2016届研究生硕士学位论文

分类号：_____ 学校代号: 10269

密　级：_____ 学　　号: 51131500028

華東師範大學

East China Normal University

硕士学位论文

MASTER'S DISSERTATION

论文题目：可扩展数据管理系统中的高可用性实现

院　　系:	计算机科学与软件工程学院
专　　业:	软件工程
研 究 方 向:	WEB数据挖掘
指 导 教 师:	宫学庆　教授
学位申请人:	庞天泽

2016 年4 月8 日

Dissertation for master's degree in 2016 School Code: 10269
Student ID: 51131500028

East China Normal University

Title: **THE IMPLEMENTATION OF HIGH AVAILABILITY IN SCALABLE DBMS**

Department:	School of Computer Science and Software Engineering
Major:	Software Engineering
Research direction:	WEB Data Mining
Supervisor:	Prof. GONG Xueqing
Candidate:	PANG Tianze

April, 2016

华东师范大学学位论文原创性声明

郑重声明：本人呈交的学位论文《可扩展数据管理系统中的高可用性实现》，是在华东师范大学攻读硕士/博士（请勾选）学位期间，在导师的指导下进行的研究工作及取得的研究成果。除文中已经注明引用的内容外，本论文不包含其他个人已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中作了明确说明并表示谢意。

作者签名:_____ **日期**: 年 月 日

华东师范大学学位论文著作权使用声明

《可扩展数据管理系统中的高可用性实现》系本人在华东师范大学攻读学位期间在导师指导下完成的硕士/博士（请勾选）学位论文，本论文的著作权归本人所有。本人同意华东师范大学根据相关规定保留和使用此学位论文，并向主管部门和学校指定的相关机构送交学位论文的印刷版和电子版；允许学位论文进入华东师范大学图书馆及数据库被查阅、借阅；同意学校将学位论文加入全国博士、硕士学位论文共建单位数据库进行检索，将学位论文的标题和摘要汇编出版，采用影印、缩印或者其它方式合理复制学位论文。

本学位论文属于（请勾选）

（ ）1.经华东师范大学相关部门审查核定的“内部”或“涉密”学位论文*，于 年 月 日解密，解密后适用上述授权。

（ ）2.不保密，适用上述授权。

导师签名_____ 本人签名_____

年 月 日

*“涉密”学位论文应是已经华东师范大学学位评定委员会办公室或保密委员会审定过的学位论文（需附获批的《华东师范大学研究生申请学位论文“涉密”审批表》方为有效），未经上述部门审定的学位论文均为公开学位论文。此声明栏不填写的，默认为公开学位论文，均适用上述授权）。

庞天泽 硕士学位论文答辩委员会成员名单

姓名	职称	单位	备注
周傲英	教授	华东师范大学	**主席**
钱卫宁	教授	华东师范大学	
高明	副教授	华东师范大学	
蔡鹏	副教授	华东师范大学	
张召	副教授	华东师范大学	

摘　　要

近年来，随着互联网不断发展和传统行业的互联网化进程加剧，金融等行业开始面对越来越多的用户人数以及巨大的并发访问量，产生了TB甚至PB级别海量数据。传统数据库及其纵向扩展方式已经不适合互联网化对于海量数据存储、数据高速增长以及自由扩展的需求。因此人们将目光投向分布式数据库技术。虽然分布式数据库能够通过横向扩展来解决数据存储的困境，但是在满足金融等业务对于数据库高可用性的需求方面面临挑战。本文针对可扩展数据管理系统的架构特点，面向金融行业关键任务的可用性需求，在分析了现有分布式一致性协议及其实现和适用范围的基础上，提出了两种关键节点选举方法。为了验证其有效性和性能，本文在开源分布式数据库OceanBase中实现了所提出的选举方法，并在银行真实环境下进行了测试和分析。本文的贡献如下：

1. 针对可扩展数据管理系统中的可用性问题，提出一种防止系统总控节点和事务处理节点出现单点故障的高可用框架。通过总控节点的自主选举来实现高可用性和主总控节点的唯一性，并进一步由稳定且唯一的主总控节点来选择主事务处理节点，这同样也保证了事务处理节点的高可用。

2. 针对Raft协议在实现中可能导致的双主问题和频繁选举问题，提出了一种改进的分布式选举协议。该协议可以在各节点时钟基本一致的情况下，保证协议的正确性，并且不会出现对于真实分布式数据管理系统较为致命的双主问题和频繁选举问题。

3. 在开源系统OceanBase中实现并通过大量在某银行真实应用环境中的密集的

实验，论文展现了所提方法在应对主节点故障和网络分区故障下的有效性。实验表明，在一个分布式数据管理系统内部，当出现主节点失效的故障时，系统可以在秒级内完成自我修复，达到企业级应用对于可靠性和可用性要求。

大量NoSQL和NewSQL系统主要面向互联网应用，由软件社区或互联网公司开发，不能直接用于国内金融行业数据存储和管理。而本文的研究工作解决了可扩展数据管理系统的高可用问题，并且已经在实际系统中被使用。这对我国自主研发适合于金融等关键行业的分布式数据管理系统有着重要意义，提供了有价值的实践经验。

关键词: 高可用性、可扩展数据管理系统、数据库恢复、分布式选举、分布式一致性维护

Abstract

In recent years, with the rapid development of the Internet, traditional industries, such as finance industry, are confronted with massive data and high concurrent access. Traditional central databases of scaling-up architecture cannot support massive data and rapidly increasing of data. Therefore, people begin to utilize distributed database technologies to solve those problems. However, distributed databases suffer from shortcoming in high-availability with strong consistency. We analyze existing distributed consensus algorithms as well as their implementation and application scope. Then we present two election algorithms for critical nodes based on scalable data management system architecture. Finally, we show that these algorithms implemented in OceanBase can provide high-availability in real application environment. Through performance studies, we quantify the cost of our algorithms.The primary contributions are:

1. We propose a kind of high-availability framework. It can deal with the single-point failure of nodes in scalable data management system, including control-nodes and transaction-nodes. This framework can achieve high-availability and the uniqueness of master control-node by leader election. And then, the unique master control-node can guarantee the high-availability of transaction-nodes.

2. To solve the problem about *split-brain* and *frequent re-election* of the Raft protocol when running on the real application scenarios, we present a modified distributed

election protocol. On the condition of basic consistency between nodes on clock, It can guarantee the validity of algorithm avoid the two critical problem.

3. The algorithms presented in this thesis are implemented on OceanBase, which is a distributed database with high scalability. A large number of intensive experiments was done in a bank's real application environment. The experimental results indicate that a distributed data management system can recover from failure in few seconds after the primary node downtime. And they can meet the requirements of reliability and availability required by enterprise applications.

Many NoSQL and NewSQL systems used for Internet are developed by software communities or Internet companies. They cannot be used to store and manage financial data in our country. We solve the problem about high-availability of scalable data management system in the thesis. And the algorithms have been used in real database system. It is significant to develop distributed data management system independently for mission-critical application in finance industry and bring valuable practical experience.

Key Words: *High Availability, Scalable Data Management System, Database Recovery, Distributed Leader Election, Distributed Maintaining of Consistency.*

目录

第一章 绪 论 1
1.1 研究背景 1
1.2 本文工作 3
1.3 本文结构 4
第二章 可扩展数据管理系统 5
2.1 可扩展数据管理系统的基本要素 5
2.2 相关系统 13
2.2.1 NoSQL 数据库系统 14
2.2.2 NewSQL数据库系统 16
2.3 可扩展数据管理系统的理论基础 17
2.3.1 数据库可用性的概念和衡量标准 17
2.3.2 CAP 定理 18
2.3.3 ACID 模型和BASE 模型 19
2.4 高可用技术实现 21
2.4.1 RAID技术 21
2.4.2 数据库复制 21
2.4.3 高可用集群 22
2.5 本章小结 23
第三章 主总控节点选举 25
3.1 问题描述 25

3.2 Raft协议及分析 27
3.2.1 Raft协议概述 28
3.2.2 Raft协议问题一：出现双主 32
3.2.3 Raft协议问题二：频繁选举 33
3.3 LEBR选举算法 35
3.3.1 LEBR基础 36
3.3.2 LEBR非选举期 40
3.3.3 LEBR选举期 44
3.3.4 算法分析 48
3.3.5 算法实现 55
3.4 本章小结 58
第四章 主事务处理节点选举 61
4.1 问题描述 61
4.2 MTS选举算法 64
4.2.1 算法设计 64
4.2.2 算法分析 67
4.2.3 算法实现 70
4.3 本章小结 72
第五章 实验 73
5.1 实验设置 73
5.2 实验结果分析 75
5.2.1 LEBR算法相关测试 75
5.2.2 MTS算法相关测试 79
5.3 本章小结 83
第六章 总结与展望 85
参考文献 89
致谢 96

发表论文和科研情况 99

插图

2.1 LSM-Tree结构模型图 10

2.2 基于LSM-Tree的读写分离存储架构图 11

2.3 ScaleDDB数据管理系统架构图 12

3.1 ScaleDDB高可用集群架构 26

3.2 Raft状态转换图 30

3.3 Raft双主问题示意图 33

3.4 Raft频繁选举示意图 34

3.5 ScaleDDB 三总控节点部署架构图 37

3.6 LEBR状态转换图 38

3.7 任期和逻辑时间 39

3.8 LEBR算法心跳信息交互图 42

3.9 失联确认窗口示意图 43

3.10 LEBR 选举流程示意图 45

3.11 多个Candidate获得多数选票情况示意图 51

3.12 算法导致网络分区的场景示意图 52

4.1 基于Quorum 的日志同步示意图 63

5.1 不同参选RS个数下的故障恢复时间 76

5.2 出现网络丢包时故障恢复时间 77

5.3 数据库负载对LEBR 算法效率的影响 78

5.4 LEBR算法对于DDL操作的影响 79
5.5 不同备选UPS 个数下的故障恢复时间 80
5.6 数据库负载对MTS 算法效率的影响 81
5.7 MTS算法对于读写访问的影响 83

表格

3.1 逻辑角色和物理角色对应表 38
3.2 主节点选举相关术语列表 49
3.3 NEStat数据结构 55
3.4 选举中涉及的消息包 56
3.5 ONStat数据结构 57
3.6 不同逻辑角色下对于消息包的处理 59

4.1 事务更新节点选择算法相关术语 68
4.2 TStat数据结构 70

5.1 OceanBase与ScaleDDB节点对应关系表 73
5.2 实验环境配置 74
5.3 YCSB参数设置 78
5.4 Sysbench参数设置 82

第一章　绪 论

1.1　研究背景

随着信息技术的不断进步和成熟，以及互联网的高速发展，互联网行业正逐步向传统行业延伸，例如金融、教育、物流运输等行业。随着传统行业互联网化进程的推进，数据规模越来越大，很多大型金融企业的数据总量达到TB（Terabyte）甚至PB（Petabyte）级别，并且每天新增数据量也以百GB （Gigabyte）计。海量数据规模对于数据存储、分析等产生巨大挑战。金融行业经过几十年的发展，信息化程度已经非常高。很多银行建设数据中心实现了全行数据的集中存储处理，不仅对外提供在线金融服务，对内也实现了智能化管理。对于关键任务（Mission-critical）系统，如网银、电子支付等，互联网化使得这些系统需要直接面对广大客户提供金融服务，这极大地增加了每日产生的数据量，同时也带来了很大的高并发负载量。在这种情况下，关键任务仍然要求系统具备高可靠性和可用性，不仅要保证已经写入数据库的数据在任何情况下不会丢失，还要使系统不可用时间尽可能短，以减少系统故障对用户造成损失。为了满足这些需求，底层的数据库系统要在能够支撑起海量数据存储和处理的情况下，具有高可用性。

以国内商业银行为例，使用关系型数据库系统来存储和管理银行数据。一般采用国外大型厂商提供的数据库解决方案，核心业务则使用Oracle和DB2 这类大型多功能数据库，以单独的小型机或大型机作为存储和管理平台，为上层应用系统提供较高的数据安全性、可靠性以及高性能。随着金融业发展程度日益加深，银行业务不断扩大，用户数量和使用频率也会迅速增加，由此带来的海量

数据存储和处理问题愈加不容忽视。传统的单机模式数据库有限的存储能力和纵向扩展（Scale Up）方式无法支撑日益增长的海量数据，也很难避免单点故障（Single-point Failure）所带来的系统不可用问题。一些数据库企业通过对自己产品的创新升级，提供多实例并行处理的集群部署方式，并针对各自产品的特点实现高可用性解决方案，例如Mysql Cluster[1]、DB2 PureScale[2] 等。然而这些数据库系统没有很好的解决海量数据存储问题，很多产品虽然在服务层实现了并行处理机制，但仍然使用共享存储。另外，为了获取高可用性，数据以全冗余方式存储在系统中各服务器上，这不仅浪费了大量存储资源，还降低了数据更新效率。

近年来，分布式数据库系统逐渐走进人们视线，出现了许多较为成熟的分布式数据管理系统，如Apache Cassandra[3]、Google Spanner[4] 和Alibaba OceanBase[5] 等。分布式数据库系统实现了完全意义上的数据分布存储，采用横向扩展（Scale Out）方式，通过增加数据节点来扩充容量，满足海量数据存储需求。另一方面，分布式数据库只需要廉价的PC服务器即可搭建并满足应用需求，具有很高的性价比，因此受到越来越多企业的关注和使用。分布式数据库系统中包含不同功能的节点，各类节点面对的可用性问题也不同。例如对于存储节点来说，其可用性体现在数据不会因为某几个节点的故障就无法访问，这可以采用数据多副本分布存储的方式实现，少于副本数的节点故障不会对系统产生任何影响。分布式数据库系统中还有一些关键节点，这些节点唯一存在并且在系统中承担重要职责，一旦出现故障便会导致数据库系统异常甚至瘫痪，例如控制节点和更新节点。控制节点，或称协调节点，负责管理集群内所有节点的状态信息，帮助各个节点协同工作，对外提供统一的访问接口；而更新节点在系统中负责处理写请求。为了实现高可用性，需要保证系统的关键节点尽可能存在，一种做法是给关键节点配置多个备用节点，这样即使出现故障，系统也能够自行选择其它节点成为新的关键节点，使数据库系统恢复服务。如何正确而高效地从备用节点中选出最合适的成为主节点，是实现分布式数据库系统高可用的关键问题，也是是分布式系统中的难点和研究热点。虽然已有一些分布式一致性协议能够处理主节点选

举，然而在实现过程中多存在问题。本文针对这些问题，面向实际应用场景开展研究工作。

1.2 本文工作

本文针对很多分布式数据库难以满足金融等行业对于系统高可用性的需求的问题，从主节点选举方面考虑，设计一种适应真实应用场景的分布式数据库高可用方法，并在OceanBase的基础上进行实现。研究的具体工作如下：

1. 总结和分析了分布式数据管理系统在架构上的共同特点，提出一种采用读写分离架构和主从模式的可扩展数据管理系统的抽象模型ScaleDDB，分析了该类系统在可用性方面存在的问题，并针对该系统架构，提出一种解决ScaleDDB总控节点和事务处理节点单点故障问题的高可用方法。

2. 研究和分析了一致性算法Raft的设计思路和实现细节，指出该协议可能导致的双主和频繁选举问题，而这在真实应用场景中可能是致命的。为了解决这两个问题，本文提出了一种改进的分布式选举算法LEBR，用于实现ScaleDDB总控节点的高可用。在节点间时钟基本一致的情况下，该算法能够保证其正确和高效，同时也保证不会出现双主问题和频繁选举问题。

3. 本文针对ScaleDDB事务更新节点的高可用问题，提出一种主事务更新节点选举算法MTS，并详细描述其设计思想和实现。该算法通过总控节点实现对主事务更新节点的选择，能够保证在总控节点稳定的情况下其选择结果的唯一性。

4. 在开源的类ScaleDDB系统OceanBase中实现了本文提出的LEBR算法和MTS算法，共同作为其高可用实现方案的一部分。为了测试主节点故障和网络分区故障下的两种算法的有效性，进行了大量在真实应用环境中的实验，实验表明，当分布式数据管理系统中，主节点故障而失效时，本文提出的算法能过

确保系统在短时间内完成故障检测和恢复，同时也说明了算法在系统不可服务时间内只占很小的比例，可以忽略。

本文工作为可扩展数据管理系统关键节点的高可用问题提供了一种高效的解决方法，并且这种方法是面向工程实践的，能够处理真实场景中存在的问题。事实上，本文提出的方法已经应用于大型银行底层的数据管理系统中，给我国实现金融领域自主可控系统的规划贡献了一份力量。

1.3 本文结构

本文内容结构安排如下：

第二章介绍了本文研究内容的相关工作成果，概括描述了新型分布式数据管理系统的抽象架构，并介绍了相关的分布式数据库实现，最后对分布式数据库高可用性的研究工作和问题进行介绍和分析。

第三章首先介绍分布式一致性算法Raft，并分析其主节点选举模块的基本原理，以及实际中使用时所存在的问题。然后本文提出了能够解决这些问题的主节点选举算法LEBR，描述其设计思路和实现方法。同时，还结合各种故障场景分析了LEBR算法的执行流程和实现效率。

第四章提出了事务处理节点的主节点选举算法MTS，描述该算法的设计和具体实现，同时也对其故障恢复流程进行分析。

第五章通过实验测试本文提出的LEBR算法和MTS算法的执行效率，以及采用这两种算法的系统在故障发生时的表现情况，并对实验结果进行分析。

第六章总结本文的研究内容，并展望未来工作。

第二章　可扩展数据管理系统

分布式系统，特别是分布式数据管理系统作为应对大规模数据存储和处理的重要手段，一直是学术界和工业界研究的热点。其中，分布式系统中的可用性和一致性是人们关注的重点问题。本章针对可扩展数据管理系统及其可用性问题展开调研工作，介绍常见的NoSQL和NewSQL分布式数据库及其特点。同时，还将介绍数据库领域中高可用相关的理论基础和实现技术，并进行分析。通过本章内容，可以了解可扩展数据管理系统及高可用性相关的基础知识和研究现状。

2.1　可扩展数据管理系统的基本要素

20 世纪70 年代中期，计算机网络与通信技术的发展极大丰富了人们使用计算机处理信息的方式，对于数据库存储和管理提出了更多的需求，传统集中式数据库系统在新的应用场景下的局限逐渐显现出来，包括：

- 大型企业的组织架构多采取各部门自治而在上层统一管理的形式。从企业数据管理的角度来看，每个部门内部只需要维护与自己工作相关的数据，但又需要保证企业信息资源不会形成“信息孤岛”（Information Island）。显然，单服务器的集中式数据库无法满足这类企业发展的需求。

- 集中式数据库由于是单节点架构，所有数据集中处理，因此在面对数据存储和访问压力时，会使用纵向扩展的方式提升数据库性能，例如添加CPU、扩大内存、增加硬盘存储容量以及优化数据库管理系统等。这种方式扩展性不高，而且效果有限。

- 数据库仅运行于一台服务器上，如果该服务器出现故障，应用程序将无法再访问数据库，这就是所谓的“单点故障”。因此，关键性行业搭建集中式数据库时，多使用低故障率的高性能服务器。但是这没有从根本上解决问题，而且还带来了很高的成本。

在这样的背景下，分布式数据库系统开始逐渐成为数据库领域新的研究方向和热点[6, 7]。分布式数据库是数据库技术与计算机网络相互碰撞、结合而产生的，与集中式数据库相比，它包含多个在地理上分离而在逻辑上集中的节点，并通过网络相互连接通信。每个节点在网络上都是独立的计算机（也可以引申为数据库进程），能够自主存储并管理本地数据，但同时所有节点通过数据库管理系统能够作为整体对外服务。分布式数据库系统的出现适应了时代发展的潮流，满足了企业对于统一数据的局部分散管理和全局协同管理的需求。

对于分布式数据库系统（Distributed Database System, DDBS），可以根据组成数据库的局部节点上的数据库管理系统数据模型分为两类[8]：同构型数据库（Homogeneous DDBS）和异构型数据库（Heterogeneous DDBS）。同构性DDBS 的特点是各个节点上的数据库数据模型都是一致的，例如全部是关系型；而如果是不一致的，就可以归为异构型DDBS。另外还有一种更普遍的分类方法，即按照分布式数据库控制系统的架构类型可以分为三类：

- 集中控制型DDBS：若数据库系统的全局控制模块和数据分布信息存储于唯一的中心节点（称为总控节点），并由该节点负责协调全局事务和全局数据访问，那么就可以称为集中控制型DDBS，如Oracle RAC(Oracle Real Application Clusters)[9]、基于切片（Sharding）技术的Mysql Fabric[10]、Microsoft SQL Azure[11] 等。这类系统的控制机制和数据一致性容易实现，但存在单点故障问题，并且当数据量规模很大时存在访问瓶颈。

- 分散控制型DDBS：分散控制型DDBS 中每个节点都可以控制整个分布式数据库的存取和访问，既是协调者也是参与者。虽然这类系统中，节点故障不会影响其它节点提供服务，但在保证数据一致性方面存在较大困难。

- 混合控制型DDBS：介于前面两类之间，当数据库节点中存在的主节点数大于1，且小于节点总数时，即为混合控制型DDBS。

在互联网高速发展的今天，数据量呈爆发式增长，数据类型也展现出多样性，大量结构化和非结构化数据需要更加高效的存储和处理。这不仅使得集中式数据库系统愈加难以满足应用需求，甚至传统架构的分布式数据库也不再适应。为了应对这些挑战，很多企业，特别是互联网企业开始寻求新的解决方案，提出新的分布式存储架构和管理方式，并实现了成熟的、适应海量数据应用需求的分布式数据库。新型分布式数据库是一种集群式架构的系统，内部通过高速网络互连，数据以多副本形式分布存储于部分节点中，其最重要的特征是高可扩展性，因此又被称作可扩展数据管理系统（Scalable Data Management System），例如HBase[12], Google Spanner[4] 等。虽然新型DDBS 与前面提到的传统DDBS存在共同点，例如同样都是构建在网络中多个节点上，实现数据多点存储，但是它们之间也具有明显的差别：

- 新型DDBS 并不是若干个集中式数据库的简单组合，各个节点仅承担数据库的一部分任务，例如数据存储、事务处理、集群管理等，它们相互协作共同完成数据库对外提供的各种功能。

- 新型DDBS 能够自动对数据划分后分布存储在多个节点上，并且对每份数据维护多个副本。数据的分布信息对用户透明。而在传统的DDBS 中，需要用户制定数据分片策略。

- 新型DDBS 拥有更加灵活的数据模式和设计架构，能够支持文本、图片、视频、XML/HTML 文档等非结构化或半结构化数据，并不局限于传统的结构化数据形式。

- 新型DDBS 是面向应用的，根据业务需求和数据格式进行定制是新型DDBS设计的出发点和遵循的重要准则，而对于SQL 和事务等传统数据库功能的支持不再是必须的。

在新型分布式数据库系统的设计过程中，经常会面对各种的业务场景和数据类型，因此不同的系统之间在架构、存储模式、读写处理等许多方面都有差异，但也存在共性。新型分布式数据库在追求高可用、高并发处理和海量数据存储时应具备的基本要素，包括主从模式、数据读写分离和横向扩展三项。下面本文将对这三个基本要素进行具体描述，并总结归纳出一种典型的分布式数据库设计架构。

• 主从模式

主从模式指在分布式数据库系统中存在一个节点负责监控和管理整个集群的运行，这个节点被称为总控节点。大多数分布式数据库中都带有总控节点，该节点能够控制数据库各个节点能够协同工作，共同完成诸如数据查询、事务处理等数据库任务，同时，使用总控节点可以使数据库系统的设计更加简洁，更容易实现全局性的控制任务，以及分布式事务和数据的一致性。主从模式的数据库中不允许出现一个以上的总控节点，就像人的身体无法由多个大脑控制一样，集群也无法同时执行多个总控节点的命令。否则会导致各种异常，例如资源竞争、数据不一致等。

• 横向扩展

新型分布式数据库的一个重要特性是，通过增加服务器节点，有效地实现数据库横向扩展，既包括读写能力的扩展，也包括存储容量的扩展，这是其能够支撑海量数据处理和存储的基础。可扩展性问题并不是简单的节点上下线管理问题。在实现时，很多其它数据库机制会对可扩展性产生影响，如分区策略、数据一致性和并发控制机制[13]：

- 分区策略是对数据进行切分的方法，决定了数据在多个节点中的分布，而数据的分布模式直接影响了扩展性的实现方法。
- 数据的多副本分布存储导致读写请求可能会转发到不同的数据副本上，通过副本间的复制机制可以保持数据一致性。但在不同模式的复制机制下，可扩展性高低也不一样。例如同步复制模式下，副本的数量越多，复制效率越

低，因此扩展性较低；反之，异步复制模式下，副本数量基本不会影响复制的效率，故可扩展性较高。

- 简单的读写锁难以满足新型分布式数据库对于并发读写性能的需求，随着节点的增多，锁性能还会进一步下降。因此，为了实现高可扩展性，很多系统使用多版本并发控制机制（Multi-Version Concurrency Control, MVCC）。

另外，在衡量可扩展性时，应当综合考虑系统扩容的便捷性、实时性、自动化程度，以及故障恢复时间等。

• 数据读写分离

读写分离是一种用于提升系统吞吐量，降低访问延迟的数据库技术。传统读写分离技术的基本思想是将读请求和写请求由不同的数据库节点分别处理，从而缓解数据库内部读写锁的竞争，节约系统开销，达到提升性能的目的。这种技术非常适用于读请求为主的业务场景，可以通过增加读服务节点来进一步提升查询效率。在集中式数据库系统中，可以通过配置主备模式来实现读写分离，备节点处理读请求，而主节点处理写请求，并通过复制技术将数据更新复制到备节点。在分布式数据库系统中，基于日志结构合并树（The Log-Structured Merge-tree, LSM-Tree）[14]实现数据读写分离是更普遍的做法。

LSM-Tree 是Patrick O’Neil等人提出的一种高效索引结构，能够显著提升数据更新性能，其基本思想是将索引分为两部分，分别由内存和磁盘进行维护管理。对数据的修改或插入操作全部由内存处理，更新内存中索引树，当数据量达到指定阈值时，再将内存中的数据批量合并到磁盘，图2.1 显示了LSM-Tree 结构模型。图中，C_{m0}和C_{d0}分别为内存和磁盘中原有的树结构。当合并发生时，C_{m0}中的两个叶子节点融合到了C_{d0} 中，在磁盘中生成新的树结构C_{d1}。而内存中的C_{m0}只剩下根节点，变成C_{d1}。至此便完成了数据合并。

这种方式通过将磁盘随机写入优化为批量写入，以及由内存高效处理所有的数据更新，大幅度提升了写入效率。但是，由于在LSM-Tree 结构中读取数据时，需要合并旧的磁盘数据（可称为基线数据）和新的内存数据（可称为增量

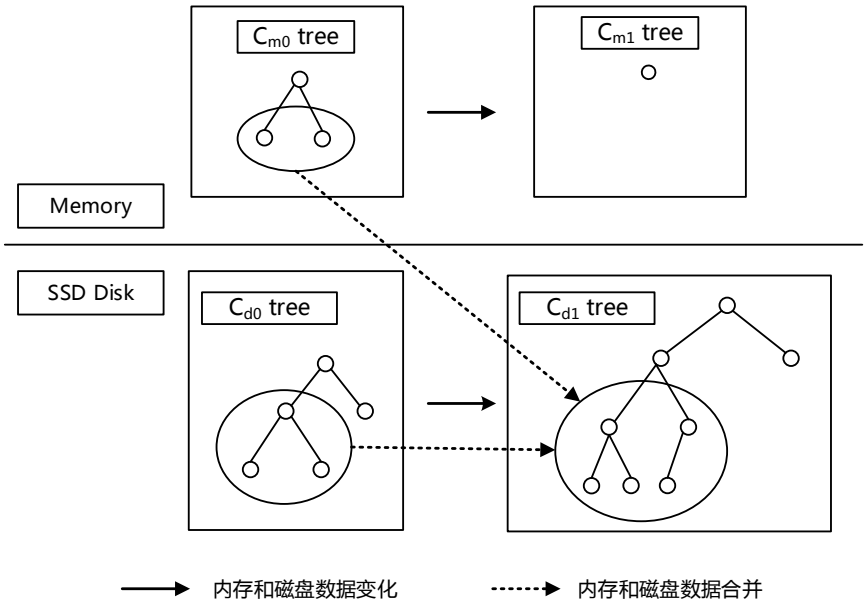

图 2.1: LSM-Tree结构模型图

数据），因此其读性能受到了一定影响，但并不显著。LSM-Tree 将数据更新和存储剥离的做法体现了读写分离的思想，其高效的写性能很好地满足了新型分布式数据库的要求，而读性能的降低并不明显，实践中通过优化手段可以弥补，甚至还能有所提升。图2.2显示了基于LSM-Tree 的读写分离存储引擎设计，其中MemTable(Memory Table) 表示内存增量数据，底层可以使用B+ 树等索引结构；SSTable(Sorted String Table) 则表示磁盘基线数据，以文件形式存储一系列有序键值对。处理客户端写请求时，数据可以直接写入MemTable；而处理读请求时，则需要访问SSTable和MemTable，并将获取的数据合并后再返回给客户端。该设计将数据库写操作集中到内存执行，避免了磁盘反复读写带来的巨大开销，也实现了数据的无锁读取。另外，对于大型分布式数据库来说，相对于全量数据，短期内的数据更新量较少，因此大部分的读请求只需要访问基线数据即可获取结果，数据合并带来的问题并不大，而且无锁访问还提升了读数据的效率。实践中有两种常用方式实现LSM-Tree 架构，其一是将增量数据和基线数据置于同一服务器，其二则是将两者部署于不同服务器。总之，在设计分布式数据库系统时，使用基

于LSM-Tree 读写分离技术一种较优的选择。

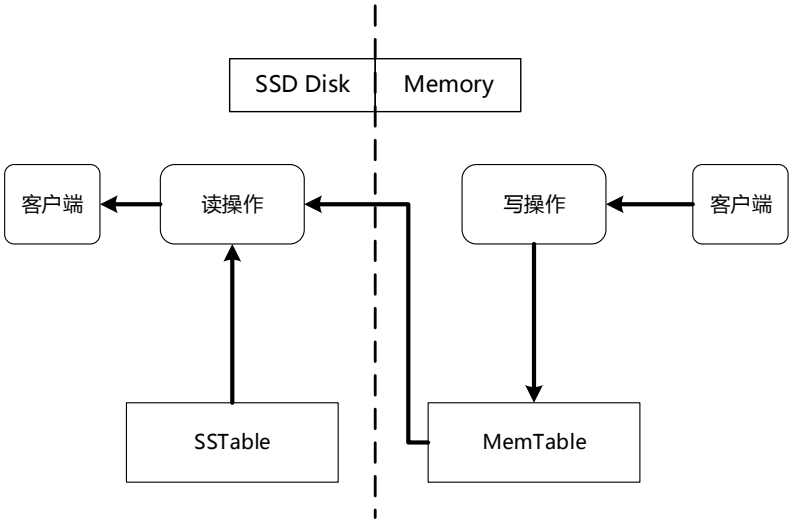

图 2.2: 基于LSM-Tree的读写分离存储架构图

结合前文所述，一个典型的可扩展数据管理系统的参考架构如图2.3所示。为了后续讨论方便，本文将符合这一框架的系统简称为ScaleDDB（Scalable Distributed Database）。现有的很多分布式数据管理系统的设计都与ScaleDDB 在架构上相类似。下面本文将对这个架构进行具体描述。ScaleDDB 使用基于LSM-Tree的存储模型，实现数据读写分离，数据的更新和读取被分配在不同的服务器节点上。

ScaleDDB 会将一段时间内（通常是一天）的数据修改量存放在MemTable 数据结构中，由于这部分数据相对较少，因此MemTable 可以完全保存在服务器的内存中；而对于数据库的基线数据，将以SSTable 格式保存在多个节点的磁盘中。ScaleDDB 通过定时任务定期将增量数据和基线数据合并。这种系统架构将写事务全部集中于单台服务器上，避免了分布式事务带来的问题，实现了较高的读写性能，比较适和于数据更新量远小于数据总量的业务场景。

根据模块功能不同，ScaleDDB 中的节点分为四类，包括事务处理节点（Transcation Node, TNode），基线数据存储节点（Data Node, DNode）、接口服务节点（Interface Node, INode）和总控节点（Control Node, CNode），每类服务器节

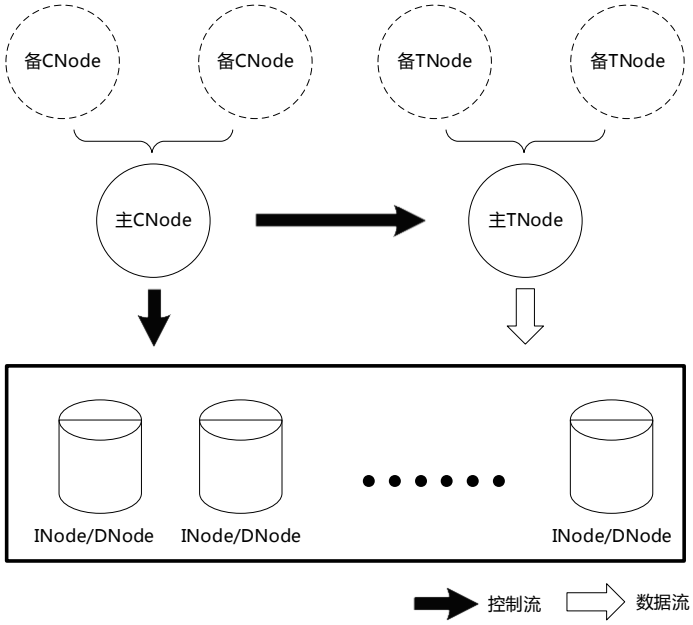

图 2.3: ScaleDDB数据管理系统架构图

点实现一个核心功能模块。其中：

- TNode 负责处理ScaleDDB 写事务，并存储增量更新的数据，是数据库集群中负载最高的节点，因此对于服务器的处理性能和内错容量有较高要求。在集群内只允许一个TNode 提供服务。TNode 可以理解为一个集中式内存数据库，实现跨行跨表的高性能事务处理。

- DNode 负责存储数据库系统的基线数据，将数据以多个副本的形式冗余保存在若干服务器节点上，以避免DNode 节点故障而导致数据丢失或不可访问。DNode 提供了对基线数据的读访问支持，并能够自动合并基线数据和TNode 上的增量数据，最终返回最新的数据。同时，为了避免每次读取数据都需要请求TNode，也为了反复利用TNode 有限的内存来存储数据更新，ScaleDDB 通过定期合并功能将增量更新数据融合到基线数据中，生成新的基线数据。

- INode 是ScaleDDB 的对外服务接口，接收并响应客户端的读写请求，负责SQL 解析，查询计划的生成和转发，以及对请求结果的合并等功能。同

时，INode 支持并发访问，能满足多用户访问需求。

- CNode 是ScaleDDB 集群的总控节点，管理集群的所有节点，实时监控系统运行状态，并执行负载均衡等系统级任务。此外，CNode还负责处理数据的分布信息和表结构模式（Table Schema）信息。如果该节点发生故障，最终将导致整个系统不可用。

从前面的介绍中可以看出，在ScaleDDB 中，TNode 和CNode 都存在单点故障，若这两类节点出现问题，会影响整个系统的使用，而DNode和INode则不存在这个问题。因此，ScaleDDB 的高可用性体现在TNode和CNode 节点上，需要采取高可用方案来解决这两类节点的单点故障。例如可以以主备模式（Master-Slave）配置TNode 节点以及CNode 节点。主备节点间相互备份，TNode 可以通过数据复制实现多数据副本的一致性，使其具有一定程度上的可用性。另外，根据数据规模可以在集群中配置多个INode 和DNode 服务器节点，这两类节点具有高可扩展性，ScaleDDb支持用户根据实际需求动态增加或减少节点数量。

2.2　相关系统

目前有很多较为成熟的可扩展数据管理系统，其中一些已经广泛应用于云计算、云存储、科学计算等领域。这些系统大致可以分为两种类型，即NoSQL(Not Only SQL)[15] 和NewSQL[16]。NoSQL 数据库指的是一类分布式的、非关系型的数据管理系统，适用于大规模数据的存储和处理，具有高并发性和高可扩展性，不支持传统的SQL 语法和事务处理，而是通过提供各种API 接口对外服务。NewSQL 数据库是介于NoSQL 和关系型数据库之间的一种解决方案，它既具备NoSQL 系统的特性和优势，又能够支持事务，使用户可以像关系型数据库一样使用NewSQL 数据库。Cattell[17]、Pokorny[18]、Sakr 等人[19] 描述了许多NoSQL 数据库系统的特点，包括它们的数据模型，基本原理和特点等。另外，文献[20, 21] 也总结了常见的NoSQL 和NewSQL 数据库的特点和区别，并从多个

方面将两类系统进行对比。

2.2.1 NoSQL 数据库系统

根据数据存储模型的不同，NoSQL 数据库系统可以分为四类[22]，包括：键值存储数据库（Key Value Store）、列簇式存储数据库（Column Family Stores）、文档数据库（Document Stores）和图形数据库（Graph Database）。其中文档数据库和图形数据库与本文工作没有关系，故不做介绍。下面本文将介绍属于键值存储数据库或列簇式存储数据库的分布式数据管理系统。

（1）键值存储数据库键值存储模型的数据库以键值对作为数据的组织形式和存储方式。一个数据项由数值型的键（Key）和它对应的值（Value）组成。键值数据库中基于键的查询和更新都非常高效，适合数据结构简单、存在高并发读写以及数据规模庞大的应用场景。

Voldemort是一个开源的键值数据管理系统[23]，数据的多个副本分布存储在各个节点中，每个副本都能够进行读写。Voldemort 为了保证高可用性而放弃了数据的强一致性，采用异步更新副本的方式，并为更新提供基于向量时钟（Vector Clock）的多版本并发控制。虽然不保证从不同节点上能够获得一致的数据，但Voldemort 的写修复（Read-repair）机制能够确保如果用户去读大多数副本，就一定能够读出最新的数据。

Riak[24]是一款分布式数据库，Riak采用单主副本更新的方式，异步将更新数据复制到其它副本中。通过设置不同节点上必须成功响应读写操作的副本的数目来实现一致性的可调性，同时能够自我修复不同步的数据。Riak使用一致性哈希（Consistent Hashing）[25]对数据进行分布存储，如果出现节点故障，数据会被自动迁移到其他节点。

Redis[26]的所有的操作都是在内存当中完成的，这使其有着优异的性能。支持向磁盘中以全量数据和操作日志两种形式转储内存数据，实现了一定程度上的容错性。Redis 集群采用主备模式，支持多个备节点，并使用异步复制方式。Redis

哨兵（Redis Sentinel）用来实现Redis 的高可用，通过在每个节点上部署Redis哨兵，可以实现故障检测、故障恢复和主备切换等功能。这种方式实际上是通过外部手段来实现高可用，比较依赖于Redis 哨兵的可用性。

DynamoDB[27]是由Amazon 发布的分布式键值存储系统，采用去中心化架构，能支持多点更新和数据最终一致性，并采用NWR 策略由用户决定读写的一致性级别。多节点更新会产生大量数据冲突，DynamoDB使用向量时钟和读时协调来解决冲突，保证写入操作的高可用。但是，DynamoDB 为了实现去中心化，引入了多种负载的处理机制，并且由于真实环境下服务器时钟不可能绝对一致，因此向量时钟来协调冲突数据存在天然缺陷。

（2）列簇式存储数据库列簇式数据库系统中，数据按列分别存储，每一列存有相同类型的数据，能够实现较高的压缩比。虽然类数据库也以表的形式组织数据，但不能进行关联（Join）操作。这类系统比较适合聚合、分组等对于列操作较多的应用，特别是针对列的查询场景，具有很高的查询性能。

Bigtable[28, 29] 是Google 设计的分布式存储系统，可以将数据依据主键切分，并分布存储在子表服务节点（Tablet Server），能够在海量数据上的实现高吞吐量和低访问延迟，具有很高的可扩展性。在Bigtable 架构中，由一个总控制节点来管理所有的子表服务节点，同时存储着数据的分布信息。显然，Bigtable 在总控节点上存在单点故障问题，而且单总控节点也限制了Bigtable 能支持的最大节点数。

HBase[12]是Bigtable的一个开源实现，其底层架构在Hadoop HDFS 之上，支持以表的形式组织数据。HBase能够自动将文件进行切分并存储在多个节点上，复制时保证数据的强一致性，避免单个节点故障而丢失数据。虽然HBase 本身可通过Zookeeper 实现集群控制和管理，但是它所依赖的HDFS 仍然存在NameNode 的单点故障问题。另外，强一致性更新使得HBase 的性能不高。

Cassandra[30] 是facebook 的开源分布式数据存储系统，由许多分布式数据库节点组成。它在设计时与DynemoDB 相互借鉴，架构上比较类似，例如同样使用无中心节点架构，不会存在单点故障问题，并且使用了提示移交（Hinted

Handoff）技术恢复从故障中修复的节点数据，可用性高。Cassandra 支持多节点写入，数据的每个副本都是平等的，都可以将数据复制到其他节点当中，实现最终一致性，具有非常高的写性能。然而，Cassandra 不适合超大文件的存储，并由于读机制的复杂性，其读性能而受到一定限制。

2.2.2 NewSQL数据库系统

NewSQL类数据库在存储架构上具有NoSQL的特点，不同的是NewSQL 可以支持SQL语法和事务处理，例如ClustrixDB[31]、MemSQL[32] 等。本文将介绍其中几种具有代表性的NewSQL数据库。

Google Spanner[4] 是全球性的分布式可扩展数据库系统，支持多数据中心部署。Spanner 采用半关系模型，数据库中的表被视作主键列到非主键列的映射关系，同时还支持类SQL 的查询语言和包括分布式事务在内的事务处理。另外，Spanner的每份数据都有多个副本，并采用Paxos[33] 协议实现日志复制。而对于跨数据中心的事务，也需要在数据中心级别也实现Paxos 来确定主副本并完成日志复制。

VoltDB[34]是Michael Stonebraker等人研发的可扩展NewSQL 关系型内存数据库，支持标准SQL 访问和ACID事务模型。VoltDB 采用非共享（Shared-nothing）存储架构，使用k-safety技术实现VoltDB 集群高可用。k-safety 中的k表示数据已配置的副本个数，当k 设置为0 时，表示没有配置副本，此时一旦有节点出现故障，将导致整个集群不可用；而当k设置为1 时，表示配置一个副本，使集群能容忍单个节点的故障。数据的副本之间没有主备关系，每次写操作都会发送给所有的副本。

OceanBase[35] 是阿里巴巴集团开发的支持海量数据存储的高性能分布式数据，将分布式存储和关系型数据库功能紧密结合在一起，面向互联网级应用，能够满足互联网行业对于海量数据存储和查询的需求。在设计上，OceanBase采用了读写分离架构，将数据更新和存储使用不同节点处理，由一个总控节点管理集群。

OceanBase 通过部署主备两个总控节点，使用Linux HA实现其高可用。但这中方法过于依赖第三方工具，一旦主备节点间网络中断，会使HA 机制失效。

2.3　可扩展数据管理系统的理论基础

可用性是评估数据库稳定性的重要指标，指在某段时间内，数据库系统正常运行的概率，或者说是它有效服务时间的期望占比。在分布式环境下，对于互联网应用来说，为了获得高性能和高可用性，牺牲了事务一致性；而对于金融行业的关键应用来说，必须保证事务一致性，同时要求数据库具有高可用性。本节首先介绍了数据库可用性的基本概念和衡量标准，然后描述了分布式一致性与事务一致性方面的理论，作为本文工作的理论背景。

2.3.1　数据库可用性的概念和衡量标准

数据库可用性指数据库系统持续提供服务的能力，是评价数据库的重要性能指标。可用性可以被衡量，一种常用的方法是使用一年内数据库正常工作时间所占的百分比[8] 来度量。高可用性的数据库应能够在很大程度上保证服务的持续性和可恢复性。影响数据库可用性的因素包括三个方面[36]：

- **计划服务停止时间：**计划服务停止时间是使用者主动停止数据库服务的时间长度，需要被安排在负载最小的时段，以尽量减轻对数据库可用性的影响。利用这段时间，可以对数据库进行例行检查、维护或升级，也可以用来检验系统的高可用性和容灾能力，提前发现可能对系统产生威胁的问题，以帮助确保数据库系统的稳健性。

- **非计划服务停止时间：**非计划服务停止时间指因各种不可预测的故障而导致的数据库服务中断时间，这些故障包括硬件故障、系统异常和用户误操作等。

- **灾难恢复能力：**实际应用中，数据库系统可能因地震、火灾等自然因素而

被整体破坏。灾难恢复能力指在这种情况下，数据库能否保护库中存储的数据，并且迅速恢复服务。

数据库的可用性还可以依据可靠性和可修复性进行计算。可靠性指系统在一定时间内出现错误或故障的概率，概率越低，可靠性越高。可靠性使用平均故障间隔时间（Mean Time Between Failure, MTBF）度量；可修复性则指当故障发生时，系统修复所需要的时间，使用平均修复时间（Mean Time To Repair, MTTR）来度量。

可用性A （Availability）的计算公式如下：

$$A = \frac{MTBF}{MTBF + MTTR}$$

根据这个计算公式，通过增加平均故障间隔时间（即提高可靠性）或者缩短平均修复时间，都可以获得更高的可用性。

2.3.2 CAP 定理

展现可用性与分布式一致性紧密相关的一个重要理论分析结果是CAP定理[37]。2000 年，Eric Brewer 提出了该定理，揭示了分布式系统的基本准则，可以用来指导相关系统的设计工作。Gilert 和Lynch 则从理论上证明了CAP 定理正确性[38]。CAP 代表着在设计和部署分布式系统时的三个核心需求：一致性（Consistency）、可用性（Availability）和分区容错性（Partition Tolerance）。

其中，一致性指分布式一致性，或者称作数据一致性，即数据的所有副本是否在任何时刻都拥有同样的值。文献[39] 又将一致性模型分为强一致性、弱一致性和最终一致性。这三种一致性模型的区别在于，强一致性模型要求数据的更新要等待所有副本全部完成才能生效，而其它两种模型则只关注主副本是更新完毕。显然，弱一致性和最终一致性模型的能实现更高的写性能。CAP 定理中的可用性指的是，当分布式系统中节点的发生故障时，系统还能否正常运行并对读写请求作出响应。而分区容错性则表示系统应确保在网络异常时仍能正常使用，除非整个网络全部故障。

CAP 定理认为在分布式系统中，上述三个需求无法兼顾，最多只能较好地满足其中两个。在分布式环境中，节点间都通过网络相互连接和通信，网络延迟、丢包等异常情况不可避免，网络分区时有发生，因此分布式系统必须要具备分区容错性。这使得系统设计者只能在可用性和一致性间作出权衡。显然，对于强一致性系统来说，节点的故障会使数据的某个副本不可用，从而导致该数据不能再被更新，因此，强一致性的系统难以具备较高的可用性。而要实现高可用，系统就必须能够容忍数据副本失效，这些系统都是采用弱一致性或最终一致性模型。

2.3.3　ACID 模型和BASE 模型

传统关系型数据库系统采用ACID[40]模型，以实现对于事务的支持。ACID代表数据库事务必须具有原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）这四个基本属性。其中，原子性表示一个事务涉及的操作要么全部成功，要么全部失败，如果事务执行时出错，已经完成的部分要恢复到事务执行前的状态；一致性则指事务如果执行成功，讲使数据库从一个一致状态进入另一个一致状态，则所有对于新状态的访问结果都是相同的；而隔离性指事务能够独立执行，不受其它并发事务的影响；最后，持久性指事务执行成功后，对于数据的所有修改会始终存在，即使数据库发生故障也不会丢失。支持数据多副本冗余存储的数据库如果要实现ACID 模型，会采取强一致性数据复制，因为只有保证数据各副本始终一致，才能确保事务的持久性。正因为如此，这类系统的可用性和写性能都不高。

需要指出的是，CAP中“C”和ACID中的“C”虽然都表示一致性。这两者的意义有所区别，前者表示的是分布式环境中数据多副本间的一致性，而后者指事务开始和结束时应满足的一致性约束。Peter Bailis认为，对于分布式数据库系统来说，可以同时实现高可用性和事务一致性[41]。这是本文能够在具有事务一致性的数据库管理系统ScaleDDB中，实现高可用方法的理论依据。

对于包括NoSQL和NewSQL等在内的大型分布式数据管理系统，其所面对的

是海量数据存储和高并发访问的应用场景，要求很高的系统吞吐量和读写性能来满足应用需求，而对于数据一致性没有太高要求。显然，ACID模型并不适合这种场景。所以，这类分布式系统多采用BASE[37]模型来设计和实现系统。BASE 模型相比于ACID 模型，更强调系统可用性，而非数据一致性。Brewer 在文献[37]中介绍了ACID 和BASE 各自的特点。BASE 这个术语表示基本可用性（Basically Available）、软状态（Soft state）和最终一致性（Eventually Consistency）。基本可用性要求系统以CAP 理论中的可用性为最优先考虑，尽可能保证系统的高可用性，期望能够在各种故障场景下都不会对用户访问造成影响。显然，强一致性模型是无法使系统达到这个要求的，因此BASE采用最终一致性。最终一致性表示系统在接收到数据更新请求后，各个副本能够在一段时间后达成一致，这段时间被称为“不一致窗口”。不一致窗口内，系统会执行复制机制，使所有副本达成一致，这个过程中系统状态可能随时发生变化，即所谓“软状态”。

ACID 模型和BASE 模型都是设计数据库系统时可以参照的标准，需要根据具体的需求进行权衡和选择。BASE 模型使用最终一致性，决定了它无法满足关键性应用对于一致性的严格要求，对于这类应用来说，强一致性的ACID 模型更为合适。然而，ACID模型虽然在集中式架构下可以有效解决一致性问题，但在分布式环境中，如果出现跨节点事务，网络分区会导致事务处理受到影响。目前，越来越多的关键性应用，如在线交易、银行转账等面对海量数据存储和处理压力，开始尝试使用分布式数据库作为底层数据管理系统，这就对分布式数据库的可用性和一致性都提出了较高的要求。

ScaleDDB采用读写分离技术，通过将集中式事务处理和分布式数据存储进行有效结合，在分布式系统中的实现了ACID。在这种架构下，大规模数据存储问题得到了解决，而系统的可用性和一致性问题转移到了总控节点和事务处理节点这两类关键性节点上。本文的研究目的就是在ScaleDDB 实现事务一致性的情况下，解决其中关键性节点的高可用问题。

2.4　高可用技术实现

高可用（HA, High-availability）技术是提高数据库可用性的一套解决方案，包含磁盘存储、数据库系统和应用系统等多个方面一系列技术。实现高可用的基本是冗余，包括数据冗余和服务冗余。只有在存在冗余情况下，数据库中节点故障时才能找到其它节点继续提供服务，因此是这是实现高可用关键。高可用技术包括独立磁盘冗余阵列（RAID）、数据库复制（Database Replication）和高可用集群（HA Cluster）等[36]。

2.4.1　RAID技术

RAID 技术是将多个普通磁盘组合成一个逻辑上统一的大容量高性能磁盘阵列，实现了数据冗余存储，并提供容错功能，保证了集中式数据库的数据安全。RAID通过组合小容量磁盘来实现大数据量的冗余存储，能够在不影响服务器运行的情况下，自动处理磁盘故障并修复数据。条带化（Striping）、镜像（Mirroring）和奇偶校验（Parity）是RAID 中三种关键技术。其中条带化指数据被存入磁盘阵列前需要进行切片，以实现将数据请求分散到多个磁盘中，提高了读写性能；镜像指磁盘阵列中数据在不同磁盘上的复制过程，可以提高读性能；奇偶校验则用于数据错误检查，也可用来进行故障恢复。

标准RAID 根据对于读写性能和可靠性做出的不同权衡，分为RAID0-RAID6共七个级别，每个级别的可靠性、性能和使用场景不尽相同，可以根据需要选择不同的RAID 级别。例如RAID1 具有高可用性和数据安全性，但成本很高；RAID5 是一种较为平衡的方案，兼顾了可用性、读写性能和成本，通过奇偶校验修复数据。

2.4.2　数据库复制

数据库复制是一种通过网络将数据共享到不同位置数据库的技术，可以拷贝全部数据，也可以仅复制数据的更新部分。实现数据库复制机制需要多个数据库，

直接对外服务的数据库被称为主库，其维护的数据为主副本，而其它的则分别称为备库和备副本，可以在必要的时候接替主库工作。数据从主库被复制到备库，使主备副本能够基本保持一致，因此，当主库发生故障时，备库能在短时间内接管主库继续服务。所以说，数据库复制技术缩短了系统的平均可修复时间，提高了数据库系统可用性和容错性。

数据库复制有三种不同方法[42]，即快照复制、合并复制和事务复制。快照复制是将某时刻整个数据库状态，即该时刻的全部数据以文件形式复制到另一台服务器中，这种方式是周期性进行的，适用于数据更新不频繁的场景，并且要能够容忍复制结束后副本仍可能不一致。合并复制指在多台服务器能够自主提供读写服务的场景下，数据库能够将各个服务器的不同的更新进行合并，解决冲突，最终统一数据更新的内容。这种方式会带来较高的延迟，因为解决更新冲突需要一定时间。事务复制指复制数据的更新操作，即事务日志，是目前大多数数据库实现进行复制时所采用的方式。事务复制同快照复制一样都是单向的，但它能够使副本间不一致的时间较短，适用于存在大量更新的场景。

2.4.3 高可用集群

高可用集群中包含两个或两个以上的服务器节点，每个节点都拥有独立的存储磁盘，并通过高速网络相互连接和通信。服务器故障会导致连接其上的应用和存储的数据失效。集群通过数据冗余和服务备份等技术，使集群内节点按照某种策略相互备份，并在发生故障时迁移业务，尽可能缩短集群服务中断的时间，从而最大程度减轻了对业务的影响[43]。高可用集群有多种配置模式，不同模式在包含的节点数目和互备关系方面有所差异，适用的场景和部署难度也不同，常用的配置模式包括[44]:

- **主备式：** 主备模式指在两个或多个功能相同的节点中，只有其中一个活跃的主服务器节点对外服务，而其它的则作为主节点的备份。主备模式用来实现数据或服务的备份，是常用的高可用集群部署方式，可以通过共享存储或

数据复制实现一致性。分布式数据库中同一份数据会配置多个副本，副本间的关系符合主备模式。当数据库进行数据更新时，首先会修改主副本，随后主副本根据某种复制策略将数据更新复制到其它备副本。

- **对称式：** 对称模式中，不同节点能够同时对外提供不同的服务，可以处理相应的请求，同时两个节点也互为备份，如果单从服务的角度看，对称模式实际上就是两个主备模式的结合，使服务器资源得到充分利用，不像主备模式中只能有一个服务器节点对外服务。

- **集群式：** 集群模式混合了主备模式和对称模式，在多个节点上部署多种服务，并互为备份。例如将不同的服务部署到两个节点上，而其它节点被配置为它们的备节点。

当高可用集群的主节点故障时，应从其它正常的节点中选择一个作为新的主节点，代替原主节点提供服务，以恢复整个集群功能。此时，如何选择主节点成为一个问题，本文将这个选择的过程称为主节点选举。主节点选举的重点是要保证选出的节点不会改变集群状态，并尽量兼顾选举效率。选举可以分为自决式选举和待决式选举，自决式选举指通过参选节点间的协调自主推选出主节点，这类算法大都是基于Paxos的，即每个节点都可以主动发起选择某个节点为主节点的流程，如果得到多数节点的认可，就能选举成功，例如Raft[45]、Bully[46] 以及Zookeeper 的ZAB 算法[47]，另外Redis[48] 也实现了类似的选举算法；而待决式选举则需要第三方节点来从参选节点中选择主节点，可以根据具体情况使用节点编号、时间戳等作为选举的依据。

2.5　本章小结

本章介绍了可扩展数据管理系统，以及高可用性的相关工作。首先，本文总结了支持海量数据存储访问的分布式数据库系统所具有的三个特点：主从模式、读写分离和横向扩展，并提出一种典型架构的可扩展数据管理系ScaleDDB；然

后，分别介绍了常见的NoSQL和NewSQL的大型分布式数据管理系统，并分析了它们在可用性和一致性方面的特点和问题；最后，本文介绍了分布式领域的系统设计准则和模型，以及高可用技术的研究现状。通过本章的系统性介绍，为本文工作提供了充足的背景知识和理论依据，为下文介绍具体的研究工作做好了铺垫。

第三章　主总控节点选举

本文在第二章描述的以ScaleDDB为基础的数据库管理系统，支持事务处理和高并发读写，具有很强的可扩展性。在图2.3所示的ScaleDDB 的架构中，使用单个TNode 节点负责事务处理，避免系统产生分布式事务；同样地，也使用单节点CNode 作为总控节点进行全局性管理。显而易见，作为系统集群中最关键的两类节点，分别只部署在一台服务器上存在很大风险。即使使用小型机等高端服务器来部署，“单点故障”问题也始终存在。而ScaleDDB 将所有节点部署于廉价PC 服务器上，与高端机器相比故障率更高。因此，单点TNode 和单点CNode是ScaleDDB 数据库最大的隐患，一旦节点失效，数据库将面临瘫痪。本章首先介绍ScaleDDB 本身实现的高可用和容错机制，分析其存在的问题；然后描述并分析了一致性算法Raft 的优劣；最后提出了一种适应于ScaleDDB 系统的分布式选举算法。

3.1　问题描述

高可用集群技术是实现分布式数据管理系统高可用的重要手段。ScaleDDB 的接口服务节点INode 和存储节点DNode相互间无主从关系，都由CNode节点管理。集群中通常部署多个INode/DNode节点，个别节点的故障基本不会对数据库产生影响。因此在设计ScaleDDB高可用方案时，并不考虑这两类节点。

图3.1中展示了ScaleDDB使用集群技术的一种高可用架构。采用一主多备的集群模式配置两个或两个以上TNode，来实现增量数据备份和TNode 功能备份。主TNode 节点收到事务请求时，如果数据发生改变，则会以操作日志的形式将数

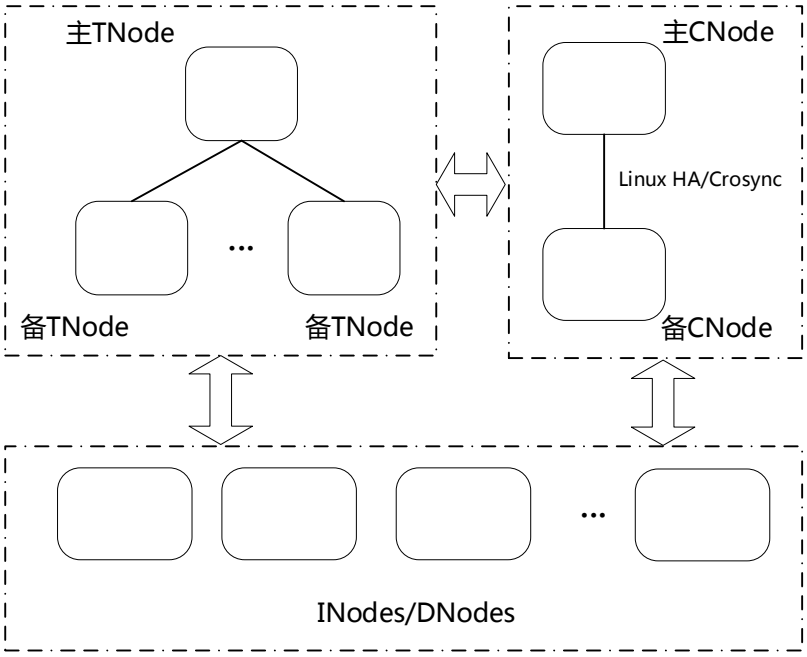

图 3.1: ScaleDDB高可用集群架构

据更新操作复制到其它备TNode 节点中，保持主备间数据一致。由于总控节点的存在，可以由CNode 节点根据某种策略来选择主TNode 节点，这部分内容将在第四章介绍。而对于CNode 节点，采用主备式架构实现高可用。将两个CNode 组成高可用集群，使用虚拟服务器地址（Virtual Internet Protocol, VIP）供外部访问，VIP 所在的服务器即为主节点。使用VIP 的优势在于，高可用集群内不论主节点在哪台服务器上，对外的地址始终不变，外部应用不需要重新连接新的服务器地址，极大降低了应用程序连接CNode 的复杂度。另外，ScaleDDB本身也提供了一定的容错机制，如通过数据多副本冗余存储在不同DNode节点上，使基线数据能容忍小于副本数的节点故障。TNode 节点则实现了数据从内存到磁盘的转储机制，以防止内存失效时丢失过多增量数据。

与TNode不同的是，CNode 作为总控节点，没有更高层的节点可以执行和控制其选举流程。ScaleDDB使用两节点互备的架构，当主CNode 不可用时，只有唯一的备节点可以选择，不需要进行主节点选举，此时需要做的仅仅是检测

到这一问题，并以迁移VIP 的方式直接将另一个CNode 节点提升为主节点。在高可用集群中，可以通过Corosync 等软件实现上述功能，这样的软件称为监控器（Monitor）。监控器通过“心跳机制”检测主备CNode 的状态，能及时发现主CNode 节点的异常，并转移VIP 到备节点上，实现了故障转移（Failover）。

CNode的高可用方法引入了监控器这一数据库系统之外的角色来负责故障检测和故障转移，实际上就是将CNode的可用性与监控器的可靠性绑定起来。只有监控器正常工作，才能使CNode基本避免单点故障而引发的问题，这种方法属于待决式选举。待决式选举必须依靠稳定的第三方节点才能得到正确的结果，然而在实际生产环境中，监控器也是通过软件实现的，受到服务器负载和网络的影响可能会出现问题。因此，对于CNode 来说，应该寻求一种自决式选举方法来实现高可用。另一方面，由于ScaleDDB 部署在普通商业服务器中，服务器故障率较高，所以需要进一步使CNode 能容忍多节点故障。

3.2　Raft协议及分析

在分布式数据库管理系统，特别是类似于ScaleDDB架构的系统中，使用基于消息传递模型来实现节点间的通信。消息通过局域网或广域网进行传输，难以避免地会发生各种错误，例如消息冗余、丢失、网络延迟等。即使网络中没有异常出现，节点在发送或处理消息时也可能产生问题。总的来说，分布式环境下，跨节点通信的不可靠使得节点间难以保持状态或数据的一致，而如何解决这个问题，是分布式领域中要研究的重要内容。

两阶段提交协议（2PC）[49]是解决分布式事务的一种算法，可以实现分布式事务的原子性提交，同时也能用来在数据复制过程中保证副本间数据的一致性。Leslie Lamport 提出的Paxos算法[33] ，以及Vertical Paxos[50]、Fast Paxos[51]等改进的Paxos 算法，是分布式领域中非常重要的分布式一致性协议，被越来越多分布式系统用来解决数据复制、分布式选举等问题。例如，Google 的分布式锁服务系统Chubby[52]，Zookeeper 的核心协议ZAB[47]都是类Paxos 算法的实现。然而，在

现实中实现Paxos算法具有较大难度，Chubby 和ZAB事实上都对Paxos 原始算法进行了一定程度上的简化和改造，

Raft[45] 是一种基于Paxos 的分布式一致性协议，相比于Paxos，它更加容易理解和实现，并且能够保证正确性和算法性能。Raft 在最初就是从工程的角度设计的，充分考虑了在现实中实现时可能存在的问题，并予以优化或规避。使用Raft实现高可用的数据库集群出现故障时，只要集群内仍存在多数个节点，就能保证其可用性和数据一致性。本文在研究可扩展分布式数据库系统的高可用方法时，参考了包括Raft在内的很多算法来设计主节点选举协议。下面本文将对Raft 算法进行简要描述，并通过分析指出Raft 在应用于类ScaleDDB的分布式数据库中时存在的问题。

3.2.1 Raft协议概述

Raft将分布式数据库系统的节点进行抽象，称为复制状态机（Replicated State Machine）。假设状态机的初始状态相同，在系统运行期间，只要每个状态机按照完全相同的序列执行同样的操作，那么它们的最终状态就是一致的。如果映射到数据库中，操作即事务日志（Transaction Log），操作序列即数据库接收并回放日志时的顺序，当数据库节点正确回放完日志时，就能拥有最新的数据。因此，保证操作（即日志）和执行顺序的一致性，就能保证状态机节点数据的一致性。Raft 解决的就是在这种场景下的一致性问题。

分布式系统在处理数据更新请求时，有两种方式，即单点更新和多点更新，具体如下：

- 单点更新：同一时刻，由集群中单个节点处理所有更新请求，并响应客户端。该节点会将更新后的数据或者更新操作复制到其它节点中以保持数据一致，多采用日志的形式。这种方式虽然存在单点写入的性能瓶颈，但逻辑简单，不容易出错。

- 多点更新：集群中所有节点都可以处理更新请求，可以独自回应客户端请求。与单点更新方式相比，写入性能更高，但是必须解决对于同一数据的不同的修改而导致的更新冲突问题，节点间可能要经过多次交互才能达成一致，实现较为复杂。

Raft为了简化逻辑，方便工程上理解和实现，采用单点更新的方式实现集群节点数据的一致性。在Raft算法中，定义了节点的三种角色状态：

- 领导者（Leader）。在一个Raft集群内，领导者负责集群与客户端的交互，接收读写请求，以及执行日志复制，即传统意义上的主节点。领导者会向其它节点发送包含日志内容的信息，Raft 中称为AppendEntries RPC。若没有写入请求，领导者则定期发送包含空日志的信息，以宣告自己的存在。

- 追随者（Follower）。被动接收领导者发送的日志信息，更新本地数据完成复制，由领导者全权控制。

- 备选者（Candidate）。试图成为新领导者的角色。追随者若想成为领导者，必须首先变为备选者。

每个节点在同一时刻只能担任其中一个角色，但在算法运行过程中角色可以发生改变。集群处于非选举状态时，只存在两种角色的节点，即领导者和追随者。整个Raft 算法的实现流程包括领导者选举（Leader Election）和日志复制（Log Replication）。顾名思义，领导者选举就是通过选举确定集群的领导者，并在后面的日志复制阶段中承担起领导者的工作。本章只讨论领导者选举算法的设计，所以在此不对日志复制进行详述。

由于在分布式环境下，不同的服务器时间不会完全相同，因此如果在选举中使用服务器时间作为比较节点状态新旧的依据并不准确。为了解决这一问题，Raft 引入“任期（Term）”作为判断时间顺序的标识，并赋予自然数编号。时间被切分为若干Term，每次选举开始都代表进入新的任期，任期编号会随之递增。若在选举过程中成功选出领导者，那么该领导者会保留自己的角色状态直到下一个

任期。每个节点都可以维护自己的任期编号，在进行领导者选举时，任期编号将作为选举权重，是决定选举结果的重要因素。

Raft选举算法的本质是多数派决议。节点在选举开始时会向所有节点提出预案，即希望自己能成为新任期内的领导者，各个节点会对预案进行投票，如果该预案获得包括自己在内的大多数（半数以上）节点的认可，那么该节点就能成功当选。选举过程中，任何一个节点都可能收到多个节点的预案投票请求（Request Vote），最终节点会投票给任期编号最大的预案。另外，Raft规定在同一任期内，每个节点只有一次投票权，优先到达的预案将得到选票，而后来的预案将被该节点否决。可以看出，在不考虑日志的情况下，节点对预案进行投票依据两个因素：预案的任期编号和节点已投票情况。

图3.2展示了Raft选举过程中，三种节点角色状态的转换情况，这是一个不确定性有限自动机（Non-deteministic Finite Automation, NFA）。随着选举的推进，节点可能转变为任一角色。下面本文根据该图简要介绍选举算法的流程。

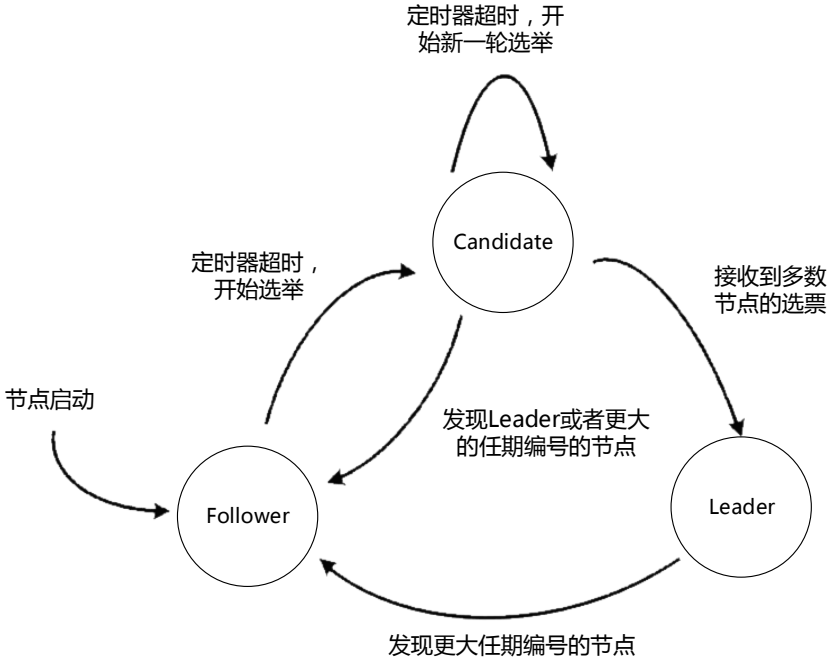

图 3.2: Raft状态转换图

当一个节点从故障中恢复或刚启动时，其角色为追随者，同时启动一个定时器（Follower Timer）。若追随者接收到领导者发送的AppendEntries RPC，会重置定时器，保持追随者角色状态不变。否则，定时器最终会超时，而追随者认为此时集群中没有有效的领导者存在，因此，它将转变为备选者，准备竞争成为新的领导者。

备选者首先提升自己的任期编号，重置自己的定时器，然后向其它节点发出投票请求并等待回复。在这之后，备选者将可能出现三种情况：

- 获得同一任期内的多数派选票，成为领导者。
- 等待期间接收到AppendEntries RPC信息，说明集群中已经有节点当选，此时转变为追随者。
- 如果前面两种情况都没有发生，备选者将持续等待，直到定时器超时，然后重新发起新一轮选举。

节点成为领导者后，便如前文所述开始发送日志信息，并重置其它节点的定时器。只要领导者没有在与其它节点通信时发现具有更高任期编号的节点，会一直保持领导者这一角色。

最后，需要注意的一点是，Raft必须保证任期内当选的领导者存储着已经提交的最新的日志，否则很容易产生数据冲突。Raft使用日志号唯一标识一条日志，由领导者节点负责生成日志号，日志号越大则说明日志越新。所以，当节点接收到投票请求后，除了判断任期编号，还会比较自己和请求投票者的最大日志号，只有当前者不大于后者时，才投票，否则拒绝投票。

本节介绍了分布式一致性协议Raft，并着重分析了Raft 的主节点选举算法。Raft 抽象描述的复制状态机与ScaleDDB 数据库系统架构相类似，但是又有很大不同：

- 首先，Raft中总控节点和更新节点是同一个节点，因此主节点选举和日志复制都是在同一组状态机上进行的；然而在ScaleDDB 中，这两者是分离的，

总控节点CNode 和数据更新节点TNode是两类不同的节点，需要通过网络来实现交互。

- 其次，虽然Raft宣称其是一种面向工程实现的一致性算法，并且考虑了很多真实场景中可能会发生的问题，但它仍然是构建于理论模型上的，应用场景较为简单；ScaleDDB 则是一个可扩展数据管理系统模型，支持事务处理和并发访问，存储数据规模庞大，应用场景更为复杂。

面对ScaleDDB这种复杂架构的系统时，Raft存在两个致命问题：双主问题（Split Brain）和频繁选举（Frequent Election）。这是Raft算法无法直接应用于ScaleDDB 的来实现数据库高可用的关键原因。下面两节将对这两个问题进行具体描述。

3.2.2 Raft协议问题一：出现双主

所谓双主问题，又称脑裂，即指分布式集群中同时存在两个主节点，亦可进一步引申为存在多个主节点，为了简化问题描述，本文只讨论双主节点。在Raft中，如果出现网络分区，就很有可能导致双主问题。网络分区指集群内的网络由于网络延迟、中断等异常，而使得某些节点失去彼此间的连接，最终使网络被分割成多个子网络，各个独立的子网络内节点连接仍然正常，但是子网络之间无法通信。如图3.3.(a)，Raft集群内包含S_1-S_5 五个节点，节点中显示的数字表示节点所处的任期。此时主节点S_1 与其它节点的网络都出现异常而中断，于是S_1 单独组成一个子网络$Subnet_1$，而S_2-S_5 也组成无主节点的子网络$Subnet_2$。由于$Subnet_2$包含四个节点，已经形成多数派，因此能在接下来的任期内会重新选举出$Subnet_2$的主节点，假设其为S_5，如图3.3.(b)最终，集群中同时存在S_1 和S_5 两个主节点。

Raft算法虽然不能避免双主问题，但是也不会受到该问题的影响，并且当网络分区恢复后，Raft能自动修复双主问题。Raft 的日志复制和提交机制保证了数据的一致性。简单来说，主节点接收到客户端的写入请求后，会将操作日志复制

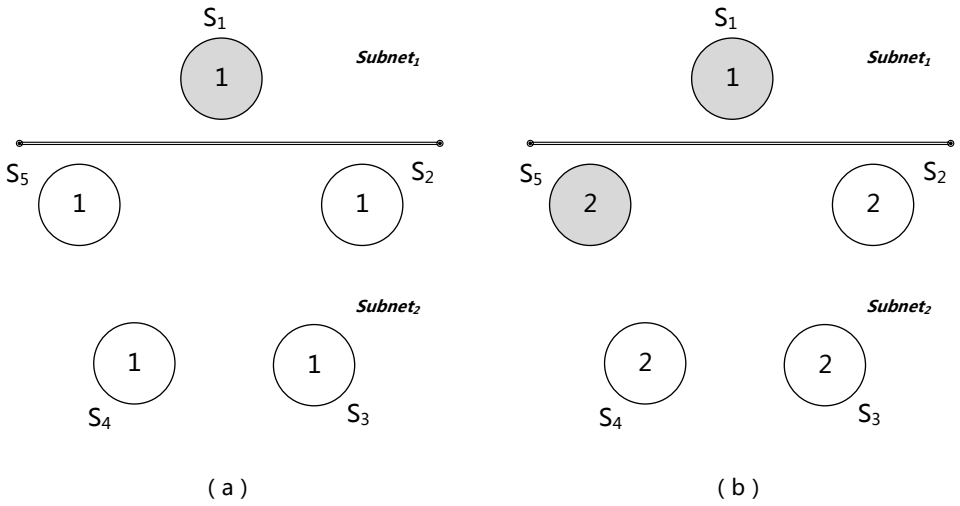

图 3.3: Raft双主问题示意图

到其它节点。只有确认包括自己在内的多数个节点成功接收后，主节点才能提交相应操作，并回复客户端。因此，即使集群中存在两个主节点，也只有处于多数派网络分区的主节点能够有效服务，而如果客户端连接了另一个主节点，则无法成功写入数据。

但是，ScaleDDB的设计架构决定了在此类可扩展分布式数据库系统中不能出现双主问题。ScaleDDB 中，总控节点CNode 并不承担日志复制的职责，它仅仅负责集群层次的管理和协调，以及保存集群的配置信息。从某种程度上来说，CNode 是“无状态”的。所谓“无状态”指多个CNode 之间不存在上下文关系，也没有优先级和存储信息上的不同。如果存在两个主CNode节点，它们会同时进行集群管理的操作，这势必导致集群的混乱和无序，使系统发生各种错误，无法正常服务。ScaleDDB的高可用方案必须要避免出现双主问题。

3.2.3 Raft协议问题二：频繁选举

Raft可能发生频繁进行主节点选举的情况，这也是由网络分区导致的。但是这里的网络分区情况，与前面提到的有所不同，它并没有形成互不通信的独立子网络。以三节点集群为例，如图3.4.(a)。S_3是任期1 内的主节点，此时S_3 和S_2网络

连接失效，但它们和S_1 的网络分别都是连通状态，这种网络分区本文中称作“半分区”。由于S_3和S_1 已经形成多数派（节点总数为3，则多数派数为2），故主节点S_3 可以正常服务，但无法向S_2 发送信息以重置其定时器。当S_2 超时时，便发起任期编号为2的选举。此时，考虑S_3可能存在的两种情况：

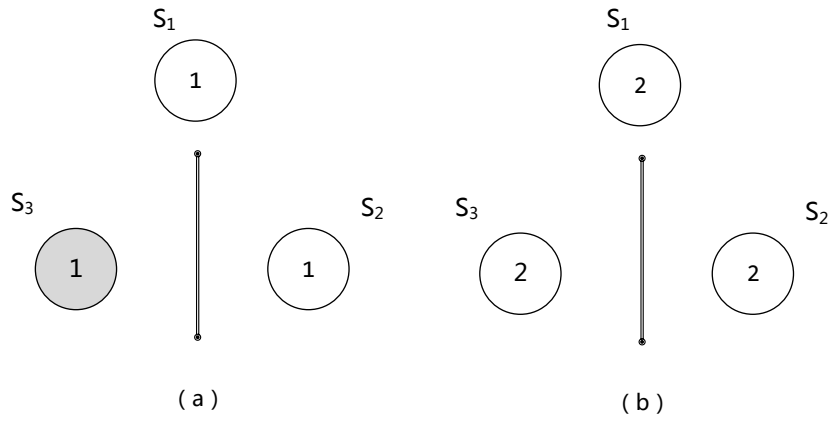

图 3.4: Raft频繁选举示意图

假设S_3作为主节点期间，已经处理过写入请求，并将日志复制到S_1 中。S_2 在网络分区发生后一直无法接收到S_3的日志信息，无法监测到主节点存在。当其定时器超时，就会开始选举并发起投票请求。此时，因为S_2 没有保存最新的日志，所以S_1 肯定会拒绝投票。但在Raft 中，任期编号具有传递性，S_1 发现更大的任期编号时，将更新自己的编号。而当它回复S_3 的AppendEntries RPC信息时，同样也会把新的任期编号传递给S_3，最终导致作为领导者的S_3 因发现到更大的任期编号而转换为追随者。这个时候，集群变成无主状态，如图3.4.(b)。此时系统不能向客户端提供服务，当有节点超时，又将开始新一轮选举。只要S_3 或S_2 成为主节点，并且分区仍存在，那么集群还会再一次回到无主状态，循环往复，发生选举频繁。

假设S_3作为主节点期间，没有接收过写入请求，此时各个节点在保存的日志上没有区别。当S_1 接收到S_2 的投票请求时，将同意并投出选票，于是S_2 当选为主节点，此时集群中也出现了双主节点的情况。S_3还会继续向S_1 发送AppendEntries

RPC信息，但是S_1 会把更高的任期编号传递给S_3 使其改变角色状态。同样的，S_3过期后又会发起选举，在半分区存在的情况下会重复前面描述的情况，导致频繁发生选举。只有当S_1当选为主节点后，由于其它两个节点都可以接收到S_1 的信息，所以不会再因超时而发起选举，集群也能稳定运行。

ScaleDDB作为支持高并发读写的可扩展数据管理系统，不仅要保证集群中只能有一个总控节点，而且要尽量避免出现没有主总控节点的情况。另外，ScaleDDB 在切换总控节点时，并不像Raft中那样只需要改变下角色状态即可，还有更多的工作，例如启动和关闭各种服务线程，重新加载配置信息等，使得切换的时间远远超过选举时间。切换期间总控节点不能正常服务，影响了整个系统的功能。因此，为了使ScaleDDB 能够正常而稳定地运行，就必须保持总控节点的稳定，

本节通过结合ScaleDDB架构对于Raft存在问题进行分析，明确了在ScaleDDB架构的数据管理系统中设计和实现主节点选举算法应注意的关键问题，对本文提出的选举算法具有一定的启发意义。

3.3　LEBR选举算法

本文提出ScaleDDB类型的可扩展数据管理系统的选举算法LEBR （Leader Election Based on Raft），用以实现总控节点的高可用。在设计算法时，本文考虑了ScaleDDB 的总控节点在实际运行时所面临的复杂场景，保证LEBR 算法不会导致双主问题或频繁选举问题，能在较短时间内实现故障自动回复，并使总控节点的选举基本不会影响数据库的正常使用。

在描述算法的时候，本文仍沿用了Raft的术语，例如任期、三种角色状态（领导者、追随者、备选者）。这样做的原因首先是由于本文提出的LEBR选举算法受到Raft 的启发和影响，沿用Raft 术语在情理之中；其次，前文在介绍Raft时对这些术语的含义和作用进行过具体描述，因此LEBR 在定义类似概念时使用同样的术语，能够使算法流程更容易理解。

3.3.1 LEBR基础

在ScaleDDB中，总控节点并不负责任何数据更新、复制的操作。虽然该节点维护着数据库中所有数据的分片位置信息，但仅仅保存这些信息的一个缓存，数据实际上分布存储在DNode 节点中，总控节点可以从DNode 中随时获取这些信息。事实上，各个总控节点相互平等，一个备用的总控节点在运行一段时间后，若要代替原来的节点管理集群，它不需要从原总控节点中获取额外信息就可以成功接管集群。因此，LEBR在设计时也贯彻了各个参选节点平等的思想，每个正常的节点都可能竞选成为主节点。LEBR算法基于Paxos的多数派基本思想，并参考了Raft 的部分流程，在一轮有时限的选举中对每个发起投票的节点都进行表决，若某节点的投票请求被多数节点接受并同意投票，那么该节点就会当选。本章接下来的部分中所述节点，若没有特别说明，均指代ScaleDDB 的总控节点。

LEBR算法虽然源于Raft，但是也有着较大的不同，包括以下两点：

- LEBR中节点间关系平等，任何一个节点在选举之初有着相等的当选可能性；而Raft中由于存在数据更新和日志复制，节点间存在状态新旧的区别，状态越新的节点越有可能当选为主节点。

- Raft在某些场景下可能出现双主问题和频繁的主节点选举，这并不会影响Raft 的正确性。但LEBR 解决的是ScaleDDB 总控节点高可用问题，不能容忍这两种情况的发生，因此LEBR算法针对这两个问题对此进行了处理。

LEBR通过对主节点增加租约机制以及限制节点改票，解决了双主和频繁选举问题，但也使选举从一阶段变为了两阶段。LEBR 的备选者节点需要通过连续两次得到多数派认同才能当选主节点，使得选举时间变长。这些问题本文将在3.3.2节和3.3.3节中进行详细介绍和分析，本小节将描述LEBR 算法的基本概念。首先，定义多数派如下：

定义3.3.1. *多数派值M：假设一组节点中节点数为N，N为正整数，则有该节点组*

的多数派值M，满足：

$$\frac{N}{2}+1 \leq M \leq N$$

本文将使用LEBR的若干节点的集合称为LEBR组，只有组中正常的节点数为多数派时，LEBR才能有效运行。因此，若希望能够容忍k个节点故障，则至少需要部署2k+1 个节点。避免“单点故障”，即指要容忍1个节点的失效，因此，为了实现高可用，LEBR 算法需要运行于至少三个节点上，所以ScaleDDB 的总控节点高可用架构也需要至少三个CNode节点支持选举算法。图3.5展示了使用LEBR的高可用架构，三个CNode节点组成一组，其中有且仅有一个主节点管理包括TNode、INode和DNode在内的其它节点。如果希望增加能够容忍的故障节点数，可以通过部署更多的CNode 节点来实现。

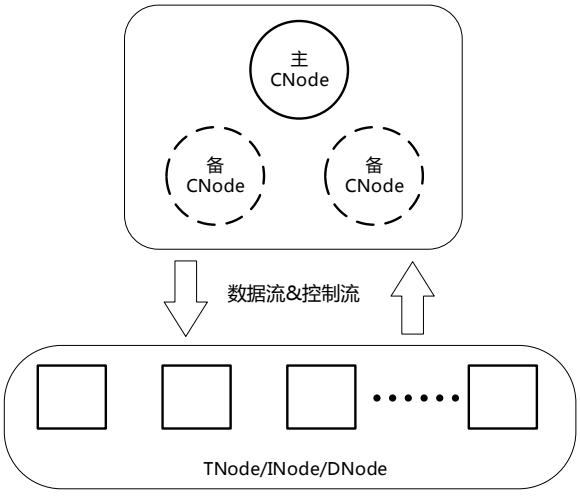

图 3.5: ScaleDDB 三总控节点部署架构图

LEBR将节点在选举过程中的身份分为三种，并延续了Raft中的名称，即领导者（Leader）、追随者（Follower）和备选者（Candidate），统称为逻辑角色。每个角色在选举中的意义和作用与Raft中的类似，不同的一点是，LEBR中的Leader不需要处理客户端的数据读写请求，也不需要执行日志复制等操作，它仅仅作为数据库的主总控节点，对集群实施控制和管理。逻辑角色可以相互转换，图3.6 展示了LEBR 算法运行过程中角色状态转换情况，这也是算法的整体框架和流程。

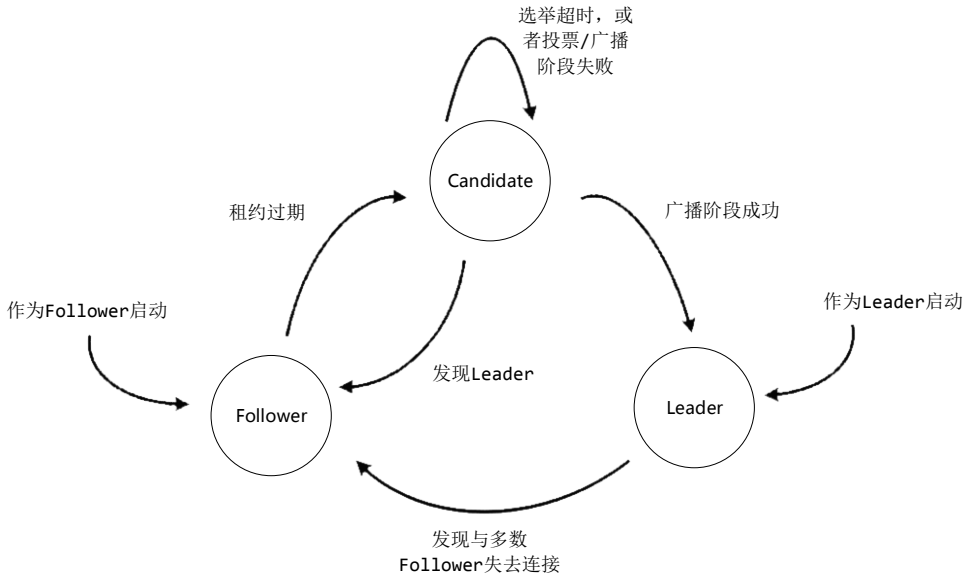

图 3.6: LEBR状态转换图

一个ScaleDDB集群部署的多个总控节点之间存在主备关系，同一时刻只能由主节点管理集群并对外服务，从这个角度看，总控节点的身份还可以分为主管理者（Master）和备管理者（Slave），统称为物理角色。逻辑角色表示的是节点在选举过程中的选举角色，是因LEBR 算法的需要而定义并作区分的；物理角色则表示的是节点在集群中真实的主或备身份，物理角色的改变才真正代表着主总控节点的变更。逻辑角色和物理角色存在这对应关系，如表3.1 所示。

表 3.1: 逻辑角色和物理角色对应表

逻辑角色	物理角色
Leader	Master
Follower	Slave
Candidate	Slave

在系统运行的过程中，如果物理角色发生改变，那么逻辑角色一定也会有变化，反之则并不成立，逻辑角色变化不一定引起物理角色的改变。本文在设计ScaleDDB的总控节点高可用方法时，令其不断检测这两种角色的变化情况，并据此作出及时的处理。例如当逻辑角色由Candidate 转变为Leader 时，相应的物理

角色则从Slave 变为Master，此时对应的总控节点就可以开始切换为主节点接管集群。

本文使用“任期”（Term）表示各个节点的逻辑时间，其值为非负整数，这与Raft 中的意义一样。逻辑时间是真实时间的离散化表示，如图3.7，时间轴以每次选举开始的时刻为界，被切分为长短不一的若干时间段，这些时间段就是逻辑时间，用Term 表示。图3.7 中的阴影部分表示执行选举的时期，选举结束后集群正常运行。但是也有可能发生一个Term内没有选举出主节点的情况，如$Term_2$。此时选举会在接下来的$Term_3$中继续进行。

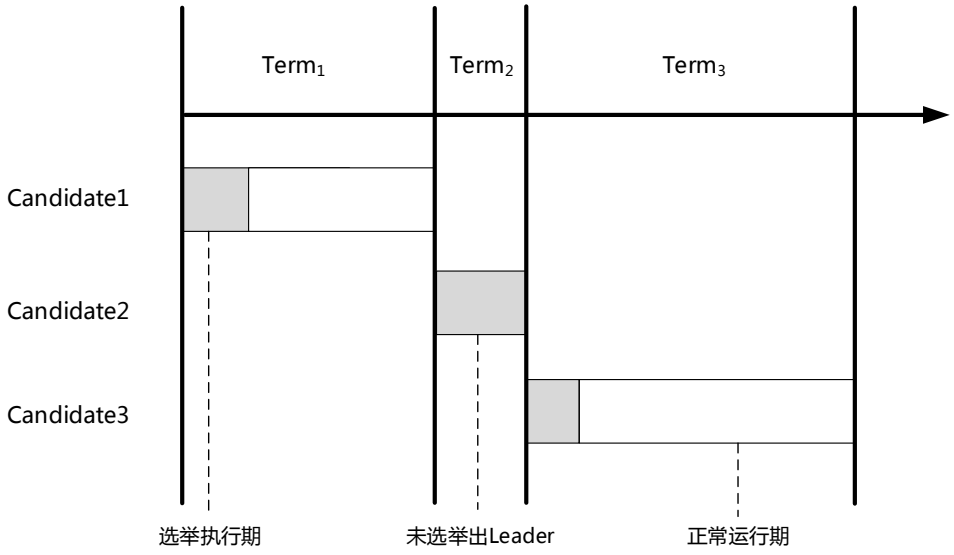

图 3.7: 任期和逻辑时间

显然，Term具有两个明显的特点：

- Term值只能随着真实时间不断增大，而不会减小。
- 每当开始新一轮选举，就会增加Term，使得每次选举时的Term都不同。从这个意义上来说，Term也可以表示节点经历的选举轮次。

在选举过程中，节点都维护自身的Term，根据自己的运行情况确定，可能发生同一时刻不同的节点的Term 不相等的情况。但是，选举结束后，如果成功

当选的Leader在所有节点中具有最高的Term，那么集群中所有节点的Term 都会与Leader 保持一致，这也遵循了前文提到的ScaleDDB 总控节点间关系平等的设计思想。更重要的是，相同的Term 使节点在下一次选举开始时，具有相同状态，这样每个节点都有当选成为Leader 的可能性。另外，LEBR 算法不仅确保了一个Term内最多只能选举出一个Leader，同时也进一步保证了真实时间上不会同时出现多个Leader。

3.3.2 LEBR非选举期

LEBR选举算法将ScaleDDB的运行周期分为交替的两个时期：非选举期和选举期。数据库在绝大部分时间里都处于非选举期，此时可以正常使用数据库服务；只有当节点或网络发生故障而导致主总控节点失效，才可能进入选举期。选举期内数据库不保证能执行全部功能，处于异常状态。选举期时间越短，ScaleDDB 的可用性越高。本节将介绍在非选举期LEBR 的处理机制，而在下一节介绍LEBR在选期的执行过程。

ScaleDDB的非选举期即为无故障运行期，在此期间，集群中的一组总控节点内有且仅有一个节点逻辑角色为Leader，即主节点，其它的节点均为Follower，即备节点。LEBR引入了租约机制（Lease）和心跳机制（Heartbeat）来尽可能保证非选举期各个总控节点状态不变，使主节点稳定地承担起集群管理责任。

（1）租约机制

非选举期内，节点的逻辑角色并非一成不变的，而是在一定期限内有效，本文将这段期限称为“租约（Lease）”。ScaleDDB中，租约是通过指定未来某个时间点来实现的，当节点的服务器时间超过指定的时间节点时，就可以认为租约过期。具体定义如下：

定义3.3.2. *租约与租约过期：假设节点在T_1时刻启动并开始运行，此时规定在T_2时刻（$T_2 \geq T_1$）之后，节点将进入选举期，则T_1与T_2间的时间段称为租约，称T_2为过期点。随着节点运行，当时间$T \geq T_1$时，则称节点租约过期。*

总控节点都拥有租约，租约期内，节点能够保持当前逻辑角色不变。由于ScaleDDB作为数据库系统需要处理各种复杂任务，用时较长，因此节点租约时间设置为秒级。初始状态下，节点会为自己设置默认的租约时间。但是在这之后，Leader将统一管理集群内所有节点的租约，其它Follower节点则不能主动更新自己的租约。一旦Leader故障或者与Follower 间的网络中断，就无法延长节点的租约时间。一段时间后，将会有Follower租约过期，并发起选举。此时节点将有机会通过主节点选举改变自己的角色。对于Leader节点来说，可能在选举结束后转变为Follower，也可能维持Leader 角色不变；而对于Follower 节点，则有可能成为集群内新的主总控节点，或者不会发生任何改变。

集群中是否存在唯一的Leader 正常工作是判断数据库系统处于何种时期的标准，而不能仅仅根据是否有节点租约过期而发起了选举来判断。例如，某个Follower 节点因故障而与其它节点通信中断，无法被Leader更新租约，当租约过期后发起了选举。但在有Leader的存在并正常运行时，此次选举将不可能成功，LEBR 的算法保证了这一点。即使该Follower 节点的问题一直存在，也不会影响集群的可用性。

Raft中，Leader一旦被选举出来，除非故障失效或发现集群中存在更大的Term的节点，否则将一直维持Leader 角色不变。但在LEBR 的设计中，Leader不会受到其它节点的Term 的影响，而是增加了租约机制，通过不断延长租约来保持状态。Leader 在维护自身租约时，与维护Follower租约不同。首先，Leader需要满足一定条件才能更新自己的租约，而对于Follower 来说，只要与Leader网络通信正常，就能够被更新租约；其次，Leader的租约时间应小于Follower 的租约时间，这是为了使Leader 在Follower 租约过期发起选举之前就放弃Leader 这一角色，以避免选举过程仍中存在Leader，以保证不会出现双主问题。

（2）心跳机制

在数据库运行过程中，我们期望其一直处于非选举期，尽量不要发生总控节点的选举和切换，以免数据库无法维持稳定的服务状态。LEBR使用心跳机制来

解决这个问题。心跳机制是集群的服务器之间用来维持通信、确认存活以及传递消息的常用机制，也是数据库高可用解决方案中的常见机制。LEBR算法中采用的是一对多的单向心跳机制，如图3.8所示。所谓“一对多”，指心跳维系在一个Leader 节点和多个Follower 节点间，而Follower之间并没有心跳通信；“单向”则指心跳消息只由Leader主动发送，而Follower被动接收。

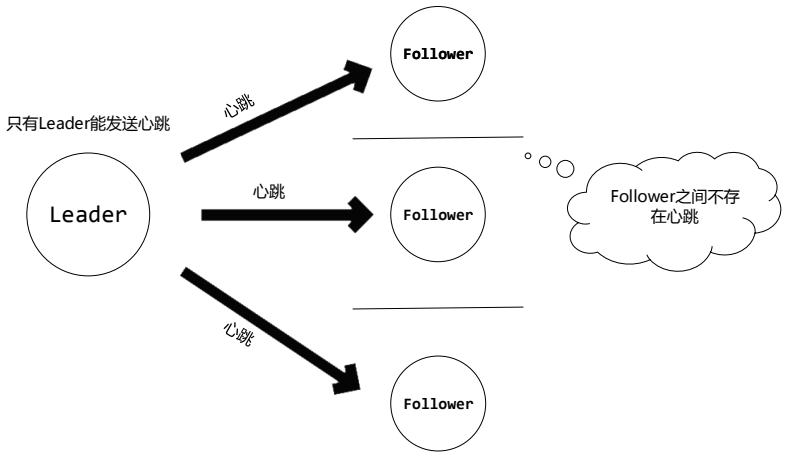

图 3.8: LEBR算法心跳信息交互图

Leader节点通过定期发送心跳延长其它各个节点的租约。只要Leader 一直存在，并且与其它节点保持通信，那么所有节点的租约都将不断被延长，不会发生租约过期，从而保证了整个数据库系统的平稳运行。除了租约信息外，Leader还会将应与主节点尽量保持一致的信息，即Leader 的Term 值，通过心跳传送到Follower 中，使集群中所有总控节点的Term 值尽可能与Leader保持一致，这样在下次选举开始时各个节点的状态基本也是一致的。前文提到Term 值的变化方向应该是递增的，即只能增大而不能减小，所以非选举期内，只有当Follower的Term值不大于Leader的Term 时，才会在收到心跳后更新自己的Term。

LEBR的心跳机制所实现消息传递有一个共同特点，即消息的发送方（即Leader）并不关心消息是否被成功接收和应用。由于故障只是偶尔发生，两次选举期的时间间隔较长。因此，Leader有充足的时间将共享信息发送给正常工作的Follower，使选举发生时这些信息是一致的。最重要的是，共享信息的不一致不

会使算法失效或对算法造成正确性方面的影响。

LEBR算法中规定，在非选举期内，Leader必须与LEBR 组内多数个节点（包括Leader本身）保持通信，该Leader 才能被承认。这是因为Leader通过多数派决议选举出来，如果发现自己与多数节点无法通信，则说明其不被大多数节点承认，也就失去作为Leader 的资格。通过心跳机制，Leader 可以检测这一点。如果发生这种情况，节点将会通过停止更新自身租约的方式放弃Leader身份，租约过期后将切换为Follower。具体的流程如下：

Leader定期向各个Follower发送异步的心跳消息（假设时间间隔为t_{inter}），Follower收到后将回复一个确认信息，Leader会记录接收每个Follower 心跳回复的时间戳。虽然Leader 默认与自身始终是连通状态，并不会给自身发送心跳消息并获得回复，但我们仍赋予Leader 一个名义上的心跳回复时间戳，以简化算法逻辑，并且便于描述。判断Leader与多数个节点失去通信联系的条件是：一段时间内Leader无法接收到多数节点的心跳回复。本文将这段时间称为“失联确认窗口”（Link-lost period）。以包含五个节点的LEBR组为例，图3.9中的时间轴上，T_{now} 表示当前时刻，T_{lose} 和T_{now} 之间即为“失联确认窗口”，而T_1-T_5 这五个时间点则表示Leader 接收到的心跳回复时间戳。可以看出，图中T_3、T_4和T_5 三个时间戳均位于“失联确认窗口”，说明此时Leader在这个窗口内已经收到多数节点的心跳回复，因此可以继续维持Leader角色。

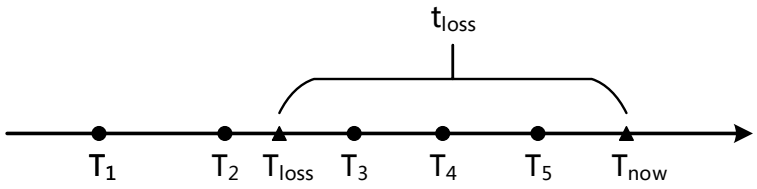

图 3.9: 失联确认窗口示意图

在具体方法上，本文将时间戳进行升序排序。需要注意的是，无论其它节点的时间戳如何排序，Leader 自身的时间戳都默认为最大。排序后，取处于中间位置的时间戳，假设为T_s，同时设“失联确认窗口”的长度为t_{loss}，则多数个心跳回

复的时间戳位于其内的条件是：

$$T_s > T_{now} - t_{loss}$$

前文中提到LEBR中Leader的租约时间（用t_{ld}表示）必须小于Follower的租约时间（用t_{fl} 表示），假设集群中节点间的一次网络传输最大延迟为t_{delay} ，那么它们与“失联确认窗口”关系为：

$$t_{fl} - t_{ld} = t_{loss} > 2t_{delay}$$

当LEBR认为Leader与多数节点失去联系时，Leader的租约也刚好同时过期，那么就可以立即切换身份，使ScaleDDB进入选举期。Leader向其它节点发送心跳后，最多需要等待两次最大网络传输时间（$2t_{delay}$）能收到回复，因此“失联确认窗口”时间长度应大于$2t_{delay}$。

3.3.3 LEBR选举期

当集群中不存在Leader节点时，ScaleDDB无法正常提供服务，将进入选举期，直到Leader被重新选出，才意味着选举期结束。由于选举期的初次选举始于Follower的租约过期，而Leader的租约总是会提前于Follower过期，因此选举开始前会存在一小段空白时期，此时集群中既没有Leader，也没有开始选举。下文将系统地描述LEBR在选举执行期的具体流程。

Follower节点租约过期后，切换为Candidate，并进入选举执行期。在选举执行期内，参与选举的节点有Candidate和Follower 两种，但只有Candidate 能够主导选举过程，并在选举结束后有成为Leader 的可能性；而Follower 只能被动地响应来自Candidate 的请求，可以自行决定是否投出选票。在一段时间内可能有多个Candidate 发起选举，LEBR 使用Term 来区分选举轮次，Term值相同的Candidate属于同一选举轮次，每轮选举设置了最长期限。算法LEBR 允许选举失败，即处于某个选举轮次下的Candidate在选举时间超期前，没有当选为Leader，也没有得

到其它节点的当选信息，那么该Candidate将提升自己的Term，并发起新一轮选举。图3.10 展示了上述的选举流程。

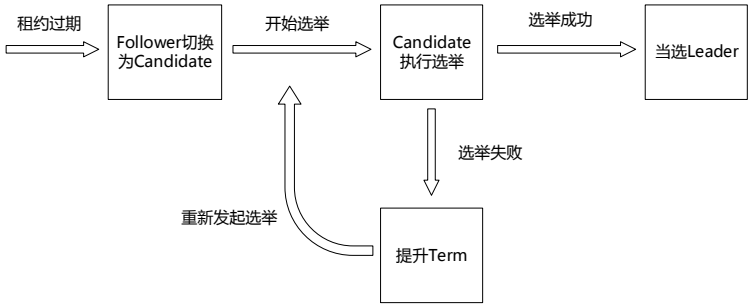

图 3.10: LEBR 选举流程示意图

从Candidate执行选举的角度来看，选举可以分为两个阶段：投票阶段和广播阶段。这两个阶段是顺序进行的，并且无论哪个阶段没有成功通过，都认为选举失败。具体如下：

- **投票阶段**

当Follower租约过期，切换为Candidate后，节点向包括自己在内的所有节点异步发送投票请求（VoteRequest），期望得到选票，请求中会附带Term 信息。投票阶段有固定期限，一般设置为几百毫秒，大于两倍的网络传输延迟。投票超期之前若Candidate 获得了多数派选票，则投票阶段成功，Candidate 会进入广播阶段；反之，说明投票阶段失败，将开始下一轮新的选举。Term 可以代表节点选举的轮次，Candidate收到的选票中也带有Term，其值与Candidate 当前的Term一致时，才认为是有效选票，否则选票视为无效。由于网络中存在延迟，节点可能会收到前面轮次选举中投票请求的回复选票，而使用Term进行区分可以避免这种情况。

- **广播阶段**

若Candidate通过了投票阶段，则说明其与多数节点保持正常的连通状态，并且已经获得过多数节点的认可，可以成为新的Leader。然而，节点接收到投票请求后，在满足一定条件时可以改票，导致可能有另一个Candidate得到多数选票。因此Candidate并不能直接确认成功当选，需要在广播阶段向所有节点异步

发送当选广播（Broadcast）来进一步确认。可以将广播视为第二次投票请求，广播中包含节点的Term。节点接收到当选广播请求，一旦确认后，将不允许改票。若Candidate 获得了多数节点的确认回复，就可以认为在本轮选举中成功当选为Leader，至此整个选举流程结束，同时也标志着ScaleDDB 结束选举期，进入非选举期，恢复正常工作。广播阶段也有一个固定的期限，时间长度与投票阶段一样。超期前若没有得到多数节点的确认回复，说明广播阶段失败，Candidate也将重新开始选举。

前面介绍了Candidate在投票和广播阶段主动竞选Leader 的流程，每个阶段中的核心都是多数派决议。选举过程中涉及的主动请求包括投票请求和当选广播请求，而参与者对两种请求的处理是LEBR中的重要内容，影响选举的最终结果。Candidate 和Follower 对请求的处理逻辑有所区别，并且对于Candidate 来说，在投票阶段和广播阶段处理请求的方式也不完全一致。

另外，选举刚结束时，除Leader外的节点并非全都跟随该新Leader，Leader需要通过心跳机制使所有节点跟随自己。因此，除了前面提到的两种请求，节点还需要处理心跳。事实上，节点接收到Leader 的心跳后，只有Leader的Term值不小于自己时，才认可该Leader并跟随它，否则将不会响应这个心跳消息。这是因为，如果节点跟随了Term比自己小Leader，那么就会导致自己的Term减小，而在LEBR中，除非重启节点，否则不允许节点Term值减小。下文中将介绍不同逻辑角色的节点对于选举中的主动请求的处理流程，显然，Leader可以直接拒绝这些主动请求，而Follower和Candidate 则要根据情况处理。

- **处理投票请求**

如果节点的逻辑角色为Follower，表示其一定正在跟随一个Leader，无论该Leader是否还在正常运行中，至少在租约期内Follower是认为Leader 存在的。LEBR算法中规定，在这种情况下Follower节点不能响应任何选举相关的请求。这样做的原因是使Follower尽可能跟随当前Leader，保持系统稳定。除非当租约过期，确认Leader 失效后才能参与到选举中，此时节点角色已经由Follower转变

为Candidate。

Algorithm 1 LEBR算法的投票请求处理流程

输入： $VoteRequestMsg$：$M(term, lease, requestor)$，接收到投票请求后开始执行算法；

输出： $granted\ or\ null$，granted表示投出选票，null表示不回复任何信息；

```
1: deserialize vote request message，解析接收到的投票请求包内容；
2: if logic role is Follower then
3:     return null;
4: else if logic role is Candidate then
5:     if node is in voteRequest phase then
6:         if voteFor is null  or  localTerm < term then
7:             set localTerm = term，更新本地Term;
8:             set localLease = lease，更新投票阶段超时时间;
9:             set voteFor = requestor，将voteFor 设置为投票请求者;
10:            return granted;
11:        else
12:            return null;
13:        end if
14:    else
15:        return null;
16:    end if
17: else
18:    return null;
19: end if
```

如果节点的逻辑角色为Candidate，那么该节点要么处于投票阶段，要么处于选举阶段。在投票阶段时，若之前没有投过票，那么当它收到投票请求后，可以直接投票同意，并将自己的Term 更新为发送方的Term 值；而若投出过选票，则比较投票请求中携带的Term 和自己当前Term，当请求中的Term 值较大时可以进行改票处理。每当Candidate同意一个投票请求时，则延长自己在投票阶段的时间，以免短期内重新发起选举。在广播阶段时，显然Candidate 已经在投票阶段得到了多数派的选票，此时正在向其它节点进行一次确认。Candidate 在这个阶段会拒绝

包括投票请求在内的任何主动请求。具体流程见算法1。

• 处理当选广播

Follower和Candidate在处理当选广播时，与处理投票请求的流程基本类似，唯一的区别是，节点只要同意了当选广播，将直接作为Follower 跟随发送广播的节点，并更新租约，另外不再同意其它节点的广播。具体流程见算法2

所有Follower的租约期都相同，当Leadr故障失效后，各个Follower将同时租约过期而切换为Candidate，并发起选举，此时它们的Term值相同，而且将自己的选票第一时间投给了自己。显然，由于Term相同，Candidate的投票请求都会被拒绝。这种冲突在初次选举时一定会产生，并且在接下来的选举轮次中也会大概率发生。为了解决这个问题，LEBR 参考Raft 算法的“随机退让”机制，令各个同时租约过期的节点在切换为Candidate后，不会立即发起投票请求，而是先等待一个随机时间。这样的机制使不同节点发起选举的时刻产生了差异，对于首先发起选举的Candidate来说，其他节点可能仍在等待中，没有投票给任何接Candidate，因此它将更有可能获得多数节点的选票而成功当选。

“随机退让”机制大大减小了选举冲突，提高了选举效率。LEBR 只在每次进入投票阶段时使用该机制。而广播阶段因为紧随投票阶段之后，也具有了随机性，因此不需要节点发起当选广播前再等待一个随机时间。本文将使用到的术语进行归纳，详见表3.2所示。

3.3.4 算法分析

本文提出的LEBR算法在ScaleDDB架构下很好地实现了总控节点的高可用，替代了传统主备模式高可用方案，使其不再依赖于第三方工具。同时，LEBR 保证不会出现双主问题和频繁选举这两种对于ScaleDDB 需要极力规避的情况。

（1）双主和频繁选举问题的解决

Raft中出现双主问题的原因之一是网络发生分区，使Leader 与多数派节点失去通信。而究其根本原因是，Raft算法本身对于Leader 节点的处理机制存在问题，

表 3.2: 主节点选举相关术语列表

符号	术语含义
t_{inter}	心跳发送间隔
t_{loss}	失联确认窗口
t_{fl}	Follower租约期
t_{ld}	Leader租约期
t_{ex}	租约延长时间
t_d	网络延迟
t_{maxd}	网络最大延迟
t_r	投票前置随机时间
t_e	选举执行时间
t_{ep}	选举期时间长度

即Leader只有发现集群中存在具有更高Term的节点时，才会放弃自己的身份。

这种机制下，一旦出现网络分区使Leader处于少数派分区中，那么当在多数派分区中节点提升Term并发起选举时，Leader 无法感知更高Term节点的存在，从而保持角色状态不变。多数派分区中选举出新的Leader 后，对于整个集群来说，就同时存在了两个Leader。问题进一步引申后，随着节点总数增加还可能同时存在更多Leader。因此，LEBR为了从根本上解决问题，对Leader 引入了租约机制，使Leader 也同其它逻辑角色的节点一样在一定期限内提供正常服务，而不是使用Raft 中的做法。LEBR 中规定Leader 的租约必须小于Follower 的租约，并且通过心跳机制，保证如果Leader处于少数派分区时，能及时停止更新租约，最终使Leader 先于Follower 租约过期，切换身份。一段时间后，当Follower 租约过期而发起选举时，集群中不会有Leader 存在，那么选举结束后当选的Leader就是唯一的主节点。

产生双主问题的另一个原因是，在选举期间允许节点改票。节点可以先后投票给多个具有不同Term的Candidate，使得多个Candidate 都能获得多数选票而当选。如图3.11中，$C_1 - C_5$五个Candidate 参与选举。C_1发送的Term为1的投票请求得到了C_3、C_4和C_5 的选票，满足多数派，因此C_1 可以进入广播阶段。然而，

C_2随后向其它节点发送了Term为2的投票请求。C_3、C_4和C_5接收到该请求后，由于请求包中Term较大，因此也会投票给C_2，使其也得到多数选票而进入广播阶段。

Algorithm 2 LEBR算法的当选广播处理流程

输入： $BroadcastMsgM(term, lease, requestor)$，接收到广播请求后开始执行算法；

输出： $granted\ or\ null$，granted表示同意当选广播，null表示不回复任何信息；

```
1: deserialize broadcast message，解析接收到的当选广播包内容；
2: if logic role is Follower then
3:     return null;
4: else if logic role is Candidate then
5:     if node is in voteRequest phase then
6:         if voteFor is null or localTerm < term then
7:             set localTerm = term，更新本地Term；
8:             set localLease = lease，更新广播阶段超时时间；
9:             set voteFor = requestor，将voteFor 设置为广播发送者；
10:            set logic role as Follower, and quit election;
11:            return granted;
12:        else
13:            return null;
14:        end if
15:    else
16:        return null;
17:    end if
18: else
19:    return null;
20: end if
```

LEBR在解决这个问题上采取了一种朴素的方法，即当节点接收到当选广播时，如果同意该请求，那么在确认后会立即切换为Follower并承认对方为Leader，于是该节点在之后不会处理选举相关的请求，即使出现Term更大的Candidate。这种做法使得成功当选的Leader 一定是得到多数派节点确认的，并且这些节点不

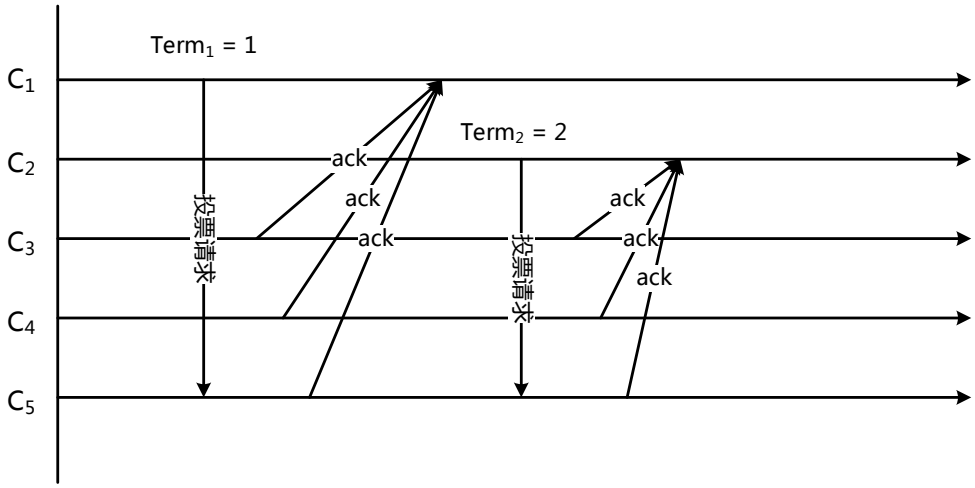

图 3.11: 多个Candidate获得多数选票情况示意图

会再同意其它Candidate的当选广播。多数派的唯一性最终保证了Leader的唯一性。总之，LEBR通过以下两点保证了不会出现双主问题：

- Leader能够自动检测出分区情况，并在选举开始之前放弃Leader身份。
- LEBR选举算法保证ScaleDDB在选举期只有一个总控节点当选Leader。

Raft在集群出现网络半分区时发生频繁选举的原因是，Follower在正常跟随Leader 时，若接收到Term较大的选举相关请求，就不再跟随原来的Leader，并且Leader 会因为收到更大的Term 而改变角色。显然，LEBR 中禁止节点确认多个节点当选广播的算法，能够避免发生频繁选举。

在不允许改票的情况下，若像Raft一样通过一次投票就决定Leader，则当多个Candidate发起选举时，很可能出现算法导致的网络分区，即出现如下场景：如图3.12五个节点组成的LEBR 组中，节点S_1和S_3（Term 值相同，都为1）短时间内相继发起选举，其中S_1 得到了S_2和S_5的选票，当选为Leader，然而S_3 只获得了S_4的选票，选举失败。此时虽然网络并没有异常，但也相当于出现了两个分区：S_1、S_2 和S_5 为一个分区，另一个分区则包括S_3 和S_4。

在算法理论模型中，只要S_3在重新发起选举前收到S_2 的心跳信息，就能终止

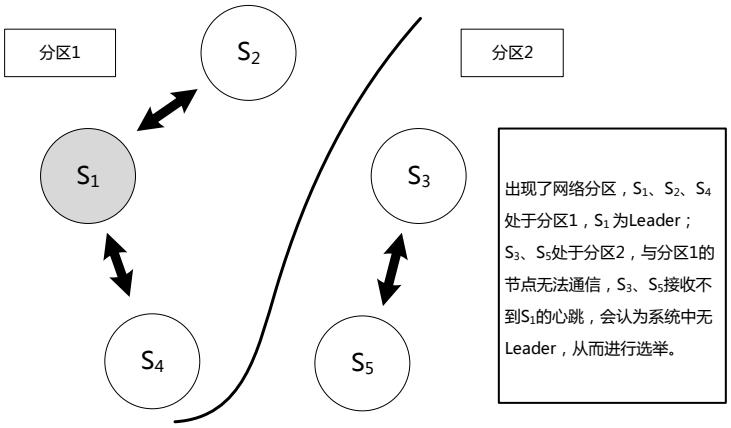

图 3.12: 算法导致网络分区的场景示意图

选举过程，S_3和S_4 都会跟随这个Leader，分区也随之消除。但是，LEBR 是面向可扩展数据库ScaleDDB 的总控节点而设计的，必须考虑节点的实际情况。在真实场景下，总控节点的逻辑角色由Candidate切换为Leader 时，节点将作为主节点管理整个集群，其物理角色也将从Slave 切换为Master，而在这个过程中，总控节点要进行加载集群的节点信息，启动各种任务管理线程等操作，整个流程至少在秒级以上。当总控节点完全切换为Master时才能确认Leader 有效，才会开始运行心跳机制。此时，S_3早已重新发起了投票请求，Term也随之增大，即使不考虑S_4状态如何，至少S_3 无法再接受Term 较低的S_1的心跳信息从而跟随这个Leader。除非重新启动S_3，重置其Term，否则这种分区状况将会一直持续。

LEBR采取的两阶段方式，能降低出现上述这种分区状况的发生概率。本文仍以图3.12 中所显示的场景为例进行说明。S_2和S_5在同意S_1 的投票请求后，也没有完全认定S_1 为Leader，之后还是可以向Term更高的Candidate 投出选票。如果S_1在投票阶段得到了多数派选票，则会立即向所有节点发送当选广播。这段时间在正常情况下一定小于两倍的最大网络延迟，所以通常能在S_3 投票阶段过期前，将广播发送到S_3，而S_3 和S_4的Term仍为1，于是能够同意S_2 的当选广播，并作为Follower跟随它。此时，当S_2切换为Leader后，集群中不会出现分区。LEBR并没有彻底杜绝这种算法导致的分区现象，当网络异常使得延迟时间超过了正常

情况下的最大延迟时间，那还是有可能出现该状况，但是出现的概率会小很多。并且，即使出现了分区，Leader 也一定是处于多数派分区中，不会影响系统的正常使用。

（2）算法时间分析

LEBR选举算法可以在多种常见的故障场景下平稳运行，确保ScaleDDB总控节点的高可用。前文介绍了算法流程，这里本文将分析其选举效率。假设每个节点的服务器时间完全一致，不考虑系统耗时，且节点间的网络延迟也都相同，不会随时间变化。分析时本文使用表3.2 中的术语符号进行描述。

结论1（选举执行时间）：

$$t_e \geq t_r + 4t_d$$

本文将选举时间t_e定义为从Follower租约过期开始，到有节点当选Leader 结束的这段时间。节点开始选举后，会等待随机时间t_r 后发起投票请求，若节点在投票阶段和广播阶段都非常顺利，能立即得到多数派认同，那么经过四次网络传输该节点就能够当选成功，时间为$4t_d$。因此在这个情况下，选举时间至少为$t_r + 4t_d$。选举中可能产生冲突（即多个Candidate节点瓜分选票，使得没有节点得到多数选票）。假如每轮选举都发生了冲突，选举过程将无法终止，理论上选举执行时间没有上限。

真实场景中，选举相关的消息包大小不会超过一千字节（1KB），在千兆网络环境下，正常的传输时间在100微秒左右。因此，选举执行时间中网络传输时间所占比例很小，时间长度基本由t_r决定。为了使t_r具有充分随机性，并考虑到选举执行时间的长度，t的随机范围可以设置为在200ms到300ms之间。多数情况下经过一至三轮选举都能选出Leader，总的选举耗时基本不超过1秒。在企业级应用中，这个时间是可以接受的。

结论2（主节点失效故障下选举期时间长度）：

$$t_{ep} \geq t_{fl} - t_{inter} + t_e$$

选举期时间不等于选举执行时间，它是从Leader失效开始计算的，完全涵盖了后者。若Leader在发送心跳前一刻失效，那么Follower 将在$t_{fl} - t_{inter}$ 时间后过期，并经过t_e时间结束选举，此时选举期最短。由于t_{fl}设置为秒级，而t_e则在毫秒级，因此对于选举期时间来说，主要取决于Follower 的租约t_{fl}。由于租约是一个固定的时间，所以选举期时间相对比较稳定，而选举执行时间对它的影响较小。

结论3（网络分区故障下选举期时间长度）：

$$t_{ep} \geq t_{loss} - t_{inter} + t_e$$

发生网络分区时，节点并不会立即失效，而是当它在t_{loss} 内没有接收到多数Follower的心跳回复信息时才会下线。此时，Follower 余下的租约时间最短为$t_{loss} - t_{inter}$，再加上选举执行时间te就是整个选举期时间。同t_{fl}一样，t_{loss} 也是一个秒级的固定时间，因此这种故障情况下的选举期时间由t_{loss} 主导，并且波动不大。

从前面三个结论的分析中可以发现，LEBR算法的正确性和效率，与网络延迟和节点时钟的一致性密切相关。事实上，这两点是本文讨论LEBR的前提，也是算法的局限性所在。

如果多数节点间网络延迟过高，超过了投票或广播阶段的超时时间的一半，会导致这两个阶段中节点的请求始终无法得到多数节点的响应，选举将不能成功，集群会一直处于无主的选举状态。节点服务器始终误差过大也会导致LEBR算法失效，这是因为Leader在准备更新各节点租约时，是以本地时间为基础计算新的租约过期时间，如果时钟不同步，那么有可能导致Follower租约提前于Leader过期，从而产生双主问题。例如，假设Leader当前时间为T_1，其它Follower 当前时间均为T_2，两者存在误差且$T_2 - T_1 > t_{ex}$ 。当Leader 开始更新租约时，它所计算的新租约为$T_2' = T_1 + t_{ex}$ ，显然，Follower 按此更新租约后，会发现$T_2 > T_2'$，租约立即处于过期状态，从而转变为Candidate并发起选举。由于Follower租约过期，都不再跟随原Leader，可以自由参与选举，所以此时将有可能在Leader存在的情况下又选举出另一个Leader，导致双主问题。考虑到网络延迟和Leader 发送心跳的时间

间隔，可以得到时钟误差的范围，即：

$$| T_2 - T_1 | < t_{ex} - t_{inter} - 2t_d$$

3.3.5 算法实现

（1）数据结构

节点选举状态（NEState）和交互信息（Message）是实现LEBR 算法需要的两类数据结构。其中，节点选举状态指的是节点存储的与LEBR选举算法相关的状态信息。前文中曾指出，节点应该是平等而无状态的，但这是从节点作为总控节点在ScaleDDB 中运行的角度来考虑，指其不应存储影响数据库系统本身状态的信息，例如已提交日志等。而从算法本身来看，参与选举的各个节点是有状态的，它们所包含的信息并不完全相同。节点选举状态包含的信息数据项详见表3.3：

表 3.3: NEStat数据结构

NEStat数据项	取值范围	意义	说明
逻辑角色（role）	{Leader,Candidate, Follower}	指节点当前的逻辑角色。	可以改变。
任期编号（term）	{0,1,2,...,N}	指节点的逻辑时间，也表示选举轮次。	Term 只能增加，不能减小。
租约（lease）	[0,+∞)	指节点维持当前角色的时限。	只有Leader 能延长该租约。
投票对象（voteFor）	服务器地址	指节点在投票阶段投出选票的对象；或指非选举期时跟随的Leader。	选举阶段可能被修改。
有主标识（hasLeader）	{true, false}	表明节点是否正在跟随某个Leader。	当hasLeader为true时，只有节点本身租约过期时才能修改这个值。

交互信息是节点之间通过网络相互传递的信息，包括三类：心跳消息、选举请求和当选广播，每种信息都包括一对请求包和回复包，因此LEBR 中共有六种

消息包来传递相关信息。消息包及其包含的数据项见表3.4，对号表示有，错号表示没有。其中，“请求结果”是投票请求回复包和当选广播回复包中特有的，是节点对于相应请求的回复，同意时取值为true，拒绝时则忽略请求。

表 3.4: 选举中涉及的消息包

消息包	发送者地址	任期编号（term）	租约（lease）	请求结果（result）
心跳包（HeartBeatMsg）	√	√	√	×
心跳回复包（heartBeatRespMsg）	√	×	×	×
投票请求包（VoteRequestMsg）	√	√	√	×
投票请求回复包（VoteRequestRespMsg）	√	√	×	√
当选广播包（BroadcastMsg）	√	√	√	×
当选广播回复包（BroadcastRespMsg）	√	√	×	√

本文使用数组otherNodes来存储LEBR组中其它节点的信息（ONStat），例如若LEBR中包含S_1、S_2和S_3 三个节点，那么S_1 中的otherNodes将保存S_2 和S_3的信息。这些信息与NEStat不同，包括在节点相互通信过程中，其它节点的回复情况，每次进行新一轮选举时需要重置数组中各个ONStat 信息。具体见表3.5。

LEBR运行过程中，若需要向其它节点发送消息，就会遍历otherNodes来获取目标服务器地址，而其它节点回复的结果也存储在该数组中。另外，VoteRes和BDRes这两个数据结构分别用来保存投票阶段和广播阶段的结果，其中包含term以及此term下的最终请求结果两个数据项，用来判断某个选举阶段是否成功。

（2）线程结构

在LEBR算法中，从节点间通信的角度来看，节点执行的操作包括消息的主

动发送和接收处理两部分，本文使用两个线程S-Thread 和R-Thread 分别进行处理。为了避免网络传输开销，S-Thread线程采用异步的方式向其它节点传递消息。所谓异步指节点发送消息包后，节点不等待回复信息，而是继续执行下面流程。R-Thread 线程与S-Thread同时运行，负责处理接收到的所有消息包。事实上，两个线程都是围绕同一个数据结构NEStat来执行相关操作，例如读取Term、更新Lease等。对于NEStat等线程共享数据，本文使用读写锁实现并发处理。LEBR算法的有效运行基本是依靠线程S-Thread 和R-Thread协作来实现。接下来本文将通过介绍这两个线程执行的具体流程，描述LEBR算法的实现。

表 3.5: ONStat数据结构

ONStat数据项	取值范围	说明
服务器地址	已配置的服务器地址集合	由IP和PORT组成
投票结果（voteResult）	{true, false}	值为true则表示同意
广播确认结果（bdResult）	{true, false}	值为true 则表示同意
心跳回复时间戳（hbTimestamp）	[0, +∞)	接收到相应节点的心跳回复时，从本地服务器获取该时间戳。

S-Thread会循环调用选举主函数检查节点的逻辑角色以及其它的选举状态，来执行相应角色的任务，并决定消息的发送类型和目标节点。

若逻辑角色为Leader，线程通过比较本地时间与租约过期时间，判断Leader的租约是否过期。如果租约过期，则将角色转变为Follower，否则将继续检查Leader接收到的心跳回复时间戳是否有多数个落于“失联确认窗口”内。如果有，线程会将本地时间延长t_{ex} 后作为新的租约过期点，此时才会更新自己的租约。

若逻辑角色为Follower，线程仅判断其租约是否过期，如果过期，那么就把角色转变为Candidate，并重置节点状态，即把节点各个状态信息恢复为默认值。

若逻辑角色为Candidate，线程首先通过选举阶段标识确认Candidate所处的阶段。如果节点处于投票阶段，则会进一步判断上一次投票请求的结果（如果存

在），根据多数派原则确定节点能否进入广播阶段。如果不能进入下一阶段，那么再判断Candidate 能否再次发起投票请求。这里本文采用类似于租约的方式，设置一个指定范围内的随机的未来时间点作为下一次允许发起投票的时刻。之所以将时间点以随机的方式设置，是为了实现LEBR 算法的“随机退让”机制。当线程发现当前时间大于这个预设的时间点时，就会异步向数组otherNodes 存储的所有节点发起投票请求。如果线程在检查投票结果时，节点得到了多数选票，那么就会通过修改选举阶段标识使节点进入广播阶段，并向其它节点异步发送当选广播请求。如果线程通过判断选举阶段标识发现其正处于广播阶段，首先检查上一次广播请求的结果（如果存在），得到多数阶段的确认回复时，节点逻辑角色会变为Leader，否则，将进入下一轮投票阶段。

节点维护一个消息包队列，当接收到选举LEBR相关的消息包时，会首先将其压入队列，该操作由另一个专门的线程P-Thread来完成，而R-Thread则从队列头取出消息包进行处理。这种实现方式使R-Thread可以专注于消息包的处理，减少了额外开销。因为这部分的实现较为单调和机械化，并没有涉及复杂的算法逻辑，所以本文不对此进行具体介绍，只罗列出处于不同逻辑角色和选举阶段下的节点处理各种消息包所执行的操作，详细见表3.6。

3.4 本章小结

本章中针对ScaleDDB总控节点的高可用问题，本文提出了基于Paxos思想的主节点选举算法LEBR，并参考分布式一致性协议Raft进行设计和实现。首先，本文分析了可扩展数据管理系统ScaleDDB本身具备的可用性和容错性，指出决定其可用性高低的是总控节点和事务处理节点。同时本文还分析了采用传统主备模式来实现总控节点高可用所存在的问题；其次，本文描述了面向工程实践的分布式协议Raft，着重分析了Raft主节点选举模块的算法思想和实现细节，指出其用于实现无状态的总控节点选举时可能产生的致命问题，即双主问题和频繁选举；最后，本文设计并实现了适合于ScaleDDB 总控节点的的主节点选举算法LEBR，能

表 3.6: 不同逻辑角色下对于消息包的处理

	Leader	Follower	Candidate
心跳包	严重错误，报错并退出进程。	只处理自己所跟随leader的心跳信息，并更新租约。	若本地term不大于心跳包中term，则将状态转为Follower并跟随该Leader，同时更新term 和voteFor。
心跳回复包	记录接收消息包的时间戳并存储到otherNodes数组中对应节点内。	忽略请求。	忽略请求。
投票请求包	忽略请求。	忽略请求。	若本地term比请求包中的term 小，则投出选票，并更新term 和voteFor。
投票请求回复包	忽略请求。	忽略请求。	只有当本地term与回复包中term相等时，才进行处理，即修改otherNodes 数组中对应节点的voteResult为true，并统计票数。
当选广播包	忽略请求。	忽略请求。	若本地term比请求包中的term小或者请求的发送方是自己此轮选举中投出选票的对象，则将状态转为Follower并跟随该Leader，同时更新term 和voteFor。
当选广播回复包	忽略请求。	忽略请求。	只有当本地term与回复包中term相等时，才进行处理，即修改otherNodes 数组中对应节点的bdResult 为true，并统计票数。

满足金融关键性业务对于可用性需求。与3.1节中提到的传统的Linux HA 高可用方案相比，LEBR算法具有能够容忍网络丢包异常、不依赖第三方软件的优点。而与Raft算法的实现相比，由于LEBR 适配了无状态节点选举的需求，因此能够避免双主与频繁选举这两个对系统可用性有极大影响的问题。所以，本文提出的LEBR算法对ScaleDDB 的应用场景更为合适。

第四章　主事务处理节点选举

第三章描述的LEBR选举算法是本文提出的可扩展数据管理系统高可用方法的一部分，用以解决无状态总控节点的高可用问题。而在本章中，本文将介绍高可用方法的另一部分内容，即主事务处理节点的选举算法MTS（Master TNode Selection）。这两部分内容是紧密结合在一起的，主事务节点算法依赖于总控节点的高可用，需要其保证稳定性和唯一性，而这一点需要LEBR 算法来实现。所以说，主事务处理节点选举算法（MTS）与主总控节点选举算法（LEBR）共同实现了ScaleDDB 架构的数据管理系统的高可用。

4.1　问题描述

在类ScaleDDB数据管理系统中，为了避免分布式事务带来的复杂处理流程和性能问题，使用单个事务处理节点TNode在内存中执行所有事务操作。这种单节点架构显然无不能避免单点故障，无法满足企业对于数据库可用性的基本要求。3.1 节描述了一种使用高可用集群技术提高TNode 节点可用性的方法，即以主备模式配置多个TNode 节点，每个节点都存储相同的数据副本。其中主节点具有唯一性，负责集中式事务处理，并将对于数据副本的操作以日志的方式实时复制到备节点中。这些日志具有连续且唯一的日志号，节点通过按序回放日志能够实现数据更新和恢复。通过日志复制能实现数据多副本的一致性，当主节点故障时，可以由总控节点CNode 选择一个与原来主节点数据一致的备节点，作为新的主节点处理事务，使系统恢复正常。

日志复制分为积极复制模式（Eager）和消极复制模式（Lazy）[53]两类。积

极复制模式指主节点将数据复制到所有副本后才响应客户端，这能够保证数据副本间的强一致性，但会极大影响系统性能和可用性；而消极复制模式中主节点处理完成后立即回复客户端，不会等待所有数据副本完全一致后再回复，这带来了高读写性能和可用性，但无法保证数据不会丢失。对于TNode 来说，无论是采取哪一种模式的日志复制算法，当主TNode 失效时，只要CNode 从剩余的备TNode中选出拥有最新日志的节点作为新主节点提供服务，那么就基本能够使ScaleDDB 恢复到它所能恢复的最新状态。积极复制模式和消极复制模式的优势和缺陷同样明显，前者只顾及数据强一致且高可用而忽略了性能，而后者为了追求较高的写入性能而不管数据是否一致。在实际应用中，分布式数据管理系统用于实现数据一致性的常用方法是基于Quorum 协议[54]的复制技术。其基本思想是，一个操作是否能够执行需要系统中各个节点投票决定。允许操作执行的最小票数被称作Quorum （法定人数），需要超过总节点数的一半。该复制机制是积极复制和消极复制的折中，能够在数据一致性的前提下兼顾系统的事务处理效率。若ScaleDDB 使用基于Quorum的日志复制技术实现TNode 数据一致，那么主TNode 节点只需要将日志复制到包括自己在内的多数个节点中，即可提交相应事务。图4.1展示了基于Quorum日志复制流程。客户端向主TNode节点发出写操作请求，主TNode处理时向备TNode复制日志，并得到了三个备TNode节点的应答，大于法定人数，因此TNode 认为日志复制成功，于是向客户端返回请求的结果。

使用Quorum方式进行日志复制时，本文将集群中配置的事务处理节点的集合称作Quorum组。Quorum 中，主事务处理节点在接收到事务请求后，必须将相关日志复制到Quorum组的多数派节点，并写入它们的磁盘后，才能够提交事务。这种做法能够保证多数事务处理节点都拥有最新的数据，只要发生故障的节点不超过半数，那么总能够从存活的节点中找出保存了全部数据的节点，保证了系统恢复后数据不会丢失，也消除了单事务处理节点架构所带来的巨大风险。提交事务时，可能存在少数节点没有接收到相关的日志而没有办法更新数据，也没有在本地提交事务，如果客户端从该节点读取数据，就会得到错误的过期数据。为了避

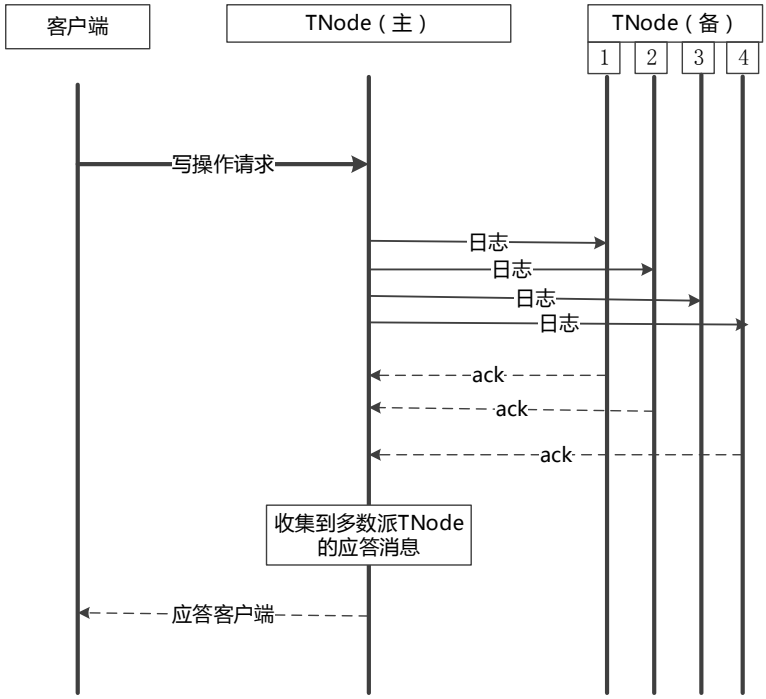

图 4.1: 基于Quorum 的日志同步示意图

免这种情况，可以采取两种处理方式:

- 所有读请求由确定保存最新数据的主节点进行处理。
- 从多数个节点中读取数据，那么最新的数据一定在这些节点中，客户端就可以从中获取到最新的数据。

CNode 在选择新主TNode时不能简单地选择具有最新日志的节点，否则可能会出现问题。考虑这样一种场景：集群中共有$S_1 - S_5$ 五个TNode 节点，其中S_1为主节点。S_1 接收到客户端的事务请求，生成了日志号为1-10 的10 条日志，并写入本地磁盘，但是由于网络异常等问题，这些日志没有复制到其它节点上，随后主节点S_1 故障宕机。其它四个节点中会选择出一个新的主节点，假设为S_2 ，此时该节点也接受到请求，由于之前S_2 中没有其它日志，因此S_2 节点上的日志号仍从1 开始，设其生成了日志号为1-5 的5 条日志，并成功复制到所有正常节点中，最后提交事务并回复客户端。这个时候，$S_2 - S_5$ 四个节点都下线，然后重启所

有的五个节点。显然，S_1 中最新日志为10 号日志，比其它四个节点都要大，如果CNode依照以前的选择算法，就会选择S_1 作为主节点，但是这很明显是错误的，因为日志1-10 并没有被提交，而且日志号为1-5 的日志在S_1和其它节点上的内容不同。

综上所述，对于使用基于Quorum的复制技术的TNode 节点组来说，其主节点选择问题应需要新的算法来解决。本文提出了针对此种日志复制技术的TNode 的主节点的选举算法MTS （Master TNode Selected）。本章中将介绍该算法的设计和实现，并从正确性和效率两方面对算法进行分析。

4.2 MTS选举算法

4.2.1 算法设计

ScaleDDB事务处理节点选举算法MTS属于被决式选举。简单来说，就是在使用多副本冗余的情况下，由总控节点CNode 负责在主TNode节点故障时选择新主节点，并且这个新主节点一定拥有最新数据。CNode 会控制整个选举的流程，确保只有一个选择主TNode 的任务处于执行状态。需要指出的是，参与选举的节点需要保持时钟同步。

CNode依据TNode的权重信息来选择主TNode节点。所谓权重信息，指TNode中可用于区分保存数据新旧的信息TNode。因此，权重信息必须能够真实反映出TNode 节点状态。常用的权重信息是日志号，因为它具有唯一性并且连续递增。但是在Quorum 日志复制中，仅依据日志号判断显然无法解决4.1 节中描述的复杂故障场景。在本文提出的选举算法中，增加“日志版本（Log Version）”这一辅助条件，与日志号共同作为TNode的权重信息，能够很好地解决前面描述的问题。

TNode的日志版本是递增的，但不一定连续，它用来表示一批日志的新旧。每次选出主节点后，日志版本由主TNode节点生成，并在自己为主节点期间使用同一个日志版本，同一个日志版本下的日志不会出现重复。生成日志时，TNode会将日志版本写入到每条日志中，与日志号一起作为日志的唯一标识。由于时间

戳严格递增的特点与日志版本一致，主TNode 在生成日志版本时，直接使用本地时间戳作为日志版本的值。另一种维护日志版本的方式是，每生成一条日志就产生一个新的日志版本。各个节点时钟同步的情况下，由于时间戳随时间严格递增，日志版本可以作为全局日志的唯一标识，不会出现同一日志版本的日志具有不同内容，并且日志版本较高的一定是最新生成的日志。这种方式的问题在于，当系统处理高并发写事务时，主TNode 节点需要短时间内生成大量写事务日志，导致获取本地时间戳的操作被频繁执行，产生了很多非必要的额外开销。显然，使用的维护日志版本方式避免了这种情况。

TNode节点有主（Master）备（Slave）两种角色，这两种角色的TNode都使用了租约机制，以便于CNode的管理。所有TNode节点的租约时间相同。TNode节点检查自己的租约，备节点租约如果过期就会重新向CNode注册，而主节点租约过期时则将自己切换为备节点。CNode中维护一个TNode 列表，保存所有向其注册过的TNode 信息。系统正常运行时，TNode 定期向列表中所有TNode以异步的方式发送心跳信息，更新TNode 的租约，同时将每个TNode 更新后的租约存入列表中。这里使用的租约和心跳机制的作用和意义与第三章中描述的一样，因此不再赘述。CNode 不断检查列表中这些TNode 的租约是否过期，并根据角色做不同的处理。

- 若租约过期的是备TNode节点：CNode 会将其从列表中删除，以后不再向其发送心跳包更新租约，直到该TNode 重新注册并加入到列表中。由于TNode租约过期意味着发生故障，所以CNode 的这种做法可以杜绝无效的TNode被选为主节点，也使CNode不用把系统资源浪费在管理失效的备节点上。

- 若租约过期的是主TNode节点：说明CNode 认为主TNode 节点已经出现问题，集群中已经不存在处理事务请求的主TNode。CNode 会将该节点从列表中删除，并在保护时间t_p 后（t_p 大于节点间时钟误差，为了确保算法执行时主TNode 失效而设置）开始执行选择主TNode 的流程。

MTS选举算法流程如下：

- CNode向TNode列表中所有节点请求获取目标节点的权重信息（即最大日志号和日志版本），必须收集到至少多数个TNode节点的权重信息才能继续执行选举流程。否则就意味着这次选举失败，等CNode再次检查到租约过期时重新执行。

- 当CNode节点获得多数TNode权重信息后，首先找出其中日志版本最大的一个或多个节点，然后在这些节点中再找出日志号最大的一个节点，这个节点就是CNode 最终选出的新的主TNode。

CNode将新主TNode的信息发送给所有的TNode节点。TNode节点接收到该信息后，如果发现新的主节点就是自己，那么将角色从Slave切换为Master，接管系统的写事务处理任务；如果新的主节点不是自己，那么就向新的主TNode进行注册，使得自己可以接受主TNode复制的日志。至此，MTS算法流程结束。整个算法的具体流程见算法3。

Algorithm 3 MTS 算法执行流程

输入： $TNode$节点集合：$S = \{T_i | i = 1, 2, ...n$，其中，$S_i$代表标号为$i$的节点$\}$；
$TNode$节点总数：$N(N > 3)$；

输出： 主$TNode$节点S_i；

1: $fetch\ W = \{W_i | i = 1, 2, ...M\}$，向所有TNode节点的请求权重信息，M代表成功获取的数量；
2: **if** $M > N/2$ **then**
3: $W_v = maxLogVersion(W)$，有$W_v = \{W_i, i \in \{1, 2, ...M\}\}$，从权重信息中取日志版本最大的若干节点信息；
4: $W_{vl} = maxLogSeq(W_v)$，从上一步中得到的结果里选择日志号最大的节点信息，则它代表的节点为主TNode；
5: $return\ S_{vl}$；
6: **else**
7: $return\ null$；
8: **end if**

4.2.2　算法分析

（1）故障场景分析

对于ScaleDDB的事务处理节点来说，常见的故障场景有如下三种，本文将分析在每一种场景下算法的有效性：

- **主TNode节点故障失效。**

  此时数据库系统处于不可用状态，不能处理读写请求。但由于CNode中的TNode 列表内主节点租约没有过期，CNode 并不会执行选择算法，也不会继续延长保存的TNode 租约。等待一个TNode 租约时间后，CNode 就可以选择出新主TNode 节点。

- **主TNode节点与CNode网络中断。**

  网络中断导致主TNode的租约无法被更新，但是在它已有的租约期内也不会失效或切换为备TNode，可以如正常时一样处理事务，直到其检查到自己租约过期而切换为备TNode 节点。由于时钟同步，CNode上保存的该TNode 租约也会过期，保护时间过后开始从备TNode 中选择主TNode 节点。

- **TNode节点组出现网络分区，且主TNode 处于少数派子分区中。**

  此时TNode不能将日志复制到多数节点并提交事务，因此不能对外正常服务。但由于与CNode 间的网络没有异常，主TNode能够接收到心跳而更新租约，不会因租约过期而切换，使得系统始终不能恢复正常。这种情况发生时需要人工介入解决。事实上，虽然本文提出的算法没有解决这种场景下的故障恢复问题，但保证了网络分区存在期间系统内不会出现多个主TNode 节点，也不会出现数据丢失或不一致。

（2）算法时间分析

MTS算法的执行时间可以从算法本身和集群能否服务两个角度来分析，该算法在理论上是存在时间上限的。本文首先列出在分析中会使用到的术语，以及它

们的符号表示和意义，列于表4.1 中；然后，假定节点间的时钟一致，并且不考虑真实场景下节点收发包的处理时间，以及TNode 的主备切换时间，在这种情况下本文得出以下几个结论：

表 4.1: 事务更新节点选择算法相关术语

术语	符号表示	说明
MTS算法执行时间	t_s	表示从CNode 正式执行算法流程开始至主TNode被选出来的这段时间。
系统不可服务时间	t_{sys}	表示因主TNode 节点问题而导致的系统不可服务时间。
保护时间	t_p	表示从CNode 检查到主TNode租约过期，正式执行算法的时间。
TNode租约时间	t_{tl}	由CNode来维护此租约。
TNode租约延长时间	t_{tex}	新的租约由旧租约加上此时间产生。
CNode发送心跳时间间隔	t_{inter}	发送心跳的时间间隔也是更新租约的时间间隔。
网络延迟	t_d	网络传输时间。
最大网络延迟	t_{maxd}	正常情况下可能的最大网络延迟。
TNode节点个数	N	部署集群时配置的所有TNode节点个数。

结论1（算法执行时间）：

$$0 < t_s = 2Nt_d$$

CNode获取TNode的权重信息时，采用的同步方式，即串行地向各个TNode节点发送请求。每个请求都需要两次网络传输时间，即$2t_d$，那么若有N个TNode 节点需要发送N次请求，总时间为$2Nt_d$。所以算法执行时间只与TNode节点个数和网络传输延迟有关。

结论2（主节点故障下系统不可服务时间）：

$$t_{tl} + t_p + t_s + t_d \leq t_{sys} \leq t_{tl} + t_{inter} + t_p + t_s + t_d$$

若主TNode节点在响应心跳信息前一刻发生故障，则CNode将在t_{tl}时间后发现主节点租约过期，这是最小时间；若主TNode 节点在响应心跳信息之后才发生故障，那么CNode 下次仍然能发送心跳信息并更新列表中TNode的租约，因此CNode 需要经过$t_{tl} + t_{inter}$ 时间才能发现TNode 过期，这是最大时间。等待t_p时间后，CNode 再经过t_s 即可选出新的主TNode 节点，然后将结果异步发送到所有TNode 中，这需要至少一次网络传输时间。

由于TNode是事务处理节点，网络和节点的负载压力较重，有时可能延迟接收到心跳信息使租约无法及时更新，导致租约过期而失效。因此，TNode 的租约时间根据应用场景的实际情况进行设置，可达数秒。与租约时间相比，其它时间，如算法执行时间等，均为毫秒级，几乎可以忽略不计。所以，主节点故障时系统不可服务时间取决于TNode节点的租约时间。

结论3（网络中断情况下系统不可服务时间）：

$$t_p + t_s + t_d <= t_{sys} <= t_{inter} + t_p + t_s + t_d$$

主TNode节点和CNode节点网络中断，CNode将无法再更新主TNode 节点的租约，而TNode在已有租约期内仍能正常工作。如果主TNode 节点回复主CNode 的心跳信息前一刻，出现网络中断问题，则CNode 会在t_{tl}时间后能检查到其租约过期，此时TNode节点上真正的租约也过期，所以这种情况下系统的不可服务时间为$t_p + t_s + t_d$；若是在回复心跳信息后一刻才出现网络故障，则CNode 下一次发送心跳时虽然没有更新TNode的实际租约，但是TNode信息列表中相应TNode 的租约会被更新，因此可以算出系统不可服务的最长时间为$t_{inter} + t_p + t_s + t_d$。显然，网络中断情况下系统不可服务时间也由TNode租约时间决定。

4.2.3 算法实现

MTS算法涉及CNode和TNode两类节点，算法的主体部分在CNode 上实现。本节中将介绍实现算法所需的数据结构和线程结构。

（1）数据结构

TStat中保存了CNode管理TNode节点时所需的信息，由CNode 负责更新和维护。CNode节点管理TNode 节点的大部分操作，例如租约检查、节点上下线等都通过此数据结构来实现。本文使用tList 数组来存储集群内所有可用TNode节点的TStat 数据。表4.2列出了TStat 所包含的数据项。

表 4.2: TStat数据结构

TStat数据项	取值范围	说明
服务器地址（server）	已配置的服务器地址集合	由IP和PORT组成。
角色（role）	{Master, Slave}	集群中节点的主备角色。
租约（lease）	[0,+∞)	由CNode 在本地维护。
租约可更新标识（needRenew）	{true, false}	标识TNode是否需要更新租约。
日志版本（logVer）	[0,+∞)	每次选举主TNode节点时提升日志版本。
最大日志号（maxLogSeq）	[0,+∞)	TNode中最大的日志号。

（2）线程结构

算法实现共需要三个线程来执行各种操作，包括CS-Thread （CNode Sender）线程、CC-Thread （CNode Checker）线程和TH-Thread （TNode Handler）线程。其中，CS-Thread 和CC-Thread 工作在CNode 节点上，前者负责定期向TNode 发送心跳，后者则负责检查租约并执行算法的流程；而TH-Thread 工作在TNode 节点

上，负责处理CNode 的心跳信息，更新租约等。下面具体介绍各个线程的执行流程。

CS-Thread线程负责定期遍历TNode信息列表tList 中各个TNode 的状态，向每个节点发送心跳更新其租约。更新租约时，CNode首先将tList 中相应TNode 的租约时间延长t_{tex} 后，存入心跳包中，随后异步发送给TNode。需要注意的是，无论心跳包是否发送成功，tList 中的租约信息一定会更新。然而这种续租方式使得无论TNode处于何种状态，CNode都认为其正常运行，无法监测到TNode的异常。因此，tList 的每个TStat中都增加了租约可更新标识needRenew，CNode据此判断是否需要向其中某个TNode 发送心跳。只有当needRenew 值为true 时，CNode才会发送心跳来更新相应的TNode 租约，随后就将该值重置为false，直到接收到该TNode心跳回复才再次置为true。通过这种机制，可以及时发现异常的TNode，停止更新tList 中该节点的租约。

CC-Thread线程负责循环检查tList中各个TNode 节点的租约，通过与CNode本地时间比较来判断租约是否过期。如果备TNode 过期，则将其对应的TStat从tList中删除；如果主TNode过期，也把该节点对应的TStat删除，并在保护时间过后，进入MTS算法流程。算法流程包括四个步骤:

- 向tList中所有TNode节点发送同步请求，获取其日志版本logVr 和最大日志号maxLogSeq，收到回复后将结果存入TStat 中。

- 比较各个TNode节点的logVr，将最大的几个TNode节点信息存入临时结果集S_{temp}中

- 比较临时结果集S_{temp}中TNode 节点的logVr 大小，选择最大的TNode 为系统新的主事务处理节点。

- 将主TNode的信息通过心跳强制发送给所有TNode 节点（强制的意思是不考虑needRenew）。至此完成整个算法流程。

TH-Thread线程负责监听来自CNode的请求，包括心跳信息和权重信息请求。当备TNode收到心跳信息时，首先根据其中的租约值来更新自己本地租约，然后检查心跳中附带的主TNode 信息，如果发现与自己之前注册的主TNode不同，那么认为主TNode 发生了改变而重新注册；当主TNode 收到心跳信息时，也先更新租约，随后检查心跳中的主TNode信息，若发现与自己相同，则不做任何处理，否则将自己切换为备节点，并向新的主节点注册。当TNode收到权重信息请求时，直接返回权重信息。

4.3 本章小结

本章描述了实现ScaleDDB事务处理节点高可用的主节点选举算法MTS，并分析了该算法的设计思路和实现方法。首先，本文介绍了事务处理节点的特点，并分析在其基于Quorum 协议的数据一致性架构的基础上，实现主节点选举和故障恢复时需要注意的问题；然后，本文对MTS算法进行了详细介绍，包括其触发条件和执行流程等，MTS 运行于总控节点上，同时也需要事务处理节点提供相关的信息；最后，本文分析了该算法的执行时间和效率，并描述了实现方式。

第五章　实验

本文提出的可扩展数据管理系统高可用方法解决了类ScaleDDB的分布式数据库系统的可用性问题，能够满足金融等行业的关键业务对于高可用性的需求。本章中，本文对算法在真实应用场景下的运行情况进行了多类型高密度的测试，通过对节点故障和网络异常的模拟实验，测试本文算法的执行效率和对于客户端读写访问的影响。

5.1　实验设置

依照本文所提出的可扩展数据管理系统的高可用方法的实现流程，在类ScaleDDB 的分布式数据库OceanBase中进行了实现，其版本号为0.4.2.21。该数据库包括RootServer(R S）、UpdateServer（UPS）、MergeServer（MS）和ChunkServer（CS）四类节点，能够与ScaleDDB中的节点一一对应，对应关系见表5.1。本文在RS 和UPS上实现了高可用方法。

表 5.1: OceanBase与ScaleDDB节点对应关系表

OceanBase节点	ScaleDDB节点
RootServer	总控节点（CNode）
UpdateServer	事务处理节点（TNode）
MergerServer	接口服务节点（INode）
ChunkServer	基线数据存储节点（DNode）

OceanBase作为一款已经在实践中成功使用的分布式数据管理系统，在其底层代码中已经实现了比较高效地线程和网络框架，以及节点间的通信机制。由

于OceanBase 目前在某银行的历史库系统中已经被用作底层数据管理平台，而本文提出的高可用算法将作为OceanBase 功能在生产环境中使用。因此，本文在实现时尽量避免改动OceanBase 原有的代码逻辑和框架，尽可能利用已有数据结构和线程来满足算法的需求。这样做可以降低未来对生产环境中部署的OceanBase进行升级时的风险。另外，为了将算法融入到OceanBase现有代码框架中，本文对算法的实现流程进行了适应性调整，可能会影响算法的理论执行时间，但是对于算法的高效执行和正确性没有任何影响。

本文在OceanBase的RS节点上实现主节点选举LEBR 算法，算法模块较为独立；而在实现MTS 算法时，不仅修改了RS 中的UPS 节点管理模块，还在UPS 中增加日志版本的生成函数和存取接口。

我们准备了9台位于同一机架的商用服务器来搭建OceanBase 集群作为测试环境。为了使测试时，各个服务器上进程的运行环境一致，在每台服务器上都部署RS、UPS、MS和CS 四个节点。服务器硬件配置见表5.2。

表 5.2: 实验环境配置

硬件	型号与参数	说明
CPU	Intel(R) Xeon(R) E5-2640*2 2.3GHZ	20 核/CPU
内存	504GB	无
网络	Intel Corporation I350 Gigabit Ethernet	以太网

本文提出的两种算法用于解决高可用问题，因此测试时应衡量系统故障的可恢复性和恢复时间，而故障的可恢复性测试是隐含在测试恢复时间中的，所以实验中将测试系统整体的故障恢复时间（即不可服务时间）。不可服务时间包括租约等待时间、算法本身的执行时间，以及节点切换时间三部分。其中租约等待时间由用户设置，不需要测试，因此本文只测试算法执行时间和节点切换时间。

实验中需要模拟分布式环境中常见的两种故障场景：主节点失效和网络分区。显然，主节点失效可以通过强制下线主节点来模拟故障。而对于网络分区问题，如果主节点在多数派子网络中，根据LEBR 算法，系统仍是正常可服务的；如果

主节点在少数派子网络中，会检测到与多数节点失去联系而下线，最终触发选举的条件仍是节点租约过期，因此也可以使用前面提出的方式来模拟故障。

综上所述，本实验将采取强制下线主节点的方式模拟故障，测试LEBR 算法和MTS 算法的系统故障恢复相关时间。需要说明的一点是，实验时为了保证测试的环境相同，我们每次测试后都会清理相关进程和文件，并重新部署整个集群。

5.2　实验结果分析

本节将从故障恢复的角度对本文提出的高可用方法进行测试和分析，以确定其是否能够满足金融行业关键应用对于数据库可用性的需求。

5.2.1　LEBR算法相关测试

图5.1展示了配置不同节点个数情况下的选举执行时间、节点切换时间，以及系统不可服务时间的关系。实验时我们分别在集群中部署3-9 台服务器，每台服务器上都启动包括RS 在内的四个进程，并且没有任何其它负载。同时，设置LEBR选举的随机等待时间范围为200ms-300ms，Follower租约时间为4s。我们对每个服务器节点数量下的Oeanbase 集群都进行100 次故障模拟，最终得出配置不同数量RS 的集群的平均不可服务时间。需要说明的是，图中横坐标取值范围为2,3,...8，是因为采用强制下线节点的方式模拟故障时，若配置n 各节点，那么最终参选的节点数为n-1。例如测试三节点集群时，下线主节点后就只有两个节点参与选举。

测试结果说明，LEBR 算法的执行时间平均在300ms 左右，并且在所有的测试中RS都只进行了一轮选举，几乎没有选举冲突产生。每次成功当选的节点一般都是率先发起选举的节点，其随机等待时间都靠近200ms，因此在一轮选举的情况下，算法执行时间相差不大。该结果还说明参与选举的节点个数与选举执行时间无关。因此在实际应用中，只考虑所需的可用性级别来配置总控节点的个数即可。

另外，从图中还可以观察到，RS切换时间在3s-4s之间，比LEBR 算法本身的

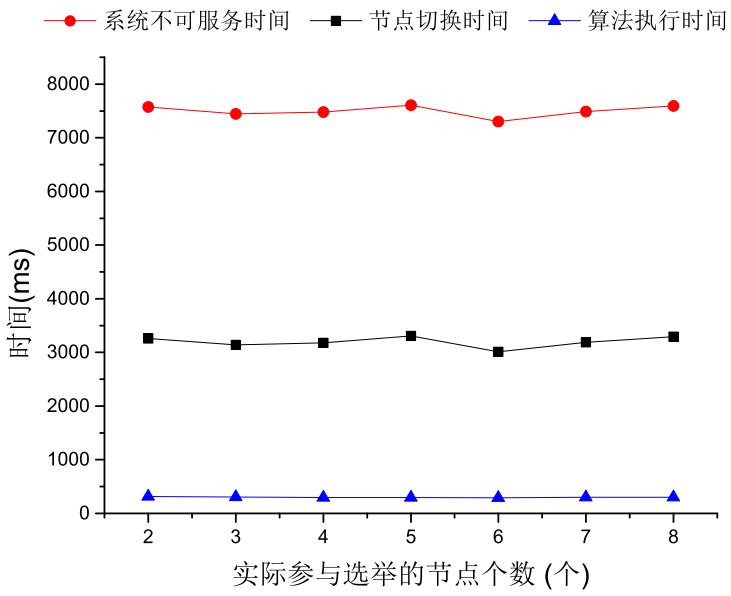

图 5.1: 不同参选RS个数下的故障恢复时间

时间高出一个数量级，而系统不可服务时间也同样在秒级区间内，说明本文提出的选举算法在对系统原始特性几乎没有影响的情况下，解决了此类分布式数据管理系统的单点故障问题，提高了可用性。

为了测试网络丢包异常对于LEBR算法的影响，我们使用Linux 的tc工具来模拟所有节点间网络的丢包情况，该工具可以针对指定的IP地址和端口设置丢包率。我们在配置5 台服务器的集群中进行实验。实验时设置算法随机等待时间区间为200ms-300ms，租约等待时间为4s。然后通过tc 工具制造不同丢包率，并在这种情况下进行多次故障模拟，测试选举相关时间，最终得到图5.2。图中很明显可以看出，随着网络丢包率的增加，算法执行时间和节点切换时间呈指数态增长，而系统不可服务时间作为前两个时间和节点租约时间的和，也同样呈指数增长，并且增长幅度更大。根据图中各时间随丢包率增长的变化趋势，当网络丢包率达到25% 时，整个系统不可服务时间就已经为15s；而当丢包率为40% 时，不可服务时间达到了45s，并且增长率越来越高。在实验过程中，继续增加丢包率进行测试已经很难得到有效的结果。丢包率为15%时，对于关键业务应用来说，理论上会

丢失15% 左右的请求和结果，系统已经无法正常提供服务，然而LEBR 算法的执行时间仍控制在1s之内，并且系统的不可服务时间也保持在可接受范围内。这说明了LEBR 算法在系统无法正常服务时，也能够高效、正确地执行。更为重要的一点是，无论丢包率设置为何值，LEBR 都保证了选举结束后主节点的唯一性，这在测试过程中得到了验证。

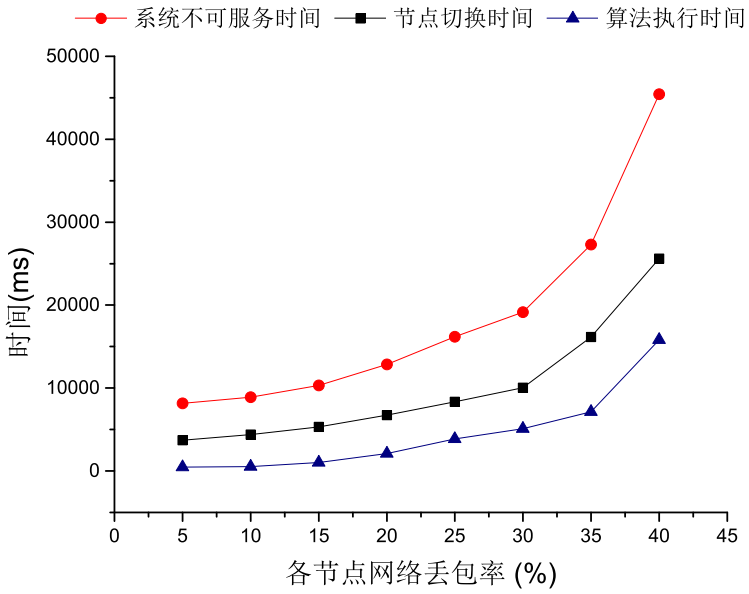

图 5.2: 出现网络丢包时故障恢复时间

图5.3展示了在数据库系统存在负载情况下，LEBR算法在5台服务器组成的集群中的运行情况。

这项测试中，我们使用YCSB 性能测试工具[55] 为系统添加读写比例为6:4 的负载，其参数设置见表5.3。通过调整YCSB 的线程数，可以产生不同大小的负载，用OPS （Operation per Second）表示。从图中可以看出，随着线程数的增减，数据库负载量也在不断上升，并且最终趋于稳定。而在整个过程中，我们在各个线程数下测试的LEBR相关时间基本维持不变。在本实验的环境下，LEBR算法的效率与数据库系统的负载无关，即使负载量达到最大，也不会对LEBR 造成影响。事实上，由于RS是总控节点，负载量大小对于数据库的压力不会落在RS上。因此

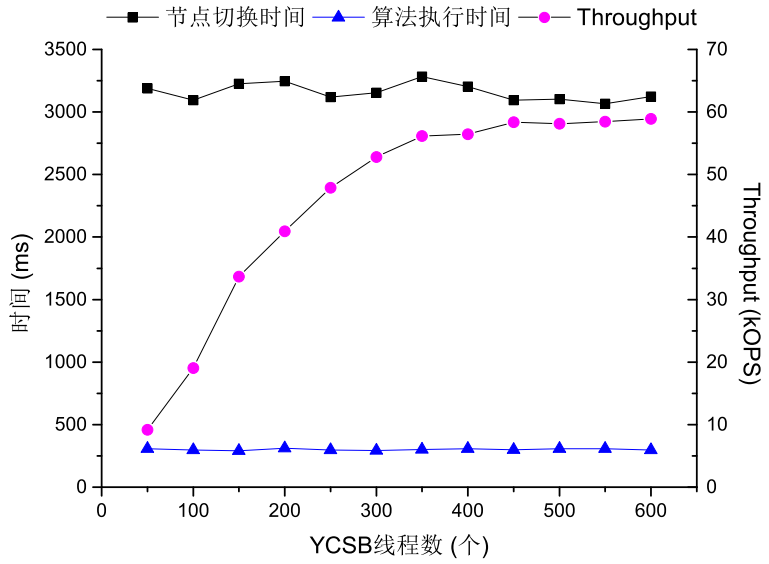

图 5.3: 数据库负载对LEBR 算法效率的影响

只要物理服务器仍有可用资源，RS 的运行就不会受到影响，在RS上实现的算法自然也不会受到影响。

表 5.3: YCSB参数设置

参数名称	参数值	参数意义
recordcount	5000000	测试表的记录数
readproportion	0.6	主键读的比例
updateproportion	0.4	主键更新的比例
threads	50600	测试线程数

OceanBase中，RS作为数据库的总控节点，不负责处理数据读写请求。故无法通过TPS （Transaction per Second）或QPS （Query per Second）的变化曲线来测试使用LEBR的情况下，主节点失效对于整个系统可服务性的影响。考虑到RS负责数据的表结构模式管理，建表、删表等DDL（Data Definition Language）操作必须通过主RS执行，于是可以利用RS的这个功能来进行测试。实验中我们部署了5台服务器，备RS 的租约时间设置为4s。然后持续向数据库发送建表请求，并且每次都重新连接数据库，最后统计每100ms 内成功的请求数。测试结果如图所示5.4。

该图显示出，RS 正常情况下平均没100ms 可以执行5次建表操作，然而当我们强制下线主RS 后，数据库将立即不能执行建表，直到新的主RS 被选举出来并接管集群。图中也可以看出，不能建表的持续时间大约为7s，与理论上的故障恢复时间（包括租约时间、算法执行时间和节点切换时间）大致相同。

实际应用中，大多数DDL操作在初次部署系统时执行。当系统上线后，一般只进行少量DDL操作，并且在系统空闲时执行。所以，LEBR 算法实现的秒级的故障检测和服务恢复时间能够满足RS对于可用性的需求。

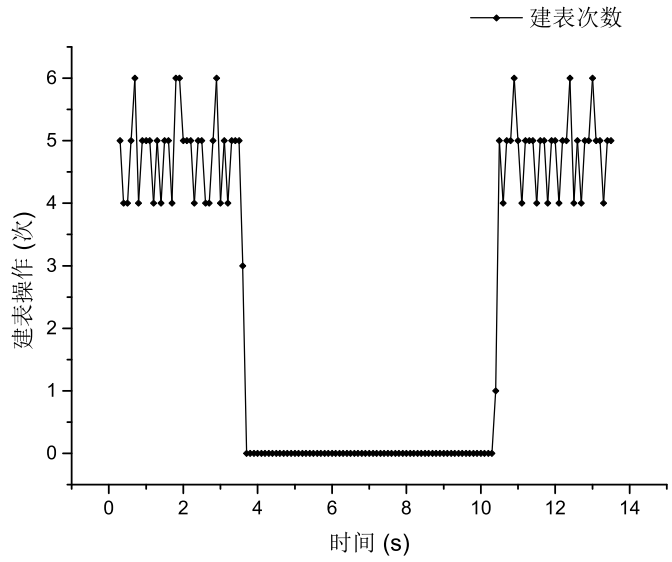

图 5.4: LEBR算法对于DDL操作的影响

5.2.2 MTS算法相关测试

进行MTS算法的相关测试时，为了得到只与本算法相关的准确结果，本文首先假设RS是稳定的，不会在测试过程中意外改变。其次，我们将UPS的租约时间设置为9s。实验时，我们同样了配置3-9 台服务器，每台服务器都上启动RS、UPS、MS 和CS共四个进程，并且服务器中无其它负载。每个UPS节点数量下进行100 次故障模拟测试，并将结果取平均值，实验结果如图5.3所示。结果显示，MTS 算法执行时间与备选的节点个数呈线性正相关，随着节点数量增加，执行时

间变长。但是这个执行时间的变化趋势在微秒（us）级别才能明显看出，而且在有8 个备选节点时，该时间仍仅为2.8ms。图5.5 中我们还可以看出，节点的切换时间维持在31ms左右，与备选节点个数无关。其实，通过分析OceanBase源码可以发现，影响UPS节点切换时间的因素与日志同步的实现有关，但这并不是本文所关注的内容，因此我们没有进行进一步的实验来探究UPS切换时间的影响因素。

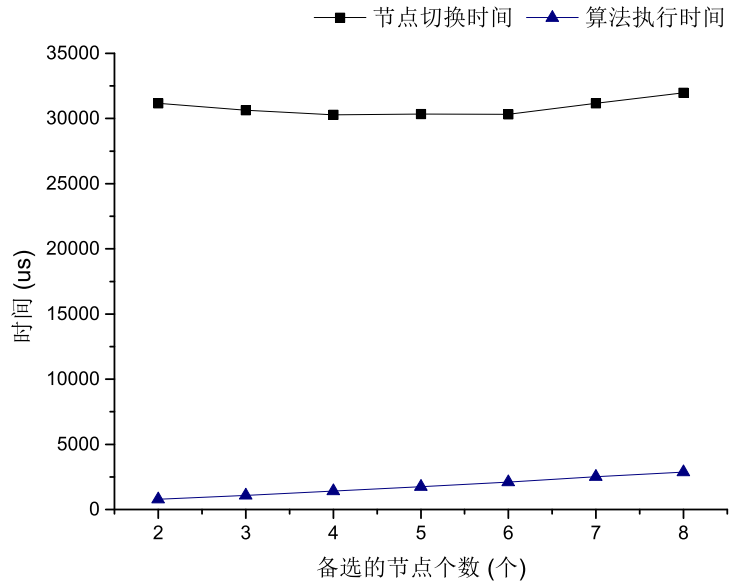

图 5.5: 不同备选UPS 个数下的故障恢复时间

另外，由于UPS实现了基于Quorum的日志同步，因此配置的UPS个数越多，则对于数据库系统事务处理性能的负影响越大，所以在实际场景中，通常只部署3-5 个UPS节点。这种情况下，MTS算法的执行时间相比于31ms左右的节点切换时间说，是可以忽略的。总体来说，系统无负载情况下MTS算法的效率十分高效，能够在发现主节点故障时迅速确定新的主节点。

对于MTS算法，我们也测试了数据库负载对于算法效率的影响。与5.2.1 节中数据库负载与LEBR运行效率关系的实验步骤相比，出了故障模拟时强制下线主UPS 而不是主RS，其余步骤完全一致。YCSB 的参数设置也参照表5.3。图5.6 所示的实验结果表明，随着YCSB 线程数的增加，数据库的负载量也不断上升，

虽然这会不断增加UPS的压力，但是MTS算法的执行时间和主备UPS 的角色切换都没有收到影响。

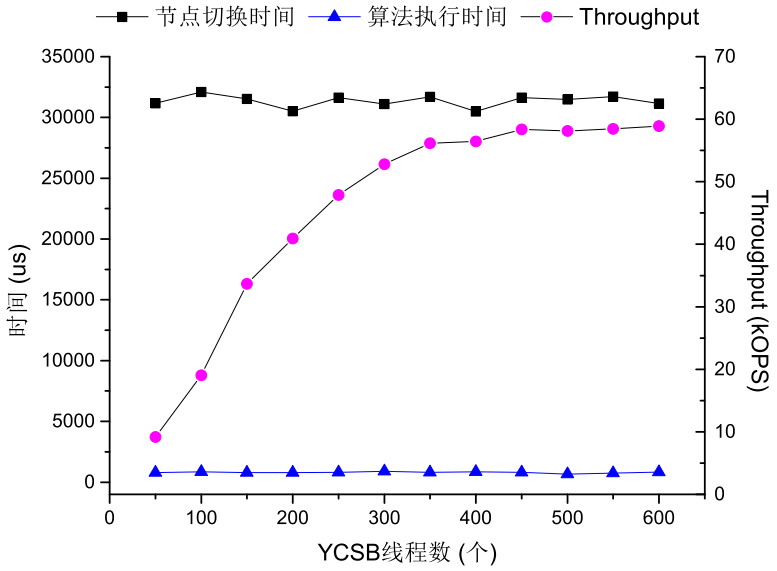

图 5.6: 数据库负载对MTS 算法效率的影响

下面的实验中，我们从用户使用的角度来测试使用MTS算法时，数据库系统的可用性。我们在5台服务器上启动UPS 等四个进程，共同组成了一个OceanBase集群。实验时使用Sysbench[56] 性能测试工具通过其中三个MS 节点给系统添加持续的负载。前面的实验中已经证明了负载对于MTS 选举没有影响，因此测试结果的准确性可以得到保证。我们使用Sysbench 而不是YCSB，是因为在Sysbench 中可以自定义SQL 语句，因此我们可以通过配置纯读和纯写SQL 语句来实现对QPS和TPS 的测试。并且Sysbench 能够每秒输出一次结果，利于观察TPS 和QPS 的变化曲线。参数设置见表5.4。图5.7 显示了配置单条等值查询SQL 时，Sysbench测出的60s内负载变化曲线。我们在这期间强制下线了主UPS，从图中可以看出，主UPS下线约9s 后，QPS 变化曲线从9000 上下急巨下降到5000 左右，但大约9s 后又重恢复到原有水平。这里面存在两个问题需要分析：

- 首先，QPS并没有在主UPS被强制下线的第一时间受到影响，而是过了一

段时间才发生变化。这是因为集群中的所有UPS都可以处理读请求，即使主UPS 失效，其它的备UPS 也能够提供读服务。从图中也可以看出，从主UPS 下线到QPS 受到影响大约经历了9s 时间，这与预设的UPS 租约时间相同。根据MTS 算法，主UPS下线后，RS 需要经过UPS 租约时间长度以上的时间后，才能确定这一事实，从而发起选择新主UPS 的流程。当新的主UPS 被选出后，所有UPS 都需要一段时间来认可新主UPS，并进行数据一致性维护工作，只有在这个时候无法继续提供读服务。而这些工作完成后，各个节点恢复正常，QPS 也恢复到下降前的水平。

- 其次，QPS下降后，并没有下降到0，尽管此时所有UPS 都不能访问。这种现象的原因是，OceanBase的MS 在运行过程中会缓存最近一段时间的查询结果，因此有些重复的查询请求能够命中MS 缓存而够得到结果。

表 5.4: Sysbench参数设置

参数名称	参数值	参数意义
table_size	1000000	数据库中每张测试表的记录数
thread_num	200	并发线程数

测试TPS时，我们在Sysbench上配置单行主键更新SQL。图5.7显示了执行Sysbench的前60s内TPS的变化曲线。在22s 左右时我们强制下线了主UPS，而此时TPS几乎瞬间下降为0。直到新的主UPS接管集群的事务处理，并正常工作时，TPS 才逐渐恢复到平均水平。可见主UPS 的故障对于写请求的影响非常大，整个无主UPS期间不能进行任何写操作。

通过实验发现，采用本文提出的MTS算法实现的事务处理节点高可用方法时，主节点故障对于写请求的影响力度和时间都要高于读请求，但在有限时间之后，当总控节点利用该算法选择出新的主事务处理节点并完全接管集群时，读写请求都能够恢复到故障前的水平。这充分说明了MTS 算法的有效及高效性。

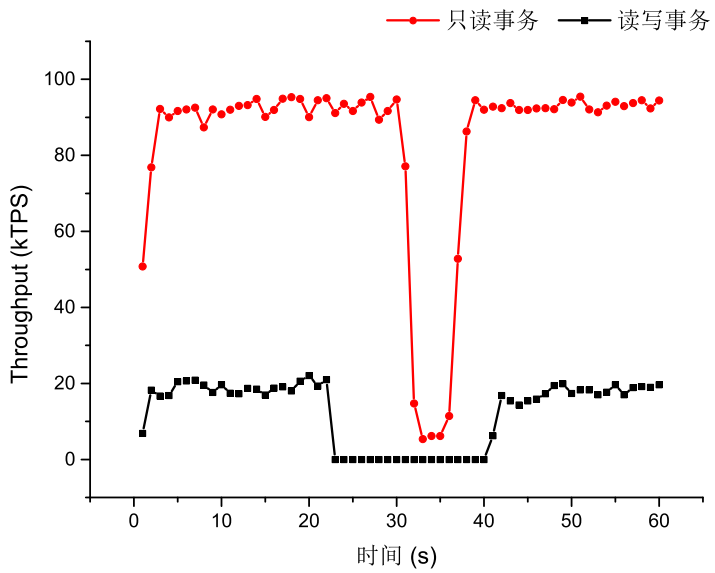

图 5.7: MTS算法对于读写访问的影响

5.3 本章小结

本章中，本文针对所提出的可扩展数据管理系统的高可用方法进行了测试。实验结果表明LEBR算法能够有效应对网络丢包异常，保证其结果具有唯一性。同时，LEBR 算法和MTS算法都能高效处理主节点故障的情况，实现了系统故障检测和快速的自动恢复。另外，本章实验还说明了在使用本文提出的高可用方法的可扩展数据管理系统中，主事务节点的故障对读写请求影响在秒级范围内，并且对于写请求的影响要高于读请求。

第六章　总结与展望

近年来随着互联网的高速发展和社会信息化进程的持续推进，很多行业都进行了互联网化转型。特别是金融行业中，大多数银行业务已经从传统的线下人工服务，转变到如今将很多包括核心业务在内的应用系统迁移到在线平台提供服务。但是，互联网化给人们带来巨大便利的同时，数据总量也在极大增长，数据增长速度也非常快。大规模数据对于数据存储和管理方式产生巨大挑战，传统的集中式数据库已经不堪负重。而逐渐成熟的分布式数据管理系统依靠其高可扩展性，将数据分布存储，能够支撑海量数据存储和处理。然而对于银行来说，其核心在线业务不仅需要底层数据库能够支持大规模数据存储，还应具有较高的系统可靠性和可用性，以保证业务持续且正常地进行。分布式数据管理系统为数据配置多个副本并分别存储在不同的物理节点上，基本能够确保已有数据不会丢失。此时，系统中唯一存在的关键节点的可用性决定了整个系统的可用性。提高这类节点的可用性，是实现分布式数据管理系统高可用的重要问题。

本文总结了可扩展数据管理系统的通用架构ScaleDDB，并以这个架构为基础，设计和实现了关键性节点的高可用方法。该方法能够避免单点故障问题，并实现了短时间的自动修复。ScaleDDB归纳了可扩展数据管理系统的特点，包括主从模式、横向扩展和读写分离。这些特点是类ScaleDDB的分布式数据管理系统能够支撑海量数据存储处理的重要因素。现有的NoSQL和NewSQL分布式数据管理系统都符合或部分符合这些特点。

本文的分析展示出，在ScaleDDB的总控节点CNode和事务处理节点TNode这两类关键节点中，存在单点问题，节点的失效将使系统无法正常使用，甚至导致

系统瘫痪。传统的集群高可用解决方案，例如Linux HA，由于对网络情况要求较高，在网络异常时容易失效，并且依赖于非ScaleDDB系统内的软件工具，存在安全隐患，不适应于金融系统关键应用的场景。而现有的分布式一致性协议，例如本文讨论的Raft，不适合无状态节点（即CNode节点）的选举，并且存在双主问题和频繁选举问题。这些问题虽然在Raft本身的协议框架下不会产生影响，但对于实际应用中的系统影响是巨大的。因此Raft 不能被直接用作实现CNode的主节点选举，以满足银行等企业的关键应用的高可用需求。

本文针对这一问题，结合ScaleDDB的整体架构特点，提出了一种用于总控节点和事务处理节点高可用框架，能够避免这两类节点的单点故障问题。该框架包括分布式选举算法LEBR和主事务节点选举算法MTS。其中，LEBR算法避免了双主和频繁选举问题，能够支持无状态节点选举，并实现了快速的状态切换；而MTS 算法则基于CNode的唯一性和稳定性，解决了TNode 节点的主节点选举问题。两种算法共同构成了ScaleDDB的高可用方法。

通过在分布式数据库OceanBase中实现本文提出的高可用方法，并在银行真实环境下进行了大量实验，测试并验证了本文提出的高可用方法。实验结果确认了本文提出的算法的正确性和较高的执行效率，实现了当主节点故障时，系统可以在秒级时间内迅速恢复并继续提供服务。

综上所述，本文针对可扩展数据管理系统高可用问题，设计并实现的解决单点故障并实现系统自动恢复的高可用方法，保证了总控节点和事务处理节点这两类关键节点的高可用。这不仅满足了金融行业关键业务对于可靠性和可用性的要求，还有利于分布式可扩展数据管理系统在金融等领域的推广，同时也为我国实现金融等重要行业数据安全和自主可控做出了贡献。

分布式可扩展数据管理系统在传统行业互联网化的背景下，其应用范围不再局限于互联网行业，而将变得更加广阔，同时，也将面对更加复杂的应用场景和需求。研究这类系统的高可用性是一项长期的、持续的工作。互联网应用背景下的分布式系统通常为了获取高可用性，而牺牲了一致性，然而当面对金融等对于

可用性和一致性都有很高要求的应用场景时，以前的高可用方案将不再适用。因此，在保持一致性的前提下实现高可用性，是该领域现在和未来都在追求的目标。本文的研究工作也正式朝着这个目标而开展的。

在未来的工作中，我们将会对本文提出的高可用方做进一步完善。首先，我们考虑到有些可扩展数据管理系统中，总控节点是有状态的节点。因此我们将完善LEBR算法，使其既能够实现无状态主节点的选举，也能兼容有状态节点的选举。另外，我们也希望能够使LEBR算法不再依赖于节点间的时钟同步，这样将使LEBR算法可以适应于更多的场景，成为更加自由灵活的选举算法。然后，我们将对MTS算法进一步扩展，使其能够支持多节点更新的架构的分布式数据管理系统的事务节点选举。最后，我们希望实现节点故障后，系统无缝切换主备节点。这不仅需要进一步提高主节点选举的效率，还需要研究系统的具体实现以提高系统本身的切换时间。这项工作已经不仅仅是主节点选举工作的内容，而是从整个系统多个方面来进行研究。

总之，我们将在今后的工作中致力于提高本文算法的效率，并扩展其适用范围，同时能够处理更多的故障场景，最终成为高效率、低成本且适用性强的分布式数据管理系统高可用方案，并应用到更多的实际应用系统中。

参考文献

[1] Oracle. MySQL Cluster. https://dev.mysql.com/doc/refman/5.6/en/mysql-cluster-overview.html. [Online; accessed 10-March-2016].

[2] IBM. DB2 pureScale. http://www.ibm.com/software/data/db2/linux-unix-windows/purescale. [Online; accessed 10-March-2016].

[3] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. Operating Systems Review, 44(2):35–40, 2010.

[4] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. ACM Trans. Comput. Syst., 31(3):8, 2013.

[5] 阳振坤. Oceanbase关系数据库架构. 华东师范大学学报（自然科学版）, 6(5):141–148, 2014.

[6] Fred J. Maryanski. A survey of developments in distributed data base management systems. IEEE Computer, 11(2):28–38, 1978.

[7] M. Tamer Özsu and Patrick Valduriez. Principles of Distributed Database Systems, Third Edition. Springer, 2011.

[8] 徐俊刚，邵佩英. 分布式数据库系统及其应用. 科学出版社, 2012.

[9] Oracle. Oracle Real Application Clusters. http://www.oracle.com/us/products/database/options/real-application-clusters/. [Online; accessed 10-March-2016].

[10] Oracle. MySQL Fabric. https://www.mysql.com/products/enterprise/fabric.html. [Online; accessed 10-March-2016].

[11] Microsoft. Microsoft SQL Azure. https://azure.microsoft.com/en-us/services/sql-database/. [Online; accessed 10-March-2016].

[12] Apache. Apache HBase. https://hbase.apache.org. [Online; accessed 15-March-2016].

[13] Katarina Grolinger, Wilson A. Higashino, Abhinav Tiwari, and Miriam AM Capretz. Data management in cloud environments: Nosql and newsql data stores. Journal of Cloud Computing: Advances, Systems and Applications, 2(1):1–24, 2013.

[14] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). Acta Inf., 33(4):351–385, 1996.

[15] ABM Moniruzzaman and Syed Akhter Hossain. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. arXiv preprint arXiv:1307.0191, 2013.

[16] Wikipedia. NewSQL. https://en.wikipedia.org/wiki/NewSQL. [Online; accessed 14-March-2016].

[17] Rick Cattell. Scalable SQL and nosql data stores. SIGMOD Record, 39(4):12–27, 2010.

[18] Jaroslav Pokorný. Nosql databases: a step to database scalability in web environment. IJWIS, 9(1):69–82, 2013.

[19] Sherif Sakr, Anna Liu, Daniel M. Batista, and Mohammad Alomari. A survey of large scale data management approaches in cloud environments. IEEE Communications Surveys and Tutorials, 13(3):311–336, 2011.

[20] Yuri Gurevich. Comparative Survey of NoSQL/NewSQL DB Systems. PhD thesis, The Open University, 2015.

[21] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. Data management in cloud environments: Nosql and newsql data stores. Journal of Cloud Computing: advances, systems and applications, 2(1):22, 2013.

[22] Robin Hecht and Stefan Jablonski. Nosql evaluation: A use case oriented survey. 2011.

[23] LinkedIn. Project Voldemort. `http://www.project-voldemort.com`. [Online; accessed 14-March-2016].

[24] Basho Technologies. Apache Cassandra. `http://docs.basho.com/riak/latest/`. [Online; accessed 14-March-2016].

[25] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997, pages 654–663, 1997.

[26] Salvatore Sanfilippo. Redis. http://redis.io. [Online; accessed 14-March-2016].

[27] Amazon. Amazon DynamoDB. https://aws.amazon.com/dynamodb/. [Online; accessed 14-March-2016].

[28] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst., 26(2), 2008.

[29] Google. Cloud Bigtable. https://cloud.google.com/bigtable/. [Online; accessed 14-March-2016].

[30] Avinash Lakshman and Prashant Malik. Apache Cassandra. http://www.aquafold.com/dbspecific/apache_cassandra_client.html. [Online; accessed 14-March-2016].

[31] Clustrix. ClustrixDB. http://www.clustrix.com. [Online; accessed 14-March-2016].

[32] MemSQL Inc. MemSQL. http://www.memsql.com. [Online; accessed 14-March-2016].

[33] Leslie Lamport. Paxos made simple, fast, and byzantine. In Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002, pages 7–9, 2002.

[34] Michael Stonebraker, Sam Madden, and Daniel Abadi. VoltDB. https://voltdb.com. [Online; accessed 14-March-2016].

[35] 杨传辉. 大规模分布式存储系统原理解析与架构实现. 机械工业出版社, 2013.

[36] 于雪平，孟丹. 数据库应用的高可用性及实现技术. 计算机应用研究, 21(5):5–8, 2004.

[37] Eric A. Brewer. Towards robust distributed systems (abstract). In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA., page 7, 2000.

[38] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News, 33(2):51–59, 2002.

[39] Andrew S. Tanenbaum and Maarten Van Steen. Distributed systems principles and paradigms. Acm Trans.prog.lang.syst, 87(3):65–73, 2002.

[40] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings, pages 144–154, 1981.

[41] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Hat, not CAP: towards highly available transactions. In 14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013, 2013.

[42] Marius Cristian Mazilu. Database replication. Database Systems Journal, 1(2):33–38, 2010.

[43] Alain Azagury, Danny Dolev, Gera Goft, John Marberg, and Julian Satran. Highly available cluster: A case study. In Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on, pages 404–413. IEEE, 1994.

[44] Wikipedia. High-availability cluster. `https://en.wikipedia.org/wiki/High-availability_cluster`. [Online; accessed 14-March-2016].

[45] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In 2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014., pages 305–319, 2014.

[46] Hector Garcia-Molina. Elections in a distributed computing system. Computers, IEEE Transactions on, 100(1):48–59, 1982.

[47] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011, pages 245–256, 2011.

[48] Salvatore Sanfilippo. Redis. `http://redis.io/topics/cluster-spec/`. [Online; accessed 14-March-2016].

[49] Philip A. Bernstein and Nathan Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. ACM Trans. Database Syst., 9(4):596–615, 1984.

[50] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009, pages 312–313, 2009.

[51] Leslie Lamport. Fast paxos. Distributed Computing, 19(2):79–103, 2006.

[52] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In 7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA, pages 335–350, 2006.

[53] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In Proceedings of the 1996 ACM SIGMOD International

Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996., pages 173–182, 1996.

[54] Dale Skeen. A quorum-based commit protocol. In Berkeley Workshop, pages 69–80, 1982.

[55] Yahoo!. Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB/. [Online; accessed 31-March-2016].

[56] Github. Sysbench. https://github.com/akopytov/sysbench/. [Online; accessed 31-March-2016].

致　　谢

时光荏苒，我的研究生生涯即将结束，心情激荡而不舍。又回想其大学时彻夜备战考研时的艰辛，和最后手捧华东师范大学研究生录取通知书时的喜悦，转眼间已经过去了三年。入学之初，看到美丽的校园，明亮的实验室，亲切地同学和学识渊博的老师，我对新的学习生活充满期待。在海量所的科研学习过程中，我逐渐褪去大学时的稚嫩，培养了对待问题时科学而严谨的态度，此外，我的学习能力、学术精神、个人品质和为人处世能力等都得到了提升。对此，我十分感谢学校各位任课老师在课堂上的授业解惑，感谢实验室同学和学长的帮助，感谢海量所老师们的指导。

我非常感谢海量所的许多同学和学长。感谢宋乐怡、郭心语、王克强、纪文迪等学长学姐，他们在我研究生生涯之初给予我非常大的帮助；感谢Oceanbase项目组的成员们，感谢郭进伟、朱涛、周欢、翁海星、樊秋实、张晨东、刘骁等，在一起进行项目研究的日子里，他们给予我很多学习和生活上的支持，我也从他们身上学习到很多优秀的品质；感谢程文亮、苏永浩、马建松、江俊文、李叶等同学对我的关心和帮助。海量所虽然实验室众多，但我们所有人都是一个团队，每个人也都认同和维护着这个团队，并且乐于互相帮助。身处其中，我感受到温暖和快乐，对于在此度过三年的研究生生活感到幸运和满足。

我特别感谢我的导师宫学庆老师，他不仅在科研学习上给予我指导和帮助，并且在生活上处处关心我。我曾经有一段时间在学习上陷入困惑，停顿不前，宫老师语重心长地与我交流，耐心为我分析梳理问题，最后帮助我走出了困境，对此我铭记在心。研究生期间，我与几位同学随宫老师在交通银行进行项目开发，宫老师丰富的学识、严谨的工作态度和工作能力令人钦佩，潜移默化地影响着我

们。此外，宫老师用心指导我完成了科研论文，并成功发表，也教会了我如何总结平日所学所思，行文成章。我还要特别感谢钱卫宁老师对于我毕业论文不辞辛苦地悉心指导，帮助我顺利完成学业论文，也感谢钱老师对我科研工作和日常生活的关心和帮助。此外，我非常感谢周傲英老师对我们的谆谆教诲，周老师充满智慧和感染力的讲演令人印象深刻，时常让人激荡不已，并且能够学习到很多有用的思想和知识。非常感谢王晓玲、何晓丰、金澈清、张蓉、张召、高明、蔡鹏等各位老师的关心和帮助，对待海量所每一位同学都十分用心，对于知识倾囊相授，并为我们提供了各种各样的学习和实践机会。

我还要感谢始终支持着我的父母，他们使我在生活上无后顾之忧，能够全身心投入到研究生的学习中去。从小到大，父母为我的成长付出辛勤的汗水，他们总是尊重我的决定，给我自由成长的空间，为我创造了良好的生活环境，并支持着我完成学业。我对父母的感激之情无以言表，唯有今后努力工作，膝下尽孝来回报他们的养育之恩。

最后，我衷心感谢所有关心和帮助过我的朋友们，谢谢!

二零一六年三月三十日

攻读硕士学位期间发表论文和科研情况

■ 已发表或录用的论文

[1] **庞天泽**, 张晨东, 高明, 宫学庆. 分布式环境中数据库模式设计实践[J]. 华东师范大学学报(自然科学版）, 6(5):290-300, 2014. (CSCD)

■ 参与的科研课题

[1]国家自然科学基金重点项目（U1401256），2014－2018，参加人

[2]国家高技术研究发展计划(863计划)课题，基于内存计算的数据管理系统研究与开发（2015AA015307），2015－2017，参加人