题能推动我们在技术深度上不断精进。

在过往的面试中，如果候选人做过高并发的项目，我通常会让对方谈谈对于高并发的理解，但是能系统性地回答好此问题的人并不多，大概分成这样几类：

1、对数据化的指标没有概念：不清楚选择什么样的指标来衡量高并发系统？分不清并发量和QPS，甚至不知道自己系统的总用户量、活跃用户量，平峰和高峰时的QPS和TPS等关键数据。

2、设计了一些方案，但是细节掌握不透彻：讲不出该方案要关注的技术点和可能带来的副作用。比如读性能有瓶颈会引入缓存，但是忽视了缓存命中率、热点key、数据一致性等问题。

3、理解片面，把高并发设计等同于性能优化：大谈并发编程、多级缓存、异步化、水平扩容，却忽视高可用设计、服务治理和运维保障。

4、掌握大方案，却忽视最基本的东西：能讲清楚垂直分层、水平分区、缓存等大思路，却没意识去分析数据结构是否合理，算法是否高效，没想过从最根本的IO和计算两个维度去做细节优化。

这篇文章，我想结合自己的高并发项目经验，系统性地总结下高并发需要掌握的知识和实践思路，希望对你有所帮助。内容分成以下3个部分：

- 如何理解高并发?
- 高并发系统设计的目标是什么?
- 高并发的实践方案有哪些?

01 如何理解高并发?

高并发意味着大流量，需要运用技术手段抵抗流量的冲击，这些手段好比操作流量，能让流量更平稳地被系统所处理，带给用户更好的体验。

我们常见的高并发场景有：淘宝的双11、春运时的抢票、微博大V的热点新闻等。除了这些典型事情，每秒几十万请求的秒杀系统、每天千万级的订单系统、每天亿级日活的信息流系统等，都可以归为高并发。

很显然，上面谈到的高并发场景，并发量各不相同，**那到底多大并发才算高并发呢?**

1、不能只看数字，要看具体的业务场景。不能说10W QPS的秒杀是高并发，而1W QPS的信息流就不是高并发。信息流场景涉及复杂的推荐模型和各种人工策略，它的业务逻辑可能比秒杀场景复杂10倍不止。因此，不在同一个维度，没有任何比较意义。

2、业务都是从0到1做起来的，并发量和QPS只是参考指标，最重要的是：在业务量逐渐变成原来的10倍、100倍的过程中，你是否用到了高并发的处理方法去演进你的系统，从架构设计、编码实现、甚至产品方案等维度去预防和解决高并发引起的问题？而不是一味的升级硬件、加机器做水平扩展。

此外，各个高并发场景的业务特点完全不同：有读多写少的信息流场景、有读多写多的交易场景，**那是否有通用的技术方案解决不同场景的高并发问题呢?**

我觉得大的思路可以借鉴，别人的方案也可以参考，但是真正落地过程中，细节上还会有无数的坑。另外，由于软硬件环境、技术栈、以及产品逻辑都没法做到完全一致，这些都会导致同样的业务场景，就算用相同的技术方案也会面临不同的问题，这些坑还得一个个趟。

因此，这篇文章我会将重点放在基础知识、通用思路、和我曾经实践过的有效经验上，希望让你对高并发有更深的理解。

02 高并发系统设计的目标是什么?

先搞清楚高并发系统设计的目标，在此基础上再讨论设计方案和实践经验才有意义和针对性。

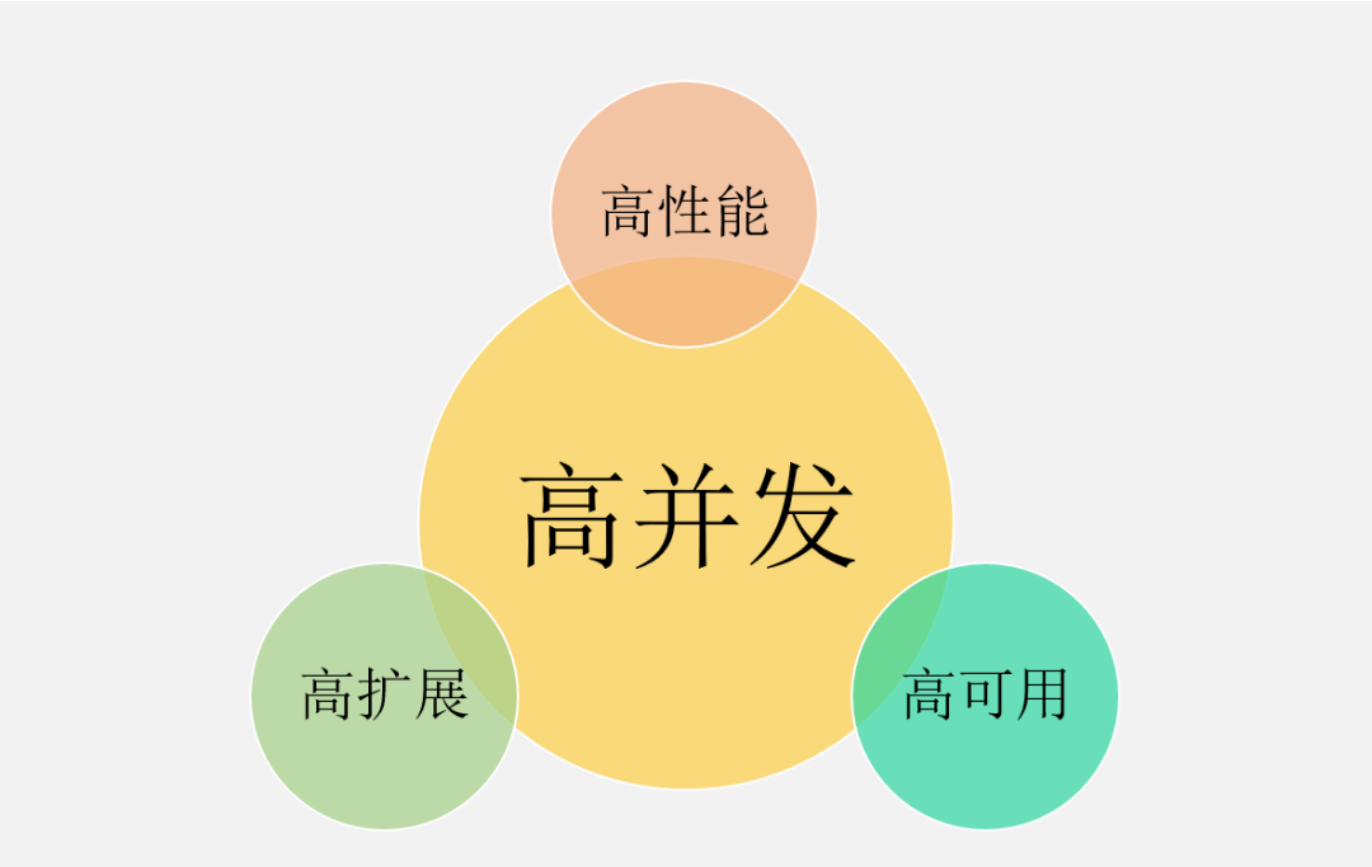
2.1 宏观目标

高并发绝不意味着只追求高性能，这是很多人片面的理解。从宏观角度看，高并发系统设计的目标有三个：高性能、高可用，以及高可扩展。

1、高性能：性能体现了系统的并行处理能力，在有限的硬件投入下，提高性能意味着节省成本。同时，性能也反映了用户体验，响应时间分别是100毫秒和1秒，给用户的感受是完全不同的。

2、高可用：表示系统可以正常服务的时间。一个全年不停机、无故障；另一个隔三差五出线上事故、宕机，用户肯定选择前者。另外，如果系统只能做到90%可用，也会大大拖累业务。

3、高扩展：表示系统的扩展能力，流量高峰时能否在短时间内完成扩容，更平稳地承接峰值流量，比如双11活动、明星离婚等热点事件。



这3个目标是需要通盘考虑的，因为它们互相关联、甚至也会相互影响。

比如说：考虑系统的扩展能力，你会将服务设计成无状态的，这种集群设计保证了高扩展性，其实也间接提升了系统的性能和可用性。

再比如说：为了保证可用性，通常会对服务接口进行超时设置，以防大量线程阻塞在慢请求上造成系统雪崩，那超时时间设置成多少合理呢？一般，我们会参考依赖服务的性能表现进行设置。

2.2 微观目标

再从微观角度来看，高性能、高可用和高扩展又有哪些具体的指标来衡量？为什么会选择这些指标呢?

✳ 性能指标

通过性能指标可以度量目前存在的性能问题，同时作为性能优化的评估依据。一般来说，会采用一段时间内的接口响应时间作为指标。

1、平均响应时间：最常用，但是缺陷很明显，对于慢请求不敏感。比如1万次请求，其中9900次是1ms，100次是100ms，则平均响应时间为1.99ms，虽然平均耗时仅增加了0.99ms，但是1%请求的响应时间已经增加了100倍。

2、TP90、TP99等分位值：将响应时间按照从小到大排序，TP90表示排在第90分位的响应时间，分位值越大，对慢请求越敏感。

3、吞吐量：和响应时间呈反比，比如响应时间是1ms，则吞吐量为每秒1000次。

通常，设定性能目标时会兼顾吞吐量和响应时间，比如这样表述：在每秒1万次请求下，AVG控制在50ms以下，TP99控制在100ms以下。对于高并发系统，AVG和TP分位值必须同时要考虑。

另外，从用户体验角度来看，200毫秒被认为是第一个分界点，用户感觉不到延迟，1秒是第二个分界点，用户能感受到延迟，但是可以接受。

因此，对于一个健康的高并发系统，TP99应该控制在200毫秒以内，TP999或者TP9999应该控制在1秒以内。

✳ 可用性指标

高可用性是指系统具有较高的无故障运行能力，可用性 = 正常运行时间 / 系统总运行时间，一般使用几个9来描述系统的可用性。

可用性	年故障时间	日故障时间
90%（1个9）	36.5天	2.4小时
99%（2个9）	3.65天	14.4分钟
99.9%（3个9）	8小时	1.44分钟
99.99%（4个9）	52分钟	8.6秒
99.999%（5个9）	5分钟	0.86秒

对于高并发系统来说，最基本的要求是：保证3个9或者4个9。原因很简单，如果你只能做到2个9，意味着有1%的故障时间，像一些大公司每年动辄千亿以上的GMV或者收入，1%就是10亿级别的业务影响。

✳ 可扩展性指标

面对突发流量，不可能临时改造架构，最快的方式就是增加机器来线性提高系统的处理能力。

对于业务集群或者基础组件来说，扩展性 = 性能提升比例 / 机器增加比例，理想的扩展能力是：资源增加几倍，性能提升几倍。通常来说，扩展能力要维持在70%以上。

但是从高并发系统的整体架构角度来看，扩展的目标不仅仅是把服务设计成无状态就行了，因为当流量增加10倍，业务服务可以快速扩容10倍，但是数据库可能就成为了新的瓶颈。

像MySQL这种有状态的存储服务通常是扩展的技术难点，如果架构上没提前做好规划（垂直和水平拆分），就会涉及到大量数据的迁移。

因此，高扩展性需要考虑：服务集群、数据库、缓存和消息队列等中间件、负载均衡、带宽、依赖的第三方等，当并发达到某一个量级后，上述每个因素都可能成为扩展的瓶颈点。

03 高并发的实践方案有哪些?

了解了高并发设计的3大目标后，再系统性总结下高并发的设计方案，会从以下两部分展开：先总结下通用的设计方法，然后再围绕高性能、高可用、高扩展分别给出具体的实践方案。

3.1 通用的设计方法

通用的设计方法主要是从「纵向」和「横向」两个维度出发，俗称高并发处理的两板斧：纵向扩展和横向扩展。

✳ 纵向扩展（scale-up）

它的目标是提升单机的处理能力，方案又包括：

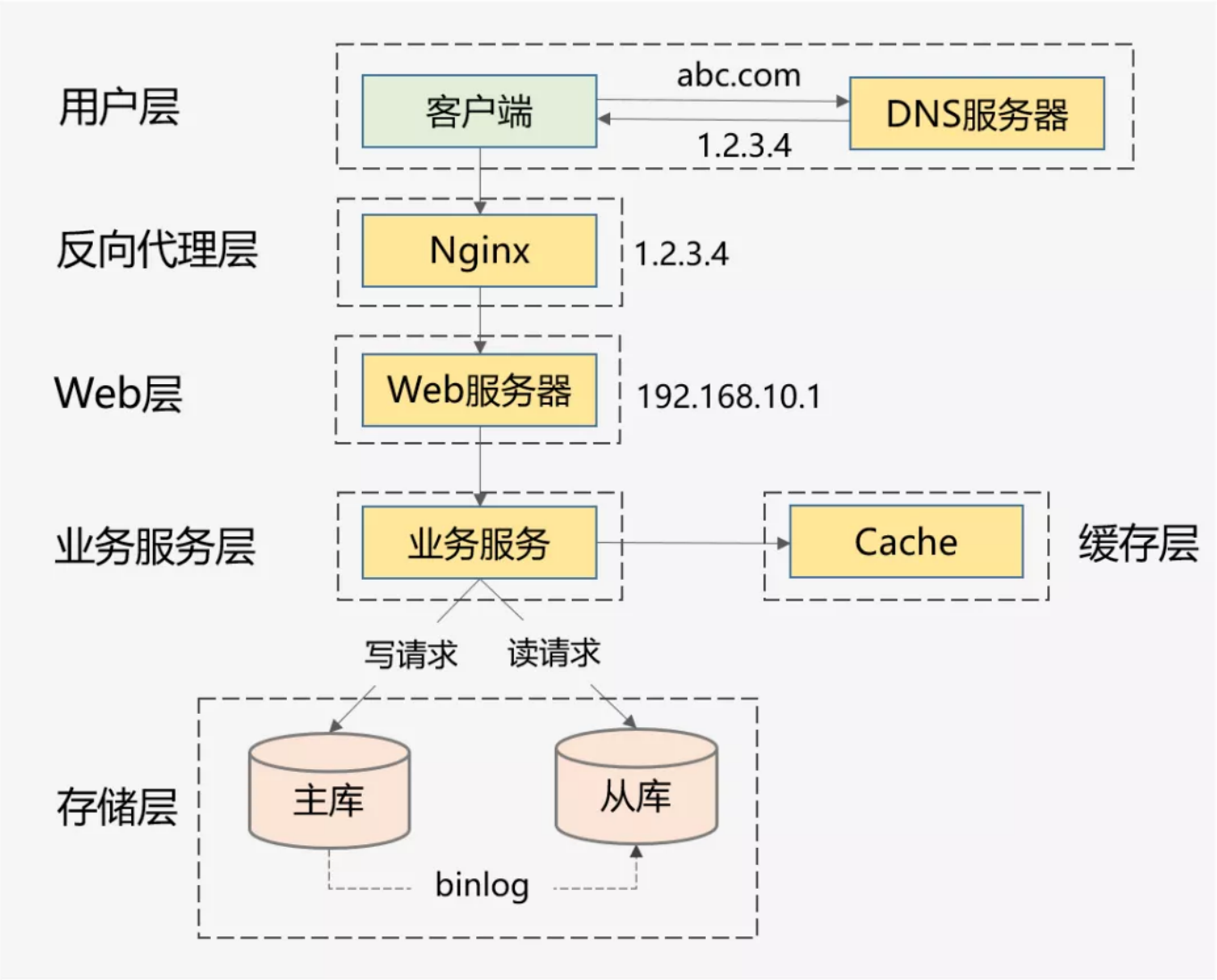
1、提升单机的硬件性能：通过增加内存、CPU核数、存储容量、或者将磁盘升级成SSD等堆硬件的方式来提升。

2、提升单机的软件性能：使用缓存减少IO次数，使用并发或者异步的方式增加吞吐量。

✳ 横向扩展（scale-out）

因为单机性能总会存在极限，所以最终还需要引入横向扩展，通过集群部署以进一步提高并发处理能力，又包括以下2个方向：

1、做好分层架构：这是横向扩展的提前，因为高并发系统往往业务复杂，通过分层处理可以简化复杂问题，更容易做到横向扩展。



上面这种图是互联网最常见的分层架构，当然真实的高并发系统架构会在此基础上进一步完善。比如会做动静分离并引入CDN，反向代理层可以是LVS+Nginx，Web层可以是统一的API网关，业务服务层可进一步按垂直业务做微服务化，存储层可以是各种异构数据库。

2、各层进行水平扩展：无状态水平扩容，有状态做分片路由。业务集群通常能设计成无状态的，而数据库和缓存往往是有状态的，因此需要设计分区键做好存储分片，当然也可以通过主从同步、读写分离的方案提升读性能。

3.2 具体的实践方案

下面再结合我的个人经验，针对高性能、高可用、高扩展3个方面，总结下可落地的实践方案。

✳ 高性能的实践方案

1、集群部署，通过负载均衡减轻单机压力。

2、多级缓存，包括静态数据使用CDN、本地缓存、分布式缓存等，以及对缓存场景中的热点key、缓存穿透、缓存并发、数据一致性等问题的处理。

3、分库分表和索引优化，以及借助搜索引擎解决复杂查询问题。

4、考虑NoSQL数据库的使用，比如HBase、TiDB等，但是团队必须熟悉这些组件，且有较强的运维能力。

5、异步化，将次要流程通过多线程、MQ、甚至延时任务进行异步处理。

6、限流，需要先考虑业务是否允许限流（比如秒杀场景是允许的），包括前端限流、Nginx接入层的限流、服务端的限流。

7、对流量进行削峰填谷，通过MQ承接流量。

8、并发处理，通过多线程将串行逻辑并行化。

9、预计算，比如抢红包场景，可以提前计算好红包金额缓存起来，发红包时直接使用即可。

10、缓存预热，通过异步任务提前预热数据到本地缓存或者分布式缓存中。

11、减少IO次数，比如数据库和缓存的批量读写、RPC的批量接口支持、或者通过冗余数据的方式干掉RPC调用。

12、减少IO时的数据包大小，包括采用轻量级的通信协议、合适的数据结构、去掉接口中的多余字段、减少缓存key的大小、压缩缓存value等。

13、程序逻辑优化，比如将大概率阻断执行流程的判断逻辑前置、For循环的计算逻辑优化，或者采用更高效的算法。

14、各种池化技术的使用和池大小的设置，包括HTTP请求池、线程池（考虑CPU密集型还是IO密集型设置核心参数）、数据库和Redis连接池等。

15、JVM优化，包括新生代和老年代的大小、GC算法的选择等，尽可能减少GC频率和耗时。

16、锁选择，读多写少的场景用乐观锁，或者考虑通过分段锁的方式减少锁冲突。

上述方案无外乎从计算和 IO 两个维度考虑所有可能的优化点，需要有配套的监控系统实时了解当前的性能表现，并支撑你进行性能瓶颈分析，然后再遵循二八原则，抓主要矛盾进行优化。

✳ 高可用的实践方案

1、对等节点的故障转移，Nginx和服务治理框架均支持一个节点失败后访问另一个节点。

2、非对等节点的故障转移，通过心跳检测并实施主备切换（比如redis的哨兵模式或者集群模式、MySQL的主从切换等）。

3、接口层面的超时设置、重试策略和幂等设计。

4、降级处理：保证核心服务，牺牲非核心服务，必要时进行熔断；或者核心链路出问题时，有备选链路。

5、限流处理：对超过系统处理能力的请求直接拒绝或者返回错误码。

6、MQ场景的消息可靠性保证，包括producer端的重试机制、broker侧的持久化、consumer端的ack机制等。

7、灰度发布，能支持按机器维度进行小流量部署，观察系统日志和业务指标，等运行平稳后再推全量。

8、监控报警：全方位的监控体系，包括最基础的CPU、内存、磁盘、网络的监控，以及Web服务器、JVM、数据库、各类中间件的监控和业务指标的监控。

9、灾备演练：类似当前的“混沌工程”，对系统进行一些破坏性手段，观察局部故障是否会引起可用性问题。

高可用的方案主要从冗余、取舍、系统运维3个方向考虑，同时需要有配套的值班机制和故障处理流程，当出现线上问题时，可及时跟进处理。

✳ 高扩展的实践方案

1、合理的分层架构：比如上面谈到的互联网最常见的分层架构，另外还能进一步按照数据访问层、业务逻辑层对微服务做更细粒度的分层（但是需要评估性能，会存在网络多一跳的情况）。

2、存储层的拆分：按照业务维度做垂直拆分、按照数据特征维度进一步做水平拆分（分库分表）。

3、业务层的拆分：最常见的是按照业务维度拆（比如电商场景的商品服务、订单服务等），也可以按照核心接口和非核心接口拆，还可以按照请求源拆（比如To C和To B，APP和H5）。

最后的话

高并发确实是一个复杂且系统性的问题，由于篇幅有限，诸如分布式Trace、全链路压测、柔性事务都是要考虑的技术点。另外，如果业务场景不同，高并发的落地方案也会存在差异，但是总体的设计思路和可借鉴的方案基本类似。

高并发设计同样要秉承架构设计的3个原则：简单、合适和演进。“过早的优化是万恶之源”，不能脱离业务的实际情况，更不要过度设计，合适的方案就是最完美的。

希望这篇文章能带给你关于高并发更全面的认识，如果你也有可借鉴的经验和深入的思考，欢迎评论区留言讨论。

<END>

大家在看：

直播平台的技术架构揭秘

线程池运用不当的一次线上事故

线上FGC问题排查，看这篇就够了！

工程师如何从技术转型做管理?

收录于话题 #技术干货·13个

上一篇
监控系统选型，这篇不可不读！

下一篇
23张图，带你入门推荐系统

文章已于2020/07/22修改

喜欢此内容的人还喜欢

当前疫情下火爆的直播应用，你了解背后的技术架构吗?
武哥漫谈IT

设计模式在业务系统中的应用
阿里技术

高并发服务优化篇：从RPC预热转发看服务端性能调优

Coder的技术之路