25 | 设计数据持久层（上）：理论分析

2019-11-06 四火

全栈工程师修炼指南 进入课程 >

讲述：四火

时长 21:13 大小 14.58M

你好，我是四火。

在基于 Web 的全栈技术下，每一层的设计都有共同点，当然，也有各自的特殊之处。你可以回想一下，我们曾经在第一章谈到的客户端和服务端交互以及 Web API 的设计，在第三章谈到的前端的设计，在第二章谈到的服务端 MVC 各层的设计，从前到后。那么，本章余下的内容，我们就来让整个设计层面上的体系变得完整，讲一讲最后面一层的数据持久层怎样设计。

持久层的设计包括持久化框架选择、持久层代码设计，以及存储技术选型等等，考虑到这其中有部分内容我们在第二章谈论 MVC 模型层的时候已经讲到过了，那么在这一讲和下一讲中，我就会先偏重于持久层的数据存储技术本身，再结合实际的设计案例来介绍怎样选择合适的技术来解决那些经典的实际问题。

关系数据库

关系数据库就是以“关系模型”为基础而建立的数据库，这里的关系模型说的是数据可以通过数学上的关系表示和关联起来，也就是说，关系模型最终可以通过二维表格的结构来表达。关系数据库除了带来了明确的 schema 和关系以外，还带来了对事务的支持，也就是对于强一致性的支持。

数据库范式

数据库的表设计，可以说是全栈工程师经常需要面对的问题。而这部分，其实是有“套路”可循的，其中一些常见的规范要求，就被总结为不同的“范式”（Normal Form）。它可以说是数据库表设计的基础，对于数据库表设计很有实际的指导意义。我注意到有很多程序员朋友都不太清楚不同范式的实际含义，那么今天，就请让我通过一个尽可能简单的图书管理系统的例子，来把它讲清楚。

1. 第一范式（1 NF）

第一范式要求每个属性值都是不可再分的。满足 1NF 的关系被称为规范化的关系，1NF 也是关系模式应具备的最起码的条件。比如下面这样的 Books 表：

BOOK_ID	BOOK_NAME	ISBN
1	Life	978-1-56619-909-4, 978-1-56619-909-5

你看，在上面这张表中，有两本书重名了，都叫“Life”，但是国际标准书号 ISBN 是不同的。放在了同一个属性 ISBN 中，并非不可再分，这显然违反了第一范式。那解决这个问题的办法就是拆分：

BOOK_ID	BOOK_NAME	ISBN
1	Life	978-1-56619-909-4
2	Life	978-1-56619-909-5

2. 第二范式（2 NF）

第二范式要求去除局部依赖。也就是说，表中的属性完全依赖于全部主键，而不是部分主键。

BOOK_ID	BOOK_NAME	AUTHOR_ID	AUTHOR_NAME
1	Life	1	James

你看，在上面这张表中，原本的设计是想让 BOOK_ID 和 AUTHOR_ID 组成联合主键，但是，BOOK_NAME 仅仅依赖于部分主键 BOOK_ID，而 AUTHOR_NAME 也仅仅依赖于部分主键 AUTHOR_ID，违背了第二范式 。解决的办法依然是拆分，把这个可以独立被依赖的部分主键拿出去，上面的表可以拆成下面这样两张表：

BOOK_ID	BOOK_NAME	AUTHOR_ID
1	Life	1

AUTHOR_ID	AUTHOR_NAME
1	James

从这个拆分中我们也可以看到，原表被拆成了 N 对 1 关系的两个表，而被不合范式依赖的那个“部分主键”，变成了“1”这头的主键。

3. 第三范式（3 NF）

第三范式要求去除非主属性的传递依赖。即在第二范式的基础上，非主属性必须直接依赖于主键，而不能传递依赖于主键。

BOOK_ID	BOOK_NAME	CATEGORY_ID	CATEGORY_NAME
1	Life	1	Story

你看上面这张表，主键是 BOOK_ID，而 CATEGORY_ID 是非关键字段，并非直接依赖于主键，而是通过这样的传递依赖：

> CATEGORY_NAME → CATEGORY_ID → BOOK_ID

因此，为了消除这个传递依赖，我们还是拆表，让这个传递链中间的非关键字段自立门户：

BOOK_ID	BOOK_NAME	CATEGORY_ID
1	Life	1

CATEGORY_ID	CATEGORY_NAME
1	Story

一般我们在设计中分析到第三范式就打住了，很少有情况会考虑更为严格的范式。例如，BC 范式和第三范式就很像，但是，第三范式只是消除了非主属性对主属性的传递依赖，而 BC 范式更进一步，要求消除主属性对主属性的传递依赖，从而，消除所有属性对主属性的传递依赖。

当然，还有第四、第五范式等等，要求更加严格，解耦更加彻底，但却不太常用了，如果你想进一步了解，可以参阅维基百科的词条。

范式的程度更高，冗余度便更低。但正如同我们在第二章介绍的“拆分”大法一样，每一次范式的升级都意味着一个拆表的过程，一旦过度解耦，拆分出太多零散的表，对于程序员的理解，脑海中数据模型的建立，甚至包括联表操作的 I/O 性能，都是不利的。因此我们需要“权衡”，掌握好这个度，**这一原则，和我们介绍过的分层设计是一致的。**

NoSQL

在上一讲中，我已经介绍了 NoSQL 的概念。在 NoSQL 出现以前，设计大型网站等 Web 应用的时候，全栈工程师的武器库里可供使用的选择，要局限得多。尤其是对一些量大且非结构化的数据，缺乏特别理想的解决方法，工程师有时不得不采用一些非常规的特殊方案，而更多时候我们只能在传统关系数据库的基础上，使用数据库的 Sharding 和 Partition 这样的操作。

虽然那时候在数据规模上面临的挑战远没有现在大，可那个时候 DBA（Database Administrator）在市场上可以说是炙手可热，一旦出了问题，有时甚至还得请专门的数据库厂商的专家，这其中的费用极其高昂，是以分钟计算的。

如今，你可能听说过许多互联网企业去 IOE 的故事（IOE 指的是 IBM 的小型机、Oracle 数据库、EMC 存储设备这三者），其中阿里巴巴的版本听起来还颇为传奇。事实上，这可不只是只有国内的阿里巴巴曾经努力做的事情，还是全球许多大型互联网企业都曾经或正在做的事情，也包括 Amazon。我曾经在 Amazon 的销量预测团队中工作，当时我们团队可以说就在整个亚马逊最大的传统（非云上）关系数据库上工作，里面存放了全部商品库存、销量等有关的信息。

Web 2.0 时代的到来，为互联网应用带来了深远的影响，**用户代替传统媒体，成为了 Web 2.0 时代主要的数据制造者。海量、不定结构、弱关联关系、高可用性和低一致性要求的数据特点，让关系数据库力不从心；而 NoSQL 则具有更好的横向扩展性、海量数据支持、易维护和廉价等等优势，犹如一剂特效药，成为了市场上这个数据难题的大杀器。**

当然，现实中依然存在大量需要强一致性和关系查询的业务场景，因此关系数据库依然是我们倚赖的重要工具。可是，**依然使用关系数据库并不代表依然靠互联网企业自己亲力亲为地做繁重的数据库管理工作**，关系数据库云服务的崛起将传统的数据库管理工作自动化，于是普通的软件工程师也可以完成以往 DBA 才能完成的工作了。从这里也能看出，DBA 也确实是一个技术变更影响技术人才市场需求的典型例子。

综上，这个数据的问题就有了两个层次的解决方案：

出现了更适合业务的非关系数据库服务，也就是 NoSQL；

把关系数据库搬到云上，从而让互联网企业从繁重的数据库管理工作中解脱出来，例如 RDS。

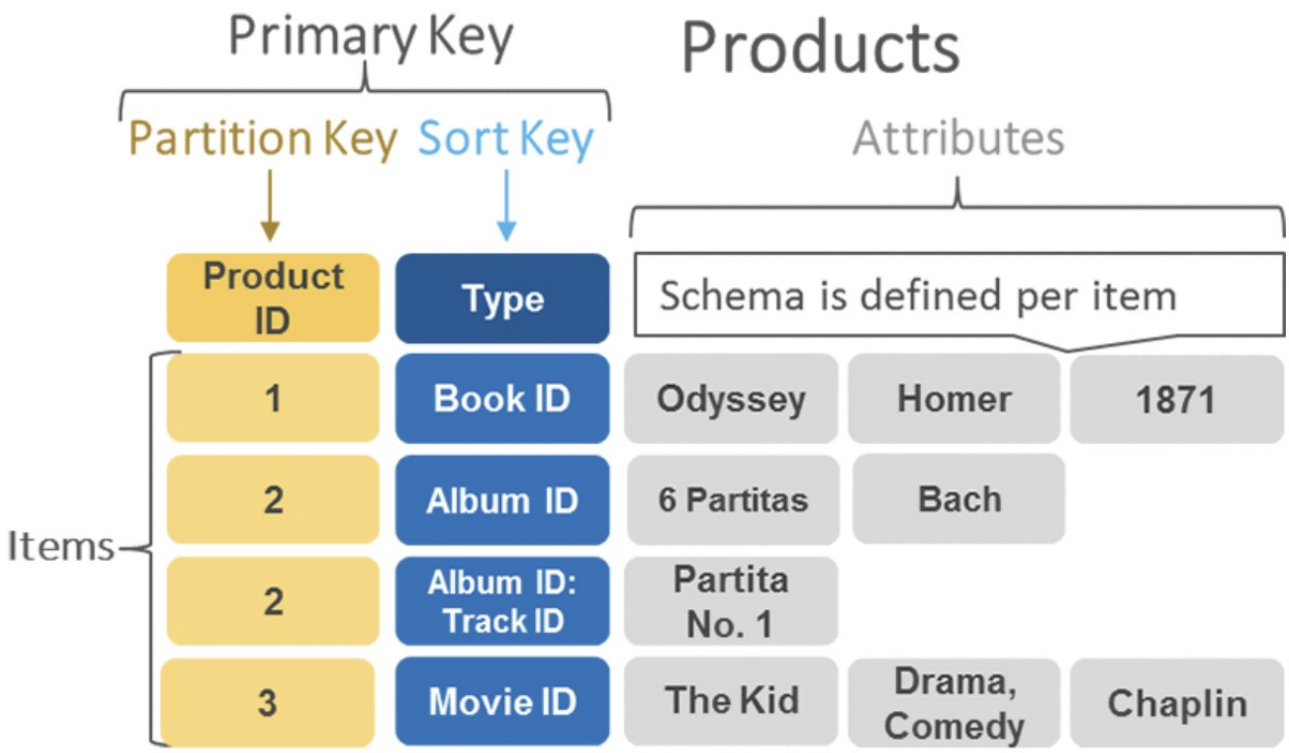
NoSQL 数据库的分类

别看 NoSQL 的数据库那么多，它们大致可以被分为这样几类，每一类也都有自己的优势和劣势。在介绍每一类的时候，我会以我比较熟悉的 AWS 上的实现来具体介绍，当然，很可能你见过的是其它的例子（比如非云上的本地版本，再比如不同的云服务厂商都会提供自己的实现），但在原理层面都是类似的。

1. 键值（Key-value）数据库

这一类 NoSQL 的数据库，采用的是 key-value 这样的访问模型，也就是说，可以根据一个唯一的 key 来获取所需要的值，这个 key 被称为主键，而这个根据 key 来获取 value 的访问的过程是通过 Hash 算法来实现的。本地的 Redis 或者云上的 DynamoDB 都属于这一类。

以 DynamoDB 为例，它的 key 由 Partition Key 和 Sort Key 两级组成，即前者用来找到数据存在哪一个存储单元（Storage Unit），而后者用来找到 value 在存储单元上的具体位置。通常来说，这个二级 Hash 的过程时间开销并不会随着数据量的增大而增大，下图来自官方的 Blog：



DynamoDB 表结构看起来和传统的关系数据库有些像，并且每一行的 schema 可以完全不同，即“列”可以是任意的。每张表的数据支持有限的范围查询，包括主键的范围查询，以及索引列的范围查询。其中，索引的数量是有着明确限制的，一种是全局的（相当于 Partition Key + Sort Key），每张表上限 20 个；一种是本地的（相当于 Partition Key 已经确定，只通过 Sort Key 索引），每张表上限 5 个。

2. 列式（Columnar）数据库

经典的数据库是面向行的，即数据在存储的时候，每一行的数据是放在一起的，这样数据库在读取磁盘上连续数据的时候，实际每次可以一气读取若干行。如果需要完整地查询出特定某些行的数据，行数据库是高效的。且看下面的示例，来自 Redshift 官方文档：

SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

但是列式数据库不一样，它是将每一列的数据放在一起。这样的话，如果我们的处理逻辑是要求取出所有数据中的特定列，那么列数据库就是更好的选择：

SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

Block 1

事实上，对于数据库来说，磁盘的读、写本身，往往还不是最慢的，最慢的是寻址操作。因此，无论是行数据库还是列数据库，如果根据实际需要，我们的实际访问能够从随机访问变成顺序访问，那么就可以极大地提高效率。在大数据处理中经常使用的 HBase 和云上的 Redshift 都属于这一类。

和行数据库相比，列数据库还有一些其它的好处。比如说，对于很多数据来说，某一个特定列都是满足某种特定的格式的，那么列数据库就可以根据这种格式来执行特定的压缩操作。

3. 文档（Document）数据库

文档数据库是前面提到的键值数据库的演化版，值是以某一种特定的文档格式来存储的，比如 JSON、XML 等等。也就是说，文档携带的数据，是已经指定了既定的编码、格式等等信息的。某一些文档数据库针对文档的特点，提供了对于文档内容查询的功能，这是比原始的键值数据库功能上强大的地方。本地的 MongoDB 和 AWS 上的 DocumentDB 都属于这个类型。

4. 对象（Object）数据库

和上面介绍的文档数据库类似，当 value 变成一个可序列化的对象，特别是一个大对象的时候，它就被归类为对象数据库了。AWS 上最常用的 NoSQL 存储，除了前面介绍过的 DynamoDB，就是 S3 了，它相对成本更为低廉，耐久性（durability，数据不丢失的级别）非常高（达到了 11 个 9），并且存储对象可以很大，因此用户经常把它当做一个“文件系统”来存放各种类型的文件，而把 key 设计成操作系统文件路径这样一级一级的形

式，S3 也就被称为“文件系统”。但是，实际上 S3 的“文件”还是和我们所熟悉的操作系统的文件系统有着很大的区别。

当然，还有一些其它的类型，包括图形（Graph）数据库、搜索（Search）数据库等等，我就不一一列举了。

演进趋势

我们在这一章已经从一致性、可用性等方面了解了关系数据库和非关系数据库各自的优势，我想，对这两方面，你应该已经有了自己的认识。现在，我想再补充两个角度，让比较更为全面，分别是扩展性和数据的结构性，我们一起来看看持久层的存储技术的演进趋势。

1. 从 Scale Up 到 Scale Out

先回顾一下“扩展性”（Scalability，也有翻译为“伸缩性”的）这个概念。

按理说，扩展性是包括“纵向（垂直）扩展”和“横向（水平）扩展”的。当然，如今使用这个词的时候，我们往往特指的是“Scale Out”，也就是横向扩展，说白了，就是通过在同一层上增加硬件资源的方式来增加扩展性，从而提高系统的吞吐量或容量。比如说，增加机器。NoSQL 技术往往具备很强的横向扩展能力，至于其中的一个典型原理，你可以回顾一下 [第 23 讲] 的选修课堂“一致性哈希”，你就会明白，为什么 NoSQL 技术可以很方便地在同一层增加硬件资源了。

可是，你知道吗，在很早以前，人们谈“扩展性”的时候，默认指的却是“Scale Up”。它指的是在硬件设施数量不变的基础上，通过增加单个硬件设施的性能和容量来达到同样的“扩展”目的。比如说，把同一台机器的 CPU 换成更好的，把内存升级，把磁盘换成容量更大的，等等。毕竟，在那时候，程序员对于很多问题的思考都还普遍停留在“单机”的层面上。

为什么横向扩展如此重要，以至于成为了“扩展性”的默认指代？

原因很简单，单个硬件设施的性能提升是非常有限的，而且极其昂贵。我记得我刚工作那会儿，去国内某电信运营商开局，众所周知他们“不差钱”，为了提升性能，我们做了很多 Scale Up 的工作，包括把 CPU 升级成了 96 核的。可是你看看现在的互联网公司，哪个会这么玩？因此在现实中，多数情况下，扩容这件事情上，Scale Out 是唯一的选择。

2. 从结构化数据到非结构化数据

我们使用关系数据库的时候，每一行数据都是严格符合表结构的定义，有多少列，每一列的类型是什么，等等，我们把这类数据叫做“结构化”（structured）数据，而这个确定的“结构”，就是 schema。结构化的数据具备最佳的查询、校验和关联能力。

但是当我们使用 DynamoDB 这样的 NoSQL 数据库的时候，我们发现，每一行数据依然可以分成一列一列的，但是有多少列，或者每一列的类型，或者表示的具体含义，却变得不再固定了。这时候我们说，这样的结构依然存在，但是共通的部分明显比结构化数据少多了，于是我们把它们叫做“半结构化”（semi-structured）的数据。

再一般化，就是“非结构化”（non-structured）数据了，上面说的 S3 就是一个很好的例子。即便 S3 上存储的文件是符合某种结构的（比如 JSON），我们也无法利用这个存储服务来完成依据结构而进行的查询等等操作了。

无论是同步还是异步的数据处理，我们总是希望数据的结构性越强越好，因为“结构”本质上意味着“规则”，越强的结构，就越容易使用简单直白的代码逻辑去处理。可是，恰恰相反的是，**我们在现实中遇到的绝大多数的数据，都是非结构化的**，或者说，很难用某一种特定的规则去套。

总结思考

今天我们从不同角度学习了关系数据库和非关系数据库，掌握了一些存储设计的原理和技巧，希望你可以将内容慢慢消化。

下面是今天的提问环节了，我想换个形式。

我们已经学习了几种常见的数据库范式，下面这张图书馆用户表的数据库表的设计是不合理的，你觉得它满足了第几范式呢？并且，你能不能通过学到的拆分方法，分析一下它的问题，把它进一步优化，消除冗余呢？

USER_ID	USER_NAME	BOOK_ID	BOOK_NAME	BORROW_DATE	RETURN_DATE
1	张晓明	2	Life	2019-11-01	2019-12-01
2	王敏	2	Life	2019-12-05	

好，今天的内容就到这里。如果有思考、有问题，欢迎在留言区发言，我们一起讨论。

扩展阅读

文中提到了 Web 2.0 的概念，我推荐你阅读 Web 2.0 的维基百科词条，以及 Web 2.0 网站的九个特点这篇文章，以对它有一个明确的认识。

对于 NoSQL 的一些特定的术语，以及数据库的分类和比较，推荐你阅读这篇什么是 NoSQL?

文中介绍了去 IOE 的事儿，正好 Amazon 最近宣称他们正式完成了从 Oracle 关系数据库迁移离开的工作，感兴趣的话可以看一看。

对于分布式存储感兴趣，并且阅读能力还可以的话，有一些经典论文可以是进一步学习的对象，但请注意它们不是我们这个阶段或当前学习周期内需要学习的内容。比如 Dynamo: Amazon’s Highly Available Key-value Store 这篇，中文译文可以参考这篇，它影响了后来很多分布式系统的设计和发展，我几年前也学习并写了一些自己的理解。

再就是 Google 著名的“三驾马车”了，GFS（中文译文），MapReduce（中文译文） 和 BigTable（中文译文）。我仅仅把它们放在这里，只是供感兴趣且有一定论文阅读能力的程序员朋友参考，而对于本专栏全栈的学习来说，不接触它们是完全没问题的。

上一篇　24 | 尽有所短，寸有所长：CAP和数据存储技术选择

下一篇　26 | 设计数据持久层（下）：案例介绍

精选留言 (2)

写留言

Middleware
2019-11-07

如果设计的话，可能差分成四张表
1：用户表（用户id、用户名）
2：图书表（图片编号、图书名称）
3：借阅表（图书编号、用户编号、借阅时间）
4：归还表（图书编号、用户编号、归还时间）...
展开

作者回复: 👍，如果没有特殊原因，借阅表和归还表可以合并。

1

leslie
2019-11-06

老师的扩展阅读确实不错：强化和补充了不少知识，对于进一步学习和提升以及真正掌握知识的这块还是蛮有用的；希望老师能在这个环节更好的分享一些东西，谢谢。

下面来回答今天的问题：其实问题非常典型-两张表的东西放在了一张表里。另外指出老师在提供表时所疏漏的一点，字段名没有注释，实际工作中字段名没有注释一律不让通过-生产系统想都不要想上；DBA会明确告诉开发，注释写好了再发过来。开发手上其实...
展开

作者回复: 👍，不过关于其中的 2) 拆分得不完整：把 borrow_date、return_date 可以拆到另外一张借书关系表里面去，因为对于同一本书，book_name 这样的信息不应该出现超过一次：

图书信息表：
book_id / book_name

借书关系表：
book_id / borrower_id / borrow_date / return_date