朱晔的互联网架构实践心得 S1E10：数据的权衡和折腾【系列完】

【下载本文 PDF 进行阅读】

本文站在数据的维度谈一下在架构设计中的一些方案对数据的权衡以及数据流转过程中的折腾这两个事情。最后进行系列文章的总结和之后系列文章写作计划的一些展望。

数据的权衡

正所谓鱼和熊掌不能兼得，舍了才能得。架构或技术设计方案中针对数据这个事情，有太多体现了权衡思想的地方。

空间和时间

我们来想想有哪些广义上空间换时间（性能）的例子，也就是通过使用更多的存储或内存空间加快了任务的单次执行速度或总体吞吐量：

- 让数据在更快的地方：也就是缓存。速度和价格本来就是矛盾的，我们不可能 10 万买到百公里加速在 4 秒内的高性能跑车。存储虽便宜但是速度慢，内存虽然贵但是速度快，使用级联的缓存存储方案我们可以在这当中做一个平衡。不仅仅是架构设计上我们几乎都会用到缓存，CPU 会有多级缓存，OS 也有页面缓存机制。
- 让数据一次性提交：也就是缓冲。在进行 IO 操作的时候，真正和磁盘和网络交互之前，我们往往都会建立缓冲区。在大多数的时候进行 IO 操作对于 10 字节和 100 字节的数据需要的 IO 时间是一样的，我们可以在缓冲区进行短时间的数据累积后一次性进行操作，这种做法不一定能提高单次执行性能但是可以增加吞吐（对于繁忙的系统，吞吐达到瓶颈后单次的执行会排队，所以反过来也可以认为提高单次性能）。
- 让数据更靠近用户：CDN 就是一个典型应用。让数据离用户更近意味数据不需要经过太多的机房和链路交换就可以到达用户终端，显然可以提高访问性能。其实说白了就是让数据在离用户更近的地方缓存一份，在客户端缓存也算。
- 让数据面向查询存储：相对于面向存储优化。常见的存储数据结构上，我们知道写入性能最好的是追加文件的日志存储，然后是 LSM 树然后是 B+树，读取性能则反过来。为了性能我们通常会在保存数据时候进行一定的排序分类然后按一定的数据结构保存，而不仅仅是把原始信息存下来，这样在查询搜索的时候避免了数据全扫。这些特殊的（甚至有的时候是额外的）数据结构的维护也体现了空间换时间。

- 让数据面向输出优化：之前说的物化视图就是一个例子，在保存数据的时候直接保存我们最终需要查询的数据，这样查询的时候就不需要做各种关联。在微博架构设计的时候我们往往会更极端，为每个人维护一个信息流队列，在发微博的时候直接为所有的粉丝的队列追加数据，这样就避免了首页查询时候非常夸张的 Join 操作。
- 让数据复制多份：复制多份数据意味着我们可以有多个相同的数据源来为读取做服务。之前也提到过，虽然这是为伸缩性考虑，但是一定程度上其实也可以提高性能。
- 让数据计算默默进行：搞两份数据，一份数据是用户现在就在看的，另一份数据是新版本的用户将要看到的数据，通过在后台另一块空间单独处理这份新数据，处理完成后再展现给用户就相当于用户不需要花费时间等待数据的处理加工了。
- 浪费一半的空间倒腾：这里说的是类似于 JVM 的新生代的复制算法。始终有两块大小一致的幸存区，在需要回收的时候可以将一个幸存区和伊甸园中的有用对象一次性搬到另一块区域中。虽然浪费了空间，但是这种每次都给我一张新的纸来画图的方式解决了碎片化的问题。

随着存储和带宽的优化，互联网架构更多考虑空间换时间，在有明显带宽和存储瓶颈的视频图片等资源的传输上，我们会采取压缩的方式用时间来换空间。

一致性和可用性

对于分布式存储，数据复制多份（分片）保存。在出现网络故障的时候，节点之间的数据无法一致，这个时候如果我们追求数据一致性那么就只能等待系统恢复再去使用这个系统，这个时候系统就不可用，当然我们也可以放弃一致性的追求先凑合用系统，这个过程中节点之间的数据无法确保一致。分布式的情况下受到外部客观因素的限制，不可能两者都保证，我们只能根据业务需求来决定放弃哪个。有关 CAP 有很多讨论，对于各种分布式存储也有按照 CP（偏向于一致性）和 AP（偏向于可用性）进行了分类，目前其实大多数的系统都把这个选择权交给了客户端交给了用户，在使用数据的时候我们可以选择是 CP 还是 AP，所以不能简单认定某个存储就是 CP 或 AP 的。

数据查询的专有 DSL（领域专用语言）和通用查询语言

类似 Mongodb、ElasticSearch、HBase 等存储系统都有自己的查询 DSL，关系型数据库大多支持以 SQL 这种通用查询语言进行数据查询。ElasticSearch 现已支持了 SQL 查询，基于 HBase 也有很多组件提供 SQL 查询（比如 Phoenix）。我们知道每一种存储引擎都有其特点，专有 DSL 可以做到让你以引擎最合适的方式来做查询，支持了通用的 SQL 后往往会产生

滥用导致的性能问题，但是可以带来无比的便利性（语言和语言之间的翻译是很痛苦的，而且这个翻译往往是有损的）。在我们自己做服务设计对外 API，设计业务逻辑处理引擎的时候，其实也会遇到这样的设计层次问题。我们是提供专门的 API 来允许外部操作我们的数据呢，还是开发出一套 DSL 允许外部使用这套语言来做复合查询，甚至直接支持类似 SQL 这种通用语言（直通数据库）。比如在做风控规则引擎的时候，我们可以把每一条规则作为写死的代码逻辑来写，也可以开发出一套 DSL 让风控专家可以配置数据字段和规则，甚至可以直接使用 SQL 进行配置，开发报表系统、监控系统也是类似的道理。

数据的折腾

从数据的角度我们来看看在互联网系统中，数据到底经历了什么，体现在数据的存储和流转两方面。就来说说用户的注册这一过程吧：

1. 用户填写了注册表单，产生了用户名、密码这一注册数据
2. Web 网站服务端从 HTTP（S）请求的 Body 中收到了 Key=Value 形式的数据
3. 数据通过 Thrift 二进制协议编码传输给了用户服务（RPC 调用），在这个过程中其实只有 Value 编码了进去参与传输
4. 在用户服务服务端收到数据后根据接口 IDL 和数据拼在一起反序列化让不同的数据字段又有了含义
5. 用户服务把这个信息保存到了关系型数据库 MySQL（以 MySQL 二进制协议在网络传输），数据成为了一份 RedoLog 后以 B 树形式数据结构在磁盘上存储
6. MySQL 主库又把数据以 SQL 语句（或数据行）的形式把数据复制到从库
7. 用户服务随后又把获得的用户 Id 以及用户的基本 Profile 信息以 Key 为字符串 Value 为 Json 文本的形式在 Redis 中进行缓存
8. 用户服务然后把有新用户注册这个事件作为一个消息（Json 序列化）推送到 MQ Broker 上
9. （异步）红包服务监听了这个消息，在收到消息后重新把字节流转换为事件 Json 对象为新用户派发红包
10. （异步）风控服务也监听了消息，从事件中提取出 IP 地址等信息，使用 CEP 复杂事件处理技术对消息进行处理然后触发报警
11. （异步）大数据仓库服务也监听了消息，从数据库查询用户完整的信息并且把数据保存到了 HBase 中以 LSM 树存储

12. 在用户服务完成处理后，Web 网站也就知道了用户注册成功了告知用户注册成功的消息，然后用户尝试登陆，由于主从同步的延迟，数据库从库没有查询到新注册的用户，用户迷茫了，我注册成功了但我是谁呢（这里开个玩笑，这是一个设计错误，一般而言对于自己产生的数据的查询原则是只能在主库查询）

所以我们看到，互联网分布式架构的复杂性在于一份数据可能会以不同的形式做转换不同的通信结构走传输，不用的数据结构做存储，在每一个环节，数据可能都以不同的形式体现，数据本身和数据代表的意义可能头尾分离。如果我们对程序内存做一份 Dump，对各个环节的 TCP 包做几份 Dump，对数据最终落地的存储做几份 Dump 然后在这几份二进制的 Dump 中找到并且看一下我们的数据长啥样，看看是否还能理解数据的意义，这是不是会是一件有趣的事情呢?

就这么简单的一份数据，为什么我们要这么折腾呢？原因在于我们不得不引入某个技术来解决处理某一方面问题的时候又引入了更多的复杂度，然后我们又需要引入更多的技术来处理，循环往复：

- 在单体架构中，我们依靠数据库提供的索引+事务可以解决大部分的性能和一致性问题
- 单体架构在性能、可扩展性、可靠性等方面不能满足需求，我们引入了分布式架构
- 在分布式架构中，我们做了数据复制的分片，不但需要解决因网络、时钟、资源等问题导致的节点的协调，而且需要在空间分布和时间错位两个层面去考虑之前已经解决的处理的很好的事务性问题
- 我们使用不同的数据库作为复合数据源来取长补短，不同的数据库在分布式架构实现上各不相同，我们需要花大量时间来做高可用的调研，同时它们在存储模型上大不相同，我们需要进行深度研究了解存储模型对于数据增量的性能衰减以及故障后的恢复等方面（诸如 Mongodb、和 ElasticSearch 这种数据库越来越有趋势把自己搞成大而全面的数据存储解决方案了，官方永远宣传的都是自己能干啥而不是不能干啥，至于不能干啥只能靠自己去研究和总结）
- 不同的网络中间件都有各自不同的协议和通讯方式，协议在处理前后兼容性方面各不相同，不同的语言对于基础数据类型都有不同的处理方式，在语言和协议的交互转化中也会有一些兼容性问题
- 我们使用各种方式来提高性能，数据会以不同的形态在多处保存，随着时间的推移数据必然会产生不一致性，对于不一致的数据我们如何发现，发现后去容忍还是补偿同步
- 随着组件的增多，我们不希望组件的不稳定造成木桶效应，我们会引入各种弹性的方案来允许组件的暂时不稳定，我们不希望出了问题找不到来源，会引入各种监控手段来做全链路全组件监控

- 数据往往不能完全掌握在自己手里，越来越多的外部三方服务会提供数据源以及垂直领域的解决方案，和外部系统进行交互保留了分布式系统所有的困难之外，还需要考虑安全、隔离等问题

架构的复杂性在于数据，数据的复杂性在于多变的结构、数据的共享同步以及数据的增长。单机单用户的软件演变为 B/S 形式的软件演变为 SAAS 形式的软件，数据从单机到对内分布式到在整个互联网上形成分布式越来越复杂。架构设计中可能一大半的时间在考虑数据的处理存储问题。

系列文章总结

系列文章到这里算是一个结束了，在本系列文章中我们没有过多涉及具体的技术和算法，我们从高层架构的角度阐述了我的一些实用性经验，力求在广度上都有涉及：

- 第一篇文章，我谈了对 All-In-One 架构起步的看法，谈了不同语言的选择和技术团队中业务和架构团队的特性。
- 第二篇文章，我谈了我认为互联网架构中最重要的三要素，微服务+消息队列+定时任务，消息队列和定时任务其实体现的是数据实时流式处理和非实时批次处理的两大流派。
- 第三篇文章，我谈了如何发挥不同类型数据库（关系型数据库 MySQL、缓存型数据库 Redis、文档型数据库 Mongodb、搜索型数据库 ElasticSearch、时间型数据库 InfluxDb）所长结合之前说的三要素做复合型数据源，当然还有更多类型的数据库，比如图数据库等等在需要的时候也可以引入做合适的业务。
- 第四篇文章，我谈了如何以开源的一些项目（ElasticSearch+Logstash+Kibana、Telegraf+InfluxDb+Grafana）快速搭建起简单的监控、日志和数据分析平台以及阐述了对打点和异常两个事情的看法。
- 第五篇文章，我谈了随着项目的发展提炼打造合适的中间件的必要性，以及介绍了一些常见中间件（配置管理、服务管理、全链路监控、数据访问、分布式缓存、任务管理、发布管理等）的需求功能以及设计上的注意点。
- 第六篇文章，我谈了架构升级迁移的步骤和注意点，以及开发人员比较容易忽略的安全方面的问题，我总结了我认为比较重要的安全意识的十个原则（木桶效应、不信任客户端、数据和代码分清楚、用户看不到不等于黑客看不到、最小化接口权限设计和复用的矛盾怒、一开始就要考虑安全、做好防刷防暴破控制、产品逻辑注意一体性、做好异常数据监控报警、对内的数据注意权限控制和审计）。

- 第七篇和第八篇文章，我针对微软分享的三十种云架构设计模式（涉及监控、性能、可扩展、数据管理、设计实现、消息、弹性、安全等方面）给出了自己的看法。
- 第九篇文章，我谈了架构设计评审时候我们针对组件选型、性能、可伸缩性、灵活性、可扩展性、可靠性、安全性、兼容性、弹性处理、事务性、可测试下、可运维性、监控等方面会一起讨论和提出的一些点，以及写好一篇概要的技术文档最主要的五个手段（需求脑图、系统架构图、对外 API 脑图、交互时序图和数据库 ER 图）。
- 本文最后从架构中最重要的数据角度展开讨论了架构中做的妥协权衡以及分布式架构设计的无限复杂性。

我想了一下，有几方面的内容可以形成其它的系列文章和大家一起探讨，尽请期待：

- Spring 框架越来越完善了，庞大的产品体系和插件式的架构让 Java 开发越来越快速和灵活了，在《你不知道的 Spring》系列文章中我们或许可以聊一下 Spring 那些我们不知道的点滴以及如何以 Spring 为核心做一些扩展。
- 对于分布式架构中的数据复制和分片、高可用、一致性处理，每一个产品都有其算法和思路，也有非常多的争论，在《分布式架构细节设计》系列文章中我们或许可以详细针对微服务和分布式架构具体的一些处理方式挖掘一些细节的点——展开讨论。
- JVM 和 JDK 经过这么多年的发展，形成了一些设计模式和技巧，理解这些更多的意义在于我们可以在我们的软件设计和架构中借鉴，在《软件内部设计模式》系列文章中可以聊聊这部分的一些所见。