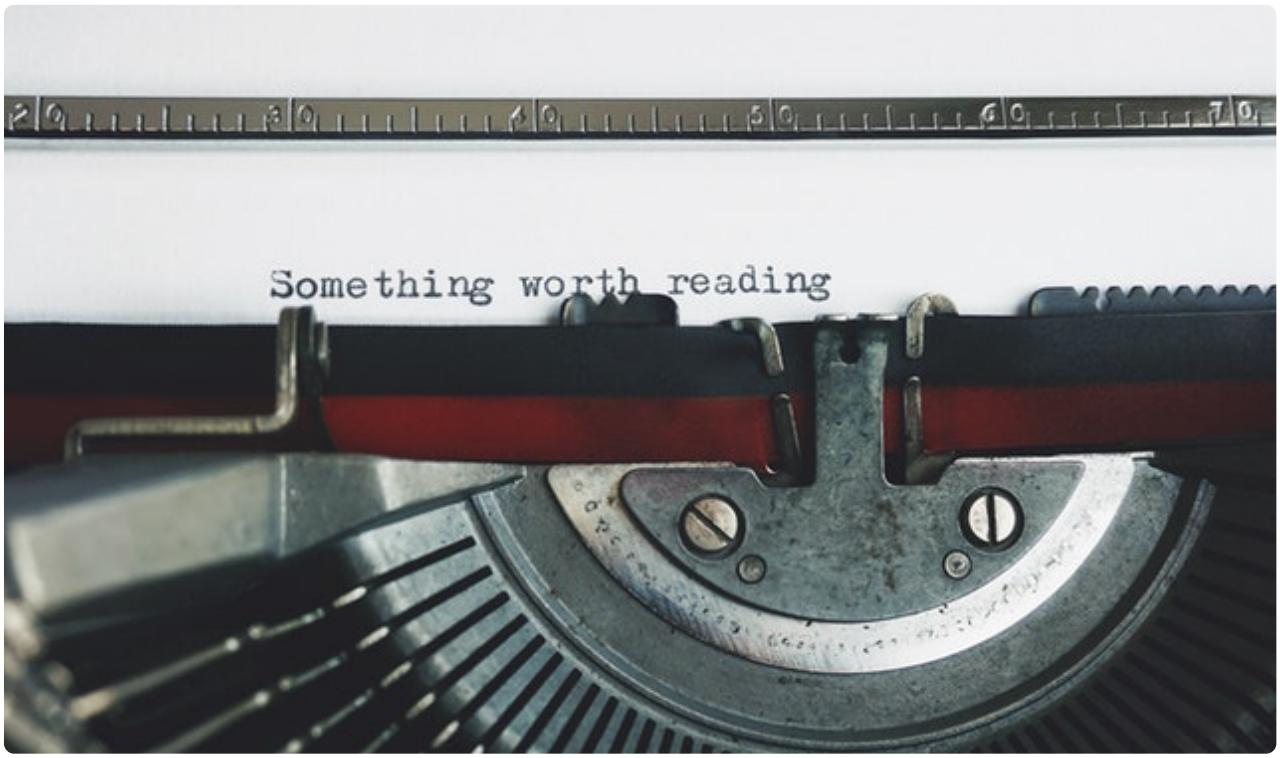
21 经常听到分库分表，但是怎样分呢？

更新时间：2020-05-11 09:27:46



> 人的差异在于业余时间。——爱因斯坦

当我们的业务不断发展、数据量急剧膨胀时，数据库性能成为系统瓶颈会越来越明显，例如：存储容量限制、连接池容量、读写性能、索引等等。由此，也就出现了各种技术或解决方案。其中，使用缓存和分库分表一定是出场率最高的两类解决方案。缓存是比较简单的，也就是把原本磁盘中（MySQL 的数据存储在磁盘中）的数据搬到内存中。而对于分库分表来说，就显得有些“神秘感”了。那么，这一节里，我将对分库分表进行详细的解读。

1. 分库分表概述

业务系统初运行时（一个大概的时间概念），单机数据库基本都是够用的。但是，随着业务扩张，我们会逐渐的切换到读写分离。即从库负责读，主库负责写，从库同步主库的数据，也就是经典的主从同步架构。但是，这解决的是并发访问的问题，如果业务发展的同时，数据量再迅速增长，就需要去考虑分库分表的架构了。

1.1 为什么要分库分表

对于 MySQL 来说，有人做过测算，单库的数据量在 5000 万以内时，性能表现是比较好的，超过这个值（并不是一定的，也要综合看数据的复杂度）之后，性能会随着数据量的增大而降低。对于单表来说，情况也是类似的，当单表的数据量达到 1000 万之后，即使添加从库、优化索引，性能也是下降的非常厉害。所以，分库分表的问题也就是单库或单表的问题：

- **单库数据量太大**：单库所在服务器磁盘空间受限、IO 瓶颈等
- **单表数据量太大**：增删改查速度慢、索引膨胀等等

当遇到这些问题之后，就需要考虑去做分库分表了。分库是将原来的单库分为多个更小的库，而分表是将原来的表分为多个更小的表。

1.2 什么时候去考虑分库分表

首先来说，分库分表是 MySQL 的高级应用，它的实现和使用难度都比较高，而且需要改动业务代码。所以，在不需要分库分表的时候，就不要这样做。这也是理所当然的，并不是说所有的表都需要切分，或者是表大到一定程度就需要切分，例如：数据报表通常很大，但是几乎不会对报表做切分。这也是关于分库分表的第一条建议：能不切分就不要切分（或者考虑使用其他的办法解决性能瓶颈）。

下面，我来对可以尝试去做分库分表的场景进行解释说明：

- 数据量过大（可以参考之前对单库和单表的介绍），对查询、插入、运维等等造成影响
- 表中存在过多大字段，例如 MEDIUMTEXT、LONGTEXT 等，考虑将这部分大字段拆分出去
- 数据量增长太快，即插入数据较多的业务系统，索引成为系统瓶颈

当然，关于分库分表的原因和场景还有很多，就像我们在做事一样，不一定要墨守成规。如果确实需要，且“故事”说得通，当然也可以去做分库分表。

2. 分库分表的策略

简单的说，分库分表就是按照某种规则，将单库或单表分为多库或多表。而这里的规则就是依据切分类型，分为**垂直切分**和**水平切分**。下面，我们就来看一看这两种切分方式的思想、方法和优缺点。

2.1 垂直切分

垂直切分可以同时作用于库和表，即垂直分库和垂直分表。垂直分库就是根据业务的特性，将关联程度不同的表放到不同的库中。它的思想与微服务的思想是类似的，就是将大的单体系统解耦，按照业务耦合的程度不同，切分成小的功能模块。例如：对于一个电商系统来说，库中可能会有用户表、商品表、订单表、反馈表等等；我们就可以将这些表分到三个库中：

- 用户信息库：用户表
- 商品及订单库：商品表、订单表
- 附属信息库：反馈表

这样，在查询不同的数据时就可以去查询不同的库，降低单库的查询和存储压力。

对于垂直分表来说，拆分的对象是数据表列，需要这样做的原因主要有两个：第一，数据表中字段数过多；第二，数据表中存在很多大数据列。此时，可以把一张表拆分成两张或者多张表，每张表中存储行记录的子集。同时，数据库以行为单位将数据加载到内存中，表中行记录较短，内存能加载更多的数据，命中率更高，减少了磁盘IO，从而提升了数据库性能。

垂直切分有很多优点，但是也有很多代价，下面，我来总结下这种切分方式的优缺点：

垂直切分的优点

- 消除库中存在的业务表耦合，使数据表之间的关系更加的清晰
- 将数据库的连接资源、单机硬件资源隔离开，更利于业务的扩展

垂直切分的缺点

- 表与表之间很难做到完整的 JOIN，只能通过多次查询的方式聚合数据
- 查询多个表会将单表事务升级为分布式事务，实现难度大大增加
- 仍然可能会存在单表数据量过大的问题

2.2 水平切分

水平切分也被称作是“横切”，它虽然是针对于数据表的。当一张数据表的数据量非常庞大，且即使是做了垂直拆分依然是存在瓶颈，这时候就需要对表进行水平切分了。即将一张表的数据记录按照一定的规则分散到多张表或多个库（多个库存储拆分出的多张表）中。最终使得每张拆分的表记录只是原表的子集，大大降低了单表的数据量。

把一张表的记录分散到多张表（表存在多个库中也是一样的道理，总的思想肯定是分出来多张表）中，那么，这个分散怎么去做呢？即水平切分的规则有哪些呢？下面，我来介绍一些经典的水平拆分规则：

- **按照时间区间或者自增 id 来切分**：这比较好理解，就是数据分段存储，例如100万行数据一个表，或者一个季度的数据切分为一个表。这样做的优点是简单、单表数据量可控；但是缺点也很明显，非常容易造成热点数据（最近的数据访问的频率最高）
- **哈希取模切分**：这个规则是计算某一列或几列的哈希值，再去取模散列到对应的表中。通常，我们会选择 user_id 做哈希取模。这种方式的优点是数据分片比较均匀，不易出现热点问题；缺点是扩容成本高（可以考虑一致性哈希）

相对来说，水平切分不会存在单表、单库数据量过大的问题，且应用改动的成本也会比较低。但同时，它也与垂直切分有着相似的缺陷。例如：数据记录跨多表时，也存在分布式事务问题；表之间的 JOIN 过程将会非常困难等等。

3. 分库分表引发的问题

分库分表虽然能够缓解单库、单表对数据库造成的压力瓶颈，但是也同样会带来许多问题。了解这些问题，并确定能够给出解决方案时，再去考虑对你的业务系统进行分库分表。否则，不要“轻举妄动”，很可能会是事倍功半的。

3.1 全局主键问题

由于分库分表之后，一张表会跨越多张表，甚至是多个库，此时，数据库的自增 id 将会变得没有意义。因此，我们必须需要单独设计全局主键，以避免主键重复，引起业务系统的 KEY 冲突问题。

其实，这是一个比较成熟的问题，即有很多解决方案。下面，我来总结说明几种常见的做法：

- **UUID**：它被称为通用唯一识别码，由一组32位数的16进制数字所构成，目的是让分布式系统中的所有元素都能有唯一的识别信息。它可以本地生成，性能很高；但是，由于它很长，会占用大量的存储空间
- **额外的自增表**：单独创建一张表，只有一个自增主键，而这个主键则用于分库分表的全局主键。这种做法需要依赖于其他表，且存在单点问题，当这张自增表出现故障，导致系统不可用
- **分布式全局唯一 ID 生成算法**：这种算法有很多，例如：Twitter 的 snowflake、美团的 Leaf 等等

由于前两种解决方案各自存在的问题较为明显，且不易解决。所以，业界在全局主键的生成问题上，都会使用 “分布式全局唯一 ID 生成算法”。记住，在有很多开源实现时，就不需要自己重复造轮子。

3.2 事务一致性问题

分库之后，当业务更新（插入、更新、删除）的数据分布在不同的库中时，就会带来跨库事务问题。对于分布式事务来说，一般可以采用 “XA 协议”、2PC（两阶段提交）或 3PC（三阶段提交）来提交。

虽然分布式事务解决方案最大程度的保证了数据库操作的原子性，但是在提交事务时需要协调多方，对提交事务造成了延迟，且会增高死锁的概率。所以，如果我们的业务系统不追求瞬时的强一致性，可以采用事务（日志）补偿的方式来达到最终的一致性。

3.3 关联查询（JOIN）问题

在分库分表之前，我们可以通过 JOIN 多表的方式来查询复杂的数据。但是切分之后，数据可能分布在不同的节点上，此时再去 JOIN 几乎是不可能的事情。所以，对于分库分表的情况，我们通常的建议是：“抛弃 JOIN”。

那么，为了解决关联查询的问题，我们可以想一些别的办法。例如：

- **字段冗余设计**：这是一种反范式的设计，也是空间换时间的典例，它是将需要多次用到的数据分布到多张表中，避免了 JOIN 查询
- **数据组装**：也就是多次查询，将多次查询的数据组装在一起构成整体数据
- **拆分查询**：注意，这里所说的查询指的是前端发起的查询请求，即前端把复杂查询（多表）拆分成多次简单查询（单表）

由此，可以得出结论：分库分表对于 “大” 业务系统几乎是不可避免的，但是，分库分表同样存在非常严重的缺陷。如果你不能解决这些问题或给出合理的解决方案，慎用分库分表，反而是可以考虑使用分布式数据库（例如 HBase）去代替。

4. 分库分表案例

说了很多分库分表的理论、注意事项与可能存在的问题，下面，我们来看看一个真实的案例，怎样对一张大表做切分，以及对于可以预见的分库分表，怎样去定义业务表。

4.1 用户信息业务场景

用户信息一定是存储在用户表中，通常我们直接称之为 user 表。随着业务发展，注册的用户越来越多（这通常都是大型项目面临的问题，不过，你可以随意扩展到用户订单表、商品表等等类似的场景），user 表中的记录也会膨胀的越来越厉害。最终，延迟过大、性能过低，不得不采取分库分表的手段解决问题。好的，我先给出 user 表的建表语句：

```
CREATE TABLE `user` (
  `user_id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '自增主键',
  `user_name` varchar(128) NOT NULL DEFAULT '' COMMENT '用户名',
  `password` varchar(128) NOT NULL DEFAULT '' COMMENT '密码',
  `email` varchar(128) NOT NULL DEFAULT '' COMMENT '电子邮箱',
  `phone` varchar(128) NOT NULL DEFAULT '' COMMENT '手机号码',
  `gender` tinyint(4) NOT NULL DEFAULT '0' COMMENT '性别',
  `age` tinyint(4) NOT NULL DEFAULT '0' COMMENT '年龄',
  `id_card` varchar(128) NOT NULL DEFAULT '' COMMENT '身份证号码',
  `intro` varchar(1024) NOT NULL DEFAULT '' COMMENT '个人信息',
  `user_company` varchar(50) NOT NULL DEFAULT '' COMMENT '用户公司',
  `user_department` varchar(45) NOT NULL DEFAULT '' COMMENT '用户部门',
  `user_duty` varchar(100) NOT NULL DEFAULT '' COMMENT '用户具体职责',
  `user_industry` varchar(100) NOT NULL DEFAULT '' COMMENT '用户所处行业',
  `user_status` int(10) NOT NULL DEFAULT '0' COMMENT '用户状态',
  `create_time` datetime NOT NULL DEFAULT '1970-01-01 00:00:00' COMMENT '创建时间',
  `update_time` datetime NOT NULL DEFAULT '1970-01-01 00:00:00' COMMENT '更新时间',
  PRIMARY KEY (`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='用户信息表';
```

注意到这张表定义了很多列（实际情况可能会更加复杂，而且会有很多索引），且有不少列是字符类型。即使是我们可以通过在 SELECT 语句中指定需要的列数据，也不可避免的会让这张表存储大量的数据。

不论是做什么，你都需要遵循一定的原则：当你想做一件事之前，你一定要把“前因后果”想清楚了。对于当前的用户表来说，我们需要搞清楚在业务逻辑中会怎样使用这张表：

- 1% 的用法：使用 user_name、email、phone 登录系统（因为登陆之后，session 会一直保存，不需要用户重复登录）
- 99% 的用法：使用 user_id 查询用户信息（各个服务模块都可能需要用户信息）

目前为止，我们已经搞清楚了业务场景和需求。对于遇到的问题，也是非常明显的：user 表会越来越大，严重影响查询、插入和更新的性能。

4.2 用户信息切分

现在开始，我们需要对 user 表进行切分了，关于切分，你需要遵守一个规则：

> 先垂直切分，再做水平切分

具体为什么要这样做，这个问题留给大家去思考。我们先按照垂直切分，把 user 表切分为两张表（或者更多的表）。如下所示：

```sql
-- 用户基本信息表
CREATE TABLE `user_base` (
  `user_id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '自增主键',
  `user_name` varchar(128) NOT NULL DEFAULT '' COMMENT '用户名',
  `password` varchar(128) NOT NULL DEFAULT '' COMMENT '密码',
  `email` varchar(128) NOT NULL DEFAULT '' COMMENT '电子邮箱',
  `phone` varchar(128) NOT NULL DEFAULT '' COMMENT '手机号码',
  `gender` tinyint(4) NOT NULL DEFAULT '0' COMMENT '性别',
  `age` tinyint(4) NOT NULL DEFAULT '0' COMMENT '年龄',
  `user_status` int(10) NOT NULL DEFAULT '0' COMMENT '用户状态',
  `create_time` datetime NOT NULL DEFAULT '1970-01-01 00:00:00' COMMENT '创建时间',
  `update_time` datetime NOT NULL DEFAULT '1970-01-01 00:00:00' COMMENT '更新时间',
  PRIMARY KEY (`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='用户基本信息表';

-- 用户附加信息表
CREATE TABLE `user_extra` (
  `user_id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '自增主键',
  `id_card` varchar(128) NOT NULL DEFAULT '' COMMENT '身份证号码',
  `intro` varchar(1024) NOT NULL DEFAULT '' COMMENT '个人信息',
  `user_company` varchar(50) NOT NULL DEFAULT '' COMMENT '用户公司',
  `user_department` varchar(45) NOT NULL DEFAULT '' COMMENT '用户部门',
  `user_industry` varchar(100) NOT NULL DEFAULT '' COMMENT '用户所处行业',
  `user_duty` varchar(100) NOT NULL DEFAULT '' COMMENT '用户具体职责',
  PRIMARY KEY (`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='用户附加信息表';
```

可以看到，user 表被拆分成了 user_base 和 user_extra 两张表，关于它们：

- 两张表各自保存 user 表的部分信息，即是 user 表数据的子集
- 两张表都定义了 user_id，作为数据关联使用（user_id 是分库分表的核心，理论上所有的表都需要定义 user_id）
- user_base 表中不包含“复杂信息”，因为这些使用的频率不是很高

经过这样分表之后，我们的大部分操作都应该直接对应到 user_base 表。除非需要更详细的信息，才需要去访问 user_extra 表。但是，垂直切分不能避免单表数据量过大，当数据量继续增长时，我们就需要做水平切分了。

对表做水平切分我们选择使用“哈希取模”的方式，这里，user_id 就是天然的划分依据。首先，需要设计两个哈希函数（简单的哈希算法就可以）：

- HASH_1：确定水平切分的记录属于哪个数据库
- HASH_2：确定水平切分的记录属于哪张表

接下来，我们需要预估用户增长的规模，例如：注册用户数能够达到1亿。那么，可以计算：

- 数据分到4个库（库的个数可以自定义）中，那么，每个库包含：1亿 / 4 = 2500万
- 每个库中定义5张表（表的个数也可以自定义），那么，每张表包含：2500万 / 5 = 500万

那么，可以先通过 HASH_1(user_id) % 4 确定用户记录属于哪个库，再通过 HASH_2(user_id) % 5 确定用户记录属于哪张表。自此，也就完成了用户信息的分库分表。

5. 总结

分库分表听起来是很有意思的，但是做起来显然没有那么简单，你需要考虑的问题太多，更让你头疼的是这些问题往往没有很好的解决方案。所以，大多数时候，理解分库分表的思想和原理即可。不到万不得已，不要轻易尝试。

6. 问题

使用一致性哈希会降低扩容成本，你知道这是为什么吗？

分库分表为什么要先垂直切分，再去水平切分？这样做有什么优点，或者说反过来有什么缺点？

对于分库分表来说，user_id 是不可或缺的，这是为什么呢？

数据表做了切分之后，你知道怎么去做数据同步呢？如果想要不中断业务运行，又该怎么做呢？

user 表拆分之后，如果我想用 user_name、email、phone 去查询用户信息，怎样做效率会高呢（注意，数据分布在多个库、多个表中，轮询自然是不合理的）？

7. 参考资料

《高性能 MySQL（第三版）》

MySQL 官方文档：Database Backup Methods

MySQL 官方文档：Optimizing for InnoDB Tables

MySQL 官方文档：InnoDB Cluster

}

20 数据汇总优化查询方案设计

22 binlog 实现增量数据收集方案的设计