转: http://zhengdl126.javaeye.com/blog/419850

[**数据库水平切分的实现原理解析－－－分库，分表，主从，集群，负载均衡器**](http://zhengdl126.javaeye.com/blog/419850)

关键字: 水平切分，分库，分表，主从，集群

第1章  引言

随着互联网应用的广泛普及，海量数据的存储和访问成为了系统设计的瓶颈问题。对于一个大型的互联网应用，每天几十亿的PV无疑对数据库造成了相当高的负载。对于系统的稳定性和扩展性造成了极大的问题。通过数据切分来提高网站性能，横向扩展数据层已经成为架构研发人员首选的方式。水平切分数据库，可以降低单台机器的负载，同时最大限度的降低了了宕机造成的损失。通过负载均衡策略，有效的降低了单台机器的访问负载，降低了宕机的可能性；通过集群方案，解决了数据库宕机带来的单点数据库不能访问的问题；通过读写分离策略更是最大限度了提高了应用中读取（Read）数据的速度和并发量。目前国内的大型互联网应用中，大量的采用了这样的数据切分方案，Taobao,Alibaba,Tencent，它们大都实现了自己的分布式数据访问层（DDAL）。以实现方式和实现的层次来划分，大概分为两个层次（Java应用为例）：JDBC层的封装，ORM框架层的实现。就JDBC层的直接封装而言，现在国内发展较好的一个项目是被称作“变形虫”(Amoeba)的项目，由阿里集团的研究院开发，现在仍然处于测试阶段（beta版），其运行效率和生产时效性有待考究。就ORM框架层的实现而言，比如Taobao的基于ibatis和Spring的的分布式数据访问层，已有多年的应用，运行效率和生产实效性得到了开发人员和用户的肯定。本文就是以ORM框架层为基础而实现的分布式数据访问层。本课题的难点在于分库后，路由规则的制定和选择以及后期的扩展性，比如：如何做到用最少的数据迁移量，达到扩充数据库容量（增加机器节点）的目的。核心问题将围绕数据库分库分表的路由规则和负载均衡策略展开。

第2章 基本原理和概念

2.1基本原理：

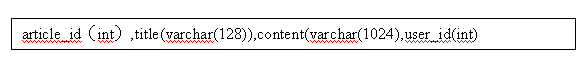
人类认知问题的过程总是这样的：what（什么）-?why(为什么)-?how(怎么

做)，接下来，本文将就这三个问题展开讨论和研究：

2.1.1什么是数据切分

"Shard" 这个词英文的意思是"碎片"，而作为数据库相关的技术用语，似乎最早见于大型多人在线角色扮演游戏中。"Sharding" 姑且称之为"分片"。Sharding 不是一门新技术，而是一个相对简朴的软件理念。众所周知，MySQL 5 之后才有了数据表分区功能，那么在此之前，很多 MySQL 的潜在用户都对 MySQL 的扩展性有所顾虑，而是否具备分区功能就成了衡量一个数据库可扩展性与否的一个关键指标(当然不是唯一指标)。数据库扩展性是一个永恒的话题，MySQL 的推广者经常会被问到：如在单一数据库上处理应用数据捉襟见肘而需要进行分区化之类的处理，是如何办到的呢? 答案是：Sharding。  Sharding 不是一个某个特定数据库软件附属的功能，而是在具体技术细节之上的抽象处理，是水平扩展(Scale Out，亦或横向扩展、向外扩展)的解决方案，其主要目的是为突破单节点数据库服务器的 I/O 能力限制，解决数据库扩展性问题。

**通过一系列的切分规则将数据水平分布到不同的DB或table中，在通过相应的DB路由 或者 table路由规则找到需要查询的具体的DB或者table，以进行Query操作。这里所说的“sharding”通常是指“水平切分”，** 这也是本文讨论的重点。具体将有什么样的切分方式呢和路由方式呢？行文至此，读者难免有所疑问，接下来举个简单的例子：我们针对一个Blog应用中的日志来说明，比如日志文章（article）表有如下字段：



面对这样的一个表，我们怎样切分呢？怎样将这样的数据分布到不同的数据库中的表中去呢？其实分析blog的应用，我们不难得出这样的结论：blog的应用中，用户分为两种：浏览者和blog的主人。浏览者浏览某个blog，实际上是在一个特定的用户的blog下进行浏览的，而blog的主人管理自己的blog，也同样是在特定的用户blog下进行操作的（在自己的空间下）。所谓的特定的用户，用数据库的字段表示就是“user\_id”。就是这个“user\_id”，它就是我们需要的分库的依据和规则的基础。**我们可以这样做，将user\_id为 1～10000的所有的文章信息放入DB1中的article表中，将user\_id为10001～20000的所有文章信息放入DB2中的 article表中，以此类推，一直到DBn。** 这样一来，文章数据就很自然的被分到了各个数据库中，达到了数据切分的目的。接下来要解决的问题就是怎样找到具体的数据库呢？其实问题也是简单明显的，既然分库的时候我们用到了区分字段user\_id，那么很自然，数据库路由的过程当然还是少不了 user\_id的。考虑一下我们刚才呈现的blog应用，不管是访问别人的blog还是管理自己的blog，总之我都要知道这个blog的用户是谁吧，也就是我们知道了这个blog的user\_id，就利用这个user\_id，利用分库时候的规则，反过来定位具体的数据库，比如user\_id是234，利用该才的规则，就应该定位到DB1，假如user\_id是12343，利用该才的规则，就应该定位到DB2。以此类推，**利用分库的规则，反向的路由到具体的DB，这个过程我们称之为“DB路由”。**

当然考虑到数据切分的DB设计必然是非常规，不正统的DB设计。那么什么样的DB设计是正统的DB设计呢？

我们平常规规矩矩用的基本都是。平常我们会自觉的按照范式来设计我们的数据库，负载高点可能考虑使用相关的Replication机制来提高读写的吞吐和性能，这可能已经可以满足很多需求，但这套机制自身的缺陷还是比较显而易见的（下文会提及）。上面提到的“自觉的按照范式设计”。考虑到数据切分的DB设计，将违背这个通常的规矩和约束，为了切分，我们不得不在数据库的表中出现冗余字段，用作区分字段或者叫做分库的标记字段，比如上面的article的例子中的user\_id这样的字段（当然，刚才的例子并没有很好的体现出user\_id的冗余性，因为user\_id这个字段即使就是不分库，也是要出现的，算是我们捡了便宜吧）。当然冗余字段的出现并不只是在分库的场景下才出现的，在很多大型应用中，冗余也是必须的，这个涉及到高效DB的设计，本文不再赘述。

2.1.2为什么要数据切分

上面对什么是数据切分做了个概要的描述和解释，读者可能会疑问，为什么需要数据切分呢？像 Oracle这样成熟稳定的数据库，足以支撑海量数据的存储与查询了？为什么还需要数据切片呢？的确，Oracle的DB确实很成熟很稳定，但是高昂的使用费用和高端的硬件支撑不是每一个公司能支付的起的。试想一下一年几千万的使用费用和动辄上千万元的小型机作为硬件支撑，这是一般公司能支付的起的吗？即使就是能支付的起，假如有更好的方案，有更廉价且水平扩展性能更好的方案，我们为什么不选择呢？

但是，事情总是不尽人意。平常我们会自觉的按照范式来设计我们的数据库，负载高点可能考虑使用相关的Replication机制来提高读写的吞吐和性能，这可能已经可以满足很多需求，但这套机制自身的缺陷还是比较显而易见的。首先它的有效很依赖于读操作的比例，Master往往会成为瓶颈所在，写操作需要顺序排队来执行，过载的话Master首先扛不住，Slaves的数据同步的延迟也可能比较大，而且会大大耗费CPU的计算能力，因为write操作在Master上执行以后还是需要在每台slave机器上都跑一次。这时候 Sharding可能会成为鸡肋了。 Replication搞不定，那么为什么Sharding可以工作呢？道理很简单，因为它可以很好的扩展。我们知道每台机器无论配置多么好它都有自身的物理上限，所以当我们应用已经能触及或远远超出单台机器的某个上限的时候，我们惟有寻找别的机器的帮助或者继续升级的我们的硬件，但常见的方案还是横向扩展, 通过添加更多的机器来共同承担压力。我们还得考虑当我们的业务逻辑不断增长，我们的机器能不能通过线性增长就能满足需求？Sharding可以轻松的将计算，存储，I/O并行分发到多台机器上，这样可以充分利用多台机器各种处理能力，同时可以避免单点失败，提供系统的可用性，进行很好的错误隔离。

综合以上因素，数据切分是很有必要的，且我们在此讨论的数据切分也是将MySql作为背景的。基于成本的考虑，很多公司也选择了Free且Open的MySql。对MySql有所了解的开发人员可能会知道，MySQL 5 之后才有了数据表分区功能，那么在此之前，很多 MySQL 的潜在用户都对 MySQL 的扩展性有所顾虑，而是否具备分区功能就成了衡量一个数据库可扩展性与否的一个关键指标(当然不是唯一指标)。数据库扩展性是一个永恒的话题，MySQL 的推广者经常会被问到：如在单一数据库上处理应用数据捉襟见肘而需要进行分区化之类的处理，是如何办到的呢? 答案也是Sharding，也就是我们所说的数据切分方案。

    我们用免费的MySQL和廉价的Server甚至是PC做集群，达到小型机+大型商业DB的效果，减少大量的资金投入，降低运营成本，何乐而不为呢？所以，我们选择Sharding，拥抱Sharding。

2.1.3怎么做到数据切分

说到数据切分，再次我们讲对数据切分的方法和形式进行比较详细的阐述和说明。

**数据切分可以是物理 上的，对数据通过一系列的切分规则将数据分布到不同的DB服务器上，通过路由规则路由访问特定的数据库，这样一来每次访问面对的就不是单台服务器了，而是N台服务器，这样就可以降低单台机器的负载压力。**

**数** **据切分也可以是数据库内的 ，对数据通过一系列的切分规则，将数据分布到一个数据库的不同表中，比如将article分为article\_001,article\_002等子表，若干个子表水平拼合有组成了逻辑上一个完整的article表，这样做的目的其实也是很简单的。** 举个例子说明，比如article表中现在有5000w条数据，此时我们需要在这个表中增加（insert）一条新的数据，insert完毕后，数据库会针对这张表重新建立索引，5000w行数据建立索引的系统开销还是不容忽视的。但是反过来，假如我们将这个表分成100 个table呢，从article\_001一直到article\_100，5000w行数据平均下来，每个子表里边就只有50万行数据，这时候我们向一张只有50w行数据的table中insert数据后建立索引的时间就会呈数量级的下降，极大了提高了DB的运行时效率，提高了DB的并发量。当然分表的好处还不知这些，还有诸如写操作的锁操作等，都会带来很多显然的好处。

综上，分库降低了单点机器的负载；**分表，提高了数据操作的效率，尤其是Write操作的效率。** 行文至此我们依然没有涉及到如何切分的问题。接下来，我们将对切分规则进行详尽的阐述和说明。

上文中提到，**要想做到数据的水平切分，在每一个表中都要有相冗余字符 作为切分依据和标记字段，通常的应用中我们选用user\_id作为区分字段，基于此就有如下三种分库的方式和规则：** （当然还可以有其他的方式）

按号段分：

**(1) user\_id为区分，1～1000的对应DB1，1001～2000的对应DB2，以此类推；**

**优点：可部分迁移**

**缺点：数据分布不均**

**(2)hash取模分：**

对user\_id进行hash（或者如果user\_id是数值型的话直接使用user\_id 的值也可），然后用一个特定的数字，比如应用中需要将一个数据库切分成4个数据库的话，我们就用4这个数字对user\_id的hash值进行取模运算，也就是user\_id%4,这样的话每次运算就有四种可能：结果为1的时候对应DB1；结果为2的时候对应DB2；结果为3的时候对应DB3；结果为0的时候对应DB4，这样一来就非常均匀的将数据分配到4个DB中。

**优点：数据分布均匀**

**缺点：数据迁移的时候麻烦，不能按照机器性能分摊数据**

**(3)在认证库中保存数据库配置**

就是建立一个DB，这个DB单独保存user\_id到DB的映射关系，每次访问数据库的时候都要先查询一次这个数据库，以得到具体的DB信息，然后才能进行我们需要的查询操作。

**优点：灵活性强，一对一关系**

**缺点：每次查询之前都要多一次查询，性能大打折扣**

以上就是通常的开发中我们选择的三种方式，**有些复杂的项目中可能会混合使用这三种方式。** 通过上面的描述，我们对分库的规则也有了简单的认识和了解。当然还会有更好更完善的分库方式，还需要我们不断的探索和发现。

第3章 本课题研究的基本轮廓

上面的文字，我们按照人类认知事物的规律，what?why?how这样的方式阐述了数据库切分的一些概念和意义以及对一些常规的切分规则做了概要的介绍。本课题所讨论的分布数据层并不仅仅如此，它是一个完整的数据层解决方案，它到底是什么样的呢？接下来的文字，我将详细阐述本研究课题的完整思想和实现方式。

分布式数据方案提供功能如下：

（1）提供分库规则和路由规则（RouteRule简称RR），将上面的说明中提到的三中切分规则直接内嵌入本系统，具体的嵌入方式在接下来的内容中进行详细的说明和论述；

（2）引入集群（Group）的概念，保证数据的高可用性；

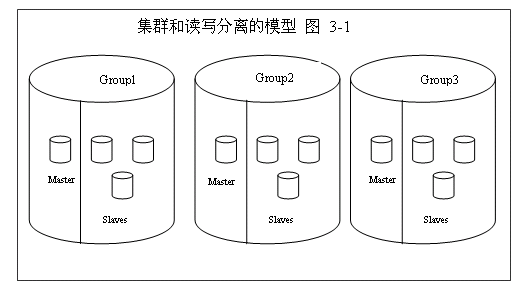
（3）引入负载均衡策略（LoadBalancePolicy简称LB）；

（4）引入集群节点可用性探测机制，对单点机器的可用性进行定时的侦测，以保证LB策略的正确实施，以确保系统的高度稳定性；

（5）引入读/写分离，提高数据的查询速度；

仅仅是分库分表的数据层设计也是不够完善的，当某个节点上的DB服务器出现了宕机的情况的时候，会是什么样的呢？是的，我们采用了数据库切分方案，也就是说有N太机器组成了一个完整的DB ，如果有一台机器宕机的话，也仅仅是一个DB的N分之一的数据不能访问而已，这是我们能接受的，起码比切分之前的情况好很多了，总不至于整个DB都不能访问。一般的应用中，这样的机器故障导致的数据无法访问是可以接受的，假设我们的系统是一个高并发的电子商务网站呢？单节点机器宕机带来的经济损失是非常严重的。也就是说，现在我们这样的方案还是存在问题的，容错性能是经不起考验的。当然了，问题总是有解决方案的。我们引入集群的概念，在此我称之为Group，也就是每一个分库的节点我们引入多台机器，每台机器保存的数据是一样的，一般情况下这多台机器分摊负载，当出现宕机情况，负载均衡器将分配负载给这台宕机的机器。这样一来，

就解决了容错性的问题。所以我们引入了集群的概念，并将其内嵌入我们的框架中，成为框架的一部分。



如上图所示，**整个数据层有Group1，Group2，Group3三个集群组成，这三个集群就是数据水平切分的结果，当然这三个集群也就组成了一个包含完整数据的DB。每一个Group包括1个Master（当然Master也可以是多个）和 N个Slave，这些Master和Slave的数据是一致的。比如Group1中的一个slave发生了宕机现象，那么还有两个slave是可以用的，这样的模型总是不会造成某部分数据不能访问的问题，除非整个 Group里的机器全部宕掉，但是考虑到这样的事情发生的概率非常小（除非是断电了，否则不易发生吧）。**

**在没有引入集群以前，我们的一次查询的过程大致如下：请求数据层，并传递必要的分库区分字段（通常情况下是user\_id）?数据层根据区分字段Route到具体的DB?在这个确定的DB内进行数据操作。** 这是没有引入集群的情况，当时引入集群会是什么样子的呢？看图一即可得知，我们的路由器上规则和策略其实只能路由到具体的Group，也就是只能路由到一个虚拟的Group，这个Group并不是某个特定的物理服务器。接下来需要做的工作就是找到具体的物理的DB服务器，以进行具体的数据操作。基于这个环节的需求，我们引入了负载均衡器的概念（LB）。负载均衡器的职责就是定位到一台具体的DB服务器。**具体的规则如下：负载均衡器会分析当前sql的读写特性，如果是写操作或者是要求实时性很强的操作的话，直接将查询负载分到Master，如果是读操作则通过负载均衡策略分配一个Slave。我们的负载均衡器的主要研究放向也就是负载分发策略，通常情况下负载均衡包括随机负载均衡和加权负载均衡 。** 随机负载均衡很好理解，就是从N个Slave中随机选取一个Slave。这样的随机负载均衡是不考虑机器性能的，它默认为每台机器的性能是一样的。假如真实的情况是这样的，这样做也是无可厚非的。假如实际情况并非如此呢？**每个Slave的机器物理性能和配置不一样的情况，再使用随机的不考虑性能的负载均衡，是非常不科学的，这样一来会给机器性能差的机器带来不必要的高负载，甚至带来宕机的危险，** 同时高性能的数据库服务器也不能充分发挥其物理性能。基于此考虑从，我们引入了加权负载均衡，也就是在我们的系统内部通过一定的接口，可以给每台DB服务器分配一个权值，然后再运行时LB根据权值在集群中的比重，分配一定比例的负载给该DB服务器。当然这样的概念的引入，无疑增大了系统的复杂性和可维护性。有得必有失，我们也没有办法逃过的。

**有了分库，有了集群，有了负载均衡器，是不是就万事大吉了呢？** 事情远没有我们想象的那么简单。虽然有了这些东西，**基本上能保证我们的数据层可以承受很大的压力** ，但是这样的设计并不能完全规避数据库宕机的危害。**假如Group1中的slave2 宕机了，那么系统的LB并不能得知，这样的话其实是很危险的，因为LB不知道，它还会以为slave2为可用状态，所以还是会给slave2分配负载。这样一来，问题就出来了，客户端很自然的就会发生数据操作失败的错误或者异常。这样是非常不友好的！怎样解决这样的问题呢？** 我们引入集群节点的可用性探测机制 ，或者是可用性的数据推送机制 。这两种机制有什么不同呢？首先说探测机制吧，顾名思义，探测即使，就是我的数据层客户端，不定时对集群中各个数据库进行可用性的尝试，实现原理就是尝试性链接，或者数据库端口的尝试性访问，都可以做到，当然也可以用JDBC尝试性链接，利用Java的Exception机制进行可用性的判断，具体的会在后面的文字中提到。那数据推送机制又是什么呢？其实这个就要放在现实的应用场景中来讨论这个问题了，一般情况下应用的DB 数据库宕机的话我相信DBA肯定是知道的，这个时候DBA手动的将数据库的当前状态通过程序的方式推送到客户端，也就是分布式数据层的应用端，这个时候在更新一个本地的DB状态的列表。并告知LB，这个数据库节点不能使用，请不要给它分配负载。一个是主动的监听机制，一个是被动的被告知的机制。两者各有所长。但是都可以达到同样的效果。这样一来刚才假设的问题就不会发生了，即使就是发生了，那么发生的概率也会降到最低。

上面的文字中提到的**Master和Slave** ，我们并没有做太多深入的讲解。如图一所示，一个Group由1个Master和N个Slave组成。为什么这么做呢？其中Master负责写操作的负载，也就是说一切写的操作都在Master上进行，而读的操作则分摊到Slave上进行。这样一来的可以大大提高读取的效率。在一般的互联网应用中，**经过一些数据调查得出结论，读/写的比例大概在 10：1左右** ，也就是说大量的数据操作是集中在读的操作，这也就是为什么我们会有多个Slave的原因。但是为什么要分离读和写呢？熟悉DB的研发人员都知道，写操作涉及到锁的问题，不管是行锁还是表锁还是块锁，都是比较降低系统执行效率的事情。我们这样的分离是把写操作集中在一个节点上，而读操作其其他的N个节点上进行，从另一个方面有效的提高了读的效率，保证了系统的高可用性。读写分离也会引入新的问题，比如我的Master上的数据怎样和集群中其他的Slave机器保持数据的同步和一致呢?这个是我们不需要过多的关注的问题，MySql的Proxy机制可以帮助我们做到这点，由于Proxy机制与本课题相关性不是太强，

在这里不做详细介绍。

综上所述，本课题中所研究的分布式数据层的大体功能就是如此。以上是对基本原理的一些讨论和阐述。接下来就系统设计层面，进行深入的剖析和研究。

第4章 系统设计

4.1系统实现层面的选择

在引言部分中提到，该系统的实现层面有两种选择，一种是基于JDBC层面上的选择，一种是基于现有数据持久层框架层面上的选择，比如Hibernate，ibatis。两种层面各有长处，也各有不足之处。基于JDBC层面上的系统实现，系统开发难度和后期的使用难度都将大大提高。大大增加了系统的开发费用和维护费用。本课题的定位是在成型的ibatis持久层框架的基础上进行上层的封装，而不是对ibatis源码的直接修改，这样一来使本系统不会对现有框架有太多的侵入性，从而也增加了使用的灵活性。之所以选择ibatis，原因如下：

（1）ibatis的学习成本非常低，熟练的Java Programmer可在非常的短时间内熟练使用ibatis；

（2）ibatis是轻量级的ORM，只是简单的完成了RO，OR的映射，其查询语句也是通过配置文件sql-map.xml文件在原生sql的层面进行简单的配置，也就是说我们没有引入诸如Hibernate那样的HQL的概念，从而增强了 sql的可控性，优秀的DBA可以很好的从sql的层面对sql进行优化，使数据层的应用有很强的可控性。Hibernate虽然很强大，但是由于 Hibernate是OR的一个重型封装，且引入HQL的概念，不便于DBA团队对sql语句的控制和性能的调优。

基于以上两点理由，本课题在ORM的产品的选择上选择了易学易用且轻量级的持久层框架ibatis。下面的讨论也都是特定于ibatis的基础上的讨论。

4.2其他开源框架的选择

在一些大型的Java应用中，我们通常会采用Spring这样的开源框架，尤其是 IoC（DI）这部分，有效的帮助开发人员管理对象的依赖关系和层次，降低系统各层次之间的实体耦合。Spring的优点和用处我相信这是开发人员众所周知的，在此不再赘述。本课题的数据层也将采用Spring做为IoC（DI）的框架。

4.3系统开发技术和工具介绍

开发语言：Java JDK1.5

集成开发环境：Eclipse 3.3.4

Web环境下测试服务器：JBoss 4.2

构建工具：淘宝自行研发的构建工具Antx（类似于Maven），当然也可以用Maven

依赖的开源Jar：Spring2.0，ibaits，commons-configuration(读取配置文件)，log4j，junit等

第5章 系统分析（待续。。）