摘 要

随着因特网应用和计算机技术的飞速发展，数据库逐渐成为信息系统的核心部分并广泛应用于企业、金融机构、政府及国防等各个领域。但是传统的关系数据库在面对日益增多的大量数据时暴露了很多问题，不能对其高效、实时的处理。对系统提出的新需求，传统的关系数据库并不能很好的解决。非关系型数据库是为弥补关系数据库的不足而产生的，OrientDB数据库就是非关系型数据库的一种。目前现有的非关系数据库对大数据量数据的处理能力很强但是在在很多关键领域，关系型数据库又是非关系型数据库无法代替的。如何结合关系型数据库和非关系型数据库的优点成为当前数据库领域的关键问题。

通过对分布式理论、关系数据库理论及非关系型数据库技术的学习和研究，本文基于Java语言实现了一个分布式的关系型数据库JSQL，JSQL是一个将关系数据库技术，非关系型数据库技术和分布式技术相结合的产物。JSQL采用常用的客服端-服务器架构，分为客服端和服务器端，服务器端又分为分布式管理节点和分布式数据库节点。分布式管理节点实现了数据库服务器的负载均衡和监控管理功能。分布式数据库节点主要包含五大模块，网络模块接收前端客服端的连接请求，对客服端进行认证和授权，并对连接进行管理，并实现基于Mysql的通信协议； Sql的解析和执行模块接收客户端的SQL请求，然后解析和执行数据库存储的调用，返回执行结果；审计模块是存储和分析所有对数据库的更改情况，向分布式管理节点提供审计数据；数据库引擎模块利用了非关系型Orientdb数据库引擎，实现可靠的数据存储；分布式模块利用hazlcast实现了数据库集群，本文提出了分布式多版本并发控制方法，用来解决分布式数据一致性问题。基于以上功能模块，ＪＳＱＬ具有高可用性，可扩展性，负载均衡等特性，同时从数据库底层考虑了数据库安全审计需求，加入了数据审计图形化界面显示审计结果。

论文对系统进行了功能和性能测试。功能测试结果表明，系统在功能上符合分布式数据库的基本要求，审计系统的功能也达到本论文的要求。论文通过对性能测试结果进行分析，认为系统的性能基本达到本论文的要求。但是本系统对复杂SQL语句的支持还不是很完善，最后提出了改进的方案。

关键词：分布式数据库，关系数据库，安全审计，Mysql，非关系型数据库

ABSTRACT

With the rapid development of Internet application and computer technology, the database has gradually become the core of information system Points and widely used in enterprises, financial institutions, government and national defense and other fields. However, the traditional relational database is facing increasing When exposed to a large number of data a lot of problems, can not be efficient and real-time processing. The new needs of the system Seeking, the traditional relational database can not be solved very well. Non-relational database is to make up for the lack of relational database, the OrientDB database Is a non-relational database. At present, the existing non-relational database has great ability to handle large amounts of data But in many key areas, relational databases are non-relational databases that can not be replaced. How to combine the advantages of relational databases and non-relational databases has become the key issue in the current database area.

Through the study of distributed theory, relational database theory and non-relational database technology, This article based on the Java language to achieve a distributed relational database JSQL, JSQL is a relational database technology, The combination of non-relational database technology and distributed technology. JSQL common client-server - server architecture, divided into customer service and server-side, Server-side is divided into distributed management nodes and distributed database nodes. Distributed management nodes to achieve a database server load balancing and monitoring and management functions. Distributed database node contains five major modules, The network module accepts the connection request of the front-end application, authenticates and authorizes the client-end, and manages the connection, And realize the communication protocol based on Mysql, so easy to migrate to Mysql users to the system; Sql parsing and execution module to accept the client's SQL request, and then parse and execute the database stored call, Return the execution result The audit module is to store and analyze all changes to the database, the distributed management node to provide audit data; The database engine module makes use of the non-relational OrientDB database engine for reliable data storage; Distributed modules use hazlcast to implement a database cluster. Based on the above functional modules, JSQL has high availability, scalability, load balancing and other characteristics, while taking into account the database security audit

needs from the bottom of the database, Joined the data audit graphical interface shows the audit results.

The thesis has carried on the function and the performance test to the system. The result of function test shows that the system conforms to the basic requirements of distributed database functionally, and the function of auditing system also meets the requirements of this dissertation. Papers through the performance test results analysis, that the performance of the system basically meet the requirements of this paper. However, the system of complex SQL statement support is not perfect, and finally proposed an improved program.

Keywords: Distributed database, relational database, security audit, Mysql, Non-relational database

目 录

第一章 绪论

1.1 研究背景和研究意义

互联网诞生以来在全球迅速蔓延。 2017年我国工业和信息化部最新发布的通信业经济运行情况显示， 2月末，我国移动电话用户总数达到13.3亿户，移动互联网用户总数达到11.2亿户，使用手机上网的用户数接近10.6亿户。互联网用户数量还有很大的增长空间，特别是亚洲人口众多的发展中国家。同时，智能手机的革命发展通过移动互联网大大提升用户体验，使移动互联网迅速发展。伴随着互联网的发展出现了各种基于互联网的应用服务，从传统媒体门户网站到BBS以及近年来社会媒体的兴起，电子商务的巨大发展，还有各种各样的移动互联网应用程序。

随着互联网和互联网应用的发展，数据存储的需求不断增长。 IDC报告显示，预计到2020年全球数据总量将超过40ZB，这一数据量是2011年的22倍。在过去几年，全球的数据量以每年百分之58的速度增长，在未来这个速度会更快。如果按照现在存储容量每年百分之40的增长速度计算，到2018年需要存储的数据量甚至会大于存储设备的总容量。

未来是“大数据”时代，这么大的存储需求，给数据存储技术的发展带来了很大的压力。有很多应用基于互联网提供的各种服务正在进入井喷时代的发展，而这些应用，背景的很大一部分面临同样的问题：怎么样以尽可能廉价的方式实现大规模数据存储和查询。如搜索服务，需要存储页面而分析，微博等社交网络需求的实时用户来表达查询的观点，在线旅游需要玩家的信息和操作来存储和查询，银行需要用户帐户信息和用户用于实时存储和查询。这种应用所面临的挑战总结为大数据可用性和可靠性要求高，部分应用需要实时查询响应和高并发性和数据一致性要求。在这样的环境下，分布式数据库近年来也取得了飞速发展。分布式数据库是一种数据库技术和网络技术结合的产品，相对于单节点数据库，分布式数据库容量和可用性具有很大的优势，因此能够更好的应对大规模和超大规模的数据存储需求的应用。分布式数据库系统通常使用大量廉价，独立的计算机系统构建，经济和性能上，可以满足互联网应用程序的对数据大小，可用性和性能的需求。

1.2 国内外研究历史与现状

数据库技术的发展始于20世纪60年代。在没有数据库的情况下，使用计算机存储数据的用户（主要是财务科研单位）以操作系统中的文件的形式存储数据。随着业务的发展，应用程序变得越来越复杂，人们开始需要开发管理数据在通用软件，这促成了数据库的诞生。 20世纪60年代以来，人们探索数据模型实现数据库，后来开发出三大数据库模型：层次数据模型，网格数据模型，关系数据模型，后来也出现在对象数据模型中。其中，可以使用关系数据模型严格的数学理论来描述数据库的组织和操作，具有简单灵活，数据库独立性高的特点特征。从20世纪70年代到90年代，关系数据库理论成熟并得到广泛应用，这个里程碑是1974年，IBM的圣荷西，加利福尼亚研究实验室的D.D. Chamberlin和Ray Boyce开发了SEQUEL结结构化查询语言，后来在1980年更名为SQL。 SQL是数据库中的标准数据查询语言，在1986年，由美国国家标准学会规范SQL，就这样成为了数据库系统的标准语言。 SQL包含三个部分：数据定义语言，数据操作语言，数据控制语言。 SQL是一种全面的通用关系数据语言，可以当它是一种高级的非程序语言，它允许用户在高级数据结构中工作。使用SQL 用户无需知道数据的具体存储方式。在SQL中一个简单的语句实现效果，使用其他编程语言需要很大一部分程序才能实现。另外，SQL通过使用这种语言允许用户掌握这种语言就可以使用这种语言来操作任何一个标准数据库产品。到20世纪90年代，关系数据库标准几乎适用于任何数据存储需求的应用程序。

分布式数据库系统是数据库系统技术与网络技术的结合。分布式数据库系统（DDBS）的研究始于20世纪70年代。但是，分布式数据库的理论与应用成为一个热门话题，是20世纪90年代的事情。 90年代，互联网网络出现爆炸式增长，同时各种应用程序对存储的需求与日俱增。在这样的环境下，分布式数据库系统的研究成为了热门话题，人们探索分布式数据库系统理论和关键技术，并快速应用理论实践，开发了各种商业应用价值的数据库产品。 2002年，Eric Brewer提出了引导分布式数据库研究的重要理论，并且后来证明是正确的，这个理论就是CAP定理，CAP定理的核心观点是：在分布式计算系统中，不可能同时满足以下三点：一致性，可用性，分区容忍性。 CAP理论认为分布式数据库产品的设计必须介于三者之间，其中至少有两个可以满足，不可能同时满足三个。根据CAP理论的指导，有学者认为，基于传统的关系数据库构建分布式数据库，很难满足当前大型数据存储需求的各类互联网应用，如搜索引擎，社交网络等。由于传统的关系数据库非常重视数据一致性，在传统的关系数据库中，交易的四点必须保证要求：ACID（Atomicity，一致性，隔离，持久性）。有部分人认为，根据CAP理论，在

满足强大的一致性之后，也希望获得互联网应用的高可用性性是非常困难的。因此，在二十世纪九十年代末和二十一世纪初，互联网应用大幅增长出现了一些人认为是革命性的分布式数据库概念：NoSQL（not only SQL）。 NoSQL和传统的关系数据库非常的不同，对于一个概念，NoSQL的显着特点是：非关系型，分布式，不提供ACID数据库设计模式。 NoSQL不支持复杂的关系操作，不提供对交易一致性的支持。由于摆脱了必须满足支持一致性的功能要求，根据NoSQL概念设计产品生成的高可扩展性高并发高可用性性支持得到了很大的改善，各种类似的开源或商业的NoSQL产品出现了，并且被大量和被各种互联网应用程序所使用。这些NoSQL产品被应用到新的产品，如博客，论坛，微博等其他社交服务。这些应用程序具有高度可扩展性，高可用性，并发性。高可用性这些功能很重要，因为这些应用压力主要来自大规模数据存储，大量并发访问及时响应， NoSQL拒绝一致性支持是合理的，因为像社交网络应用程序一样用户丢失数据库的后果不会太严重，毕竟，这不会导致用户遭受巨大的损失，如经济利益。作为分布式数据库，NoSQL在过去两年爆发式增长，均受益于互联网高速发展，也是各种应用到“大数据”的发展趋势，CAP理论再一次证明了它的正确性。

毕竟，NoSQL放弃了一致性的支持，这些产品可以应用到请求的一致性不高在应用上，随着互联网的快速发展和“大数据”时代的到来，那些需要关系操作，需要高级一致性并且面临大规模数据存储查询要求的应用，比如电子商务，这样的应用程序不能使用NoSQL产品。所以，目前的分布式数据库开发方向，根据CAP理论，一个是尽可能的减少一致性，提高可用性，另一个方向是需要尝试确保操作之间的关系，一致性的支持，同时利用分布式技术的优势，提供尽量高性能的服务。

1.3 论文的主要工作

通过对分布式理论、关系数据库理论及非关系型数据库技术的学习和研究，论文基于Java语言实现了一个分布式的关系型数据库JSQL ，JSQL结合了非关系型数据库OrientDB和关系型数据库的优点，同时实现了Mysql的通信协议，是一个兼容Mysqk通信协议的分布式数据库。论文的主要工作包括：

1.利用OrientDB非关系型存储引擎设计和实现了关系型数据库的本地存储。

2.实现了Mysql通信协议，使得可以通过Mysql客服端来连接JSQL分布式数据库，方便Mysql用户迁移到本数据库系统。

3.实现了对SQL语句的解析和执行。

4.实现了数据库的分布式架构，使得数据可以存储在多台计算机上面。

5.实现了系统的审计系统，使得系统的安全性得到增强。

6.实现了分布式数据库节点的负载均衡功能。

1.4 本论文的结构安排

第一章作为本论文的绪论，首先介绍了论文的选题背景和本论文进行研究的意义。接着阐述了数据库的发展历史和现状。最后一节里介绍了论文的主要工作。

第二章，主要介绍了本论文相关的理论基础和相关的技术，首先给数据库分类，给出关系数据库和非关系型数据库相关的概念，然后对分布式相关的技术进行了介绍，主要包括负载均衡技术，数据分片技术以及数据库高可用技术。

第三章，作为论文的需求分析，本章给出了分布式数据库JSQL的实现目标，对系统进行需求分析。从系统用户的角度出发，对分布式内存数据库进行详细、切合实际的需求分析，并在需求分析的基础上，对分布式数据库的整体架构等进行模块划分，按照功能分解结构细化、模块化系统功能，并对各个模块进行再划分。同时也对系统关键模块需要的技术进行了分析。

第四章，是系统的设计，包括总体设计和模块划分，本章对系统的总体架构和各个模块的详细架构给出了阐述。系统主要包括分布式管理节点和分布式数据库几点。本章分别对各个几点的各个模块进行了详细的分析。

第五章，是关于系统的具体实现，本章分别从数据库模块，集群架构模块，数据库审计模块详细说明了每个模块的实现。接着就每个关键模块的实现进行了讲解。最后对系统主要功能的流程进行讲解。

第六章，论文对jsq分布式数据库进行测试，包括功能测试和性能测试。第七章，论文总结了论文的工作，对本系统的优点和缺点进行了说明。并对未来的工作进行了展望。

最后部分，是关于致谢和参考资料。

1.5 本章小结

本章首先阐述了论文的背景，并表明本论文的研究对于该领域的发展具有非常重要的意义。然后介绍本课题的国内外研究历史和发展现状。最后介绍了本文的主要工作和章节部分安排。

第二章 相关理论和技术

本节介绍数据库有关的理论知识，以及分布式系统系统有关的算法和协议。

2.1 数据库系统

2.1.1 数据库系统概念

数据库管理系统是一种操纵和管理数据库的大型软件，用于建立、使用和维护数据库。它对数据库进行统一的管理和控制，以保证数据库的安全性和完整性。用户通过数据库系统访问数据库中的数据，数据库管理员也通过数据库系统进行数据库的维护工作。它可使多个应用程序和用户用不同的方法在同时或不同时刻去建立，修改和询问数据库。大部分数据库系统提供对数据的追加、删除等操作。数据库管理系统是数据库系统的核心，是管理数据库的软件。数据库管理系统就是实现把用户意义下抽象的逻辑数据处理，转换成为计算机中具体的物理数据处理的软件。有了数据库管理系统，用户就可以在抽象意义下处理数据，而不必顾及这些数据在计算机中的布局和物理位置。

2.1.2 关系数据库系统

2.1.2.1 关系模型

用于数据库管理的关系模型是基于谓词逻辑和集合论的一种数据模型，广泛被使用于关系型数据库之中。最早于1969年由埃德加·科德提出。关系模型的基本假定是所有数据都表示为数学上的关系，关系模型是采用二维表格结构表达实体类型及实体间联系的数据模型. 关系模型允许设计者通过数据库规范化的提炼，去建立一个信息的一致性的模型。

基本的关系建造块是域或者叫数据类型。元组是属性的有序多重集，属性是域和值的有序对。关系变量是域和名字的有序对（序偶）的集合，它充当关系的表头。关系是元组的集合。尽管这些关系概念是数学上的定义的，它们可以宽松的映射到传统数据库概念上。表是关系的公认的可视表示；元组类似于行的概念。关系模型的基本原理是信息原理：所有信息都表示为关系中的数据值。所以，关系变量在设计时刻是相互无关联的；反而，设计者在多个关系变量中使用相同的域，如果一个属性依赖于另一个属性，则通过参照完整性来强制这种依赖性。

关系数据库，是建立在关系数据库模型基础上的数据库，借助于集合代数等概念和方法来处理数据库中的数据，同时也是一个被组织成一组拥有正式描述性

的表格，该形式的表格作用的实质是装载着数据项的特殊收集体，这些表格中的数据能以许多不同的方式被存取或重新召集而不需要重新组织数据库表格。关系数据库的定义造成元数据的一张表格或造成表格、列、范围和约束的正式描述。每个表格包含用列表示的一个或更多的数据种类。每行包含一个唯一的数据实体，这些数据是被列定义的种类。当创造一个关系数据库的时候，你能定义数据列的可能值的范围和可能应用于那个数据值的进一步约束。

2.1.3 事务与并发控制

2.1.3.1 事务

事务象征内执行工作单元的数据库管理系统中独立于其他事务的连贯和可靠的方式针。一个事务通常代表数据库中的任何变化。数据库环境中的事务有两个主要目的：为了提供可靠的工作单元，允许从故障中正确恢复，并保持数据库一致，即使在系统故障的情况下，执行停止，数据库上的许多操作仍未完成，状态不清楚。在同时访问数据库的程序之间提供隔离。如果没有提供这种隔离，程序的结果可能是错误的。根据定义，数据库事务必须是原子的，一致的，隔离的和持久的。数据库从业者通常使用缩写ACID来引用数据库事务的这些属性。交易提供了“全或无”的命题，指出在数据库中执行的每个工作单元必须完整或完全没有任何影响。此外，系统必须将每个事务与其他事务隔离开来，结果必须符合数据库中的现有约束，并且成功完成的事务必须写入持久存储。

将数据完整性视为最重要的数据库和其他数据存储通常包括处理事务以维护数据完整性的能力。单个事务由一个或多个独立的工作单元组成，每个读取和或写入数据库或其他数据存储的信息。当发生这种情况时，确保所有此类处理使数据库或数据存储处于一致状态时，通常很重要。

2.1.3.2 并发控制

在信息技术和计算机科学领域，特别是在计算机编程，操作系统，多处理器和数据库领域，并发控制可以确保并发操作的正确结果，同时尽快获得这些结果。计算机系统，软件和硬件都由模块或组件组成。每个组件被设计为正确地操作，即遵守或符合某些一致性规则。当通过消息传递或通过共享访问的数据同时进行操作的组件时，某个组件的一致性可能被另一个组件违反。并发控制的一般领域提供了规则，方法，设计方法和理论以保持组件在交互时并发运行的一致性，从而保持整个系统的一致性和正确性。将并发控制引入系统意味着应用操作约束，这通常会导致某些性能下降。操作一致性和正确性应尽可能高效地实现，而不会

将性能降低到合理的水平以下。与更简单的顺序算法相比，并发控制可能需要大量额外的复杂性和并发算法的开销。例如，并发控制失败可能导致数据损坏，从而导致读取或写入操作受损。

在并发控制数据库管理系统，其它事务对象和相关的分布式应用程序（例如，网格计算和云计算）确保数据库事务被执行同时在不违反各数据库的数据完整性。因此，并发控制是任何系统中的正确性的基本要素，其中两个数据库事务或更多的数据库事务或多个时间重叠执行，可以访问相同的数据，例如实际上在任何通用数据库系统中。因此，自20世纪70年代初出现数据库系统以来，已经积累了大量相关研究。上述参考文献中提出了一个完善的数据库系统并发控制理论：序列化理论，可以有效地设计和分析并发控制方法和机制。中提出了一种用于抽象数据类型的原子事务并发控制的替代理论。

为了确保正确性，一个DBMS通常可以保证只有序列化的交易时间表中产生，除非串行化是故意轻松提高性能，但只有在应用程序的正确性不受损害的情况。为了在失败（中止）事务（由于许多原因总是可能发生）的情况下维护正确性，计划也需要具有可恢复性属性。DB MS还保证不会导致提交的事务的影响，并且不会中止任何影响（回滚）交易保留在相关数据库中。整体交易表征通常由以下ACID规则汇总。随着数据库已经成为分布，或在分布式环境中进行合作需要，并发控制机制有效配置受到格外的关注。下面是并发控制的主要方法：

1.锁定

通过分配给数据的锁控制对数据的访问。对另一个事务锁定的数据项的事务的访问可能被阻止，直到锁定释放。

2.串行化

检查计划图中的周期并通过中止来破坏它们。

3.时间戳排序

为事务分配时间戳，并按时间戳顺序控制或检查对数据的访问。

4.承诺排序，控制或检查交易的提交事件的时间顺序与其各自的优先顺序兼容。

5.多分支并发控制

通过在每次写入对象时生成新版本的数据库对象，并根据调度方式允许事务对每个对象的最后相关版本的读取操作来增加并发性和性能。

6.索引并发控制

将访问操作同步到索引，而不是用户数据。专业方法提供了显着的性能提升。

每个事务都为其访问的数据维护一个私有工作空间，只有在事务提交之后，其更改的数据才会在事务外部变得可见。这种模式在许多情况下提供了不同的并发控制行为，并带来好处。

2.1.4 NoSQL数据库系统

NoSQL，泛指非关系型的数据库。随着互联网web2.0网站的兴起，传统的关系数据库在应付web2.0网站，特别是超大规模和高并发的SNS类型的web2.0纯动态网站已经显得力不从心，暴露了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。NoSQL数据库的产生就是为了解决大规模数据集合多重数据种类带来的挑战，尤其是大数据应用难题。虽然NoSQL的流行与火起来才短短一年的时间，但是不可否认，现在已经开始了第二代运动。尽管早期的堆栈代码只能算是一种实验，然而现在的系统已经更加的成熟、稳定。不过现在也面临着一个严酷的事实：技术越来越成熟——以至于原来很好的NoSQL数据存储不得不进行重写，也有少数人认为这就是所谓的2.0版本。该工具可以为大数据建立快速、可扩展的存储库。

对于NoSQL并没有一个明确的范围和定义，但是他们都普遍存在下面一些共同特征：

1.不需要预定义模式：不需要事先定义数据模式，预定义表结构。数据中的每条记录都可能有不同的属性和格式。当插入数据时，并不需要预先定义它们的模式。

2.无共享架构：相对于将所有数据存储的存储区域网络中的全共享架构。NoSQL往往将数据划分后存储在各个本地服务器上。因为从本地磁盘读取数据的性能往往好于通过网络传输读取数据的性能，从而提高了系统的性能。

3.弹性可扩展：可以在系统运行的时候，动态增加或者删除结点。不需要停机维护，数据可以自动迁移。

4.分区：相对于将数据存放于同一个节点，NoSQL数据库需要将数据进行分区，将记录分散在多个节点上面。并且通常分区的同时还要做复制。这样既提高了并行性能，又能保证没有单点失效的问题。

5.异步复制：和RAID存储系统不同的是，NoSQL中的复制，往往是基于日志的异步复制。这样，数据就可以尽快地写入一个节点，而不会被网络传输引起迟延。缺点是并不总是能保证一致性，这样的方式在出现故障的时候，可能会丢失少量的数据。

6.BASE：相对于事务严格的ACID特性，NoSQL数据库保证的是BASE特性。BASE是最终一致性和软事务。

2.1.4.1 OrientDB介绍

OrientDB是一种NoSQL数据库，虽然它不是关系数据库系统，但是它底层的存储引擎支持事务。所以本系统在后面就直接用的这个这个存储引擎减少系统的开发时间，同时也增加系统的稳定性。

2.2 分布式数据库

2.2.1 分布式数据库概述

分布式数据库是一个数据库，其中存储设备没有全部连接到一个共同的处理器。它可以存储在位于相同物理位置的多台计算机中; 或者可以分散在互连计算机的网络上。与其中处理器紧密耦合并构成单个数据库系统的并行系统不同，分布式数据库系统由松散耦合的站点组成，共享没有物理组件。

2.2.2 分布式数据库的特点

1.数据独立性

分布式数据库系统中，每个系统节点都具有数据独立性。数据独立性包括了数据的逻辑独立性和物理独立性，逻辑独立性是指用户的应用程序与数据库的逻辑结构是相互独立的，即当数据的逻辑结构改变时，用户程序也可以不变；物理独立性是指用户的应用程序与存储在磁盘上的数据库中数据是相互独立的。即，数据在磁盘上怎样存储由 DBMS 管理，用户程序不需要了解，这样当数据的物理存储改变了，应用程序不用改变。

2.分布透明性

分布式数据库还具有分布透明性，该特性使用户不必关心数据的逻辑分片，不必关心数据物理位置分布的细节及局部场地上数据库支持哪种数据模型。分布透明性的优点显而易见；有了分布透明性，用户的应用程序书写起来就如同数据没有分布一样，当数据从一个场地移到另一个场地时，不必改写应用程序，当增加某些数据的重复副本时，也不必改写应用程序。

3.节点自治性

构成分布式数据库的每一个物理数据库都是独立的，可以由数据库管理员分别进行管理，如同一个非网络本地数据库。

4.复制透明性

用户不用关心数据库在网络中各个节点的复制情况。被复制的数据，一般更新都由系统自动完成，在分布式数据库系统中，可以把一个场地的数据复制到其他场地存放，应用程序可以使用复制到本地的数据在本地完成分

布式操作，避免通过网络传输数据，从而提高了系统的运行和查询效率。但是对于复制数据的更新操作，就要涉及到对所有复制数据的更新。

5.易于扩展性

在大多数网络环境中，单个数据库服务器最终会不满足使用。如果服务器软件支持透明的水平扩展，那么就可以增加多个服务器来进一步分布数据和分担处理任务。

2.2.3 数据分布和负载均衡

分布式系统如何拆解输入数据，将数据分发到不同的机器中。下面将介绍几种不同的数据分布方式。

2.2.3.1 哈希分布

哈希方式是最常见的数据分布方式,其方法是按照数据的某一特征计算哈希值, 并将哈希值与机器中的机器建立映射关系,从而将不同哈希值的数据分布到不同的机器上。只要哈希散列性比较好，数据就能均匀到分发到不同机器中。同时,需要管理的元信息很少,只需要知道哈希函数和模（一般是机器总数）。但是有个明显的缺点,扩展性很差。如果我想把集群规模扩大,可能所有的数据需要被重新迁移。针对哈希方式扩展性差的问题,一种思路是不再简单的将哈希值与机器做除法取模映射,而是将对应关系作为元数据由专门的元数据服务器管理。访问数据时,首先计算哈希值并查询元数据服务器,获得该哈希值对应的机器。同时,哈希值取模个数往往大于机器个数,这样同一台机器上需要负责多个哈希取模的余数。不过，需要管理的元数据就多了。

2.2.3.2 顺序分布

按数据范围分布是另一个常见的数据分布式, 将数据按特征值的值域范围划分为不同的区间,使得集群中每台(组)服务器处理不同区间的数据。对于上面哈希方式某个用户数据特别多我们就可以通过采用数据范围分布解决,动态划分范围空间,实现负载均衡。按数据范围分布数据需要记录所有的数据分布情况。一般的,往往需要使用专门的服务器在内存中维护数据分布信息, 称这种数据的分布信息为一种元数据。实际工程中,一般也不按照某一维度划分数据范围,而是使用全部数据划分范围, 从而避免数据倾斜的问题。使用范围分布数据的方式的最大优点就是可以灵活的根据数据量的具体情况拆分原有数据区间, 拆分后的数据区间可以迁移到其他机器,一旦需要集群完成负载均衡时, 与哈希方式相比非常灵活。另外,当集群需要扩容时,可以随意添加机器,而不限为倍增的方式, 只需将原机器上

的部分数据分区迁移到新加入的机器上就可以完成集群扩容。而缺点就是元数据可能会成为瓶颈。

2.2.3.3 一致性哈希

一致性哈希的基本方式是使用一个哈希函数计算数据或数据特征的哈希值,令该哈希函数的输出值域为一个封闭的环,即哈希函数输出的最大值是最小值的前序。将节点随机分布到这个环上,每节点负责处理从自己开始顺时针至下一个节点的全部哈希值域上的数据。

一致性哈希的优点在于可以任意动态添加、删除节点,每次添加、删除一个节点仅影响一致性哈希环上相邻的节点。但也有很明显的缺点,随机分布节点的方式使得很难均匀的分布哈希值域, 尤其在动态增加节点后,即使原先的分布均匀也很难保证继续均匀,由此带来的另一个较为严重的缺点是 ,当一个节点异常时,该节点的压力全部转移到相邻的一个节点,当加入一个新节点时只能为一个相邻节点分摊力。为此，一种改进是引入“虚节点”的概念。系统初始时就创建许多虚节点, 虚节点的个数一般远大于未来集群中机器的个数, 将虚节点均匀分布到一致性哈希值域环上,其功能与基本一致性哈希算法中的节点相同。为每个节点分配若干虚节点。操作数据时,首先通过数据的哈希值在环上找到对应的虚节点,进而查找元数据找到对应的真实节点。这样,一旦某个节点不可用,该节点将使得多个虚节点不可用,从而使得多个相邻的真实节点负载失效节点的压里。同理,一旦加入一个新节点,可以分配多个虚节点,从而使得新节点可以负载多个原有节点的压力,从全局看,较容易实现扩容时的负载均衡。

2.2.3.4 负载均衡

在计算中，负载平衡可以改善多个计算资源（如计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器）之间的工作负载分配。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间，并避免任何单一资源的过载。使用负载均衡而不是单个组件的多个组件可以通过冗余来增加可靠性和可用性。负载平衡通常涉及专用软件或硬件，如多层交换机或域名系统服务器进程。有很多种负载均衡的算法可用，下面是常用的一些算法：

1.轮询法

轮询法是负载均衡中最常用的算法，它容易理解也容易实现。轮询法是指负载均衡服务器将客户端请求按顺序轮流分配到后端服务器上，以达到负载均衡的目的。

2.加权轮询法

简单的轮询法并不考虑后端机器的性能和负载差异。给性能高、负载低的机器配置较高的权重，让其处理较多的请求；而性能低、负载高的机器，配置较低的权重，让其处理较少的请求。加权轮询法可以很好地处理这一问题，它将请求顺序且按照权重分派到后端服务器。

3.最小连接数法

即使后端机器的性能和负载一样，不同客户端请求复杂度不一样导致处理时间也不一样。最小连接数法根据后端服务器当前的连接数情况，动态地选取其中积压连接数最小的一台服务器来处理当前的请求，尽可能提高后端服务器的利用效率，合理地将请求分流到每一台服务器。为什么根据连接数可以合理地利用服务器处理请求呢？考虑一个客户端请求的处理逻辑较复杂，需要服务器的处理时间较长，由于客户端需要等待服务器的响应，故需要保持与服务器的连接，这样一来，客户端就需要与服务器保持较长时间的连接。

4.随机法

随机法也很简单，就是随机选择一台后端服务器进行请求的处理。由于每次服务器被挑中的概率都一样，客户端的请求可以被均匀地分派到所有的后端服务器上。

5.源地址哈希法

源地址哈希的思想是根据获取客户端的IP地址，通过哈希函数计算得到的一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是客服端要访问服务器的序号。采用源地址哈希法进行负载均衡，同一IP地址的客户端，当后端服务器列表不变时，它每次都会映射到同一台后端服务器进行访问。如果后端服务器是一缓存系统，当后端服务器增加或者减少时，采用简单的哈希取模的方法，会使得命中率大大降低，这个问题可以采用一致性哈希的方法来解决。

2.2.4 数据复制和一致性

2.2.4.1 复制的概述

复制几乎是构成分布式系统，尤其是分布式存储和分布式数据库的关键所在，那么本文就来综合谈论下复制技术。

简单说复制本身可以分为同步复制和异步复制，两者的区别在于前者需要等待所有副本返回写入确认，而后者只需要一个返回确认即可。从用途上，复制可

以分为两类，一类用于确保不同副本的表现行为一致，另一类则用于允许不同副本之间的数据差异，先来看看前者。有若干种手段用于确保不同副本之间的状态一致。

第一种叫主从复制。主从之间可以是异步复制，也可以是同步复制。例如MySQL，在默认情况下采用异步复制，异步复制容易引起数据丢失，比如主从结构中，主节点的写入请求还没有复制到从节点就挂了，当从节点被选为新的主节点之后，在这之前写入没有同步的数据就会被丢失。即便采用了同步复制，也只能提供相对较弱的基本保障，考虑如下情形：主接收写入请求然后发到从节点，从节点写入成功后并发送确认给主，如果此时主节点正准备发送确认信息给客户端时挂了，那么客户端就会认为提交失败，可是从节点已经提交成功了，如果这是从节点被提升为主，那么就出现问题了。在主从复制结构里，异步复制相比同步复制具备更高的吞吐量和更低延迟，因此，结合同步和异步复制是一个常见选项。第三种则引入分区一致性算法Paxos，又叫复制状态机。复制状态机在数据库开发的很多领域都可以遇到，比如Google Megastore，针对不同分区的每次提交采用复制状态机来确保每个分区的全局事务提交时序； Google Spanner在单分区内也采用了类似的设计。复制状态机主要用于满足两点需求：客户端在面对任何一个副本时都具备完全一致的访问行为；每个副本在执行请求时都需要按照完全一致的顺序来进行。

2.2.4.2 一致性和可用性

在理论计算机科学中，CAP定理，又被称作布鲁尔定理，它指出对于一个分布式计算系统来说，不可能同时满足以下三点：

1.一致性

2.可用性

3.容忍网络分区

根据定理，分布式系统只能满足三项中的两项而不可能满足全部三项。理解CAP理论的最简单方式是想象两个节点分处分区两侧。允许至少一个节点更新状态会导致数据不一致，即丧失了C性质。如果为了保证数据一致性，将分区一侧的节点设置为不可用，那么又丧失了A性质。除非两个节点可以互相通信，才能既保证C又保证A，这又会导致丧失P性质。

CAP理论在互联网界有着广泛的知名度，知识稍微宽泛一点的工程师都会把其作为衡量系统设计的准则。大家都非常清楚地理解了CAP：任何分布式系统在

可用性、一致性、分区容错性方面，不能兼得，最多只能得其二，因此，任何分布式系统的设计只是在三者中的不同取舍而已。

CAP还有一个方面很多人认识不清，那就是放弃一致性其实有隐藏负担，即需要明确了解系统中存在的不变性约束。满足一致性的系统有一种保持其不变性约束的自然倾向，即便设计师不清楚系统中所有的不变性约束，相当一部分合理的不变性约束会自动地维持下去。相反，当设计师选择可用性的时候，因为需要在分区结束后恢复被破坏的不变性约束，显然必须将各种不变性约束一一列举出来，可想而知这件工作很有挑战又很容易犯错。放弃一致性为什么难，其核心还是“并发更新问题”，跟多线程编程比顺序编程难的原因是一样的。

2.3 本章小结

本节介绍数据库有关的理论知识。包括硬件知识和数据库事务相关理论，以及分布式系统系统有关的算法和协议。另外，本章最后讨论了目前几种成熟的分布式的系统，作为实例学习和学习研究。

第三章 系统分析

系统需求分析的工作阶段是软件开发过程的开始阶段，是软件生命周期中的一个重要环节，对于整个软件开发过程以及软件产品的质量是至关重要的，因为构建软件系统最艰难的一个部分就是准确的决定要构建什么，没有其他的概念性工作像建立详细的技术需求这样难，包括所有对人的界面、对机器的接口、以及对其他软件系统的接口。如果做错了会对最终系统造成相当大的损害，并且后期难以调整。而设计是将问题转换为解决方案的的创造性过程，系统设计是在系统分析的基础上研究系统如何实现需求中所描述的功能，给出满足需求的解决方案。本章的主要内容是介绍分布式内存数据库系统的需求分析与设计，需求分析主要包括整个系统的需求分析与功能分析，同时也包括了对各个功能模块的实现分析。

3.1 系统需求分析

需求阶段的目标是理解客户的问题和需要，需求分析是在综合分析用户对系统提出的一组需求（功能、性能、数据等方面）的基础上，构造一个从抽象到具体的逻辑模型表达软件将要实现的需求，并以“软件需求规格说明书”的形式作为本阶段工作的结果，为下一阶段的软件设计提供设计基础。

3.1.1 系统功能需求

本论文所研究的分布式数据库系统，主要对大数据量进行高效处理，需要系统具备大容量存储和高速率运算。传统的关系数据库在应付日益增长的大量数据时暴露了越来越多难以克服的问题，而现在的各种NOSQL数据虽然支持大量数据的存储，但是对关系的操作又很少。因此，本文结合关系型技术与分布式NOSQL技术，来设计能够对大量数据进行高效处理的分布式数据库系统。作为项目的前期工作，本文所设计的分布式数据库系统的主要功能如下：

1.提供负载均衡 本文所述的分布式系统，为了实现系统中各个数据库节点的负载均衡，需要系统具有良好的分散性，即客户端发送的请求消息能够均匀的发送到各数据库节点。因此本文提出了分布式管理几点，该几点的功能构成本文的分布式管理模块的主要功能，其为客服端选择合适的后端分布式数据库几点，从而达到负载均衡的效果。分布式管理节点并不存储消息，它只存储分布式数据库几点的元数据和负载均衡算法的设置，为客服端选择合适的后端服务器。

2.提供数据库功能

站在客户的角度，需要本系统能够提供增加、删除、修改、查询等数据库基本功能操作。本文结合非关系型数据库和关系数据库技术。实现了一种分布式数据库功能。对用户来说，本系统提供的功能接口就是SQL接口，更准确的说，是Mysql接口，这样常用的关系操作都是支持的。同时本系统利用非关系型数据库OrientDB存储，实现数据的高效存储和访问。

3.提供管理监控功能 JSQL系统在分布式管理节点和分布式数据库节点的结合下，从数据库的底层实现了数据库的安全审计的功能。在分布式管理节点，实现了审计监控的界面功能，管理员通过管理计算机连接分布式管理节点就可以对数据库进行管理和维护。同时也能进行监控报警功能的设置。

因此，基于对整个分布式数据库系统的功能需求的分析，系统可分为两部分，提供负载均衡功能和管理监控功能的分布式管理节点，提供增加、删除、修改、查询等数据库基本功能操作的数据库节点。具体的，提供的数据库功能主要包括：

1.数据定义：系统提供数据定义语言DDL，供用户定义数据库的三级模式结构、两级映像以及完整性约束和保密限制等约束。 DDL主要用于建立、修改数据库的库结构。DDL所描述的库结构仅仅给出了数据库的框架，数据库的框架信息被存放在数据字典中。

2.数据操作：系统提供数据操作语言DML ，供用户实现对数据的追加、删除、更新、查询等操作。

3.数据库的运行管理：数据库的运行管理功能是系统的运行控制、管理功能，包括多用户环境下的并发控制、安全性检查和存取限制控制、完整性检查和执行、运行日志的组织管理、事务的管理和自动恢复，即保证事务的原子性。这些功能保证了数据库系统的正常运行。

4.数据组织、存储与管理：系统要分类组织、存储和管理各种数据，包括数据字典、用户数据、存取路径等，需确定以何种文件结构和存取方式在存储级上组织这些数据，如何实现数据之间的联系。数据组织和存储的基本目标是提高存储空间利用率，选择合适的存取方法提高存取效率。

5.数据库的保护：数据库中的数据是信息社会的战略资源，所以数据的保护至关重要。系统对数据库的保护通过4个方面来实现：数据库的恢复、数据库的并发控制、数据库的完整性控制、数据库安全性控制。系统的其他保护功能还有系统缓冲区的管理以及数据存储的某些自适应调节机制等。

6.数据库的维护：这一部分包括数据库的数据载入、转换、转储、数据库的

重组合重构以及性能监控等功能，这些功能分别由各个使用程序来完成。

7.通信：系统具有与操作系统的联机处理、分时系统及远程作业输入的相关接口，负责处理数据的传送。对网络环境下的数据库系统，还应该包括系统与网络中其他软件系统的通信功能以及数据库之间的互操作功能。

3.1.2 系统功能用例

用例图用于描述一组用例、参与者及它们之间的连接关系。一个用例描述了一组动作序列，每一个序列表示系统的外部设施（系统的参与者）与系统本身的交互。本系统的用例图是从参与者使用系统的角度来描述系统中的信息，即站在系统外部查看系统应该具有何种的功能，而不是描述功能在系统内是如何实现的。本文设计的分布式数据库系统的主要功能为新增数据、查询数据、修改数据、删除数据。系统总体的用例图如图3-1所示。

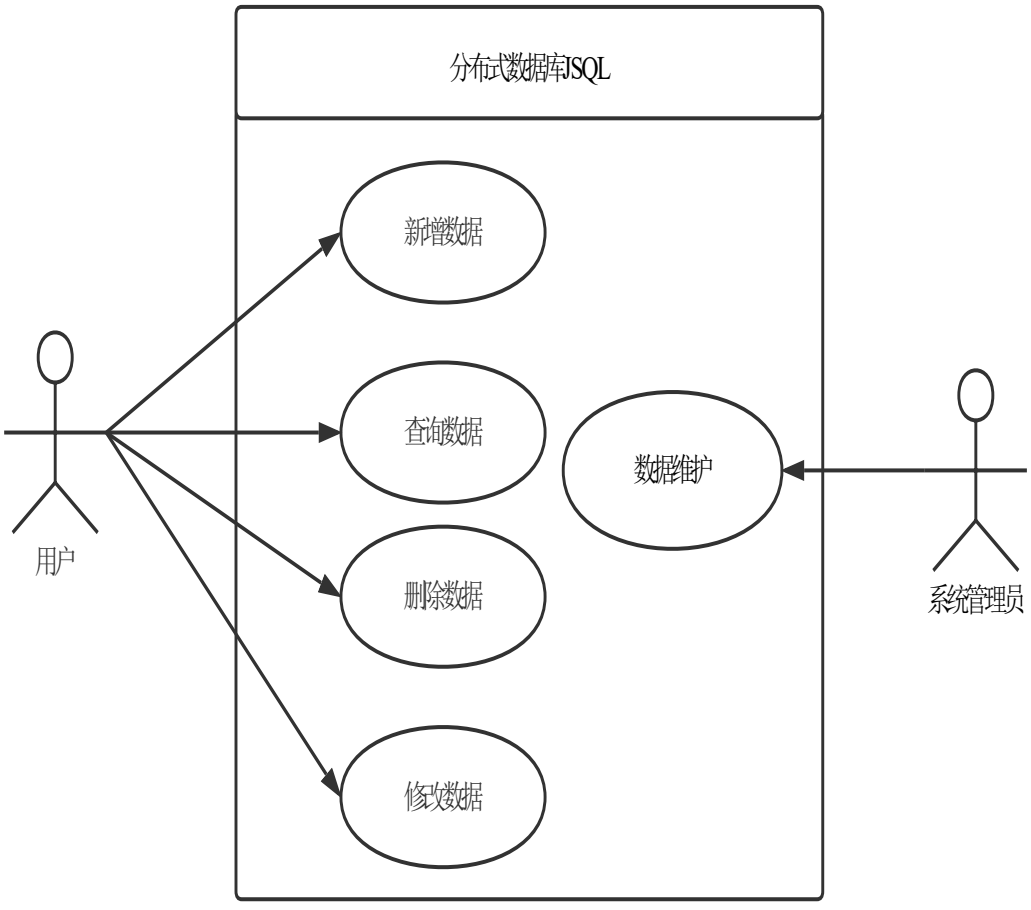


图 3-1 系统用例图

分布式内存数据库的主要功能用例为新增数据、查询数据、修改数据、删除数据等，下面对这几种操作的用例进行说明。

1.新增数据是用户从客户端向系统发起新增数据请求，具体用例见表3-1。

表 3-1 新增数据用例

用例名称	新增数据
用例描述	客户端发送新增数据的请求到分布式数据库
参与者	分布式数据库的用户
前置条件	1、 系统各个模块运行正常。 2、 用户已经成功成功登陆系统。
后置条件	分布式数据库系统能够正确处理客户端请求，成功保存数据。
基本操作	1、用户运行客户端，用户登陆。 2、在客户端上发送新增数据的命令。 3、在客户端上确认发送信息后，请求消息被发送到分布式数据库。 4、数据库存储成功后返回操作成功，如果失败，返回失败原因。
业务规则	如果数据信息已经存在，重复发送命令会增加新的数据

2.数据查询是指利用用户在客户端上输入指定数据标识作为关键字，查询该条数据的基本信息，具体用例见表3-2。

表 3-2 查询数据用例

用例名称	查询数据
用例描述	用户通过客户端发起数据查询的请求
参与者	分布式数据库用户
前置条件	1、 分布式数据库系统各模块运行正常。 2、 数据信息已经存入数据库。 3、 用户已经成功登陆客户端。
后置条件	分布式数据库系统能够处理客户端请求，正确返回用户查询数据。
基本操作	1、 运行客户端，登陆分布式数据库。 2、 在客户端上发送查询数据的命令。 3、 在客户端上确认信息后，请求消息被发送到存储该数据的数据库。 4、 数据库服务器查询成功后返回数据信息，如果失败，返回失败原因。
业务规则	客户端发送查询命令时，携带数据标识字段作为查找关键字。

3.删除数据是指当数据无效时，客户端发送删除信息请求分布式数据库系统删除该条数据，具体用例见表3-3。

4.修改数据是指当数据需要修正时，管理员通过客户端发送修改消息请求，通知分布式内存数据库系统存入新数据，刷新数据变化部分，具体用例见表3-4。

表 3-3 删除数据用例

用例名称	删除数据
用例描述	管理员或者用户通过客户端发起数据删除的请求
参与者	分布式数据库管理员和用户
前置条件	1、 分布式数据库系统运行正常。 2、 数据信息已经存在。 3、 管理员或者用户已成功登陆系统。
后置条件	分布式数据库系统能够正确处理客户端请求，删除该条数据。
基本操作	1、 运行客户端，登陆数据库服务器。 2、 在客户端上发送删除数据库的命令。 3、 在客户端上确认信息后，请求消息被发送到分布式数据库。 4、 数据库处理成功后返回删除成功响应，如果失败，返回失败原因。
业务规则	1、 客户端发送删除命令是，必须要指定关键字。 2、 有权限的用户才能删除数据。

表 3-4 修改数据用例

用例名称	修改数据
用例描述	管理员或者用户通过客户端发起数据修改请求
参与者	分布式数据库管理员和用户
前置条件	1、 分布式数据库系统运行正常。 2、 数据信息已经存在。 3、 管理员或者用户已成功登陆系统。
后置条件	分布式数据库系统能够处理客户端请求，正确修改该条数据发生变化的部分。
基本操作	1、 运行客户端，登陆分布式数据库服务器。 2、 在客户端上发送修改数据的命令。 3、 在客户端上确认信息后，请求消息被发送到分布式数据库。 4、 数据库查询成功后返回修改成功响应，如果失败，返回失败原因。
业务规则	1、 客户端填写数据信息时，必须要指定关键字。 2、 不能修改数据标识，其余修改的值在有效范围内。 3、有权限的用户才能修改数据。

3.2 技术和框架分析

3.2.1 系统实现语言选择

本数据库系统选择了JAVA语言和KOTLIN语言作为开发语言，之所以这样选择，主要基于以下几个原因：

1.JAVA是跨平台的开发语言，用JAVA开发实现数据库系统，可用实现所有主流平台的数据库部署。

2.JAVA是企业开发的首选语言，其安全性比本地语言更高。

3.非关系型数据库引擎OrienDB也是用JAVA开发的，为了方便调用。本系统也应该使用同一种语言。

4.JAVA易于开发和调试，作为学生时期个人开发的数据库系统，一个人的精力是有限的，如何选择高效的开发工具和语言对我们来说非常重要。

3.2.2 网络实现技术分析

所有的服务软件都需要实现网络模块，这样才能连接客服端的请求。JSQL用java语言开发，主要用到的是java的网络开发模块。

网络IO的方式通常分为同步阻塞的BIO、同步非阻塞的NIO和异步非阻塞的AIO[1]。

在JDK1.4出来之前，我们建立网络连接的时候采用BIO模式，需要先在服务端启动一个ServerSocket，然后在客户端启动Socket来对服务端进行通信，默认情况下服务端需要对每个请求建立一堆线程等待请求，而客户端发送请求后，先咨询服务端是否有线程相应，如果没有则会一直等待或者遭到拒绝请求，如果有的话，客户端会线程会等待请求结束后才继续执行。

NIO本身是基于事件驱动思想来完成的，其主要想解决的是BIO的大并发问题：在使用同步I/O的网络应用中，如果要同时处理多个客户端请求，或是在客户端要同时和多个服务器进行通讯，就必须使用多线程来处理。也就是说，将每一个客户端请求分配给一个线程来单独处理。这样做虽然可以达到我们的要求，但同时又会带来另外一个问题。由于每创建一个线程，就要为这个线程分配一定的内存空间（也叫工作存储器），而且操作系统本身也对线程的总数有一定的限制。如果客户端的请求过多，服务端程序可能会因为不堪重负而拒绝客户端的请求，甚至服务器可能会因此而瘫痪。

在高性能的I/O设计中，有两个比较著名的模式Reactor和Proactor模式，其中Reactor模式用于同步I/O，而Proactor运用于异步I/O操作。在比较这两个模式之前，我们首先的搞明白几个概念，什么是阻塞和非阻塞，什么是同步和异步,同步和异步是针

对应用程序和内核的交互而言的，同步指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪，而异步是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知。而阻塞和非阻塞是针对于进程在访问数据的时候，根据IO操作的就绪状态来采取的不同方式，说白了是一种读取或者写入操作函数的实现方式，阻塞方式下读取或者写入函数将一直等待，而非阻塞方式下，读取或者写入函数会立即返回一个状态值。

Netty的主要目的是建立基于NIO的高性能协议服务器，分离和松散耦合网络和业务逻辑组件。它可能会实现一个广为人知的协议，如HTTP或您自己的特定协议。 Netty的线程模型如如4-7所示。 Netty是一个非阻塞框架。与阻塞IO相比，这

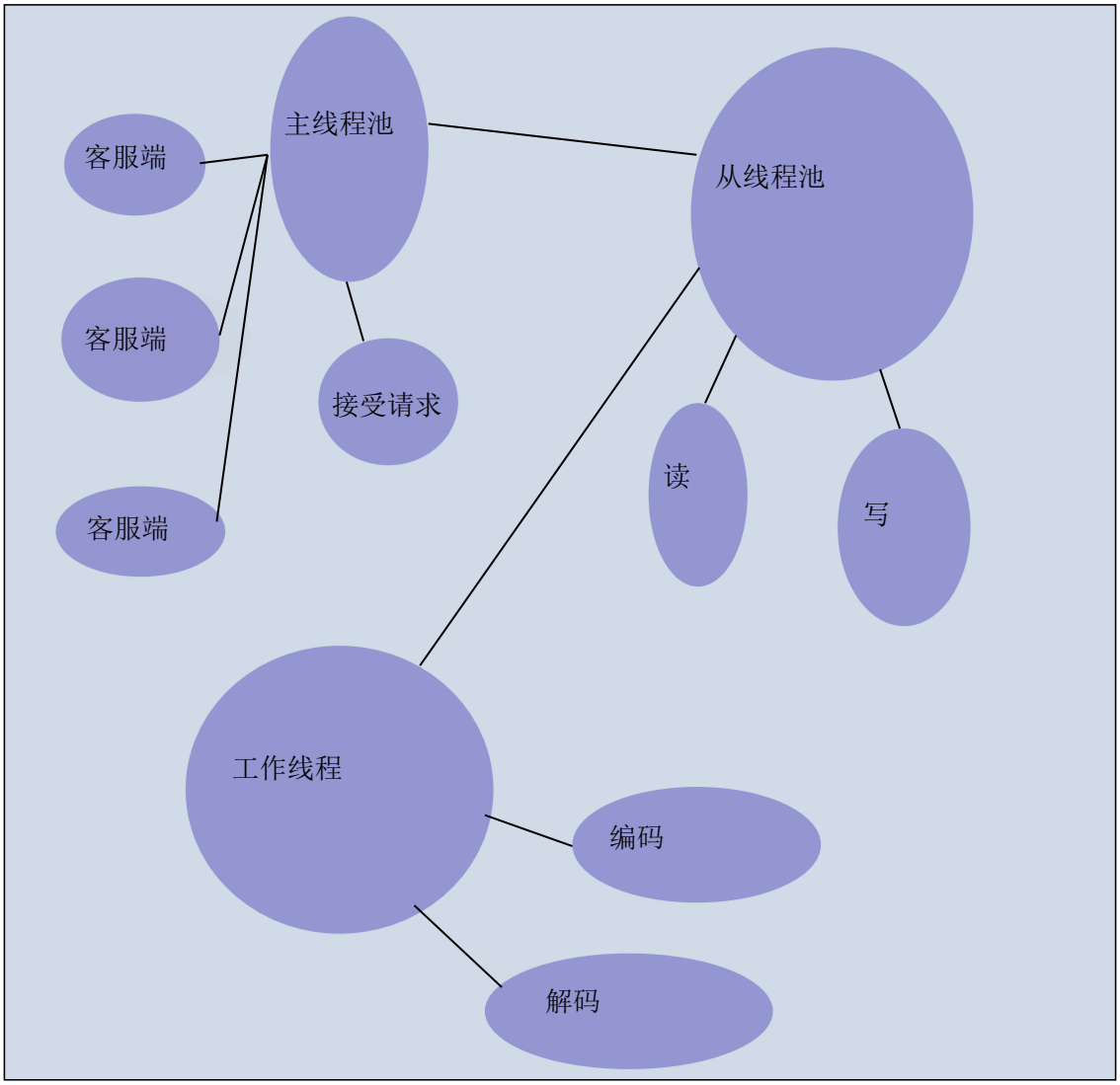


图 3-2 Netty

导致高吞吐量。 Netty使用事件驱动的应用程序范例，因此数据处理的流水线是一系列事件处理程序。

因为Netty的这些特效，利用它可以实现一个高性能的网络应用程序，所以本系统的网络模块也是用的Netty来实现的，它直接高并发的客服端访问。

3.2.3 通信协议分析

每个服务器和客服端通信都要实现自己的通信协议，考虑到mysql使用的广泛性。jsql采用mysql的通信协议。这样就不用自己开发协议了。在MySQL数据库通信过程开始时，服务器会使用TCP监听一个本地socket 端口或本地socket链接。当一个客户端的连接请求到达，就会执行握手和权限验证。如果验证成功，会话开始。客户端发送消息，服务器会以一个适合该发送命令的数据类型的数据集或一条消息进行回复。当客户端发送完成后，会发送一个特殊的命令，告诉服务器己发送，然后会话结束。通信的基本单位是应用程序包。多个指令可以合成一个包；答复可以包含多个包。 MySQL客户端与服务器的交互主要分为两个阶段：握手认证阶段和命令执行阶段：

1.握手认证阶段

握手认证阶段为客户端与服务器建立连接后进行，交互过程如下：

(a) 服务器发送给客户端握手初始化报文。

(b) 客服端回复服务器端登陆认证报文。

(c) 服务器发送给客户端认证结果报文。

2.命令执行阶段

客户端认证成功后，会进入命令执行阶段，交互过程如下：

(a) 客户端发送服务器执行命令报文。

(b) 服务器发送给客服端命令执行结果。

MySQL客户端与服务器之间的完整交互过程如图4-8所示。

3.2.4 SQL实现分析

结构化查询语言是一种特殊目的编程语言，用于数据库中的标准数据查询语言，IBM公司最早使用在其开发的数据库系统中。1986年10月，美国国家标准协会对SQL进行规范后，以此作为关系式数据库管理系统的标准语言， 1987年得到国际标准组织的支持下成为国际标准。不过各种通行的数据库系统在其实践过程中都对SQL规范作了某些编改和扩充。所以，实际上不同数据库系统之间的SQL不能完全相互通用 , 甚至不同版本间也可能无法互通。

SQL是高级的非过程化编程语言，它允许用户在高层数据结构上工作。它不要求用户指定对数据的存放方法，也不需要用户了解其具体的数据存放方式。而

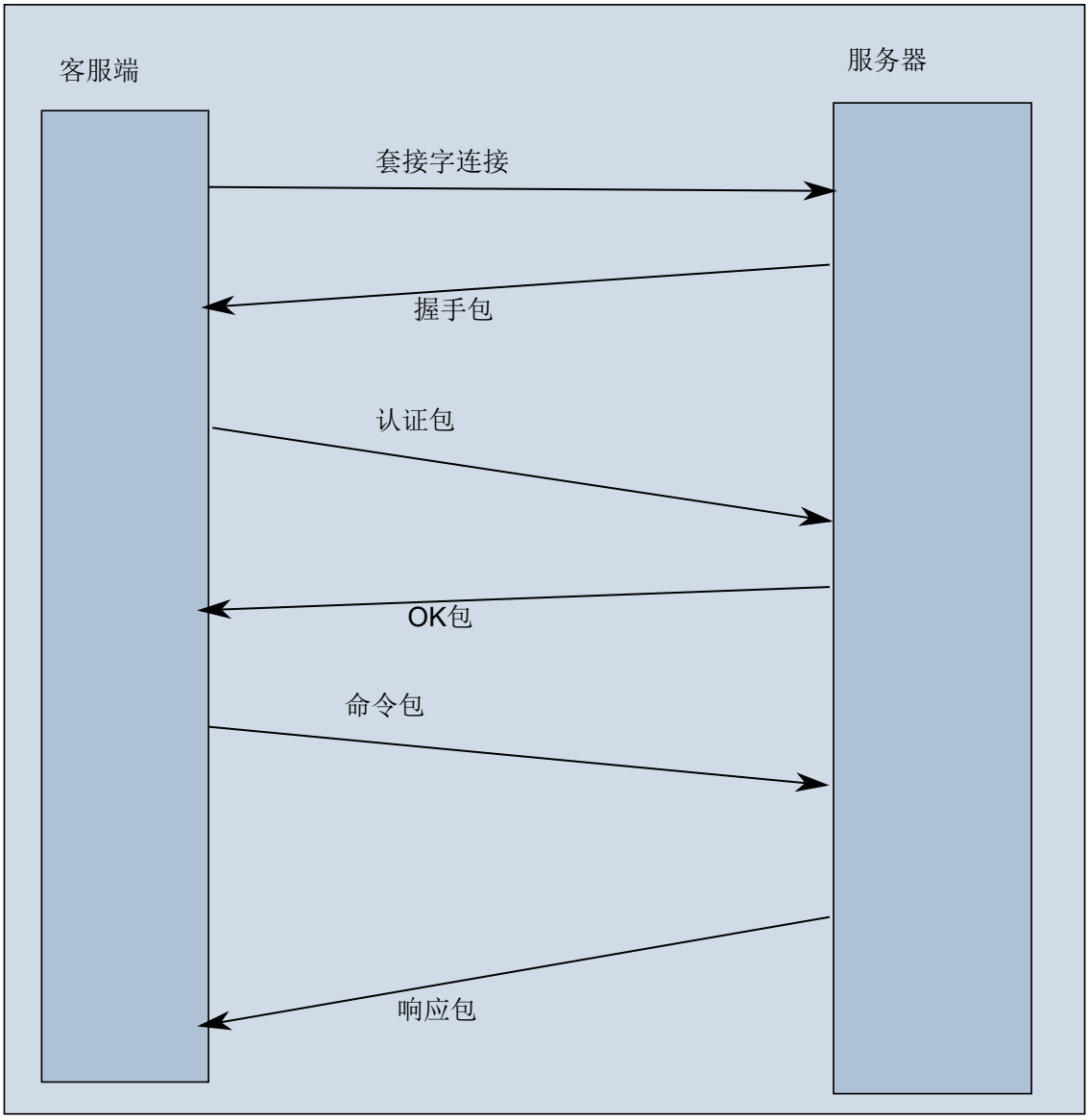


图 3-3 协议交互流程图

它的界面，能使具有底层结构完全不同的数据库系统和不同数据库之间，使用相同的SQL作为数据的输入与管理。

Mysql支持的sql语句和标准的sql语句不全一样，它支持下面这几种语句类型：

1.数据定义语句
2.数据操作语句
3.交易和锁定声明
4.复制语句
5.准备的SQL语句语法
6.复合语句语法
7.数据库管理语句
8.效用声明

其中每种语句类型又分为很多种sql语句，所以mysql支持的sql语句非常的多，因为本系统选择了兼容mysql的协议，所以我们也要解析这些不同的sql语句，本系统利用Druid框架实现了对SQL语句的解析功能。

3.2.5 存储引擎分析

存储引擎是存储系统的发动机，直接决定了存储系统能够提供的性能和功能。存储系统的基本功能包括：增、删、读、改，其中，读取操作又分为随机读取和顺序扫描。每种存储引擎底层都基于一种数据结构。比如常用的哈希表结构和B+树结构。本系统所用的OrientDB存储引擎底层就是用的B+树结构。利用非关系数据库OrientDB实现可靠的数据存储。

3.2.6 分布式实现分析

虽然现在很多NOSQL数据库都实现了分布式，但是本系统还是利用Hazelcast自己开发分布式功能。在计算中，Hazelcast是基于Java的开源内存数据网格。它也是开发产品的公司的名称。Hazelcast公司由风险投资资助。在一个Hazelcast网格中，数据被一个均匀的节点之间分配计算机集群，从而允许水平缩放的处理和可用存储。备份也分布在节点之间，以防止任何单个节点的故障。Hazelcast通过内存中访问经常使用的数据和可弹性扩展的数据网格，提供中央，可预测的应用程序扩展。这些技术减少了对数据库的查询负载，并提高了速度利用Hazelcast可以轻松开发出各种的分布式应用程序，而且部署非常简单，只需要嵌入它的二进制包就可以直接使用，避免再部署另外的系统来实现分布式，所以本系统选择了它来实现分布式的功能。

3.2.7 监控功能实现分析

监控功能主要在分布式管理节点部署，管理员通过连接分布式管理节点，就能对分布式数据库进行监控。对数据进行操作。本系统实现最简单的监控模块，一个监控模块首先要存储所有的日志数据，而且这个数据不能随意的更改，所以我改了Elasticsearch的源代码，让它来存储所有的sql更新记录，然后用可视化的框架来显示出结果。之所以用这个框架，主要是因为它是一种搜索引擎，能够非常快速的检束出我们需要的各种信息，这对于信息审计来说非常的重要。

3.3 本章小结

本章对分布式数据库系统进行需求分析。首先，站在用户的角度对系统的需求进行具体的分析，描述系统的功能需求，并结合系统用例图给出功能用例。然后本章分析了系统每个功能模块所需要的技术，每个模块的实现都要用到不同的技术，对每种技术进行了分析。

第四章 系统设计

在软件工程中，需求分析之后就是设计阶段。首先，开发者需要对软件系统进行概要设计，即系统设计。概要设计需要对软件系统的设计进行考虑，包括系统的基本处理流程、系统的组织结构、模块划分、功能分配、接口设计、运行设计、数据结构设计和出错处理设计等，为软件的详细设计提供基础。在概要设计的基础上，开发者需要进行软件系统的详细设计。在详细设计中，描述实现具体模块所涉及到的主要算法、数据结构、类的层次结构及调用关系，需要说明软件系统各个层次中的每一个程序(每个模块或子程序)的设计考虑，以便进行编码和测试。应当保证软件的需求完全分配给整个软件。详细设计应当足够详细，能够根据详细设计报告进行编码。只有经过仔细的系统设计，在编码阶段才能提前减少不必要的错误。

4.1 系统总体设计

4.1.1 系统架构设计

原型化设计意味着构建一个系统的小版本，通常只有有限的功能，它可用于：

1.帮助用户和客户标识系统的关键需求。

2.证明设计或方法的可行性。

通常，原型化过程是迭代的：首先构建原型，然后对原型进行评估（利用用户和客户的反馈），考虑如何改变可以改进产品或设计，之后再构建另外一个原型。当开发团队和客户认为解决方案满意时，迭代过程就终止了。由于本文所涉及的分布式数据库系统是作为实验室数据库，并没有做实际上线的考虑，因此采用了原型化模型的软件开发方式开发一个数据库原型。重点针对分布式数据库系统的大容量可扩展性进行设计，主要完成数据库的增加、查询、修改、删除等功能，所以对系统的架构作了简化调整。系统的网络拓扑结构如图4-1所示。 在系统网络拓扑图中，主要有下面几种节点：

1.客服端节点

客服端节点作为连接分布式数据库服务器的代理，发送SQL命令请求到数据库服务器，请求数据，客服端节点可以是原生的Mysql节点，也可以是自己开发的具有负载均衡的JDBC客服端。如果是用Mysql客服端，就可以直接连接分布式数据库节点，可以任意选择一台分布式数据库节点执行数

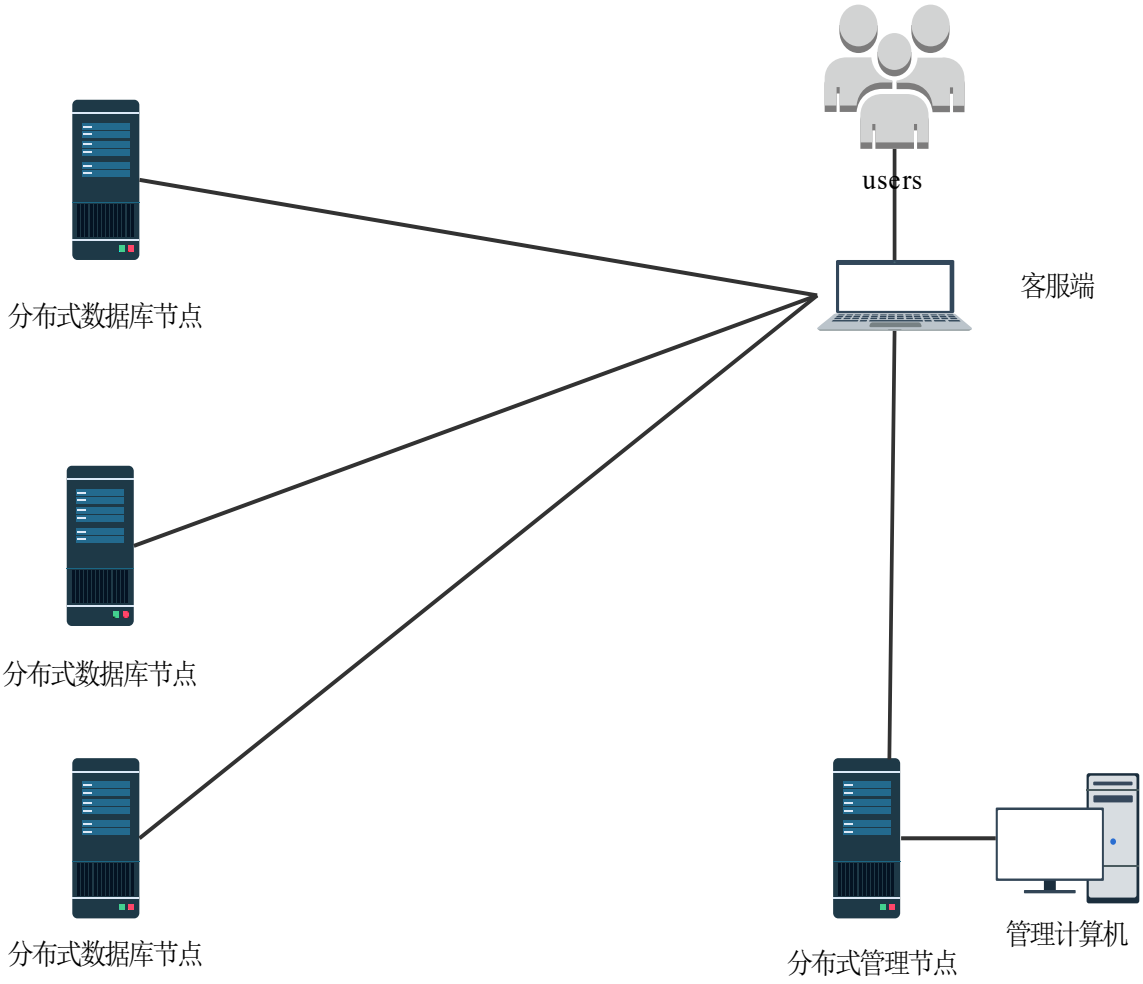


图 4-1 系统网络拓扑结构图

据库操作命令，其中数据库会自动同步到其他数据库节点。当用自己开发的JDBC客服端的时候，客服端就会首先连接分布式管理节点，得到具体的分布式数据库节点后，才能连接数据库节点执行具体的数据库命令。

2.分布式管理节点

分布式管理节点管理分布式数据库节点的元数据和负载均衡算法。管理计算机通过连接分布式管理节点，可以对分布式节点进行管理，也可以选择适合的负载均衡算法。这样当客服端请求到来的适合，分布式管理节点就可以返回适合的分布式数据库节点，从而达到负载均衡的效果。

3.分布式数据库节点

分布式节点存储具体的数据，数据库系统功能主要在这个节点上面执行，响应客服端的增删改查的命令请求。当客服端连接分布式数据库节点以后，会检查客服端的权限，只有有权限的用户才能执行命令。

4.管理计算机

管理员通过管理计算机管理分布式数据库几点和负载均衡算法。也可以通过管理计算机来查看审计信息，设置监控报警功能。

在系统网络拓扑结构图中，客户端模拟负责发送命令的终端设备，各个分布式数据库节点和分布式管理节点构成了系统的服务器端。其中，分布式管理节点作为中心节点接收所有的控制消息，通过分布式管理节点才能找到分布式数据库节点的IP地址。得到地址以后，客服端再连接数据库节点，得到具体的数据，分布式管理节点并不存储数据信息，数据只存储在分布式存储节点中。根据原型化模型和系统的网络拓扑结构图，本文所涉及的分布式数据库系统的整体结构是：有两种计算机，一台作为客户端使用，另一台作为服务器使用，根据需求设定服务器端的数据库节点数个数N。这N存数据库之间相互独立、存储各自的数据信息。它们的数据结构、存储方式都是相同的，由服务器端的分布式管理节点负责把客户端发来的数据分发到对应的数据库节点，因此不同的数据库节点中存放着不同的数据。每一个数据库节点构成了自己的数据库单元，每个数据库单元具有独立处理的能力，它可以执行局部应用。同时，如果这 N 个数据库物理上应用于多台计算机时，每个数据库节点也能通过网络通信子系统执行全局应用。本系统的架构图如图4-2所示。 本系统中，客户端模拟真实终端设备。分布式管理节点收到客服端请求以后，返回给客服端数据库节点的地址，然后客服端再连接数据库节点，数据库节点负责对请求消息进行处理，包括增加、删除、修改、查询等操作，并将响应消息返回给客户端。分布式管理节点与各个数据库模块在同一台机器中运行，也可以分布到不同的计算机上面。在系统可靠性保护方面，当出现

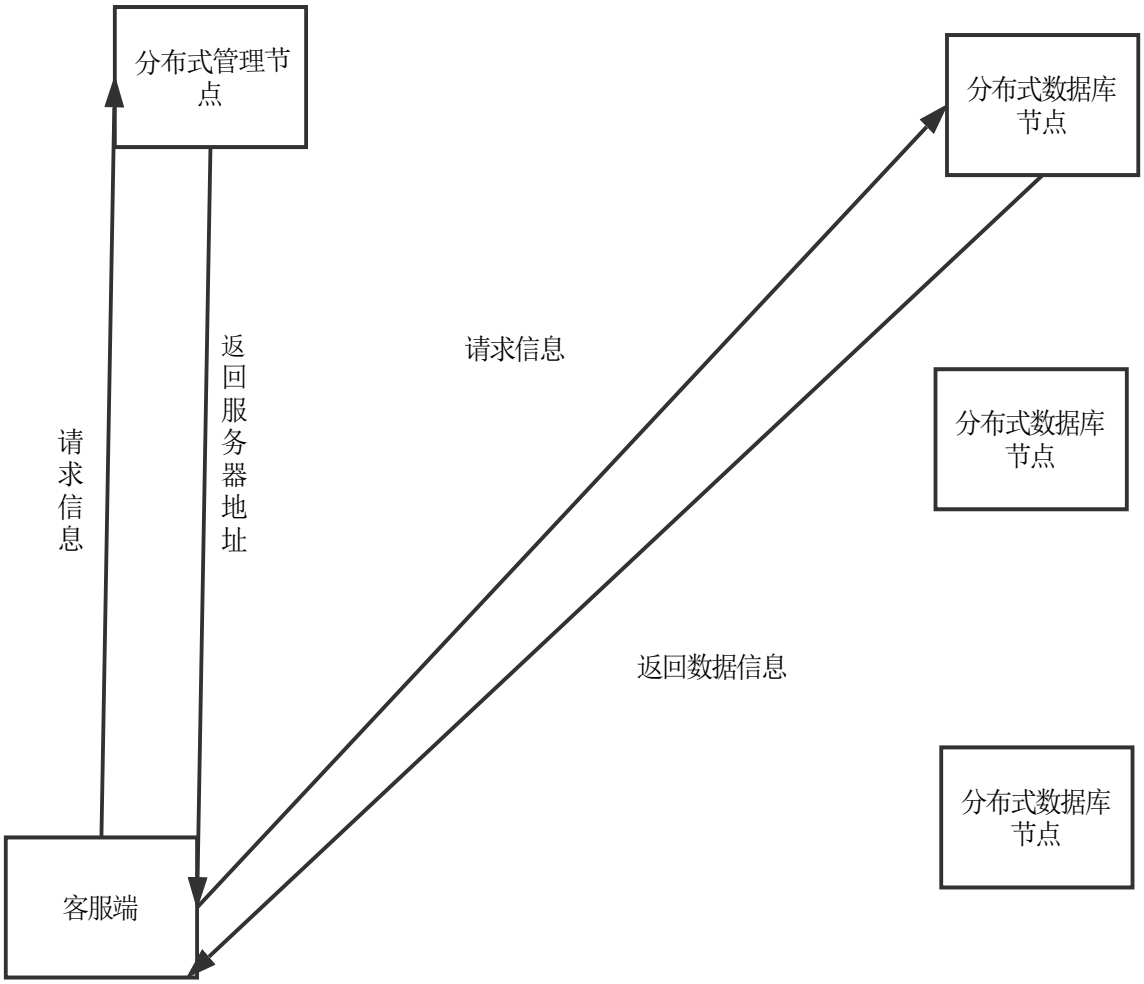

分布式管理节点
分布式数据库节点
分布式数据库节点
分布式数据库节点
请求信息
返回服务器地址
请求信息
返回数据信息
客服端

图 4-2 系统总体架构图

某个数据库模块发生故障时，分布式管理节点会实时的更新管理信息，同时其他数据库节点也会同步这台计算机的信息。数据因为在不同的计算机上面都有备份，所以不存储丢失的危险。

4.1.2 系统功能设计

根据系统的需求分析，采用模块化分解结构划分本系统，使系统能够进行增加、删除、查询、修改等数据库基本操作。系统采用传统的C/S架构，分布式管理节点和数据库节点共同组成了服务器端。分布式数据库系统的功能模块分解图如图4-3所示。

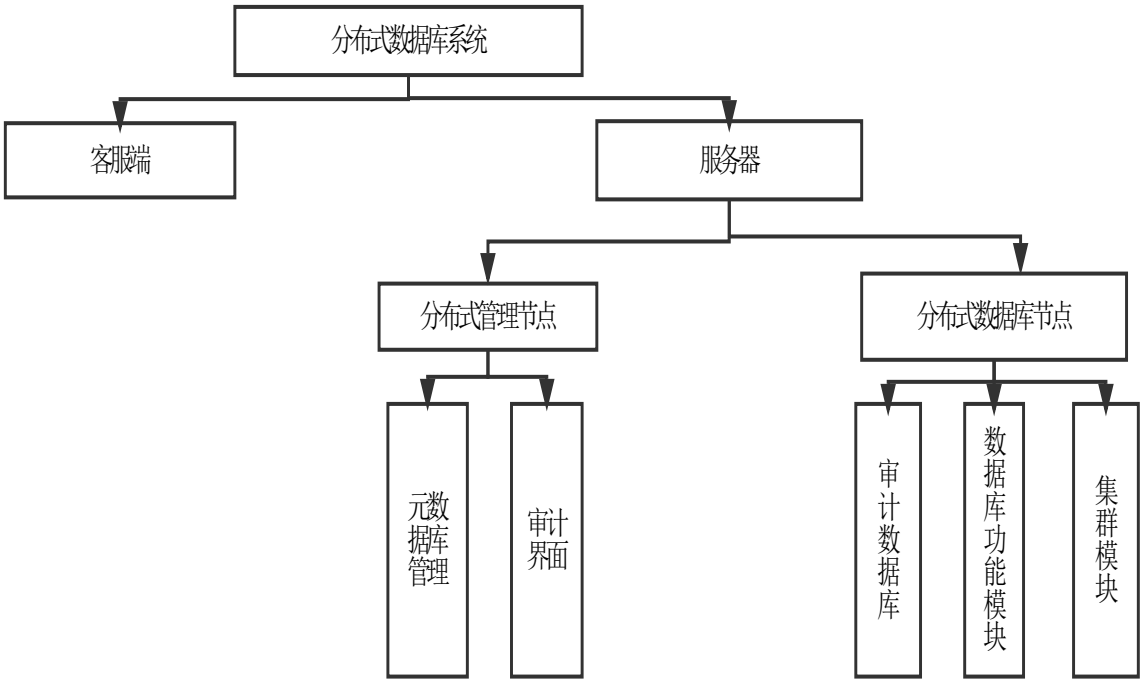


图 4-3 系统总体架构图

以下对图4-3中服务器端各个模块的功能进行简要说明。分布式管理节点按逻辑功能划分为两个子模块，元数据管理模块、审计界面模块。下面分别对两个子模块的功能作介绍。

1.元数据管理模块管理服务器的所有元数据，包括每个数据库节点的 IP地址和端口号。分布式管理节点利用这些元数据和适当的负载均衡算法来为客服端选择合适的分布式数据库节点。

2.审计界面管理模块对管理员使用，它可以让管理员管理所有的数据节点。审计功能除了审计界面以外，还包括分布式数据库节点的审计数据库。只有结合了分布式数据库的审计模块和分布式管理节点的审计模块，管理员才能对分布式系统进行审计管理。

分布式数据库节点由审计数据库模块、数据库功能模块、集群功能模块组成，下面分别对 3 个模块的功能进行介绍。

1.审计数据库模块用来收集所有的日志信息，作为审计使用。所有的数据库节点在接受到数据更改请求以后，就会把这些日志信息发送到审计数据库。然后分布式管理节点就能利用这个数据为管理员提供审计功能。

2.数据库功能模块完成具体的数据库功能。为用户提供操作接口，数据库功能模块可以分为网络模块，协议模块，解析模块和存储模块。在收到用户的请求命令以后。数据库功能模块就会对用户返回正确的信息。

3.集群功能实现每个数据库节点的通信。本系统利用Hazelcast实现了数据库的集群。每个分布式数据库节点的数据都能自动同步。同时，利用分布式多版本并发控制技术解决了数据的一致性问题。

4.2 客服端模块设计

分布式系统需要解决的一个重要问题便是决定数据在集群中的分布策略，好的分布策略应该能将数据均衡地分布到所有节点上，并且还应该能适应集群节点的变化，本文采用的分布式管理节点较好地满足了这两点。分布式管理节点存储所有数据库节点的元数据信息，同时也能存储负载均衡的算法信息，管理员可以决定采用什么算法那分配服务器资源。对于分布式系统，系统中每个节点的负载均衡是非常重要的，分布式算法必须具有良好的分散性，使消息能够均匀的转发到各个数据库节点。哈希算法对分散性的支持很好的满足了系统对分散性的要求，并且利用哈希算法能够保证同一号码经过哈希算法计算后，每次的哈希值都是相同的，这可以保证分布式管理节点对同一数据标识的消息多次转发时，每次都能将消息正确转发到同一数据库中。但是随着数据库节点数量的变化，客服端必须要重新计算服务器节点地址，所以当我们想要使用负载均衡的时候我们必须要先连接管理节点，再连接数据库节点。客服端连接的流程图如图4-4所示。 每个流程解释如下：

1.首先客服端连接到分布式管理节点，管理节点返回给客服端正确的数据库节点的地址。

2.然后客服端连接到正确的数据库节点。

3.最后数据库节点返回给客服端具体的数据。

当然我们也可以直接连接分布式数据库节点，其中的数据会自动同步到其他的分布式数据库节点，这样可以直接用Mysql的客服端，减少用户的使用成本。使用Mysql客服端，可以执行几乎所有的Mysql命令。在用户看来，分布式数据库节

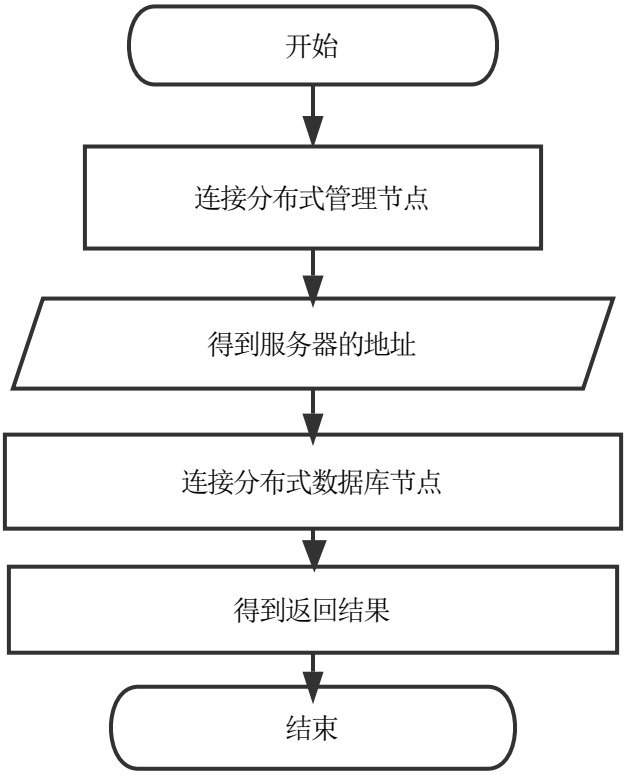


图 4-4 客服端连接流程图

点就像一个Mysql数据库服务器一样。只是本系统能自动同步每个分布式数据库节点的数据。

4.3 分布式管理模块

要实现负载均衡和其他管理功能，只实现客服端是不够的。还需要一个分布式管理节点。分布式管理模块存储所有数据库节点的元数据和负载均衡的算法，当启动分布式管理节点的时候，它会去查找所有的数据库节点，然后存储到数据库里面保存。同时管理员也可以对负载均的算法进行管理。分布式管理节点的启动流程图如图4-5所示。 分布式管理节点和分布式数据库节点一样，都是服务器

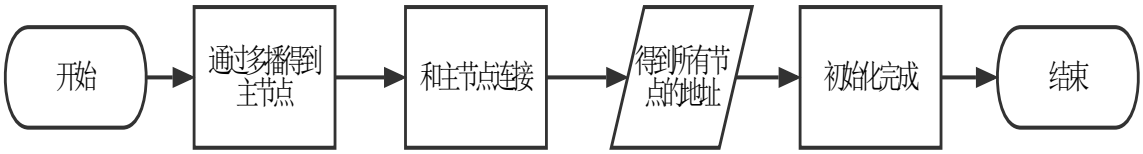


图 4-5 分布式管理节点初始化过程

端的节点，都实现了同一个分布式功能。他们之间可以通过多播通信来交换信息。分布式管理节点得到这些信息以后。就能使用正确的负载均衡算法来选择合适的分布式数据库节点。分布式管理节点初始化成功以后，客服端就可以通过自己开发的基于JDBC的客服端来连接后端的分布式数据库节点，发送数据查询命令。

4.4 数据库功能模块详细设计

数据库功能模块主要负责根据命令消息类型来完成对应的业务逻辑处理。本模块利用资源管理模块提供的接口，完成数据的增加、删除、修改、查询等业务功能，并将响应消息返回给客户端。下面设计出每个操作业务的流程。

4.4.1 数据库功能操作流程

4.4.1.1 新增数据过程

第一步：客户端发送新增数据请求消息，消息包到达服务器端的分布式管理节点，由分布式管理节点返回客服端的连接会话信息，客服端得到服务器的地址。第二步：客服端连接具体的数据库节点，发送新增数据请求，请求发送到具体的数据库节点。数据库节点经过网络模块和通信模块，得到请求信息以后，调用数据库引擎的接口，得到具体的数据，返回给客服。

4.4.1.2 数据查询和修改过程

第一步：客户端发送查询和修改数据请求消息，消息包到达服务器端的分布式管理节点，由分布式管理节点返回客服端的连接会话信息，客服端得到服务器的地址。第二步：客服端连接具体的数据库节点，发送新增数据请求，请求发送到具体的数据库节点。数据库节点经过网络模块和通信模块，得到请求信息以后，调用数据库引擎的接口，得到具体的数据，返回给客服。

4.4.1.3 数据删除过程

第一步：客户端发送删除数据请求消息，消息包到达服务器端的分布式管理节点，由分布式管理节点返回客服端的连接会话信息，客服端得到服务器的地址。第二步：客服端连接具体的数据库节点，发送新增数据请求，请求发送到具体的数据库节点。数据库节点经过网络模块和通信模块，得到请求信息以后，调用数据库引擎的接口，得到具体的数据，返回给客服。数据库模块是本系统的主要开发模块，其详细模块图如图4-6所示。 下面对每个功能模块进行详细的说明。

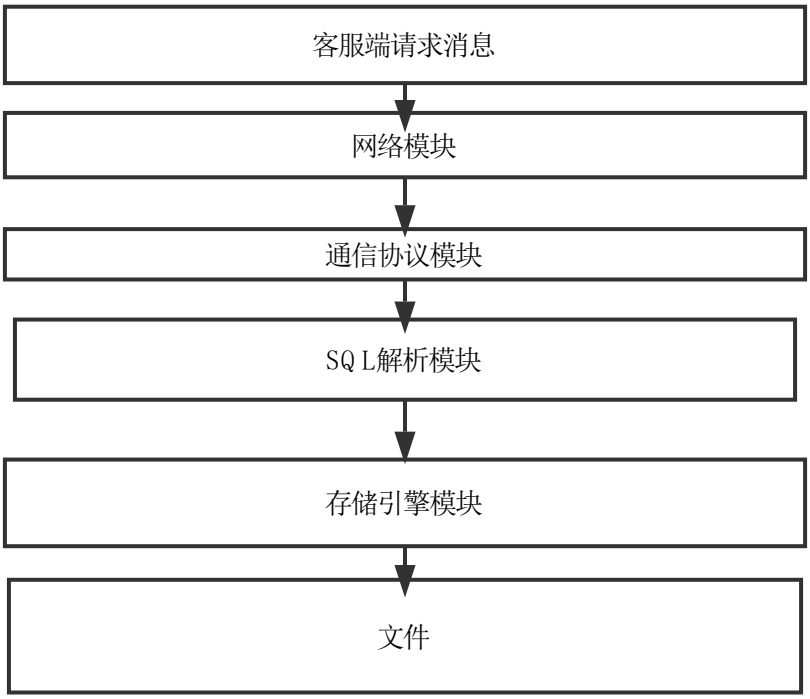


图 4-6 数据库功能模块图

4.4.2 网络模块设计

所有的服务软件都需要实现网络模块，这样才能连接客服端的请求。JSQL用java语言开发，主要用到的是java的网络开发模块。但是直接用java语言开发网络功能非

常的麻烦，所以本文基于Netty来开发直接的网络模块。具体的网络实现，包括线程池的规划都按照Netty的建议来开发。Netty的线程模型如图4-7所示。 根据图4-

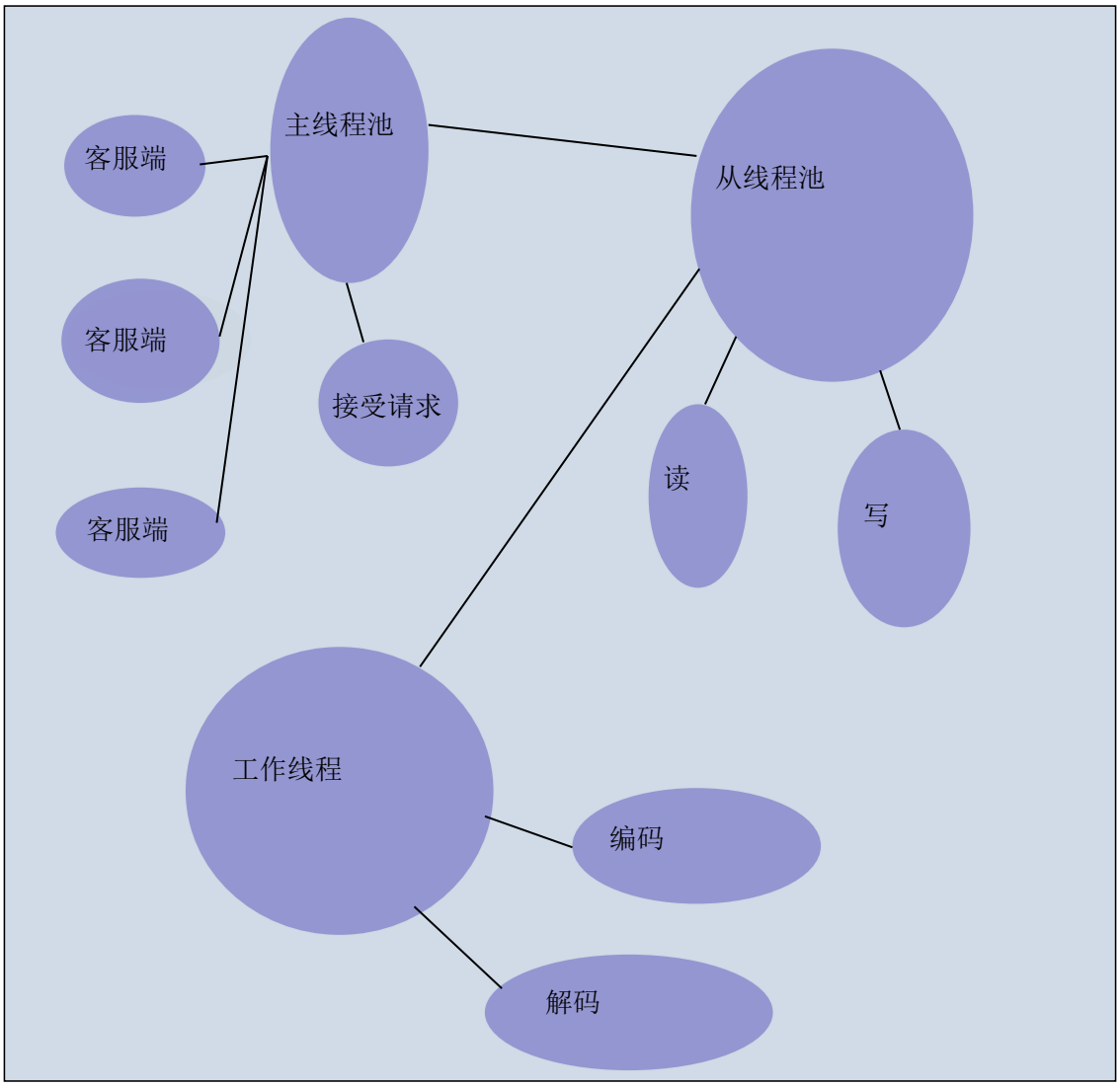


图 4-7 Netty线程模型

7所示，netty主要有三种线程：

1.主线程池主要接受客服端的网络连接请求，当主线程接受到客服端的请求以后，就可以把客服端的请求交个从线程池来处理。这样主线程就可以接受更多的其他客服端的连接请求。使得服务器的吞吐量大大的提高。

2.从线程从主线程得到客服端的连接以后，执行具体的读取操作。如果需要执行其他的事务逻辑，比如说读取磁盘，那么就需要把这个请求发送给工作线程来处理，这样客服端才不会因为从线程的忙碌而堵塞。

3.工作线程主要用来执行时间消耗比较大的任何，比如任何和磁盘文件有关的操作都应该放在工作线程来处理。

通过使用合适的线程模型，服务器具有很高的吞吐量，能接受成千上万的客服端连接。

4.4.3 通信协议设计

每个服务器和客服端通信都要实现自己的通信协议，考虑到mysql使用的广泛性。jsql采用mysql的通信协议。这样就不用自己开发协议了。在MySQL数据库通信过程开始时，服务器会使用TCP监听一个本地socket 端口或本地socket链接。当一个客户端的连接请求到达，就会执行握手和权限验证。如果验证成功，会话开始。客户端发送消息，服务器会以一个适合该发送命令的数据类型的数据集或一条消息进行回复。当客户端发送完成后，会发送一个特殊的命令，告诉服务器己发送，然后会话结束。通信的基本单位是应用程序包。多个指令可以合成一个包；答复可以包含多个包。 MySQL客户端与服务器的交互主要分为两个阶段：握手认证阶段和命令执行阶段。

1.握手认证阶段

握手认证阶段为客户端与服务器建立连接后进行，交互过程如下：

(a) 服务器发送给客户端握手初始化报文。

(b) 客服端回复服务器端登陆认证报文。

(c) 服务器发送给客户端认证结果报文。

2.命令执行阶段

客户端认证成功后，会进入命令执行阶段，交互过程如下：

(a) 客户端发送服务器执行命令报文。

(b) 服务器发送给客服端命令执行结果。

MySQL客户端与服务器之间的完整交互过程如图4-8所示。

每个报文分为报文头和报文数据两部分，其中报文头占用固定的4个字节，报文数据长度由报文头中的长度字段决定。报文头后面三个字节用于标记当前请求报文的实际数据长度值，以字节为单位，最大值为0xFFFFFF，即接近16 MB大小(比16MB少1个字节)。除了报文长度还有报文序列号，在一次完整的请求和响应交互过程中，用于保证报文顺序的正确，每次客户端发起请求时，序号值都会从0开始计算。每个报文后面的字节就是报文数据，报文数据用于存放请求的内容及响应的数据，长度由报文头中的长度值决定。

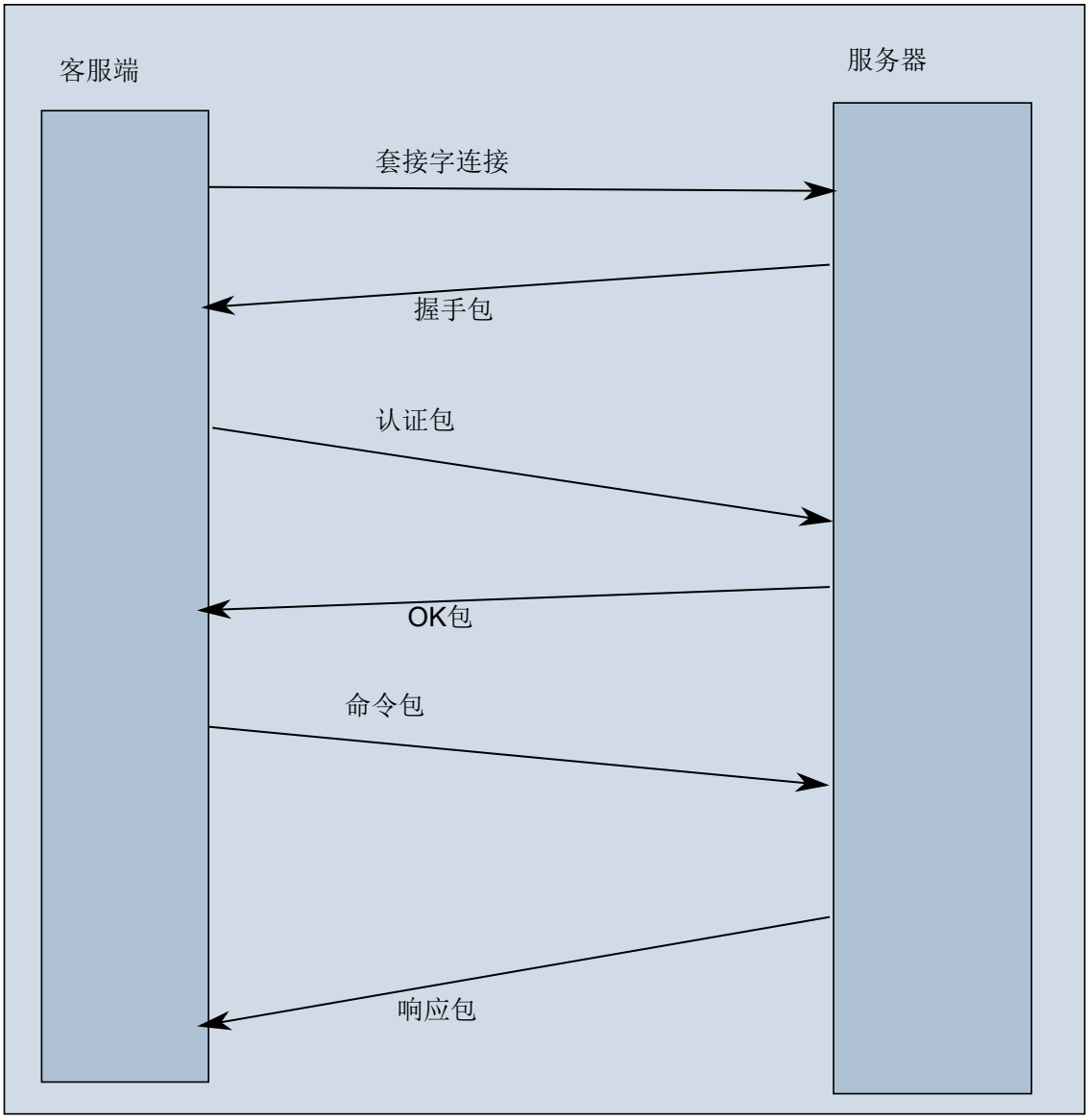


图 4-8 协议交互流程图

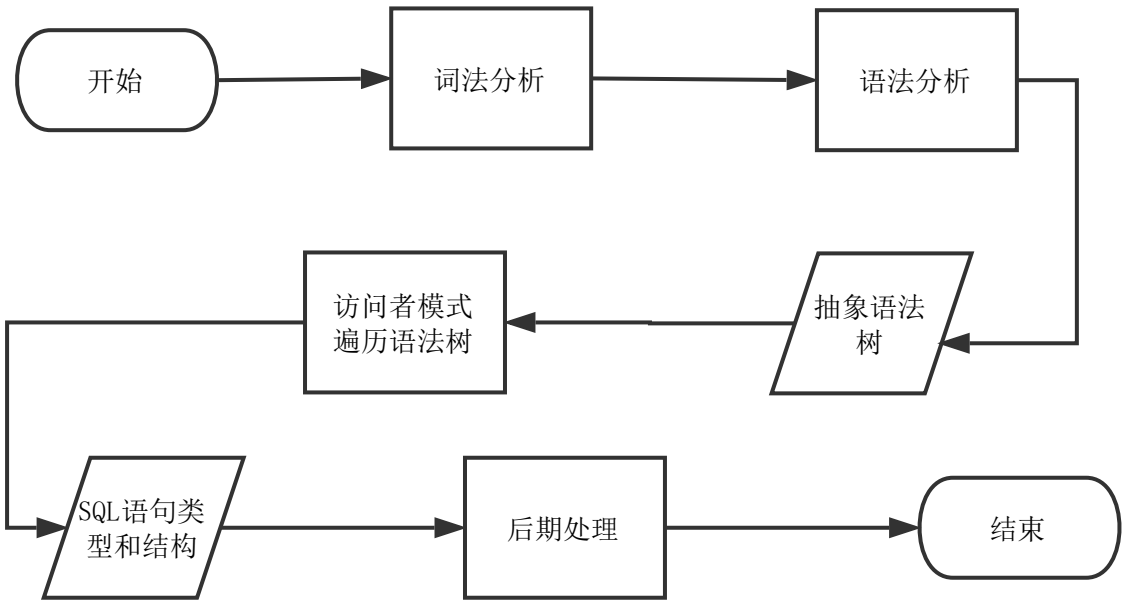


图 4-9 SQL解析流程图

4.4.4 SQL引擎设计

本文基于成熟的开源框架Druid 来实现自己的解析模块，解析流程如图4-9所示。 SQL引擎层从协议层解析到语句以后，就要对这个语句进行解析，才能交给下面的存储引擎处理具体的请求。解析模块首先对语句进行词法分析和语法分析，得到抽象语法树，然后用访问者模式去遍历抽象语法树，得到我们需要的数据结构。这个结构被封装成类，每种语句都有不同的类对应。得到语句类型以后，我们就可以对他进行具体的分析和处理。最后交个存储引擎层。

4.4.5 存储引擎设计

存储引擎是存储系统的发动机，直接决定了存储系统能够提供的性能和功能。存储系统的基本功能包括：增、删、读、改，其中，读取操作又分为随机读取和顺序扫描。每种存储引擎底层都基于一种数据结构。比如常用的哈希表结构和B+树结构。本系统采用了一种NoSQL数据库引擎，这样可以结合关系数据库和非关系型数据库的优点。在OrientDB这个存储引擎基础之上，封装成关系型的操作。完成关系型表和NoSQL数据库的转化。对上一层来说，我们提供了关系型存储引擎的接口调用。在存储引擎的底层，我们调用OrientDB的接口来完成具体的功能。这样能结合关系型操作的便利和非关系型数据库的优点。

4.5 集群架构详细设计

要实现集群功能，肯定要用到网络功能，还要考虑各种不可靠的因素。为了系统的稳定性和可靠性，本文用到了一个开源的分布式的开发框架hazelcast，基于hazelcast的应用程序的结构图如图4-10所示。 分布式数据库节点和分布式管理

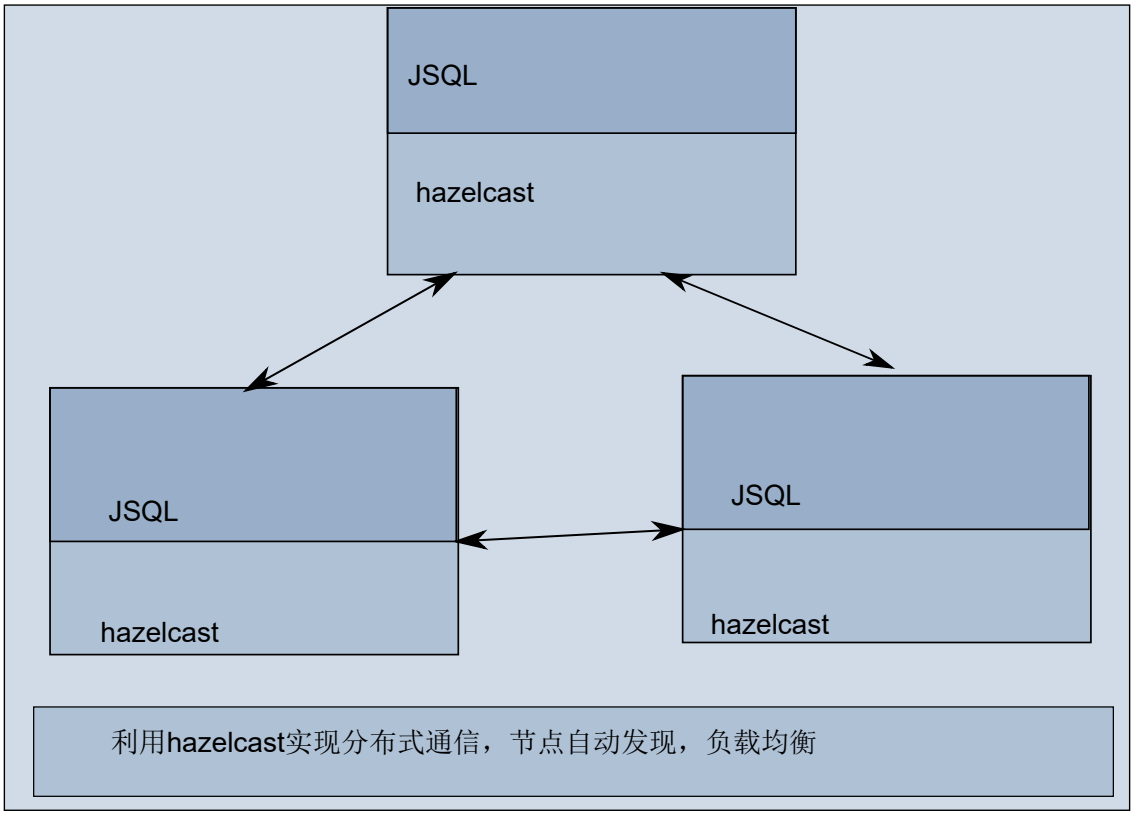


图 4-10 集群架构图

节点都嵌入了Hazelcast模块，这样分布式管理节点就能很容易的得到每个数据库节点的详细信息，直接调用框架的接口就可以得到这些信息。 hazelcast分布式原理如图4-11所示。 在分布式数据库集群中，不用设计主节点，集群会自己发现和设置主节点。在一个集群中，最先启动的计算机就是主节点，主节点拥有其他所有节点的信息。当其他节点启动的时候，通过多播技术，加入多播网络。主节点发现以后，就会把自己的信息和集群所有其他计算机的信息发送给这个新的计算机，同时更新集群的元数据。当第一个主节点关掉的时候，再一次启动的计算机就自动成为主节点。这样就不需要用户频繁的设置和更新元数据。分布式数据库集群在增加和删除节点的时候，不需要任何人为的干预。

4.6 审计模块详细设计

审计模块部署在管理计算机和分布式管理节点上面。管理员通过管理计算

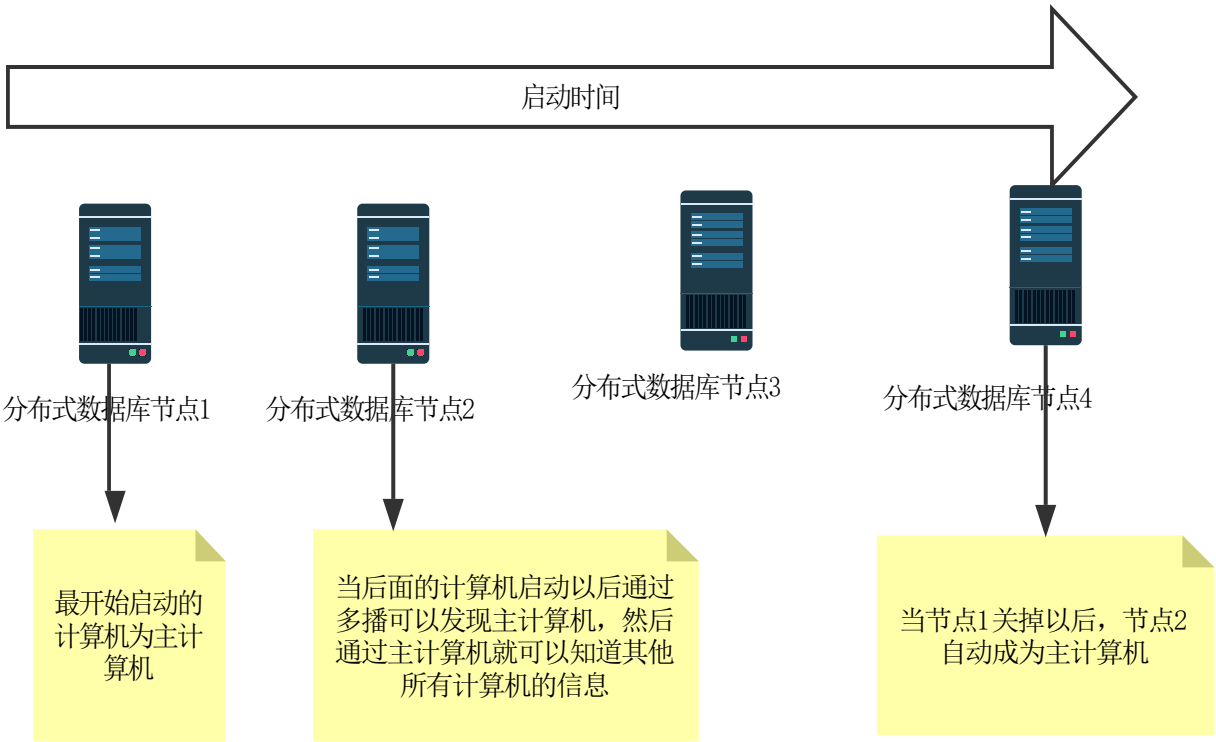


图 4-11 集群架构图

机连接分布式管理节点就可以管理所有的数据。监控所有数据的更改情况。图4-12显示了审计模块的实现结构图。从上大下一共可以分为下面4个模块：

1.grafana可视化和报警模块，主要用到了开源的图形框架，来显示底层的审计数据，同时也可以让用户设定预警数据，这样以后就可以在一定的情况下向用户报警。

2.elasticsearch模块用来存储监控数据，监控数据为了登陆日志和sql执行日志我们需要更改它的源代码才能用来作为审计数据库，因为审计数据库不能随意的更改和删除，只能查询和增加。

3.JSQL接口层，主要是为上面和下面的各层提供连接功能，为上面层提供数据接口，为下面层提供显示接口，使得更加容易使用。

4.本地日志层，主要是在文件系统中存储所有需要的日志记录，这样可以随时和审计数据库的数据来对比。审计数据的安全和一致性。

4.7 本章小结

任何系统只有经过仔细的系统设计，在编码阶段才能提前减少不必要的错误。本章从总体上设计了系统的部署模式和模块图，对每个模块进行了详细了的模块设计，便于后面的具体实现。

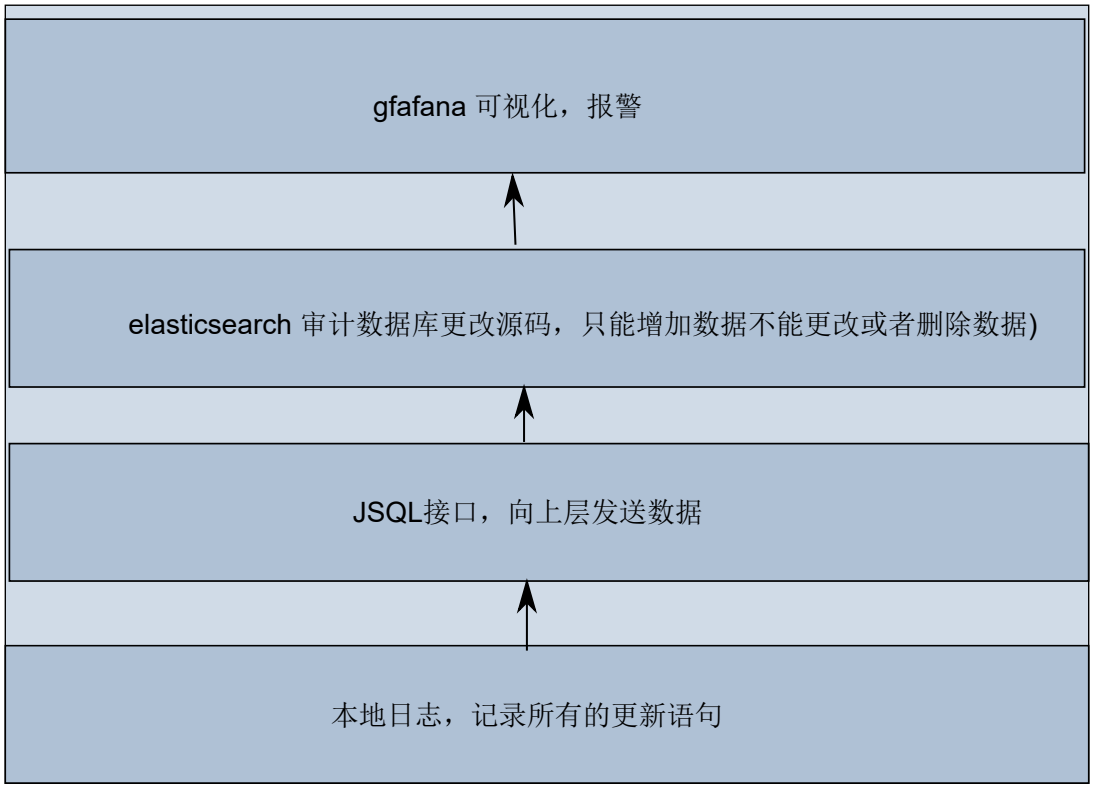

gfafana 可视化，报警
elasticsearch 审计数据库更改源码，只能增加数据不能更改或者删除数据)
JSQL接口，向上层发送数据
本地日志，记录所有的更新语句

图 4-12 监控模块

第五章 系统实现

按照前一章的设计，本章给出每个功能模块的具体实现。其中包括对关键代码的分析。

5.1 代码规范和总体结构

本数据库系统采用和JAVA语言类似的 Kotlin语言开发。JAVA中的代码以包为组织单位，这样组织代码的好处是逻辑结构分明。表5-1描述了本系统所有的包和每个包的详细功能。 图5-2给出了config包下面每个类的具体作用，这个包主要是

表 5-1 本系统所有包和各个包的作用

包	作用
io.jsql	启动或者关闭服务器
io.jsql.audit	审计监控功能
io.jsql.cache	数据库缓存
io.jsql.config	配置功能，读取外部配置文件
io.jsql.hazelcast	集群功能实现
io.jsql.mysql	实现mysql通信协议
io.jsql.netty	Netty实现高性能网络前端
io.jsql.orientstorage	嵌入式存储引擎模块
io.jsql.shutdown	关闭数据库功能
io.jsql.springaop	面向切面，实现日志等功能
io.jsql.sql	实现SQL解析模块
io.jsql.storage	存储引擎接口
io.jsql.test	测试包
io.jsql.util	所有常用的工具类

表 5-2 config包下面每个类的作用

类	作用
Capabilities	处理能力标识定义
ErrorCode	所有的错误代码
Fields	字段类型及标识定义
Isolations	事务隔离级别定义
Versions	本系统的版本号，通讯包

让用户可以配置数据库的各个方面，比如配置数据库的端口号，配置最大的连接数等等。后面讲详细解释其他包功能具体的实现。

5.2 客服端功能实现

客服端主要实现负载均衡功能，利于JDBC连接后端的分布式数据库，利用网络连接到分布式管理节点得到分布式数据库节点的IP地址。在分布式管理节点为客服端返回正确数据库地址以后，客服端就可以通过JDBC连接数据库节点。客服端连接功能流程图如图5-1所示。

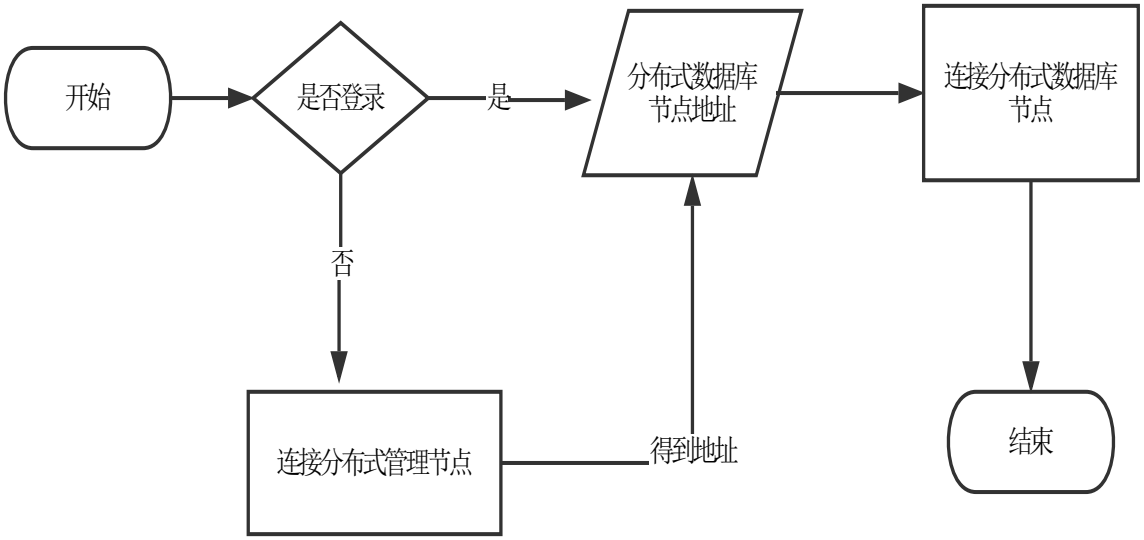


图 5-1 客服端连接功能流程图

首先，判断客服端有没有已经登录的会话，如果已经有已经登录的会话，那么就可以直接和对应的分布式数据库节点连接。发送数据库命令并且得到返回结果。当客服端没有连接会话的时候，客服端就首先连接分布式管理节点，分布式管理为客服端返回合适的数据节点地址，然后客服端利用这个地址再连接数据库服务器。通过这样一个步骤，分布式管理节点能利用合适的负载均衡算法为客服端选择合适的数据库节点，有利于服务器资源的均衡利用。

5.3 分布式管理节点实现

管理节点实现了负载均衡功能，实现了分布式数据库节点的均衡使用。同时也为管理员提供管理和监控功能。本节主要对分布式管理节点的负载均衡的实现进行说明。分布式管理节点负载均衡算法示意图如图5-2所示。在接收到客服端的连接请求以后，分布式管理节点首先需要删除当前客服端之前的所有会话信息，同时更新响应的分布式节点的元数据。然后管理节点利用管理员设置的负载均衡算法和元数据节点信息计算出选择的节点地址。最后通过网络接口返回给客服端地址。

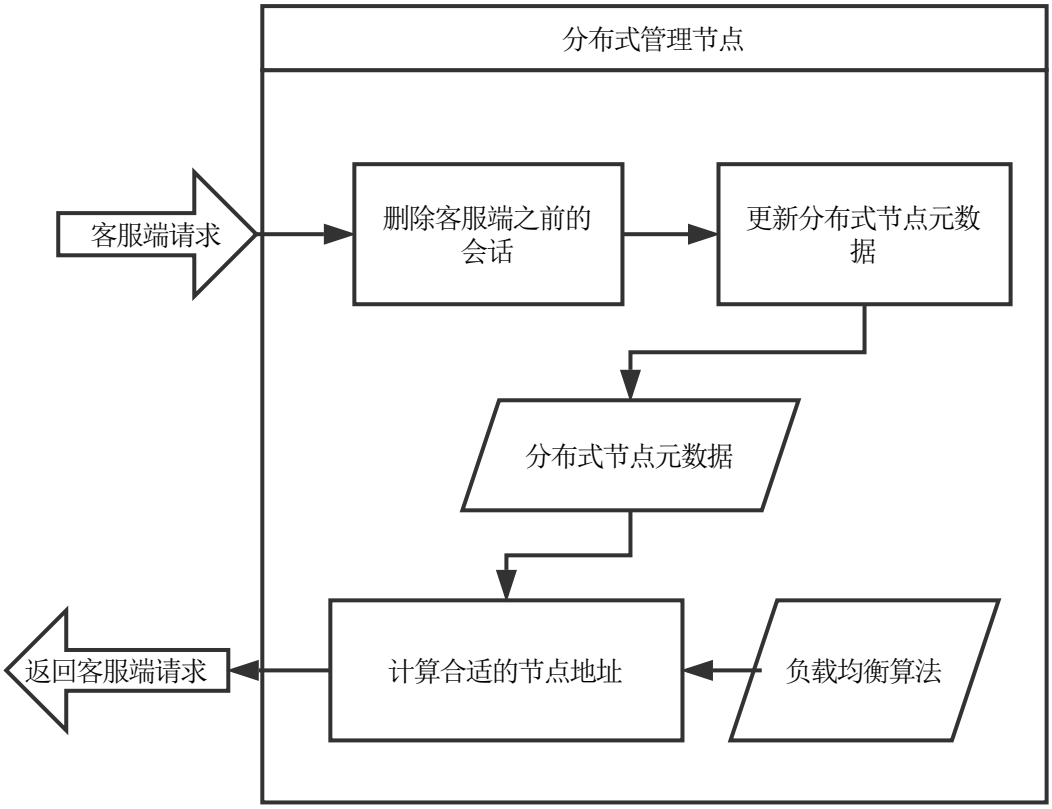


图 5-2 分布式管理节点负载均衡算法示意图

5.4 数据库系统实现

SQL包下面的类主要是作为数据库管理类，让用户启动数据库服务器或者关闭数据库服务器。表5-3描述了SQL包下每个类的作用。

表 5-3 SQL包的每个类的作用

类	功能
ShutdownMain	关闭本机服务器
SpringMain	启动类，通过spring boot启动。 ioc，aop功能

数据库系统主要包括网络模块，SQL解析模块和存储引擎模块，下面分别描述每个模块具体的实现。

5.4.1 网络模块

网络模块主要是用了Netty来实现的，所有和网络有关的代码都在netty包下面。表5-4描述了每个类的功能。 下面的代码是netty包下面主要的代码，用来启

表 5-4 网络模块中各个类的作用

类	作用
ByteToMysqlDecoder	接受客服端的字节消息，转化为mysql的消息格式
ByteToMysqlPacket	包字节格式转化为包对象
MysqlPacketHander	解码器，处理接受到的包对象
NettyServer	前端线程池，接受客服端的请求

动网络服务器，接受用户的连接请求。

```
/**
 *
 * 服务器
 */
@Service
class NettyServer {
    fun start() {
        val group = DefaultEventExecutorGroup(8)
        val bossGroup = NioEventLoopGroup(1)
        val workerGroup = NioEventLoopGroup()
        try {
            val b = ServerBootstrap()
            b.group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel::class.java)
```

```
15                      .option(ChannelOption.SO_BACKLOG, 100)
16                      .childHandler(object : ChannelInitializer<SocketChannel>(){
17                          @Throws(Exception::class)
18                          public override fun initChannel(ch: SocketChannel)
{
19                              val p = ch.pipeline()
20                               //p.addLast(new LoggingHandler(LogLevel.INFO));
21                               p.addLast("idle", IdleStateHandler(10,
5, 0))
22                               p.addLast("decoder", byteToMysqlDecoder)
23                               p.addLast("packet", bytetomysql)
24                               p.addLast(group, "hander", mysqlPacketHander)
25                           }
26                      })
27              val f = b.bind(PORT).sync()
28              // Wait until the server socket is closed.
29              logger.info("server start  complete...................
")
30              Minformation_schama.init_if_notexits()
31              if (config.distributed) {
32                  myHazelcast.inits()
33              }
34              f.channel().closeFuture().sync()
35          } finally {
36              // Shut down all event loops to terminate all threads.
37              bossGroup.shutdownGracefully()
38              workerGroup.shutdownGracefully()
39          }
40      }
41 }
```

除了实现网络模块以外，我们还要实现通信协议，系统采用了和mysql一样的通信协议，在mysql包下面实现了mysql的通信协议。表5-5给出了实现通信协议所

用到的所有的类。

表 5-5 通信协议实现所用到的类

文件	类
BindValue	class BindValue
BindValueUtil	object BindValueUtil
BufferUtil	object BufferUtil
ByteUtil	object ByteUtil
CharsetUtil	object CharsetUtil
MBufferUtil	object MBufferUtil
MySQLMessage	class MySQLMessage
PacketUtil	object PacketUtil
PreparedStatement	class PreparedStatement
SecurityUtil	object SecurityUtil
StreamUtil	object StreamUtil

网络服务器接受到客服端的连接以后，就要解析mysql的通信协议，包每一个包封装到具体的对象里面，表5-6描述了所有的mysql通信协议的封装对象。 mysql里面有很多的包，每个包都需要一个类来实现，下面是命令包的源代码，其他包的实现大体相同。

```
01 class CommandPacket : MySQLPacket() {
02     var command: Byte = 0
03     var arg: ByteArray? = null
04     override fun read(data: ByteArray) {
05         val mm = MySQLMessage(data)
06         packetLength = mm.readUB3()
07         packetId = mm.read()
08         command = mm.read()
09         arg = mm.readBytes()
10     }
11     @Throws(IOException::class)
12     fun write(out: OutputStream) {
13         StreamUtil.writeUB3(out, calcPacketSize())
14         StreamUtil.write(out, packetId)
15         StreamUtil.write(out, command)
16         out.write(arg)
17     }
18 }
```

表 5-6 mysql所有协议包的封装对象

文件	解释
AuthPacket	From client to server during initial handshake.
BinaryPacket	class BinaryPacket : MySQLPacket
CommandPacket	From client to server , the client wants the server to do something
EOFPacket	the EOF packet contains a warning
EmptyPacket	class EmptyPacket : MySQLPacket
ErrorPacket	From server to client in response to command, if error.
ExecutePacket	class ExecutePacket : MySQLPacket
FieldPacket	part of Result Set Packets. One for each column in the result set.
HandshakePacket	From server to client during initial handshake
HandshakeV10Packet	Connection Phase The Connection Phase performs these tasks
HeartbeatPacket	From client to server when the client do heartbeat between jsql cluster
LongDataPacket	class LongDataPacket : MySQLPacket
MySQLPacket	abstract class MySQLPacket
OkPacket	From server to client ,if no error and no result set
PingPacket	class PingPacket : MySQLPacket
PreparedOkPacket	class PreparedOkPacket : MySQLPacket
QuitPacket	class QuitPacket : MySQLPacket
Reply323Packet	class Reply323Packet : MySQLPacket
RequestFilePacket	load data local infile
ResetPacket	class ResetPacket : MySQLPacket
ResultSetHeaderPacke	The Result Set Header Packet is the first of several packets
RowDataPacket	From server to client. One packet for each row in the result set

协议主要有2个阶段，一个是认证阶段，一个是命令阶段，认证阶段就是接受客服端的请求，然后检查用户的认证信息，比如用户名和密码，下面代码主要用来实现用户认证的功能。

```
/**
 * 前端认证处理器
 * 处理auth包
 */
@Component
open class MysqlAuthHander : MysqlPacketHander {
    private fun handle0(auth: AuthPacket, source: OConnection) {
        source.schema = auth.database
        // check password
        if (!checkPassword(auth.password!!, auth.user!!)) {
            if (config.audit) {
            LoginLog(auth.user ?: "null", source.host, false).sendesServer()
            }
            failure(ErrorCode.ER_ACCESS_DENIED_ERROR,
      "Access denied for user '" + auth.user + "',
      because password is error ", source)
        } else {
            success(auth, source)
        }
    }
}
```

认证成功以后就是命令阶段，服务器接受客服端的命令，然后处理，下面是命令处理的具体代码。

```
/**
 * 前端命令处理器
 * 处理命令包
 */
@Component
```

```
06 class MysqlCommandHandler : MysqlPacketHander {
07     private fun handle0(data: CommandPacket, source: OConnection) {
08         logger.debug(data.toString())
09         logger.info("command info")
10         when (data.command) {
11             MySQLPacket.COM_INIT_DB -> initDB(data, source)
12             MySQLPacket.COM_QUERY -> query(data, source)
13             MySQLPacket.COM_PING -> ping(source)
14             MySQLPacket.COM_QUIT -> close("quit cmd", source)
15             MySQLPacket.COM_PROCESS_KILL -> kill(data, source)
16             MySQLPacket.COM_STMT_PREPARE -> stmtPrepare(data, source)
17             MySQLPacket.COM_STMT_SEND_LONG_DATA->stmtSendLongData(data,source)
18             MySQLPacket.COM_STMT_RESET -> stmtReset(data, source)
19             MySQLPacket.COM_STMT_EXECUTE -> stmtExecute(data, source)
20             MySQLPacket.COM_STMT_CLOSE -> stmtClose(data, source)
21             MySQLPacket.COM_HEARTBEAT -> heartbeat(data, source)
22             else -> source.writeErrMessage(ErrorCode.ER_UNKNOWN_COM_ERROR,
23                     "Unknown command")
24         }
25     }
26     private fun query(data: CommandPacket, source: OConnection) {
27         val mm = MySQLMessage(data.arg!!)
28         mm.position(0)
29         try {
30             val sql = mm.readString(source.charset)
31             source.sqlHander.handle(sql!!, source)
32         } catch (e: UnsupportedEncodingException) {
33             source.writeErrMessage(ErrorCode.ER_UNKNOWN_CHARACTER_SET,
"Unknown charset '" +
34       source.charset + "'")
35             e.printStackTrace()
36         }
37     }
```

```
38 }
```

5.4.2 SQL解析模块

表 5-7 SQL前端连接模块相关的类

类	作用
MysqlSQLhander	接受前端的语句，处理用户的请求
OConnection	前端连接对象，一个客服端一个连接
ONullConnection	继承连接对象，但是不处理具体任务，用在分布式中
OconnectionPool	管理前端的连接，作为一个连接池对象

sql解析主要用到了druid开源的框架，解析sql以后就要做具体的数据处理，下面的代码主要用处理用户的sql语句。

```
01 /**
02  * 处理sql语句的入口
03  */
04 @Component
05 class MysqlSQLhander : SQLHander {
06     @Autowired
07     lateinit private var allHanders: AllHanders
08   //所有的sql处理器容器
09     @PostConstruct
10     override fun handle(sql: String, c: OConnection){
11   //处理正常的sql语句，前端连接
12         if (config.audit) {
13             SqlLog(sql, c.user ?: "null", c.host).sentoELServer()
14         }
15         logger.info(sql)
16         if (logger.isDebugEnabled) {
17             logger.debug(sql)
18         }
19         val sqlStatement: SQLStatement
20         try {
21             sqlStatement = sql.tosql()
22             val hander = allHanders.handerMap[sqlStatement.javaClass]
```

```
23              if (hander != null) {
24                  hander.handle(sqlStatement, c)
25              } else {
26                  sqlStatement.accept(MSQLvisitor(c))
27              }
28              if (isupdatesql(sql)) {
29                  if (config.distributed) {
30                      myHazelcast.exeSql(sql, if (c.schema == null) ""
31            else c.schema!!)
32                  } else {
33                      myHazelcast.exesqlLocal(sql, if (c.schema == null)
34            "" else c.schema!!)
35                  }
36              }
37              return
38          } catch (e: Exception) {
39      //如果不是合法的mysql语句，就报错
40          //druid支持的语句就用上面的方法语句处理，如果不支持，就会有
异常，
41        //就自己写代码解析sql语句，处理。
42           //下面是drop event语句的例子，这个例子druid不支持，所以自
己写
43              handleotherStatement(sql, c, e)
44          }
45      }
46      fun handle(sql: SqlUpdateLog, c: OConnection) {
47    //处理来自其他服务器的sql语句，同步，不需要前端连接
48          logger.info(sql.toString())
49          if (logger.isDebugEnabled) {
50              logger.debug(sql.toString())
51          }
52          val sqlStatement: SQLStatement
53          try {
```

```
54              val parser = MySqlStatementParser(sql.sql)
55              sqlStatement = parser.parseStatement()
56              val hander = allHanders.handerMap[sqlStatement.javaClass]
57              if (hander != null) {
58                  hander.handle(sqlStatement, c)
59              } else {
60                  sqlStatement.accept(MSQLvisitor(c))
61              }
62              return
63          }
64      }
65  }
```

```
01  /**
02   * 所有的sql语句处理器必须是这个类的子类.
03   * 比如Mupdate。Mselect
04   */
05  abstract class SqlStatementHander {
06
07        //返回值只有4种可能，不然报错!!!
08        //一种是long类型， 一种是MyResultSet 一种是null ,
09      // 一种是string表示错误的消息
10        //返回其他对面都是错误的
11      fun handle(sqlStatement: SQLStatement, connection: OConnection)
{
12          try {
13              val result = handle0(sqlStatement, connection)
14              when (result) {
15                  null -> connection.writeok()
16                  is MyResultSet -> onsuccess(result.data,
17          result.columns, connection)
18                  is Long -> onsuccess(result, connection)
19                  is String -> connection.writeErrMessage(result)
```

```
20                  else -> connection.writeok()
21              }
22          } catch (e: Exception) {
23              e.printStackTrace()
24              onerror(e, connection)
25          }
26
27      }
28  }
```

mysql当中的sql语句有很多种，每一种语句都要做不同的处理，5-8描述了sql模块下实现的各种类型的语句。

表 5-8 SQL模块下实现的各种类型的语句

类	作用
io.jsql.sql.handler.adminstatement	处理管理语句
io.jsql.sql.handler.componed_statement	处理组合语句
io.jsql.sql.handler.data_define	处理数据定义语句
io.jsql.sql.handler.data_mannipulation	处理数据操作数据
io.jsql.sql.handler.preparestatement	处理准备语句
io.jsql.sql.handler.replication_statement	处理复制语句
io.jsql.sql.handler.tx_and_lock	处理事务和锁控制语句
io.jsql.sql.handler.utilstatement	处理其他工具语句

每一种类型语句下面又回有很多种具体语句的实现，表5-9给出了数据操纵类型语句实现的所有相关的类和响应的功能。

表 5-9 数据操纵数据实现所相关的类

文件	作用
MSelectHandler	处理查找语句
Mcall	处理函数调用语句
Mdelete	处理删除语句
Mdo	处理mysql中的do语句
Mhandler	所有类的基类
Minsert	处理插入语句
MloaddataINfile	处理导入文件的语句
Mloadxml	处理导入xml的语句
Mrepelace	处理解释语句
MselectVariables	处理查找变量的语句
Msubquery	处理子查询
Mupdate	处理更新语句

5.4.3 存储引擎模块

存储引擎有关的功能主要在storage包下面实现，表5-10给出了 storage包下面各种文件的功能。 其中DB类作为底层存储引擎的接口，他的接口如表5-11所示。

表 5-10 storage包下面各种文件的作用

文件	功能
DB	接口，规定所有的存储引擎应该实现的和数据库有关的功能
Table	接口，规定所有的存储引擎应该实现的和表有关的功能

和表有关的接口如表5-12所示。

表 5-11 数据库存储引擎的函数接口

接口	函数签名
close	abstract fun close(): Unit
createdbAsyn	abstract fun createdbAsyn(dbname: String): Unit
createdbSyn	abstract fun createdbSyn(dbname: String): Unit
deletedbAyn	abstract fun deletedbAyn(dbname: String): Unit
deletedbSyn	abstract fun deletedbSyn(dbname: String): Unit
exe	abstract fun exe(sql: String, db: String): Unit
exesqlNoResultAsyn	abstract fun exesqlNoResultAsyn(sql: String, dbname: String): Unit
exesqlforResult	abstract fun exesqlforResult(sql: String, dbname: String):Oresult
getallDBs	abstract fun getallDBs(): List<String>
getdb	abstract fun getdb(dbname: String): ODatabaseDocument
query	abstract fun query(sqlquery: String, dbname: String): Stream<OElement>

表 5-12 数据库引擎和表有关功能的接口

函数	函数签名
createtableSyn	abstract fun createtableSyn(dbname: String, createTableStatement
droptableSyn	abstract fun droptableSyn(dbname: String, table: String): Unit
getalltable	abstract fun getalltable(dbname: String): List<String>
gettableclass	abstract fun gettableclass(tablename: String, db: String): OClass
selectSyn	abstract fun selectSyn(oClass: OClass, dbname: String): Stream<OElement>

前面说过，本系统实现的存储引擎用的是哈希树这种数据结构。哈希树的实现主要有2个类，一个类实现哈希树的节点，一个类实现哈希树给外部模块提供调用接口。下面是其中实现的关键代码。

```
01 public class HtreeNode {
02     public int high;//root is 0
03     public int hashtable_size;
04     public HtreeNode[] childs;
```

```
05     public boolean hasV;
06     public String key;
07     public Object values;
08     public HtreeNode(int high, String key, Object values) {
09         this.high = high;
10         hashtable_size = HashCodes.codes[high];
11         this.key = key;
12         this.values = values;
13         childs = new HtreeNode[hashtable_size];
14         hasV = key != null;
15     }
16 }
```

```
01 public class HTreeMap implements Map<String, Object> ,Serializable{
02     @Override
03     public Object get(Object key) {
04         if (key == null) {
05             throw new NullPointerException("key not null");
06         }
07         int hashcode = Math.abs(key.hashCode());
08         HtreeNode htreeNode = root;
09         while (htreeNode != null) {
10             LOGGER.debug("this node is:" + htreeNode.key);
11             if (htreeNode.hasV && key.equals(htreeNode.key)) {
12                 LOGGER.debug("find " + key);
13                 return htreeNode.values;
14             }
15             int hashindex = hashcode % htreeNode.hashtable_size;
16             htreeNode = htreeNode.childs[hashindex];
17         }
18         return null;
19     }
20     @Override
```

```
21      public Object put(String key, Object value) {
22          if (key == null || value == null) {
23              throw new NullPointerException("key not null");
24          }
25          int hashcode = Math.abs(key.hashCode());
26          HtreeNode htreeNode = root;
27          HtreeNode pre = null;
28          while (htreeNode != null) {
29              if (htreeNode.hasV && key.equals(htreeNode.key)) {
30                  Object stemp = htreeNode.values;
31                  htreeNode.values = value;
32                  return stemp;
33              } else if (!htreeNode.hasV) {
34                  htreeNode.hasV = true;
35                  htreeNode.values = value;
36                  htreeNode.key = key;
37                  return null;
38              }
39              pre = htreeNode;
40              int hashindex = hashcode % htreeNode.hashtable_size;
41              htreeNode = htreeNode.childs[hashindex];
42          }
43          int hashindex = hashcode % pre.hashtable_size;
44          pre.childs[hashindex] = new HtreeNode(pre.high + 1, key, value);
45          allnodes.add(pre.childs[hashindex]);
46              return null;
47      }
48      @Override
49      public Object remove(Object key) {
50          if (key == null) {
51              throw new NullPointerException("key not null");
52          }
53          int hashcode = Math.abs(key.hashCode());
```

```
54          HtreeNode htreeNode = root;
55          while (htreeNode != null) {
56              if (htreeNode.hasV && key.equals(htreeNode.key)) {
57                  htreeNode.hasV = false;
58                  return htreeNode.values;
59              }
60              int hashindex = hashcode % htreeNode.hashtable_size;
61              htreeNode = htreeNode.childs[hashindex];
62          }
63          return null;
64      }
65  }
```

除了实现数据结构以外还要实现存储引擎，也就是把数据存储在文件系统当中，其中主要用到内存映射文件，下面是其中的关键代码。

```
01 /**
02  * The type M storage.
03  * 文件前面2m保留做头信息1024*1024*2/4096=512页面
04  */
05 class MStorage {
06     public void write(long pageNumber, ByteBuffer data) throws IOException
{
07         FileChannel f = getChannel(pageNumber);
08         int offsetInFile = (int) ((Math.abs(pageNumber) %
09     Pagesize.max_page_number) * Pagesize.page_size);
10         MappedByteBuffer b = buffers.get(f);
11         if (b.limit() <= offsetInFile) {
12             b = addfilesize(f, offsetInFile, b);
13         }
14         //write into buffer
15         b.position(offsetInFile);
16         data.rewind();
17         b.put(data);
```

```
18      }
19      public ByteBuffer read(long pageNumber) throws IOException {
20          FileChannel f = getChannel(pageNumber);
21          int offsetInFile = (int) ((Math.abs(pageNumber) %
22      Pagesize.max_page_number) * Pagesize.page_size);
23          MappedByteBuffer b = buffers.get(f);
24          if (b == null) { //not mapped yet
25              b = f.map(FileChannel.MapMode.READ_WRITE, 0, f.size());
26          }
27          //增加文件大小，64m为单位
28          if (b.limit() <= offsetInFile) {
29              b = addfilesize(f, offsetInFile, b);
30          }
31          b.position(offsetInFile);
32          ByteBuffer ret = b.slice();
33          ret.limit(Pagesize.page_size);
34          if (!transactionsDisabled || readonly) {
35              // changes written into buffer will be directly written
into file
36              // so we need to protect buffer from modifications
37              ret = ret.asReadOnlyBuffer();
38          }
39          return ret;
40      }
41  }
```

```
01  /**
02   * 和内存指针差不多，new 后得到地址,这里地址是page index
03   * 每个页面开始分别是type，记录大小，数据，页面后2个字节用来连接每
个页面
04   * 一个页面pagesize-2-4-4
05   */
06  public class DiscIO implements MdiscIO {
```

```
07      @Override
08      public int write(Object o) {
09          byte[] bytes = ObjectSeriaer.getbytes(o);
10          int[] pages = pagemanager.getfreepanages(bytes.length);
11          if (pages.length == 1) {
12              try {
13                  ByteBuffer buffer = storage.read(pages[0]);
14                  if (o instanceof DHtree)
15                      buffer.putShort(Pagesize.pagehead_tree);
16                  else if (o instanceof DHtreeNode) {
17                      buffer.putShort(Pagesize.pagehead_node);
18                  } else {
19                      buffer.putShort(Pagesize.pagehead_other);
20                  }
21                  buffer.putInt(bytes.length);
22                  buffer.put(bytes);
23                  storage.write(pages[0], buffer);
24                  ObjectMap.putorupdate(o, pages[0]);
25                  return pages[0];
26              } catch (IOException e) {
27                  e.printStackTrace();
28              }
29          } else {
30              return writemorepage(bytes, pages, o);
31          }
32          return 0;
33      }
34      @Override
35      public int update(Object o, int recid) {
36          if (!ObjectMap.map.containsKey(recid)) {
37              return -1;
38          }
39          byte[] bytes = ObjectSeriaer.getbytes(o);
```

```
40          if (bytes.length <= Pagesize.page_size_for_content) {
41              try {
42                  ByteBuffer buffer = storage.read(recid);
43                  if (o instanceof DHtree)
44                      buffer.putShort(Pagesize.pagehead_tree);
45                  else if (o instanceof DHtreeNode) {
46                      buffer.putShort(Pagesize.pagehead_node);
47                  } else {
48                      buffer.putShort(Pagesize.pagehead_other);
49                  }
50                  buffer.putInt(bytes.length);
51                  buffer.put(bytes);
52                  storage.write(recid, buffer);
53                  ObjectMap.putorupdate(o, recid);
54                  return recid;
55              } catch (IOException e) {
56                  e.printStackTrace();
57                  return -1;
58              }
59          } else {
60              updatemoredata(bytes, recid, o);
61              return recid;
62          }
63      }
64 }
```

5.5 集群架构的实现

集群功能主要是利用了开源的hazelcast框架来实现，其中的功能全在hazelcast包下面实现，表5-13给出了该包下面每个类的具体的作用。 其中最关键的代码如下。

```
01 //sql队列 复制队列命令队列。2个锁。发布
02 @Component
```

表 5-13 集群模块下面各个类的作用

类	作用
LogFile	本地文件系统中存储日志文件
MyHazelcast	利用Hazelcast的分布式数据结构实现集群，比如分布式队列和分布式锁
ReplicationCMD	分布式对象，在不同集群之间传输
SqlUpdateLog	当前系统LSN最大值，新的事务日志LSN将在此基础上生成

```
03 class MyHazelcast : ItemListener<SqlUpdateLog> {
04     private fun exeSqlforReplication() {
05         Collections.sort(localqueneReplication)
06         var log: SqlUpdateLog? = localqueneReplication.poll()
07         var lastlsn: Long = 0
08         while (log != null) {
09             locals_maxlsn = log.LSN
10             logger.info("exeSqlforReplication exe sql " + log)
11             oNullConnection!!.schema = log.db
12             sqLhander.handle(log, oNullConnection)
13             logFile!!.write(log)
14             lastlsn = log.LSN
15             log = localqueneReplication.poll()
16         }
17         isreplicating = false
18         log = localquene.poll()
19         val remotel = iAtomic_remote_lsn!!.get()
20         while (log != null) {
21             locals_maxlsn = log.LSN
22             logger.info("exe  Sql : " + log)
23             oNullConnection!!.schema = log.db
24             sqLhander.handle(log, oNullConnection)
25             logFile!!.write(log)
26             lastlsn = log.LSN
27             if (lastlsn > remotel) {
28                 remotequene!!.offer(log)
```

```
29              }
30              log = localqueneReplication.poll()
31          }
32          if (lastlsn > remotel) {
33              iAtomic_remote_lsn!!.set(lastlsn)
34          }
35      }
36      /**
37       *  本机发出的sql语句.记录到本地logfile。
38      同时发布到其他服务器*/
39      fun exeSql(sql: String, db: String) {
40          if (isreplicating) {
41              val l = iAtomic_remote_lsn!!.get()
42              val log = SqlUpdateLog(l + 1, sql, db)
43              localquene.addLast(log)
44          } else {
45              locals_maxlsn++
46              val l = iAtomic_remote_lsn!!.addAndGet(1)
47              val log = SqlUpdateLog(l, sql, db)
48              logger.info("exe sql " + log)
49              logFile!!.write(log)
50              remotequene!!.offer(log)
51          }
52      }
53      private fun exesqlforremoteData() {
54          Collections.sort(localquene)
55          var log: SqlUpdateLog? = localquene.poll()
56          var last: Long = 0
57          while (log != null) {
58              locals_maxlsn = log.LSN
59              logger.info("exesqlforremoteData()  " + log)
60              oNullConnection!!.schema = log.db
61              sqLhander.handle(log, oNullConnection)
```

```
62              logFile!!.write(log)
63              last = log.LSN
64              log = localquene.poll()
65          }
66          val l = iAtomic_remote_lsn!!.get()
67          if (last > l) {
68              iAtomic_remote_lsn!!.set(last)
69          }
70
71      }
72
73  }
```

5.6 数据审计模块的实现

审计模块有关的功能全部在audit包下面实现，表5-14给出了audit包下面每个类的具体的作用。

表 5-14 audit包下面每个类的作用

类	作用
LoginLog	简单对象，记录登陆日志
SqlLog	简单对象，记录SQL执行日志
elasticUtil	工具类，包日志记录发送到ELK

5.6.1 审计数据库

审计数据库用Elasticsearch来实现，作为审计数据库，不能让用户随意的更改，所以本系统更改了它的源代码，使得它只能增加数据和查找数据不能更改和删除数据。其中主要存储的对象如图5-14所示。

5.6.2 审计管理器

审计管理功能全部在audit下面实现，主要是存储本地的日志文件，然后发布到审计数据库。提供给前端可视化的接口。

5.6.3 审计可视化模块的实现

审计可视化模块的实现用到了grafana，主要用来监控ELK中的数据。

5.7 本章小结

本文前面一章设计了本系统的架构图和各个模块的详细功能，本章给出每个功能模块的具体实现。。

第六章 系统测试

6.1 测试环境

本文的测试环境为个人的计算机。它的硬件和软件参与如下

1.硬件参数：

(a) CPU：intel(R) Xeon(R) E5620

(b) 内存：8G

(c) 磁盘：500G

(d) 网络：100Mbit/s

2.软件参数：

(a) 操作系统：windows10

(b) java版本：8.0

6.2 功能测试

6.2.1 数据库功能测试

功能测试设计的细节非常多，项目繁杂。这里将给出几个关键的功能测试的条件、目的、步骤和结果。其他更为细节的，比如对失败的查询的测试，或其他类似的，比如与插入、更新类似的删除数据，删除表，由于篇幅原因，就不再细述。

1.系统启动测试

(a) 测试条件：代码编码完成，系统功能正常

(b) 测试步骤：

i. 启动系统

ii. 用mysql客服端连接系统

(c) 测试结果：系统能正常启动，而且mysql客服端也可以连接上系统

2.工具语句测试

(a) 测试条件：系统启动成功，客服端连接上系统

(b) 测试步骤：

i. 打开命令行客服端

ii. 输入语句show databases，发送给服务器

(c) 测试结果：系统返回当前所有的数据库。

3.建表语句测试

(a) 测试条件：系统启动成功，客服端连接上系统

(b) 测试步骤：

i. 打开命令行客服端

ii. 输入语句create table test(id int,name varchar(100))

iii. 回头，发送命令给服务器

(c) 测试结果：系统返回正确，而且文件系统里面多了一个文件。

4.插入语句测试

(a) 测试条件：系统启动成功，客服端连接上系统

(b) 测试步骤：

i. 打开命令行客服端

ii. 输入语句insert into test(id,name) values(1,'changhong');

iii. 回头，发送命令给服务器

(c) 测试结果：系统返回正确

5.查询语句测试

(a) 测试条件：系统启动成功，客服端连接上系统

(b) 测试步骤：

i. 打开命令行客服端

ii. 输入语句select * from test;

iii. 回头，发送命令给服务器

(c) 测试结果：系统返回一行数据。

6.更新语句测试

(a) 测试条件：系统启动成功，客服端连接上系统

(b) 测试步骤：

i. 打开命令行客服端

ii. 输入语句update test set name='changhong1'

iii. 回头，发送命令给服务器

iv. 输入语句select * from test

v. 回头，发送命令给服务器

(c) 测试结果：系统执行正确，返回一条数据，名字字段weichangohng1

7.更新语句测试

(a) 测试条件：系统启动成功，客服端连接上系统

(b) 测试步骤：

i. 打开命令行客服端
ii. 输入语句delete from test
iii. 回头，发送命令给服务器
iv. 输入语句select * from test
v. 回头，发送命令给服务器

(c) 测试结果：系统执行正确，没有返回数据

6.2.2 集群功能测试

集群的功能测试和单机版一样，考虑到不可能每条语句的测试。本次测试只测试最常用的insert语句：

1.测试条件：

(a) 数据库1启动成功，没有任何数据
(b) 数据库2启动成功，没有任何数据
(c) 数据库1和数据库2都有一个test表

2.测试步骤：

(a) 打开命令行客服端，连接数据库1
(b) 输入语句insert into test(id,name) values(1,'changhong')并且执行;
(c) 断开连接，连接数据库2
(d) 输入语句select * from test;
(e) 回头，发送命令给服务器

3.测试结果

(a) 数据库1执行语句正确
(b) 语句2执行正确，并且返回一条数据

6.2.3 审计功能测试

审计功能的测试主要是测试可视化的显示功能。

1.测试条件：

(a) 数据库1启动成功，没有任何数据
(b) 数据库1有一个test表

2.测试步骤：

(a) 打开命令行客服端，连接数据库1
(b) 输入语句insert into test(id,name) values(1,'changhong')并且执行;
(c) 打开grafana web界面

3.测试结果

(a) 数据库1执行语句正确

(b) 通过web界面可以观察到插入的数据和执行者的信息

(c) 图6-1,图 6-2和图 6-3显示了监控的可视化结果

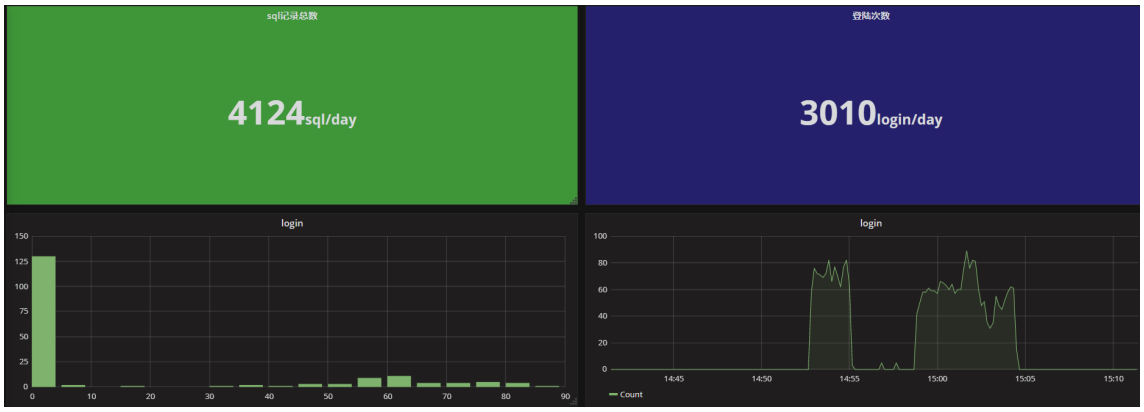


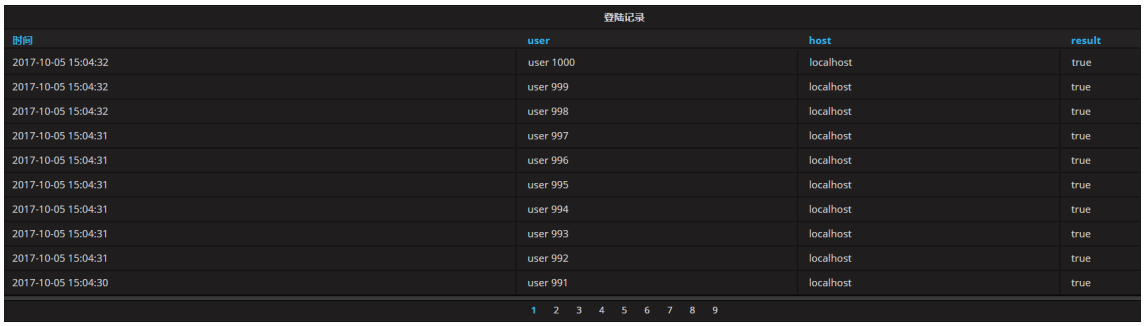
图 6-1 监控可视化结果

图 6-2 监控可视化结果

6.3 性能测试

数据库的更删改查是最常用的操作，其中又以查找使用的场景更多，所以本节主要测试数据库的查询性能。jsql和mysql都可以通过JDBC来连接，下面的源代码是测试代码，用来测试查询次数和执行时间的关系。执行结果如图6-4所示。

```
01    fun test() {
02        //sql执行次数
03        val sqlnumber = intArrayOf(10, 100, 1000, 3000, 5000, 10000)
```



登陆记录

时间	user	host	result
2017-10-05 15:04:32	user 1000	localhost	true
2017-10-05 15:04:32	user 999	localhost	true
2017-10-05 15:04:32	user 998	localhost	true
2017-10-05 15:04:31	user 997	localhost	true
2017-10-05 15:04:31	user 996	localhost	true
2017-10-05 15:04:31	user 995	localhost	true
2017-10-05 15:04:31	user 994	localhost	true
2017-10-05 15:04:31	user 993	localhost	true
2017-10-05 15:04:31	user 992	localhost	true
2017-10-05 15:04:30	user 991	localhost	true

1 2 3 4 5 6 7 8 9

图 6-3 监控可视化结果

```
04          val mysql = "jdbc:mysql://localhost:3306/changhong?" + "user=root&"
+
05                  "password=0000&useUnicode=" + "true&characterEncoding=UTF8"
06          var connection = DriverManager.getConnection(mysql)
07          var statement = connection.createStatement()
08          println("mysql sqlnumber to times:")
09          for (number in sqlnumber) {
10              val start = System.currentTimeMillis()
11              for (i in 1..number) {
12                  statement.executeQuery("select * from test")
13              }
14              val end = System.currentTimeMillis()
15              println("$number      ${end - start}")
16          }
17          connection.close()
18          val jsql = "jdbc:mysql://localhost:9999/changhong?" + "user=root&"
+
19                  "password=0000&useUnicode=" + "true&characterEncoding=UTF8"
20          connection = DriverManager.getConnection(jsql)
21          statement = connection.createStatement()
22          println("jsql sqlnumber to times:")
23          for (number in sqlnumber) {
24              val start = System.currentTimeMillis()
25              for (i in 1..number) {
26                  statement.executeQuery("select * from test")
```

```
27              }
28              val end = System.currentTimeMillis()
29              println("$number      ${end - start}")
30          }
31          connection.close()
32      }
33  }
```

SQL执行次数	MYSQL完成时间（毫秒）	JSQL完成时间（毫秒）
10	5	4
100	31	40
1000	252	315
3000	1207	941
5000	648	1504
10000	1297	4495

图 6-4 mysql和jsql的性能比较

从图6-4可以看出，随着SQL执行次数的增加，mysql和jsql的执行时一般都响应的增加，在3000执行次数为3000以下时，jsql和mysql的性能相当，当执行次数大于5000时，mysql的性能小幅度的超过jsql。再从变化幅度来看，当执行语句是5000次数的时候，mysql的执行时候审计比执行3000次数的时候还要少，这点可能是因为mysql本身的缓存或者优化有关。而jsql就相对来说随着执行次数的增加，时间就随着增加，符合预期。

除了jdbc的测试以外，本次测试还用了一个流行的测试框架JMeter。 Apache JMeter是Apache组织开发的基于Java的压力测试工具。用于对软件做压力测试，它最初被设计用于Web应用测试，但后来扩展到其他测试领域。它可以用于测试静态和动态资源，例如静态文件、Java 小服务程序、CGI 脚本、Java 对象、数据库、FTP 服务器，等等。JMeter 可以用于对服务器、网络或对象模拟巨大的负载，来自不同压力类别下测试它们的强度和分析整体性能另外，JMeter能够对应用程序做功能/回归测试，通过创建带有断言的脚本来验证你的程序返回了你期望的结果。为了最大限度的灵活性，JMeter允许使用正则表达式创建断言。用JMeter测试jsql的结果如图6-5所示。用JMeter测试jsql的结果如图6-6所示。

从测试结果来看，jsql相对mysql的性能有一定的差距。其中一个原因是jsql是用纯java语言开发的，而mysql用c语法开发，其中自然有一定的性能损耗。

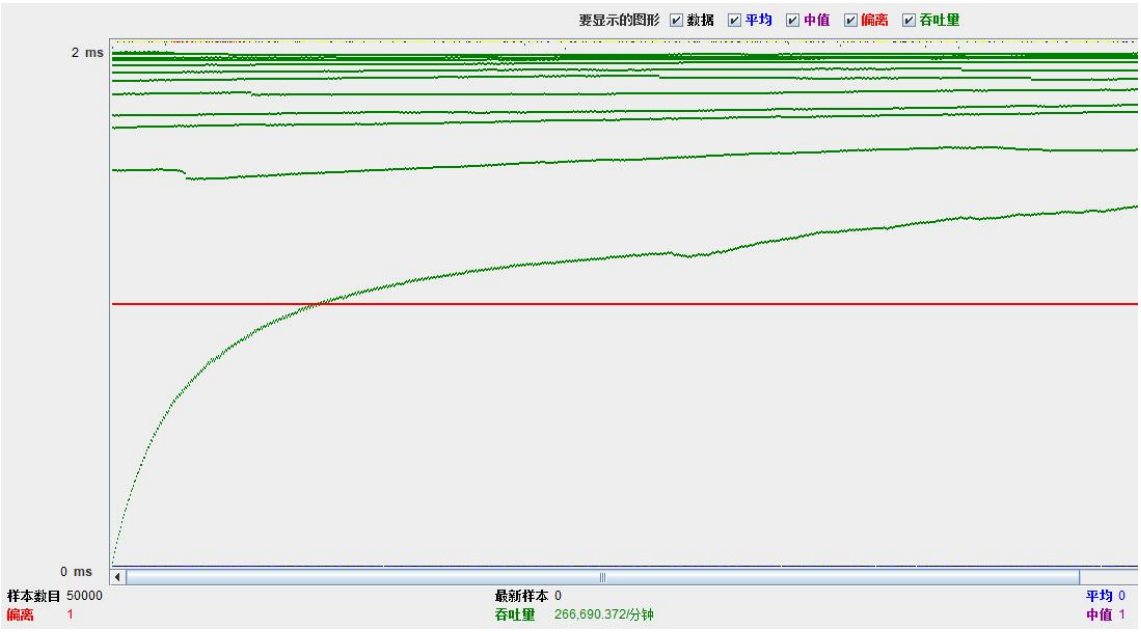


图 6-5 jsql的JMeter性能测试

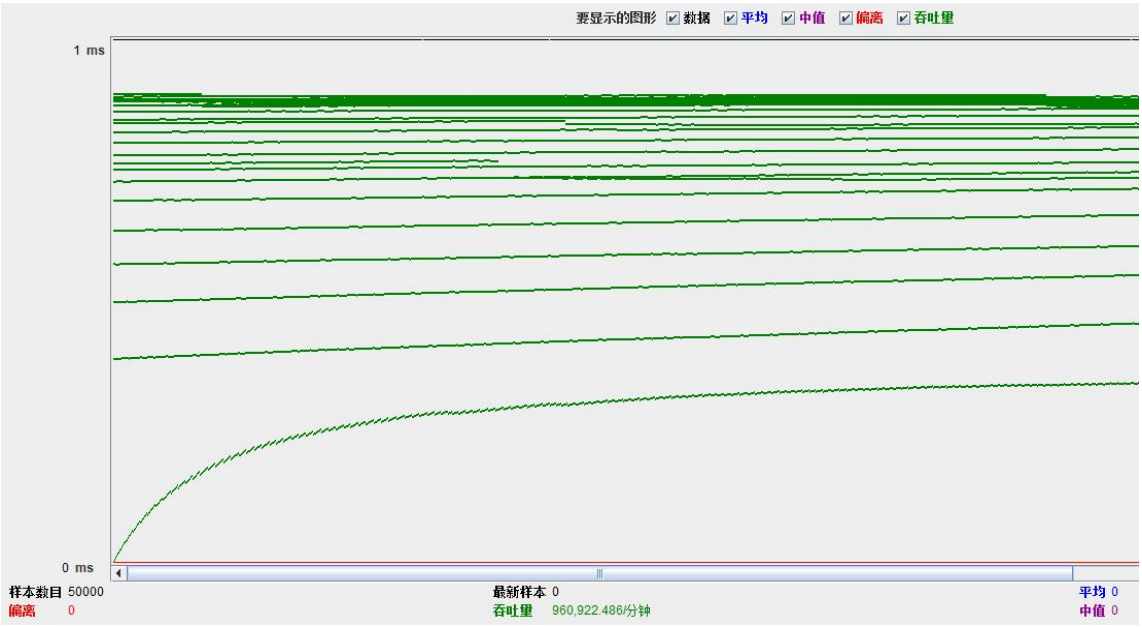


图 6-6 mysql的JMeter性能测试

6.4 本章小结

本章对JSQL数据库系统进行测试，主要分为功能测试和性能测试，功能测试包括数据库功能测试，集群功能测试以及审计功能的测试。性能测试分为JDBC和JMeter测试两部分。

第七章 总结与展望

现在社会的发展，离不开各种各样的数据库系统。关系型数据库在很多关键场合有不可替代的优势，对关系型数据库有关的理论研究和实践非常有意义。因此，在导师的指导下，在研究生阶段作者主要做数据库有关的开发工作。最终，将理论知识和实践技能相结合，开发出了这套分布式数据库系统。

分布式系统的研发是一个不小的挑战。作为这个系统的开发者，我首先在理论方面做了很多学习和研究，包括硬件硬件有关的知识，和数据库事务相关的理论，以及各种分布式相关的协议。理论只能作为指导，而不能成为一个系统。在实现方面，我主要深入学习了JAVA语言，对多线程和高性能网络开发技能也进行了深入的学习。最终实现的这套数据库系统作者认为有如下优先：

1.从数据库引擎到SQL模块完全采用JAVA语言编写，具有跨平台和安全的特点。

2.利用哈希树实现了高性能的存储引擎，特别适合现在的应用程序。

3.在系统的架构设计上，采用了面向对象的思想，对各个模块进行了很好的。划分，有利于对系统进行进一步的完善。

4.从数据库系统本身加入了审计功能。

当然，由于作者水平有限，本系统难免还有很多需要完善的地方，总结起来有下面几个方面：

1.系统的数据恢复机制不够完善，后续需要对存储引擎进行进一步的开发，加入日志等功能。

2.分布式数据库中数据迁移目前还没有实现，这是值得进一步研究的课题。

3.关系型数据库大都有存储过程和触发器这些功能，本系统作为一个实验室产品，目前还没有完全的实现这些功能。

作者认为，虽然现在出现了很多的Nosql数据库系统，但是关系型数据库的作用是无法替代的，电子商务和各种银行业务都需要关系型数据库的支持，所以对关系型数据库的学习和实践特别有意义。另外，数据库的安全问题越发严重，时常发生管理员篡改数据的情况发生，所以作者觉得，每个数据库系统，都要从底层加入安全审计功能，这样我们的数据才能保证最基本的可靠性。关于本系统，需要做的事情还有很多，但是我相信随着进一步的开发，本系统的功能会进一步完善。

致 谢

时间过的真快，转眼间，2年多的研究生生活就快要结束了。回顾自己的研究生学习生涯，感慨万千。论文的完成，除了自己的努力以外，更离不开老师和学弟们的帮助，在论文成稿之际，衷心感谢给予自己悉心指导和热情帮助的各位老师和学弟们。

论文的完成，首先要感谢我的校内导师曹晟教授，在整个论文的的写作过程中，曹老师都给予了我很大的关心和帮助。从作者毕业论文的选题、写作一直到最终完成的过程中，曹老师都是在百忙的工作中以一贯认真负责的态度认真仔细阅读作者的论文，给予作者耐心的指导，使得论文能够顺利的完成。他严肃的科学态度，严谨的治学精神，以及精益求精的工作作风，深深地感染和激励着我。

在这一年多的时间里，我还要感谢我的学弟们。感谢你们陪我一起开发这个分布式数据库系统，我们一起学习，一个努力，才能让本系统顺利完成。任何一个系统都要靠团队合作，更何况分布式系统这个既具有挑战性的工程项目，没有你们的帮助，就不可能按时完成这个系统。对学弟们的帮助，在此表示非常的感谢。

最后，作者非常感谢负责评审论文的教师、专家和教授，感谢你们认真负责的阅读论文，感谢你们为论文提出的宝贵意见和建议。

参考文献

[1] 刘蓬. NIO高性能框架的研究与应用[D]. 长沙: 湖南大学, 2013, 50–60

[2] 杨东, 谢菲. 分布式数据库技术的研究与实现[J]. 电子科学技术, 2015, 02(01):68–71

[3] 杨传辉. 大规模分布式存储系统[M]. 北京: 机械工业出版社, 2013, 56–60

[4] 马应龙. 分布式系统概念和设计[M]. 北京: 机械工业出版社, 2013, 56–60

[5] 安延文. 数据库审计系统中MySQL协议的研究和解析[D]. 河北: 华北电力大学, 2013, 50–60

[6] 贺杰. 分布式数据库中数据复制的研究和实现[D]. 南京: 东南大学, 2014, 50–60

[7] 邓蕾. 基于关联规则的数据库安全审计系统[D]. 长沙: 中南大学, 2011, 50–60

[8] 黄贵. OceanBase分布式存储引擎[N]. 华东师范大学学报, 2014年9月

[9] 李黎明. 安全数据库概述和前瞻[R]. 北京: 南京航天大学信息与技术学院, 2005年5月

[10] Wikipedia. Hard disk drive [EB/OL]. https://en.wikipedia.org/wiki/Hard_disk_drive

[11] Wikipedia. B+ tree [EB/OL]. https://en.wikipedia.org/wiki/B

[12] Wikipedia. Log-structured merge-tree [EB/OL]. https://en.wikipedia.org/wiki/Log-structured_merge-tree

[13] Wikipedia. Netty [EB/OL]. https://en.wikipedia.org/wiki/Netty

[14] Wikipedia. Elasticsearch [EB/OL]. https://en.wikipedia.org/wiki/Elasticsearch

[15] Wikipedia. Apache JMeter [EB/OL]. https://en.wikipedia.org/wiki/Apache_JMeter

[16] Wikipedia. Hazelcast [EB/OL]. https://en.wikipedia.org/wiki/Hazelcast

[17] 王智慧. 安全数据库审计子系统[D]. 上海: 复旦大学, 2011, 50–60

[18] 张瑞芳. 分布式数据库的查询优化方法设计与实现[D]. 成都: 电子科技大学, 2010, 50–60

[19] 王威. MySQL 数据库源代码分析及存储引擎的设计[D]. 南京: 南京邮电大学, 2012, 50–60

[20] 别小凡. 分布式内存数据库的设计与实现[D]. 西安: 西安电子科技大学, 2012, 50–60

[21] 解辉. 嵌入式数据库的设计与实现[D]. 太原: 太原科技大学, 2008, 50–60

[22] 杨磊. 数据库安全审计检测系统的设计与实现[D]. 北京: 北京交通大学, 2014, 50–60

[23] 张忠能. 分布式数据库关键技术研究与应用[D]. 上海: 上海交通大学, 2015, 50–60

[24] 王琳. 数据库监控系统的设计与实现[D]. 天津: 南开大学, 2011, 50–60

[25] 谢鹏. 分布式数据库存储子系统设计与实现[D]. 成都: 电子科技大学, 2013, 50–60

[26] 张俊民. 在线审计建模与实现[D]. 河北: 华北电力大学, 2015, 50–60