

26 | 设计数据持久层（下）：案例介绍

2019-11-08 四火

全栈工程师修炼指南

[进入课程 >](#)



讲述：四火

时长 22:22 大小 15.37M



你好，我是四火。

本章我们已经学习了不少持久化，特别是有关存储的技术。那在实际业务中，复杂的问题是一个接着一个的，面对这些琳琅满目的具体技术，我们该怎样运用自己所掌握的知识，做出合理的选择呢？今天我们就来接触一些典型的系统，看看对于它们来说，该做出怎样的持久化设计和技术选型。我相信我们实际接触的系统也有相当程度的类比性，可以带来应用的参考意义。

搜索引擎


小到 BBS 网站的帖子搜索，大到互联网数据搜索引擎，搜索引擎可以说是我们日常接触的几大系统之一。可是，搜索数据的存储该怎么设计呢？

有一些反应迅速的程序员朋友，也许会设想这样的存储结构，利用关系数据库，创建这样一个存储文本（文章）的关系数据库表 ARTICLES：

ARTICLE_ID	TITLE	CONTENT
1	设计数据持久层	本章我们已经学习了不少持久化，特别是有关存储的技术.....

那么，假如现在的搜索关键字是“存储”，我们就可以利用字符串匹配的方式来对 CONTENT 列进行匹配查询：

```
1 select * from ARTICLES where CONTENT like '% 存储 %';
```

 复制代码

这很容易就实现了搜索功能。但是，这样的方式有着明显的问题，即使用 % 来进行字符串匹配是非常低效的，因此这样的查询需要遍历整个表（全表扫描）。几篇、几十篇文章的时候，还不是什么问题，但是如果有几十万、几百万的文章，这种方式是完全不可行的。且不说单独的关系数据库表就不能容纳那么大的数据了，就是能够容纳，要扫描一遍，这里的时间代价是难以想象的，就算我们的系统愿意做，用户可都不愿意等啊。

于是，我们就要引入“**倒排索引**”（**Inverted Index**）的技术了。在前面所述的场景下，我们可以把这个概念拆分为两个部分来解释：

“倒排”，指的是存储的结构不再是先定位到文章，再去文章的内容中找寻关键字了；而是反过来，先定位到关键字，再去看关键字属于哪些文章。

“索引”，指的是关键字，是被索引起来的，因此查询的速度会比较快。

好，那上面的 ARTICLES 表依然存在，但现在需要添加一个关键字表 KEYWORDS，并且，KEYWORD 列需要添加索引，因此这条关键字的记录可以被迅速找到：

KEYWORD_ID	KEYWORD
1	存储

当然，我们还需要一个关联关系表把 KEYWORDS 表和 ARTICLES 表结合起来，KEYWORD_ID 和 ARTICLE_ID 作为联合主键：

KEYWORD_ID	ARTICLE_ID
1	1
1	2

你看，这其实是一个多对多的关系，即同一个关键字可以出现在多篇文章中，而一篇文章可以包含多个不同的关键字。这样，我们可以先根据被索引了的关键字，从 KEYWORDS 表中找到相应的 KEYWORD_ID，进而根据它在上面的关联关系表找到 ARTICLE_ID，再根据它去 ARTICLES 表中找到对应的文章。

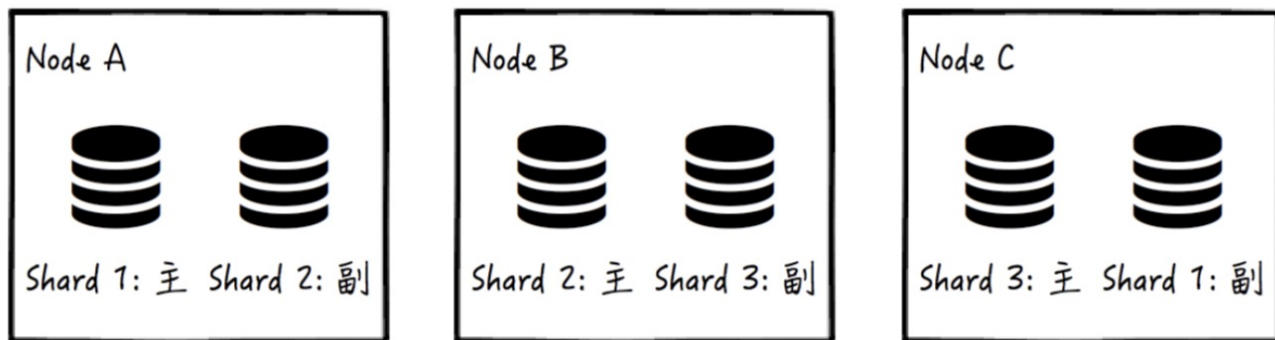
这看起来是三次查找，但是因为每次都走索引，就免去了全表扫描，在数据量较小的时候速度并不慢，并且，在使用 SQL 实现的时候，这个过程完全可以放到一个 SQL 语句中。在数据量较小的时候，上面的方法已经足够好用了。

但是，这个方法只解决了全表扫描和字符串 % 匹配查询造成的性能问题，并且，在数据量较大时，并没有解决数据量本身在单机模式下造成的性能问题。

于是，我们可以考虑搭建和使用 [Elasticsearch](#)，或者干脆使用云上的版本。

Elasticsearch 将关键字使用哈希算法分散到多个不同的被称为 “Shard” 的虚拟节点，并且把它们部署到不同的机器节点上，且每一个 shard 具备指定数量的冗余副本

(Replica)，这些副本要求被放置到不同的物理机器节点上。通过这样的方式，我们就可以保证每台机器都只管理稳定且可控的数据量，并且保证了搜索服务数据的可用性。



对于每一个关键字，都可以配置指向文章和文章中位置的映射。比如有这样两篇文章：

文章 1 的正文是：今天介绍存储技术。

文章 2 的正文是：存储技术有多种分类。

那么，就有如下映射关系（下表仅用于表示在 Shard 中的数据映射，并非关系数据库表）：

TERM	SHARD	DOCUMENT
存储	A	1:4, 2:0
今天	B	1:0
...

你看，DOCUMENT 这一部分，每一行都可以存放若干个“文章 id：文中关键字的位置”的组合。

地理信息系统

有这样一款订餐软件，上面有这样一个功能，在地图上可以列出距离当前用户最近的签约饭馆，并且随着用户缩放地图，还可以控制这个距离的大小。每个饭馆的位置可以简单考虑为经度和纬度组合的坐标（下图来自 Google 地图，仅示意用）。



简言之，这个功能就是“显示一定范围内的目标集合”，可它该怎样实现呢？

在考虑这个功能以前，我们可以类比地想一想，它其实是一个相当常见且通用的功能，常常应用于订餐、导航软件、旅游网站等等这类 LBS（Location-Based Service，基于位置的服务）应用中，因此这个问题是具有一定典型意义的。


这个背后的数据结构以及存储又是怎样的呢？我们顺着这个“经纬度”的思路往下想，那么，如果就把这样的地理信息，放到一张 LOCATIONS 表上，就会是这样：

LONGITUDE	LATITUDE	RESTAURANT_ID
39.912705	116.456789	1
39.456789	116.654321	2

当然了，还有一张 RESTAURANTS 表：

RESTAURANT_ID	RESTAURANT_NAME
1	东门烤鸭馆
2	西门烤鸡店

于是，要查出范围内的饭馆，我们就可以写这样的 SQL：

 复制代码

```
1 select * from LOCATIONS l, RESTAURANTS r where
2   l.RESTAURANT_ID = r.RESTAURANT_ID and
3   l.LONGITUDE >= 经度下界 and
4   l.LONGITUDE <= 经度上界 and
5   l.LATITUDE >= 纬度下界 and
6   l.LATITUDE <= 纬度上界 ;
```

其中，这个经度、纬度的上下界，是根据用户所在位置，以及地图缩放程度折算出来的。显然，这需要一个全表扫描，加一个笛卡尔积，复杂度偏高，能否优化一下它呢？在往下阅读前，你可以先想一想。

思路 1：给单一维度加索引

嗯，如果经纬度可以分开处理，是不是就可以搞定？比方说，只考虑经度的话，给经度一列建立索引，所有饭馆按照从小到大的顺序排好。这样的话，当给定范围的时候，我们就可以快速找到经度范围内所有满足经度条件的饭馆。从时间复杂度的角度来考虑，在不做额外优化的情况下，以在有序经度列上的二分查找为例，这个复杂度是 $\log(n)$ 。

当再考虑纬度的时候，假如有 m 家满足经度条件的饭馆，接下去我们就只能挨个去检查这 m 家饭馆，找出它们中满足纬度条件的了，也就是说，总的时间复杂度是 $m \cdot \log(n)$ 。这种方法比较简单，在数据量不太大的情况下也没有太大问题，因此这已经是很好的方法了。但是，在某些场景下这个 m 还有可能比较大，那么，有进一步优化的办法吗？

思路 2：GeoHash

其实，经度和纬度的大致思路可以，但是在框选饭馆的时候，不能把经度和纬度分别框选，而应该结合起来框选，并且把复杂度依然控制在 $\log(n)$ 的级别。

其中一个办法就是 [GeoHash](#)，它的大致思路就是降维。即把一个经度和纬度的二维坐标用一个一维的数来表示。具体实现上，有一种常见的办法就是把经度和纬度用一个长位数的数来表示，比如：

```
1 经度: 101010.....
2 纬度: 100110.....
```

[复制代码](#)

接着把二者从左到右挨个位拼接，黑色字符来自经度，蓝色字符来自纬度：

110010011100.....

在这种方式下，从结果的左边最高位开始，取任意长度截断所得到的前缀，可以用来匹配距离目标位置一定距离范围的所有饭馆。当用户选取的地图范围越大，前缀长度就越长，这个匹配精度也就越高，匹配到的饭馆数量也就越少。通过这种方式，区域不断用前缀的方式来细分，相当于给每个子区域一个标记号码。

那么，我们数据库表中的经度和纬度就可以合并为一列，再令这一列为主键，或者做索引，就能够进行单列的范围查询了。

GEO_HASH	RESTAURANT_ID
szh51cgs3g6r	1
szh51cgs3h7g	2

最后，我们已经走到这一步了，接下来该怎么把这个表落实到数据库中呢？这就有多种方式可供选择了。比如我们可以使用关系数据库（例如 MySQL），也可以使用 NoSQL 中的键值数据库（例如 Redis 等）。这方面可以根据其它业务需求，以及实际开发的限制来选择，具体选择的策略，请继续阅读下文。

SQL or NoSQL?

我们在实际的存储系统选择时，经常会涉及到 SQL 数据库和 NoSQL 数据库的选择，也就是关系数据库和非关系数据库的选择。举个例子，如果是电子商务网站（这可能是我们平时听到的最多的例子之一了），应该选择 SQL 还是 NoSQL？

两个前提角度

设计和选型方面，有很多问题都不是黑白分明的，而是要拆分开来一块一块分析。我听到过很多“一刀切”的答案，比如有的人说用 MySQL，有的说用 Redis，我认为这样的结论都是不妥的。那么怎样来选择呢？下面我就来介绍一些 SQL 和 NoSQL 选择的原则。但是在讲原则以前，我觉得需要从两个“前提角度”去厘清我们的问题。

以电商网站的设计为例，这两个角度就是这样的。

1. 数据分类

电子商务网站，这个概念所意味着的数据类型太多了。简单举几个例子：

商品元数据，即商品的描述、厂家等等信息；

媒体数据，比如图片和视频；

库存数据，包括在某个地点的库房某商品还有多少件库存；

交易信息，比如订单、支付、余额管理；

用户信息，涉及的功能包括登陆、注册和用户设置。

因此，在讨论什么存储适合数据和访问的时候，我们最好明确，到底具体是哪一种类型的数据。毕竟，看起来上面的业务场景将有着巨大差别。

2. 数据规模

电子商务网站有大有小，可别一想到电商网站，脑海里就是淘宝和京东，商品可以上千万，甚至上亿。但其实，我们大多数接触的系统，都不会有那么大的规模，电商网站完全可以小到一个提供在线购物业务的私人体育用品专营店，商品数量可以只有几十到几百。

只有把问题做如上的展开并明确以后，我们再去思考和讨论数据结构、一致性、可用性等等这些我们“熟悉”的方面，才准确。因此，从上面的例子来看，**我们在选择技术的时候，很可能要针对每一类数据选择“一组”技术，而不是笼统地选择“一项”技术了。**

选择的思路

那么下一步，我们该怎样来选择 SQL 或是 NoSQL 数据库呢？这部分可以说，不同的人有着颇为不同的看法。下面我想根据我的认识，谈谈一个大致的选择思路，请注意这只是一个粗略的基于经验的分类，具体的技术选择还要具体问题具体分析和细化。

1. 对于中小型系统，在数据量不大且没有特殊的吞吐量、可用性等要求的情况下，或者在多种关系和非关系数据库都满足业务要求的情况下，优先考虑关系数据库。

关系数据库提供了成熟且强大的功能，包括强 schema 定义、关系查询、事务支持等等。关系数据库能够带来较强的扩展能力，未来在业务发展的时候，通过增加索引、增加表、增加列、增加关系查询，就可以迅速解决问题。

在从内存模型到实际存储数据的 ORM 转换的时候，有非常成熟的且支持程度各异的框架，有的把 ORM 完全自动化，让程序员可以关注在核心业务模型上面；有的则是把 ORM 定义让出来，提供足够的灵活性（这部分可以参见 [🔗\[第 12 讲\]](#) 的加餐部分）。

值得注意的是，**不要觉得 NoSQL 是大数据量的一个必然选择**。事实上，即便数据量增大，关系数据库有时也依然是一个选择。当然需要明确的是，通常单表在数据量增大时，会产生性能方面的问题，但是可以使用 Sharding 和 Partitioning 技术来缓和（扩展阅读中有这方面技术的介绍）；而数据可用性的问题，也可以使用集群加冗余技术来解决，当然，有得必有失，这种情况下，通常会牺牲一定程度的一致性。

那么，这个数据量多大算大到关系数据库无法承担了呢？我可以给你一个事实，即微博和 Twitter 都是使用 MySQL 作为主要推文存储的（你可以参看扩展阅读中的文章），因此你可以看到在实际应用中，关系数据库对于特大数据量的支持也是有成功实践的。

2. 是否具备明确的 schema 定义，是否需要支持关系查询和事务？如果有一项回答“是”，优先考虑关系数据库。

关系数据库，首当其冲的特点就是“关系”。因此，可能会有朋友说，不对，电商网站的“商品”其实 schema 是不确定的啊——例如，服装一类商品，都有“尺寸”信息；而电器类呢，都有“功率”信息，这样特定类型的属性决定了商品很难被抽象成某一个统一的表啊。

没错，但是为什么要做到如此牵强的“统一”？通用的商品属性，例如厂家、商品唯一编号当然可以放到“统一”的商品表里面，但余下的信息还是可以根据商品类型放到各自类型的特定表里面，这就好像基类和派生类一样，抽象和统一只能做到某一个层次，层次太高反而不利于理解和维护。

对于一些需要事务的需求，例如订购，往往需要关系数据库的支持。当然，这只是多数情况，NoSQL 也有例外，即允许选择 CAP 中的 CP，具备强一致性且支持事务机制的，例如 DynamoDB。

而有一些系统和数据，则变化很大。比如用户数据，在多数情况下，schema 往往是比较明确的，而且数量上也没有订单等数据一般有特别大的伸缩性要求，因此往往也放到关系数据库里面；但是，在另外一些系统中，用户信息的组成不确定，或者说，schema 不确定，用户信息会放到 JSON 等松散结构的文本中，这种情况下文档数据库也是一个常见的选择；但是在搜索等某些相关功能的实现上，可能又会使用搜索引擎等不同于上面任一者的其它方式。

3. 如果符合结构不定（包括半结构化和无结构化）、高伸缩性、最终一致性、高性能（高吞吐量、高可用性、低时延等）的特点和要求，可以考虑非关系数据库。

简单来说，这时的具体技术选择，可以按照这样两个步骤来落地：

如果你还记得 [🔗\[第 24 讲\]](#) 中介绍的那个 NoSQL 三角，根据一致性、可用性的要求，我们可以选择这个三角中的某一条边；

而在 [🔗\[第 25 讲\]](#) 中则介绍了具体的 NoSQL 技术的分类和适用的问题类型，可以依其做进一步选择，比如根据数据结构化的程度，在那条边上，来进一步选择某一类 NoSQL 的存储技术。

比如说，这个电商问题中提到的媒体数据，即图片和视频，通常来说可用性更为重要，而这一类的大文件，没有内部的 schema，可以考虑使用擅长存放无结构大对象的文档型数据库。当然，也可以直接存放为文件，利用 CDN 的特性同步到离用户更近的节点去，特别对于视频来说，要提供好的用户体验，砸钱到 CDN 服务几乎是必选。

再比如，商品页在很大程度上都是可以缓存的，而缓存基本上都是为了保证可用性而会牺牲一定的一致性。除了上面介绍的 CDN 实现了对媒体内容的缓存以外，商品页本身，或是大

部分区域的信息，都是可以利用 Memcached 等缓存服务缓存起来的。

总结思考

到这一章末尾，我们对于数据持久层的介绍就完结了。不妨来回顾一下，设计持久层，都有哪些需要考虑的方面呢？

首先，是代码层面的设计：

提供数据服务的设计，即 MVC 中模型层的设计，你可以阅读 [🔗\[第 08 讲\]](#) 来复习回顾；

对于模型到关系数据库的映射（ORM）和技术选择，在 [🔗\[第 12 讲\]](#) 的加餐部分有所介绍；

其次，是系统层面的设计：

持久层内部或者持久层之上的缓存技术，请参阅 [🔗\[第 21 讲\]](#) 和 [🔗\[第 22 讲\]](#)；

对于持久化的核心关注点之一 —— 一致性，包括存储系统扩容的基础技术一致性哈希，请参阅 [🔗\[第 23 讲\]](#)；

关于分布式数据存储涉及到的 CAP 理论和应用，以及相关的 ACID、BASE 原则，请参阅 [🔗\[第 24 讲\]](#)；

持久层存储技术的选择，你可以阅读 [🔗\[第 25 讲\]](#) 来回顾，更进一步地，今天这一讲提供了一些具体问题实例和技术选择的思路。

希望通过这些内容的学习，你的持久层部分的知识，可以形成体系，而非零零散散的一个个孤岛。

最后，在这里我留一下今天的作业，这是一个开放性的思考题：

假如说，你要实现一个简化了的微信聊天功能，用户可以一对一聊天，也可以加好友，那么，你该怎么选择技术，并怎样设计持久层呢？

好，今天的内容就到这里，希望你在阅读后能有所收获。关于本章的内容，如果有想法，欢迎和我讨论。

扩展阅读

文中介绍了倒排索引，感兴趣的话你可以进一步阅读 [搜索引擎是如何设计倒排索引的？](#)

文中介绍了 GeoHash，在 [geohash.org](#) 的网站上，可以通过给出经纬度坐标，在地图上找到这个实际位置。感兴趣的话，还可以进一步了解：本质上，类似这种二维到一维的降维方式，都属于 [空间填充曲线](#)，比如说最有名的 [希尔伯特曲线](#)。

Sharding 和 Partitioning 是在数据库中常见的“拆分”方式，文中也提到了，但是这两个概念经常被混用，具体含义你可以参考维基百科 [Shard](#) 和 [Partition](#)。

有一篇介绍 Twitter 怎样应用推、拉模式，处理和存储消息，应对高访问量的 [文章](#)，相应的，微博的技术专家也写了一篇文章来介绍微博的 [处理方式](#)，你可以比较阅读。



全栈工程师修炼指南

从全栈入门到技能实战

熊燚

Oracle 首席软件工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有[现金](#)奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 25 | 设计数据持久层（上）：理论分析

下一篇 27 | 特别放送：聊一聊代码审查

精选留言 (2)

写留言



靠人品去赢

2019-11-08

其实类似微信的通讯工具，他会在你本地有个类似NoSQL的东西，搜个关键字什么的都支持，但是你发现你要找更久的，他会弹出一个框，你可以设置日期其实这个最后要走的数据库去查询。毕竟大家都把NoSQL作为自己的中间件，提高响应缓冲服务器压力之类的，到最后数据还是要乖乖的存在MySQL这些关系型数据库。

展开 ∨



leslie

2019-11-08

关于老师今天的题目给出两种答案：

一种是文档型数据库mongodb。虽然是NOSQL阵营，但是其支持类SQL且用的是JSON，而现在mysql5.7开始的版本用sql，但是支持JSON；主要是文档型数据库适合这种场景；另外一种方式就是redis+MySQL：其原因不言而喻了，太多类似方式了

展开 ∨

