小，它总会发生。”对应到更换数据库这件事情上，就是在更换数据库的过程中，只要有一点可能会出问题的地方，哪怕出现问题的概率非常小，它都会出问题。

实际上，无论是新版本的程序还是新的数据库，即使我们做了严格的验证测试，实现了高可用方案，对于刚刚上线的系统，它的稳定性也是不够好的。需要有一个磨合的过程，才能逐步达到一个稳定的状态，这是客观规律。这个过程中一旦出现故障，如果不能及时恢复，那么其所造成的损失往往是我们难以承担的。

所以我们在设计迁移方案的时候，一定要保证每一步都是可逆的。也就是必须保证，每执行完一个步骤，一旦出现任何问题，都能快速回滚到上一个步骤。这是很多开发人员在设计这种升级类技术方案的时候比较容易忽略的问题。

接下来，我们还是以订单库为例来说明这个迁移方案应该如何设计。

首先要做的一点是，把旧库的数据全部复制到新库中。因为旧库还在服务线上业务，所以不断会有订单数据写入旧库，我们不仅要向新库复制数据，还要保证新旧两个库的数据是实时同步的。所以，需要用一个同步程序来实现新旧两个数据库的实时同步。

可以使用 Binlog 实现两个异构数据库之间数据的实时同步，具体操作请回顾第 12 章对这个方法的讲解。这一步不需要回滚，因为这里只增加了一个新库和一个同步程序，对系统的旧库和程序没有任何改变。即使新上线的同步程序影响到了旧库，停掉同步程序也就可以了。此时系统的架构如图 13-1 所示。

然后，需要改造一下订单服务，业务逻辑部分不需要变动，DAO 层需要进行如下改造。

1）支持双写新旧两个库，并且预留热切换开关，能通过开关控制三种写状态：只写旧库、只写新库和同步双写。

2）支持读取新旧两个库，同样预留热切换开关，控制读取旧库还是新库。

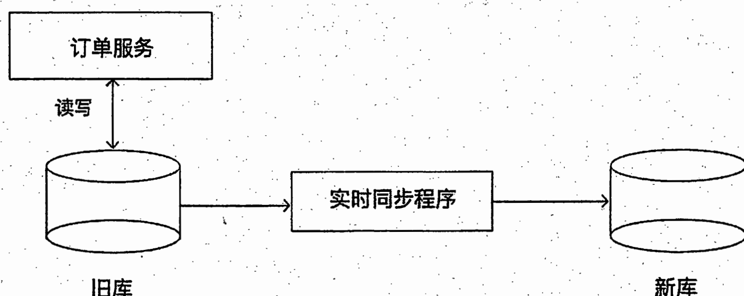


图 13-1　增加一个新库和实时同步程序

然后，上线新版的订单服务，这个时候，订单服务仍然是只读写旧库，不读写新库。让这个新版的订单服务稳定运行至少一到两周的时间，其间我们不仅要验证新版订单服务的稳定性，还要验证新旧两个订单库中的数据是否保持一致。这个过程中，如果新版订单服务出现任何问题，都要立即下线新版订单服务，回滚到旧版本的订单服务。

稳定一段时间之后，就可以开启订单服务的双写开关了。开启双写开关的同时，需要停掉同步程序。这里有一个需要特别注意的问题是，这里双写的业务逻辑，一定是先写旧库，再写新库，并且以旧库的结果为准。

如果旧库写成功，新库写失败，则返回成功，但这个时候要记录日志，后续我们会根据这个日志来验证新库是否还有问题。如果旧库写失败，则直接返回失败，同时也不再写新库了。这么做的原因是，不能让新库影响到现有业务的可用性和数据准确性。上面这个过程如果出现任何问题，都要关闭双写，回滚到只读写旧库的状态。

切换到双写之后，新库与旧库的数据可能会出现不一致的问题。原因有两点：一是停止同步程序和开启双写，这两个过程很难做到无缝衔接；二是双写的策略也不能保证新旧库的强一致性。对于这个问题，我们需要上线一个比对和补偿的程序，用于比对旧库最近的数据变更，然后检查新库中的数据是否一致，如果不一致，则需要进行补偿。此时系统的架构如图 13-2 所示。

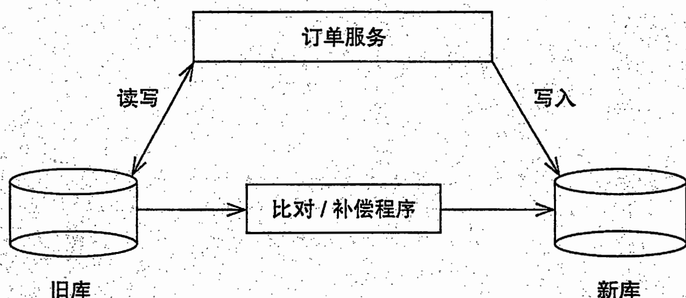


图 13-2 切换到双写并开启数据比对和补偿

开启双写之后，还需要稳定运行至少几周的时间，并且在这期间我们需要不断地检查，以确保不能有旧库写成功、新库写失败的问题。如果在几周之后比对程序发现新旧两个库的数据没有不一致的情况，那就可以认为新旧两个库的数据一直都是保持同步的。

接下来就可以用类似灰度发布的方式把读请求逐步切换到新库上。同样，运行期间如果出现任何问题，都要再切回到旧库。将全部读请求都切换到新库上之后，其实读写请求已经全部切换到新库上了，虽然实际的切换已经完成，但后续还有需要收尾的步骤。

再稳定一段时间之后，就可以停掉比对程序，把订单服务的写状态改为只写新库。至此，旧库就可以下线了。注意，在整个迁移过程中，只有这个步骤是不可逆的。由于这一步的主要操作就是摘掉已经不再使用的旧库，因此对于正在使用的新库并不会有什么影响，实际出问题的可能性已经非常小了。

至此，我们完成了在线更换数据库的全部流程。双写版本的订单服务也完成了它的历史使命，可以在下一次升级订单服务版本的时候下线双写功能。

13.2 如何实现比对和补偿程序

在上面的数据库切换过程中，如何实现比对和补偿程序是整个切换设

计方案中的一个难点。这个比对和补偿程序的实现难点在于，我们要比对的是两个随时都在变化的数据库中的数据。在这种情况下，我们没有类似复制状态机这样理论上严谨、实际操作还很简单的方法来实现比对和补偿。但我们还是可以根据业务数据的实际情况，有针对性地实现比对和补偿，经过一段时间之后，把新旧两个数据库的差异逐渐收敛到一致。

像订单这类时效性比较强的数据，是比较容易进行比对和补偿的。因为订单一旦完成之后，几乎就不会再改变了，比对和补偿程序就可以根据订单完成时间，每次只比对这个时间窗口内完成的订单。补偿的逻辑也很简单，发现不一致的情况后，直接用旧库的订单数据覆盖新库的订单数据就可以了。

这样，切换双写期间，对于少量不一致的订单数据，等到订单完成之后，补偿程序会将其修正。后续在双写的时候，只要新库不是频繁写入失败，就可以保证两个库的数据完全一致。

比较麻烦的是更一般的情况，比如像商品信息之类的数据，随时都有可能会发生变化。如果数据上带有更新时间，那么比对程序就可以利用这个更新时间，每次从旧库中读取一个更新时间窗口内的数据，到新库中查找具有相同主键的数据进行比对，如果发现数据不一致，则还要比对一下更新时间。如果新库数据的更新时间晚于旧库数据，那么很可能是比对期间数据发生了变化，这种情况暂时不要补偿，放到下个时间窗口继续进行比对即可。另外，时间窗口的结束时间不要选取当前时间，而是要比当前时间早一点，比如 1 分钟之前，这样就可以避免比对正在写入的数据了。

如果数据没带时间戳信息，那就只能从旧库中读取 Binlog，获取数据变化信息，然后到新库中查找对应的数据进行比对和补偿。

这里需要特别说明的一点是，如果严格推敲，上面这些方法都不是百分之百严谨的，并不能保证在任何情况下经过比对和补偿后，新库的数据与旧库的是完全一样的。但是，在大多数情况下，这些实践方法还是可以有效地收敛新旧两个库的数据差异，大家可以酌情采用。

13.3 小结

设计在线切换数据库的技术方案，首先要保证数据的安全性，确保对于每个步骤，一旦失败，都可以快速回滚。此外，我们还要确保在迁移的过程中不会丢失数据，这一点主要是依靠实时同步程序和比对 / 补偿程序来实现。

下面就来把这个复杂的切换过程按照顺序总结成 7 个要点，供大家参考。

1）上线同步程序，把数据从旧库中复制到新库中，并保持实时同步。

2）上线双写订单服务，只读写旧库。

3）开启双写，同时停止同步程序。

4）开启比对和补偿程序，以确保新旧数据库的数据完全一样。

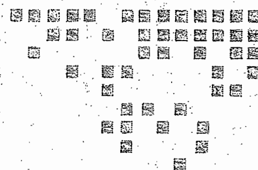
5）将服务对数据库的读请求逐步切换到新库上。

6）下线比对补偿程序，关闭双写，将读写请求都切换到新库上。

7）下线旧库和订单服务的双写功能。

13.4 思考题

在数据库的整个切换方案中，只有一个步骤是不可逆的，那就是由双写切换为新库单写。请思考：如果可以不计成本，我们应该如何修改迁移方案，让这一步也能实现快速回滚？



第 14 章 Chapter 14

对象存储：最简单的分布式存储系统

存储像图片、音视频之类的大文件，最佳的选择就是对象存储。对象存储不仅有很好的大文件读写性能，还可以通过水平扩展实现近乎无限的存储容量，除此之外，还可以兼顾服务高可用性、数据高可靠性这些特性。

对象存储之所以能够做到这么“全能”，最主要的原因是，对象存储是原生的分布式存储系统。这里的“原生的分布式存储系统”，是相对于MySQL、Redis 这类单机存储系统来说的。虽然这些非原生的存储系统，也具备一定的集群能力，但是使用它们构建大规模的分布式集群，其实是非常困难的。

随着云计算的普及，很多新生代的存储系统，都是原生的分布式系统，它们最初的设计目标之一就是分布式存储集群，比如，Elasticsearch、Ceph 和国内很多一线云服务厂商推出的新一代数据库，大多可以实现如下特性。

- 近乎无限的存储容量。
- 超高的读写性能。
- 数据的高可靠性：节点磁盘损毁不会导致丢数据的问题。

- 服务的高可用性：节点宕机不会影响集群对外提供服务。

那么，这些原生的分布式存储系统又是如何实现这些特性的呢？

实际上，这些分布式存储系统也有很多共通之处。这一点我们也可以理解，除了存储的数据结构，以及提供的查询服务不一样之外，这些分布式存储系统，所面临的很多问题都是一样的，因此它们的实现方法基本相同这一点也是可以理解的。

对象存储的查询服务和数据结构都非常简单，是最简单的原生分布式存储系统。本章就来一起研究一下对象存储——一种最简单的原生分布式存储，并通过对象存储来了解分布式存储系统的一些共性。掌握了这些共性之后，再想学习和了解其他的分布式存储系统和数据库，就会容易得多。

14.1 对象存储数据是如何保存大文件的

对象存储对外提供的服务，其实就是一个近乎无限容量的大文件 KV 存储。而对于分布式文件系统来说，每个文件都有一个全局唯一的路径，这个路径可以理解为文件的“Key”，所以对象存储和分布式文件系统之间，并没有非常明确的界限。对象存储的内部，肯定会有很多存储节点，用于保存这些大文件，因此称其为数据节点的集群。

另外，为了管理这些数据节点及节点中的文件，我们还需要一个存储系统，用于保存集群的节点信息、文件信息及它们之间的映射关系信息。我们将这些为了管理集群而存储的数据，称为元数据（Metadata）。

对于一个存储集群来说，元数据非常重要，所以保存元数据的存储系统必须也是一个集群。由于元数据集群存储的数据量比较少，数据的变动不是很频繁，再加上客户端或网关也会缓存一部分元数据，因此元数据集群对并发的要求并不高。一般使用类似 ZooKeeper 或 etcd 这类分布式存储即可满足要求。

另外，存储集群为了对外提供访问服务，还需要一个网关集群，对外

接收外部请求，对内访问元数据和数据节点。网关集群中的各个节点不需要保存任何数据，都是无状态的节点。有些对象存储没有网关，取而代之的是客户端，但它们的功能和作用都是一样的。典型的对象存储集群架构如图 14-1 所示。

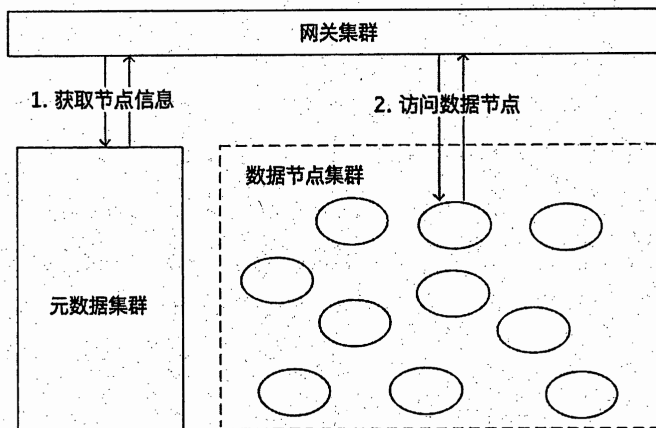


图 14-1 典型的对象存储集群架构

那么，对象存储又是如何处理对象读写请求的呢？由于对象存储处理读请求和写请求的流程是一样的，因此我们统一来说。网关收到对象的读写请求后，首先根据请求中的键，到元数据集群中找到这个键所在的数据节点，然后访问对应的数据节点读写数据，最后把结果返回给客户端。

上面比较粗略地梳理了一下对象存储处理读写请求的大致流程，实际上其中还包含了很多细节，我们暂时没有展开来讲，就是为了让大家在整体上对对象存储，以至于分布式存储系统，有一个清晰的认知。

虽然图 14-1 所示的是对象存储集群的结构，但是把图中的名词改一改，也完全可以套用到绝大多数分布式文件系统和数据库上，比如 HDFS。

14.2 如何拆分和保存大文件对象

接下来就来讨论对象存储到底是如何保存大文件对象的。一般来说，

对象存储中保存的文件都是图片、视频之类的大文件。在对象存储中，每个大文件都会被拆成多个大小相等的块（Block），拆分的方法其实很简单，就是把文件从头到尾按照固定的块大小，切成若干个小块，最后一个块的长度很有可能不足一个块的大小，也按一个块来处理。块的大小一般配置为几十 KB 到几 MB 左右。

把大对象文件拆分成块的目的有两个，具体如下。

第一是为了提升读写性能，这些块可以分散到不同的数据节点上，这样就可以并行读写了。

第二是把文件分成大小相等的块，便于更好地维护和管理。

对象被拆分成块之后，会过于碎片化，如果直接管理这些块，则会导致元数据的数据量变得非常大，同时也没有必要管理到这么细的粒度。所以，一般都会把块再聚合起来，放到块的容器里面。这里所说的“容器”就是存放一组块的逻辑单元。容器这个名词，并没有统一的叫法，比如，Ceph 将其称为 Data Placement，大家理解其含义就行。容器内块的数量大多是固定的，所以容器的大小也是固定的。

至此我们可以看出，容器的概念，比较类似于之前讲 MySQL 和 Redis 时提到的“分片”的概念，都是复制、迁移数据的基本单位。每个容器都会有 N 个副本，这些副本的数据都是一样的。其中有一个主副本，其他的都是从副本，主副本负责数据的读写，从副本则是从主副本中复制数据，来保证主从数据的一致性。

不过，这里也有一点与之前所讲的不一样的地方，对象存储一般是不会记录类似于 MySQL 的 Binlog 这样的日志。主从复制的时候，复制的不是日志，而是整块的数据。这样设计的原因有两个，具体如下。

第一个原因是基于性能的考虑。众所周知，操作日志中实际上就包含着数据。在更新数据的时候，先记录操作日志，再更新存储引擎中的数据，相当于是在磁盘上串行写了两次数据。对于像数据库这种每次更新的数据都很少的存储系统，这个开销是可以接受的。但是对于对象存储来说，其

每次写入的块都很大，两次磁盘 I/O 的开销就有些不值得了。

第二个原因是它的存储结构比较简单，即使没有日志，只要按照顺序，整块地复制数据，也仍然可以保证主从副本的数据一致性。

上面所说的对象（也就是文件）、块和容器，都是逻辑层面的概念。当数据落实到副本上之后，这些副本就是真实的物理存在了。这些副本会被分配到数据节点上保存起来。这里的数据节点就是运行在服务器上的服务进程，主要负责在本地磁盘上保存副本的数据。图 14-2 展示了大文件在对象存储中拆分和存储的逻辑。

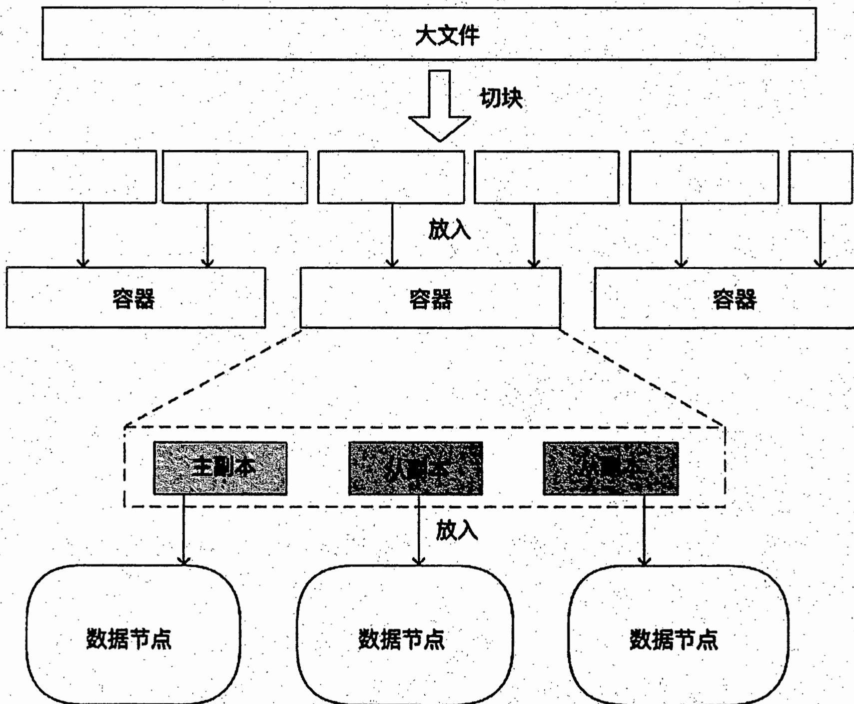


图 14-2　对象存储中数据拆分和存储的逻辑

了解了对象拆分和存储在数据节点上的逻辑之后，我们再来回顾一下数据访问的流程。当我们请求访问某个键的时候，网关首先会从元数据中查找该键的元数据。然后根据元数据中记录的对象长度，计算出对象有多

少个块。接下来的流程就可以分块并行处理了。对于每个块，我们还需要再去元数据中，找到它所在的容器。

前文提到过，容器就是分片，把块映射到容器中的方法就是第 11 章中讲到的几种分片算法。不同的系统选择实现的方式也不一样，有用哈希分片的，也有用查表法把对应的关系保存在元数据中的。找到容器之后，再到元数据中查找容器的 N 个副本所在的数据节点，然后，网关直接访问对应的数据节点就可以读写数据了。

14.3 小结

对象存储是最简单的分布式存储系统，主要由数据节点集群、元数据集群和网关集群（或者客户端）三部分构成。数据节点集群负责保存对象数据，元数据集群负责保存集群的元数据，网关集群或客户端对外提供简单的访问 API，对内则是访问元数据和数据节点以读写数据。

为了便于维护和管理，大的对象会拆分为若干个固定大小的块，块又被封装到容器（也就是分片）中，每个容器有一主 N 从多个副本，这些副本又会分散到集群的数据节点上保存。

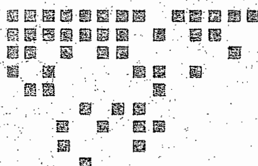
对象存储虽然简单，但是它具备一个分布式存储系统的全部特征。所有分布式存储系统共通的特性，对象存储全都具备，比如，数据如何分片，如何通过多副本保证数据的可靠性，如何在多个副本间复制数据，如何确保数据的一致性，等等。

希望大家在学习本章的时候，不仅要学会对象存储，还要对比分析一下，对象存储与其他分布式存储系统（比如 MySQL 集群、HDFS、Elasticsearch 等）之间有什么共通之处，又有哪些差异。想通了这些问题之后，大家对分布式存储系统的认知，一定会上升到一个全新的高度。这之后再去看一些之前不了解的存储系统，就会简单很多。

14.4 思考题

本章中提到过，对象存储并不是基于日志来进行主从复制的。假设我们的对象存储是一主二从三个副本，采用半同步的方式复制数据，也就是主副本和任意一个从副本更新成功后，就向客户端返回成功响应。主副本所在的节点宕机之后，这两个从副本中，至少有一个副本上的数据与宕机的主副本是一样的，我们需要找到这个副本作为新的主副本，才能保证宕机后数据不丢失。

请思考，没有日志的情况下，如果这两个从副本上的数据不一样，应该如何确定哪个从副本上面的数据与主副本是一样新的呢？



Chapter 15 第 15 章

海量数据的存储与查询

15.1 如何存储前端埋点之类的海量数据

对于大部分互联网公司来说，数据量最大的几类数据是：前端埋点数据、监控数据和日志数据。其中，“前端埋点数据”也称为“点击流”，是指在 App、小程序和 Web 页面上的埋点数据，这些埋点数据主要用于记录用户的行为，比如，打开了哪个页面，点击了哪个按钮，在哪个商品上停留了多久等信息。

当然，我们不用因此而担心自己的隐私问题，互联网公司记录用户的行为数据，并不是为了监控用户，而是为了从统计学上分析群体用户的行为，从而改进产品和运营。比如，浏览某件商品的人很多，在其上停留的时间也很长，最后下单购买的人却很少，那么采销人员就要考虑，这件商品的定价是不是太高了。

除了点击流数据之外，监控和日志也是大家常用的数据，这里就不再赘述了。

上述三种数据都是真正的“海量”数据，相比于订单、商品之类的业