第 12 章 MySQL 可扩展设计的基本原则

前言：

随着信息量的飞速增加，硬件设备的发展已经慢慢的无法跟上应用系统对处理能力的要求了。此时，我们如何来解决系统对性能的要求？只有一个办法，那就是通过改造系统的架构体系，提升系统的扩展能力，通过组合多个低处理能力的硬件设备来达到一个高处理能力的系统，也就是说，我们必须进行可扩展设计。可扩展设计是一个非常复杂的系统工程，所涉及的各个方面非常的广泛，技术也较为复杂，可能还会带来很多其他方面的问题。但不管我们如何设计，不管遇到哪些问题，有些原则我们还是必须确保的。本章就将可扩展设计过程中需要确保的原则做一个简单的介绍。

12.1 什么是可扩展性

在讨论可扩展性之前，可能很多朋有会问：常听人说起某某网站某某系统在可扩展性方面设计的如何如何好，架构如何如何出色，到底什么是扩展？怎样算是可扩展？什么又是可扩展性呢？其实也就是大家常听到的 Scale，Scalable 和 Scalability 这三个词。

从数据库的角度来说，Scale（扩展）就是让我们的数据库能够提供更强的服务能力，更强的处理能力。而 Scalable（可扩展）则是表明数据库系统在通过相应升级（包括增加单机处理能力或者增加服务器数量）之后能够达到提供更强处理能力。在理论能上来说，任何数据库系统都是 Scalable 的，只不过是所需要的实现方式不一样而已。最后，Scalability（扩展性）则是指一个数据库系统通过相应的升级之后所带来处理能力提升的难以程度。虽然理论上任何系统都可以通过相应的升级来达到处理能力的提升，但是不同的系统提升相同的处理能力所需要的升级成本（资金和人力）是不一样的，这也就是我们所说的各个数据库应用系统的 Scalability 存在很大的差异。

在这里，我所说的不同数据库应用系统并不是指数据库软件本身的不同（虽然数据库软件不同也会存在 Scalability 的差异），而是指相同数据库软件的不同应用架构设计，这也正是本章以及后面几张将会所重点分析的内容。

首先，我们需要清楚一个数据库据系统的扩展性实际上是主要体现在两个方面，一个是横向扩展，另一个则是纵向扩展，也就是我们常说的 Scale Out 和 Scale Up。

Scale Out 就是指横向的扩展，向外扩展，也就是通过增加处理节点的方式来提高整体处理能力，说的更实际一点就是通过增加机器来增加整体的处理能力。

Scale Up 则是指纵向的扩展，向上扩展，也就是通过增加当前处理节点的处理能力来提高整体的处理能力，说白了就是通过升级现有服务器的配置，如增加内存，增加 CPU，增

加存储系统的硬件配置，或者是直接更换为处理能力更强的服务器和更为高端的存储系统。

通过比较两种 Scale 方式，我们很容易看出各自的优缺点。

- Scale Out 优点：
  1. 成本低，很容易通过价格低廉的 PC Server 搭建出一个处理能力非常强大的计算集群；
  2. 不太容易遇到瓶颈，因为很容易通过添加主机来增加处理能力；
  3. 单个节点故障对系统整体影响较小；也存在缺点，更多的计算节点，大部分时候都是服务器主机，这自然会带来整个系统维护复杂性的提高，在某些方面肯定会增加维护成本，而且对应用系统的架构要求也会比 Scale Up 更高，需要集群管理软件的配合。
- Scale Out 缺点：
  1. 处理节点多，造成系统架构整体复杂度提高，应用程序复杂度提高；
  2. 集群维护难以程度更高，维护成本更大；
- Scale Up 优点：
  1. 处理节点少，维护相对简单；
  2. 所有数据都集中在一起，应用系统架构简单，开发相对容易；
- Scale Up 缺点
  1. 高端设备成本高，且竞争少，容易受到厂家限制；
  2. 受到硬件设备发展速度限制，单台主机的处理能力总是有极限的，容易遇到最终无法解决的性能瓶颈；
  3. 设备和数据集中，发生故障后的影响较大；

从短期来看，Scale Up 会有更大的优势，因为可以简化运维成本，简化系统架构和应用系统的开发，对技术方面的要求要会更简单一些。

但是，从长远影响来看，Scale Out 会有更大的优势，而且也是系统达到一个规模之后的必然趋势。因为不管怎样，单台机器的处理能力总是会受到硬件技术的限制，而硬件技术的发展速度总是有限的，很多时候很难跟得上业务发展的速度。而且越是高处理能力的高端设备，其性价比总是会越差。所以通过多台廉价的 PC Server 构建高处理能力的分布式集群，总是会成为各个公司节约成本，提高整体处理能力的一个目标。虽然在实现这个目标的时候可能会遇到各种各样的技术问题，但总是值得去研究实践的。

后面的内容，我们将重点针对 Scale Out 方面来进行分析设计。要能够很好的 Scale Out，势必需要进行分布式的系统设计。对于数据库，要想较好的 Scale Out，我们只有两个方向，一个是通过数据的不断复制来实现很多个完全一样的数据源来进行扩展，另一个就是通过将一个集中的数据源切分成很多个数据源来实现扩展。

下面我们先看看在设计一个具有很好的 Scalability 的数据库应用系统架构方面，需要遵循一些什么样的原则。

12.2 事务相关性最小化原则

搭建分布式数据库集群的时候，很多人都会比较关心事务的问题。毕竟事务是数据库中非常核心的一个功能。

在传统的集中式数据库架构中，事务的问题非常好解决，可以完全依赖数据库本身非常成熟的事务机制来保证。但是一旦我们的数据库作为分布式的架构之后，很多原来在单一数据库中所完成的事务现在可能需要跨多个数据库主机，这样原来单机事务可能就需要引入分布式事务的概念。

但是大家肯定也有一些了解，分布式事务本身就是一个非常复杂的机制，不管是商业的大型数据库系统还是各开源数据库系统，虽然大多数数据库厂家基本上都实现了这个功能，但或多或少都存在各种各样的限制。而且也存在一些 Bug，可能造成某些事务并不能很好的保证，或者是不能顺利的完成。

这时候，我们可能就需要寻求其他的替代方案来解决这个问题，毕竟事务是不可忽视的，不关我们如何去实现，总是需要实现的。

就目前来说，主要存在的一些解决方案主要有以下三种：

第一、进行 Scale Out 设计的时候合理设计切分规则，尽可能保证事务所需数据在同一个 MySQL Server 上，避免分布式事务。

如果可以在设计数据切分规则的时候就做到所有事务都能够在单个 MySQL Server 上面完成，我们的业务需求就可以比较容易的实现，应用程序就可以做到通过最少的调整来满足架构的改动，使整体成本大大减少。毕竟，数据库架构改造并不仅仅只是 DBA 的事情，还需要很多外围的配合与支持。即使是在设计一个全新系统的时候，我们同样要考虑到各个环境各项工作的整体投入，既要考虑数据库本身的成本投入，同时也要考虑到相应的开发代价。如果各环节之间出现“利益”冲突，那我们就必须要作出一个基于后续扩展以及总体成本的权衡，寻找出一个最适合当前阶段平衡点。

不过，即使我们的切分规则设计的再高明，也很难让所有的事务所需的数据都在同一个 MySQL Server 上。所以，虽然这种解决方案所需要付出的成本最小，但大多数时候也只能兼顾到一些大部分的核心事务，也不是一个很完美的解决方案。

第二、大事务切分成多个小事务，数据库保证各个小事务的完整性，应用控制各个小事务之间的整体事务完整性。

和上一个方案相比，这个方案所带来的应用改造就会更多，对应用的要求也会更为苛刻。应用不仅需要分拆原来的很多大事务，同时还需要保证各个小事务的之间的完整性。也就是说，应用程序自己需要具有一定的事务能力，这无疑会增加应用程序的技术难度。

但是，这个方案也有不少自己的优势。首先我们的数据的切分规则就会更为简单，很难遇到限制。而且更简单，就意味着维护成本更低。其次，没有数据切分规则的太多限制，数据库方面的可扩展性也会更高，不会受到太多的约束，当出现性能瓶颈的时候可以快速进行进一步拆分现有数据库。最后，数据库做到离实际业务逻辑更远，对后续架构扩展也就更为有利。

第三、结合上述两种解决方案，整合各自的优势，避免各自的弊端。

前面两种解决方案都存在各自的优缺点，而且基本上都是相互对立的，我们完全可以利用两者各自的优势，调整两个方案的设计原则，在整个架构设计中做一个平衡。比如我们可以在保证部分核心事务所需数据在同一个 MySQL Server 上，而其他并不是特别重要的事务，则通过分拆成小事务和应用系统结合来保证。而且，对于有些并不是特别重要的事务，我们也可以通过深入分析，看是否不可避免一定需要使用事务。

通过这样相互平衡设计的原则，我们既可以避免应用程序需要处理太多的小事务来保证其整体的完整性，同时也能够避免拆分规则太多复杂而带来后期维护难度的增加及扩展性受阻的情况。

当然，并不是所有的应用场景都非要结合以上两种方案来解决。比如对于那些对事务要求并不是特别严格，或者事务本身就非常简单的应用，就完全可以通过稍加设计的拆分规则就可满足相关要求，我们完全可以仅仅使用第一中方案，就可以避免还需要应用程序来维护某些小事务的整体完整性的支持。这在很大程度上面可以降低应用程序的复杂度。

而对于那些事务关系非常复杂，数据之间的关联度非常高的应用，我们也就没有必要为了保持事务数据能够集中而努力设计，因为不管我们如何努力，都很难满足要求，大都是遇到顾此失彼的情景。对于这种情况，我们还不如让数据库方面尽可能保持简洁，而让应用程序做出一些牺牲。

在当前很多大型的互联网应用中，不论是上面哪一种解决方案的使用案例都有，如大家所熟知的 Ebay，在很大程度上就是第三种结合的方案。在结合过程中以第二种方案为主，第一种方案为辅。选择这样的架构，除了他们应用场景的需求之外，其较强的技术实力也为开发足够强壮的应用系统提供了保证。又如某国内大型的 BBS 应用系统（不便公开其真实名称），其事务关联性并不是特别的复杂，各个功能模块之间的数据关联性并不是特别的高，就是完全采用第一种解决方案，完全通过合理设计数据拆分的规则来避免事务的数据源跨多个 MySQL Server。

最后，我们还需要明白一个观点，那就是事务并不是越多越好，而是越少越好越小越好 。不论我们使用何种解决方案，那就是在我们设计应用程序的时候，都需要尽可能做到让数据的事务相关性更小，甚至是不需要事务相关性。当然，这只是相对的，也肯定只有部分数据能够做到。但可能就是某部分数据做到了无事务相关性之后，系统整体复杂度就会降低很大一个层次，应用程序和数据库系统两方面都可能少付出很多的代价。

12.3 数据一致性原则

不论是 Scale Up 还是 Scale Out，不论我们如何设计自己的架构，保证数据的最终一致性都是绝对不能违背的原则，保证这个原则的重要性我想各位读者肯定也都是非常明白清楚的。

而且，数据一致性的保证就像事务完整性一样，在我们对系统进行 Scale Out 设计的时候，也可能会遇到一些问题。当然，如果是 Scale Up，可能就很少会遇到这类麻烦了。当然，在很多人眼中，数据的一致性在某种程度上面也是属于事务完整性的范畴。不过这里为了突出其重要性和相关特性，我还是将他单独提出来分析。

那我们又如何在 Scale Out 的同时又较好的保证数据一致性呢？很多时候这个问题和保证事务完整性一样让我们头疼，也同样受到了很多架构师的关注。经过很多人的实践，大家最后总结出了 BASE 模型。即：基本可用，柔性状态，基本一致和最终一致。这几个词看着挺复杂挺深奥，其实大家可以简单的理解为非实时的一致性原则。

也就是说，应用系统通过相关的技术实现，让整个系统在满足用户使用的基础上，允许数据短时间内处于非实时状态，而通过后续技术来保证数据在最终保证处于一致状态。这个理论模型说起来确实听简单，但实际实现过程中我们也会遇到不少难题。

首先，第一个问题就是我们需要让所有数据都是非实时一致吗？我想大多数读者朋友肯定是投反对票的。那如果不是所有的数据都是非实时一致，那我们又该如何来确定哪些数据需要实时一致哪些数据又只需要非实时的最终一致呢？其实这基本可以说是一个各模块业务优先级的划分，对于优先级高的自然是规属于保证数据实时一致性的阵营，而优先级略低的应用，则可以考虑划分到允许短时间端内不一致而最终一致的阵营。这是一个非常棘手的问题。我们不能随便拍脑袋就决定，而是需要通过非常详细的分析和仔细的评估才能作出决定。因为不是所有数据都可以出现在系统能不短时间段内不一致状态，也不是所有数据都可以通过后期处理的使数据最终达到一致的状态，所以之少这两类数据就是需要实时一致的。而如何区分出这两类数据，就必须经过详细的分析业务场景商业需求后进行充分的评估才能得出结论。

其次，如何让系统中的不一致数据达到最终一致？一般来说，我们必须将这类数据所设计到的业务模块和需要实时一致数据的业务模块明确的划分开来。然后通过相关的异步机制技术，利用相应的后台进程，通过系统中的数据，日志等信息将当前并不一致的数据进行进一步处理，使最终数据处于完全一致状态。对于不同的模块，使用不同的后台进程，既可以避免数据出现紊乱，也可以并发执行，提高处理效率。如对用户的消息通知之类的信息，就没有必要做到严格的实时一致性，只需要现记录下需要处理的消息，然后让后台的处理进程依次处理，避免造成前台业务的拥塞。

最后，避免实时一致与最终一致两类数据的前台在线交互。由于两类数据状态的不一致性，很可能会导致两类数据在交互过程中出现紊乱，应该尽量让所有非实时一致的数据和实时一致数据在应用程序中得到有效的隔离。甚至在有些特别的场景下，记录在不同的 MySQL

Server 中来进行物理隔离都是有必要的。

12.4 高可用及数据安全原则

除了上面两个原则之外，我还想提一下系统高可用及数据安这两方面。经过我们的 Scale Out 设计之后，系统整体可扩展性确实是会得到很大的提高，整体性能自然也很容易得到较大的改善。但是，系统整体的可用性维护方面却是变得比以前更为困难。因为系统整体架构复杂了，不论是应用程序还是数据库环境方面都会比原来更为庞大，更为复杂。这样所带来的最直接影响就是维护难度更大，系统监控更难。

如果这样的设计改造所带来的结果是我们系统经常性的 Crash,经常性的出现 Down 机事故，我想大家肯定是无法接受的，所以我们必须通过各种技术手段来保证系统的可用性不会降低，甚至在整体上有所提高。

所以，这里很自然就引出了我们在进行 Scale Out 设计过程中另一个原则，也就是高可用性的原则。不论如何调整设计系统的架构，系统的整体可用性不能被降低。

其实在讨论系统可用性的同时，还会很自然的引出另外一个与之密切相关的原则，那就是数据安全原则。要想达到高可用，数据库中的数据就必须是足够安全的。这里所指的安全并不针对恶意攻击或者窃取方面来说，而是针对异常丢失。也就是说，我们必须保证在出现软/硬件故障的时候，能够保证我们的数据不会出现丢失。数据一旦丢失，根本就无可用性可言了。而且，数据本身就是数据库应用系统最核心的资源，绝对不能丢失这一原则也是毋庸置疑的。

要确保高可用及数据安全原则，最好的办法就是通过冗余机制来保证。所有软硬件设备都去除单点隐患，所有数据都存在多份拷贝。这样才能够较好的确保这一原则。在技术方面，我们可以通过 MySQL Replication，MySQL Cluster 等技术来实现。

12.5 小结

不论我们如何设计架构，不管我们的可扩展性如何变化，本章中所提到的一些原则都是非常重要的。不论是解决某些问题的原则，还是保证性的原则，不论是保证可用性的原则，还是保证数据安全的原则，我们都应该在设计中时时刻刻都关注，谨记。

MySQL 数据库之所以在互联网行业如此火爆，除了其开源的特性，使用简单之外，还有一个非常重要的因素就是在扩展性方面有较大的优势。其不同存储引擎各自所拥有的特性可以应对各种不同的应用场景。其 Replication 以及 Cluster 等特性更是提升扩展性非常有效的手段。

第 13 章 可扩展性设计之 MySQL Replication

前言：

MySQL Replication 是 MySQL 非常有特色的一个功能，他能够将一个 MySQL Server 的 Instance 中的数据完整的复制到另外一个 MySQL Server 的 Instance 中。虽然复制过程并不是实时而是异步进行的，但是由于其高效的性能设计，延时非常之少。MySQL 的 Replication 功能在实际应用场景中被非常广泛的用于保证系统数据的安全性和系统可扩展设计中。本章将专门针对如何利用 MySQL 的 Replication 功能来提高系统的扩展性进行详细的介绍。

13.1 Replication 对可扩展性设计的意义

在互联网应用系统中，扩展最为方便的可能要数最基本的 Web 应用服务了。因为 Web 应用服务大部分情况下都是无状态的，也很少需要保存太多的数据，当然 Session 这类信息比较例外。所以，对于基本的 Web 应用服务器很容易通过简单的添加服务器并复制应用程序来做到 Scale Out。

而数据库由于其特殊的性质，就不是那么容易做到方便的 Scale Out。当然，各个数据库厂商也一直在努力希望能够做到自己的数据库软件能够像常规的应用服务器一样做到方便的 Scale Out，也确实做出了一些功能，能够基本实现像 Web 应用服务器一样的 Scalability，如很多数据库所支持的逻辑复制功能。

MySQL 数据库也为此做出了非常大的努力,MySQL Replication 功能主要就是基于这一目的所产生的。通过 MySQL 的 Replication 功能，我们可以非常方便的将一个数据库中的数据复制到很多台 MySQL 主机上面，组成一个 MySQL 集群，然后通过这个 MySQL 集群来对外提供服务。这样，每台 MySQL 主机所需要承担的负载就会大大降低，整个 MySQL 集群的处理能力也很容易得到提升。

为什么通过 MySQL 的 Replication 可以做到 Scale Out 呢？主要是因为通过 MySQL 的 Replication,可以将一台 MySQL 中的数据完整的同时复制到多台主机上面的 MySQL 数据库中，并且正常情况下这种复制的延时并不是很长。当我们各台服务器上面都有同样的数据之后，应用访问就不再只能到一台数据库主机上面读取数据了，而是访问整个 MySQL 集群中的任何一台主机上面的数据库都可以得到相同的数据。此外还有一个非常重要的因素就是 MySQL 的复制非常容易实施，也非常容易维护。这一点对于实施一个简单的分布式数据库集群是非常重要的，毕竟一个系统实施之后的工作主要就是维护了，一个维护复杂的系统肯定不是一个受欢迎的系统。

13.2 Replication 机制的实现原理

要想用好一个系统，理解其实现原理是非常重要的事情，只有理解了其实现原理，我们才能够扬长避短，合理的利用，才能够搭建出最适合我们自己应用环境的系统，才能够在系统实施之后更好的维护他。

下面我们分析一下 MySQL Replication 的实现原理。

13.2.1 Replication 线程

Mysql 的 Replication 是一个异步的复制过程，从一个 Mysql instace（我们称之为 Master）复制到另一个 Mysql instance（我们称之 Slave）。在 Master 与 Slave 之间的实现整个复制过程主要由三个线程来完成,其中两个线程(Sql 线程和 IO 线程)在 Slave 端，另外一个线程（IO 线程）在 Master 端。

要实现 MySQL 的 Replication ，首先必须打开 Master 端的 Binary Log（mysql-bin.xxxxxx）功能，否则无法实现。因为整个复制过程实际上就是 Slave 从 Master 端获取该日志然后再在自己身上完全顺序的执行日志中所记录的各种操作。打开 MySQL 的 Binary Log 可以通过在启动 MySQL Server 的过程中使用“—log-bin” 参数选项,或者在 my.cnf 配置文件中的 mysqld 参数组（[mysqld]标识后的参数部分）增加 “log-bin” 参数项。

MySQL 复制的基本过程如下：

1. Slave 上面的 IO 线程连接上 Master，并请求从指定日志文件的指定位置（或者从最开始的日志）之后的日志内容；

2. Master 接收到来自 Slave 的 IO 线程的请求后，通过负责复制的 IO 线程根据请求信息读取指定日志指定位置之后的日志信息，返回给 Slave 端的 IO 线程。返回信息中除了日志所包含的信息之外，还包括本次返回的信息在 Master 端的 Binary Log 文件的名称以及在 Binary Log 中的位置；

3. Slave 的 IO 线程接收到信息后，将接收到的日志内容依次写入到 Slave 端的 Relay Log 文件(mysql-relay-bin.xxxxxx)的最末端，并将读取到的 Master 端的 bin-log 的文件名和位置记录到 master-info 文件中，以便在下一次读取的时候能够清楚的高速 Master“我需要从某个 bin-log 的哪个位置开始往后的日志内容，请发给我”

4. Slave 的 SQL 线程检测到 Relay Log 中新增加了内容后，会马上解析该 Log 文件中的内容成为在 Master 端真实执行时候的那些可执行的 Query 语句，并在自身执行这些 Query。这样，实际上就是在 Master 端和 Slave 端执行了同样的 Query，所以两端的数据是完全一样的。

实际上，在老版本中，MySQL 的复制实现在 Slave 端并不是由 SQL 线程和 IO 线程这两个线程共同协作而完成的，而是由单独的一个线程来完成所有的工作。但是 MySQL 的工程师们很快发现，这样做存在很大的风险和性能问题，主要如下：

首先，如果通过一个单一的线程来独立实现这个工作的话，就使复制 Master 端的，Binary Log 日志，以及解析这些日志，然后再在自身执行的这个过程成为一个串行的过程，性能自然会受到较大的限制，这种架构下的 Replication 的延迟自然就比较长了。

其次，Slave 端的这个复制线程从 Master 端获取 Binary Log 过来之后，需要接着解析这些内容，还原成 Master 端所执行的原始 Query，然后在自身执行。在这个过程中，Master 端很可能又已经产生了大量的变化并生成了大量的 Binary Log 信息。如果在这个阶段 Master 端的存储系统出现了无法修复的故障，那么在这个阶段所产生的所有变更都将永远的丢失，无法再找回来。这种潜在风险在 Slave 端压力比较大的时候尤其突出，因为如果 Slave 压力比较大，解析日志以及应用这些日志所花费的时间自然就会更长一些，可能丢失的数据也就会更多。

所以，在后期的改造中，新版本的 MySQL 为了尽量减小这个风险，并提高复制的性能，将 Slave 端的复制改为两个线程来完成，也就是前面所提到的 SQL 线程和 IO 线程。最早提出这个改进方案的是 Yahoo!的一位工程师“Jeremy Zawodny”。通过这样的改造，这样既在很大程度上解决了性能问题，缩短了异步的延时时间，同时也减少了潜在的数据丢失量。

当然，即使是换成了现在这样两个线程来协作处理之后，同样也还是存在 Slave 数据延时以及数据丢失的可能性的，毕竟这个复制是异步的。只要数据的更改不是在一个事务中，这些问题都是存在的。

如果要完全避免这些问题，就只能用 MySQL 的 Cluster 来解决了。不过 MySQL 的 Cluster 知道笔者写这部分内容的时候，仍然还是一个内存数据库的解决方案，也就是需要将所有数据包括索引全部都 Load 到内存中，这样就对内存的要求就非常大的大，对于一般的大众化应用来说可实施性并不是太大。当然，在之前与 MySQL 的 CTO David 交流的时候得知，MySQL 现在正在不断改进其 Cluster 的实现，其中非常大的一个改动就是允许数据不用全部 Load 到内存中，而仅仅只是索引全部 Load 到内存中，我想信在完成该项改造之后的 MySQL Cluster 将会更加受人欢迎，可实施性也会更大。

13.2.2 复制实现级别

MySQL 的复制可以是基于一条语句（Statement Level），也可以是基于一条记录（Row level），可以在 MySQL 的配置参数中设定这个复制级别，不同复制级别的设置会影响到 Master 端的 Binary Log 记录成不同的形式。

1. Row Level：Binary Log 中会记录成每一行数据被修改的形式，然后在 Slave 端再对相同的数据进行修改。

优点：在 Row Level 模式下，Binary Log 中可以不记录执行的 sql 语句的上下文相关的信息，仅仅只需要记录那一条记录被修改了，修改成什么样了。所以 Row Level 的日志内容会非常清楚的记录下每一行数据修改的细节，非常容易理解。而且不会出现某些特定情况下的存储过程，或 function，以及 trigger 的调用和触发无法被正确复制的问题。

缺点：Row Level 下，所有的执行的语句当记录到 Binary Log 中的时候，都将以每行记录的修改来记录，这样可能会产生大量的日志内容，比如有这样一条 update 语句：UPDATE group_message SET group_id = 1 where group_id = 2，执行之后，日志中记录的不是这条 update 语句所对应的事件（MySQL 以事件的形式来记录 Binary Log 日志），而是这条语句所更新的每一条记录的变化情况，这样就记录成很多条记录被更新的很多个事件。自然，Binary Log 日志的量就会很大。尤其是当执行 ALTER TABLE 之类的语句的时候，产生的日志量是惊人的。因为 MySQL 对于 ALTER TABLE 之类的 DDL 变更语句的处理方式是重建整个表的所有数据，也就是说表中的每一条记录都需要变动，那么该表的每一条记录都会被记录到日志中。

2. Statement Level:每一条会修改数据的 Query 都会记录到 Master 的 Binary Log 中。Slave 在复制的时候 SQL 线程会解析成和原来 Master 端执行过的相同的 Query 来再次执行。

优点：Statement Level 下的优点首先就是解决了 Row Level 下的缺点，不需要记录每一行数据的变化，减少 Binary Log 日志量，节约了 IO 成本，提高了性能。因为他只需要记录在 Master 上所执行的语句的细节，以及执行语句时候的上下文的信息。

缺点：由于他是记录的执行语句，所以，为了让这些语句在 slave 端也能正确执行，那么他还必须记录每条语句在执行的时候的一些相关信息，也就是上下文信息，以保证所有语句在 slave 端杯执行的时候能够得到和在 master 端执行时候相同的结果。另外就是，由于 Mysql 现在发展比较快，很多的新功能不断的加入，使 mysql 得复制遇到了不小的挑战，自然复制的时候涉及到越复杂的内容，bug 也就越容易出现。在 statement level 下，目前已经发现的就有不少情况会造成 mysql 的复制出现问题，主要是修改数据的时候使用了某些特定的函数或者功能的时候会出现，比如：sleep()函数在有些版本中就不能真确复制，在存储过程中使用了 last_insert_id()函数，可能会使 slave 和 master 上得到不一致的 id 等等。由于 row level 是基于每一行来记录的变化，所以不会出现类似的问题。

从官方文档中看到，之前的 MySQL 一直都只有基于 Statement 的复制模式，直到 5.1.5 版本的 MySQL 才开始支持 Row Level 的复制。从 5.0 开始，MySQL 的复制已经解决了大量老版本中出现的无法正确复制的问题。但是由于存储过程的出现，给 MySQL 的复制又带来了更大的新挑战。另外，看到官方文档说，从 5.1.8 版本开始，MySQL 提供了除 Statement Level 和 Row Level 之外的第三种复制模式：Mixed Level，实际上就是前两种模式的结合。在 Mixed 模式下，MySQL 会根据执行的每一条具体的 Query 语句来区分对待记录的日志形式，也就是在 Statement 和 Row 之间选择一种。新版本中的 Statment level 还是和以前一样，仅仅记录执行的语句。而新版本的 Mysql 中队 Row Level 模式也被做了优化，并不是所有的修改都会以 Row Level 来记录，像遇到表结构变更的时候就会以 statement 模式来记录，如果 Query 语句确实就是 UPDATE 或者 DELETE 等修改数据的语句，那么还是会记录所有行的变更。

13.3 Replication 常用架构

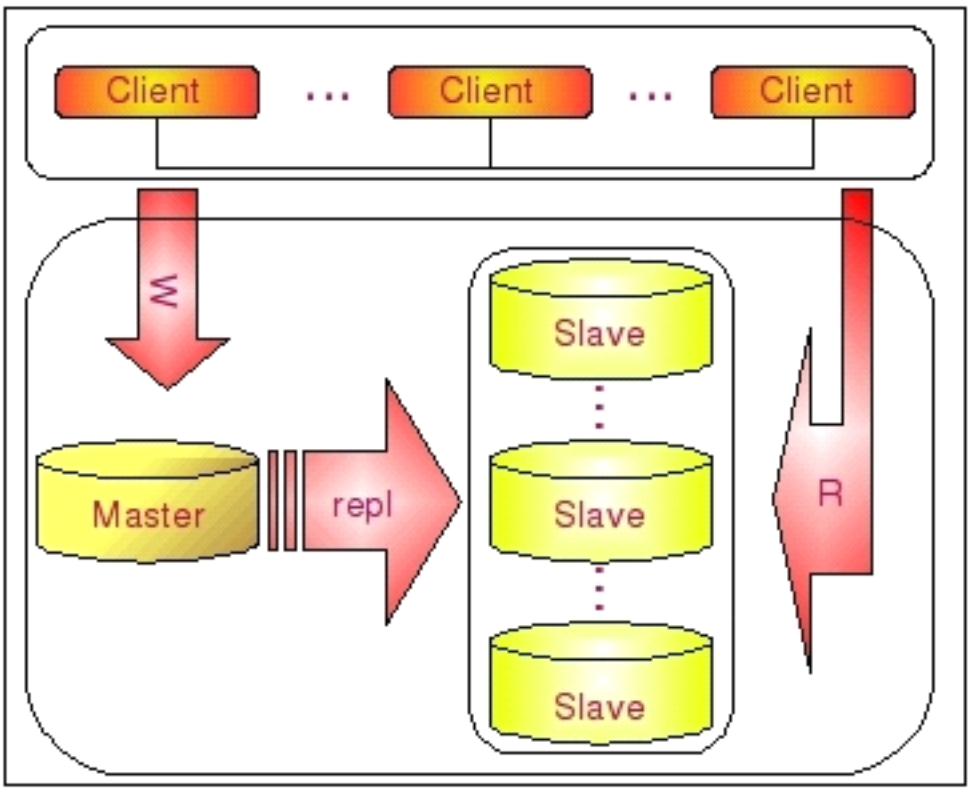
MySQL Replicaion 本身是一个比较简单的架构，就是一台 MySQL 服务器（Slave）从另一台 MySQL 服务器（Master）进行日志的复制然后再解析日志并应用到自身。一个复制环境仅仅只需要两台运行有 MySQL Server 的主机即可，甚至更为简单的时候我们可以在同一台物理服务器主机上面启动两个 mysqld instance，一个作为 Master 而另一个作为 Slave 来完成复制环境的搭建。但是在实际应用环境中，我们可以根据实际的业务需求利用 MySQL Replication 的功能自己定制搭建出其他多种更利于 Scale Out 的复制架构。如 Dual Master 架构，级联复制架构等。下面我们针对比较典型的三种复制架构进行一些相应的分析介绍。

13.3.1 常规复制架构(Master - Slaves)

在实际应用场景中，MySQL 复制 90% 以上都是一个 Master 复制到一个或者多个 Slave 的架构模式，主要用于读压力比较大的应用的数据库端廉价扩展解决方案。因为只要

Master 和 Slave 的压力不是太大（尤其是 Slave 端压力）的话，异步复制的延时一般都很少很少。尤其是自从 Slave 端的复制方式改成两个线程处理之后，更是减小了 Slave 端的延时问题。而带来的效益是，对于数据实时性要求不是特别 Critical 的应用，只需要通过廉价的 pc server 来扩展 Slave 的数量，将读压力分散到多台 Slave 的机器上面，即可通过分散单台数据库服务器的读压力来解决数据库端的读性能瓶颈，毕竟在大多数数据库应用系统中的读压力还是要比写压力大很多。这在很大程度上解决了目前很多中小型网站的数据库压力瓶颈问题，甚至有些大型网站也在使用类似方案解决数据库瓶颈。

这个架构可以通过下图比较清晰的展示：



一个 Master 复制多个 Slave 的架构实施非常简单，多个 Slave 和单个 Slave 的实施并没有实质性的区别。在 Master 端并不 Care 有多少个 Slave 连上了自己，只要有 Slave 的 IO 线程通过了连接认证，向他请求指定位置之后的 Binary Log 信息，他就会按照该 IO 线程的要求，读取自己的 Binary Log 信息，返回给 Slave 的 IO 线程。

大家应该都比较清楚，从一个 Master 节点可以复制出多个 Slave 节点，可能有人会想，那一个 Slave 节点是否可以从多个 Master 节点上面进行复制呢？至少在目前来看，MySQL 是做不到的，以后是否会支持就不清楚了。

MySQL 不支持一个 Slave 节点从多个 Master 节点来进行复制的架构，主要是为了避免冲突的问题，防止多个数据源之间的数据出现冲突，而造成最后数据的不一致性。不过听说已经有人开发了相关的 patch，让 MySQL 支持一个 Slave 节点从多个 Master 结点作为数据源来进行复制，这也正是 MySQL 开源的性质所带来的好处。

对于 Replication 的配置细节，在 MySQL 的官方文档上面已经说的非常清楚了，甚至介绍了多种实现 Slave 的配置方式，在下一节中我们也会通过一个具体的示例来演示搭建一个 Replication 环境的详细过程以及注意事项。

13.3.2 Dual Master 复制架构(Master - Master)

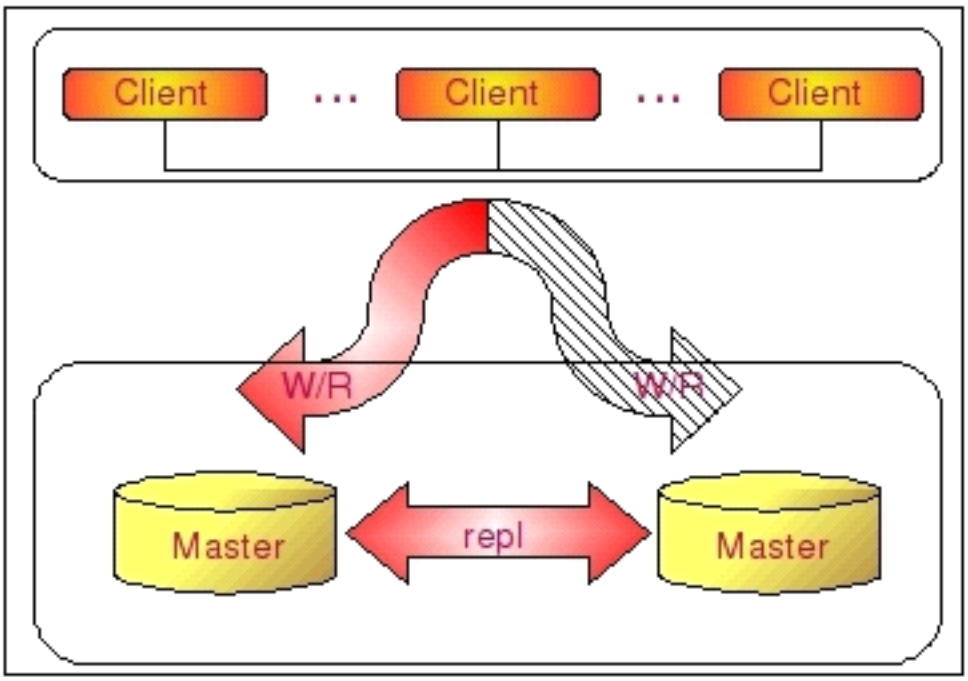
有些时候，简单的从一个 MySQL 复制到另外一个 MySQL 的基本 Replication 架构，可能还会需要在一些特定的场景下进行 Master 的切换。如在 Master 端需要进行一些特别的维护操作的时候，可能需要停 MySQL 的服务。这时候，为了尽可能减少应用系统写服务的停机时间，最佳的做法就是将我们的 Slave 节点切换成 Master 来提供写入的服务。

但是这样一来，我们原来 Master 节点的数据就会和实际的数据不一致了。当原 Master 启动可以正常提供服务的时候，由于数据的不一致，我们就不得不通过反转原 Master - Slave 关系，重新搭建 Replication 环境，并以原 Master 作为 Slave 来对外提供读的服务。重新搭建 Replication 环境会给我们带来很多额外的工作量，如果没有合适的备份，可能还会让 Replication 的搭建过程非常麻烦。

为了解决这个问题，我们可以通过搭建 Dual Master 环境来避免很多的问题。何谓 Dual Master 环境？实际上就是两个 MySQL Server 互相将对方作为自己的 Master，自己作为对方的 Slave 来进行复制。这样，任何一方所做的变更，都会通过复制应用到另外一方的数据库中。

可能有些读者朋友会有一个担心，这样搭建复制环境之后，难道不会造成两台 MySQL 之间的循环复制么？实际上 MySQL 自己早就想到了这一点，所以在 MySQL 的 Binary Log 中记录了当前 MySQL 的 server-id，而且这个参数也是我们搭建 MySQL Replication 的时候必须明确指定，而且 Master 和 Slave 的 server-id 参数值比需要不一致才能使 MySQL Replication 搭建成功。一旦有了 server-id 的值之后，MySQL 就很容易判断某个变更是从哪一个 MySQL Server 最初产生的，所以就很容易避免出现循环复制的情况。而且，如果我们不打开记录 Slave 的 Binary Log 的选项（--log-slave-update）的时候，MySQL 根本就不会记录复制过程中的变更到 Binary Log 中，就更不用担心可能会出现循环复制的情形了。

下如将更清晰的展示 Dual Master 复制架构组成：



通过 Dual Master 复制架构，我们不仅能够避免因为正常的常规维护操作需要的停机所带来的重新搭建 Replication 环境的操作，因为我们任何一端都记录了自己当前复制到对方的什么位置了，当系统起来之后，就会自动开始从之前的位置重新开始复制，而不需要人为去进行任何干预，大大节省了维护成本。

不仅仅如此，Dual Master 复制架构和一些第三方的 HA 管理软件结合，还可以在我们当前正在使用的 Master 出现异常无法提供服务之后，非常迅速的自动切换另外一端来提供相应的服务，减少异常情况下带来的停机时间，并且完全不需要人工干预。

当然，我们搭建成一个 Dual Master 环境，并不是为了让两端都提供写的服务。在正常情况下，我们都只会将其中一端开启写服务，另外一端仅仅只是提供读服务，或者完全不提供任何服务，仅仅只是作为一个备用的机器存在。为什么我们一般都只开启其中的一端来提供写服务呢？主要还是为了避免数据的冲突，防止造成数据的不一致性。因为即使在两边执行的修改有先后顺序，但由于 Replication 是异步的实现机制，同样会导致即使晚做的修改也可能会被早做的修改所覆盖，就像如下情形：

时间点	MySQL A	MySQL B
1	更新 x 表 y 记录为 10	
2		更新 x 表 y 记录为 20
3		获取到 A 日志并应用，更新 x 表的 y 记录为 10（不符合期望）
4	获取 B 日志更新 x 表 y 记录为 20（符合期望）	

这中情形下，不仅在 B 库上面的数据不是用户所期望的结果，A 和 B 两边的数据也出现了不一致。

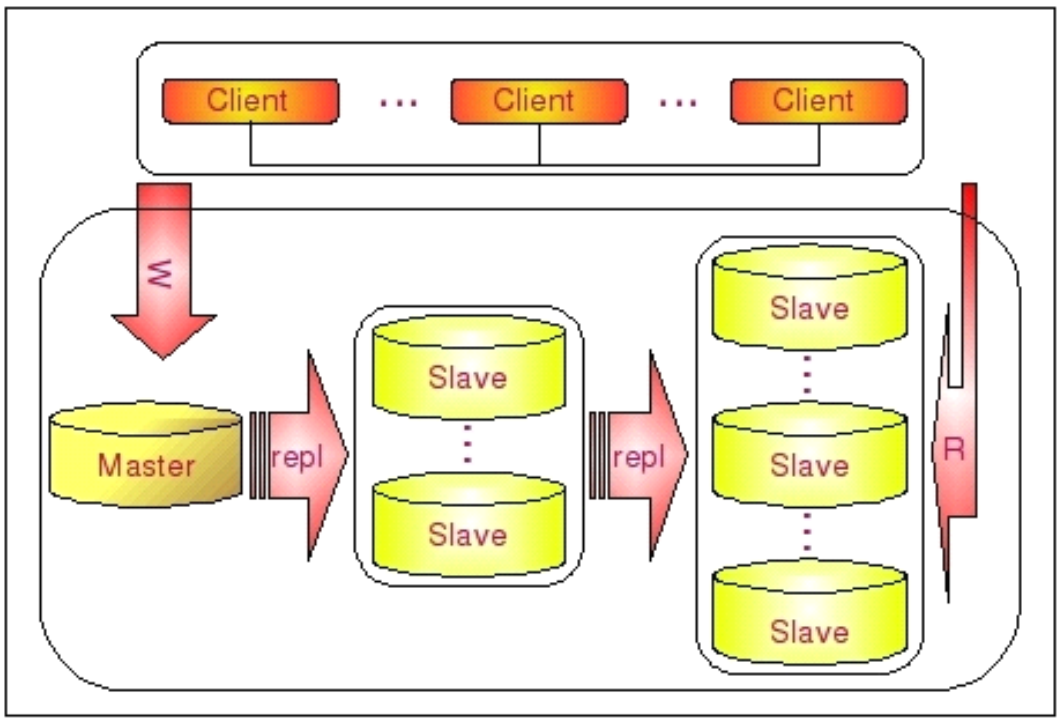
当然，我们也可以通过特殊的约定，让某些表的写操作全部在一端，而另外一些表的写操作全部在另外一端，保证两端不会操作相同的表，这样就能避免上面问题的发生了。

13.3.3 级联复制架构(Master - Slaves - Slaves ...)

在有些应用场景中，可能读写压力差别比较大，读压力特别的大，一个 Master 可能需要上 10 台甚至更多的 Slave 才能够支撑注读的压力。这时候，Master 就会比较吃力了，因为仅仅连上来的 Slave IO 线程就比较多了,这样写的压力稍微大一点的时候,Master 端因为复制就会消耗较多的资源，很容易造成复制的延时。

遇到这种情况如何解决呢？这时候我们就可以利用 MySQL 可以在 Slave 端记录复制所产生变更的 Binary Log 信息的功能，也就是打开 —log-slave-update 选项。然后，通过二级（或者是更多级别）复制来减少 Master 端因为复制所带来的压力。也就是说，我们首先通过少数几台 MySQL 从 Master 来进行复制，这几台机器我们姑且称之为第一级 Slave 集群，然后其他的 Slave 再从第一级 Slave 集群来进行复制。从第一级 Slave 进行复制的 Slave，我称之为第二级 Slave 集群。如果有需要，我们可以继续往下增加更多层次的复制。这样，我们很容易就控制了每一台 MySQL 上面所附属 Slave 的数量。这种架构我称之为 Master - Slaves - Slaves 架构

这种多层级联复制的架构，很容易就解决了 Master 端因为附属 Slave 太多而成为瓶颈的风险。下图展示了多层级联复制的 Replication 架构。



当然，如果条件允许，我更倾向于建议大家通过拆分成多个 Replication 集群来解决

上述瓶颈问题。毕竟 Slave 并没有减少写的量，所有 Slave 实际上仍然还是应用了所有的数据变更操作，没有减少任何写 IO。相反，Slave 越多，整个集群的写 IO 总量也就会越多，我们没有非常明显的感觉，仅仅只是因为分散到了多台机器上面，所以不是很容易表现出来。

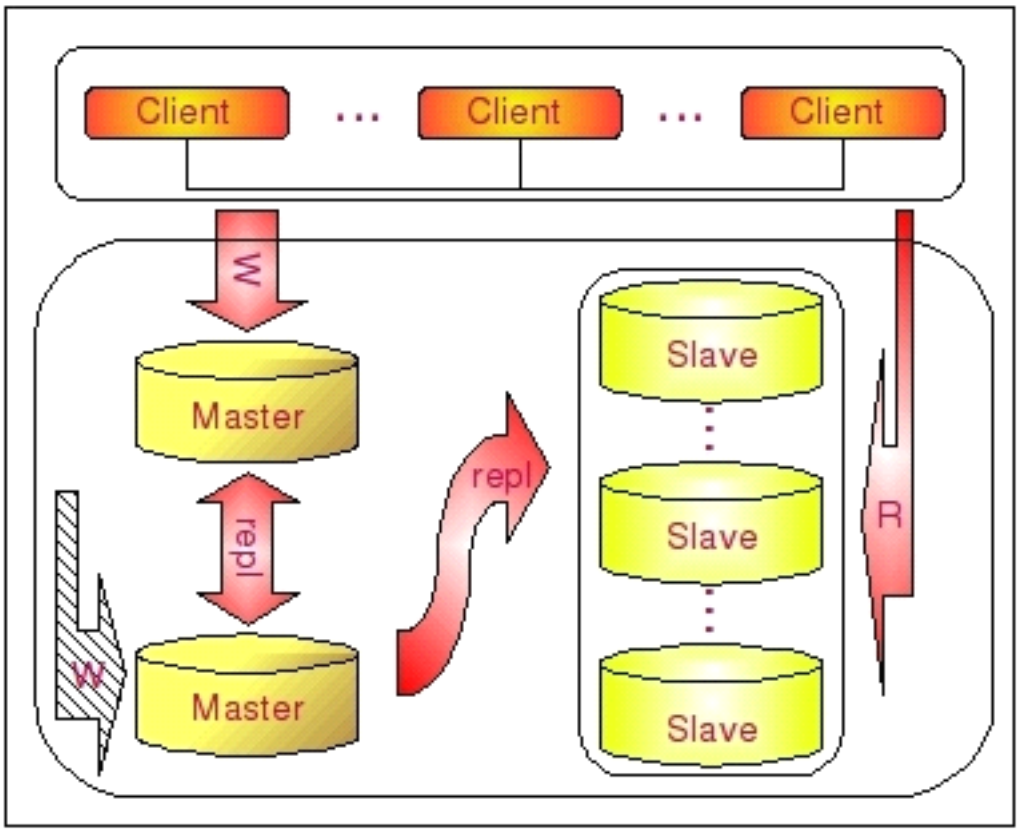
此外，增加复制的级联层次，同一个变更传到最底层的 Slave 所需要经过的 MySQL 也会更多，同样可能造成延时较长的风险。

而如果我们通过分拆集群的方式来解决的话，可能就会要好很多了，当然，分拆集群也需要更复杂的技术和更复杂的应用系统架构。

13.3.4 Dual Master 与级联复制结合架构(Master - Master - Slaves)

级联复制在一定程度上面确实解决了 Master 因为所附属的 Slave 过多而成为瓶颈的问题，但是他并不能解决人工维护和出现异常需要切换后可能存在重新搭建 Replication 的问题。这样就很自然的引申出了 Dual Master 与级联复制结合的 Replication 架构，我称之为 Master - Master - Slaves 架构

和 Master - Slaves - Slaves 架构相比，区别仅仅只是将第一级 Slave 集群换成了一台单独的 Master，作为备用 Master，然后再从这个备用的 Master 进行复制到一个 Slave 集群。下面的图片更清晰的展示了这个架构的组成：



这种 Dual Master 与级联复制结合的架构，最大的好处就是既可以避免主 Master 的写入操作不会受到 Slave 集群的复制所带来的影响，同时主 Master 需要切换的时候也基本上不会出现重搭 Replication 的情况。但是，这个架构也有一个弊端，那就是备用的 Master 有可能成为瓶颈，因为如果后面的 Slave 集群比较大的话，备用 Master 可能会因为过多的 Slave IO 线程请求而成为瓶颈。当然，该备用 Master 不提供任何的读服务的时候，瓶颈出现的可能性并不是特别高，如果出现瓶颈，也可以在备用 Master 后面再次进行级联复制，架设多层 Slave 集群。当然，级联复制的级别越多，Slave 集群可能出现的数据延时也会更为明显，所以考虑使用多层级联复制之前，也需要评估数据延时对应用系统的影响。

13.4 Replication 搭建实现

MySQL Replication 环境的搭建实现比较简单，总的来说其实就是四步，第一步是做好 Master 端的准备工作。第二步是取得 Master 端数据的“快照”备份。第三步则是在 Slave 端恢复 Master 的备份“快照”。第四步就是在 Slave 端设置 Master 相关配置，然后启动复制。在这一节中，并不是列举一个搭建 Replication 环境的详细过程，因为这在 MySQL 官方操作手册中已经有较为详细的描述了，我主要是针对搭建环境中几个主要的操作步骤中可以使用的各种实现方法的介绍，下面我们针对这四步操作及需要注意的地方进行一个简单的分析。

1. Master 端准备工作

在搭建 Replication 环境之前，首先要保证 Master 端 MySQL 记录 Binary Log 的选项打开，因为 MySQL Replication 就是通过 Binary Log 来实现的。让 Master 端 MySQL 记录 Binary Log 可以在启动 MySQL Server 的时候使用 —log-bin 选项或者在 MySQL 的配置文件 my.cnf 中配置 log-bin[=path for binary log]参数选项。

在开启了记录 Binary Log 功能之后，我们还需要准备一个用于复制的 MySQL 用户。可以通过给一个现有帐户授予复制相关的权限，也可以创建一个全新的专用于复制的帐户。当然，我还是建议用一个专用于复制的帐户来进行复制。在之前“MySQL 安全管理”部分也已经介绍过了，通过特定的帐户处理特定一类的工作，不论是在安全策略方面更有利，对于维护来说也有更大的便利性。实现 MySQL Replication 仅仅只需要“REPLICATION SLAVE”权限即可。可以通过如下方式来创建这个用户：

```
root@localhost : mysql 04:16:18> CREATE USER 'repl'@'192.168.0.2'
    -> IDENTIFIED BY 'password';
Query OK, 0 rows affected (0.00 sec)

root@localhost : mysql 04:16:34> GRANT REPLICATION SLAVE ON *.*
    -> TO 'repl'@'192.168.0.2';
Query OK, 0 rows affected (0.00 sec)
```

这里首先通过 CREATE USER 命令创建了一个仅仅具有最基本权限的用户 repl，然后再通过 GRANT 命令授予该用户 REPLICATION SLAVE 的权限。当然，我们也可以仅仅执行上面的第二条命令，即可创建出我们所需的用户，这已经在“MySQL 安全管理”部分介绍过了。

2. 获取 Master 端的备份“快照”

这里所说的 Master 端的备份“快照”，并不是特指通过类似 LVM 之类的软件所做的 snapshot，而是所有数据均是基于某一特定时刻的，数据完整性和一致性都可以得到保证的备份集。同时还需要取得该备份集时刻所对应的 Master 端 Binary Log 的准确 Log Position，因为在后面配置 Slave 的时候会用到。

一般来说，我们可以通过如下集中办法获得一个具有一致性和完整性的备份集以及所对应的 Log Position：

- 通过数据库全库冷备份

  对于可以停机的数据库，我们可以通过关闭 Master 端 MySQL，然后通过 copy 所有数据文件和日志文件到需要搭建 Slave 的主机中合适的位置，这样所得到的备份集是最完整的。在做完备份之后，然后再启动 Master 端的 MySQL。

  当然，这样我们还仅仅只是得到了一个满足要求的备份集，我们还需要这个备份集所对应的日志位置才能可以。对于这样的备份集，我们有多种方法可以获取到对应的日志位置。如在 Master 刚刚启动之后，还没有应用程序连接上 Master 之前，通过执行 SHOW Master STATUS 命令从 Master 端获取到我们可以使用的 Log Position。如果我

们无法在 Master 启动之后控制应用程序的连接，那么可能在我们还没有来得及执行 SHOW Master STATUS 命令之前就已经有数据写进来了，这时候我们可以通过 mysqlbinlog 客户端程序分析 Master 最新的一个 Binary Log 来获取其第一个有效的 Log Position。当然，如果你非常清楚你所使用的 MySQL 版本每一个新的 Binary Log 第一个有效的日志位置，自然就不需要进行任何操作就可以。

◆ 通过 LVM 或者 ZFS 等具有 snapshot 功能的软件进行“热备份”

如果我们的 Master 是一个需要满足 365 * 24 * 7 服务的数据库，那么我们就无法通过进行冷备份来获取所需要的备份集。这时候，如果我们的 MySQL 运行在支持 Snapshot 功能的文件系统上面（如 ZFS），或者我们的文件系统虽然不支持 Snapshot，但是我们的文件系统运行在 LVM 上面，那么我们都可以通过相关的命令对 MySQL 的数据文件和日志文件所在的目录就做一个 Snapshot，这样就可以得到了一个基本和全库冷备差不多的备份集。

当然，为了保证我们的备份集数据能够完整且一致，我们需要在进行 Snapshot 过程中通过相关命令（FLUSH TABLES WITH READ LOCK）来锁住所有表的写操作，也包括支持事务的存储引擎中 commit 动作，这样才能真正保证该 Snapshot 的所有数据都完整一致。在做完 Snapshot 之后，我们就可以 UNLOCK TABLES 了。可能有些人会担心，如果锁住了所有的写操作，那我们的应用不是就无法提供写服务了么？确实，这是无法避免的，不过，一般来说 Snapshot 操作所需要的时间大都比较短，所以不会影响太长时间。

那 Log Position 怎么办呢？是的，通过 Snapshot 所做的备份，同样需要一个该备份所对应的 Log Position 才能满足搭建 Replication 环境的要求。不过，这种方式下，我们可以比进行冷备份更容易获取到对应的 Log Position。因为从我们锁定了所有表的写入操作开始到解锁之前，数据库不能进行任何写入操作，这个时间段之内任何时候通过执行 SHOW MASTER STATUS 明令都可以得到准确的 Log Position。

由于这种方式在实施过程中并不需要完全停掉 Master 来进行，仅仅只需要停止写入才做，所以我们也可以称之为“热备份”。

◆ 通过 mysqldump 客户端程序

如果我们的数据库不能停机进行冷备份，而且 MySQL 也没有运行在可以进行 Snapshot 的文件系统或者管理软件之上，那么我们就需要通过 mysqldump 工具来将 Master 端需要复制的数据库（或者表）的数据 dump 出来。为了让我们的备份集具有一致性和完整性，我们必须让 dump 数据的这个过程处于同一个事务中，或者锁住所有需要复制的表的写操作。要做到这一点，如果我们使用的是支持事务的存储引擎（如 Innodb），我们可以在执行 mysqldump 程序的时候通过添加 —single-transaction 选项来做到，但是如果我们的存储引擎并不支持事务，或者是需要 dump 表仅仅只有部分支持事务的时候，我们就只能先通过 FLUSH TABLES WITH READ LOCK 命令来暂停所有写入服务，然后再 dump 数据。当然，如果我们仅仅只需要 dump 一个表的数据，就不需要这么麻烦了，因为 mysqldump 程序在 dump 数据的时候实际上就是每个表通过一条 SQL 来得到数据的，所以单个表的时候总是可以保证所取数据的一致性的。

上面的操作我们还只是获得了合适的备份集，还没有该备份集所对应的 Log Position，所以还不能完全满足搭建 Slave 的要求。幸好 mysqldump 程序的开发者早就考虑到这个问题了，所以给 mysqldump 程序增加了另外一个参数选项来帮助我们获取到对应的 Log Position，这个参数选项就是 —master-data 。当我们添加这个参数选项之后，mysqldump 会在 dump 文件中产生一条 CHANGE MASTER TO 命令，命令中记录了 dump 时刻所对应的详细的 Log Position 信息。如下：

测试 dump example 数据库下的 group_message 表：

```
sky@sky:~$ mysqldump --master-data -usky -p example group_message >
group_message.sql
Enter password:
```

然后通过 grep 命令来查找一下看看：

```
sky@sky:~$ grep "CHANGE MASTER" group_message.sql
CHANGE MASTER TO MASTER_LOG_FILE='mysql-bin.000035', MASTER_LOG_POS=399;
```

连 CHANGE MASTER TO 的命令都已经给我们准备好了，还真够体贴的，呵呵。

如果我们是要一次性 dump 多个支持事务的表的时候，可能很多人会选择通过添加—single-transaction 选项来保证数据的一致性和完整性。这确实是一个不错的选择。但是，如果我们需要 dump 的数据量比较大的时候，可能会产生一个很大的事务，而且会持续较长的时间。

◆ 通过现有某一个 Slave 端进行“热备份”

如果现在已经有 Slave 从我们需要搭建 Replication 环境的 Master 上进行复制的话，那我们这个备份集就非常容易取得了。我们可以暂时性的停掉现有 Slave（如果有多台则仅仅只需要停止其中的一台），同时执行一次 FLUSH TABLES 命令来刷新所有表和索引的数据。这时候在该 Slave 上面就不会再有任何的写入操作了，我们既可以通过 copy 所有的数据文件和日志文件来做一个全备份，同时也可以通过 Snapshot（如果支持）来进行备份。当然，如果支持 Snapshot 功能，还是建议大家通过 Snapshot 来做，因为这样可以使 Slave 停止复制的时间大大缩短，减少该 Slave 的数据延时。

通过现有 Slave 来获取备份集的方式，不仅仅得到数据库备份的方式很简单，连所需要 Log Position，甚至是新 Slave 后期的配置等相关动作都可以省略掉，只需要新的 Slave 完全基于这个备份集来启动，就可以正常从 Master 进行复制了。

整个过程中我们仅仅只是在短暂时间内停止了某台现有 Slave 的复制线程，对系统的正常服务影响很小，所以这种方式也基本可以称之为“热备份”。

◆ 通过

3. Slave 端恢复备份“快照”

上面第二步我们已经获取到了所需要的备份集了，这一步所需要做的就是将上一步所得到的备份集恢复到我们的 Slave 端的 MySQL 中。

针对上面四种方法所获取的备份集的不同，在 Slave 端的恢复操作也有区别。下面就针对四种备份集的恢复做一个简单的说明：

◆ 恢复全库冷备份集

由于这个备份集是一个完整的数据库物理备份，我们仅仅只需要将这个备份集通过 FTP 或者是 SCP 之类的网络传输软件复制到 Slave 所在的主机，根据 Slave 上 my.cnf 配置文件的设置，将文件存放在相应的目录，覆盖现有所有的数据和日志等相关文件，然后再启动 Slave 端的 MySQL，就完成了整个恢复过程。

◆ 恢复对 Master 进行 Snapshot 得到的备份集

对于通过对 Master 进行 Snapshot 所得到的备份集，实际上和全库冷备的恢复方法基本一样，唯一的差别只是首先需要将该 Snapshot 通过相应的文件系统 mount 到某个目录下，然后才能进行后续的文件拷贝操作。之后的相关操作和恢复全库冷备份集基本一致，就不再累述。

◆ 恢复 mysqldump 得到的备份集

通过 mysqldump 客户端程序所得到的备份集，和前面两种备份集的恢复方式有较大的差别。因为前面两种备份集的都属于物理备份，而通过 mysqldump 客户端程序所做的备份属于逻辑备份。恢复 mysqldump 备份集的方式是通过 mysql 客户端程序来执行备份文件中的所有 SQL 语句。

使用 mysql 客户端程序在 Slave 端恢复之前，建议复制出通过 —master-data 所得到的 CHANGE MASTER TO 命令部分，然后在备份文件中注销掉该部分，再进行恢复。因为该命令并不是一个完整的 CHANGE MASTER TO 命令，如果在配置文件（my.cnf）中没有配置 MASTER_HOST,MASTER_USER,MASTER_PASSWORD 这三个参数的时候，该语句是无法有效完成的。

通过 mysql 客户端程序来恢复备份的方式如下：

```
sky@sky:~$ mysql -u sky -p -Dexample < group_message.sql
```

这样即可将之前通过 mysqldump 客户端程序所做的逻辑备份集恢复到数据库中了。

◆ 恢复通过现有 Slave 所得到的热备份

通过现有 Slave 所得到的备份集和上面第一种或者第二种备份集也差不多。如果是通过直接拷贝数据和日志文件所得到的备份集，那么就和全库冷备一样的备份方式，如果是通过 Snapshot 得到的备份集，就和第二种备份恢复方式完全一致。

4. 配置并启动 Slave

在完成了前面三个步骤之后， Replication 环境的搭建就只需要最后的一个步骤

了，那就是通过 CHANGE MASTER TO 命令来配置 然后再启动 Slave 了。

CHANGE MASTER TO 命令总共需要设置 5 项内容，分别为：
MASTER_HOST：Master 的主机名（或者 IP 地址）；
MASTER_USER：Slave 连接 Master 的用户名，实际上就是之前所创建的 repl 用户；
MASTER_PASSWORD：Slave 连接 Master 的用户的密码；
MASTER_LOG_FILE：开始复制的日志文件名称；
MASTER_LOG_POS：开始复制的日志文件的位置，也就是在之前介绍备份集过程中一致提到的 Log Position。

下面是一个完整的 CHANGE MASTER TO 命令示例：
CHANGE MASTER TO
```
root@localhost : mysql 08:32:38> CHANGE MASTER TO
    -> MASTER_HOST='192.168.0.1',
    -> MASTER_USER='repl',
    -> MASTER_PASSWORD='password',
    -> MASTER_LOG_FILE='mysql-bin.000035',
    -> MASTER_LOG_POS=399;
```

执行完 CHANGE MASTER TO 命令之后，就可以通过如下命令启动 SLAVE 了：

```
root@localhost : mysql 08:33:49> START SLAVE;
```

至此，我们的 Replication 环境就搭建完成了。读者朋友可以自己进行相应的测试来尝试搭建，如果需要了解 MySQL Replication 搭建过程中更为详细的步骤，可以通过查阅 MySQL 官方手册。

13.5 小结

在实际应用场景中,MySQL Replication 是使用最为广泛的一种提高系统扩展性的设计手段。众多的 MySQL 使用者通过 Replication 功能提升系统的扩展性之后，通过简单的增加价格低廉的硬件设备成倍甚至成数量级的提高了原有系统的性能，是广大 MySQL 中低端使用者最为喜爱的功能之一，也是大量 MySQL 使用者选择 MySQL 最为重要的理由之一。

第 14 章 可扩展性设计之数据切分

前言

通过 MySQL Replication 功能所实现的扩展总是会受到数据库大小的限制，一旦数据库过于庞大，尤其是当写入过于频繁，很难由一台主机支撑的时候，我们还是会面临到扩展瓶颈。这时候，我们就必须许找其他技术手段来解决这个瓶颈，那就是我们这一章所要介绍恶的数据切分技术。

14.1 何谓数据切分

可能很多读者朋友在网上或者杂志上面都已经多次见到关于数据切分的相关文章了，只

不过在有些文章中称之为数据的 Sharding。其实不管是称之为数据的 Sharding 还是数据的切分，其概念都是一样的。简单来说，就是指通过某种特定的条件，将我们存放在同一个数据库中的数据分散存放到多个数据库（主机）上面，以达到分散单台设备负载的效果。数据的切分同时还可以提高系统的总体可用性，因为单台设备 Crash 之后，只有总体数据的某部分不可用，而不是所有的数据。

数据的切分（Sharding）根据其切分规则的类型，可以分为两种切分模式。一种是按照不同的表（或者 Schema）来切分到不同的数据库（主机）之上，这种切可以称之为数据的垂直（纵向）切分；另外一种则是根据表中的数据的逻辑关系，将同一个表中的数据按照某种条件拆分到多台数据库（主机）上面，这种切分称之为数据的水平（横向）切分。

垂直切分的最大特点就是规则简单，实施也更为方便，尤其适合各业务之间的耦合度非常低，相互影响很小，业务逻辑非常清晰的系统。在这种系统中，可以很容易做到将不同业务模块所使用的表分拆到不同的数据库中。根据不同的表来进行拆分，对应用程序的影响也更小，拆分规则也会比较简单清晰。

水平切分于垂直切分相比，相对来说稍微复杂一些。因为要将同一个表中的不同数据拆分到不同的数据库中，对于应用程序来说，拆分规则本身就较根据表名来拆分更为复杂，后期的数据维护也会更为复杂一些。

当我们某个（或者某些）表的数据量和访问量特别的大，通过垂直切分将其放在独立的设备上后仍然无法满足性能要求，这时候我们就必须将垂直切分和水平切分相结合，先垂直切分，然后再水平切分，才能解决这种超大型表的性能问题。

下面我们就针对垂直、水平以及组合切分这三种数据切分方式的架构实现及切分后数据的整合进行相应的分析。

14.2 数据的垂直切分

我们先来看一下，数据的垂直切分到底是如何一个切分法的。数据的垂直切分，也可以称之为纵向切分。将数据库想象成为由很多个一大块一大块的“数据块”（表）组成，我们垂直的将这些“数据块”切开，然后将他们分散到多台数据库主机上面。这样的切分方法就是一个垂直（纵向）的数据切分。

一个架构设计较好的应用系统，其总体功能肯定是由很多个功能模块所组成的，而每一个功能模块所需要的数据对应到数据库中就是一个或者多个表。而在架构设计中，各个功能模块相互之间的交互点越统一越少，系统的耦合度就越低，系统各个模块的维护性以及扩展性也就越好。这样的系统，实现数据的垂直切分也就越容易。

当我们的功能模块越清晰，耦合度越低，数据垂直切分的规则定义也就越容易。完全可以根据功能模块来进行数据的切分，不同功能模块的数据存放于不同的数据库主机中，可以很容易就避免掉跨数据库的 Join 存在，同时系统架构也非常的清晰。

当然，很难有系统能够做到所有功能模块所使用的表完全独立，完全不需要访问对方的表或者需要两个模块的表进行 Join 操作。这种情况下，我们就必须根据实际的应用场景进行评估权衡。决定是迁就应用程序将需要 Join 的表的相关某快都存放在同一个数据库中，还是让应用程序做更多的事情，也就是程序完全通过模块接口取得不同数据库中的数据，然后在程序中完成 Join 操作。

一般来说，如果是一个负载相对不是很大的系统，而且表关联又非常的频繁，那可能数据库让步，将几个相关模块合并在一起减少应用程序的工作的方案可以减少较多的工作量，是一个可行的方案。

当然，通过数据库的让步，让多个模块集中共用数据源，实际上也是简介的默许了各模块架构耦合度增大的发展，可能会让以后的架构越来越恶化。尤其是当发展到一定阶段之后，发现数据库实在无法承担这些表所带来的压力，不得不面临再次切分的时候，所带来的架构改造成本可能会远远大于最初的时候。

所以，在数据库进行垂直切分的时候，如何切分，切分到什么样的程度，是一个比较考验人的难题。只能在实际的应用场景中通过平衡各方面的成本和收益，才能分析出一个真正适合自己的拆分方案。

比如在本书所使用示例系统的 example 数据库，我们简单的分析一下，然后再设计一个简单的切分规则，进行一次垂直垂直拆分。

系统功能可以基本分为四个功能模块：用户，群组消息，相册以及事件，分别对应为如下这些表：

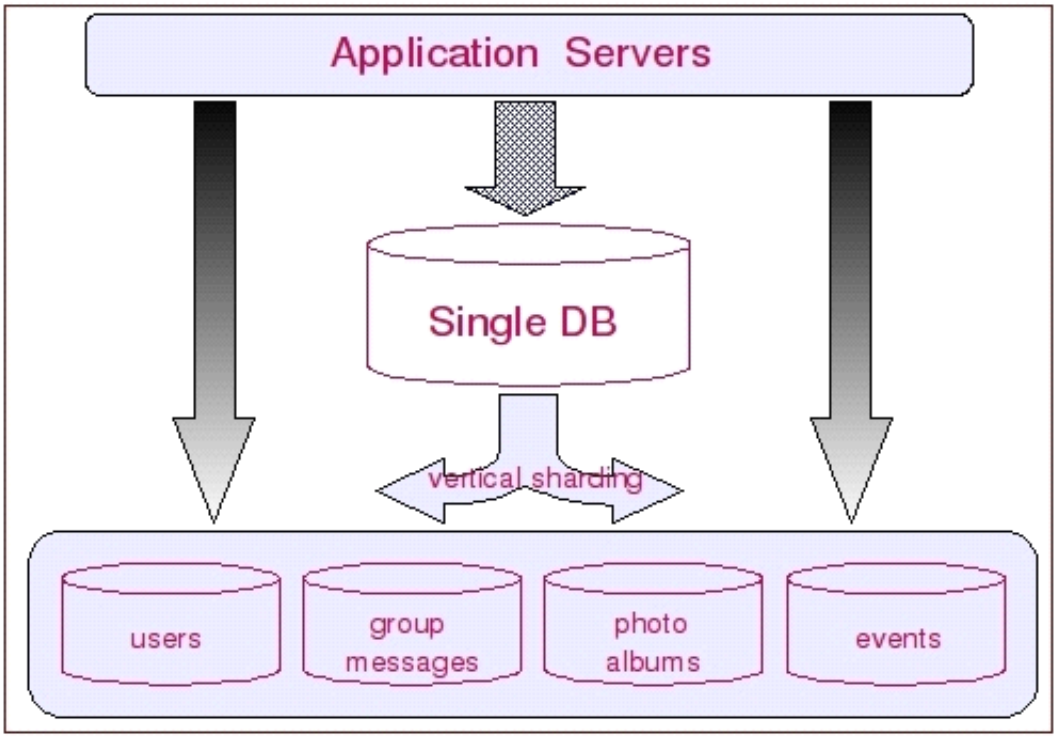
1. 用户模块表：user,user_profile,user_group,user_photo_album
2. 群组讨论表：groups,group_message,group_message_content,top_message
3. 相册相关表：photo,photo_album,photo_album_relation,photo_comment
4. 事件信息表：event

初略一看，没有哪一个模块可以脱离其他模块独立存在，模块与模块之间都存在着关系，莫非无法切分？

当然不是，我们再稍微深入分析一下，可以发现，虽然各个模块所使用的表之间都有关联，但是关联关系还算比较清晰，也比较简单。

- 群组讨论模块和用户模块之间主要存在通过用户或者是群组关系来进行关联。一般关联的时候都会是通过用户的 id 或者 nick_name 以及 group 的 id 来进行关联，通过模块之间的接口实现不会带来太多麻烦；
- 相册模块仅仅与用户模块存在通过用户的关联。这两个模块之间的关联基本就有通过用户 id 关联的内容，简单清晰，接口明确；
- 事件模块与各个模块可能都有关联，但是都只关注其各个模块中对象的 ID 信息，同样可以做到很容易分拆。

所以，我们第一步可以将数据库按照功能模块相关的表进行一次垂直拆分，每个模块所涉及的表单独到一个数据库中，模块与模块之间的表关联都在应用系统端通过藉口来处理。如下图所示：



通过这样的垂直切分之后，之前只能通过一个数据库来提供的服务，就被分拆成四个数据库来提供服务，服务能力自然是增加几倍了。

垂直切分的优点

- 数据库的拆分简单明了，拆分规则明确；
- 应用程序模块清晰明确，整合容易；
- 数据维护方便易行，容易定位；

垂直切分的缺点

- 部分表关联无法在数据库级别完成，需要在程序中完成；
- 对于访问极其频繁且数据量超大的表仍然存在性能平静，不一定能满足要求；
- 事务处理相对更为复杂；
- 切分达到一定程度之后，扩展性会遇到限制；
- 过读切分可能会带来系统过渡复杂而难以维护。

针对于垂直切分可能遇到数据切分及事务问题，在数据库层面实在是很难找到一个较好的处理方案。实际应用案例中，数据库的垂直切分大多是与应用系统的模块相对应，同一个模块的数据源存放于同一个数据库中，可以解决模块内部的数据关联问题。而模块与模块之间，则通过应用程序以服务接口方式来相互提供所需要的数据。虽然这样做在数据库的总体操作次数方面确实会有所增加，但是在系统整体扩展性以及架构模块化方面，都是有益的。

可能在某些操作的单次响应时间会稍有增加，但是系统的整体性能很可能反而会有一定的提升。而扩展瓶颈问题，就只能依靠下一节将要介绍的数据水平切分架构来解决了。

14.3 数据的水平切分

上面一节分析介绍了数据的垂直切分，这一节再分析一下数据的水平切分。数据的垂直切分基本上可以简单的理解为按照表按照模块来切分数据，而水平切分就不再是按照表或者是功能模块来切分了。一般来说，简单的水平切分主要是将某个访问极其平凡的表再按照某个字段的某种规则来分散到多个表之中，每个表中包含一部分数据。

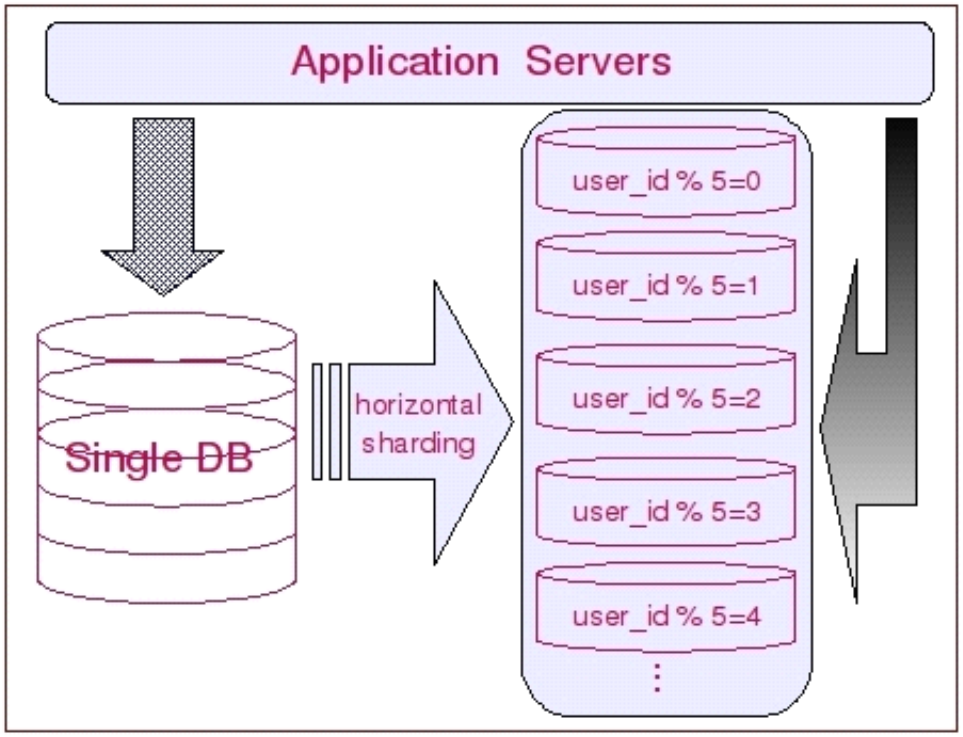
简单来说，我们可以将数据的水平切分理解为是按照数据行的切分，就是将表中的某些行切分到一个数据库，而另外的某些行又切分到其他的数据库中。当然，为了能够比较容易的判定各行数据被切分到哪个数据库中了，切分总是都需要按照某种特定的规则来进行的。如根据某个数字类型字段基于特定数目取模，某个时间类型字段的范围，或者是某个字符类型字段的 hash 值。如果整个系统中大部分核心表都可以通过某个字段来进行关联，那这个字段自然是一个进行水平分区的上上之选了，当然，非常特殊无法使用就只能另选其他了。

一般来说，像现在互联网非常火爆的 Web2.0 类型的网站，基本上大部分数据都能够通过会员用户信息关联上，可能很多核心表都非常适合通过会员 ID 来进行数据的水平切分。而像论坛社区讨论系统，就更容易切分了，非常容易按照论坛编号来进行数据的水平切分。切分之后基本上不会出现各个库之间的交互。

如我们的示例系统，所有数据都是和用户关联的，那么我们就可以根据用户来进行水平拆分，将不同用户的数据切分到不同的数据库中。当然，唯一有点区别的是用户模块中的 groups 表和用户没有直接关系，所以 groups 不能根据用户来进行水平拆分。对于这种特殊情况下的表，我们完全可以独立出来，单独放在一个独立的数据库中。其实这个做法可以说是利用了前面一节所介绍的“数据的垂直切分”方法，我将在下一节中更为详细的介绍这种垂直切分与水平切分同时使用的联合切分方法。

所以，对于我们的示例数据库来说，大部分的表都可以根据用户 ID 来进行水平的切分。不同用户相关的数据进行切分之后存放在不同的数据库中。如将所有用户 ID 通过 2 取模然后分别存放于两个不同的数据库中。每个和用户 ID 关联上的表都可以这样切分。这样，基本上每个用户相关的数据，都在同一个数据库中，即使是需要关联，也可以非常简单的关联上。

我们可以通过下图来更为直观的展示水平切分相关信息：



水平切分的优点

- 表关联基本能够在数据库端全部完成；
- 不会存在某些超大型数据量和高负载的表遇到瓶颈的问题；
- 应用程序端整体架构改动相对较少；
- 事务处理相对简单；
- 只要切分规则能够定义好，基本上较难遇到扩展性限制；

水平切分的缺点

- 切分规则相对更为复杂，很难抽象出一个能够满足整个数据库的切分规则；
- 后期数据的维护难度有所增加，人为手工定位数据更困难；
- 应用系统各模块耦合度较高，可能会对后面数据的迁移拆分造成一定的困难。

14.4 垂直与水平联合切分的使用

上面两节内容中，我们分别，了解了“垂直”和“水平”这两种切分方式的实现以及切分之后的架构信息，同时也分析了两种架构各自的优缺点。但是在实际的应用场景中，除了那些负载并不是太大，业务逻辑也相对较简单的系统可以通过上面两种切分方法之一来解决扩展性问题之外，恐怕其他大部分业务逻辑稍微复杂一点，系统负载大一些的系统，都无法通过上面任何一种数据的切分方法来实现较好的扩展性，而需要将上述两种切分方法结合使用，不同的场景使用不同的切分方法。

在这一节中，我将结合垂直切分和水平切分各自的优缺点，进一步完善我们的整体架构，让系统的扩展性进一步提高。

一般来说，我们数据库中的所有表很难通过某一个（或少数几个）字段全部关联起来，所以很难简单的仅仅通过数据的水平切分来解决所有问题。而垂直切分也只能解决部分问题，对于那些负载非常高的系统，即使仅仅只是单个表都无法通过单台数据库主机来承担其负载。我们必须结合“垂直”和“水平”两种切分方式同时使用，充分利用两者的优点，避开其缺点。

每一个应用系统的负载都是一步一步增长上来的，在开始遇到性能瓶颈的时候，大多数架构师和 DBA 都会选择先进行数据的垂直拆分，因为这样的成本最先，最符合这个时期所追求的最大投入产出比。然而，随着业务的不断扩张，系统负载的持续增长，在系统稳定一段时期之后，经过了垂直拆分之后的数据库集群可能又再一次不堪重负，遇到了性能瓶颈。

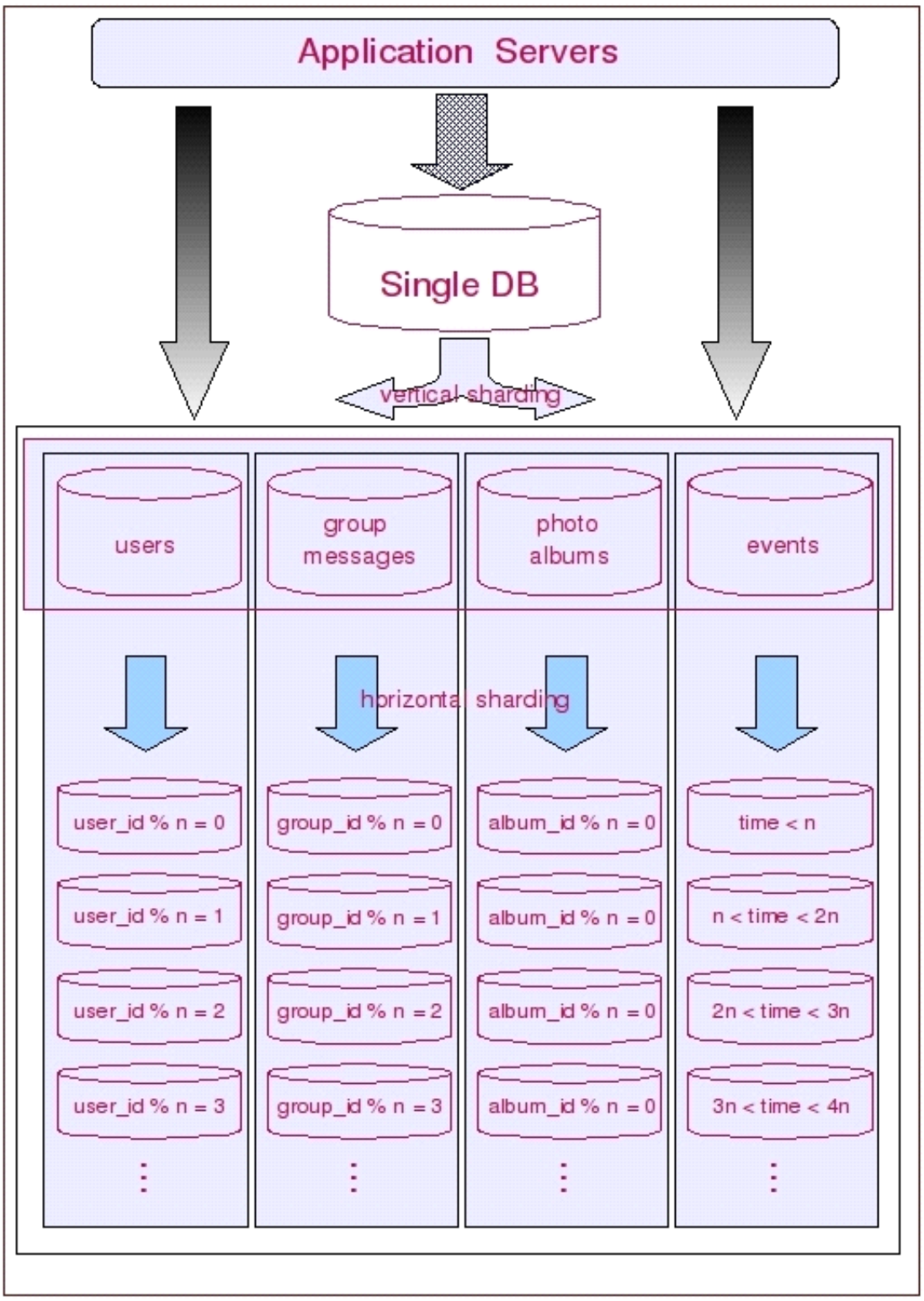
这时候我们该如何抉择？是再次进一步细分模块呢，还是寻求其他的办法来解决？如果我们再一次像最开始那样继续细分模块，进行数据的垂直切分，那我们可能在不久的将来，又会遇到现在所面对的同样的问题。而且随着模块的不断的细化，应用系统的架构也会越来越复杂，整个系统很可能会出现失控的局面。

这时候我们就必须要通过数据的水平切分的优势，来解决这里所遇到的问题。而且，我们完全不必要在使用数据水平切分的时候，推倒之前进行数据垂直切分的成果，而是在其基础上利用水平切分的优势来避开垂直切分的弊端，解决系统复杂性不断扩大的问题。而水平拆分的弊端（规则难以统一）也已经被之前的垂直切分解决掉了，让水平拆分可以进行的得心应手。

对于我们的示例数据库，假设在最开始，我们进行了数据的垂直切分，然而随着业务的不断增长，数据库系统遇到了瓶颈，我们选择重构数据库集群的架构。如何重构？考虑到之前已经做好了数据的垂直切分，而且模块结构清晰明确。而业务增长的势头越来越猛，即使现在进一步再次拆分模块，也坚持不了太久。我们选择了在垂直切分的基础上再进行水平拆分。

在经历过垂直拆分后的各个数据库集群中的每一个都只有一个功能模块，而每个功能模块中的所有表基本上都会与某个字段进行关联。如用户模块全部都可以通过用户 ID 进行切分，群组讨论模块则都通过群组 ID 来切分，相册模块则根据相册 ID 来进切分，最后的事件通知信息表考虑到数据的时限性（仅仅只会访问最近某个事件段的信息），则考虑按时间来切分。

下图展示了切分后的整个架构：



实际上，在很多大型的应用系统中，垂直切分和水平切这两种数据的切分方法基本上都是并存的，而且经常在不断的交替进行，以不断的增加系统的扩展能力。我们在应对不同的应用场景的时候，也需要充分考虑到这两种切分方法各自的局限，以及各自的优势，在不同的时期（负载压力）使用不同的结合方式。

联合切分的优点

- 可以充分利用垂直切分和水平切分各自的优势而避免各自的缺陷；
- 让系统扩展性得到最大化提升；

联合切分的缺点

- 数据库系统架构比较复杂，维护难度更大；
- 应用程序架构也相对更复杂；

14.5 数据切分及整合方案

通过前面的章节，我们已经很清楚了通过数据库的数据切分可以极大的提高系统的扩展性。但是，数据库中的数据在经过垂直和（或）水平切分被存放在不同的数据库主机之后，应用系统面临的最大问题就是如何来让这些数据源得到较好的整合，可能这也是很多读者朋友非常关心的一个问题。这一节我们主要针对的内容就是分析可以使用的各种可以帮助我们实现数据切分以及数据整合的整体解决方案。

数据的整合很难依靠数据库本身来达到这个效果，虽然 MySQL 存在 Federated 存储引擎，可以解决部分类似的问题，但是在实际应用场景中却很难较好的运用。那我们该如何来整合这些分散在各个 MySQL 主机上面的数据源呢？

总的来说，存在两种解决思路：

1. 在每个应用程序模块中配置管理自己需要的一个（或者多个）数据源，直接访问各个数据库，在模块内完成数据的整合；
2. 通过中间代理层来统一管理所有的数据源，后端数据库集群对前端应用程序透明；

可能 90%以上的人在面对上面这两种解决思路的时候都会倾向于选择第二种，尤其是系统不断变得庞大复杂的时候。确实，这是一个非常正确的选择，虽然短期内需要付出的成本可能会相对更大一些，但是对整个系统的扩展性来说，是非常有帮助的。

所以，对于第一种解决思路我这里就不准备过多的分析，下面我重点分析一下在第二种解决思路中的一些解决方案。

★ 自行开发中间代理层

在决定选择通过数据库的中间代理层来解决数据源整合的架构方向之后，有不少公司（或者企业）选择了通过自行开发符合自身应用特定场景的代理层应用程序。

通过自行开发中间代理层可以最大程度的应对自身应用的特定，最大化的定制很多个性化需求，在面对变化的时候也可以灵活的应对。这应该说是自行开发代理层最大的优势了。

当然，选择自行开发，享受让个性化定制最大化的乐趣的同时，自然也需要投入更多的成本来进行前期研发以及后期的持续升级改进工作，而且本身的技术门槛可能也比简单的 Web 应用要更高一些。所以，在决定选择自行开发之前，还是需要进行比较全面的评估为好 。

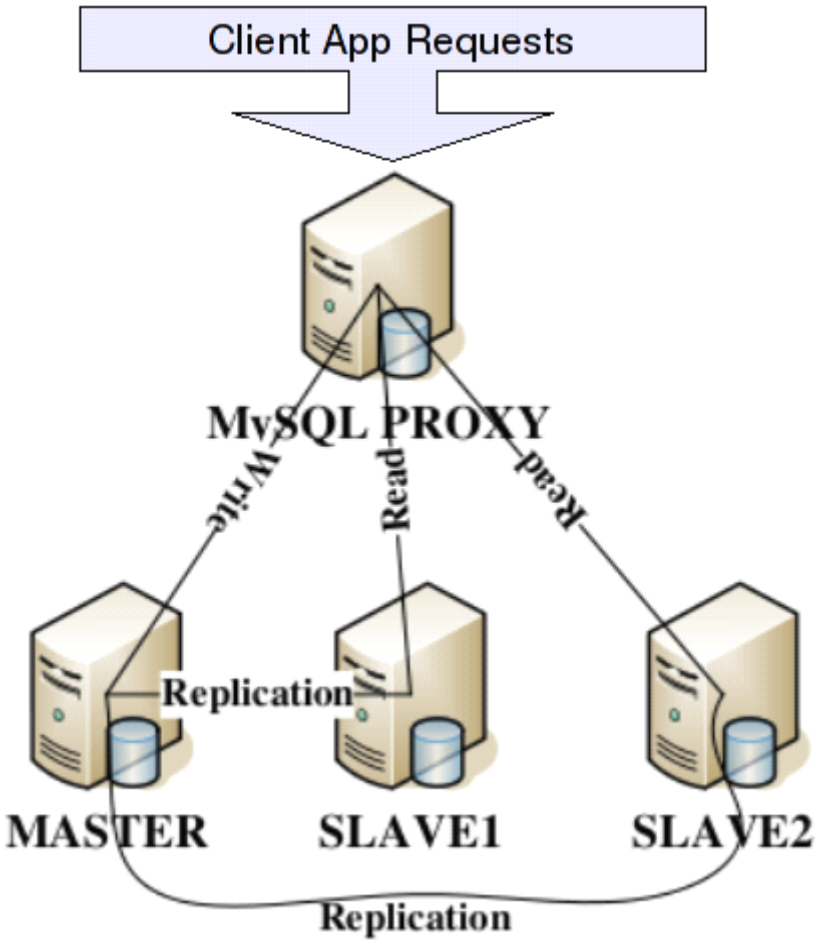
由于自行开发更多时候考虑的是如何更好的适应自身应用系统，应对自身的业务场景，所以这里也不好分析太多。后面我们主要分析一下当前比较流行的几种数据源整合解决方案。

★ 利用 MySQL Proxy 实现数据切分及整合

MySQL Proxy 是 MySQL 官方提供的一个数据库代理层产品，和 MySQL Server 一样，同样是一个基于 GPL 开源协议的开源产品。可用来监视、分析或者传输他们之间的通讯信息。他的灵活性允许你最大限度的使用它，目前具备的功能主要有连接路由，Query 分析，Query 过滤和修改，负载均衡，以及基本的 HA 机制等。

实际上，MySQL Proxy 本身并不具有上述所有的这些功能，而是提供了实现上述功能的基础。要实现这些功能，还需要通过我们自行编写 LUA 脚本来实现。

MySQL Proxy 实际上是在客户端请求与 MySQL Server 之间建立了一个连接池。所有客户端请求都是发向 MySQL Proxy，然后经由 MySQL Proxy 进行相应的分析，判断出是读操作还是写操作，分发至对应的 MySQL Server 上。对于多节点 Slave 集群，也可以起做到负载均衡的效果。以下是 MySQL Proxy 的基本架构图：



通过上面的架构简图，我们可以很清晰的看出 MySQL Proxy 在实际应用中所处的位置，以及能做的基本事情。关于 MySQL Proxy 更为详细的实施细则在 MySQL 官方文档中有非常详细的介绍和示例，感兴趣的读者朋友可以直接从 MySQL 官方网站免费下载或者在线阅读，我这里就不累述浪费纸张了。

★ 利用 Amoeba 实现数据切分及整合

Amoeba 是一个基于 Java 开发的，专注于解决分布式数据库数据源整合 Proxy 程序的开源框架，基于 GPL3 开源协议。目前，Amoeba 已经具有 Query 路由，Query 过滤，读写分离，负载均衡以及 HA 机制等相关内容。

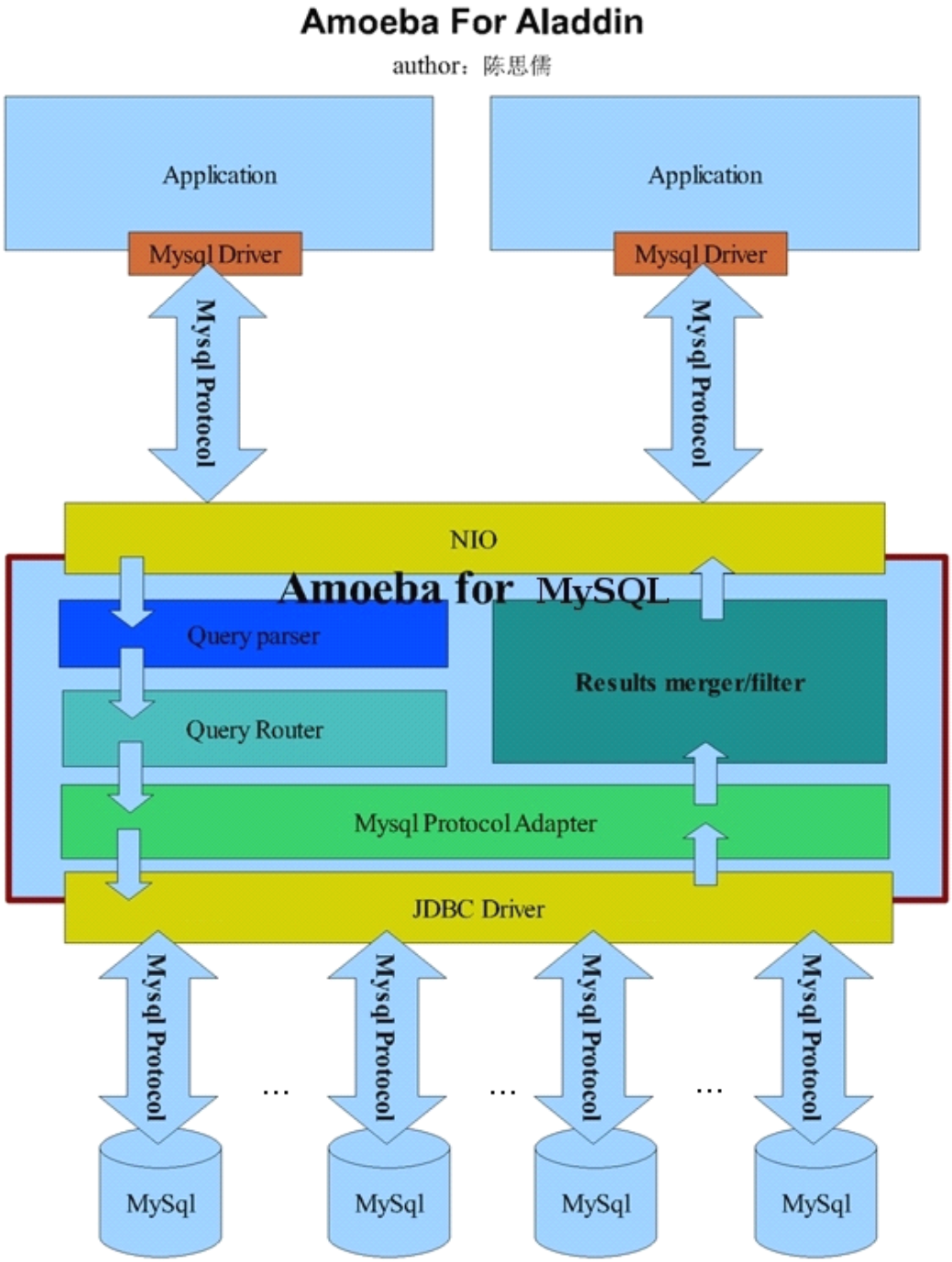
Amoeba 主要解决的以下几个问题：

1. 数据切分后复杂数据源整合；
2. 提供数据切分规则并降低数据切分规则给数据库带来的影响；
3. 降低数据库与客户端的连接数；
4. 读写分离路由；

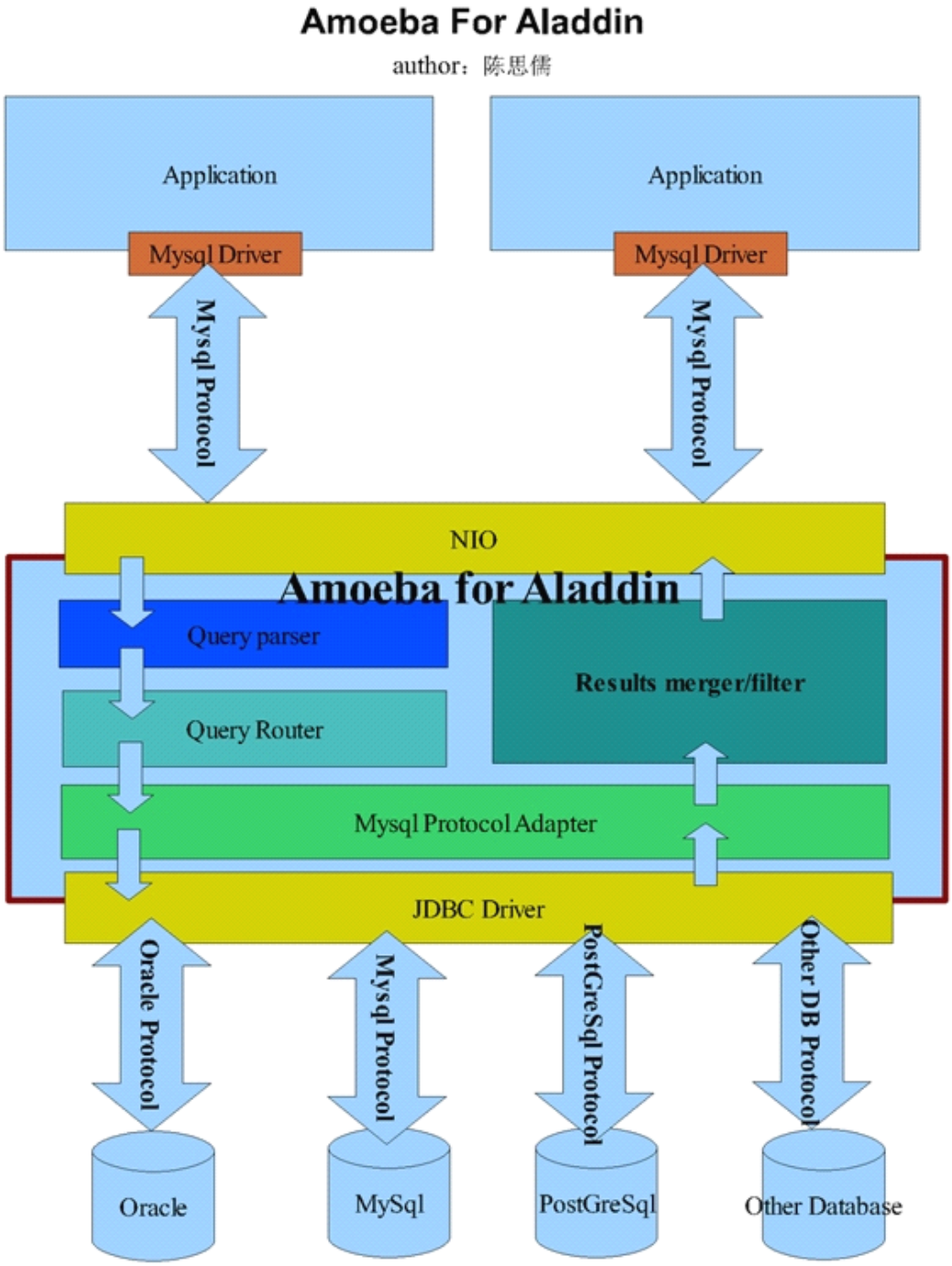
我们可以看出，Amoeba 所做的事情，正好就是我们通过数据切分来提升数据库的扩展性所需要的。

Amoeba 并不是一个代理层的 Proxy 程序，而是一个开发数据库代理层 Proxy 程序的开发框架，目前基于 Amoeba 所开发的 Proxy 程序有 Amoeba For MySQL 和 Amoeba For Aladin 两个。

Amoeba For MySQL 主要是专门针对 MySQL 数据库的解决方案，前端应用程序请求的协议以及后端连接的数据源数据库都必须是 MySQL。对于客户端的任何应用程序来说，Amoeba For MySQL 和一个 MySQL 数据库没有什么区别，任何使用 MySQL 协议的客户端请求，都可以被 Amoeba For MySQL 解析并进行相应的处理。下如可以告诉我们 Amoeba For MySQL 的架构信息（出自 Amoeba 开发者博客）：



Amoeba For Aladin 则是一个适用更为广泛，功能更为强大的 Proxy 程序。他可以同时连接不同数据库的数据源为前端应用程序提供服务，但是仅仅接受符合 MySQL 协议的客户端应用程序请求。也就是说，只要前端应用程序通过 MySQL 协议连接上来之后，Amoeba For Aladin 会自动分析 Query 语句，根据 Query 语句中所请求的数据来自动识别出该所 Query 的数据源是在什么类型数据库的哪一个物理主机上面。下图展示了 Amoeba For Aladin 的架构细节（出自 Amoeba 开发者博客）：



咋一看，两者好像完全一样嘛。细看之后，才会发现两者主要的区别仅在于通过 MySQL Protocal Adapter 处理之后，根据分析结果判断出数据源数据库，然后选择特定的 JDBC

驱动和相应协议连接后端数据库。

其实通过上面两个架构图大家可能也已经发现了 Amoeba 的特点了，他仅仅只是一个开发框架，我们除了选择他已经提供的 For MySQL 和 For Aladin 这两款产品之外，还可以基于自身的需求进行相应的二次开发，得到更适应我们自己应用特点的 Proxy 程序。

当对于使用 MySQL 数据库来说，不论是 Amoeba For MySQL 还是 Amoeba For Aladin 都可以很好的使用。当然，考虑到任何一个系统越是复杂，其性能肯定就会有一定的损失，维护成本自然也会相对更高一些。所以，对于仅仅需要使用 MySQL 数据库的时候，我还是建议使用 Amoeba For MySQL。

Amoeba For MySQL 的使用非常简单，所有的配置文件都是标准的 XML 文件，总共有四个配置文件。分别为：

- amoeba.xml：主配置文件，配置所有数据源以及 Amoeba 自身的参数设置；
- rule.xml：配置所有 Query 路由规则的信息；
- functionMap.xml：配置用于解析 Query 中的函数所对应的 Java 实现类；
- rullFunctionMap.xml：配置路由规则中需要使用到的特定函数的实现类；

如果您的规则不是太复杂，基本上仅需要使用到上面四个配置文件中的前面两个就可完成所有工作。Proxy 程序常用的功能如读写分离，负载均衡等配置都在 amoeba.xml 中进行。此外，Amoeba 已经支持了实现数据的垂直切分和水平切分的自动路由，路由规则可以在 rule.xml 进行设置。

目前 Amoeba 少有欠缺的主要就是其在线管理功能以及对事务的支持了，曾经在与相关开发者的沟通过程中提出过相关的建议，希望能够提供一个可以进行在线维护管理的命令行管理工具，方便在线维护使用，得到的反馈是管理专门的管理模块已经纳入开发日程了。另外在事务支持方面暂时还是 Amoeba 无法做到的，即使客户端应用在提交给 Amoeba 的请求是包含事务信息的，Amoeba 也会忽略事务相关信息。当然，在经过不断完善之后，我相信事务支持肯定是 Amoeba 重点考虑增加的 feature。

关于 Amoeba 更为详细的使用方法读者朋友可以通过 Amoeba 开发者博客（http://amoeba.sf.net）上面提供的使用手册获取，这里就不再细述了。

★ 利用 HiveDB 实现数据切分及整合

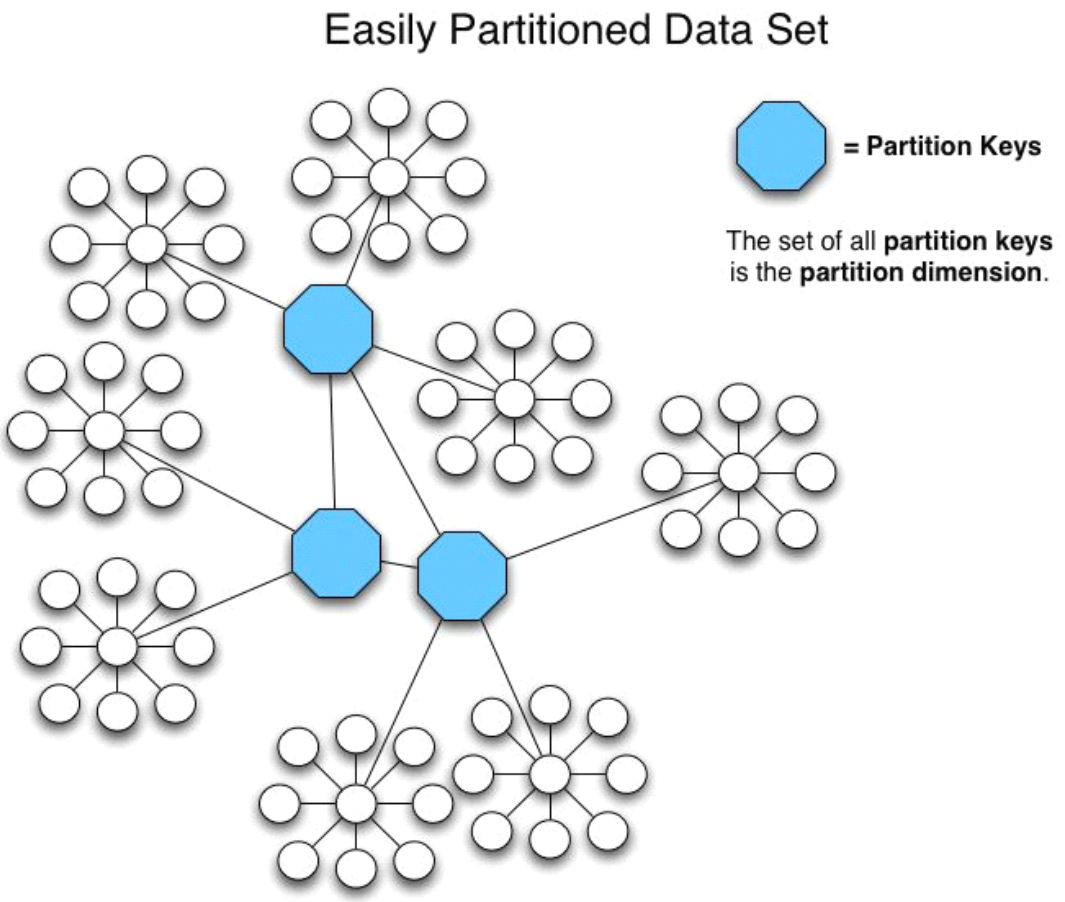
和前面的 MySQL Proxy 以及 Amoeba 一样，HiveDB 同样是一个基于 Java 针对 MySQL 数据库的提供数据切分及整合的开源框架，只是目前的 HiveDB 仅仅支持数据的水平切分。主要解决大数据量下数据库的扩展性及数据的高性能访问问题，同时支持数据的冗余及基本的 HA 机制。

HiveDB 的实现机制与 MySQL Proxy 和 Amoeba 有一定的差异，他并不是借助 MySQL 的 Replication 功能来实现数据的冗余，而是自行实现了数据冗余机制，而其底层主要是基于 Hibernate Shards 来实现的数据切分工作。

在 HiveDB 中,通过用户自定义的各种 Partition keys(其实就是制定数据切分规则),将数据分散到多个 MySQL Server 中。在访问的时候，在运行 Query 请求的时候，会自动分析过滤条件，并行从多个 MySQL Server 中读取数据，并合并结果集返回给客户端应用程序。

单纯从功能方面来讲，HiveDB 可能并不如 MySQL Proxy 和 Amoeba 那样强大，但是其数据切分的思路与前面二者并无本质差异。此外，HiveDB 并不仅仅只是一个开源爱好者所共享的内容，而是存在商业公司支持的开源项目。

下面是 HiveDB 官方网站上面一章图片，描述了 HiveDB 如何来组织数据的基本信息，虽然不能详细的表现出太多架构方面的信息,但是也基本可以展示出其在数据切分方面独特的一面了。



★ 其他实现数据切分及整合的解决方案

除了上面介绍的几个数据切分及整合的整体解决方案之外,还存在很多其他同样提供了数据切分与整合的解决方案。如基于 MySQL Proxy 的基础上做了进一步扩展的 HSCALE，通过 Rails 构建的 Spock Proxy，以及基于 Pathon 的 Pyshards 等等。

不管大家选择使用哪一种解决方案，总体设计思路基本上都不应该会有任何变化，那就是通过数据的垂直和水平切分，增强数据库的整体服务能力，让应用系统的整体扩展能力尽可能的提升，扩展方式尽可能的便捷。

只要我们通过中间层 Proxy 应用程序较好的解决了数据切分和数据源整合问题，那么数据库的线性扩展能力将很容易做到像我们的应用程序一样方便，只需要通过添加廉价的 PC Server 服务器，即可线性增加数据库集群的整体服务能力，让数据库不再轻易成为应用系统的性能瓶颈。

14.6 数据切分与整合中可能存在的问题

这里，大家应该对数据切分与整合的实施有了一定的认识了，或许很多读者朋友都已经根据各种解决方案各自特性的优劣基本选定了适合于自己应用场景的方案，后面的工作主要就是实施准备了。

在实施数据切分方案之前，有些可能存在的问题我们还是需要做一些分析的。一般来说，我们可能遇到的问题主要会有以下几点：

- 引入分布式事务的问题；
- 跨节点 Join 的问题；
- 跨节点合并排序分页问题；

1. 引入分布式事务的问题

一旦数据进行切分被分别存放在多个 MySQL Server 中之后，不管我们的切分规则设计的多么的完美（实际上并不存在完美的切分规则），都可能造成之前的某些事务所涉及到的数据已经不在同一个 MySQL Server 中了。

在这样的场景下，如果我们的应用程序仍然按照老的解决方案，那么势必需要引入分布式事务来解决。而在 MySQL 各个版本中，只有从 MySQL 5.0 开始以后的各个版本才开始对分布式事务提供支持，而且目前仅有 Innodb 提供分布式事务支持。不仅如此，即使我们刚好使用了支持分布式事务的 MySQL 版本，同时也是使用的 Innodb 存储引擎，分布式事务本身对于系统资源的消耗就是很大的，性能本身也并不是太高。而且引入分布式事务本身在异常处理方面就会带来较多比较难控制的因素。

怎么办？其实我们可以可以通过一个变通的方法来解决这种问题，首先需要考虑的一件事情就是：是否数据库是唯一一个能够解决事务的地方呢？其实并不是这样的，我们完全可以结合数据库以及应用程序两者来共同解决。各个数据库解决自己身上的事务，然后通过应用程序来控制多个数据库上面的事务。

也就是说，只要我们愿意，完全可以将一个跨多个数据库的分布式事务分拆成多个仅处于单个数据库上面的小事务，并通过应用程序来总控各个小事务。当然，这样作的要求就是

我们的俄应用程序必须要有足够的健壮性，当然也会给应用程序带来一些技术难度。

2. 跨节点 Join 的问题

上面介绍了可能引入分布式事务的问题，现在我们再看看需要跨节点 Join 的问题。数据切分之后，可能会造成有些老的 Join 语句无法继续使用，因为 Join 使用的数据源可能被切分到多个 MySQL Server 中了。

怎么办？这个问题从 MySQL 数据库角度来看，如果非得在数据库端来直接解决的话，恐怕只能通过 MySQL 一种特殊的存储引擎 Federated 来解决了。Federated 存储引擎是 MySQL 解决类似于 Oracle 的 DB Link 之类问题的解决方案。和 Oracle DB Link 的主要区别在于 Federated 会保存一份远端表结构的定义信息在本地。咋一看，Federated 确实是解决跨节点 Join 非常好的解决方案。但是我们还应该清楚一点，那就似乎如果远端的表结构发生了变更，本地的表定义信息是不会跟着发生相应变化的。如果在更新远端表结构的时候并没有更新本地的 Federated 表定义信息，就很可能造成 Query 运行出错，无法得到正确的结果。

对待这类问题，我还是推荐通过应用程序来进行处理，先在驱动表所在的 MySQL Server 中取出相应的驱动结果集，然后根据驱动结果集再到被驱动表所在的 MySQL Server 中取出相应的数据。可能很多读者朋友会认为这样做对性能会产生一定的影响，是的，确实是会对性能有一定的负面影响，但是除了此法，基本上没有太多其他更好的解决办法了。而且，由于数据库通过较好的扩展之后，每台 MySQL Server 的负载就可以得到较好的控制，单纯针对单条 Query 来说，其响应时间可能比不切分之前要提高一些，所以性能方面所带来的负面影响也并不是太大。更何况，类似于这种需要跨节点 Join 的需求也并不是太多，相对于总体性能而言，可能也只是很小一部分而已。所以为了整体性能的考虑，偶尔牺牲那么一点点，其实是值得的，毕竟系统优化本身就是存在很多取舍和平衡的过程。

3. 跨节点合并排序分页问题

一旦进行了数据的水平切分之后，可能就并不仅仅只有跨节点 Join 无法正常运行，有些排序分页的 Query 语句的数据源可能也会被切分到多个节点，这样造成的直接后果就是这些排序分页 Query 无法继续正常运行。其实这和跨节点 Join 是一个道理，数据源存在于多个节点上，要通过一个 Query 来解决，就和跨节点 Join 是一样的操作。同样 Federated 也可以部分解决，当然存在的风险也一样。

还是同样的问题，怎么办？我同样仍然继续建议通过应用程序来解决。

如何解决？解决的思路大体上和跨节点 Join 的解决类似，但是有一点和跨节点 Join 不太一样，Join 很多时候都有一个驱动与被驱动的关系，所以 Join 本身涉及到的多个表之间的数据读取一般都会存在一个顺序关系。但是排序分页就不太一样了，排序分页的数据源基本上可以说是一个表（或者一个结果集），本身并不存在一个顺序关系，所以在从多个数据源取数据的过程是完全可以并行的。这样，排序分页数据的取数效率我们可以做的比跨库 Join 更高，所以带来的性能损失相对的要更小，在有些情况下可能比在原来未进行数据切分的数据库中效率更高了。当然，不论是跨节点 Join 还是跨节点排序分页，都会使我们

的应用服务器消耗更多的资源，尤其是内存资源，因为我们在读取访问以及合并结果集的这个过程需要比原来处理更多的数据。

分析到这里，可能很多读者朋友会发现，上面所有的这些问题，我给出的建议基本上都是通过应用程序来解决。大家可能心里开始犯嘀咕了，是不是因为我是 DBA，所以就很多事情都扔给应用架构师和开发人员了？

其实完全不是这样，首先应用程序由于其特殊性，可以非常容易做到很好的扩展性，但是数据库就不一样，必须借助很多其他的方式才能做到扩展，而且在这个扩展过程中，很难避免带来有些原来在集中式数据库中可以解决但被切分开成一个数据库集群之后就成为一个难题的情况。要想让系统整体得到最大限度的扩展，我们只能让应用程序做更多的事情，来解决数据库集群无法较好解决的问题。

14.7 小结

通过数据切分技术将一个大的 MySQL Server 切分成多个小的 MySQL Server，既解决了写入性能瓶颈问题，同时也再一次提升了整个数据库集群的扩展性。不论是通过垂直切分，还是水平切分，都能够让系统遇到瓶颈的可能性更小。尤其是当我们使用垂直和水平相结合的切分方法之后，理论上将不会再遇到扩展瓶颈了。

第 15 章 可扩展性设计之 Cache 与 Search 的利用

前言：

前面章节部分所分析的可扩展架构方案，基本上都是围绕在数据库自身来进行的，这样是否会使我们在寻求扩展性之路的思维受到“禁锢”，无法更为宽广的发散开来。这一章，我们就将跳出完全依靠数据库自身来改善扩展性的问题，将数据服务扩展性的改善向数据库之外的天地延伸！

15.1 可扩展设计的数据库之外延伸

数据库主要就是为应用程序提供数据存取相应的服务，提高数据库的扩展性，也是为了更好的提供数据存取服务能力，同时包括可靠性，高效性以及易用性。所以，我们最根本的目的就是让数据层的存储服务能力得到更好的扩展性，让我们的投入尽可能的与产出成正比。

我们都明白一点，数据本身肯定都会需要有一个可以持久化地方，但是我们是否有必要让我们的所有冗余数据都进行持久化呢？我想读者朋友们肯定都会觉得没有这个必要，只要保证有至少两份冗余的数据进行持久化就足够了。而另外一些为了提高扩展性而产生的冗余数据，我们完全可以通过一些特别的技术来替代需要持久化的数据库，如内存 Cache，Search 以及磁盘文件 Cache 和 Search 等等。

寻求数据库软件本身之外的 Cache 和 Search 来解决数据本身的扩展性，已经成为目前各个大型互联网站点都在积极尝试的一个非常重要的架构升级。因为这不仅仅能更大程度的在整个应用系统提升数据处理层本身的扩展性，而且还能更大限度的提升性能。

对于这种架构方式，目前已经比较成熟的解决方案主要有基于对象的分布式内存 Cache

解决方案 Memcached，高性能嵌入式数据库编程库 Berkeley DB ，功能强大的全文搜索引擎 Lucene 等等。

当然，在使用成熟的第三方产品的同时，偶尔自行实现一些特定应用场景下的 Cache 和 Search，也未尝不是一件值得尝试的事情，而且对于公司的技术积累来说也是很有意义的一件事情。当然，决定自行开发实现之前进行全面的评估是绝对必要的，不仅仅包括自身技术实力，对应用的商业需求也需要有一定的评估才行。

其实，不论是使用现成的第三方成熟解决方案还是自主研发，都是需要在开发资源方面有一定投入的。首先要想很好的和现有 MySQL 数据库更好的结合，就有多种思路存在。可以在数据库端实现和 Cache 或者 Search 的数据通讯（数据更新），也可以在应用程序端直接实现 Cache 与 Search 的数据更新。数据库和 Cache 与 Search 可以处于整体架构中不同的层次，也可以并存于相同的层次。

下面我分别针对使用第三方成熟解决方案以及自主研发来进行一些针对性的分析和架构思路探讨，希望对各位读者朋友有一定的帮助。

15.2 合理利用第三方 Cache 解决方案

使用较为成熟的第三方解决方案最大的优势就在于在节省自身研发成本的同时，还能够在互联网上面找到较多的文档信息，帮助我们解决一些日常遇到的问题还是非常有帮助的。

目前比较流行的第三方 Cache 解决方案主要有基于对象的分布式内存 Cache 软件 Memcached 和嵌入式数据库编程库 Berkeley DB 这两种。下面我将分别针对这两种解决方案做一个分析和架构探讨。

15.2.1 分布式内存 Cache 软件 Memcached

相信对于很多读者朋友来说，Memcached 并不会太陌生了吧，他现在的流行程度已经比 MySQL 并不会差太多了。Memcached 之所以如此的流行，主要是因为以下几个原因：

- 通信协议简单，API 接口清晰；
- 高效的 Cache 算法，基于 libevent 的事件处理机制，性能卓越；
- 面向对象的特性，对应用开发人员来说非常友好；
- 所有数据都存放于内存中，数据访问高效；
- 软件开源，基于 BSD 开源协议；

对于 Memcached 本身细节，这里我就不涉及太多了，毕竟这不是本书的重点。下面我们重点看看如何通过 Memcached 来帮助我们你提升数据服务（这里如果再使用数据库本身

可能会不太合适了）的扩展性。

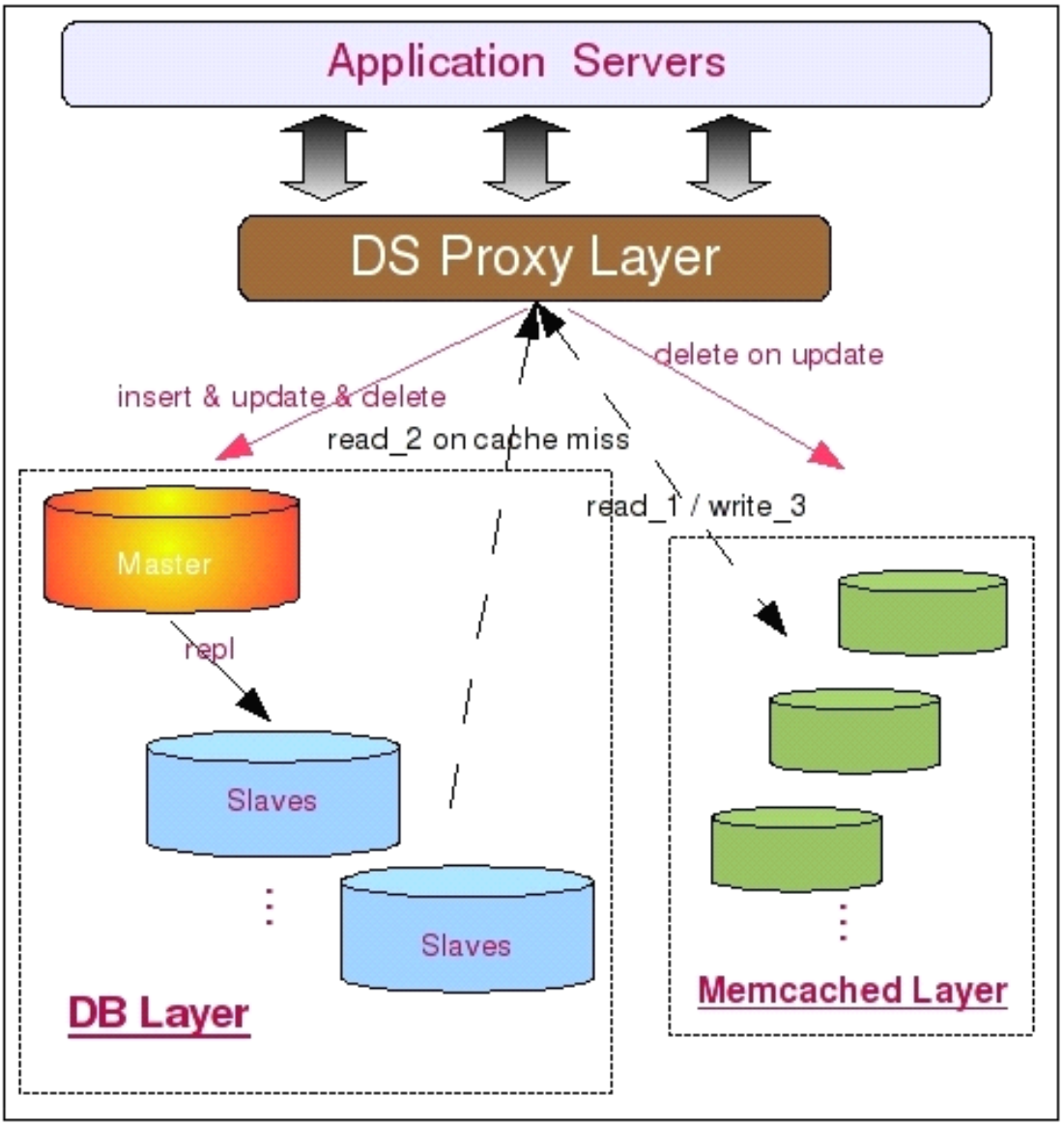
要将 Memcached 较好的整合到系统架构中，首先要在应用系统中让 Memcached 有一个准确的定位。是仅仅作为提升数据服务性能的一个 Cache 工具，还是让他与 MySQL 数据库较好的融合在一起成为一个更为更为高效理想的数据服务层。

1. 作为提升系统性能的 Cache 工具

如果我们仅仅只是系统通过 Memcached 来提升系统性能，作为一个 Cache 软件，那么更多的是需要通过应用程序来维护 Memcached 中的数据与数据库中数据的同步更新。这时候的 Memcached 基本可以理解为比 MySQL 数据库更为前端的一个 Cache 层。

如果我们将 Memcached 作为应用系统的一个数据 Cache 服务，那么对于 MySQL 数据库来说基本上不用做任何改造，仅仅通过应用程序自己来对这个 Cache 进行维护更新。这样作最大的好处就在于可以做到完全不用动数据库相关的架构，但是同时也会有一个弊端，那就是如果需要 Cache 的数据对象较多的时候，应用程序所需要增加的代码量就会增加很多，同时系统复杂度以及维护成本也会直线上升。

下面是将 Memcached 用为简单的 Cache 服务层的时候的架构简图。



从图中我们可以看到，所有数据都会写入 MySQL Master 中，包括数据第一次写入时候的 INSERT，同时也包括对已有数据的 UPDATE 和 DELETE。不过，如果是对已经存在的数据，则需要在 UPDATE 或者 DELETE MySQL 中数据的同时，删除 Memcached 中的数据，以此保证整体数据的一致性。而所有的读请求首先会发往 Memcached 中，如果读取到数据则直接返回，如果没有读取到数据，则再到 MySQL Slaves 中读取数据，并将读取得到的数据写入到 Memcached 中进行 Cache。

这种使用方式一般来说比较适用于需要缓存对象类型少，而需要缓存的数据量又比较大的环境，是一个快速有效的完全针对性能问题的解决方案。由于这种架构方式和 MySQL 数据库本身并没有太大关系，所以这里就不涉及太多的技术细节了。

2. 和 MySQL 整合为数据服务层

除了将 Memcached 用作快速提升效率的工具之外，我们其实还可以将之利用到提高数

据服务层的扩展性方面，和我们的数据库整合成一个整体，或者作为数据库的一个缓冲。

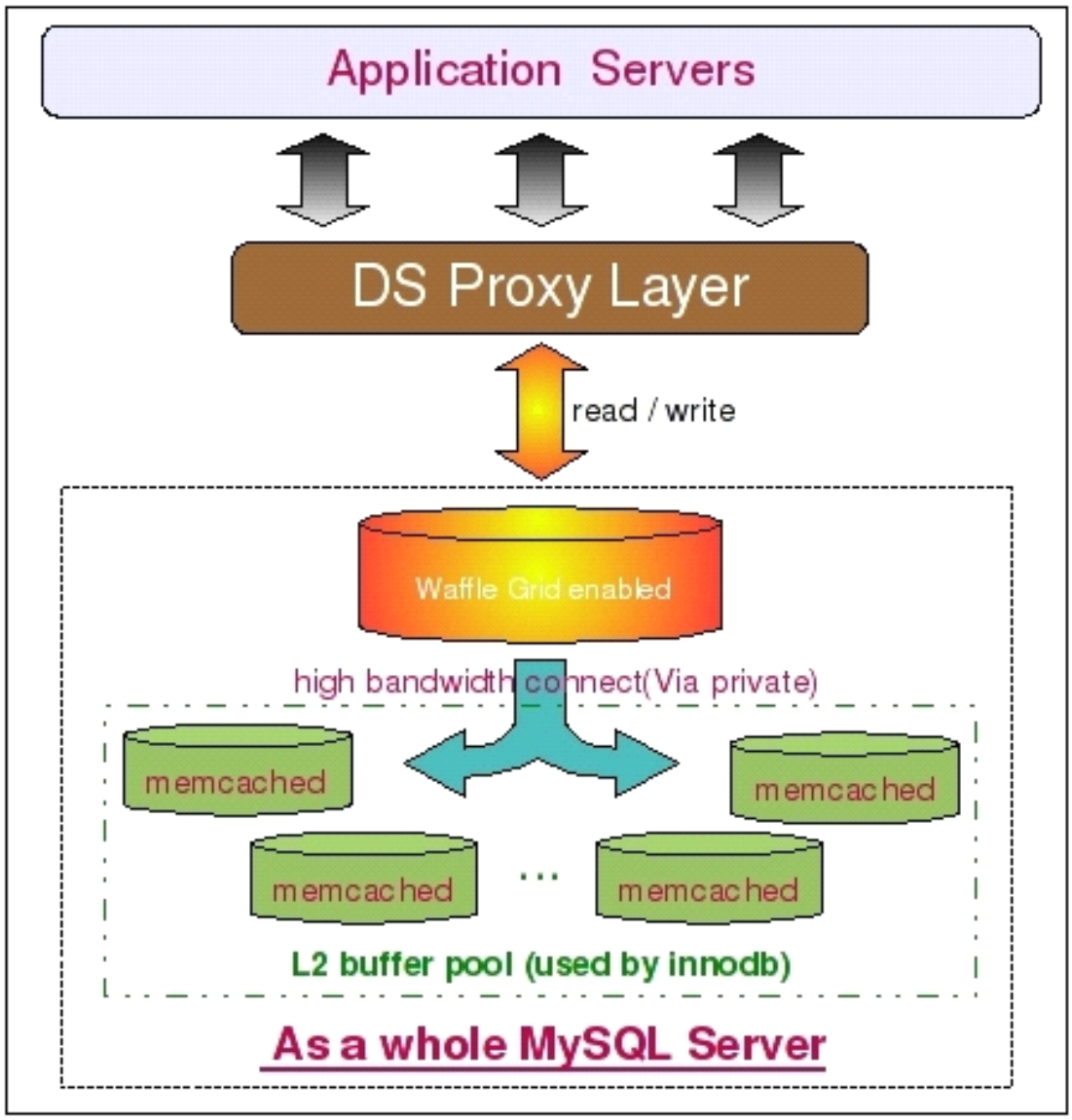
我们首先看看如何将 Memcached 和MySQL 数据库整合成一个整体来对外提供服务吧。一般来说，我们有两种方式将 Memcached 和 MySQL 数据库整合成一个整体来对外提供数据服务。一种是直接利用 Memcached 的内存容量作为 MySQL 数据库的二级缓存，提升 MySQL Server 的缓存大小，另一种是通过 MySQL 的UDF 来和 Memcached 进行数据通信，维护和更新 Memcached 中的数据，而应用端则直接通过 Memcached 来读取数据。

对于第一种方式，主要用于业务要求非常特殊，实在难以进行数据切分，而且有很难通过对应用程序进行改造利用上数据库之外的 Cache 的场景。

当然，在正常情况下是肯定无法做到这一点的，之少目前必须借助外界的力量，开源项目 Waffle Grid 就是我们需要借助的外部力量。I

Waffle Grid 是国外的几位 DBA 在工作之余突发奇想出来的一个点子：既然 PC Server 的低廉成本如此的吸引我们，而其 Scale Up 的能力又很难有一个较大的突破，何不利用上现在非常流行的 Memcached 作为突破单台 PC Server 的内存上限呢？就在这个想法的推动下，几位小伙子启动了 Waffle Grid 这个开源项目，利用 MySQL 和 Memcached 双双开源的特性，结合 Memcached 通信协议简单的特点，将 Memcached 成功实现成为 MySQL 主机的外部“二级缓存”，目前仅支持用于 Innodb 的 Buffer Pool。

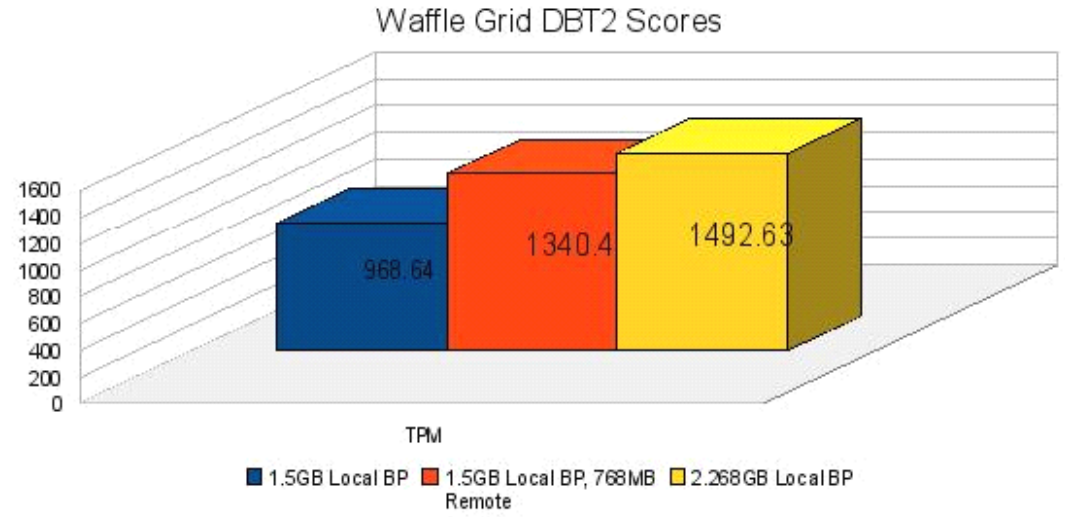
Waffle Grid 的实现原理其实并不复杂，他所做的事情就是当 Innodb 在本地的 Buffer Pool（我们姑且称其为 Local Buffer Pool 吧）的时候，在从磁盘数据文件读取数据之前，先通过 Memcached 的通信 API 接口尝试从 Memcached 中读取相应的缓存数据(我们称之为 Remote Buffer 吧)，只有在 Remote Buffer 中也不存在需要的数据的时候，Innodb 才会访问磁盘文件来读取数据。而且，只有处于 Innodb Buffer pool 中的 LRU List 中的数据会被发送到 Remote Buffer Pool 中，而这些数据一旦被修改，就会 Innodb 就会将之移入 FLUSH List ，Waffle Grid 同时会将进入 FLUSH List 的数据从 Remote Buffer Pool 中清除掉。所以可以说，Remote Buffer Pool 中永远不会存在 Dirty Pages，这也保证了当 Remote Buffer Pool 出现故障的时候不会产生数据丢失的问题。下图是使用 Waffle Grid 项目时候的架构简图：



如架构图上所示，我们首先在 MySQL 数据库端应用 Waffle Grid Patch，通过他连与其他的 Memcached 服务器通信。为了保证网络通信的性能，MySQL 与 Memcached 之间尽可能用高带宽私有网络。

另外，这里的架构图中并没有再将数据库区分 Master 和 Slave 了，并不是说一定不能区分，只是一个示意图。在实际应用过程中，大部分时候只需要在 Slave 上面应用 Waffle Grid 即可，Master 本身并不需要如此大的内存。

看了 Waffle Grid 的实现原理，可能有些读者朋友会有些疑问了。这样做不是所有需要产生物理读的 Query 的性能就会受到直接影响了吗？所有读取 Remote Buffer 的操作都需要通过网络来获取，其性能是否足够高呢？对此，我同样使用作者对 Waffle 的实测数据来接触大家的疑虑：

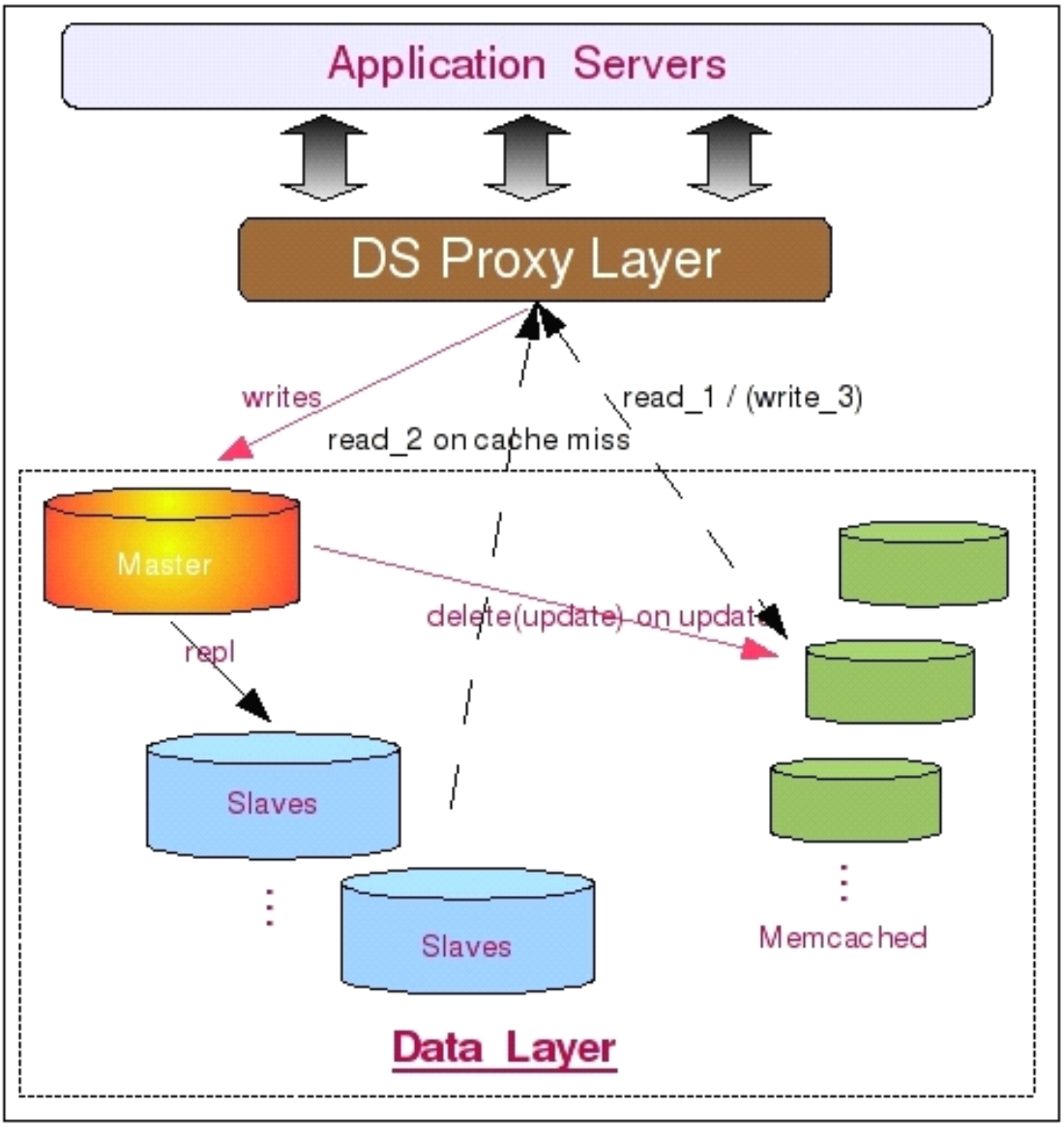


通过 DBT2 所得到的这组测试对比数据，在性能我想并不需要太多的担忧了吧。至于 Waffle Grid 是否适合您的应用场景，那就只能依靠各位读者朋友自己进行评估了。

下面我们再来介绍一下 Memcached 和 MySQL 的另外一种整合方式，也就是通过 MySQL 所提供的 UDF 功能，自行编写相应的程序来实现 MySQL 与 Memcached 的数据通信更新操作。

这种方式和 Waffle Grid 不一样的是 Memcached 中的数据并不完全由 MySQL 来控制维护，而是由应用程序和 MySQL 一起来维护数据。每次应用程序从 Memcached 读取数据的时候，如果发现找不到自己需要的数据，则再转为从数据库中读取数据，然后将读取到的数据写入 Memcached 中。而 MySQL 则控制 Memcached 中数据的失效清理工作，每次数据库中有数据被更新或者被删除的时候，MySQL 则通过用户自行编写的 UDF 来调用 Memcached 的 API 来通知 Memcached 某些数据已经失效并删除该数据。

基于上面的实现原理，我们可以设计出如下这样的一个数据服务层架构：



如图中所示，此架构和上面将 Memcached 完全和 MySQL 读离开作为常规的 Cache 服务器来比较，最大的区别在于 Memcached 的数据变为由 MySQL 数据库来维护更新，而不是应用程序来更新。首先数据被应用程序写入 MySQL 数据库，这时候将会触发 MySQL 上面用户自行编写的相关 UDF，然后通过该 UDF 调用 Memcached 的相关通信接口，将数据写入 Memcached。而当 MySQL 中的数据被更新或者删除的时候，MySQL 中的相关 UDF 同样会更新或者删除 Memcached 中的数据。当然，我们也可以让 MySQL 做更少一些的事情，仅仅只是遇到数据被更新或者删除的时候，通过 UDF 来删除 Memcached 中的数据，写入工作则像前面的架构一样由应用程序来作。

由于 Memcached 基于对象的数据存取，以及通过 Hash 进行数据检索的特性，所以所有存储在 Memcached 中的数据都需要我们设定一个用于标识该数据的 Key，所有数据的存取操作都通过该 Key 来进行。也就是说，如果您并不能像 MySQL 的 Query 语句一样通过某一个（或者多个）关键字条件来读取包含多条数据的结果集，仅适用于通过某个唯一键来获取单条数据的数据读取方式。

15.2.2 嵌入式数据库编程库 Berkeley DB

说实话，数据库编程库这个叫法实在有些别扭，但我也实在找不到其他合适的名词来称呼 Berkeley DB 了，那就姑且使用网上较为通用的叫法吧。

Memcached 所实现的是内存式 Cache，如果我们对性能的要求并没有如此之高，在预算方面也不是太充裕的话，我们还可以选择 Berkeley DB 这样的数据库型 Cache 软件。可能很多读者朋友又会产生疑惑了，我们使用的 MySQL 数据库，为什么还要再使用一个 Berkeley DB 这样的“数据库”呢？实际上 Berkeley DB 在之前也是 MySQL 的存储引擎之一，只不过后期不知道是何原因（获取与商业竞争有关吧），被 MySQL 从支持的存储引擎中移除了。之所以在使用数据库的同时还使用 Berkeley DB 这样的数据库型 Cache，是因为我们可以充分发挥出二者各自的优势，在使用传统通用型数据库的同时，同时可以利用 Berkeley DB 高效的键值对存储方式作为高效数据检索的性能补充，以得到更好的数据服务层扩展性和更高的整体性能。

Berkeley DB 自身架构可以分为五个功能模块，五个模块的在整个系统中相对比较独立，而且可以设置使用或者禁用某一个（或者几个）模块，所以可能称之为五个子系统会更为恰当一些。这五个子系统及基本介绍分别如下：

- 数据存取

  数据存取子系统主要负责最主要也是最基本的数据存与取的工作。而且 Berkeley DB 同时支持了以下四种数据的存储结果方式：Hash，B-Tree，Fixed Length 以及 Dynamic Length。实际上，这四种方式对应了四种数据文件存储的实际格式。数据存储子系统可以完全单独使用，也是必须开启的一个子系统。

- 事务管理

  事务管理子系统主要是针对有事务要求的数据处理服务，提供完整的 ACID 事务属性。在开启事务管理子系统的时候，出了需要开启最基本的数据存取子系统外，还至少需要开启锁管理子系统和日志系统来帮助实现事务的一致性和完整性。

- 锁管理

  锁管理系统主要就是为了保证数据的一致性而提供的共享数据控制功能。支持行级别和页级别的锁定机制，同时为事务管理子系统提供服务。

- 共享内存

  共享内存子系统我想大家看到名称就应该基本知道是做什么事情的了，就是用来管理维护共享 Cache 和 Buffer 的，为系统提升性能而提供数据缓存服务。

- 日志系统

  日志系统主要服务于事务管理系统，为保证事务的一致性，Berkeley DB 也采用先写日志再写数据的策略，一般也都是与事务管理系统同时使用同时关闭。

基于 Berkeley DB 的特性，我们很难像使用 Memcached 那样将他和 MySQL 数据库结合的那么紧密。数据的维护与更新操作主要还是需要通过应用程序来完成。一般来说，在使用 MySQL 的同时还要使用 Berkeley DB 的主要原因就是为了提升系统的性能及扩展性。所以，大多数时候都主要是使用 Hash 和 B-Tree 这两种结构的数据存储格式，尤其是 Hash 格式，是使用最为广泛的，因为这种方式也是存取效率最高的。

在应用程序中，每次数据请求，都先通过预先设定的 Key 到 Berkeley DB 中取查找一次，如果存在数据，则返回取得的数据，如果位检索到数据，则再次到数据库中读取。然后将读取到的数据按照预先设定的 Key，整条存入 Berkeley DB 中，再返回给客户端。而当发生数据修改的时候，应用程序在修改 MySQL 中的数据之后必须还要将 Berkeley DB 中的数据删除。当然，如果您愿意，也可以直接修改 Berkeley DB 中的数据，但是这样就可能引入更多的数据一致性风险并提高系统复杂度了。

从原理来看，使用 Berkeley DB 的方式和将 Memcached 作为纯 Cache 来使用差别不大嘛，为什么我们不用 Memcached 来做呢？其实主要有两个原因，一个是 Memcached 是使用纯内存来存放数据的，而 Berkeley DB 则可以使用物理磁盘，两者在成本方面还是有较大差别的。另外一个原因就是 Berkeley DB 所能支持的数据存储方式除了 Memcached 所使用的 Hash 存储格式之外，同时还可以使用其他存储格式，如 B-Tree 等。

由于和 Memcached 的基本使用原理区别不大，所以这里就不再画图示意了。

15.3 自行实现 Cache 服务

实际上，除了使用比较成熟的现成第三方软件的解决方案之外，如果有一定的技术实力，我们还可以通过自行实现的 Cache 软件来达到完全相同的效果。

当然，您也不要被上面所说的“技术实力”所吓倒，其实也并没有想象中的那么难。只要您不要一开始就希望作出一个能够解决所有问题，而且包含所有其他第三方 Cache 软件的所有优点，还不能遗留任何缺点的软件，不要一开始就希望作出一个多么完美的产品花的软件。从小做起，从精做起。千万别希望一口气吃成一个胖子，这样的解决很可能就是被咽死。

自主研发实现 Cache 服务软件的前提是系统中存在比较特殊的应用场景，通过自主研发可以最大限度的实现比较个性化的需求。当然，也可以针对自己的应用场景进行特定的优化方式来最大限度的提升扩展性和性能。毕竟，只有我们自己才是真正最了解我们的应用系统的人。

决策是否需要自行开发最需要考虑的一个问题就是我的英勇系统场景是否真的如此特别，以至于现成的第三方软件很难解决目前的主要问题？

如果目前的第三方软件已经基本解决了我们系统当前遇到的 80%以上的问题，可能就需要考虑是否有必要完全自主研发了。毕竟我们选择的所有第三方软件都是开源的，如果有某

些小地方无法满足要求，我们完全可以在第三方软件的基础上增加一些我们自己的东西，来满足一些个性化需求。

当我们选择自主研发 Cache 服务软件之后，有以下几点内容是需要注意的：

1. 功能需求
   a) 是完全内存还是可以部分磁盘？
   b) 需要实时同步更新还是可以允许 Cache 数据有延时？
   c) 是否需要支持分布式？

这里所说的功能，实际上就是需求范围的设定。在开始研发之前，我们比需要有一个非常清晰的需求范围，而不是天马行空的边开发边调整，想到啥做啥。毕竟任何软件系统，都是需要以第一线的需求为导向，而且一旦开始开发之后，需求的控制也不能马虎。要不然，很可能就会中途夭折，以失败而告终。

2. 技术实现
   a) 数据同步（或异步）更新机制；
   b) 数据存储方式（Hash Or B-Tree）；
   c) 通讯协议；

技术实现可能会成为研发过程中很大的一个难点，能否有稳定可靠的数据同步（或异步）更新机制决定了该 Cache 软件最终的成败。当然，你可以说数据同步（或异步）更新完全交由需要访问数据的应用程序来自行维护，但是你是否有足够的能力申请在自行研发实现出一个 Cache 软件的同时，还需要前端应用程序作出巨大的调整来适应这个 Cache 软件是一个很大的未知数。老板很可能会说，既然你都自行研发实现了，为啥不能完成数据更新维护功能呢？而数据存储方式直接决定数据的访问方式，同时实现算法也直接决定了软件的性能。最后，数据传输的通讯协议可能也会让人伤透脑筋。如何设计一个足够简单，但是又做到尽可能不会限制后期的扩展升级的通讯协议，可能并不是一件太轻松的事情。毕竟，如果每次升级都需要动到数据传输通讯协议，那每次升级所带来的应用改造成本也太大了。而太过复杂呢，很可能又会影响到前端应用使用的便利性，而且对性能可能也会有一定影响。

3. 可维护性
   a) 方便的管理接口；
   b) 高可用支持（自动或人工切换）；
   c) 基本监控接口；

千万不要忽视了软件系统的维护成本，一个软件一旦开始使用之后，主要工作就是对其进行各种维护。如果可维护性太差，很可能带来极大的维护工作量，甚至带来一线应用人员和运维人员对该软件的信任和使用热情。

使用自行研发的 Cache 服务基于不同的功能特性，可能会有不同的架构组成，但基本上和上面使用 Memcached 所使用的架构区别不大了，所以这里也就不再详细讨论了。

最后，我个人有一个建议就是，在使用比较通用 Cache 服务（也包括自行实现的 Cache 软件服务）的时候，我们应该尽可能将该 Cache 软件与我们的 MySQL 数据库 进行一定的

整合，让彼此能够互补。而且前端的应用程序尽量不要直接操作后端的数据服务集群，尽量通过一个中间代理层来接受处理所有的数据处理服务，对前端应用透明化。这样才能够尽可能做到后端数据服务（数据库与 Cache）层在进行任何扩展的时候，影响到的仅仅只是中间代理层，而对前端完全透明，让我们的数据层拥有真正的高扩展性。

15.4 利用 Search 实现高效的全文检索

不论是使用 Memcached 还是使用 Berkeley DB ，大多数时候都只能通过特定的方式来进行数据的检索，只能满足少部分的检索需求。而数据库本身对于全模糊 LIKE 操作的性能大家应该也很清楚，是非常低下的，因为这种操作无法利用索引。虽然 MySQL 的 MyISAM 存储引擎支持了全文索引，而且官方版本还不支持多字节字符集的数据，所以对于需要存放中文或者需要使用 MyISAM 之外的存储引擎的用户来说，是完全无法使用的。

对于这种情况，我们只有一个办法可以解决，那就是通过全文索引软件，也就是我们常说的 Search（搜索引擎）对数据进行全文索引，才能达到较为高效的数据检索效率。

同样，Search 软件的使用也有使用较为成熟的第三方解决方案与自行研发两种方式。目前最为有名的第三方解决方案主要就是基于 Java 实现的 Lucene，隶属于 Apache 软件基金 Jakarta 项目组下面的一个子项目。当然，他并不是一个完整的搜索引擎工具，而是一个全文检索引擎的框架，他同时提供了完整的用于检索的查询引擎和数据索引引擎。

这里我就不深入讨论 Lucene 本身的技术细节了，感兴趣的读者朋友可以通过访问官方站点（http://lucene.apache.org） 来了解更多也更为权威的细节。我这里主要是介绍一下 Luence 能够给我们带来什么，我们可以怎样来使用他。

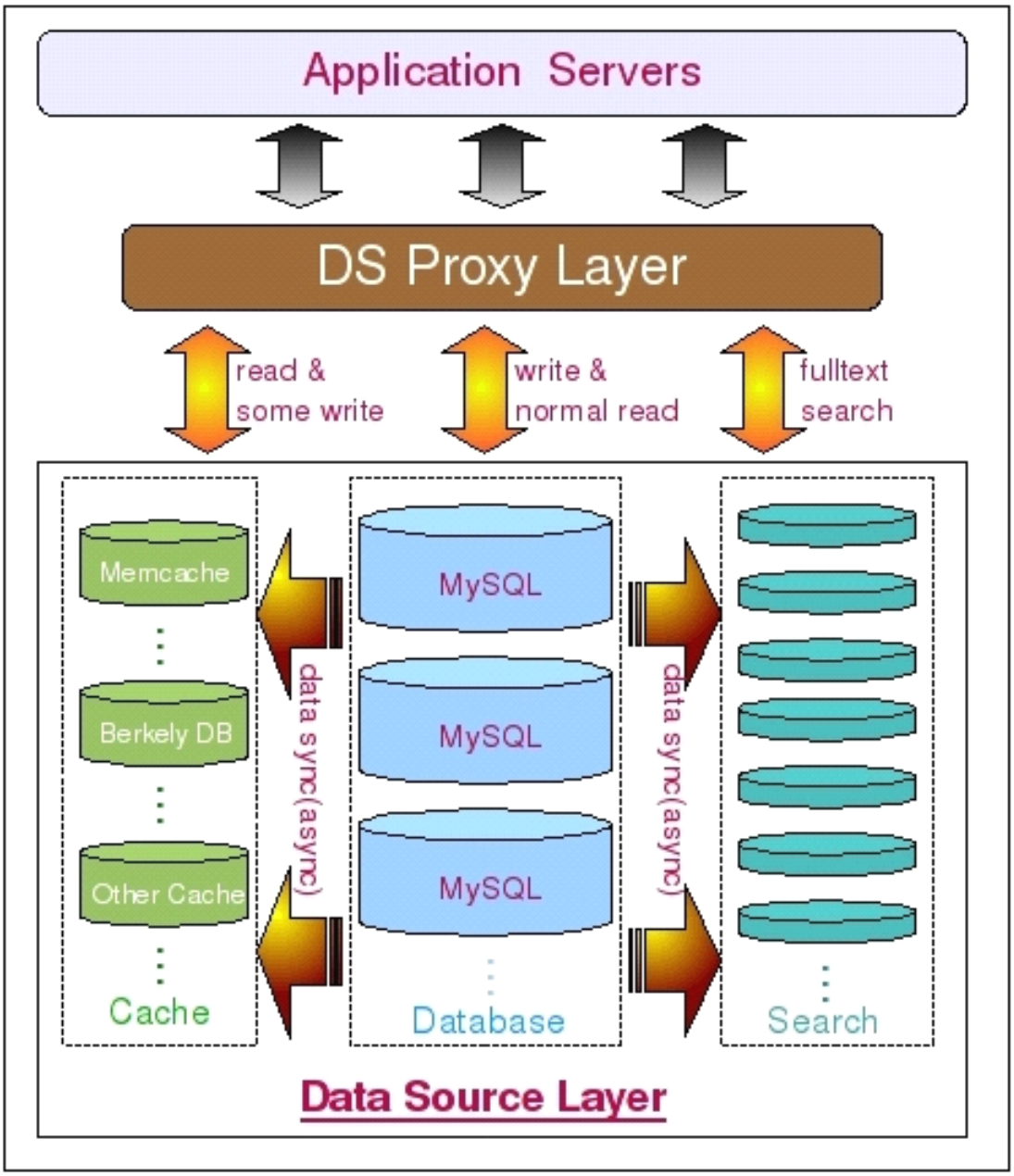
由于 Lucene 高效的全文索引和分词算法，以及高效的数据检索实现，我们完全可以很好的利用这一优点来解决数据库和传统的 Cache 软件完全无法解决的全文模糊搜索功能。我们的需求和传统的通用全网搜索引擎并不一样，并不需要“Spider”到处去爬取互联网上面的数据，只需要将我们数据库中被持久化下来的数据通过应用程序调用 Lucene 的相关 API 写入，并利用 Lucene 创建好索引，然后就可以通过调用 Lucene 所提供的数据检索 API 得到需要访问的数据，而且可以进行全模糊匹配。由于从数据库到 Lucene 这一过程完全由我们自己来实现，所以我们非常容易控制数据的实时性。可以做到完全实时，同样也可以做到固定（或动态）时间段刷新。

虽然 Lucene 的数据也是存放在磁盘上而不是内存中，但是由于高效的分词算法和索引结构，其效率也是非常的好。看到很多网友在网上讨论，当数据量稍微大一些如几十个 G 之后 Lucene 的效率会下降的非常快，其实这是不科学的说法，就从我亲眼所见的场景中，就有好几百 G 的数据在 Lucene 中，性能仍然很出色。这几年性能优化的工作经历及经验中我有一个很深的体会，那就是一个软件性能的好坏，实际上并不仅仅只由其本身所决定，很多时候一个非常高效的软件不同的人使用会有截然不同效果。所以，很多时候当我们使用的第三方软件性能出现问题的时候，不要急着下结论认为是这个软件的问题，更多的是先从自身找找看我们是否真的正确使用了他。

除了使用第三方的 Search 软件如 Lucene 之外，我们也可以自行研发更适用于我们自身应用场景的 Search 软件。就像我目前所供职的公司一样，自行研发了一套纯内存存储的更适合于自身应用场景的高性能分布式 Search 软件，让各个应用系统能够作出很多高效的更为个性化的特色功能。通过多年的技术和经验的积累，现在都已经发展成为和数据库并列的另一个应用系统数据源了。

当然，自行研发 Search 软件的技术门槛可能也比较高，有此技术实力的开发团队并不是很多，所以在决定自行研发之前，一定要做好各方面的评估。不过，如果我们无法实现一个很通用的 Search 软件，但是仅仅只是针对某些特定功能来说，可能实现也并没有想象的那么复杂，更何况如今的开源世界里各种各样的软件数不胜数，利用现有的工具，加上自身个性化定制的二次开发，对于有些特定功能的实现可能就会比较轻松了。

加入了 Search 软件来实现高效的全文检索功能之后，我们的架构可以通过如下这张图来展示：



15.5 利用分布式并行计算实现大数据量的高性能运算

说到大规模大数据量的高性能运算的时候，可能很多人都会想到最近风靡整个 IT 界的一个关键词：云计算，亦或是几年前的“网格计算”。

曾经有朋友建议我将本节标题中的“分布式并行计算”更改为“云计算”，考虑再三之后还是没有更改。说实话，从我个人的理解，不论是“云计算”还是“网格计算”，其实其实质都是一样，都是“分布式并行计算”，只不过是各个商业公司为了吸引大家的眼球所玩的一些“概念游戏”而已，个人认为纯属商业行为。当然，可能有人会认为从“分布式并行计算”到“网格计算”再到“云计算”，每一次“升级”都是在可以利用的计算资源上面有

所扩展。可这种扩展都是在概念上面的扩展，而真正技术实现方面所依靠的并不是这些概念，而是各种软硬件的发展。更何况，最初的“分布式并行计算”概念中本就没有限定我们只能以哪种方式使用哪些资源。

目前比较流行的分布式并行计算框架主要就是以 Google 的 MapReduce 和 Yahoo 的 Hadoop 二者。其实更为准确的说应该是 Google 的 MapReduce + GFS + BigTable 以及 Yahoo 的 Hadoop + HDFS + HBase 这两大架构体系。二者都是由三个负责不同功能的组件组成，MapReduce 与 Hadoop 同为解决认任务分解与合并的功能，GFS 与 HDFS 都是分布式文件系统，解决数据存储的基础设施问题，最后 BigTable 与 HBase 则同为处理结构化数据存储格式的类数据库模块。三大模块共同协作，最终组成一个分布式并行计算的框架体系整体。

其实这分属于两家互联网巨头的分布式并行计算架构框架体系的实现原理基本上可以说是完全一样的。通过前面端的任务分解合并引擎将计算（或者数据存取）任务分解成多个任务，同时发送给多台计算（或数据）节点来进行计算，而后面的每一个节点利用分布式文件系统来作为存储计算数据的基础平台，当然，不论是计算前还是计算后的数据，都是通过 BigTable 或者是 Hbase 这样的模块进行组织。三者组成一个完整的整体，相互依赖。当然，不得不说的是 Hadoop 本身也是由 Google 最初的一篇关于 MapReduce 的论文原理为思想所开发出来的。只不过 Google 在 Open 思想方面所做的贡献很多时候仅限于论文形式，对于其自身技术架构方面的信息公开的实在是有些少。

其实除了这两个重量级的分布式计算框架之外，完全利用现有开源数据库实现的完整解决方案也有一些，如 Inforbright 与 MySQL 合作实现的 BI 解决方案，Greenplum 公司与 Sun 利用 PostGresql 开源数据库实现的 Greenplum 系统，而且两个系统都是依赖 MapReduce 理论所实现。

虽然这两个系统目前并没有前面两个分布式计算框架那样大的伸缩性，但是所针对的场景是实实在在的 DB 场景，数据访问接口完全实现了 SQL 规范。对于使用习惯了通过 SQL 语句来玩数据库的分析方法来说，无疑是非常有诱惑力的，更何况这两个系统基本上都不怎么需要开发，已经是一个完整的产品了。

考虑到篇幅以及非本书重点所在，这里就不深入讨论相关的技术细节了，大家如果对这些内容比较感兴趣，可以通过各自官方网站了解更多更为全面的信息。

15.6 小结

数据库只是存储数据的一种工具，其特殊性只是能将数据持久化，且提供统一规范的访问接口而已。除了数据库，其实我们还可以很多其他的数据存储处理方式，结合各种数据存储处理方式，充分发挥各自的特性，扬长避短，形成一个综合的数据中心，这样才能让系统的数据处理系统的扩展性得到最大的提升，性能得到最优化。

第 16 章 MySQL Cluster

前言：

MySQL Cluster 是一个基于 NDB Cluster 存储引擎的完整的分布式数据库系统。不仅仅具有高可用性，而且可以自动切分数据，冗余数据等高级功能。和 Oracle Real Cluster Application 不太一样的是，MySQL Cluster 是一个 Share Nothing 的架构，各个 MySQL Server 之间并不共享任何数据，高度可扩展以及高度可用方面的突出表现是其最大的特色。虽然目前还只是 MySQL 家族中的一个新兴产品，但是已经有不少企业正在积极的尝试使用了。本章我们将通过对 MySQL Cluster 的了解来寻找其在可扩展设计方面的优势。

16.1 MySQL Cluster 介绍

简单的说，MySQL Cluster 实际上是在无共享存储设备的情况下实现的一种完全分布式数据库系统，其主要通过 NDB Cluster（简称 NDB）存储引擎来实现。MySQL Cluster 刚刚诞生的时候可以说是一个可以对数据进行持久化的内存数据库，所有数据和索引都必须装载在内存中才能够正常运行，但是最新的 MySQL Cluster 版本已经可以做到仅仅将所有索引装载在内存中即可，实际的数据可以不用全部装载到内存中。

一个 MySQL Cluster 的环境主要由以下三部分组成：

a) SQL 层的 SQL 服务器节点(后面简称为 SQL 节点)，也就是我们常说的 MySQL Server。

主要负责实现一个数据库在存储层之上的所有事情，比如连接管理，Query 优化和响应，Cache 管理等等，只有存储层的工作交给了 NDB 数据节点去处理了。也就是说，在纯粹的 MySQL Cluster 环境中的 SQL 节点，可以被认为是一个不需要提供任何存储引擎的 MySQL 服务器，因为他的存储引擎有 Cluster 环境中的 NDB 节点来担任。所以，SQL 层各 MySQL 服务器的启动与普通的 MySQL Server 启动也有一定的区别，必须要添加 ndbcluster 参数选项才行。我们可以添加在 my.cnf 配置文件中，也可以通过启动命令行来指定。

b) Storage 层的 NDB 数据节点，也就是上面说的 NDB Cluster。

最初的 NDB 是一个内存式存储引擎，当然也会将数据持久化到存储设备上。但是最新的 NDB Cluster 存储引擎已经改进了这一点，可以选择数据是全部加载到内存中还是仅仅加载索引数据。NDB 节点主要是实现底层数据存储功能，来保存 Cluster 的数据。每一个 Cluster 节点保存完整数据的一个 fragment，也就是一个数据分片（或者一份完整的数据，视节点数目和配置而定），所以只要配置得当，MySQL Cluster 在存储层不会出现单点的问题。一般来说，NDB 节点被组织成一个一个的 NDB Group，一个 NDB Group 实际上就是一组存有完全相同的物理数据的 NDB 节点群。

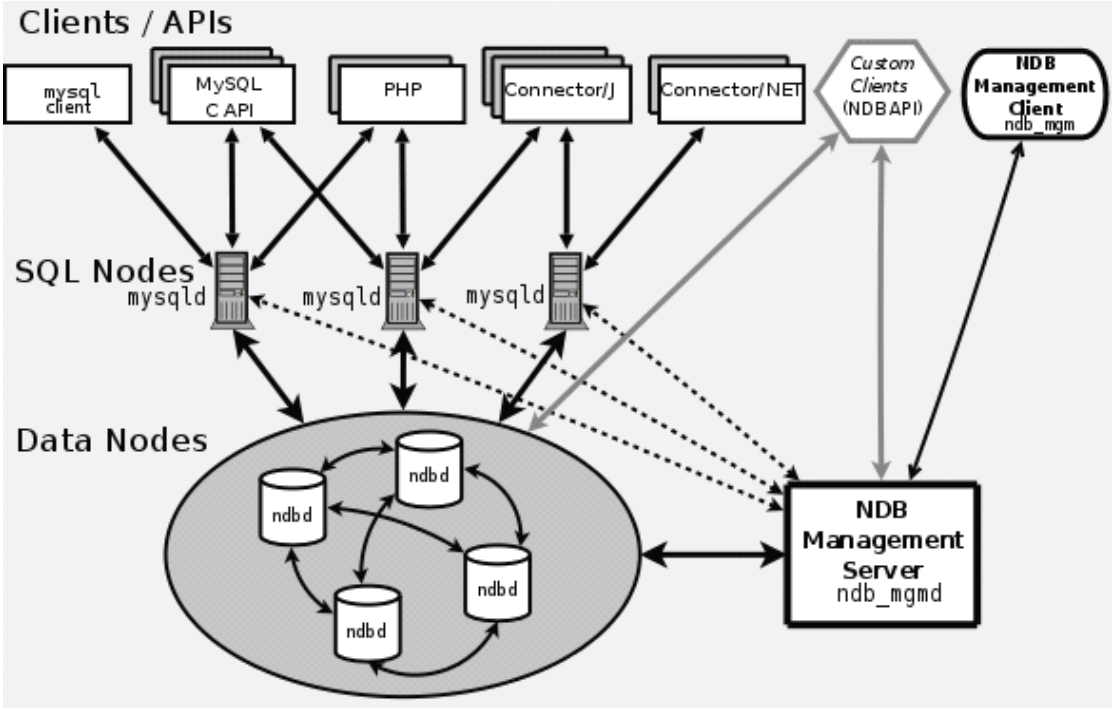
上面提到了 NDB 各个节点对数据的组织，可能每个节点都存有全部的数据也可能只保存一部分数据，主要是受节点数目和参数来控制的。首先在 MySQL Cluster 主配置文件（在管理节点上面，一般为 config.ini）中，有一个非常重要的参数叫 NoOfReplicas，这个参数指定了每一份数据被冗余存储在不同节点上面的份数，该参数一般至少应该被设置成 2，也只需要设置成 2 就可以了。因为正常来说，两个互为冗余的节点同时出现故障的概率还是非常小的，当然如果机器和内存足够多的话，也可以继续增大来更进一步减小出现故障的概率。此外，一个节点上面是保存所有的数据还是一部分数据还受到存储节点数目的限制。NDB 存储引擎首先保证 NoOfReplicas 参数配置的要求来使用存储节点，对数据进行冗余，然后再

根据节点数目将数据分段来继续使用多余的 NDB 节点。分段的数目为节点总数除以 NoOfReplicas 所得。

c) 负责管理各个节点的 Manage 节点主机：

管理节点负责整个 Cluster 集群中各个节点的管理工作，包括集群的配置，启动关闭各节点，对各个节点进行常规维护，以及实施数据的备份恢复等。管理节点会获取整个 Cluster 环境中各节点的状态和错误信息，并且将各 Cluster 集群中各个节点的信息反馈给整个集群中其他的所有节点。由于管理节点上保存了整个 Cluster 环境的配置，同时担任了集群中各节点的基本沟通工作，所以他必须是最先被启动的节点。

下面是一幅 MySQL Cluster 的基本架构图（出自 MySQL 官方文档手册）：



通过图中我们可以更清晰的了解整个 MySQL Cluster 环境各个节点以及客户端应用之间的关系。

由于 MySQL Cluster 目前的成熟使用并不是太多，实现也较普通的 MySQL 略复杂，所以本章将首先从如何搭建一个 MySQL Cluster 环境开始来介绍他。

16.2 MySQL Cluster 环境搭建

搭建 MySQL Cluster 首先需要至少一个管理节点主机来实现管理功能，一个 SQL 节点主机来实现 MySQL server 功能和两个 ndb 节点主机实现 NDB Cluster 的功能。在后面的介绍中，我采用双 SQL 节点来搭建测试环境，具体信息如下：

1、硬件准备

a) MySQL 节点 1　　192.168.0.1

b) MySQL 节点 2　　192.168.0.2
c) ndb 节点 1　　192.168.0.3
d) ndb 节点 2　　192.168.0.4
e) 管理节点　　192.168.0.5

2、软件安装

首先在上面 5 个节点的主机上尽量确保环境基本一致，然后从 MySQL 官方下载相应的软件包并分发到两台 SQL 节点和两台 NDB 节点上，以备后面的安装时候的使用。

我的测试环境 OS（RedHat Linux）如下（非必须）：

root@mysql1:/usr/local>uname -a

Linux oratest1 2.6.9-42.ELsmp #1 SMP Wed Jul 12 23:27:17 EDT 2006 i686 i686 i386 GNU/Linux

a) 安装 MySQL 节点：

在 MySQL 节点上面需要安装支持 cluster 的 MySQL Server，可以通过自编译源代码安装也可以选择 MySQL 官方提供的编译好的 tar 包或者 rpm 安装包，我是通过源代码自行编译的，实际上完全可以通过 MySQL 官方提供的经过优化编译的二进制 tar 包，只是我自己习惯了而已，我的编译设置参数如下：

```
root@mysql1>./configure \
--prefix=/usr/local/MySQL \
--without-debug \
--without-bench \
--enable-thread-safe-client \
--enable-assembler \
--with-charset=utf8 \
--with-extra-charsets=complex \
--with-client-ldflags=-all-static \
--with-MySQLd-ldflags=-all-static \
--with-ndbcluster \
--with-server-suffix=-max \
--datadir=/data/mysqldata \
--with-unix-socket-path=/usr/local/MySQL/sock/mysql.sock
...

root@mysql1>make
...

root@mysql1>make install
...
```

然后是配置设置配置文件/etc/my.cnf，由于是测试环境，所以我仅仅设置了 ndbcluster 所需要的最基本的两个配置项，其他所有的配置均用默认配置（后面会有较为详细的配置说明），如下：

root@mysql1>vi /etc/my.cnf
[client]
socket = /usr/local/mysql/sock/mysql.sock #由于编译时候特殊指定了,所以设置在这里，方便以后登入的时候使用
[MySQLd]
socket = /usr/local/mysql/sock/mysql.sock
ndbcluster

[MySQL_cluster]
ndb-connectstring = 192.168.0.5

继续完成后面的 MySQL 安装过程：
root@mysql1>cd /usr/local/mysql
root@mysql1>bin/mysql_install_db --user=mysql --socket=/usr/local/mysql/sock/mysql.sock
Installing MySQL system tables...
OK
Filling help tables...
OK

To start MySQLd at boot time you have to copy
support-files/MySQL.server to the right place for your system

PLEASE REMEMBER TO SET A PASSWORD FOR THE MySQL root USER !
To do so, start the server, then issue the following commands:
/usr/local/mysql/bin/MySQLadmin -u root password 'new-password'
/usr/local/mysql/bin/MySQLadmin -u root -h ointest_stb password 'new-password'

Alternatively you can run:
/usr/local/mysql/bin/MySQL_secure_installation

which will also give you the option of removing the test
databases and anonymous user created by default. This is
strongly recommended for production servers.

See the manual for more instructions.

You can start the MySQL daemon with:
cd /usr/local/MySQL ; /usr/local/mysql/bin/MySQLd_safe &

You can test the MySQL daemon with MySQL-test-run.pl
cd MySQL-test ; perl MySQL-test-run.pl

```
Please report any problems with the /usr/local/mysql/bin/MySQLbug script!

The latest information about MySQL is available on the web at
http://www.mysql.com
Support MySQL by buying support/licenses at http://shop.mysql.com

root@mysql1>chown -R root  .
root@mysql1>chgrp -R mysql .
root@mysql1>chown -R mysql.mysql /usr/local/mysql/etc
root@mysql1>chown -R mysql.mysql /usr/local/mysql/sock
root@mysql1>chown -R mysql.mysql /usr/local/mysql/log
root@mysql1>:/usr/local/mysql# ls -l
total 40
drwxr-xr-x    2 root     MySQL        4096 May  4 14:47 bin
drwxr-xr-x    2 MySQL    MySQL        4096 May  4 14:20 etc
drwxr-xr-x    3 root     MySQL        4096 May  4 14:46 include
drwxr-xr-x    2 root     MySQL        4096 May  4 14:46 info
drwxr-xr-x    3 root     MySQL        4096 May  4 14:46 lib
drwxr-xr-x    2 root     MySQL        4096 May  4 14:47 libexec
drwxr-xr-x    2 MySQL    MySQL        4096 May  4 14:20 log
drwxr-xr-x    4 root     MySQL        4096 May  4 14:47 man
drwxr-xr-x    9 root     MySQL        4096 May  4 14:47 MySQL-test
drwxr-xr-x    2 MySQL    MySQL        4096 May  5 22:16 sock
root@mysql1>:/usr/local/mysql#
```

b) 安装 ndb 节点：

如果希望尽可能的各环境保持一致，建议在 NDB 节点也和 SQL 节点一样安装整个带有 NDB Cluster 存储引擎的 MySQL Server。由于安装细节和上面的 SQL 节点完全一样，所以这里就不再累述。

另外，如果只是为了保证能够完整的 MySQL Cluster 这个环境，则在 NDB 节点上完全可以仅安装 NDB 存储引擎（mysql ndb storage engine）即可。安装 NDB 存储引擎好像目前是找不到源码来自行编译安装的，只能通过 MySQL AB 官方提供的 rpm 包来安装。安装过程非常简单，和其他的 rpm 软件包安装没有任何区别。

c) 管理节点：

管理节点所需要的安装更简单，实际上只需要 ndb_mgm 和 ndb_mgmd 两个程序即可，这两个可执行程序可以在上面的 MySQL 节点的 MySQL 安装目录中的 bin 目录下面找到。将这两个程序 copy 到管理节点上面合适的位置（自行考虑，我一般会放在 /usr/local/mysql/bin 下面），并在 path 制定的目录中建立两个同名的 soft link 在到这两个程序上面，就可以了。

以上即是 MySQL Cluster 环境的软件安装过程，看上去并不复杂是吧，希望大家的安装过程也能够一切顺利，当然如果遇到了什错误也不用担心，MySQL 官方手册中也提供了非常详细的安装过程说明。

3、基本配置

在上面所有节点的软件安装完成之后，就是MySQL Cluster环境的配置工作了。如果不考虑其他一些优化和个性化的配置需求，MySQL Cluster的基本配置是比较简单的。这里暂时先仅仅完成一个简单的测试环境的配置，详细的配置说明请看后面的MySQL Cluster配置介绍的章节。

对于MySQL节点和ndb节点在上面的安装过程中已经完成了，仅需要设置[MySQL_cluster]参数组的ndb-connectstring参数即可完成最基本的配置。

管理节点的配置稍微复杂一点，因为他需要配置出Cluster环境中每一个节点的基本信息。配置文件并不需要一个特别固定的位置和名称，都由用户自行设定，只需要在启动过程中指定配置文件即可。在我们的测试环境中配置为建名称为/var/lib/MySQL-cluster/config.ini，内容如下：

```
[root@mysqlMgm ~]# cat  /var/lib/mysql-cluster/config.ini
[NDBD DEFAULT]
NoOfReplicas=2
DataMemory=64M
IndexMemory=16M

[TCP DEFAULT]
portnumber=2202

#管理节点
[NDB_MGMD]
id=1
hostname=192.168.0.5
datadir=/var/lib/mysql-cluster

#第一个ndbd节点:
[NDBD]
id=2
hostname=192.168.0.3
datadir=/data/mysqldata

#第二个ndbd节点:
[NDBD]
id=3
hostname=192.168.0.4
datadir=/drbddata/mysqldata

# SQL node options:
[MySQLD]
id=4
hostname=192.168.0.1
```

```
[MySQLD]
id=5
hostname=10.0.65.203
[root@mysqlMgm ~]#
```

1) SQL 节点的配置:

MySQL 节点的配置和普通的 MySQL Server 的配置区别主要是需要在 my.cnf 文件中增加[mysql_cluster]这个配置选项组，并至少指定 ndb-connectstring=192.168.0.5，也就是制定管理节点的 ip 地址或者 hostname。另外，如果希望能在启动 MySQLd 的时候不用手动指定 ndbcluster 参数，则在[mysqld]参数选项组中增加 ndbcluster 项参数。除了这两项之外，其他的所有参数都可以可以使用默认值。

2) NDB 存储节点的配置：

NDB 存储节点的配置就更简单的了，仅仅需[mysql_cluster]中的 ndb-connectstring = 192.168.0.5 参数，其他所有的都可以不再配置了。

4、环境测试

在 MySQL Cluster 环境搭建完成后，首先肯定要对新搭建的环境进行一些基本的功能和异常测试，以确认搭建的环境是否已经可以正常提供服务。

1) 首先检测 ndb 引擎是否已经正常工作

通过任意客户端连接任意选定的一个 SQL 节点，测试各种基本的 ddl，dml 操作，然后再通过客户端连接上 Cluster 环境中另外的 SQL 节点校验所作的草食是否在其他节点同样可见了。下面是测试 create table 后再插入一条数据的示例：

在节点 4 上面：

```
mysql>use test;
mysql>create table t1 ( a int) engine=ndb;
Query ok, 0 rows affected (0.00 sec)
mysql>insert into t1 values(100);
Query ok, 1 rows affected (0.00 sec)
```

然后在节点 5 上面：

```
mysql>use test;
mysql>select * from t1;
+-----+
| id  |
+-----+
| 100 |
+-----+
1 row in set (0.00 sec)
```

可见，在节点 4 上面所插入的数据，已经在节点 5 上面了，说明 ndb 引擎工作正常的。其他的测试与此类似，大家可以自行测试。

如果在测试中发现在某两个节点之间出现不一致现象，那么可以肯定的是，Cluster 环境的配置有问题。在管理节点上面通过“ndb_mgm -e SHOW”命令查看各节点状态是否正

常，是否都已经连接到了管理节点上面。并检查不正常节点的 my.cnf 配置文件，是否已经配置好了以 ndbcluster 方式启动 MySQLd，是否有正确配置[mysql_cluster]这个参数组的最基本的 ndb-connectstring 参数。然后检查管理节点上面的 config 文件，里面是否有正确配置好各所有节点的配置，尤其是不正常的 SQL 节点的配置。

2) 检测冗余环境的单点故障问题

a、模拟 NDB 节点 Crash

由于是模拟 Crash，所以我们通过在节点 2 上面 kill 掉 ndb 进程，然后再分别通过两个 SQL 节点去访问 t1 表，查看是否可以正常访问，数据是否一样。

在节点 4 上面：

```
mysql> use test;
mysql> select * from t1;
+-----+
| id  |
+-----+
| 100 |
+-----+
1 row in set (0.00 sec)
mysql> insert into t1 values(200);
Query ok, 1 rows affected (0.00 sec)
```

在节点 5 上面：

```
mysql>use test;
mysql>select * from t1;
+-----+
| id  |
+-----+
| 100 |
| 200 |
+-----+
2 row in set (0.00 sec)
mysql> delete from t1 where id = 100;
Query ok, 1 rows affected (0.00 sec)
```

再回到节点 4 上面：

```
mysql> select * from t1;
+-----+
| id  |
+-----+
| 200 |
+-----+
1 row in set (0.00 sec)
```

可以看到，不仅 t1 仍然可以正常访问，数据也没有任何丢失，且仍然可以正常插入，删除数据。可见，在有一个 NDB 节点 Crash 之后，真个 MySQL Cluster 环境仍然可以正

常提供服务。当然，如果两个NDB节点都Crash之后，MySQL Cluster环境就无法正常提供服务了，大家也可以自行测试一下。

b、模拟SQL节点Crash

同样和测试NDB节点Crash一样，kill掉一个SQL节点（比如节点4）的mysqld进程，然后通过节点5进行访问：

在节点5上面：

```
mysql> use test;
mysql> select * from t1;
+-----+
| id  |
+-----+
| 200 |
+-----+
1 row in set (0.00 sec)
mysql> insert into t1 values(300);
Query ok, 1 rows affected (0.00 sec)
mysql> select * from t1;
+-----+
| id  |
+-----+
| 200 |
| 300 |
+-----+
2 row in set (0.00 sec)
```

可以看到，当节点4 Crash之后，节点5仍然能够提供正常的服务。当然，如果在应用环境中,应用环境需要至少支持当一个SQL节点出现问题的时候能够自行切换到剩下的正常的SQL节点来访问。

c、管理节点的单点

一般情况来说，管理节点是最容易控制的，实施也非常简单，只需要将配置文件和两个可执行程序（ndb_mgmd和ndb_mgm）存放在多台机器上面即可，所以一般来说不需要太多考虑单点故障。

16.3 MySQL Cluster配置详细介绍（config.ini）

在MySQL Cluster环境的配置文件config.ini里面，每一类节点都有两个（或以上）的相应配置项组，每一类节点的配置项都主要由两部分组成，一部分是同类所有节点相同的配置项组，在[NDB_MGM DEFAULT]、[NDBD DEFAULT]和[MySQLD DEFAULT]这三个配置组里面,而且每一个配置组只出现一次;而另外一部分则是针对每一个节点独有配置内容的配置项组[NDB_MGM]、[NDBD]和[MySQLD]，由于这三类配置组中配置的每一个节点独有的个性化配置,

所以每一个配置组都可能会出现多次（每一个节点一次）。下面是每一类节点的各种配置说明：

1、管理节点相关配置

在整个 MySQL Cluster 环境中，管理节点相关的配置为[NDBD_MGM DEFAULT]和[NDB_MGMD]相关的两组：

1）[NDB_MGMD DEFAULT]中各管理节点的共用配置项：

PortNumber：配置管理节点的服务端程序（ndb_mgmd）监听客户端（ndb_mgm）连接请求和发送的指令，从文档上可以查找到，默认端口是 1186 端口。一般来说这一项不需要更改，当然如果是为了在同一台主机上面启动多个管理节点的话，肯定需要将两个管理节点启动不同的监听端口；

LogDestination：配置管理节点上面的 cluster 日志处理方式。

a）可以写入文件如：LogDestination=FILE:filename=my-cluster.log,maxsize=500000,maxfiles=4;

b）也可以通过标准输出来打印出来如：LogDestination=CONSOLE;

c）还可以计入 syslog 里面如：LogDestination=SYSLOG:facility=syslog;

d）甚至多种方式共存：LogDestination=CONSOLE;SYSLOG:facility=syslog;FILE:filename=/var/log/cluster-log

Datadir:设置用于管理节点存放文件输出的位置。如 process 文件(.pid),cluster log 文件（当 LogDestination 有 FILE 处理方式存在时候）。

ArbitrationRank：配置各节点在处理某些事件出现分歧的时候的级别。有 0，1,2 三个值可以选择。

a）0 代表本节点完全听其他节点的，不参与决策

b）1 代表本节点有最高优先权，“一切由我来决策”

c）2 代表本节点参与决策，但是优先权较 1 低，但是比 0 高

ArbitrationRank 参数不仅仅管理节点有，MySQL 节点也有。而且一般来说，所有的管理节点一般都应该设置成 1，所有 SQL 节点都设置成 2。

2）[NDB_MGMD]是每个管理节点配置一组，所需配置项如下（下面的参数只能设置在[NDB_MGMD]参数组中）：

Id：为节点指定一个唯一的 ID 号，要求在整个 Cluster 环境中唯一；

Hostname：配置该节点的 IP 地址或者主机名，如果是主机名，则该主机名必须要在配置文件所在的节点的/etc/hosts 文件中存在，而且绑定的 IP 是准确的。

上面[NDB_MGMD DEFAULT]里面的所有参数项，都可以设置在下面的[NDB_MGMD]参数组里面，但是 Id 和 Hostname 两个参数只能设置在[NDB_MGMD]里面，而不能设置在[NDB_MGMD DEFAULT]里面，因为这两个参数项针对每一个节点都是不相同的内容。

2、NDB 节点相关配置

NDB 节点和管理节点一样，既有各个节点共用的配置信息组[NDBD DEFAULT]，也有每一个节点个性化配置的[NDBD]配置组（实际上 SQL 节点也是如此）。

1) [NDBD DEFAULT]中的配置项：

NoOfReplicas：定义在 Cluster 环境中相同数据的分数，通俗一点来说就是每一份数据存放 NoOfReplicas 份。如果希望能够冗余，那么至少设置为 2（一般情况来说此参数值设置为 2 就够了），最大只能设置为 4。另外，NoOfReplicas 值得大小，实际上也就是 node group 大小的定义。NoOfReplicas 参数没有系统默认值，所以必须设定，而且只能设置在[NDBD DEFAULT]中，因为此数值在整个 Cluster 集群中一个 node group 中所有的 NDBD 节点都需要一样。另外 NoOfReplicas 的数目对整个 Cluster 环境中 NDB 节点数量有较大的影响，因为 NDB 节点总数量是 NoOfReplicas * 2 * node_group_num；

DataDir：指定本地的 pid 文件，trace 文件，日志文件以及错误日志子等存放的路径，无系统默认地址，所以必须设定；

DataMemory：设定用于存放数据和主键索引的内存段的大小。这个大小限制了能存放的数据的大小，因为 ndb 存储引擎需属于内存数据库引擎，需要将所有的数据（包括索引）都 load 到内存中。这个参数并不是一定需要设定的，但是默认值非常小（80M），只也就是说如果使用默认值，将只能存放很小的数据。参数设置需要带上单位，如 512M，2G 等。另外，DataMemory 里面还会存放 UNDO 相关的信息，所以，事务的大小和事务并发量也决定了 DataMemory 的使用量，建议尽量使用小事务；

IndexMemory：设定用于存放索引（非主键）数据的内存段大小。和 DataMemory 类似，这个参数值的大小同样也会限制该节点能存放的数据的大小，因为索引的大小是随着数据量增长而增长的。参数设置也如 DataMemory 一样需要单位。IndexMemory 默认大小为 18M；

实际上，一个 NDB 节点能存放的数据量是会受到 DataMemory 和 IndexMemory 两个参数设置的约束，两者任何一个达到限制数量后，都无法再增加能存储的数据量。如果继续存入数据系统会报错“table is full”。

FileSystemPath：指定 redo 日志，undo 日志，数据文件以及 meta 数据等的存放位置，默认位置为 DataDir 的设置，并且在 ndbd 初始化的时候，参数所设定的文件夹必须存在。在第一次启动的时候，ndbd 进程会在所设定的文件夹下建立一个子文件夹叫 ndb_id_fs，这里的 id 为节点的 ID 值，如节点 id 为 3 则文件夹名称为 ndb_3_fs。当然，这个参数也不一定非得设置在[NDBD DEFAULT]参数组里面让所有节点的设置都一样（不过建议这样设置），还可以设置在[NDBD]参数组下为每一个节点单独设置自己的 FileSystemPath 值；

BackupDataDir：设置备份目录路径，默认为 FileSystemPath/BACKUP。

接下来的几个参数也是非常重要的，主要都是与并行事务数和其他一些并行限制有关的参数设置。

MaxNoOfConcurrentTransactions：设置在一个节点上面的最大并行事务数目，默认为 4096，一般情况下来说是足够了的。这个参数值所有节点必须设置一样，所以一般都是设置在[NDBD DEFAULT]参数组下面；

MaxNoOfConcurrentOperations：设置同时能够被更新（或者锁定）的记录数量。一般来说可以设置为在整个集群中相同时间内可能被更新(或者锁定)的总记录数,除以NDB节点数，所得到的值。比如，在集群中有两个NDB节点，而希望能够处理同时更新（或锁定）100000条记录，那么此参数应该被设置为：100000 / 4 = 25000。此外，这里的记录数量并不是指单纯的表里面的记录数，而是指事物里面的操作记录。当使用到唯一索引的时候，表的数据和索引两者都要算在里面，也就是说，如果是通过一个唯一索引来作为过滤条件更新某一条记录，那么这里算是两条操作记录。而且即使是锁定也会产生操作记录，比如通过唯一索引来查找一条记录，就会产生如下两条操作记录：通过读取唯一索引中的某个记录数据会产生锁定，产生一条操作记录，然后读取基表里面的数据，这里也会产生读锁，也会产生一条操作记录。MaxNoOfConcurrentOperations参数的默认值为32768。当我们额度系统运行过程中，如果出现此参数不够的时候，就会报出“Out of operation records in transaction coordinator”这样的错误信息；

MaxNoOfLocalOperations：此参数默认是MaxNoOfConcurrentOperations * 1.1的大小，也就是说，每个节点一般可以处理超过平均值的10%的操作记录数量。但是一般来说，MySQL建议单独设置此参数而不要使用默认值，并且将此参数设置得更较大一些；

以下的三个参数主要是在一个事务中执行一条query的时候临时用到存储(或者内存）的情况下所使用到的，所使用的存储信息会在事务结束（commit或者rollback）的时候释放资源；

MaxNoOfConcurrentIndexOperations：这个参数和MaxNoOfConcurrentOperations参数比较类似，只不过所针对的是Index的record而已。其默认值为8192，对伊一般的系统来说都已经足够了，只有在事务并发非常非常大的系统上才有需要增加这个参数的设置。当然，此参数越大，系统运行时候为此而消耗的内存也会越大；

MaxNoOfFiredTriggers：触发唯一索引（hash index）操作的最大的操作数，这个操作数是影响索引的操作条目数，而不是操作的次数。系统默认值为4000，一般系统来说够用了。当然，如果系统并发事务非常高，而且涉及到索引的操作也非常多，自然也就需要提高这个参数值的设置了；

TransactionBufferMemory：这个buffer值得设置主要是指定用于跟踪索引操作而使用的。主要是用来存储索引操作中涉及到的索引key值和column的实际信息。这个参数的值一般来说也很少需要调整，因为实际系统中需要的这部分buffer量非常小，虽然默认值只是1M，但是对于一般应用也已经足够了；

下面要介绍到的参数主要是在系统处理中做table scan或者range scan的时候使用的一些buffer的相关设置，设置的恰当可以既节省内存又达到足够的性能要求。

MaxNoOfConcurrentScans：这个参数主要控制在Cluster环境中并发的table scan和range scan的总数量平均分配到每一个节点后的平均值。一般来说，每一个scan都是通过并行的扫描所有的partition来完成的，每一个partition的扫描都会在该partition

所在的节点上面使用一个 scan record。所以，这个参数值得大小应该是“scan record”数目 * 节点数目。参数默认大小为 256，最大只能设置为 500；

MaxNoOfLocalScans：和上面的这个参数相对应，只不过设置的是在本节点上面的并发 table scan 和 range scan 数量。如果在系统中有大量的并发而且一般都不使用并行的话，需要注意此参数的设置。默认为 MaxNoOfConcurrentScans * node 数目；

BatchSizePerLocalScan：该参用于计算在 Localscan（并发）过程中被锁住的记录数，文档上说明默认为 64；

LongMessageBuffer：这个参数定义的是消息传递时候的 buffer 大小，而这里的消息传递主要是内部信息传递以及节点与节点之间的信息传递。这个参数一般很少需要调整，默认大小为 1MB 大小；

下面介绍一下与 log 相关的参数配置说明，包括 log level。这里的 log level 有多种，从 0 到 15，也就是共 16 种。如果设定为 0，则表示不记录任何 log。如果设置为最高 level，也就是 15，则表示所有的信息都会通过标准输出来记录 log。由于这里的所有信息实际上都会传递到管理节点的 cluster log 中，所以，一般来说，除了启动时候的 log 级别需要设置为 1 之外，其他所有的 log level 都只需要设置为 0 就可以了。

NoOfFragmentLogFiles：这个参数实际上和 Oracle 的 redo log 的 group 一样的。其实就是 ndb 的 redo log group 数目，这些 redo log 用于存放 ndb 引擎所做的所有需要变更数据的事情，以及各种 checkpoint 信息等。默认值为 8；

MaxNoOfSavedMessages：这个参数设定了可以保留的 trace 文件（在节点 crash 的时候参数）的最大个数，文档上面说此参数默认值为 25。

LogLevelStartup：设定启动 ndb 节点时候需要记录的信息的级别（不同级别所记录的信息的详细程度不一样），默认级别为 1；

LogLevelShutdown：设定关闭 ndb 节点时候记录日志的信息的级别，默认为 0；

LogLevelStatistic：这个参数是针对于统计相关的日志的，就像更新数量，插入数量，buffer 使用情况，主键数量等等统计信息。默认日志级别为 0；

LogLevelCheckpoint：checkpoint 日志记录级别（包括 local 和 global 的），默认为 0；

LogLevelNodeRestart：ndb 节点重启过程日志级别，默认为 0；

LogLevelConnection：各节点之间连接相关日志记录的级别，默认 0；

LogLevelError：在整个 Cluster 中错误或者警告信息的日志记录级别，默认 0；

LogLevelInfo：普通信息的日志记录级别，默认为 0。

这里再介绍几个用来作为 log 记录时候需要用到的 Buffer 相关参数，这些参数对于性能都有一定的影响。当然，如果节点运行在无盘模式下的话，则影响不大。

UndoIndexBuffer：undo index buffer 主要是用于存储主键 hash 索引在变更之后产生的 undo 信息的缓冲区。默认值为 2M 大小，最小可以设置为 1M，对于大多数应用来说，2M 的默认值是够的。当然，在更新非常频繁的应用里面，适当的调大此参数值对性能还是有一定帮助的。如果此参数太小，会报出 677 错误：Index UNDO buffers overloaded；

UndoDataBuffer：和 undo index buffer 类似，undo data buffer 主要是在数据发生变更的时候所需要的 undo 信息的缓冲区。默认大小为 16M，最小同样为 1M。当这个参数值太小的时候，系统会报出如下的错误：Data UNDO buffers overloaded，错误号为 891；

RedoBuffer：Redo buffer 是用 redo log 信息的缓冲区，默认大小为 8M，最小为 1M。如果此 buffer 太小，会报 1221 错误：REDO log buffers overloaded。

此外，NDB 节点还有一些和 metadata 以及内部控制相关的参数，但大部分参数都基本上不需要任何调整，所以就不做进一步介绍。如果有兴趣希望详细了解，可以根据 MySQL 官方的相关参考手册，手册上面都有较为详细的介绍。

3、SQL 节点相关配置说明

1）和其他节点一样，先介绍一些适用于所有节点的[MySQLD DEFAULT]参数

ArbitrationRank：这个参数在介绍管理节点的参数时候已经介绍过了，用于设定节点级别（主要是在多个节点在处理相关操作时候出现分歧时候设定裁定者）的。一般来说，所有的 SQL 节点都应该设定为 2；

ArbitrationDelay：默认为 0，裁定者在开始裁定之前需要被 delay 多久，单位为毫秒。一般不需要更改默认值。

BatchByteSize：在做全表扫描或者索引范围扫描的时候，每一次 fatch 的数据量，默认为 32KB；

BatchSize：类似 BatchByteSize 参数，只不过 BatchSize 所设定的是每一次 fetch 的 record 数量，而不是物理总量，默认为 64，最大为 992（暂时还不知道这个值是基于什么理论而设定的）。在实际运行 query 的过程中，fetch 的量受到 BatchByteSize 和 BatchSize 两个参数的共同制约，二者取最小值；

MaxScanBatchSize：在 Cluster 环境中，进行并行处理的情况下，所有节点的 BatchSize 总和的最大值。默认值为 256KB，最大值为 16MB。

2）每个节点独有的[MySQLD]参数组，仅有 id 和 hostname 参数需要配置，在之前各类节点均有介绍了，这里就不再累述。

16.4 MySQL Cluster 基本管理与维护

MySQL Cluster 的管理和普通的 MySQL Server 管理区别较大，基本上大部分管理工作都是在管理节点上面完成，仅有少数管理内容需要在其他节点实施。

1、各节点启动与关闭

要想 Cluster 环境能够正常工作，只好要启动一个 NDB 节点和一个 SQL 节点，另外为了完成管理，也至少要启动一个管理节点。各类节点的启动顺序也有要求，首先是管理节点，然后是 NDB 节点，最后才是 SQL 节点。

1) 按顺序启动各节点：

a、 启动管理节点：

[root@localhost MySQL-cluster]# ndb_mgmd -f /var/lib/MySQL-cluster/config.ini

这里执行的 ndb_mgmd 命令实际上就是 MySQL Cluster 管理服务器，可以通过-f config_file_name 或者--config=config_filename 来指定 MySQL Cluster 集群的参数文件。如果想了解更多关于 ndb_mgmd 的参数信息，可以通过运行 ndb_mgmd --help 来获取更详细的信息。

b、 启动用于存储数据的 ndb 节点

要启动存储节点，必须在每一台 ndb 节点主机上面都执行 ndbd 程序，如果是第一次启动，则需要添加--initial 参数，以便进行 ndb 节点的初始化工作。但是，在以后的启动过程中，是不能添加该参数的，否则 ndbd 程序会清除在之前建立的所有用于恢复的数据文件和日志文件。启动命令如下

root@ndb1:/root>ndbd --initial

c、 启动 SQL 节点

SQL 节点的启动和普通 MySQL Server 的启动没有太多明显的差别，不过有一个前提就是需要在 MySQL Server 的配置文件 my.cnf 设置好[MySQL_cluster]配置组中的 ndb-connectstring 参数和[MySQLd]配置组中的 ndbcluster 参数。

root@mysql1:/root>MySQLd_safe --user=MySQL &

2) 节点状态检查：

在各节点都启动完成后，回到管理节点，可以通过 ndb_mgm 来查看各节点状态：

```
[root@localhost MySQL-cluster]# ndb_mgm -e SHOW
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=2    @192.168.0.3  (Version: 5.0.51, Nodegroup: 0, Master)
id=3    @192.168.0.4  (Version: 5.0.51, Nodegroup: 0)

[ndb_mgmd(MGM)] 1 node(s)
id=1    @192.168.0.5  (Version: 5.0.51)
```

[MySQLd(API)] 2 node(s)
id=4 @192.168.0.1 (Version: 5.0.51)
id=5 @10.0.65.203 (Version: 5.0.51)

这里显示出整个集群有 5 个几点，其中各节点信息如下：

a) 2 个 NDBD 节点：

[ndbd(NDB)] 2 node(s)
id=2 @192.168.0.3 (Version: 5.0.51, Nodegroup: 0, Master)
id=3 @192.168.0.4 (Version: 5.0.51, Nodegroup: 0)

b) 两个 SQL 节点:

[MySQLd(API)] 2 node(s)
id=4 @192.168.0.1 (Version: 5.0.51)
id=5 @10.0.65.203 (Version: 5.0.51)

c) 1 个管理节点:

[ndb_mgmd(MGM)] 1 node(s)
id=1 @192.168.0.5 (Version: 5.0.51)

3) 节点的关闭操作：

在 MySQL Cluster 环境中，NDB 节点和管理节点的关闭都可以在管理节点的管理程序中完成，但是 SQL 节点却没办法。所以，在关闭整个 MySQL Cluster 环境或者关闭某个 SQL 节点的时候，首先必须到 SQL 节点主机上来关闭 SQL 节点程序。关闭方法和 MySQL Server 的关闭一样，就不累述。而 NDB 节点和管理节点则都可以在管理节点通过管理程序来完成：

```
ndb_mgm> shutdown
Connected to Management Server at: localhost:1186
Node 3: Cluster shutdown initiated
Node 2: Cluster shutdown initiated
Node 2: Node shutdown completed.
Node 3: Node shutdown completed.
2 NDB Cluster node(s) have shutdown.
Disconnecting to allow management server to shutdown.
```

2、基本管理维护

前面运行的命令 ndb_mgm 如果不带任何参数，实际上是进入 MySQL Cluster 的命令行管理界面。在命令行管理界面里面可以做大量的维护工作，如下：

```
[root@localhost MySQL-cluster]# ndb_mgm
-- NDB Cluster -- Management Client --
ndb_mgm>
```

然后同样执行 show 命令：

```
ndb_mgm>show
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]  2 node(s)
```

```
id=2 (not connected, accepting connect from 192.168.0.3)
id=3    @192.168.0.4  (Version: 5.0.51, Nodegroup: 0, Master)

[ndb_mgmd(MGM)] 1 node(s)
id=1    @192.168.0.5  (Version: 5.0.51)

[MySQLd(API)]   2 node(s)
id=4    @192.168.0.1  (Version: 5.0.51)
id=5    @10.0.65.203  (Version: 5.0.51)
```

我们可以看到结果和上面的完全一样。可以通过在 ndb 控制界面下执行 help 命令查看可以查看很多基本的维护管理命令：

```
ndb_mgm> help
---------------------------------------------------------------------------
 NDB Cluster -- Management Client -- Help
---------------------------------------------------------------------------
HELP                                    Print help text
HELP COMMAND                            Print detailed help for COMMAND(e.g.
SHOW)
SHOW                                    Print information about cluster
START BACKUP [NOWAIT | WAIT STARTED | WAIT COMPLETED]
                                        Start backup (default WAIT COMPLETED)

ABORT BACKUP <backup id>                Abort backup
SHUTDOWN                                Shutdown all processes in cluster
CLUSTERLOG ON [<severity>] ...          Enable Cluster logging
CLUSTERLOG OFF [<severity>] ...         Disable Cluster logging
CLUSTERLOG TOGGLE [<severity>] ...      Toggle severity filter on/off
CLUSTERLOG INFO                         Print cluster log information
<id> START                              Start data node (started with -n)
<id> RESTART [-n] [-i]                  Restart data or management server
node
<id> STOP                               Stop data or management server node
ENTER SINGLE USER MODE <id>             Enter single user mode
EXIT SINGLE USER MODE                   Exit single user mode
<id> STATUS                             Print status
<id> CLUSTERLOG {<category>=<level>}+   Set log level for cluster log
PURGE STALE SESSIONS                    Reset reserved nodeid's in the mgmt
server
CONNECT [<connectstring>]               Connect to management server
(reconnect if already connected)
QUIT                                    Quit management client
```

```
    <severity> = ALERT | CRITICAL | ERROR | WARNING | INFO | DEBUG
    <category> = STARTUP | SHUTDOWN | STATISTICS | CHECKPOINT | NODERESTART |
CONNECTION | INFO | ERROR | CONGESTION | DEBUG | BACKUP
    <level>    = 0 - 15
    <id>       = ALL | Any database node id

    For detailed help on COMMAND, use HELP COMMAND.
```

也可以通过执行 help 后面跟命令名称而获取各种命令的操作说明帮助信息：

```
    ndb_mgm> help start
    ---------------------------------------------------------------------------
     NDB Cluster -- Management Client -- Help for START command
    ---------------------------------------------------------------------------
    START  Start data node (started with -n)

    <id> START         Start the data node identified by <id>.
                       Only starts data nodes that have not
                       yet joined the cluster. These are nodes
                       launched or restarted with the -n(--nostart)
                       option.

                       It does not launch the ndbd process on a remote
                       machine.
    ndb_mgm> help shutdown
    ---------------------------------------------------------------------------
     NDB Cluster -- Management Client -- Help for SHUTDOWN command
    ---------------------------------------------------------------------------
    SHUTDOWN  Shutdown the cluster

    SHUTDOWN           Shutdown the data nodes and management nodes.
                       MySQL Servers and NDBAPI nodes are currently not
                       shut down by issuing this command.
    ndb_mgm> help PURGE STALE SESSIONS
    ---------------------------------------------------------------------------
     NDB Cluster -- Management Client -- Help for PURGE STALE SESSIONS command
    ---------------------------------------------------------------------------
    PURGE STALE SESSIONS  Reset reserved nodeid's in the mgmt server

    PURGE STALE SESSIONS
                       Running this statement forces all reserved
                       node IDs to be checked; any that are not
                       being used by nodes acutally connected to
                       the cluster are then freed.
```

```
This command is not normally needed, but may be
required in some situations where failed nodes
cannot rejoin the cluster due to failing to
allocate a node id.
```

通过上面的几个帮助命令所获取的信息得知，我们可以通过在管理节点上面通过执行 restart，stop，shutdown 等基本的命令来重启某个节点，关闭某个节点，还可以同时一次性关闭所有节点。

此外，还可以通过执行备份相关的命令在管理节点对整个 Cluster 环境进行备份，以及通过日志相关命令实施对日志的相关管理。

16.5 基本优化思路

MySQL Cluster 虽然是一个分布式的数据库系统，但是在大部分地方的优化思路和方法还是和普通的 MySQL Server 一样。和常规 MySQL Server 在优化方面的区别主要提现在各节点之间的协作配置以及网络环境相关的优化。

由于 MySQL Cluster 是一个分布式的环境，而且所有访问都是需要经过超过一个节点（至少有一个 SQL 节点和一个 NDB 节点）才能完成，所以各个节点之间的协作配合就显得尤为重要。

首先，由于各个节点之间存在大量的数据通讯，所以节点之间的内部互联网络带宽一定要保证足够使用。为了适应不同的网络环境和性能需求，MySQL Cluster 支持了多种内部网络互联的协议和方式。最为常用的自然是通过 TCP/IP 来进行互联。此外还可以有 SCI Socket 方式来进行互联，还支持 Myrinet，Infiniband，VIA 接口等等。

其次，SQL 节点和 NDB 节点的主机性能配比应该合适，而不应该出现某一类节点过早出现瓶颈的时候，另外一类节点却还处于非常空闲的状态。如果在我们遇到的环境中出现这样的情况，那么我们就该重新评估两类节点的硬件设备配比了。否则，有一类节点的硬件资源就相当处于浪费状态了。

最后，就是 SQL 节点 和 NDB 节点两者软件配置方面的优化了。对于 SQL 节点的配置，和普通的 MySQL 区别不是太大。各类参数的配置原则也和普通 MySQL 基本相同。NDB Cluster 存储引擎的主要配置参数在前面的配置介绍中也基本都进行了性能相关的说明，这里就不再累述了。

16.6 小结

MySQL Cluster 的核心在于 NDB Cluster 存储引擎，他不仅对数据进行了水平切分，

还对数据进行了跨节点冗余。既解决了数据库的扩展问题，同时也在很大程度上提高了数据库整体可用性。

虽然目前 MySQL Cluster 的应用还不如普通的 MySQL Server 应用那么广泛，但是我想随着他的不断成熟和改善，将会被越来越广泛的使用。这种 Share Nothing 的 Cluster 架构也很可能会成为未来的趋势，就让我们共同期待越来越成熟稳定高效的 MySQL Cluster 吧。

第 17 章 高可用设计之思路及方案

前言：

数据库系统是一个应用系统的核心部分，要想系统整体可用性得到保证，数据库系统就不能出现任何问题。对于一个企业级的系统来说，数据库系统的可用性尤为重要。数据库系统一旦出现问题无法提供服务，所有系统都可能无法继续工作，而不像软件中部分系统出现问题可能影响的仅仅只是某个功能无法继续服务。所以，一个成功的数据库架构在高可用设计方面也是需要充分考虑的。本章内容将针对如何构建一个高可用的 MySQL 数据库系统来介绍各种解决方案以及方案之间的比较。

17.1 利用 Replication 来实现高可用架构

做维护的读者朋友应该都清楚，任何设备（或服务），只要是单点，就存在着很大的安全隐患。因为一旦这台设备（或服务） crash 之后，在短时间内就很难有备用设备（或服务）来顶替其功能。所以稍微重要一些的服务器或者应用系统，都会存在至少一个备份以供出现异常的时候能够很快的顶替上来提供服务。

对于数据库来说，主备配置是非常常见的设计思路。而对于 MySQL 来说，可以说天生

就具备了实现该功能的优势，因为其 Replication 功能在实际应用中被广泛的用来实现主备配置的功能。

我想，在大家所接触的 MySQL 环境中大多数都存在通过 MySQL Replication 来实现两台（或者多台） MySQL Server 之间的数据库复制功能吧。可能有些是为了增强系统扩展性，满足性能要求实现读写分离。亦或者就是用来作为主备机的设计，保证当主机 crash 之后在很短的时间内就可以将应用切换到备机上面来继续运行。

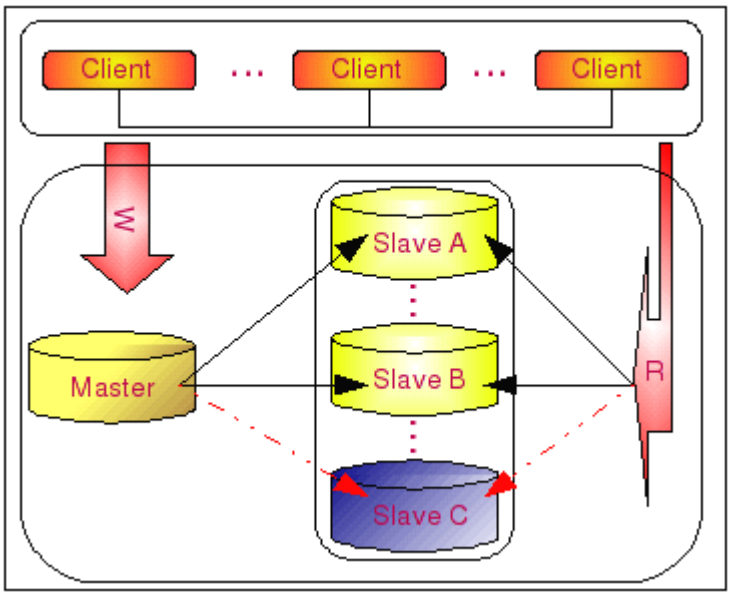
通过 MySQL Replication 来实现数据库的复制实际上在前面第 13 章中的内容中已经做过详细的介绍了，也介绍了多种 Replication 架构的实现原理及实现方法。在之前，主要是从扩展性方面来介绍 Replication。在这里，我将主要介绍的是从系统高可用方面如何和利用 Replication 的多种架构实现解决高可靠性的问题。

17.1.1 常规的 Master - Slave 解决基本的主备设计

在之前的章节中我提到过，普通的 Master - Slave 架构是目前很多系统中使用最为常见的一种架构方式。该架构设计不仅仅在很大程度上解决的系统的扩展性问题，带来性能的提升，同时在系统可靠性方面也提供了一定的保证。

在普通的一个 Master 后面复制一个或者多个 Slave 的架构设计中，当我们的某一台 Slave 出现故障不能提供服务之后，我们还有至少一台 MySQL 服务器（Master）可以提供服务，不至于所有和数据库相关的业务都不能运行下去。如果 Slave 超过一台，那么剩下的 Slave 也仍然能够不受任何干扰的继续提供服务。

当然，这里有一点在设计的时候是需要注意的，那就是我们的 MySQL 数据库集群整体的服务能力要至少能够保证当其缺少一台 MySQL Server 之后还能够支撑系统的负载，否则一切都是空谈。不过，从系统设计角度来说，系统处理能力留有一定的剩余空间是一个比较基本的要求。所以正常来说，系统中短时间内少一台 MySQL Server 应该是仍然能够支撑正常业务的。



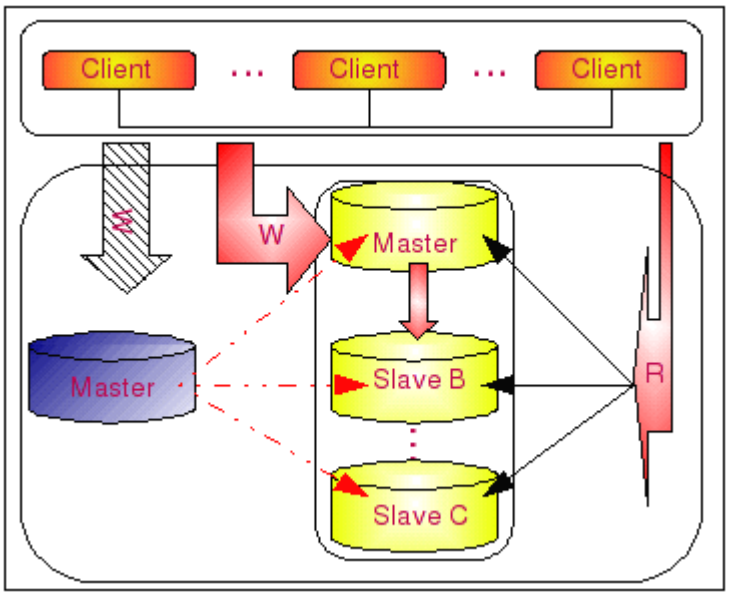
如上图所示，当我们的 Slave 集群中的一台 Slave C 出现故障 crash 之后，整个系统的变化仅仅只是从 Master 至 Slave C 的复制中断，客户端应用的 Read 请求也不能再访问 Slave C。当时其他所有的 MySQL Server 在不需要任何调整的情况下就能正常工作。客户端的请求 Read 请求全部由 Slave A 和 Slave B 来承担。

17.1.2 Master 单点问题的解决

上面的架构可以很容易的解决 Slave 出现故障的情况，而且不需要进行任何调整就能继续提供服务。但是，当我们的 Master 出现问题后呢？当我们的 Master 出现问题之后所有客户端的 Write 请求就都无法处理了。

这时候我们可以有如下两种解决方案，一个是将 Slave 中的某一台切换成 Master 对外提供服务，同时将其他所有的 Slave 都以通过 CHANGE MASTER 命令来将通过新的 Master 进行复制。另一个方案就是新增一台 Master，也就是 Dual Master 的解决方案。

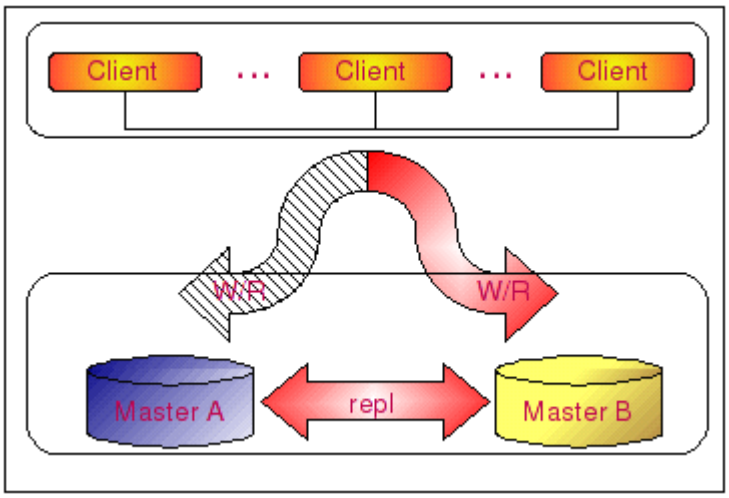
我们先来看看第一种解决方案，将一台 Slave 切换成 Master 来解决问题，如图：



当 Master 出现故障 crash 之后，原客户端对 Master 的所有 Write 请求都会无法再继续进行下去了，所有原 Master 到 Slave 的复制也自然就断掉了。这时候，我们选择一台 Slave 将其切换成 Master。假设选择的是 Slave A，则我们将其他 Slave B 和 Slave C 都通过 CHANGE MASTER TO 命令更换其 Master，从新的 Master 也就是原 Slave A 开始继续进行复制。同时将应用端所有的写入请求转向到新的 Master。对于 Read 请求，我们可以去掉对新 Master 的 Read 请求，也可以继续保留。

这种方案最大的一个弊端就是切换步骤比较多，实现比较复杂。而且，在 Master 出现故障 crash 的那个时刻，我们的所有 Slave 的复制进度并不一定完全一致，有可能有少量的差异。这时候，选择哪一个 Slave 作为 Master 也是一个比较头疼的问题。所以这个方案的可控性并不是特别的高。

我们再来看看第二种解决方案，也就是通过 Dual Master 来解决 Master 的点问，为了简单明了，这里就仅画出 Master 部分的图，如下：



我们通过两台 MySQL Server 搭建成 Dual Master 环境，正常情况下，所有客户端的 Write 请求都写往 Master A，然后通过 Replication 将 Master A 复制到 Master B。一旦 Master A 出现问题之后，所有的 Write 请求都转向 Master B。而在正常情况下，当 Master B 出现问题的时候，实际上不论是数据库还是客户端的请求，都不会受到实质性的影响。

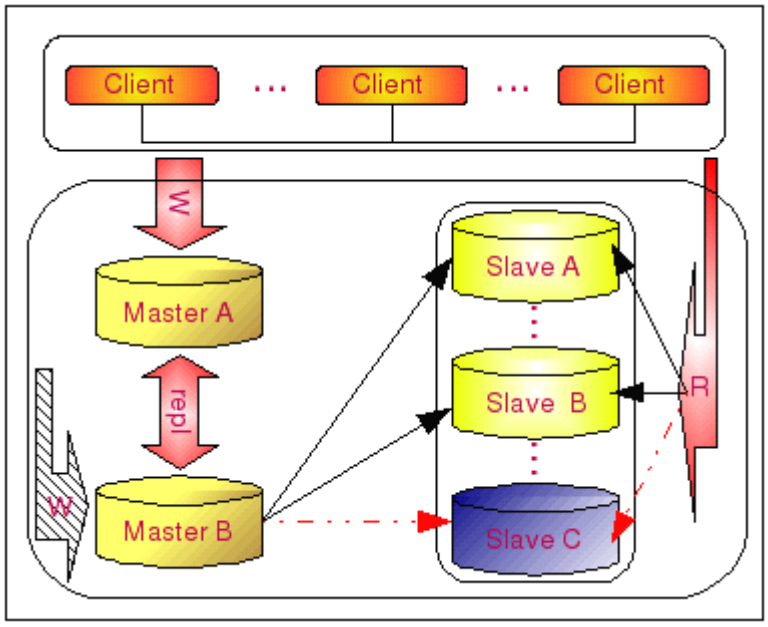
这里，可能有读者朋友会想到，当我们的 Master A 出现问题的时候，应用如何做到自动将请求转向到 Master B 呢？其实很简单，我们只需要通过相应的硬件设备如 F5 或者 Cluster 管理软件如 Heartbeat 来设置一个 VIP，正常情况下该 VIP 指向 Master A，而一旦 Master A 出现异常 crash 之后，则自动切换指向到 Master B，前端所的应用都通过这个 VIP 来访问 Master。这样，既解决了应用的 IP 切换问题，还保证了在任何时刻应用都只会见到一台 Master，避免了多点写入出现数据紊乱的可能。

这个方案最大的特点就是在 Master 出现故障之后的处理比较简单，可控性比较大。而弊端就是需要增加一台 MySQL 服务器，在成本方面投入更大。

17.1.3 Dual Master 与级联复制结合解决异常故障下的高可用

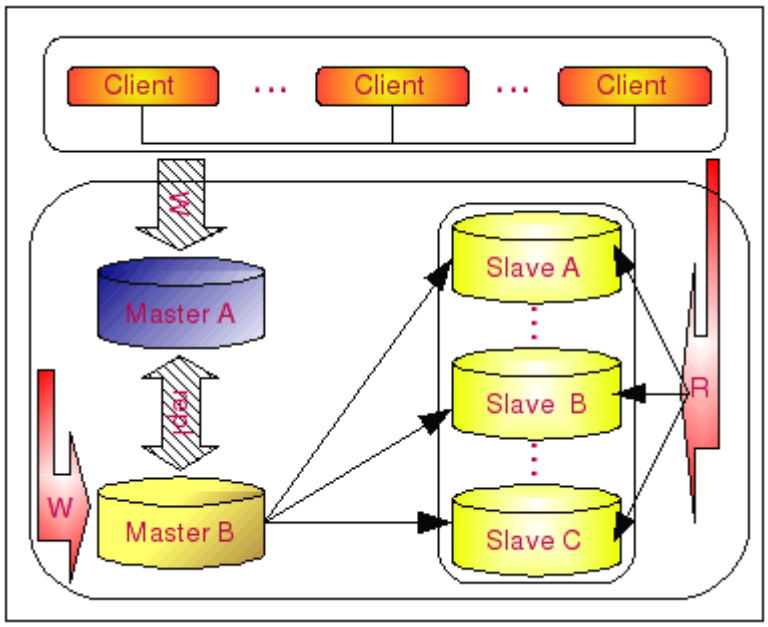
通过前面的架构分析，我们分别得到了 Slave 出现故障后的解决方案，也解决了 Master 的单点问题。现在我们再通过 Dual Master 与级联复制结合的架构，来得到一个整体的解决方案，解决系统整体可靠性的问题。

这个架构方案的介绍在之前的章节中已经详细的描述过了，这里我们主要分析一下处于高可靠性方面考虑的完善和异常切换方法。



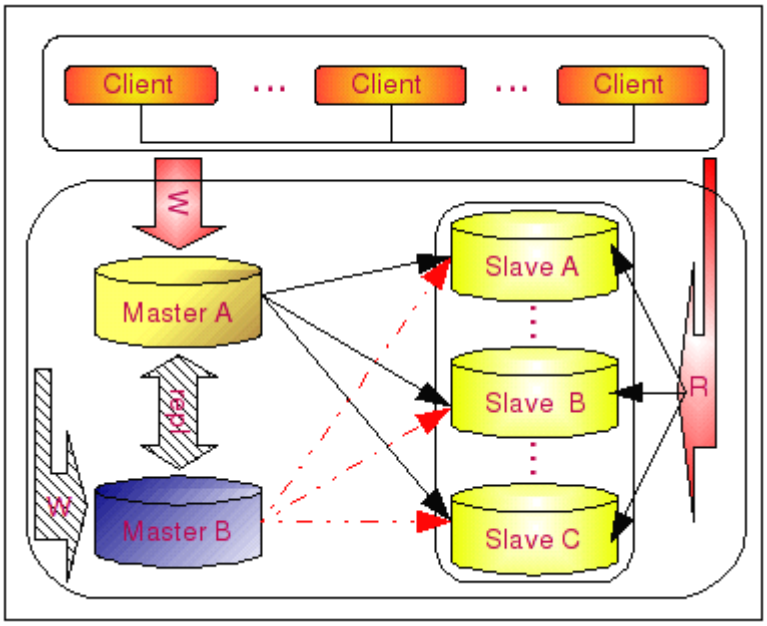
如上图所示，首先考虑 Slave 出现异常的情况。在这个架构中，Slave 出现异常后的处理情况和普通的 Master - Slave 架构的处理方式完全一样，仅仅需要在应用访问 Slave 集群的访问配置中去掉一个 Slave 节点即可解决，不论是通过应用程序自己判断，还是通过硬件解决方案如 F5 都可以很容易的实现。

下面我们再看看当 Master A 出现故障 crash 之后的处理方案。如下图：



当 Master A 出现故障 crash 之后，Master A 与 Master B 之间的复制将中断，所有客户端向 Master A 的 Write 请求都必须转向 Master B。这个转向动作的实现，可以通过上一节中所介绍的第二中方案中所介绍的通过 VIP 的方式实现。由于之前所有的 Slave 就都是从 Master B 来实现复制，所以 Slave 集群不会受到任何的影响，客户端的所有 Read 请求也就不会受到任何的影响，整个过程可以完全自动进行，不需要任何的人为干预。不过这里有一个隐患就是当 Master A crash 的时候如果 Master B 作为 Slave 的 IO 线程如果还没有读取完 Master A 的二进制日志的话，就会出现数据丢失的问题。要完全解决这个问题，我们只能通过第三方 patch（google 开发）来镜像 MySQL 的二进制日志到 Master B 上面，才能完全避免不丢失任何数据。

那么当 Master B 出现故障 crash 之后的情况又如何呢？如下图所示：



如果出现故障的不是 Master B 而是 Master A 又会怎样呢？首先可以确定的是我们的所有 Write 请求都不会受到任何影响，而且所有的 Read 请求也都还是能够正常访问。但所有 Slave 的复制都会中断，Slave 上面的数据会开始出现滞后的现象。这时候我们需要做的就是将所有的 Slave 进行 CHANGE MASTER TO 操作，改为从 Master A 进行复制。由于所有 Slave 的复制都不可能超前最初的数据源，所以可以根据 Slave 上面的 Relay Log 中的时间戳信息与 Master A 中的时间戳信息进行对照来找到准确的复制起始点，不会造成任何的数据丢失。

17.1.4 Dual Master 与级联复制结合解决在线 DDL 变更问题

当我们使用 Dual Master 加级联复制的组合架构的时候，对于 MySQL 的一个致命伤也就是在线 DDL 变更来说，也可以得到一定的解决。如当我们需要给某个表 tab 增加一个字段，可以通过如下在上述架构中来实现：

1、 在 Slave 集群中抽出一台暂时停止提供服务，然后对其进行变更，完成后再放回集群继续提供服务；
2、 重复第一步的操作完成所有 Slave 的变更；
3、 暂停 Master B 的复制，同时关闭当前 session 记录二进制日志的功能，对其进行变更，完成后再启动复制；
4、 通过 VIP 切换，将应用所有对 Master A 的请求切换至 Master B；
5、 关闭 Master A 当前 session 记录二进制日志的功能，然后进行变更；
6、 最后再将 VIP 从 Master B 切换回 Master A，至此，所有变更完成。

变更过程中有几点需要注意的：

1、 整个 Slave 集群需要能够承受在少一台 MySQL 的时候仍然能够支撑所有业务；
2、 Slave 集群中增加或者减少一台 MySQL 的操作简单，可通过在线调整应用配置来

实现；

3、 Dual Master 之间的 VIP 切换简单，且切换时间较短，因为这个切换过程会造成短时间段内 应用无法访问 Master 数据库。
4、 在变更 Master B 的时候，会出现短时间段内 Slave 集群数据的延时，所以如果单台主机的变更时间较长的话，需要在业务量较低的凌晨进行变更。如果有必要，甚至可能需要变更 Master B 之前将所有 Slave 切换为以 Master B 作为 Master。

当然，即使是这样，由于存在 Master A 与 Master B 之间的 VIP 切换，我们仍然会出现短时间段内应用无法进行写入操作的情况。所以说，这种方案也仅仅能够在一定程度上面解决 MySQL 在线 DDL 的问题。而且当集群数量较多，而且每个集群的节点也较多的情况下，整个操作过程将会非常的复杂也很漫长。对于 MySQL 在线 DDL 的问题，目前也确实还没有一个非常完美的解决方案，只能期待 MySQL 能够在后续版本中尽快解决这个问题了。

17.2 利用 MySQL Cluster 实现整体高可用

在上一章中已经详细介绍过 MySQL Cluster 的相关特性，以及安装配置维护方面的内容。这里，主要是介绍一下如何利用 MySQL Cluster 的特性来提高我们系统的整体可用性。

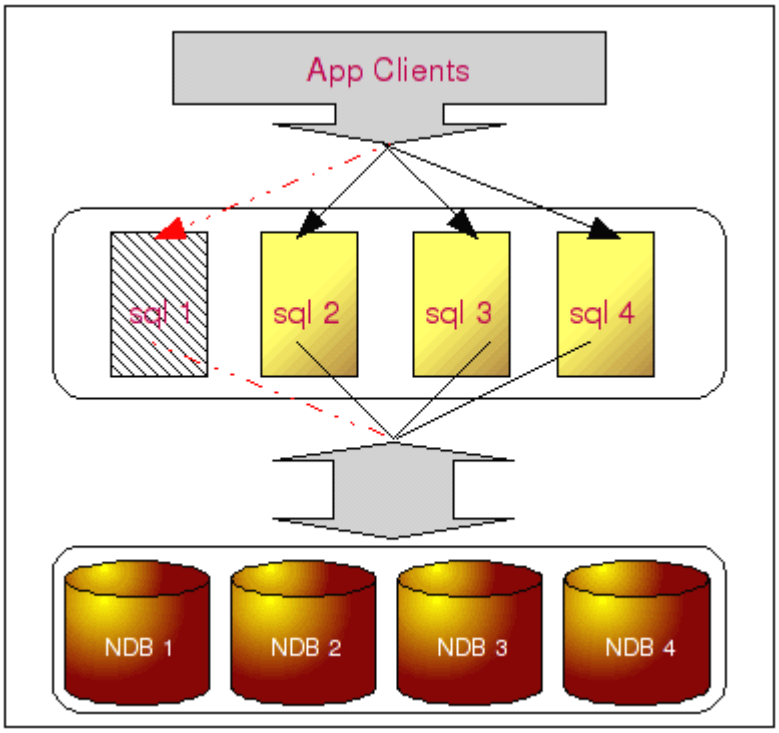
由于 MySQL Cluster 本身就是一个完整的分布式架构的系统，而且支持数据的多点冗余存放，数据实时同步等特性。所以可以说他天生就已经具备了实现高可靠性的条件了，是否能够在实际应用中满足要求，主要就是在系统搭建配置方面的合理设置了。

由于 MySQL Cluster 的架构主要由两个层次两组集群来组成，包括 SQL 节点（mysqld）和 NDB 节点（数据节点），所有两个层次都需要能够保证高可靠性才能保证整体的可靠性。下面我们从两个方面分别来介绍 MySQL Cluster 的高可靠性。

17.2.1 SQL 节点的高可靠性保证

MySQL Cluster 中的 SQL 节点实际上就是一个多节点的 mysqld 服务，并不包含任何数据。所以，SQL 节点集群就像其他任何普通的应用服务器一样，可替代性很高，只要安装了支持 MySQL Cluster 的 MySQL Server 端即可。

当该集群中的一个 SQL 节点 crash 掉之后，由于只是单纯的应用服务，所以并不会造成任何的数据丢失。只需要前端的应用数据源配置兼容了集群中某台主机 crash 之后自动将该主机从集群中去除就可以了。实际上，这一点对于应用服务器来说是非常容易做到的，无论是通过自行开发判断功能的代理还是通过硬件级别的负载均衡设备，都可以非常容易做到。当然，前提自然也是剩下的 SQL 节点能够承担整体负载才行。



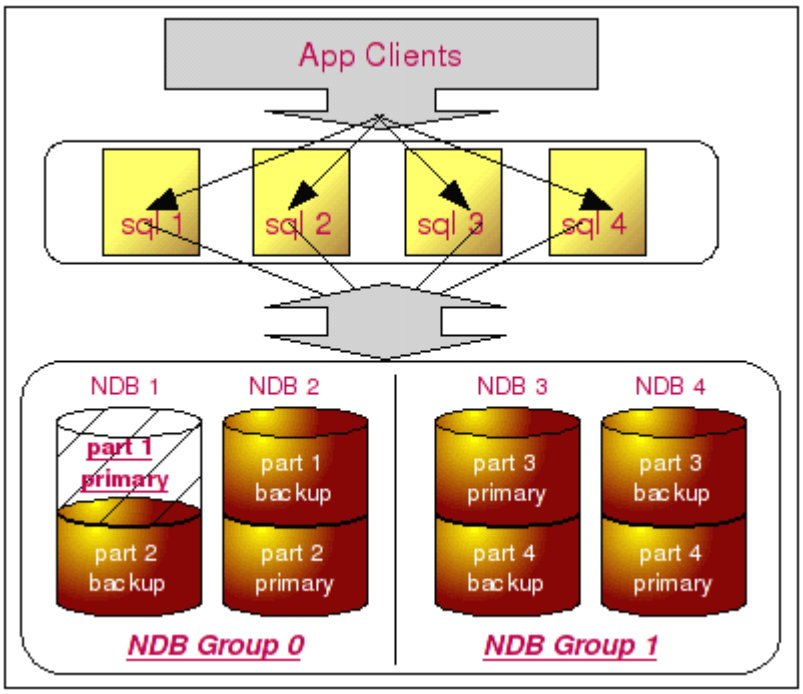
如上图，当 SQL 1 crash 之后，实际上仅仅只是访问到数据的很多条途径中的某一条中断了，实际上仍然还有很多条途径可以获取到所需要的数据。而且，由于 SQL 的可替代性很高，所以更换也非常简单，即使更换整台主机，也可以在短时间内完成。

17.2.2 NDB 节点的高可靠性保证

MySQL Cluster 的数据冗余是有一个前提条件的，首先必须要保证有足够的节点，实际上是至少需要 2 个节点才能保证数据有冗余，因为，MySQL Cluster 在保存冗余数据的时候，是比需要确保同一份数据的冗余存储在不同的节点之上。在保证了冗余的前提下，MySQL Cluster 才会将数据进行分区。

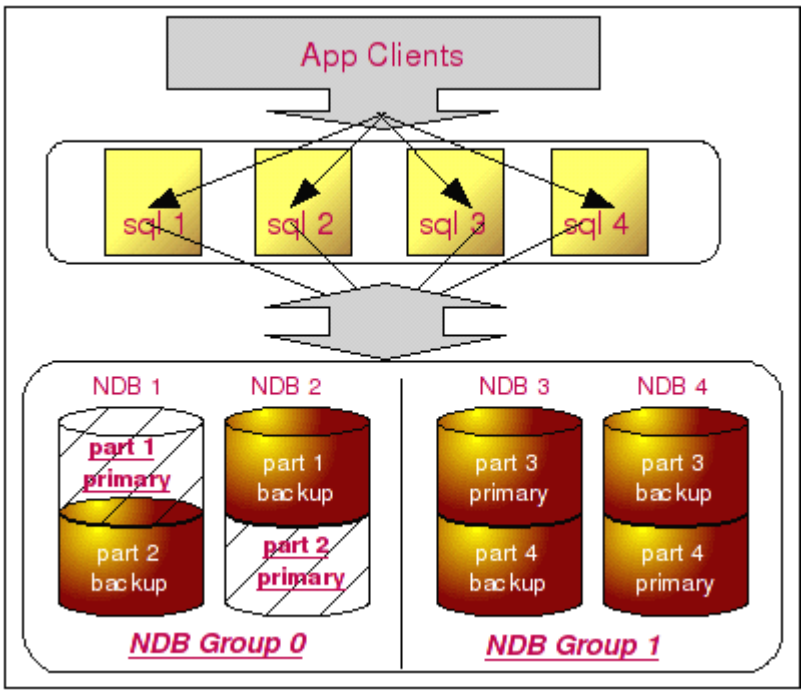
假设我们存在 4 个 NDB 节点，数据被分成 4 个 partition 存放，数据冗余存储，每份数据存储 2 份，也就是说 NDB 配置中的 NoOfReplicas 参数设置为 2，4 个节点将被分成 2 个 NDB Group。

所有数据的分布类似于下图所示：



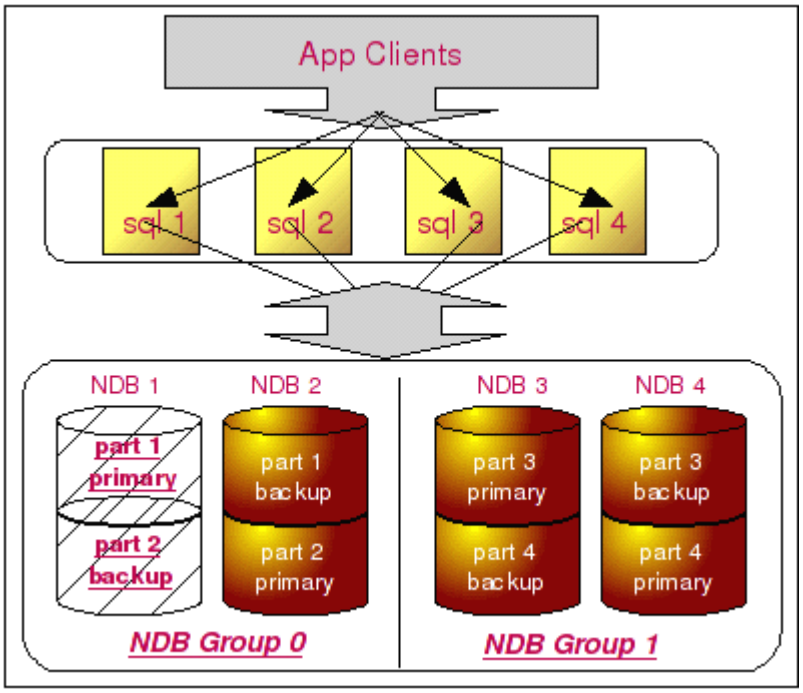
在这样的配置中，假设我们 NDB Group 0 这一组中的某一个 NDB 节点（假如是 NDB 1）出现问题，其中的部分数据（假设是 part 1）坏了，由于每一份数据都存在一个冗余拷贝，所以并不会对系统造成任何的影响，甚至完全不需要人为的操作，MySQL 就可以继续正常的提供服务。

假如我们两个节点上面都出现部分数据损坏的情况，结果会怎样？如下图：



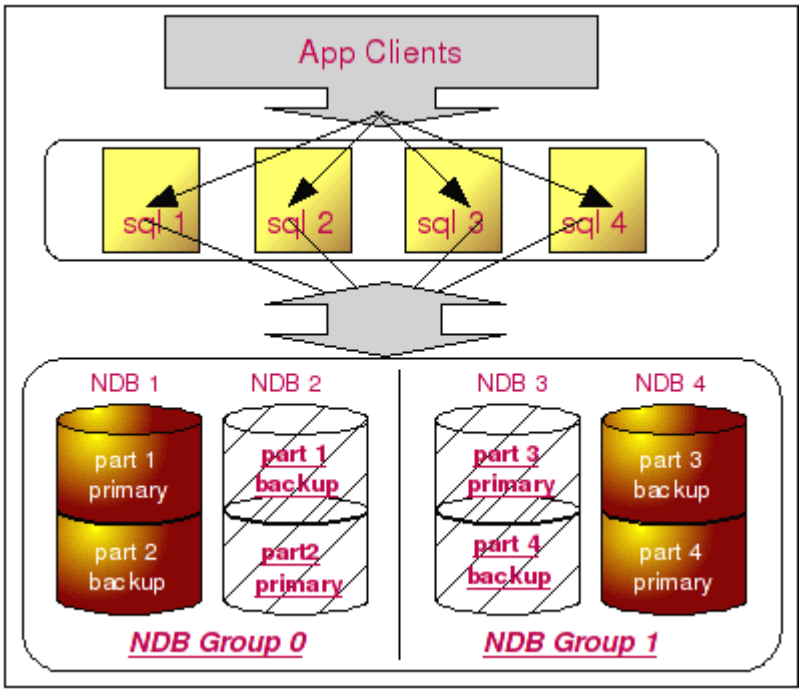
如果像上图所示，如果两个损坏部分数据的节点属于同一个 NDB Group，只要损坏部分并没有包含完全相同的数据，整个 MySQL Cluster 仍然可以正常提供服务。但是，如果损坏数据中存在相同的数据，即使只有很少的部分，都会造成 MySQL Cluster 出现问题，不能完全正常的提供服务。此外，如果损坏数据的节点处于两个不同的 NDB Group，那么非常幸运，不管损坏的是哪一部分数据，都不会影响 MySQL Cluster 的正常服务。

可能有读者朋友会说，那假如我们的硬件出现故障，整个 NDB 都 crash 了呢？是的，确实很可能存在这样的问题，不过我们同样不用担心，如图所示：



假设我们整个 NDB 节点由于硬件(或者软件)故障而 crash 之后，由于 MySQL Cluster 保证了每份数据的拷贝都不在同一台主机上，所以即使整太主机都 crash 了之后，MySQL Cluster 仍然能够正常提供服务，就像上图所示的那样，即使整个 NDB 1 节点都 crash 了，每一份数据都还可以通过 NDB 2 节点找回。

那如果是同时 crash 两个节点会是什么结果？首先可以肯定的是假如我们 crash 的两个节点处于同一个 NDB Group 中的话，那 MySQL Cluster 也没有办法了，因为两份冗余的数据都丢失了。但是只要 crash 的两个节点不在同一个 NDB Group 中，MySQL Cluster 就不会受到任何影响，还是能够继续提供正常服务。如下图所示的情况：



从上面所列举的情况我们可以知道，MySQL Cluster 确实可以达到非常高的可靠性，毕竟同一时刻存放相同数据的两个 NDB 节点都出现大故障的概率实在是太小了，要是这也能够被遇上，那只能自然倒霉了。

当然，由于 MySQL Cluster 之前的老版本需要将所有的数据全部 Load 到内存中才能正常运行，所有由于受到内存空间大小的限制，使用的人非常少。现在的新版本虽然已经支持仅仅只需要所有的索引数据 Load 到内存中即可，但是由于实际的成功案例还并不是很多，而且发展时间也还不是太长，所以很多用户朋友对于 MySQL Cluster 目前还是持谨慎态度，大部分还处于测试阶段。

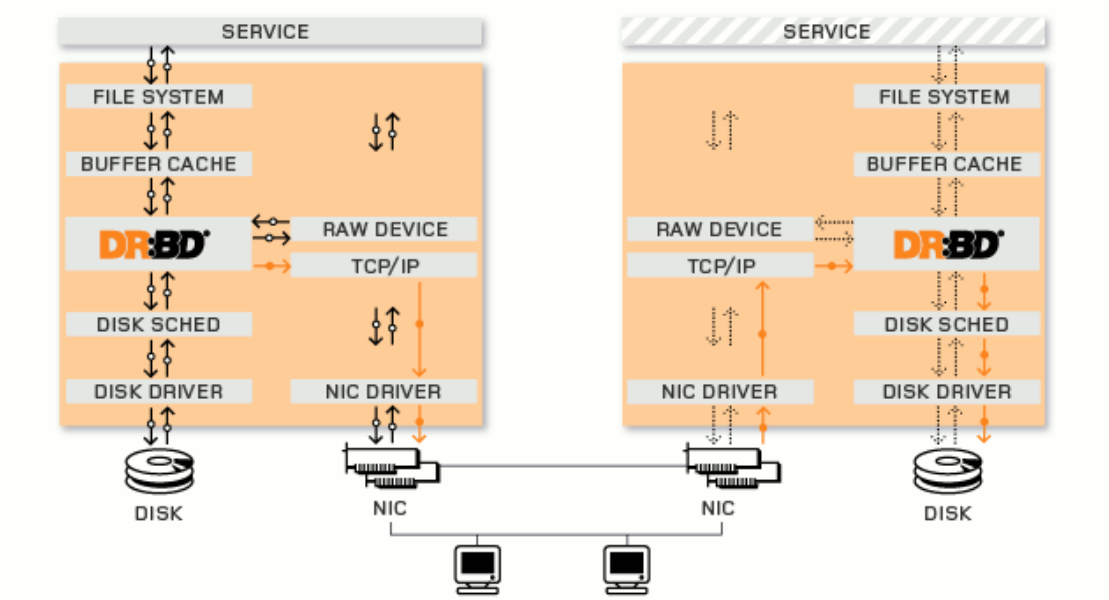
17.3 利用 DRBD 保证数据的高安全可靠

17.3.1 DRBD 介绍

对于很多多这朋友来说，DRBD 的使用可能还不是太熟悉，但多多少少可能有一些了解。毕竟，在 MySQL 的官方文档手册的 High Availability and Scalability 这一章中将 DRBD 作为 MySQL 实现高可用性的一个非常重要的方式来介绍的。虽然这一内容是直到去年年中的时候才开始加入到 MySQL 的文档手册中，但是 DRBD 本身在这之前很久就已经成为很多应用场合实现高可靠性的解决方案，而且在不少的 MySQL 使用群体中也早就开始使用了。

简单来说，DRBD 其实就是通过网络来实现块设备的数据镜像同步的一款开源 Cluster

软件，也被俗称为网络 RAID1。官方英文介绍为：DRBD refers to block devices designed as a building block to form high availability (HA) clusters. This is done by mirroring a whole block device via an assigned network. It is shown as network raid-1- DRBD。下面是 DRBD 的一个概览图：



从图中我们可以看出，DRBD 介于文件系统与磁盘介质之间，通过捕获上层文件系统的所有 IO 操作，然后调用内核中的 IO 模块来读写底层的磁盘介质。当 DRBD 捕获到文件系统的写操作之后，会在进行本地的磁盘写操作的同时，以 TCP/IP 协议将，通过本地主机的网络设备(NIC)将 IO 传递至远程主机的网络设备。当远程主机的 DRBD 监听到传递过来的 IO 信息之后，会立即将该数据写入到该 DRBD 所维护的磁盘设备。至此，整个 IO 才做完成。

实际上 DRBD 在处理远程数据写入的时候有三种复制模式（或者称为级别）可以选择，不同的复制模式保证了远程数据写入的三种可靠性。三种级别的选择可以通过 DRBD 的通用配置部分的 protocal。不同的复制模式，实际上是影响了一个 IO 完成所代表的实际含义。因为当我们使用 DRBD 的时候，一个 IO 完成的标识（DRBD 返回 IO 完成）是本地写入和远程写入这两个并发进程都返回完成标识。下面我来详细介绍一下这三种复制模式所代表的含义：

Protocol A：这种模式是可靠性最低的模式，而且是一个异步的模式。当我们使用这个模式来配置的时候，写远程数据的进程将数据通过 TCP/IP 协议发送进入本地主机的 TCP send buffer 中，即返回完成。

Protocol B：这种模式相对于 Protocol A 来说，可靠性要更高一些。因为写入远程的线程会等待网络信息传输完成，也就是数据已经被远程的 DRBD 接受到之后返回完成。

Protocol C：Protocol C 复制模式是真正完全的同步复制模式，只有当远程的 DRBD 将数据完全写入磁盘成功后，才会返回完成。

对比上面三种复制模式，C 模式可以保证不论出现任何异常，都能够保证两端数据的一致性。而如果使用 B 模式，则可能当远程主机突然断电之后，将丢失部分还没有完全写入磁盘的信息，且本地与远程的数据出现一定的不一致情况。当我们使用 A 复制模式的话，

可能存在的风险就要更大了。只要本地网络设备突然无法正常工作（包括主机断电），就会丢失将写入远程主机的数据，造成数据不一致现象。

由于不同模式所要求的远程写入进程返回完成信息的时机不一样，所以也直接决定了IO 写入的性能。通过三个模式的描述，我们可以很清楚的知道，IO 写入性能与可靠程度成反比。所以，各位读者朋友在考虑设置这个参数的时候，需要仔细评估各方面的影响，尽可能得到一个既满足实际场景的性能需求，又能满足对可靠性的要求。

DRBD 的安装部署以及相关的配置说明，在 MySQL 的官方文档手册以及我的个人网站博客（http://www.jianzhaoyang.com）中都有相关的描述，而且在 DRBD 的官方网站上面也可以找到最为全面的说明，所以这里就不再介绍了。下面我主要介绍一下 DRBD 的一些特殊的特性以及限制，以供大家参考。

17.3.2 DRBD 特性与限制

DRBD 之所以能够得到广泛的采用，甚至被 MySQL 官方写入文档手册作为官方推荐的高可用方案之一，主要是其各种高可靠特性以及稳定性的缘故。下面介绍一下 DRBD 所具备的一些比较重要的特性。

1、 非常丰富的配置选项，可以适应我们的应用场景中的情况。无论是可靠性级别与性能的平衡，还是数据安全性方面，无论是本地磁盘，还是网络存储设备，无论是希望异常自动解决，还是希望手动控制等等，都可以通过简单的配置文件就可以解决。当然，丰富的配置在带来极大的灵活性的同时，也要求使用者需要对他有足够的了解才行，否则在那么多配置参数中也很难决策该如何配置。幸好 DRBD 在默认情况下的各项参数实际上就已经满足了大多数典型需求了，并不需要我们每一项参数都设置才能运行；

2、 对于节点之间出现数据不一致现象，DRBD 可以通过一定的规则，进行重新同步。而且可以通过相关参数配置让 DRBD 在固定时间点进行数据的校验比对，来确定数据是否一致，并对不一致的数据进行标记。同时还可以选择是 DRBD 自行解决不一致问题还是由我们手工决定如何同步；

3、 在运行过程中如果出现异常导致一端 crash，并不会影响另一端的正常工作。出现异常之后的所有数据变更，都会被记录到相关的日志文件中。当 crash 节点正常恢复之后，可以自动同步这段时间变更过的数据。为了不影响新数据的正常同步，还可以设定恢复过程中的速度，以确保网络和其他设备不会出现性能问题；

4、 多种文件系统类型的支持。DRBD 除了能够支持各种常规的文件系统之外，还可以支持 GFS，OCFS2 等分布式文件系统。而且，在使用分布式文件系统的时候，还可以实现各结带你同时提供所有 IO 操作。

5、 提供对 LVM 的支持。DRBD 既可以使用由 LVM 提供的逻辑设备，也可以将自己对外提供的设备成为 LVM 的物理设备，这极大的方便的运维人员对存储设备的管理。

6、 所有 IO 操作，能够绝对的保证 IO 顺序。这也是对于数据库来说非常重要的一个特性尤其是一些对数据一致性要求非常苛刻的数据库软件来说。

7、 可以支持多种传输协议，从 DRBD 8.2.7 开始，DRBD 开始支持 Ipv4，IPv6 以及 SuperSockets 来进行数据传输。

8、 支持 Three-Way 复制。从 DRBD 8.3.0 开始，DRBD 可以支持三个节点之间的复制了，有点类似于级联复制的特性。

当然，DRBD 在拥有大量让人青睐的特性的同时，也存在一定的限制，下面就是 DRBD 目前存在的一些比较重要的限制：

1、 对于使用常规文件系统（非分布式文件系统）的情况下，DRBD 只能支持单 Primary 模式。在单 Primary 模式下，只有一个节点的数据是可以对外提供 IO 服务的。只有当使用 GFS 或者 OCFS2 这样的分布式文件系统的时候，才能支持 Dual Primary 模式。
2、 Sblit Brain 的解决。因为某些特殊的原因，造成两台主机之间的 DRBD 连接中断之后双方都以 Primary 角色来运行之后的处理还不是太稳定。虽然 DRBD 的配置文件中可以配置自动解决 Split Brain，但是从我之前的测试情况来看，并不是每次的解决都非常令人满意，在有些情况下，可能出现某个节点完全失效的可能。
3、 复制节点数目的限制，即使是目前最新的 DRBD 版本来说，也最多只支持三个节点之间的复制。

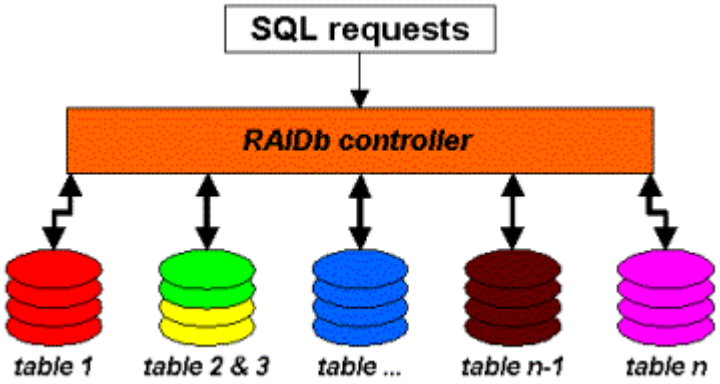
以上几个限制是在目前看来可能对使用者造成较大影响的几个限制。当然，DRBD 还存在很多其他方面的限制，大家在决定使用之前，还是需要经过足够测试了解，以确保不会造成上线后的困扰。

17.4 其他高可用设计方案

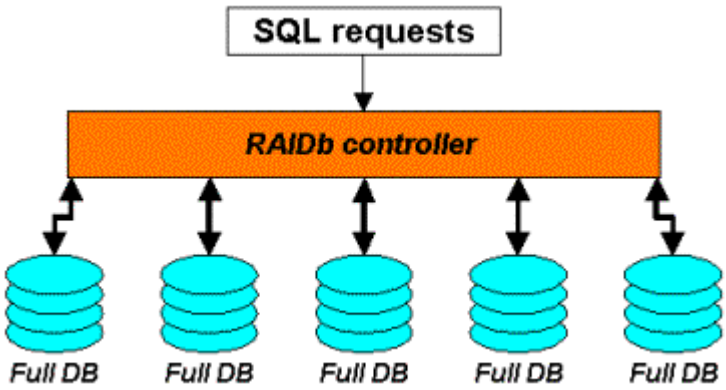
除了上面几种高可用方案之外，其实还有不少方案可以供大家选择，如 RaiDB，共享存储解（SAN 或者 NAS） 等等。

对于 RaiDB，可能很多读者朋友还比较陌生，其全称为 Redundant Arrays of Inexpensive Databases。也就是通过 Raid 理念来管理数据库的数据。所以 RaiDB 也存在数据库 Sharding 的概念。和磁盘 Raid 一样，RaiDB 也存在多种 Raid，如 RaiDB-0，RaiDB-1，RaiDB-2，RaiDB-0-1 和 RaiDB-1-0 等。商业 MySQL 数据库解决方案提供商 Continuent 的数据库解决方案中，就利用了 RaiDB 的概念。大家可以通过一下几张图片清晰的了解 RaiDB 的各种 Raid 模式。

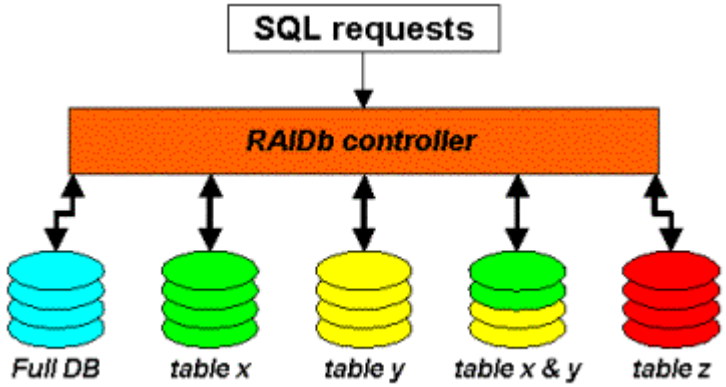
RaiDB-0:



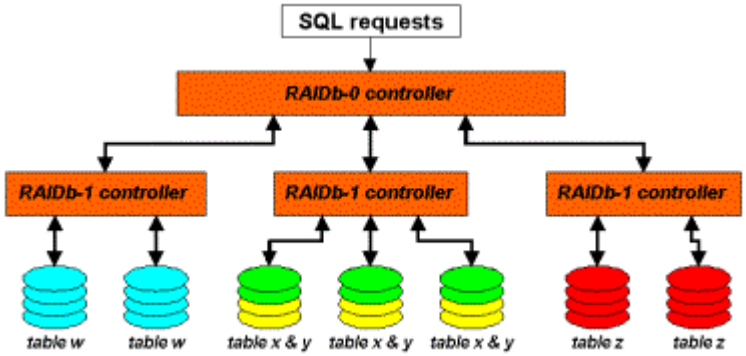
RaiDB-1:



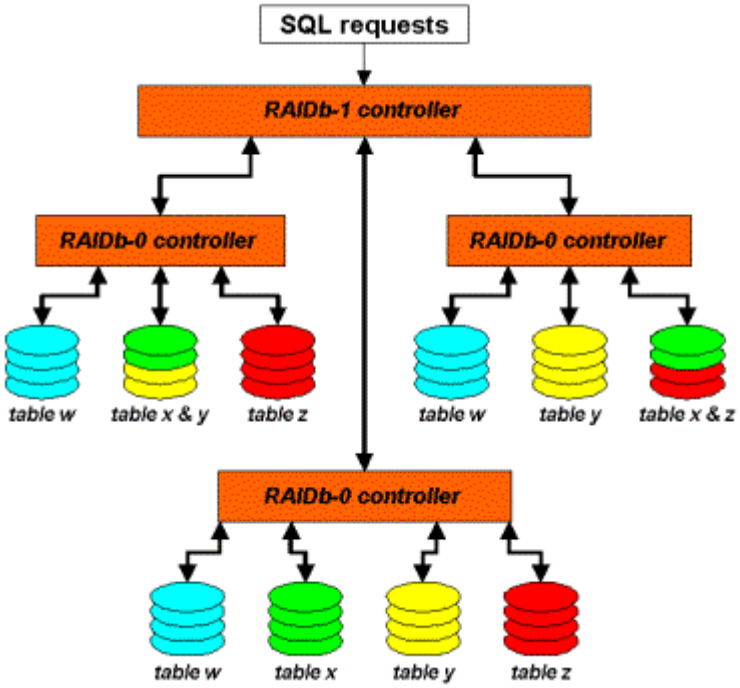
RaiDB-2:



RaiDB-0-1:



RaiDB-1-0:



从图中的标识，大家应该就比较清楚 RaiDB 各种 Raid 模式下的数据如何分布以及工作方式了吧。至于为什么这样作可以保证高可用性，就和磁盘通过 Raid 来保证数据高可靠性一样。当然，要真正理解，前提还是大家已经具备了清晰的 Raid 概念。

而对于共享存储的解决方案，则是一个相对来说比较昂贵的解决方案了。运行 MySQL Server 的主机上面并不存放数据，只不过通过共享协议（FC 或者 NFS）来将远程存储设备上面的磁盘空间 Mount 到本地，当作本地磁盘来使用。

为什么共享存储的解决方案可以满足高可靠性要求呢？首先，数据不是存在 MySQL Server 本地主机上面，所以当本地 MySQL Server 主机出现任何故障，都不会造成数据的丢失，完全可以通过其他主机来找回存储设备上面的数据。其次，无论是通过通过 SAN 还是 NAS 来作为共享存储，存储本身具备多种高可用解决方案，可以做到完全不丢失任何数据。甚至即使 MySQL Server 与共享存储之间的连接通道，也有很多成熟的高可用解决方案，来保证连接的高可用性。由于共享存储的解决方案本身违背了我个人提倡的通过 MySQL 构建廉价的企业级高性能高可靠性方案，又考虑到篇幅问题，所以这里就不详细深入介绍了。如果读者朋友对这方面比较感兴趣，可以通过其他图书再深入了解 SAN 存储以及 NAS 存储相关的知识。

17.5 各种高可用方案的利弊比较

从前面各种高可用设计方案的介绍中读者们可能已经发现，不管是哪一种方案，都存在自己独特的优势，但也都或多或少的存在一些限制。其实这也是很正常的，毕竟任何事物都不可能是完美的，我们只能充分利用各自的优势来解决自己的问题，而不是希望依赖某种方案一劳永逸的解决所有问题。这一节将针对上面的几种主要方案做一个利弊分析，以供大家

选择过程中参考。

1、 MySQL Replication

优势：部署简单，实施方便，维护也不复杂，是 MySQL 天生就支持的功能。且主备机之间切换方便，可以通过第三方软件或者自行编写简单的脚本即可自动完成主备切换。

劣势：如果 Master 主机硬件故障且无法恢复，则可能造成部分未传送到 Slave 端的数据丢失；

2、MySQL Cluster

优势：可用性非常高，性能非常好。每一分数据至少在不同主机上面存在一份拷贝，且冗余数据拷贝实时同步。

劣势：维护较为复杂，产品还比较新，存在部分 bug，目前还不一定适用于比较核心的线上系统。

3、DRBD 磁盘网络镜像方案

优势：软件功能强大，数据在底层快设备级别跨物理主机镜像，且可根据性能和可靠性要求配置不同级别的同步。IO 操作保持顺序，可满足数据库对数据一致性的苛刻要求。

劣势:非分布式文件系统环境无法支持镜像数据同时可见,性能和可靠性两者相互矛盾，无法适用于性能和可靠性要求都比较苛刻的环境。维护成本高于 MySQL Replication。

17.6 小结

本章重点针对 MySQL 自身具备的两种高可用解决方案以及 MySQL 官方推荐的 DRBD 解决方案做了较为详细的介绍，同时包括了各解决方案的利弊对比。希望这些信息能够给各位读者带来一些帮助。不过，MySQL 的高可用解决方案远远不只上面介绍过的集中方案，还存在着大量的其他方案可供各位读者朋友进行研究探索。开源的力量是巨大的，开源贡献者的力量更是无穷的。

第 18 章 高可用设计之 MySQL 监控

前言：

一个经过高可用可扩展设计的 MySQL 数据库集群，如果没有一个足够精细足够强大的监控系统，同样可能会让之前在高可用设计方面所做的努力功亏一篑。一个系统，无论如何设计如何维护，都无法完全避免出现异常的可能，监控系统就是根据系统的各项状态的分析，让我们能够尽可能多的提前预知系统可能会出现的异常状况。即使没有及时发现将要发生的异常，也要在异常出现后的第一时间知道系统已经出现异常，否则之前的设计工作很可能就白费了。

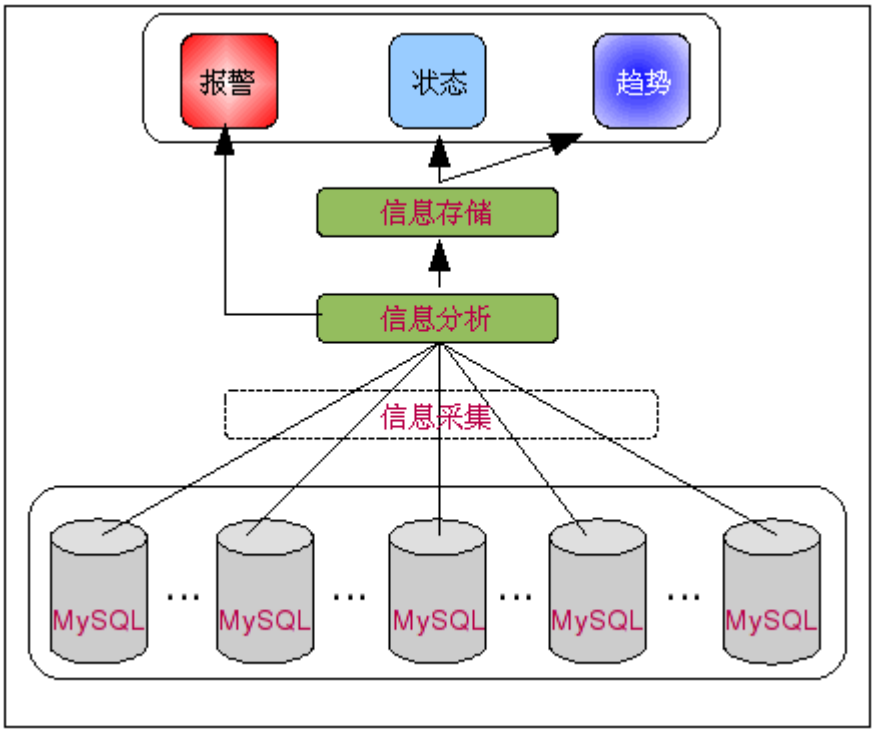
18.1 监控系统设计

系统监控在很多人眼中是一个没有多少技术含量的事情，其实并不是这样的。且不说一个大型网站的所有设备的监控，就仅仅只是搭建一个比较完善的几十台 MySQL 集群系统的监控，很可能就会让很多人束手无策，或者功能不够完善。

其实一个完善的大型集群系统的监控系统，和少数几台主机的监控是有很大差别的。少数机台主机的监控在大多数时候可以通过几个简单的脚本（Shell 或者 Perl 等），发发邮件，再高一点的报警信息发发手机短信，基本就搞定了。监控点少，发出的信息量也少。很多状态信息甚至都可以通过维护人员登录到主机上面来定时 Check 即可。可如果有上百台主机，仍然仅仅依靠在每台主机上面部署几个简单的脚本的方式来进行监控，后期的管理维护成本就会非常高了。

当 MySQL 主机达到一定规模之后，我们基本上很难通过人工到各个主机上面来定时检查各自的状态，不论是运行状态还是性能状态。甚至都不能像只有少数 MySQL 主机的时候那样简单的通过发送邮件的方式将相关信息发出。毕竟，量大了之后，检查邮件的时间成本也是很大的。这时候就需要我们进行统一的监控信息采集、分析、存储、处理系统来帮助我们过滤掉可以忽略的正常信息，并画出相关信息的趋势图，以帮助判断系统的运行状况和发展趋势。

所以，MySQL 分布式集群的监控系统整体架构体系应该如下图所示：



1、 信息采集

信息采集可以是一个统一的模块以主动的方式从各个 MySQL 主机上面来获取信息然后存放到监控信息存储中心，也可以是分布在各个 MySQL 主机上面的模块以被动的方式来反向将相关数据推向监控信息存储中心。

一般来说，较小规模的监控点可以采用轮询的方式主动采集数据，但是当监控点达到一定规模以后，轮询的主动采集方式可能就会遇到一定的性能瓶颈和信息延时问题，尤其是当需要采集的数据比较多的时候尤为突出。而如果要采用从各个 MySQL 节点进行被动的推送，则可能需要开发能够支持网络通信的监控程序，使采集的信息能够顺利的到达信息分析模块以即时得到分析，成本会稍微高一些。

不论是采用主动还是被动的方式来进行数据采集，我们都需要在监控主机上面部署采集相关信息的程序或脚本，包括主机信息和数据库信息。

2、 信息分析

当前端采集模块的监控程序获取到 MySQL 主机当前的各种状态信息之后，分析模块需要实时地对数据进行分析，识别出系统是正常还是异常，如果是异常，就必须通过相关机制立即发出报警通知，并将信息发送到存储模块进行持久化。如果正常，则不需要进行任何处理就可以将数据传递到存储模块持久化。对信息分析模块来讲，最重要的事情（？要求）就是处理及时、准确，当监控点到达一定的量之后，性能可能会成为瓶颈。

3、 信息存储

存储监控程序采集的信息也是一件非常重要的事情，因为不论是查看当前状态，还是绘制趋势图，都需要用到这些信息。此外还有一个非常重要的原因就是通过对积累下来的这些监控信息的分析挖掘，为系统的容量规划和性能模型设计带来非常大的帮助。

4、 信息处理

最后根据采集到的各种状态、性能信息，通过应用相应规则进行分析挖掘运算，绘制出各种状态的趋势图以供维护人员查看。此外，通过对各种趋势的分析，发掘出业务发展与数据库负载之间的关系，以及与主机硬件的关系。这些关系数据，对于系统发展规划的制定将是非常有意义的。

18.2 健康状态监控

健康状态信息一般来说还是比较简单的，但也是非常重要的。因为对于监控来说，需要了解的健康状态基本上也就只有“正常”或“不正常”这两种。下面分别从主机状态信息和数据库状态信息来分别介绍一下。

18.2.1 主机健康状态监控（？）

在数据库运行环境，需要关注的主机状态主要有网络通信、系统软硬件错误、磁盘空间 、内存使用，进程数量等。

- 网络通信：网络通信基本上可以说是最容易检测的了，基本上只需要通过网络 ping 就可以获知是否正常。如果还不放心，或者所属网段内禁用了 ping，也可以从监控主机进行固定端口的 telnet 尝试或者 ssh 登录尝试。由于网络出现故障的时候被动的信息采集方式也会失效（无法与外界通信），所以网络通信的检测主要还是依靠主动检测。

- 系统软硬件错误：系统软硬件错误，一般只能通过检测各种日志文件的信息来实现，如主机的系统日志中基本上都会记录下 OS 能够检测到的大部分错误信息，如硬件错误，IO 错误等等。我们一般使用文本监控软件，如 sec、logwatch 等日志监控专用软件，通过配置相应的匹配规则，从日志文件中捕获满足条件的错误信息，再发送给信息分析模块。

- 磁盘空间：对于磁盘空间的使用状况监控，我们通过最简单的 shell 脚本就可以轻松搞定，得出系统中各个分区的当前使用量，剩余可用空间。积累一定时间段的信息之后，很容易就能得出系统数据量增长趋势。

- 内存使用：系统物理内存使用量的信息采集同样非常简单，只需要一个基本的系统命令“free"，就可以获得当前系统内存总量，剩余使用量，以及文件系统的 buffer 和 cache 两者使用量。而且，除了物理内存使用情况，还可以得到 swap 使用量。

通过 shell 脚本对这些输出信息进行简单的处理，即可获得足够的信息。当然，如果你希望获取更多的信息，如当前系统使用内存最多的进程，可能还需要借助其他命令（如：top）才能得到。当然，不同的 OS 在输出值的处理方面可能会稍有区别，

- 进程数量：系统进程总数，或者某个用户下的进程数，都可以通过“ps”命令经过简单的处理来获得。如获取 mysql 用户下的进程总数：

  ```
  ps -ef | awk '{print $1}' | grep "mysql" | grep -v "grep" | wc -l
  ```

  如要获得更为详细的某个或者某类进程的信息，同样可以根据上述类似命令得到。

18.2.2 数据库健康状态信息

除了主机的状态信息之外，MySQL Server 自身也有很多的状态信息需要监控。下面就详细介绍一下 MySQL Server 需要监控的内容以及监控方法。

- 服务端口（3306）：MySQL 端口是一个必不可少的监控项，因为这直接反应了 MySQL Server 是否能够正常为外部请求提供服务。有些时候从主机层面来检查 MySQL 可能很正常，可外部应用却无法通过 TCP/IP 连接上 MySQL。产生这种现象的原因可能有多种，如网络防火墙的问题，网络连接问题，MySQL Server 所在主机的网络设置问题，以及 MySQL 本身问题都可能造成上述现象。
  服务端口状态的监控和主机网络连接的监控同样非常简单，只需要对 3306 端口进行 telnet 尝试即可。通过对 3306 端口的 telnet 尝试，同时还可以完成对主机网络状况的监控。

- socket 文件：对于有些环境来说，socket 的状态监控可能并不如网络服务端口的监控重要，因为很多 MySQL Server 的连接并不会通过本地 socket 进行连接。但是也有不少小型应用（或者某些特殊的应用）还是和 MySQL 数据库处于同一台主机上，并通过本地 socket 连接。另外，不少本地被动监控的信息采集脚本也是通过本地 socket 来连接 MySQL Server。
  本地 socket 的监控最好是通过本地 socket 的实际连接尝试的方式，虽然也可以通过检测 socket 文件是否存在来监控，但是即使 socket 文件确实存在，也并不一定就可以确保能够通过 socket 正常登录。毕竟，在某些异常情况下，socket 文件存在并不代表就可以正常使用。

- mysqld 和 mysqld_safe 进程：mysqld 进程是 MySQL Server 最核心的进程。mysqld 进程 crash 或者出现异常，MySQL Server 基本上也就无法正常提供服务了。当然，如果我们是通过 mysqld_safe 来启动 MySQL Server，则 mysqld_safe 会帮助我们来监控 mysqld 进程的状态，当 mysqld 进程 crash 之后，mysqld_safe 会马上帮助我们重启 mysqld 进程。但前提是我们必须通过 mysqld_safe 来启动 MySQL Server，这也是 MySQL AB 强烈推荐的做法。如果我

们通过 mysqld_safe 来启动 MySQL Server，那么我们也必须对 mysqld_safe 进程进行监控。
无论是 mysqld 还是 mysqld_safe 进程的监控,都可以通过 ps 命令来得到实现:

```
ps -ef | grep "mysqld_safe" | grep -v "grep"
```

和

```
ps -ef | grep "mysqld" | grep -v "mysqld_safe"| grep -v "grep"
```

- Error log：Error log 的监控目的主要是即时检测 MySQL Server 运行过程中发生的各种错误，如连接异常，系统 bug 等。
  Error log 的监控和系统软硬件状态的监控一样,都是通过对文本文件内容的监控来实现的。同样使用文本监控软件，通过配置各种错误信息的匹配规则来捕获日志文件中的错误信息。

- 复制状态：如果我们的 MySQL 数据库环境使用了 MySQL Replication，就必须增加对 Slave 复制状态的监控。对 Slave 的复制状态的监控包括对 IO 线程和 SQL 线程二者的运行状态的监控。当然，如果希望能够监控 Replication 更多的信息,如两个线程各自运行的进度等,同样可以在 Slave 节点上执行相应命令轻松得到,如下：

```
sky@localhost : (none) 04:30:38> show slave status\G
*************************** 1. row ***************************
               Slave_IO_State: Connecting to master
                  Master_Host: 10.0.77.10
                  Master_User: repl
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000001
          Read_Master_Log_Pos: 196
               Relay_Log_File: mysql-relay-bin.000001
                Relay_Log_Pos: 4
        Relay_Master_Log_File: mysql-bin.000001
             Slave_IO_Running: No
            Slave_SQL_Running: Yes
              Replicate_Do_DB: example,abc
          Replicate_Ignore_DB: mysql,test
           Replicate_Do_Table:
       Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table:
                   Last_Errno: 0
                   Last_Error:
                 Skip_Counter: 0
          Exec_Master_Log_Pos: 196
```

```
                  Relay_Log_Space: 106
                  Until_Condition: None
                   Until_Log_File:
                    Until_Log_Pos: 0
               Master_SSL_Allowed: No
               Master_SSL_CA_File:
               Master_SSL_CA_Path:
                  Master_SSL_Cert:
                Master_SSL_Cipher:
                   Master_SSL_Key:
          Seconds_Behind_Master: NULL
Master_SSL_Verify_Server_Cert: No
                Last_IO_Errno: 0
                Last_IO_Error:
               Last_SQL_Errno: 0
               Last_SQL_Error:
```

通过如上命令，我们可以获取获取关于 Replication 的各种信息，不论是复制的监控状况，还是复制的进度，都可以很容易的获取。上面输出的各项内容中，最重要的内容就是 Slave_IO_Running 和 Slave_SQL_Running 这两项，分别代表了 IO 线程和 SQL 线程的运行状态，如上面的输出就表明 SQL 线程运行正常，但是 IO 线程并没有运行。其他各项的详细解释，请参考本书附录。

-

18.3 性能状态监控

性能状态的监控和健康状态的监控有一定的区别，他所反应出来的主要是系统当前提供服务的响应速度，影响的更多是客户满意度。性能状态的持续恶化，则可能升级为系统不可用，也就是升级为健康状态的异常。如系统的过度负载，可能导致系统的 crash。性能状态的监控同样可以分为主机和数据库层面。

18.3.1 主机性能状态

主机性能状态主要表现在三个方面：CPU、IO 以及网络，可以通过以下五个监控量来监控。

- 系统 load 值：系统 load 所包含的最关键含义是 CPU 运行等待的数量，也就是侧面反应了 CPU 的繁忙程度。只不过 load 值并不直接等于等待队列中的进程数量。

load 值的监控也非常简单，通过运行 uptime 命令即可获得当前时间之前 1 秒、5 秒和 15 秒的 load 平均值。

```
sky@sky:~$ uptime
17:27:44 up  4:12,  3 users,  load average: 0.87, 0.66, 0.61
```

如上面输出内容中的“load average: 0.87, 0.66, 0.61”中的三个数字，分别代表了 1 秒、5 秒和 15 秒的 load 平均值。
一般来说，load 值在不超过系统物理 cpu 数目（或者 cpu 总核数）之前，系统不会有太大问题。

- CPU 使用率：CPU 使用率和系统 load 值一样，从另一个角度反应出 CPU 的总体繁忙程度，只不过可以反应出更为详细的信息，如当前空闲 的 CPU 比率，系统占用的 CPU 比率，用户进程占用的 CPU 比率，处于 IO 等待的 CPU 比率等。
  CPU 使用率可以通过多种方法来获取，最为常用的方法是使用命令 top 和 vmstat 来获取 。当然，不同的 OS 系统上的 top 和 vmstat 的输出可能有些许不同，且输出格式也可能各不一样，如 Linux 下的 top 包含 IO 等待的 CPU 占用律，但是 Solaris 下的 top 就不包含，各位读者朋友请根据自己的 OS 环境进行相应的处理。

- 磁盘 IO 量：磁盘 IO 量直接反应出了系统磁盘繁忙程度，对于数据库这种以 IO 操作为主的系统来说，IO 的负载将直接影响到系统的整体响应速度。
  磁盘 IO 量同样也可以通过 vmstat 来获取，当然，我们还可以通过 iostat 来获得更为详细的系统 IO 信息。包括各个磁盘设备的 iops，每秒吞吐量等。

- swap 进出量：swap 的使用主要表现了系统在物理内存不够的情况下使用虚拟内存的情况。当然，有些时候即使系统还有足够物理内存的时候，也可能出现使用 swap 的情况，这主要是由系统内核中的内存管理部分来决定。如果希望系统完全不使用 swap，最直接的办法就是关闭 swap。当然，前提条件是我们必须要有足够的物理内存，否则很可能会出现内存不足的错误，严重的时候可能会造成系统 crash。
  swap 使用量的总体情况可以通过 free 命令获得，但 free 命令只能获得当前系统 swap 的总体使用量。如果希望获得实时的 swap 使用变化，还是得依赖 vmstat 来得到。vmstat 输出信息中包含了每秒 swap 的进出量，当然，不同的 OS 输出可能存在一定的差异。

- 网络流量：作为数据库系统，网络流量也是一个不容忽视的监控点。毕竟数据库系统的数据进出比普通服务器的量还是要大很多的。不论是总体吞吐量还是网络 iops，都需要给予一定的关注。当然，一般非数据仓库类型的数据库，网络流量成为瓶颈的可能性还是比较小的。
  网络流量的监控很少有系统自带的命令可以直接完成，而需要自行编写脚本或者通过一些第三方软件来获取数据。当然，通过对网络交换机的监控，也可以获得非常详细的网络流量信息。自行编写脚本可以通过调用 ifconfig 命令来计算出基本准确的网络流量信息。第三方软件如 ifstat、iftop 和 nload 等则是需要另外安装的监控 linux 下网络流量第三方软件，三者各有特点，读者朋友可以根据三款软

件官方介绍的功能特点自行选择。

18.3.2 数据库性能状态

MySQL 数据库的性能状态监控点非常之多，其中很多量都是我们不能忽视的必须监控的量，且 90% 以上的内容可以在连接上 MySQL Server 后执行“SHOW /*!50000 GLOBAL */ STATUS” 以及 “SHOW /*!50000 GLOBAL */ VARIABLES” 的输出值获得。需要注意的是上述命令所获得状态值实际上是累计值，所以如果要计算(单位/某个)时间段内的变化量还需要稍加处理，可以在附录中找到两个命令输出值的详细说明。下面看看几项需要重点关注的性能状态：

- QPS（每秒 Query 量）：这里的 QPS 实际上是指 MySQL Server 每秒执行的 Query 总量，在 MySQL 5.1.30 及以下版本可以通过 Questions 状态值每秒内的变化量来近似表示，而从 MySQL 5.1.31 开始，则可以通过 Queries 来表示。Queries 是在 MySQL 5.1.31 才新增的状态变量。主要解决的问题就是 Questions 状态变量并没有记录存储过程中所执行的 Query（当然，在无存储过程的老版本 MySQL 中则不存在这个区别），而 Queries 状态变量则会记录。二者获取方式：

  QPS = Questions(or Queries) / Seconds

  获取所需状态变量值：
  SHOW /*!50000 GLOBAL */ STATUS LIKE 'Questions'
  SHOW /*!50000 GLOBAL */ STATUS LIKE 'Queries'

  这里的 Seconds 是指累计出上述两个状态变量值的时间长度，后面用到的地方也代表同样的意思。

- TPS（每秒事务量）： 在 MySQL Server 中并没有直接事务计数器，我们只能通过回滚和提交计数器来计算出系统的事务量。所以，我们需要通过以下方式来得到客户端应用程序所请求的 TPS 值：

  TPS = (Com_commit + Com_rollback) / Seconds

  如果我们还使用了分布式事务，那么还需要将 Com_xa_commit 和 Com_xa_rollback 两个状态变量的值加上。

- Key Buffer 命中率：Key Buffer 命中率代表了 MyISAM 类型表的索引的 Cache 命中率。该命中率的大小将直接影响 MyISAM 类型表的读写性能。Key Buffer 命中率实际上包括读命中率和写命中率两种，MySQL 中并没有直接给出这两个命中率的值，但是可以通过如下方式计算出来：

  key_buffer_read_hits = (1 - Key_reads / Key_read_requests) * 100%

key_buffer_write_hits= (1 - Key_writes / Key_write_requests) * 100%

获取所需状态变量值：

```
sky@localhost : (none) 07:44:10> SHOW /*!50000 GLOBAL */ STATUS
    -> LIKE 'Key%';
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
    ... ...
| Key_read_requests    | 10    |
| Key_reads            | 4     |
| Key_write_requests   | 0     |
| Key_writes           | 0     |
+----------------------+-------+
```

通过这两个计算公式，我们很容易就可以得出系统当前 Key Buffer 的使用情况

- Innodb Buffer 命中率：这里 Innodb Buffer 所指的是 innodb_buffer_pool，也就是用来缓存 Innodb 类型表的数据和索引的内存空间。类似 Key buffer，我们同样可以根据 MySQL Server 提供的相应状态值计算出其命中率：

innodb_buffer_read_hits=(1-Innodb_buffer_pool_reads/Innodb_buffer_pool_read_requests) * 100%

获取所需状态变量值：

```
sky@localhost : (none) 08:25:14> SHOW /*!50000 GLOBAL*/ STATUS
    -> LIKE 'Innodb_buffer_pool_read%';
+---------------------------------+-------+
| Variable_name                   | Value |
+---------------------------------+-------+
    ... ...
| Innodb_buffer_pool_read_requests | 5367  |
| Innodb_buffer_pool_reads         | 507   |
+---------------------------------+-------+
```

- Query Cache 命中率：如果我们使用了 Query Cache，那么对 Query Cache 命中率进行监控也是有必要的，因为他可能告诉我们是否在正确的使用 Query Cache。Query Cache 命中率的计算方式如下：

Query_cache_hits= (Qcache_hits / (Qcache_hits + Qcache_inserts)) * 100%

获取所需状态变量值：

```
sky@localhost : (none) 08:32:01> SHOW /*!50000 GLOBAL*/ STATUS
-> LIKE 'Qcache%';
+----------------------+-------+
```

```
| Variable_name           | Value |
+-------------------------+-------+
    ... ...
| Qcache_hits             | 0     |
| Qcache_inserts          | 0     |
    ... ...
+-------------------------+-------+
```

- Table Cache 状态量：Table Cache 的当前状态量可以帮助我们判断系统参数 table_open_cache 的设置是否合理。如果状态变量 Open_tables 与 Opened_tables 之间的比率过低，则代表 Table Cache 设置过小，个人认为该值处于 80% 左右比较合适。注意，这个值并不是准确的 Table Cache 命中率。

  获取所需状态变量值：

```
sky@localhost : (none) 08:52:00> SHOW /*!50000 GLOBAL*/ STATUS
-> LIKE 'Open%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
    ... ...
| Open_tables             | 51    |
    ... ...
| Opened_tables           | 61    |
+-------------------------+-------+
```

- Thread Cache 命中率：Thread Cache 命中率能够直接反应出我们的系统参数 thread_cache_size 设置的是否合理。一个合理的 thread_cache_size 参数能够节约大量创建新连接时所需要消耗的资源。Thread Cache 命中率计算方式如下：

  Thread_cache_hits = (1 - Threads_created / Connections) * 100%

  获取所需状态变量值：

```
sky@localhost : (none) 08:57:16> SHOW /*!50000 GLOBAL*/ STATUS
-> LIKE 'Thread%';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
    ... ...
| Threads_created | 3     |
    ... ...
+-----------------+-------+
4 rows in set (0.01 sec)

sky@localhost : (none) 09:01:33> SHOW /*!50000 GLOBAL*/ STATUS
    -> LIKE 'Connections';
```

```
+--------------+-------+
| Variable_name | Value |
+--------------+-------+
| Connections   | 11    |
+--------------+-------+
```

正常来说，Thread Cache 命中率要在 90% 以上才算比较合理。

- 锁定状态：锁定状态包括表锁和行锁两种，我们可以通过系统状态变量获得锁定总次数，锁定造成其他线程等待的次数，以及锁定等待时间信息。

```
sky@localhost : (none) 09:01:44> SHOW /*!50000 GLOBAL*/ STATUS
    -> LIKE '%lock%';
+------------------------------+-------+
| Variable_name                | Value |
+------------------------------+-------+
   ... ...
| Innodb_row_lock_current_waits | 0     |
| Innodb_row_lock_time          | 0     |
| Innodb_row_lock_time_avg      | 0     |
| Innodb_row_lock_time_max      | 0     |
| Innodb_row_lock_waits         | 0     |
   ... ...
| Table_locks_immediate         | 44    |
| Table_locks_waited            | 0     |
+------------------------------+-------+
```

通过上述系统变量，我们可以得出表锁总次数，其中造成其他现线程等待的次数。同时还可以得到非常详细的行锁信息，如行锁总次数，行锁总时间，每次行锁等待时间，行锁造成最大等待时间以及当前等待行锁的线程数。通过对这些量的监控，我们可以清晰的了解到系统整体的锁定是否严重。如当 Table_locks_waited 与 Table_locks_immediate 的比值较大，则说明我们的表锁造成的阻塞比较严重，可能需要调整 Query 语句，或者更改存储引擎，亦或者需要调整业务逻辑。当然，具体改善方式必须根据实际场景来判断。而 Innodb_row_lock_waits 较大，则说明 Innodb 的行锁也比较严重，且影响了其他线程的正常处理。同样需要查找出原因并解决。造成 Innodb 行锁严重的原因可能是 Query 语句所利用的索引不够合理（Innodb 行锁是基于索引来锁定的），造成间隙锁过大。也可能是系统本身处理能力有限，则需要从其他方面（如硬件设备）来考虑解决。

- 复制延时量：复制延时量将直接影响了 Slave 数据库处于不一致状态的时间长短。如果我们是通过 Slave 来提供读服务，就不得不重视这个延时量。我们可以通过在 Slave 节点上执行“SHOW SLAVE STATUS”命令，取 Seconds_Behind_Master 项的值来了解 Slave 当前的延时量（单位：秒）。当然，该值的准确性依赖于复制是

否处于正常状态。每个环境下的 Slave 所允许的延时长短与具体环境有关，所以复制延时多长时间是合理的，只能由读者朋友根据各自实际的应用环境来判断。

- Tmp table 状况：Tmp Table 的状况主要是用于监控 MySQL 使用临时表的量是否过多，是否有临时表过大而不得不从内存中换出到磁盘文件上。临时表使用状态信息可以通过如下方式获得：

```
sky@localhost : (none) 09:27:28> SHOW /*!50000 GLOBAL*/ STATUS
-> LIKE 'Created_tmp%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Created_tmp_disk_tables | 1     |
   ... ...
| Created_tmp_tables      | 46    |
+-------------------------+-------+
```

从上面的状态信息可以了解到系统使用了 46 次临时表，其中有 1 次临时表比较大，无法在内存中完成，而不得不使用到磁盘文件。如果 Created_tmp_tables 非常大，则可能是系统中排序操作过多，或者是表连接方式不是很优化。而如果是 Created_tmp_disk_tables 与 Created_tmp_tables 的比率过高，如超过 10%，则我们需要考虑是否 tmp_table_size 这个系统参数所设置的足够大。当然，如果系统内存有限，也就没有太多好的解决办法了。

- Binlog Cache 使用状况：Binlog Cache 用于存放还未写入磁盘的 Binlog 信息。相关状态变量如下：

```
sky@localhost : (none) 09:40:38> SHOW /*!50000 GLOBAL*/ STATUS
    -> LIKE 'Binlog_cache%';
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| Binlog_cache_disk_use | 0     |
| Binlog_cache_use      | 0     |
+----------------------+-------+
```

如果 Binlog_cache_disk_use 值不为 0，则说明 Binlog Cache 大小可能不够，建议增加 binlog_cache_size 系统参数大小。

- Innodb_log_waits 量：Innodb_log_waits 状态变量直接反应出 Innodb Log Buffer 空间不足造成等待的次数。

```
sky@localhost : (none) 09:43:53> SHOW /*!50000 GLOBAL*/ STATUS
-> LIKE 'Innodb_log_waits';
```

```
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| Innodb_log_waits | 0     |
+-----------------+-------+
```

该变量值发生的频率将直接影响系统的写入性能，所以当该值达到每秒 1 次时就该增加 系统参数 innodb_log_buffer_size 的值，毕竟这是一个系统共用的缓存，适当增加并不会造成内存不足的问题。

上面这些监控量只是我个人认为比较重要的一些 MySQL 性能监控量，各位读者朋友还可以根据各自的需要，通过 MySQL 所提供的系统状态变量增加其他监控内容。

18.4 常用开源监控软件

前面已经介绍过了监控系统的设计思路，也分析了我们需要关注的部分健康状态和性能状态监控点，这一节再介绍几种常用的第三方监控软件，为大家提供一点搭建监控系统的思路。当然，推荐原则仍然是以开源（或免费）产品为主。当然，前提是我们暂时没有自行开发一套监控系统的打算，而希望通过一些开源的软件工具来实现。

在介绍监控软件之前，有一个软件是我不得不提的：RRDTool。RRDTool 全称为 Round Robin Database Tool，也就是环状循环数据库工具，在多种监控软件中充当非常重要的数据存储角色。所谓环状循环数据库，就是数据的存储方式类似于在一个环形空间中，没有确切的起始位置和结束位置。当写完一圈之后，新的数据会覆盖老数据。当然，由于环状循环数据库所存放的数据大多都是存放用于统计方面的数据，而且可以通过一些加和平均之类的统计算法通过老数据得出相应的统计结果。当老数据需要被覆盖的时候，他早已失去实际价值了。所以，RRDTool 在很多监控工具中都被用来存放各种性能数据，然后根据这些数据画出相应的趋势图，以及不同时间段内的平均值曲线。如果有哪位读者朋友有兴趣自行开发一个监控工具，RRDTool 同样可以作为您用来存放相关数据非常合适的选择。不过，这里有一点需要注意的是他只能存放数字。

1、 Nagios

Nagois 是一个非常著名的运行在 Linux/Unix 上的对 IT 设备或服务的运行状态进行监控的软件。实际上，我个人觉得称其为一个监控平台可能更为合适，因为它不仅仅自带了丰富的监控工具，同时还支持用户自己编写各种各样的监控脚本以插件形式嵌入其中。

Nagios 自带的监控功能不仅包含与主机资源相关的 CPU 负载，磁盘使用等，还包括了网络相关的服务，如 SMTP、POP3、HTTP、NNTP、PING 等。当然，Nagios 流行的另外一个重要原因是其简单的插件设计允许用户可以非擦方便地开发自己需要的服务检查。

对于大多数的监控场景来说，Nagios 的现有功能都能够满足。不论是主动检测，还是被动监控，都可以很好的实现。对于不同重要级别的监控点，可以分别设置不同的数据采集频率。对于异常的处理，可以设定为邮件、Web 以及其他自定义的异常通知机制（如手机短

信或者 IM 工具通知)。不仅如此，Nagios 还可以设置报警前的异常出现次数，如连续多少次访问超时之后再发出警告信息。而当我们需要进行正常维护的时候，还可以通过设置一个暂时忽略异常的 DownTime 时间段。

Nagios 分为客户端与服务器端两部分，客户端实际上就是大量的 Nagios 监控插件，负责采集监控点的各种数据，而服务器端则是配置管理、数据分析、告警发送、用户自助管理以及相关信息展示等功能。服务器端的 Web 展示界面的功能也非常强大，可以非常准且的展示出当前各个监控点的状态，上一次检测时间等信息。同时还能根据监控点追溯一定的历史记录信息。

当然，Nagios 也有一个缺点，那就是目前他仅仅支持健康状态数据的采集分析，而不支持通过采集性能数据来绘制性能趋势曲线图。当然，这可以结合其他的软件工具共同完成这一工作。

如需更为详细的 Nagios 的搭建使用手册,请各位读者朋友前往其官方网站获取更为权威的信息：http://www.nagios.org/docs

2、MRTG

MRTG 应该算是一款比较老牌的监控软件了，功能比较简单， 最初是为了监控网络链路流量而产生的。但是经过原开发者的不断改造，以及网友们的集体智慧，其应用场景已经远远超出了网络链路流量监控。

MRTG 的实现原理其实很简单，他通过 snmp 协议，调用监控设备上面相应的脚本，获取需要的返回值，然后通过 RRDTool 保存起来。然后再通过 RRDTool 将保存的数据进行相应的统计平均计算，最后画成图片，通过 html 的形式展现给前端浏览者。

对于 MRTG 来说，它不需要知道我们监控的到底是什么数据，只需要告诉他到底什么时候该调用什么脚本来取得监控返回值，同时配置好存放位置，即可完成一个监控项的监控配置。对于我们来说，最为重要的是写好取得监控值的脚本，设定好 MRTG 的采集频率，然后就是查看各种监控数据的曲线图了。

更为详细的 MRTG 使用及搭建方法，请至官方网站(http://oss.oetiker.ch/mrtg) 上查找相应文档。

3、 Cacti

Cacti 和 Nagios 最大的区别在于前者具有非常强大的数据采集、存储以及展现功能，但在告警管理这一块稍弱于后者。其开发语言是 PHP,配置存储在 MySQL 数据库中,数据采集同样利用了 SNMP 协议，采集数据的存储则利用了本节最前面介绍的 RRDTool。

在数据的绘图展现方面，相对于其他有些监控软件来说，也有一定的优势，如单个图上面可以有无限多个数据项共存，而不像 MRTG 每张图片上只能有两个数据项的曲线。

除了非常强大的数据灰土展现功能之外，Cacti 另外一个比较有吸引力的功能就是可以

通过用户管理，设定多种权限角色，让更多的用户自行定义维护各自的监控配置项。这个特性对于监控设备（或服务）涉及到多个部门的很多人的应用场景下，是非常有用的。

当然，除了上面的这几个比较有特点特性之外，Cacti 还存在很多很多其他的特性。大家可以从 Cacti 官方文档获得更详细的信息：http://www.cacti.net/documentation.php。

不同的监控需求，可以采用不同的监控软件来实现，如我个人的很多环境就同时使用了 Nagios 来实现健康状况的监控和告警。而性能数据的采集与绘图则利用了 MRTG 和 RRDTool 来实现。各位读者朋友完全可以根据自己的需求来决定如何搭建一个更为合适的监控系统，来帮助大家更进一步提高系统的可用性。

18.4 小结

系统的监控在很多环境中都没有得到足够的重视，可其实际意义却非常大。虽然监控系统本身并不能让系统运行的更好，却能够给在系统出现问题之后的第一时间通知我们，缩短了发现问题的时间，也间接提高了系统的可用性。甚至可以根据监控所收集的各种性能数据，让我们提前发现系统的性能问题，防范于未然。

本章通过对 MySQL 数据库环境的健康状态、性能状态以及相应的监控方式的分析，完成了搭建一个高可用数据库环境的最后一步，希望能够对各位读者朋友有一点帮助。