# 08 | 如何使用分库分表支持海量数据的写入?

在上一模块里，我们讲解了如何使用懒加载、全量缓存等机制构建一个能够支撑百万并发的读服务，同时介绍了如何使用自动化回放来提升读服务的测试与回归效率，直接跳读到本模块的同学有空可以返回去学习一下。

在本模块的四讲里，将会介绍如何构建与读服务有着相反差异的写服务。并发百万的读服务每一次请求都不会产生新数据，是无状态的。而写服务不说并发百万，只要并发上万，一天产生的数据量也在亿级左右。本讲将要介绍如何存储这些海量数据，同时保证相对应的写入和查询的性能，以及业务流程不发生太大变化。

不管是打车的订单、电商里的支付订单，还是外卖或团购的支付订单，都是后台服务中最重要的一环，关乎公司的营收。因此，本讲及本模块都将以**订单业务**作为案例进行分析。

## 是否真的要分库?

分库当然能够解决存储的问题，假设原先单库只能最多存储 2 千万的数据量。采用分库之后，存储架构变成下图 1 所示的分库架构，每个分库都可以存储 2 千万数据量，容量的上限一下提升了。

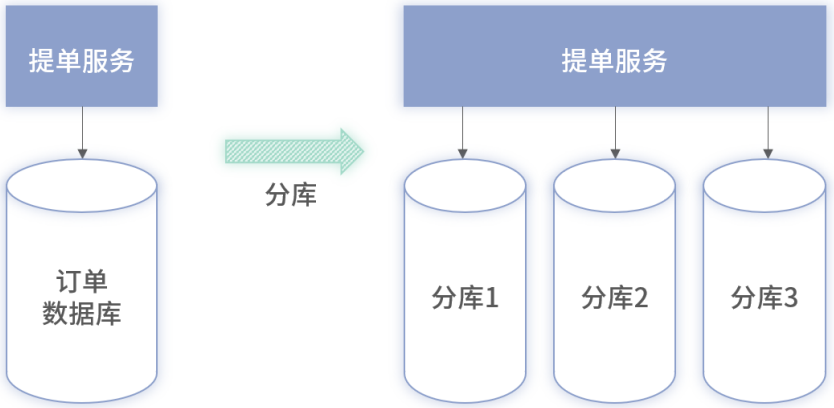




图 1：分库架构图

容量提升了，但也带来了很多其他问题。比如：

1. 分库数据间的数据无法再通过数据库直接查询了。比如跨多个分库的数据需要多次查询或借助其他存储进行聚合再查询。
2. 分库越多，出现问题的可能性越大，维护成本也变得更高。
3. 无法保障跨库间事务，只能借助其他中间件实现最终一致性。

所以在解决容量问题上，可以根据业务场景选择，不要一上来就要考虑分库，分表也是一种选择。

分表是指所有的数据均存在同一个数据库实例中，只是将原先的一张大表按一定规则，划分成多张行数较少的表。**它与分库的区别是，分表后的子表仍在原有库中，而分库则是子表移动到新的数据库实例里并在物理上单独部署**。分表的拆分架构如下图 2 所示：

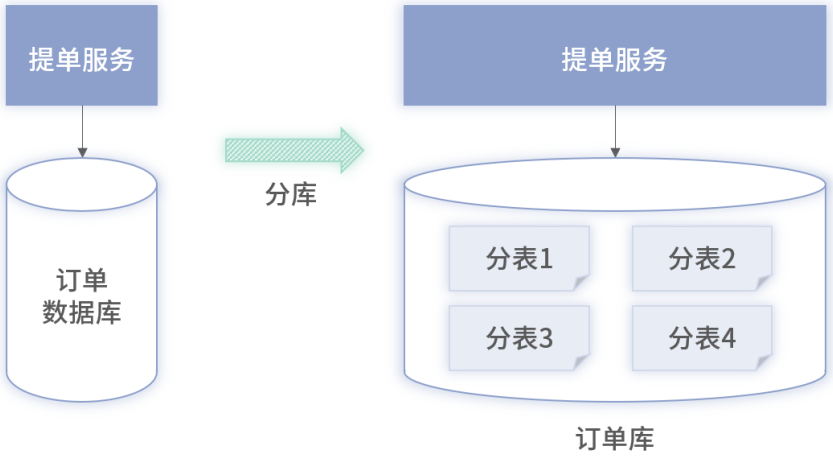




图 2：分表架构图

以本模块的订单案例来说，假设订单只是**单量多而每一单的数据量较小，这就适合采用分表**。单条数据量小但行数多，会导致写入（因为要构建索引）和查询非常慢，但整体对于容量的占用是可控的。采用分表后，大表变成小表，写入时构建索引的性能消耗会变小，其次小表的查询性能也更好。如果采用了分库，虽然解决了写入和查询的问题，但每张表所占有的磁盘空间很少，也会产生资源浪费。两种方案的对比如下图 3 所示：

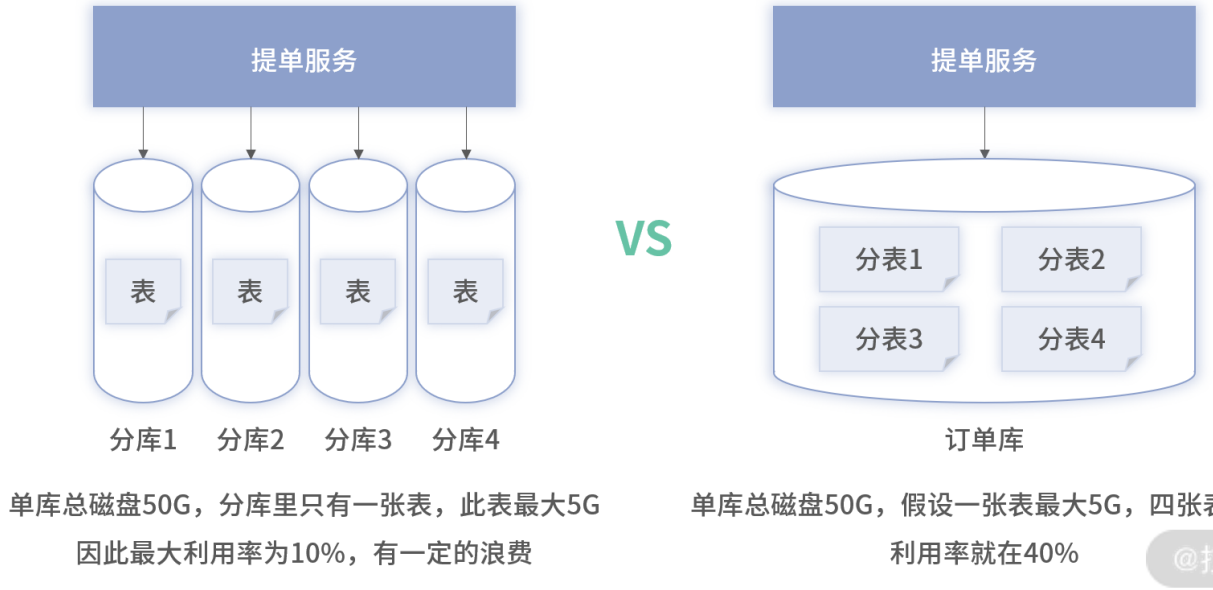




图 3：单表行数多单数据量小的对比图

在实际场景里，因为要详细记录用户的提单信息，单个订单记录的数据量均较多，所以不存在行数多但单条数据量小的情况。但在其他写入服务里，经常会出现上述场景，你可以优先采用分表的方案。因为**分表除了能解决容量问题，还能在一定程度上解决分库所带来的三个问题**。

1. 分表后可以通过 join 等完成一些富查询，相比分库简单得多。
2. 分表的数据仍存储在一个数据库里，不会出现很多分库。无须引入一些分库中间件，因此维护成本和开发成本均较低。
3. 因为在同一个数据库里，也可以很好地解决事务问题。

接下来将介绍如何应对行数多且单行数据量较大的场景。通过我们前面的分析，我想你已经知道答案了——采用分库的方案。

## 如何实现分库?

在决定对数据库进行分库后，首先要解决的问题便是**如何选择分库维度**。不同的分库维度**决定了部分查询是否能直接使用数据库，以及是否存在数据倾斜的问题。**

## 分库维度选择

下面以订单为案例，介绍两种常见不同维度的分库方式：按**直接满足最重要的业务场景划分和最细粒度随机分**。

首先我们来看按**直接满足最重要的业务场景划分。在业务上，所有的订单数据都是隶属于某一个用户的。在选择分库维度时，可以按订单归属的用户**这个字段进行分库。按此维度分库后，同一个用户的订单都在某一个分库里。分库后的场景如下图 4 所示：

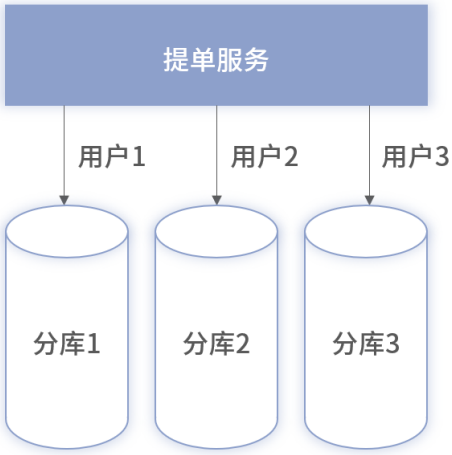



图 4：按购买用户进行分库的架构图

订单模块除了提供提交订单接口外，还会提供给售卖商家对自己店铺的订单进行查询及修改等功能。这些维度的查询和修改需求，在采用了按购买用户进行分库之后，均无法直接满足了。

这里请你思考一个问题，**订单模块最重要的功能是什么**？

答案是保证客户（即买家）的各项订单功能能够正常使用，比如下单、下单后立刻（无延迟）查看已购的订单信息、待支付、待发货、待配送的订单列表等。相对来说，订单里的商品售卖方（即卖家）所使用的功能并不是优先级最高的。因为当我们要对卖家和买家的功能做取舍时，卖家是愿意降低优先级的，毕竟卖家是买卖的受益方。

按购买用户划分后，用户的使用场景都可以直接通过分库支持，而不需要通过异构数据（存在数据延迟）等手段解决，对用户来说体验较好。其次，在同一个分库中，便于修改同一用户的多条数据，因此也不存在分布式事务问题。

我们可以通过上述订单案例抽象出一个分库准则，即**在确定分库字段时应该以直接满足最重要的业务场景为准**。很多其他的业务都参考了这一准则，比如：

1. 对于微博和知乎等用户生产内容（UGC）的业务，均会按用户进行分库。因为用户新发布文章后就会去查看列表。
2. 支付系统里，也会按用户的支付记录进行分库。
3. 在技术上，比如一个微服务下的监控数据，同样会按微服务进行划分。同一个微服务的监控数据均存储在一个分库里，你可以直接在一个分库里查看微服务下的所有监控数据。

上述划分方法虽然直接满足了最重要的场景，但可能会出现数据倾斜的问题，比如出现一个超级客户（如企业客户），购买的订单量非常大，导致某一个分库数据量巨多，就会重现分库前的场景。这属于最极端的情况之一。

对于倾斜的问题，可以采用**最细粒度的拆分**，即按数据的唯一标示进行拆分，对于订单来说唯一标示即为订单号。采用订单号进行分库之后，用户的订单会按 Hash 随机均匀地分散到某一个分库里。这样就解决了某一个分库数据不均匀的问题。

对于上个小节里的案例，也可以用此手段进行处理。比如：

1. 按用户的每一条微博随机分库；
2. 按用户的每一笔支付记录随机分库；

3. 同一个微服务里的每一个监控点的数据随机分库。

采用最细粒度分库后，虽然解决了数据均衡的问题，但又带来了其他问题。

1. 首先便是除了细粒度查询外，其他任何维度的查询均不支持。这就需要通过异构等方式解决，但异构有延迟、对业务是有损的。
2. 其次采用最细粒度后，对于防重逻辑在数据库层面已经无法支持。比如用户对同一个订单在业务上只能支付一次这一诉求，在支付系统按支付号进行分库后便不能直接满足了。因为上述分库方式会导致不同支付单分散在不同的分库里，此时，期望在数据库中通过订单号的唯一索引进行支付防重就不可实施了。

上述两种分库的方式，在解决问题的同时又带来一些新的问题。**在架构中，没有一种方案可以解决所有问题的，更多的是根据场景去选择更适合自己的方案**。

## 全局唯一标示

不管采用何种维度的分库方式，使用原有单库的数据库自增主键生产数据标示的方案已经不可以使用了。对于全局的数据唯一标示，有两种常见的生成方式。

**1. 使用算法随机生成。**

比如使用机器 IP、时间戳、随机数等进行组合，生成一个唯一编号。业界成熟的有 Twitter 推出的雪花算法。需要注意的是，为了保证唯一性，雪花算法增加了很多随机因子，导致计算出来的唯一标示特别长，达到 19 位。

在 JavaScript 里，数据精度和 Java 等语言不完全一致，太长的雪花 ID 在前端存在溢出的问题。因为雪花算法生成的 ID 为 Long 类型，可以采用类似 Base64 等算法，对原始 ID 进行压缩转换为 String 类型，降低长度并避免和 JavaScript 精度不统一导致的问题。

**2. 基于数据库主键构建一个 ID 生成服务。**

虽然不能在插入的时候使用数据库唯一主键，但可以在插入前通过一个服务获取全局唯一的 ID。ID 生产服务可以基于一张单表实现，每一次外部请求时，均生产一个新的 ID。通过此方式，可以获得长度较短且为数值类型的全局唯一编号。

但如果每次获取 ID 时，ID 生成服务都需要从数据库实时获取，性能会比较差。为了解决性能问题，可以在生成 ID 的数据库前置一个具备持久化功能的内存缓存，预生成一批 ID。具体架构如下图 5 所示：

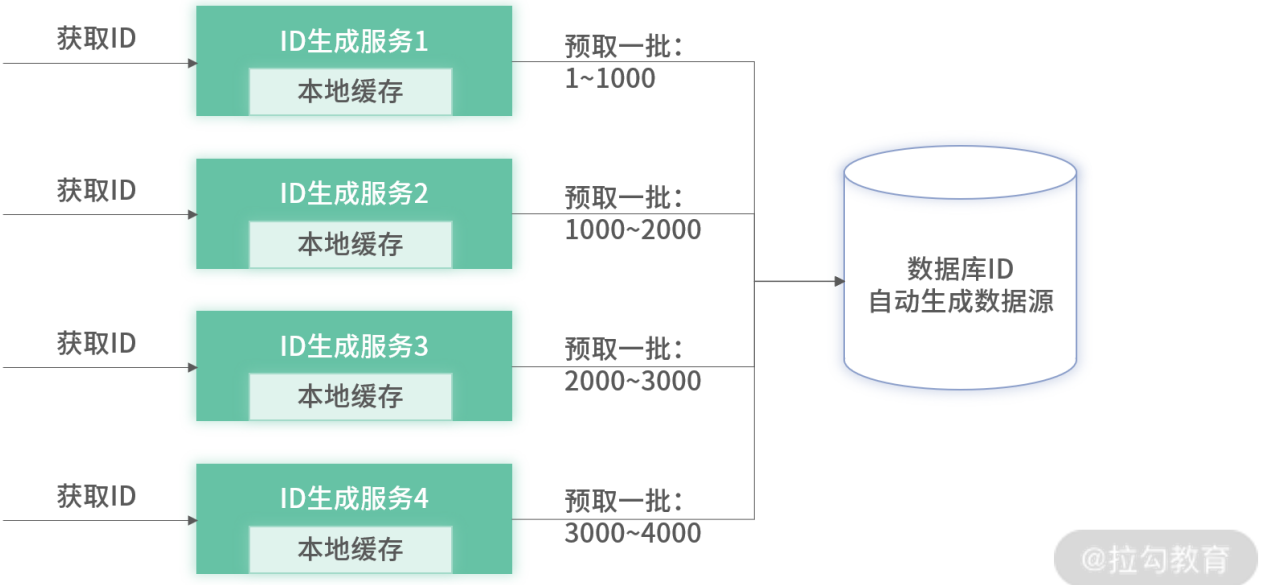


图 5：预生成 ID 架构图

## 分库中间件选择

现在开源提供分库支持的中间件较多，如 MyCat 等，**整体上各类分库中间件可以分为两大类：一种是代理式、另外一种是内嵌式。**

**代理式分库中间件**对于业务应用无任何侵入，业务应用和未分库时一样使用数据库，分库的选择及分库的维度对业务层完全隐藏，接入和使用成本极低。代理式的架构如下图 6 所示：

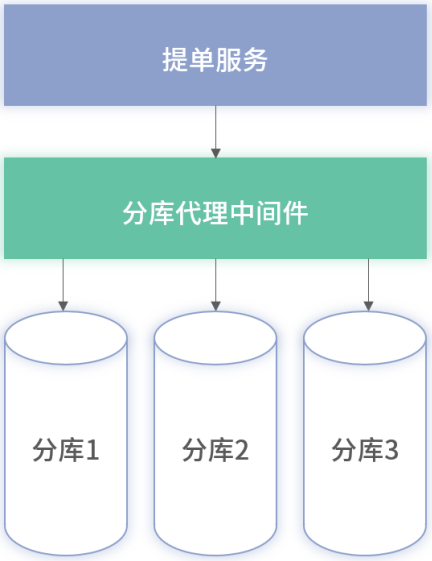




图 6：代理式分库架构图

代理式虽有使用成本低的好处，但也存在其他一些问题。

1. 代理式在业务应用和数据库间增加了一层，导致了性能下降。
2. 代理式需要解析业务应用的 SQL，并根据 SQL 中的分库字段进行路由。它需要解析和适配所有 SQL 语法，增加了代理模块复杂度和出错的可能性。
3. 代理层是单独进程，需要部署占用资源，带来一定的成本。

**内嵌式分库中间件**是将分库中间件内置在业务应用中，它只负责分库的选择，并不会解析用户的 SQL。在使用时，业务应用需将分库字段传递给内嵌中间件去计算具体对应的分库。它相比代理式性能更好。内嵌式的架构如下图 7 所示：

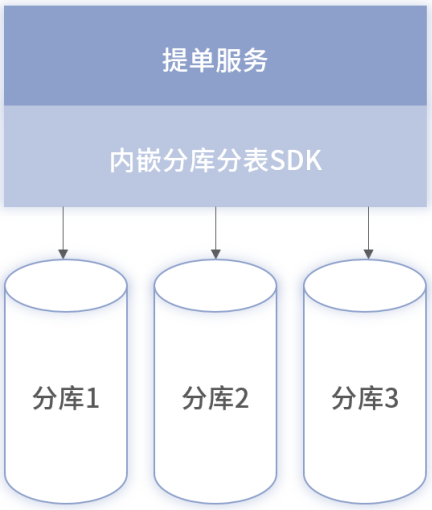




图 7：内嵌式的分库架构图

除了性能优势外，内嵌式同样存在问题。

1. 有一定侵入性，业务应用与原始单库模式相比，需要进行一定的改造去适配内嵌式的 API。
2. 分库在故障转移、数据迁移等运维工作时，需要业务应用感知。不过现在的一些内嵌式代理，已经具备非常良好的配置功能，在分库运维时，业务应用需要配合的内容较少。

## 其他问题

接下来，再来看几个常见问题的应对策略。

**1. 是否一定需要进行分表或者分库呢**?

不一定。虽然很多互联网公司的体量很大，用户非常多，但你千万不要被这些现象迷惑了。实际上，90% 以上的系统能够发展到上百万、上千万数据量已经很不错了。对于千万的数据量，开源的 MySQL 都可以很好地应对，更别说一些商业数据库了。

另外，当数据增长到一定量级后，可以在业务层面做一些处理。比如根据业务特点，对无效数据、软删除数据，以及业务上不会再查询的数据进行统一归档，这也是一个成本低、效果明显的方式了。

**2. 使用业务字段分库后，如何处理数据倾斜**?

如果数据量不是特别大，可以在分库基础上，再进行分表。针对数据量较大的场景，可以使用二次分库的方式。对于订单量较多的用户，可以在用户账号基础上再增加一个字段，做进一步的分库，但此用户的查询就会有损了。

此外，还有另外两个问题，由于需要用到暂未讲解的知识，所以我将放在后面的章节结合相关知识详细讲解，今天仅做提及。

**3. 如何满足富查询?**

富查询是一个无法回避的问题，即采用分库分表之后，如何满足跨越分库的查询？对于此问题，我将在“**第 11 讲**”进行详细讲解。

**4. 如何解决跨多库的修改导致的分布式事务?**

跨多库的修改及多个微服务间的写操作导致的分布式事务问题，我将在“**第 19 讲**”里集中讲解。

## 总结

不断进行分库分表一定能解决容量问题，但“杀敌一千，自损八百”的事情少做为宜。使用分库分表会将代码和架构的复杂度变高，带来资源成本上升等问题。另外，在使用系统时，用户（不管是客户还是管理员）的查询体验也存在一定的降级。

在使用分库分表前，你需要确定这是否是最优选择，是否能通过其他更简单的手段处理无效数据清理？架构是通过最小代价解决问题，而不是技术工具的比拼。

**最后，我再给你留一道讨论题，你知道的分库分表的问题还有哪些或者上述问题你还有哪些解决方案**？欢迎留言，我们一起在留言区讨论。

下一讲将介绍 09 | 如何打造无状态的存储实现随时切库的写入服务?