2017 届硕士专业学位研究生学位论文（全日制研究生）

分类号：_____ 学校代码：10269

密　级：_____ 学　　号：51141500091

華東師範大學

East China Normal University

硕士专业学位论文

MASTRER'S DEGREE THESIS (PROFESSIONAL)

论文题目：可扩展数据管理系统存储过程设计与实现

院 系 名 称：	计算机科学与软件工程学院
专业学位类别：	工程硕士
专业学位领域：	软件工程
指 导 教 师：	周傲英 教授
学 位 申 请 人：	祝君

2016 年 10 月 10 日

Dissertation for professional master's degree in 2017

University Code: 10269

Student ID: 51141500091

East China Normal University

Title: **THE DESIGN AND IMPLEMENTATION OF STORED PROCEDURE IN SCALABLE DBMS**

Department:	**School of Computer Science and Software Engineering**
Professional degree category:	**Master of Engineering**
Professional degree field:	**Software Engineering**
Supervisor:	**Prof. ZHOU Aoying**
Candidate:	**ZHU Jun**

October, 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存储过程设计与实现》，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存储过程设计与实现》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，本论文的研究成果归华东师范大学所有。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，并向主管部门和相关机构如国家图书馆、中信所和“知网”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）。

祝君 硕士专业学位论文答辩委员会成员名单

姓名	职称	单位	备注
钱卫宁	教授	华东师范大学	主席
翁楚良	教授	华东师范大学	
张蓉	副教授	华东师范大学	
蔡鹏	副教授	华东师范大学	
沙朝锋	副教授	复旦大学	

摘要

随着互联网产业的不断发展，其数据增长速度以及其并发访问规模都对传统的数据库管理系统产生了巨大的挑战。企业或机构为了提高自己的数据管理和业务支撑能力，正在把注意力从集中式数据库管理系统的使用转向可扩展分布式数据库系统的研发。

虽然分布式数据库是解决海量数据存储和高并发访问的有效手段之一，但是由于其应用和推广较晚，造成大部分该类数据库管理系统的功能还不完善。Oceanbase 是阿里巴巴开发的可扩展并行分布式数据库系统，尽管可以高效地解决其内部业务的数据管理和访问要求，但是其仍然不是一个功能完善的系统，造成它很难迁移到新的业务平台。本文以 Oceanbase 为研究对象，设计在其框架下的分布式存储过程，并进行实现；通过充足的实验对实现的存储过程进行测试，验证其可用性。本文的主要贡献总结如下：

1. 运行时管理：解决了可扩展数据管理系统 OceanBase 不支持过程式 SQL 语言的问题。设计了过程式 SQL 语言 PL/cedarSQL；实现支持 PL/cedarSQL 语言的编译器和解释器；使用运行时状态解决解释执行过程中变量作用域控制问题。
2. 存储过程管理：设计了存储过程管理模块，实现存储过程的存储、创建、执行问题。 使用系统表持久化存储过程的源码，并采用缓存机制将存储过程源码和中间代码缓存在内存中，提高存储过程在分布式环境下的执行速度。
3. 缓存管理：设计了在分布式环境下的同步和异常恢复策略，保证在集群内的缓存和集群间存储过程的一致性，并且在异常情况下能够恢复正常。

本文在 OceanBase 上实现了存储过程功能，并开展了丰富的实验验证本文设计方案的可靠性和可用性。

关键词：分布式数据库，PL/cedarSQL，存储过程，解释执行，一致性；

Abstract

With the continuous development of the Internet industry, the data growth rate and concurrent access scale have a huge challenge on the traditional database management system. In order to improve their data management and business support abilities, Enterprises or organizations are change their attention from the use of centralized database management system to the development of scalable distributed database systems.

The distributed database is one of the effective means to solve the problem of massive data storage and high concurrent access. However, its application and promotion is so late that most of the functions of the database management system are not complete. OceanBase is a scalable distributed database system developed by alibaba, which can efficiently solve the data management and access requirements of its internal business, but it is still not a fully functional system, making it difficult to be migrated to the new business platforms. This paper takes OceanBase as the research object, designed and implemented the distributed stored procedure under its framework. The feasibility of stored procedure is verified by enough experiments. The main contributions of this paper are summarized as follows:

1. Run-time management: solve the problem that the scalable data management system OceanBase do not support Procedural Language/SQL. design a SQL language named PL\cedarSQL, design a PL\cedarSQL language supporting compiler and interpreter. solve variable scope problem in runtime environment.
2. Stored procedure management: design stored procedure management module, achieve stored procedure storage, creation, execution problem. use the system table to persist the stored procedure source code and cache the stored procedure

source code and intermediate code in memory to improve the execution speed of the stored procedure in the distributed environment.

3. Cache management: design synchronization and exception recovery strategy in distributed environment to ensure the cache consistency in cluster and stored procedure consistency between clusters and can return to normal under unusual circumstances.

This paper implemented the function of stored procedure in OceanBase and carried out a lot of experiments to verify the reliability and usability of the design .

Key words: Distributed Database, PL\cedarSQL, Interpretation Execution, Stored Procedure, Consistency;

目录

插图

表格

第一章 绪论

1.1 研究背景

随着互联网的迅速发展，传统的生活方式也在逐渐变化，10 年前人们在电脑上使用网络收发邮件、看新闻，现在人们在智能手机使用移动网络发语音、看视频、网购、移动支付等。科技的进步带来了生活方式的变化，但是由于移动互联网用户的极速增长，如：微信 2016 年 Q1 月活跃数达到 8.06 亿[1]。在互联网中各类应用产生的数据越来越多，结构也越来越复杂，传统的集中式数据库管理系统难以处理如此巨大的数据，海量数据的存储和分析成为了新的研究方向，同时也为数据管理系统的发展带来了新的机遇。数据库的发展经历了层次数据库、网状数据库、关系数据库等几个发展阶段，数据库的功能也随着技术的发展越来越完善。关系数据库是迄今为止应用最为广泛的数据库，大部分企业都是用关系型数据库来满足业务需求，但是传统的集中式数据库的可扩展性和高并发受单台服务器计算能力以及存储容量的限制，不可避免的成为瓶颈。

在数据库压力增大性能降低的时候，传统的方式是通过增加服务器的内存、硬盘，更换多核的 CPU 以及更高带宽的网卡等来增加服务器的处理能力，从而提高数据库的性能，这种方式并不能根本解决问题甚至有时候并不起任何作用，而且硬件采购周期和更换时间比较长，并且在高峰期结束过后新增硬件资源得不到充分利用，造成资源浪费。另一种解决方案是采用分片的方式来解决高速增长的业务，依据业务特点对数据库进行不同维度拆分，实现数据表的水平切分/垂直切分[2]，存储到不同的数据库服务器上。垂直分表是将关系型数据库中的表拆分为更细粒度的表即小表，一般情况下一张表中的所有字段的使用频率是不一样的，有的字段使用率很高而有的字段使用率却很低，这时候将使用频率较低的字段使用另外的一张表来进行存储，每一条记录和主表有相同的键值，这样可以减少主表中的数据量，在查询的时候减少 I/O 次数提高查询速度。水平分库分表是将一

个表中的数据横向划分为多个片段分布到不同数据库的数据表中，降低单表的数据量，常见形式是按照主键或者时间等字段进行分割。虽然分片方式在高并发和海量数据场景下能够有效缓解单机和单库的性能瓶颈和压力，但随着业务和数据量不断增长，需要不断地增加服务器以实现更细粒度的数据表切分，这种方法需要大量的人工维护成本，并且会带来一些新的问题，如垂直分表只能在定义数据库结构的时候确定，在查询所有列的时候需要 join 多张表，而水平分库分表，将会带来跨分片的复杂查询和事务。

随着计算机技术的发展，分布式系统的理论逐渐成熟，为了满足对系统性能、成本以及扩展性的新需求，一大批新的分布式数据管理系统，如以 HBase[3]、MongoDB[4]、Bigtable[5]等为代表的 NoSQL[6]数据库和以 VoltDB[7]、Spanner[8]以及 OceanBase[9]为代表的 NewSQL[10]分布式数据库如雨后春笋般不断涌现出来。分布式数据库的出现，解决了企业面对海量数据的窘境，为海量数据的管理带来了希望，分布式数据库以其良好的可扩展性和容错性迅速在数据库领域占得一席之地。相比于传统的关系型数据库来说这类系统虽然拥有较强的数据存储和管理能力，但是 NoSQL 数据库为了解决性能和扩展性问题，放弃了强一致性、分布式事务以及传统的 SQL 功能的支持，NewSQL 虽然在事务处理能力以及 SQL 支持上强于 NoSQL，但是在其它基础能力方面依然所不足，因此这类系统很难被直接应用于银行业务系统等传统的大型信息系统中。例如，在传统数据库中提供存储过程技术[11]来实现对数据库的编程控制，但是这一技术在分布式数据库中并没有得到广泛的支持。

虽然在网络和硬件性能高速发展的今天，传统的存储过程技术所带来的性能提升已经显得微不足道，但是存储过程作为传统的技术已经广泛的在企业和银行的使用和支持。如在当前的主流的数据库，如 Oracle[12]，DB2[13]，PostgreSQL[14,15]等，都实现了存储过程机制。在关系型数据库广泛使用的今天，分布式数据库作为新事物的出现吸引了一大批企业和用户准备将现有的业务迁移到分布式数据库中，但是对于使用存储过程的业务来说，迁移意味着在不支持存储过程的系统中需要在应用层实现业务逻辑代码，这对于系统的迁移造成了极

大困难，增加了迁移成本和工作量。因此，在分布式数据库系统中对存储过程的支持是很有必要的。表 1.1 中列出了一些 NewSQL 数据库对 SQL 以及存储过程的支持情况。

表 1.1 分布式数据库中存储过程支持情况

系统名称	SQL 支持	存储过程支持	支持语言
Google Spanner	√	×	
VoltDB	√	√	Java
ClustrixDB	√(部分不支持)	√	SQL/PSM
MemSQL	√	×	
OceanBase	√(部分不支持)	×	

由表 1.1 可知在 NewSQL 中对 SQL 的支持是比较完备的，虽然其中一部分支持存储过程但是还是有很多不支持存储过程特性的。Oceanbase 是阿里巴巴集团为了满足天猫、淘宝以及支付宝等业务的需求而研发出的可扩展数据管理系统，将分布式存储和关系型数据库功能紧密结合在一起，主要面向互联网企业级应用，满足互联网行业对于海量数据存储和查询的需求。OceanBase 可以实现数千亿条记录、数百 TB 数据的跨行跨表事务,目前已经广泛应用于淘宝、天猫和支付宝等多个线上系统[16]。OceanBase 实现了关系数据库的重要特征，也支持 SQL 语言查询，但是和主流的关系型数据库系统 PostgreSQL、MySQL 等相比较，功能上还存在一些不足的地方，如：不支持存储过程和游标[17]等。存储过程在现代企业中应用十分广泛，大多数企业的业务逻辑都采用该技术实现，而 OceanBase 作为关系型数据库想要在企业和机构中被广泛的应用，就需要支持存储过程，以便于企业和机构将存储过程所写的业务能够顺利的迁移到 OceanBase 中。

1.2 本文工作

本文通过深入分析主流的开源数据库系统的存储过程功能，对存储过程实现的一些关键技术进行研究，并且在分布式的环境下进行实践与探索，并结合 OceanBase 数据库的架构以及其现有的 SQL 引擎，提出了一种适合 OceanBase 数据库架构的存储过程设计和实现方案，对分布式环境下的一些异常情况进行分析与解决，保证了分布式环境下的可用性。本文的具体贡献如下所示：

1. **PL/cedarSQL 运行环境**。为了实现存储过程机制，本文定义了一种名为

PL/cedarSQL 的过程式 SQL 语言，并实现了该语言运行环境，包括编译器和解释器两部分。其中编译器主要包括词法解析，语法解析以及中间代码生成。解释器主要对生成的中间代码进行解释执行，利用哈希表和栈设计了运行时状态，不仅获得了对变量良好的查询效率，还解决了嵌套语句块中的变量的可见性问题。

2. **存储过程管理**。对存储过程进行管理，包括创建、存储、删除、执行等。通过增加系统表实现对存储过程的持久化，通过缓存机制减少了存储过程对存储过程源码的编译，提高存储过程执行速度。
3. **分布式缓存管理**。在优化存储过程的时候使用了缓存，所带来的问题就是如何保证在集群内部缓存的一致，以及多个集群之间的存储过程如何保持一致。实现上，在集群内部采用心跳机制和异步推送方式来进行同步，在不同的集群之间利用集群间的日志同步机制实现存储过程的同步。另外，并给出了在异常情况下的解决方法。

1.3 本文结构

根据本文的研究内容，本文的结构安排如下：

第二章介绍了本文研究内容的相关工作和需要解决的问题。首先介绍了可扩展数据管理系统的发展历程、分布式数据库系统的特征和分类，然后介绍了 OceanBase 数据库的架构及原理，最后介绍了在 OceanBase 中为了实现存储过程需要解决的问题。

第三章首先介绍了存储过程运行环境 PL/cedarSQL 引擎的架构，介绍了引擎的编译部分，以及解释器的原理和相关算法。并且对编译过程生成的中间代码结构进行分析，设计并实现了运行时状态来解决变量作用域的问题。

第四章具体的介绍了存储过程管理模块的架构，以及介绍了存储过程的创建、存储、执行的流程，并对输入输出参数的实现原理进行分析。

第五章对存储过程模块中使用缓存来进行优化所带来的缓存数据同步和可能不一致的问题进行分析和解决，主要包含三个问题：1、集群内的同步问题；2、集群间同步问题；3、异常情况下的解决方法。并且对分布式环境下创建存储

过程带来的延迟生效的问题进行了详细分析。

第六章介绍了使用 OLTPBench 对存储过程的性能进行测试，分别使用不同数量的客户端以及 warehouse 进行测试，分析不同情况的 TPS 和 Latency。

第七章总结本文的研究内容，并提出了一些待解决的研究内容。

第二章 相关工作与问题描述

2.1 可扩展数据管理系统

在数据量急速膨胀以及数据类型不断增多的大数据时代，传统关系型数据库为了满足互联网应用对性能以及可靠性的苛刻要求，只能通过不断的加强硬件配置来满足需求。这种方式过度依赖硬件，而且高端的存储设备和服务器的价格往往很贵，并不是所有企业都能够负担的起的。互联网的业务往往增长迅速或者会有特殊的高峰访问时段，这就要求数据库在可扩展性上一定要强并且能够动态的伸缩。分布式数据库的出现很好的解决了上面的问题，如今学术圈和企业都围绕着分布式系统的可用性和功能扩展展开了许多的研究。本节针对分布式数据库系统架构和分类问题展开分析调研工作，介绍当前比较流行的 NoSQL 和 NewSQL 这两种类型的分布式数据库及其特点。

2.2.1 分布式数据库简介

从个人计算机的发明到移动智能手机的今天，数据库在人们的生活中无处不发挥着作用，但是由于现在移动互联网以及物联网的迅速发展导致信息量剧增，对数据库系统的存储和管理提出了更多的挑战，传统集中式数据库系统在新的应用场景下的弊端逐步显现出来了，包括：

- 扩展性问题，传统的数据库是部署在单机上面的，存储容量和性能受单机存储能力和处理器计算能力的限制。当业务拓展快速数据库的性能不满足需求时，只能够通过增强硬件配置来改善数据库的性能。例如将机械硬盘换成 SSD 固态硬盘，更换性能更好的 CPU 以及采用速度更快的网卡等。这种方式在一般情况下能够起到一定作用，但是效果有限，甚至有的时候导致情况更糟。
- 可用性问题，单机部署的集中式的数据库若遇到网络故障或者宕机时意味着不能提供数据访问，应用程序处于不可用状态，只有通过人工干预

如重启数据库服务，或网络恢复后才可以继续提供服务。对于可靠性要求较高的政府部门或其他行业通常是采用可靠性高的高端硬件来保证数据库的可用性，但是采用高规格的硬件还是不能保证数据库的在任何时候都处于可用状态，并且高端硬件的价格不是一般小企业能够承受的。

为了解决集中式数据库的弊端，分布式数据库应运而生，并逐渐取代传统数据库的位置成为企业机构的新宠。与传统数据库相比，分布式数据库不在局限于单机，它通过网络和其它不同地理位置[18]的服务器进行连接，每个节点都有独立提供服务的能力。虽然由物理上分散的节点组成，但数据库管理系统将所有节点进行统一管理，这些节点被看做是一个整体并对外提供服务。分布式数据库所采用的这种架构带来了非常高的扩展性，使得数据库能够通过横向扩展提高性能，不在局限于提升硬件配置这种方法。在业务量上升时，直接通过增加节点的方式可以显著的提高性能，并且可以在业务量降低时减少机器，机动能力很强。

2.2.2 分布式数据库的分类

对于分布式数据库系统可以根据两种不同的方法进行分类：根据分布在不同网络中节点上数据库的数据模型类别进行分类；按照分布式数据库上的全局控制方式进行分类。

按照第一种的分类方法可以将分布式数据库分为两类[19]：同构型分布式数据库（Homogeneous DDBS）和异构型分布式数据库（Heterogenous DDBS）。前者是指构成分布式数据的多个服务器上的数据模型都是相同的（如都是关系模型数据库），如果不相同的类型则是异构的。按照第二种分类方法进行分类的话可以划分为三类：

- 全局控制集中型分布式数据库：数据在系统中的分布信息只保存在一个服务器节点上，并且该节点对系统进行全局控制，如基于切片技术的 MySQL Fabric、Microsoft SQL Azure 和 Oracle RAC 等。
- 全局控制分散型分布式数据库：每一个节点都包含全局控制信息的一个副本，并且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。

- 全局控制可变型分布式数据库：构成分布式数据库的节点根据是否包含全局控制信息分为两类，其中含有该数据的节点是主节点，剩余的称为副节点，主节点数至少为 1，但是主节点数要小于节点总数。

随着大数据库时代的来临，互联网行业迅速崛起，大量增长的互联网用户产生了大量的有价值的数据，互联网应用的数据类型种类丰富，数据来源也多种多样，包含大量结构化和非结构化的数据，而传统关系数据库适合存储结构化的数据，对于非结构化数据的存储关系型数据库显得力不从心，甚至传统架构的分布式数据对于这样的需求也显得捉襟见肘。为了迎接这些挑战，很多的互联网企业和机构开始从理论研究转向了工业实践，根据实际生产中所遇到的问题有针对性的进行研究，提出了新的分布式数据库架构以及数据的管理。它采用的是一种集群式的架构，集群内部通过高速网络相互联结[20]，不同节点存储的数据只是全部数据的一个子集，并且在在部分节点上相同的数据存在多个副本，这样可以规避单点故障，这种架构带来的好处是系统有极好的扩展性。因此，这种架构的数据系统又被称为可扩展数据管理系统（Scalable Data Management System）[20]。在互联网企业中对新型分布式数据库的开发的往往会针对企业中遇到的某种业务、数据类型以及其他一些问题进行优化，因此不同厂商所开发的系统有各自特点。

虽然新型分布式数据库的研究起步较晚，但是在开源社区的帮助下使得一些开源的系统发展迅速，目前有许多的较为成熟和稳定的开源可扩展数据管理系统，其中一些被广泛的用在电子商务、云计算、云存储等行业。根据对事务的支持能力又可以将这些系统分为两个类别，即 NoSQL(Not Only SQL)和 NewSQL。NoSQL 是指一类分布式的，不遵守传统关系数据库的范式约束的数据管理系统，不支持 SQL 查询和不支持 ACID 性质的事务处理，外部只能使用它提供的各种 API 接口进行操作，正是由于上述的特点使得 NoSQL 数据库在海量数据存储和扩展性上面有着出众的能力。而 NewSQL 数据库是介于关系型数据库和 NoSQL 数据库之间的一种解决方案，拥有传统关系数据库血统又有 NoSQL 的特点和优势，NewSQL 不仅支持事务处理，还提供 SQL 支持，用户可以像使用关系型数据库一样使用

NewSQL 数据库。

2.2.3 可扩展数据库 OceanBase

OceanBase 是 Alibaba 研发的支持海量数据存储的高性能分布式数据库，OceanBase 实现了关系型数据库的重要特征，将分布式存储和关系型数据库功能紧密的结合在了一起，具有可扩展、高可用、高可靠以及低成本等特性，能够满足互联网行业对于海量数据的存储、处理和查询需求。在架构上，OceanBase 采用了读写分离架构[21]，将数据的更新和存储分别使用不同节点处理，由一个中心节点管理集群。也就是将数据分为基准数据和增量数据，基准数据按照一定的划分规则进行分片，冗余的分布在不同的节点上。增量数据是指一段时间内对于数据的增量和修改，增量数据存储在内存中（Redo log 仍然存储在磁盘）又被称为 MemTable[16]。OceanBase 的架构如图 2.1 所示：

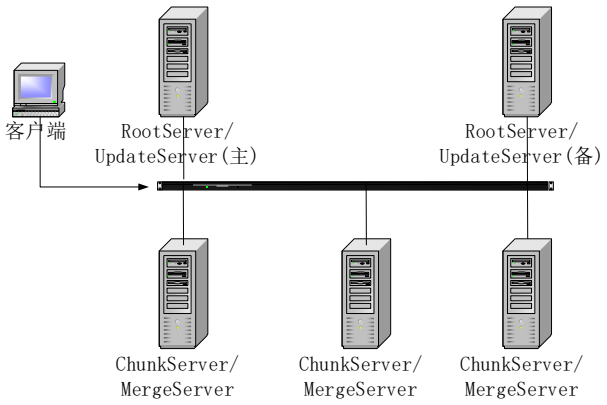


图 2.1 OceanBase 架构图

OceanBase 数据库系统由分布在网络中的四种不同类型服务器组成：

- RootServer：负责管理和控制系统中所有节点，如 ChunkServer 节点的上下线；发起 UpdateServer 的选主。管理系统中数据的分布，对数据进行负载均衡等。
- UpdateServer：存储系统的增量数据，是整个系统中唯一个能写入数据的模块，当增量数据大小超过一定阈值后，会生成快照文件存到磁盘中，提供事务支持。
- ChunkServer：存储系统中的基线数据，提供数据的读取服务，进行数据

分发和定期合并。

- MergeServer：处理来自用户的查询请求，根据请求内容通过词法解析和语法分析[22]并构建对应的逻辑查询计划和物理查询计划[23]，根据数据的分布信息，对查询所涉及到的数据所在的 ChunkServer 发起查询请求，将查询到的数据合并后返回给用户。

在一般情况下 RootServer 采用了一主一备的方案，主备之间采用数据强同步策略[16][51]，目的是规避单主节点宕机的风险。为了实现跨行跨表的事务，并且避免分布式事务，系统中只采用了一个 UpdateServer 来维护增量数据。同样的为了防止单节点出现异常时数据库服务不可用，UpdateServer 也采用主备的部署方案。由于 UpdateServer 在某些场景下会成为系统的瓶颈，为了保证系统的高性能，在主备之间采用的同步模式是可以配置的。同步模式包括强同步模式弱同步模式，其中弱同步模式主要是用于异地机房的容灾，弱同步可以达到最终一致性[24]。在 ChunkServer 中为了增加可用性防止在一个节点失效的时候导致数据不可用，将基线数据存储 3 份副本，也可以根据具体需要进行配置。

2.2 存储过程

存储过程是现代数据库管理系统为增加可编程性提供的一个重要特性，存储过程能够完成带有逻辑控制的任务，并且存储过程中提供对 SQL 的执行，就像 C 语言中可以直接嵌入汇编语言一样，能够方便的访问数据库。客户端调用存储过程的时候只需要传递需要调用的过程名字和参数给数据库，即可执行相应的存储过程，客户端和数据库服务器只进行两次网络通信，这种方式能有效的减少客户端和数据库的通讯次数，并且过程中对数据库的访问是本地的能够加快任务的执行。在定义存储过程的时候在过程体中可以定义变量，使用顺序，条件控制和循环控制等结构，并且可以定义存储过程的调用参数。虽然存储过程没有返回值，但是存储过程可以通过定义输出参数来实现返回结果的功能，这些特性都使得存储过程成为数据库系统必不可少的功能。

在一些比较复杂的业务逻辑中，程序需要和数据库进行多次的交互，每次交互的时候都会传送大量的临时的结果集，这不仅增加应用和数据库之间的网络通

信代价，而且在网络质量较差的时候还会影响其他正常用户的查询速度。因此，对于这类数据库访问密集的业务需求通常可以使用存储过程来完成[25]。存储过程所具有的优点如下：

1）上层应用只需要给出存储过程名称和参数，一般只需要两次网络通信，没有中间结果的传输，只返回最后执行结果。

2）可预先编译，运行速度快，存储过程源码只需要编译一次并将中间代码存入缓存中，下次执行时不需要再次编译。

3）对数据库的访问进行模块化封装，通过传递参数进行调用，可以防止普通的 SQL 注入式攻击，安全性得到提升。

虽然使用存储过程带来了以上的优点，但是存储过程也存在一些不可忽视的缺点：

1）存储过程调试困难，存储过程是解释执行的，并且在执行的过程中会涉及到数据库内部的 SQL 引擎，没有相应的调试工具能进行调试。因此，在编写存储过程的时候调试代码是比较困难的。

2）代码移植困难，在数据库技术发展的早期，SQL 标准中并没有对存储过程的规范做出定义，但是各大数据库厂商为了满足用户需求采用私有的语言开发了存储过程。不同厂商之间的存储过程是不兼容的，导致在企业用户切换数据库系统的时候不得不将原来的存储过程用新数据库的语言重写，增加了企业更换数据库的成本。

3）数据库服务器负担加大。存储过程虽然减少了程序和数据库之间的交互次数和网络通信量，但是将原本在应用层的一部分业务逻辑转移到了数据库端，这无疑增加数据库的负担[25]。例如，在没有使用存储过程的时候，应用层需要完成相应的逻辑和计算，数据库只需要返回应用层查询请求的结果集。如果采用存储过程的话，应用只需要简单的调用该过程，此时数据库不仅仅要进行数据的查询，还需要完成相应的流程控制和计算任务，并最终返回执行结果给上层应用。

2.3 编译原理

编译原理是现代计算机操作系统中的重要组成部分之一，大多数的计算机操

作系统都包含有多个高级语言的编译程序。高级语言和低级语言相比，高级语言的抽象能力更强，因此使用高级程序设计语言编写程序不仅方便，而且开发效率高。高级语言是易于人类理解的一种描述性语言，但是计算机只能执行二进制的程序，高级语言编写的源程序在计算机中是一种文本文件，不能够直接被计算机运行。为此，需要将由高级语言编写的程序用翻译程序翻译成为计算机能够识别的机器语言程序，这个翻译程序一般被称为编译程序或者编译器。编译程序的翻译转化工作如下图所示：

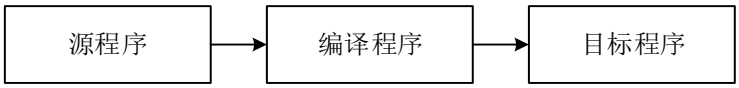


图 2.2 编译程序功能

编译是一个十分复杂的工作，按照不同任务可以将编译分为以下几个阶段：词法分析、语法分析、语义分析、中间代码生成优化、目标代码生成[26]。下面介绍这个几个阶段的主要任务：

1) 词法分析阶段

词法分析的任务是根据读入的源程序字符，按照语言的词法解析规则，将单词从句子中识别并分解出来，提供给后续的语法分析阶段使用。

2) 语法分析阶段

语法分析在词法分析之后进行，其任务是根据语言的语法规则来检查源程序中的语句是否符合语法规则。如果不符合语法规则就进行相应的错误处理，如显示错误类型、错误位置等。如果符合语法规则，则将单词符号组成表示各类结构的语法树节点。

3) 语义分析阶段

语法分析阶段只能够保证在语法上是符合规范的，不能确定其中的逻辑或者语义是否正确。语义分析的目标是保证常数和标识符的正确使用，保证标识符在全局中的含义一致，为代码生成阶段收集必要的信息、保存到符号表[27]或者中间代码库中，并进行相应的处理。例如，对赋值语句的左表达式和右表达式进行类型匹配检查，如果类型不匹配则报错。

4) 中间代码生成

有的编译器在完成语法分析和语义分析的工作之后，编译器会将源程序转换成为一种抽象的中间表示形式，这种表示形式被称为中间代码[28]。在选择何种中间表示形式时，通常有两个关键点：一是易于生成；二是能够轻松的将它翻译成为目标代码。常见的中间表示形式有三元式、四元式、逆波兰式、树形表示等[28]。

5) 代码优化阶段

在上一阶段生成了中间代码后，为了生成更小更高效的目标代码，会对生成的中间代码进行优化，如调整和改变一下代码的次序，尽量在时间和空间上得到充分的优化。

6) 目标代码生成

这是编程程序的最后阶段。这一阶段的任务是把中间代码或者优化过的中间代码转换成特定的目标语言的指令代码，例如机器指令或汇编指令[28]，它的工作与硬件系统结构和目标语言的语句有关。

上述的几个阶段是典型的编译程序的处理模型，并不是所有的编译程序都按照上述的阶段划分或者包含以上的所有处理阶段。编译器除了上述的内容外，还有一些其它的概念。

2.3.1 编译方式和解释方式

计算机语言的处理方式有两种，分别是编译方式和解释方式。编译方式和解释方式最大的区别是：在编译方式下，机器上运行的是与源程序等价的目标程序，在目标程序的执行过程中，源程序和编译程序不再进行编译过程；而解释方式下，在程序的运行过程中，解释程序和源程序（或者中间表示形式）需要不断的进行解析，程序的运行依赖于解释程序，控制权在解释程序。编译方式在可以根据硬件产生更高效的可执行代码，而解释执行方式可以通过在不同的平台实现解释器从而达到跨平台执行。

2.4 问题描述

存储过程的定义方式一般是通过使用过程化 SQL 语言定义或者使用动态语言

进行定义。OceanBase 本身采用 C++开发，如果使用 Java 或者 C#等动态语言定义需要增加虚拟机[29]或 CLR(公共语言运行库)支持，这种方式成本和复杂度较高。考虑到 OceanBase 内部已有 SQL 引擎，且对 SQL 支持比较完善，因此我们对存储过程的定义采用过程化 SQL 语言来实现。因此我们设计了一种名为 PL/cedarSQL 的过程化 SQL 语言，用于支持存储过程定义。

2.4.1 语法规范

OceanBase 所支持的 SQL 语法规范和 MySQL 相同，为了保持整体语法的一致性，在设计 PL/cedarSQL 语法的时候参考了部分 MySQL 的 SQL/PSM 的语法，而 MySQL 的存储程序遵循 SQL:2003 语法[30]。PL/cedarSQL 通过增加其它过程性语言中的结构对 SQL 进行拓展，主要包括以下几种结构：

- 块结构

语句块(Block)是构成 PL/cedarSQL 的基本单元。语句块能按照用户指定的次序执行。语句块中也可以根据逻辑关系进行嵌套，即一个语句块可以由其它语句块组成。针对具体变量的定义和执行流程控制的功能均可在单独的一个语句块，语法定义如下:

```
BEGIN
   [statement_list]
END
```

其中 statement_list 代表过程语句列表，可以是一条单独的语句也可以是多条语句。当 statement_list 表示多条语句的集合时，每条语句之间需要用(;)将他们独立开来。

- 变量声明

在 PL/cedarSQL 中可以声明数据库支持的任何数据类型的变量。声明后的变量可以被作用域范围内的其它语句引用。在一个语句块中变量的作用域服从局部性原则，即声明的变量只在当前块中以及块中的其它作用域内起作用。变量声明的语法如下所示：

```
DECLARE var_name[, var_name…] type [DEFAULT value];
```

变量支持的类型有：int，varchar，decimal，time 等。在声明变量时可以批量

声明多个同种类型的变量，并且可以有缺省值和初始值。PL/cedarSQL 通过赋值语句给变量赋值，语法格式如下所示：

```
SET var_name = value[, var_name = value]…;
```

赋值语句中 value 可以是一个表示值的表达式，可以是一个变量名，也可以是一个常量值，甚至可以是一元操作符或二元操作符构成的表达式。

- 控制结构

PL/cedarSQL 相比于一般 SQL 而言最大的改变就是在 SQL 的基础增加了控制结构。因此，PL/cedarSQL 不仅可以使用 SQL 语句有效的操作 OceanBase 数据库，而且可以通过条件控制以及循环控制结构实现复杂的逻辑。PL/cedarSQL 支持的控制结构有 CASE、IF、WHILE 等

1) 条件控制

条件控制语句（又称为分支语句），它通过判断用户给定的条件来选取众多分支中的某一分支来执行。根据不同的情况在程序运行时候动态的做出选择不同的处理方案和动作，IF 语句和 CASE 语句可以很好的满足这种需求。

```
IF expr THEN
   statement_list;
[ELSEIF expr THEN]
   statement_list;
[ELSE]
   statement_list;
END IF;
```

CASE 语句语法定义

```
CASE case_expr
WHEN when_expr THEN
   statement_list;
…
[ELSE]
   statement_list;
END CASE;
```

2) 循环控制

循环控制是为了实现对某些逻辑的重复执行，在 PL/cedarSQL 中提供了 WHILE

循环控制语句。同时，循环体的执行与否是由该结构中的表达式通过计算值来确定的。在程序段中也可以通过 break 关键字跳出当前循环。

```
WHILE expr DO
    [statement_list]
END WHILE;
```

除了上述的语法外，为了支持对存储过程的控制，还需要增加管理存储过程的语法，如创建，删除，执行等语法。

3) 存储过程创建语法

```
CREATE PROCEDURE sp_name([proc_parameter,proc_parameter...])
[routine_body]
```

应用举例：

```
CREATE PROCEDURE demo_sp(IN @in_param int,OUT @out_param int)
BEGIN
END
```

上面的代码创建了一个名为 demo_sp 并且一个参数是 int 类型的输入参数，另一个参数是 int 类型的输出参数。但在声明存储过程的时候需要注意，参数钱必须有 in，out，inout 修饰，并且变量名必须以@开始，参数项可以为空，但是()不能被省略，statement_list 可以为空，表示该存储过程为一个空的存储过程。

4) 调用存储过程的语法

```
CALL sp_name(argument_list)
```

5) 删除存储过程的语法

```
DROP PROCEDURE [IF EXISTS] sp_name
```

2.4.2 多节点服务

对于分布式系统来说，单个节点的性能并不是很突出，为此需要将服务平均的负载到不同的节点上去，存储过程的服务也不是单点提供的，所以在每个进行服务的节点上都有完整的存储过程管理和执行模块。

存储过程最初的出发点之一就是减少网络通信代价，分布式数据库中，数据是分布式的如果在每次调用存储过程的时候都需要进行一次查询(通常是跨节点的)，所带来的网络通信代价是很高的，这就违背了存储过程的初衷。并且分布式

系统中节点的通信对分布式系统环境下的查询性能有着较大影响，这点在高并发访问数据时尤为突出。为了减少存储过程执行时反复的跨节点查询系统表，为此在每个节点上缓存了一份包含所有存储过程源码的副本，目的是减少对系统表的查询次数，以及减少网络通讯带来的延迟，减少对分布式系统原有查询性能的影响，提存储过程执行的高响应速度。

为了确保在分布式系统中各节点上的副本数据的一致性，可以采用同步的方式来保证，同步方式分为两种：一种是强同步、一种是异步同步。根据 CAP 理论[31]可以知道，在追求强一致性的时候必然带来性能上的损失，因为需要根据不同的需求确定同步方式，所以在分布式系统中实现存储过程需要考虑这个问题。

2.5 本章小结

本章介绍了可扩展数据管理系统、存储过程以及编译原理的相关工作。首先，介绍传统集中式数据库所面临的问题，分布式数据库的产生历史原因，介绍新型分布式数据库 NoSQL 和 NewSQL 的相关概念，以及可扩展数据管理系统 OceanBase 的基本架构和原理。然后，介绍了存储过程的基本原理，使用存储过程带来的优势，以及存储过程的一些弊端。其次，介绍了编译器所涉及的编译原理的基本概念，编译基本流程和方法。最后，介绍了本文所需要解决的问题，根据问题描述，提出并设计了名为 PL/cedarSQL 语言，并讨论分布式环境下多节点使用存储过程带来的问题等。

第三章 PL/cedarSQL 运行时环境

本文设计了一种名为 PL/cedarSQL 的过程化 SQL 语言，在第二章中已经介绍了 PL/cedarSQL 的语法规范。在本章中设计并实现了 PL/cedarSQL 的运行环境，也称为 PL/cedarSQL 引擎，包括编译器和解释器执行两部分，PL/cedarSQL 引擎是存储过程的运行环境，是实现存储过程机制的关键。

3.1 引擎架构

PL/cedarSQL 引擎（在本节中简称“引擎”）是负责执行由用户使用 PL/cedarSQL 语言所创建的存储过程的模块。为了降低引擎的实现复杂度，尽可能的从功能和职责的角度出发减少每个模块之间的耦合度，将整个引擎划分为两大模块：一个是编译模块，另一个是执行模块。

过程式 SQL 语言和传统的结构化语言如 C 和 Pascal 语言在编译上有所不同，结构化语言在编译的时候都是将所有的代码统一编译生成目标代码，而 PL/cedarSQL 是结构化语言和 SQL 融合的产物，因此在编译技术上采用结构化语言的编译方法和 SQL 语言的编译方法相结合。在一些支持过程式 SQL 语言的数据库系统中，对于过程式 SQL 语言的编译采用的方式是部分编译，即将过程 SQL 语言中的所有流程控制语句编译为中间代码，将其中 SQL 语句代码原样保留，在执行的时候用数据库的 SQL 引擎执行，这种方式降低了引擎的复杂程度和实现难度，但是在执行性能上受到 SQL 编译速度的影响。引擎在设计的时候为了充分利用原有系统的 SQL 编译器，虽然把过程式语句和 SQL 语句分开处理[32]，但并不是把 SQL 语句原样保留，而是通过已有的 SQL 编译器生成可执行的单元。

如图 3.1 所示，是 PL/cedarSQL 引擎的架构图，其中词法分析器、语法分析器、逻辑计划生成器以及物理计划生成器属于编译模块，解释执行器属于执行模块。在 PostgreSQL 数据库中 PL/pgSQL 引擎只将存储过程编译成语法树，在解释执行时遇到 SQL 语句节点则调用系统提供的内部接口执行，由于 OceanBase 是在执行物理计划的时候才进行变量绑定，因此不需要在编译阶段就确定变量的值，

为了提升引擎的执行速度，在生成中间代码的时候将 SQL 语句生成物理执行计划，在引擎解释执行中间代码的时候无需再次编译 SQL，从而提高引擎的执行速度。

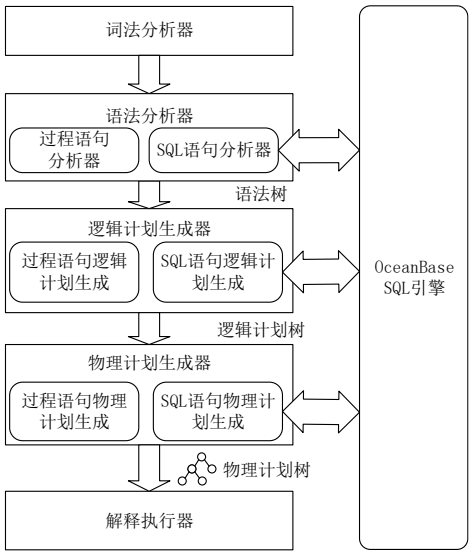


图 3.1 PL/cedarSQL 引擎架构

3.2 编译器实现

程序设计语言人用来向计算机表达计算方法和过程的一种记号。正如我们所知道的，现在的计算机依赖于程序设计语言，因为在所有计算机上运行的所有软件都是用某种程序设计语言编写的。但是任何一个程序在可以运行之前，它首先需要被翻译成为一种能够被计算机识别和执行的形式，完成这项翻译工作的软件系统称为编译器[26]。在数据库系统中 SQL 语言要能够被数据库系统识别完成查询任务，也需要一个编译器来将 SQL 语句转为数据库系统能够识别的形式，而 PL/cedarSQL 语言同样需要一个编译器，用来将它编写的程序翻译成为解释器能够执行的中间代码。

3.2.1 词法分析

编译的第一阶段是词法分析。源程序是由字符组成的，词法分析程序将源程序读入工作区，然后按照一定的规则将字符组成词素[22]，生成并输出一个词法单元[22]序列。这些词法单元序列将会被送到语法分析器进行语法分析，具体流程如图 3.3 所示。

词法分析器和语法分析器的交互是由后者发起的，语法分析器向词法分析器发起调用请求，然后词法分析器不断的从输入中读取字符直到组成一个词素，根据这个词素生成对应的词法单元返回给语法分析器使用。词法分析器在开始工作前会对源程序进行一些处理，以便在后续的扫描中更容易处理，如：删除源程序中的空白字符、制表符以及其它不可见字符；删除源程序中的注释。词法分析器在遇到错误的时候，会生成具体错误的信息并且和源程序关联起来[28]。

随着计算机编译技术的发展，在词法分析方面已经有着很成熟的理论体系，词法分析领域有着大量的开源工具，例如 Java 语言实现的 JavaCC（Java Compiler Compiler）[33]以及 Linux 下的 Lex（LEXical compiler）在编译器构造领域应用非常广泛，Lex 基于正则表达式与有穷自动机理论实现的词法分析器。Flex（fast lexical analyser generator）[34]是 Lex 的另一个替代品，由 Vern Paxson 于 1987 年用 C 语言重写，由 BSD 和 GNU 项目发布。

Flex 的输入文件有 3 个部分组成[34]：声明部分、转换规则、辅助函数。转换规则部分是 Flex 进行词法分析的关键部分，每组转换规则包含一个或多个正则表达式模式以及一段 C 语言编写的代码，当输入的源程序匹配这些正则表达式匹配的时候就执行对应代码。

```
[0-9]+ {
  ParseNode* node = (ParseNode*)malloc(sizeof(ParseNode));
  memset(node, 0 , sizeof(ParseNode));
  yylval->node = node;
  node->type_ = T_INIT;
  node->num_child_ = 0;
  node->str_value_ = yytext;
  node->value_ = atoll(node->str_value_);
  return INTNUM;
}
```

a.数字识别正则文法

```
{@[A-Za-z_][A_Za-z0-9_]*} {
  ParseNode* node = (ParseNode*)malloc(sizeof(ParseNode));
  memset(node, 0 , sizeof(ParseNode));
  node->type_ = T_TEMP_VARIABLE;
  node->num_child_ = 0;
  node->str_value_ = yytext;
  node->value_ = strlen(node->str_value_);
  return TEMP_VARIABLE;
}
```

b.变量正则识别文法

图 3.2 Flex 正则文法示例

词法分析示例见图 3.2，其中 a 中是识别数字正则文法，b 是识别变量名正则文法。遇到匹配该正则的字符时触发动作函数的执行，将构建出一个表示该数字或变量名的结构体并设置相应的类型和值，返回代表数字或变量的词法单元。

3.2.2 语法分析

语法分析是编译器中最重要的一个阶段，它的功能有三个：验证词法单元序列是否符合该语言的语法规则；将符合语法规则的词法单元序列构建一颗表示该

语法的语法树节点；收集该阶段所遇到的错误信息，并能够明确的方式报告给用户。对于符合语法规则的词法单元输入序列，语法分析器能够构造出一颗语法分析树来，并把它交给编译器的其它部分进行进一步的处理。词法分析和语法分析整个流程如图 3.3 所示：

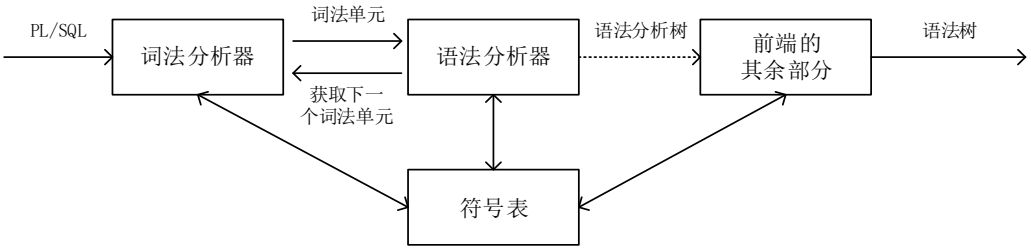


图 3.3 编译器的构成

从功能上看，分析器可以看是一个输入是词法单元序列，输出是语法分析树的模块。对于语法分析可以使用编程程序自动产生工具 Bison[34]的实现，Bison 最初由 Robert Corbett 和 Richard Stallman 编写，GNU 项目帮助发布，它来源于 Yacc[35]，是一种 Yacc 的向前兼容版本，与 Yacc 相比它加入了大量的新特性。Bison 采用 Backus-Naur 范式(BackusNaur Form，BNF）[36]来描述 LALR(1)上下文无关文法(Context-Free Grammar，CFG)[28]。BNF 它通过引入形式化的符号来描述上下文无关文法，该语言有递归特性，能够递归的定义各种复杂的语法结构。Bison 的工作原理就是将 BNF 表示的上下文无关文法转换为 C 语言代码，通过代码实现对该文法的分析。

Bison 程序包含了与 Flex 程序的三个部分：声明部分、规则部分和辅助函数部分。声明部分的代码是会被原样拷贝到目标分析程序开头的 C 代码，通过使用%{和%}来声明，随后使用%token 声明记号，以便于告诉在词法分析程序中记号的名称。第二部分的规则部分包含了通过简单的 BNF 定义的规则，在每一条规则之后同样会有大括号括起来的 C 语言动作代码。第三部分中定义的辅助函数部分 Flex 不进行任何处理，这些辅助函数最后会被直接拷贝到输出文件中。

下面将给出 PL/cedarSQL 中部分语法的 BNF 语法定义：

```
stmt_while  : WHILE expr DO control_sect END WHILE ';'
              {
                malloc_non_terminal_node($$,T_PROCEDURE_WHILE, 2, $2, $4);
              }
            ;
drop_sp     : DROP PROCEDURE IF EXISTS NAME
              {
                malloc_non_terminal_node($$, T_PROCEDURE_DROP, 2, $5,$5);
              }
            | DROP PROCEDURE NAME
              {
                malloc_non_terminal_node($$, T_PROCEDURE_DROP, 1, $3);
              }
            ;
```

图 3.4 BNF 语法定义

图 3.4 中给出了 WHILE 循环结构和 DROP 存储过程的 BNF 语法定义，其中 expr 和 control_sect 结构也是由 BNF 定义的结构，当有词法单元序列满足上述的定义后就会执行花括号里面的代码，malloc_non_terminal_node 生成的节点是非终止符节点，也就是说它是由其他节点组成，而还有一类是终止符，通常是表示数字、字符串以及变量名的节点由 malloc_terminal_node 函数生成。

3.2.3 符号表设计

编译的过程中可能会对源程序曾出现过的数据信息进行查询，符号表就是在编译过程中为编译器提供已经出现过的标识符名字和相关的属性[37]，这些标识符集合就称为符号表，这些属性代表了标识符在源程序中的语义特征，如一个变量的名字、数据类型等。符号表的作用贯穿了整个编译程序的多个阶段，因此符号表的数据结构设计对编译器程序的性能和内存占用起着重要作用。

符号表的组织方式通常有两种：第一种是将所有类型的符号都存储在一张包含所有类型符号属性的大表中，这种实现方式简单，但是内存浪费严重；第二种组织方式是根据类型进行分类，将类型相同的标识符存储在相同的表中，每种类型对应的表都是根据该类型属性定制的，只存储该类型拥有的字段属性，减少不必要字段空间的浪费，虽然这种方式节约了一部分存储空间，但是对于多张符号表的管理成为一个难题。对于第一种方式来说，如果能够降低内存的浪费，则是一个良好的方法。因此在内存分配上可以采用动态的方式，在符号表中只存储表示一个标识符的指针，具体的标识符根据不同的类型进行内存分配，这种方式对于多种类型标识符的存储具有一定优势，减少了不必要的存储开销。虽然不同类

型标识符的属性有差别，但是一般情况下标识符由名字栏和属性栏组成，不同的类型的属性栏中包含的属性数量是不相同的。所以在定义一个能够通用表示不同类型标识符的数据结构时，主要包含两个属性：名字、类型。可以通过类型字段判断标识符的类型，然后在访问该类型特有的属性字段。

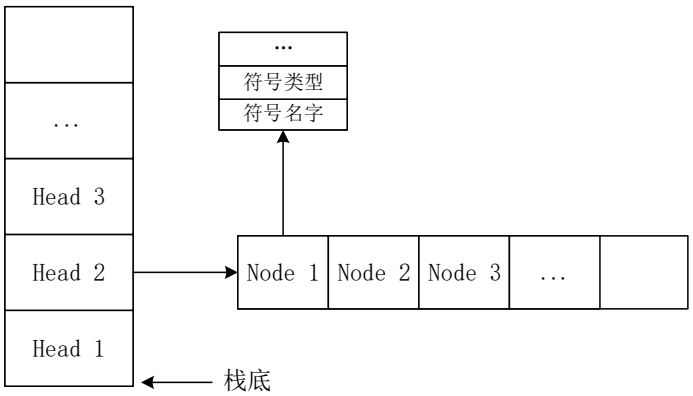


图 3.5 动态符号表

符号表除了上述的作用外，还有一个重要的作用就是在编译阶段对源程序中对标识符的作用域进行控制，同一作用域中一般不允许对同一个标识符进行重复定义。作用域是指一个标识符在程序中的作用范围，为了实现对标识符作用域的控制，一般有两种实现方式：

1) 为每个标识符符号增加一个表示作用域的属性；
2) 为不同的作用域范围建立专属的符号表，负责存储该作用域范围内的标识符信息的管理；

将每个作用域的符号表组织成一个堆栈，利用栈的出栈和入栈特性控制标识符的作用域是支持嵌套定义的语言的最佳选择。符号表还可以提供上下文语义的合法性依据，在一个作用域范围中同一个标识符所代表的含义应该是一致的，因此在遇到一个已经出现过的标识符时需要验证后者的含义是否满足符号表中该标识符已存在的语义属性，如：在一个作用域范围中定义一个普通变量 *a*，后面想通过数组下标 *a[pos]*去访问值，这就需要检查标识符 *a* 属性在上下文中是否一致[38][39]，因为对于后者来说期待 *a* 是一个数组类型，而在符号表中记录的 *a* 是一个非数组类型的变量，所以这就是一种错误的标识符定义。

综合上面的分析后本文最后使用一种基于堆栈和链表的动态符号表的数据

结构，如图 3.5 所示。堆栈的每一层是一个链表的表头指针，链表的每一元素都是一个标识符项，包含了名字、类型以及各自的属性。每一层对应的链表都存储了语句块作用域范围中的所有标识符信息；在编译的过程中每当进入新的一个块结构时会对堆栈进行压栈操作，添加一个新的链表用来存储这层结构中的标识符信息，而在退出块结构的时候会对堆栈进行弹栈的操作，释放栈顶所对应链表的存储空间。当需要查找一个符号的时候，首先遍历当前层所指向的链表，如果没有找到则遍历堆栈的下一层所指向的链表，直到遍历到堆栈的最后一层为止，用这种层次结构实现在标识符的作用域控制。

3.2.3 语法树生成

中间代码的生成依赖于解释器的形式[39]，通常有两种类型的解释器：虚拟机(Virtual Machine)、树形结构解释器(Tree Interpreter)。

虚拟机产生的目的是为了屏蔽不同计算机底层架构和硬件差异，为上层应用提供统一的系统调用接口。虚拟机运行的代码是硬件无关的，通常采用字节码作为中间码，字节码是由抽象的指令组成的二进制文件，虚拟机依次解释执行字节码中的指令。虚拟机中最出名的就是 JVM 了，它是 Java 的运行时环境，虚拟机的出现为跨平台的语言出现提供了条件。虚拟机是一个高度复杂的系统，在实现难度和成本上都是不可控的，对于普通解释性的语言来过于庞大和臃肿。

树形结构解释器顾名思义，就是中间代码的逻辑结构是树状的，在解释执行的时候根据树的逻辑结构进行递归的解释执行。树形结构解释器在流程控制上面依赖于中间代码的逻辑结构，也可以根据不同条件动态的决定执行路径，利用这种特性在执行中可以将 SQL 语句利用数据库系统现有的模块执行，充分利用系统已有模块减少耦合。基于以上的分析和对比，PL/cedarSQL 引擎将采用树形结构的中间代码。

在 PostgreSQL 中每一个语法都有一个对应的结构体[15]表示，与 PostgreSQL 不同的是，在 OceanBase 系统中的语法树数据结构只有一种，该结构体包括一个枚举类型变量 type，和 PostgreSQL 一样，该字段代表该结构体所对应的类型。同时还有两组属性，对应终止符节点的 64 位整型值和字符串值；非终止符节点使

用了后两个字段一个表示节点数量，一个指向子节点数组的首地址。

代码 3.1 OceanBase 语法树结构体

```
typedef struct ParseNode
{
    ObItemType     type;          /* 节点类型 */
    int64_t        value;         /* 终止符节点的真实的值 */
    const char*    str_value;     /* 非终止符节点的孩子节点*/
    int32_t        num_child;     /* 孩子节点的个数*/
    struct _ParseNode** children;
} ParseNode;
```

在 OceanBase 中，一个节点要么是终止符节点要么是非终止符节点，它只会使用两组属性中的一组。int，long，float，double，string 等都是终止符类型，在 OceanBase 中 int，long 都是用 64 位整型 int64_t 表示，而 float，double，string 则用 char *字符串表示，终止符的 num_child 值为 0，children 为 null。

语法树的节点数据结构的设计，主要是为了解决在编译器内部如何表达语法结构的问题，不同的数据库有不同的具体实现。OceanBase 采用终止符和非终止符的分类方法，使 OceanBase 的设计极具灵活性，能够对各种复杂的语句进行表示。在 PL/cedarSQL 的语法中，控制语句的有 3 种，分别是 IF、WHILE、CASE 等。这些结构在通过词法分析以及语法分析后，编译器将会构建出能够表示上述结构的语法树，结构如下所示：

- IF 控制结构

IF 的作用是根据条件进行分支选择，它生成的抽象语法树的结构如图 3.6 所示：

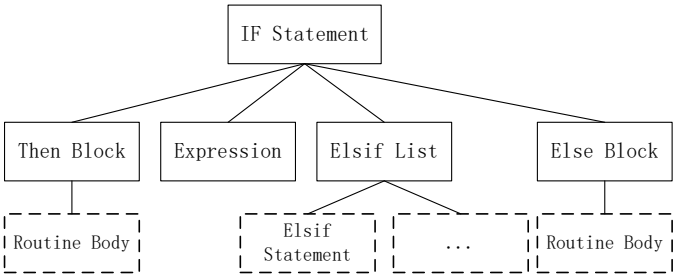


图 3.6 IF 结构语法树

IF 控制结构的语法树总共有四个孩子节点，其中 Expression 节点是表达式节

点，代表 IF 用来判断是否进入分支的条件。Then Block 是一个复合语句块，表示当条件表达式判定为真值的时候需要执行的分支。Elsif List 是由多个 Elsif Statement 节点构成的集合，其中 Elsif Statement 只有两个孩子节点，一个是 Expression 和 Then Block，分别代表了 Elsif 节点的分支条件和语句块。Else Block 代表的是 Else 分支，当表达式判定为假的时候并且所有的 Else if 的表达式也都判定为假的时候才会执行的块。

- WHILE 循环结构

WHILE 结构主要是为了完成各种需要循环的操作，只有当表达式判断为假或者遇到 Break 时才停止执行，它生成的抽象语法树结构如图 3.7 所示，从图中可看出 WHILE 循环结构的语法树由一个表示循环条件的 Expression 表达式和一个表示循环体的 Then Block 组成，Then Block 是一个复合语句块，只有在表达式值为真才会被执行，在循环体中可以使用 Break 语句来退出循环。

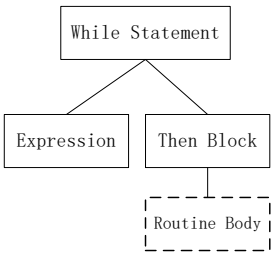


图 3.7 WHILE 结构语法树

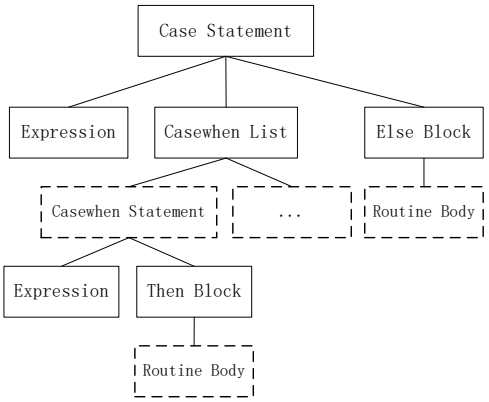


图 3.8 Case 结构语法树

- CASE 选择结构

选择结构实现一个复杂的条件构造，它对应生成的抽象语法树结构如图 3.8 所示。CASE 结构的语法树有三个孩子节点，其中一个是 Expression 表达式节点，表示整个结构的搜索条件。然后是一个 Casewhen List 节点，它由 Casewhen Statement 组成，每个 Casewhen Statement 由一个表达式和一个复合语句节点构成。最后是一个 Else Block 节点，该节点表示当 CASE 结构的搜索表达式的值与所有的 case when 都没有匹配时执行的语句块。

在 PL/cedarSQL 语法中除了流程控制结构，还有变量的声明和赋值结构，以

及数据选择结构，分别是 DECLARE、ASSIGN、SELECT INTO 语句，这几种结构的语法树定义如下。

- 变量声明和赋值

在 PL/cedarSQL 的语法中允许声明指定数据类型的变量，同样也允许对变量执行赋值，所对应的语法就是 DECLARE 和 ASSIGN 语句，它们所对应的语法树结构如图 3.9 和图 3.10 所示：

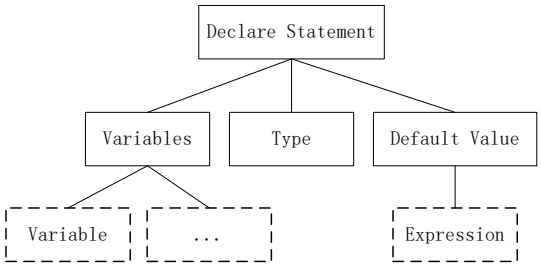


图 3.9 DECLARE 结构语法树

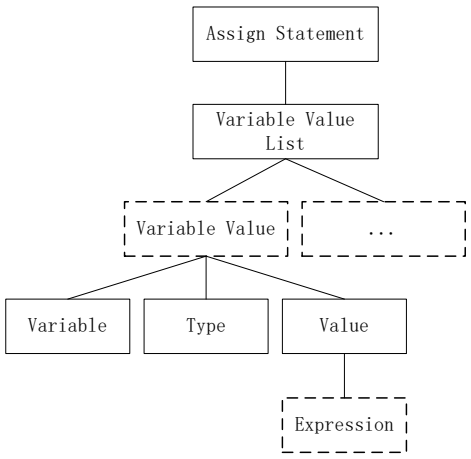


图 3.10 ASSIGN 结构语法树

DECLARE 结构的语法树总共有三个孩子节点，Type 节点代表所声明变量的数据类型（int、varchar、decimal 等）。Variables 是由多个变量组成的列表，Variable 表示的是一个变量，包含了变量名、数据类型、值等。Default Value 是可选项，如果存在的话就是表示在声明变量的时候给变量赋一个默认值。

如图 3.10 所示 ASSIGN 赋值语句的语法树，ASSIGN 语法树由多个 Variable Value 节点组成，每个 Variable Value 节点表示对一个变量的赋值操作，每个值都是由一个 Expression 节点表示，使用 Assign 语句可以同时对多个变量进行赋值操作。

- SELECT INTO 赋值语句

在 PL/cedarSQL 中为了把从数据表中选定的列数据直接存储到变量，引入了 SELECT INTO 语句，该语句本质上是一个 Select 语句构成的，只是在 Select 基础上增加了一个变量列表，SELECT INTO 语法树结构如图 3.11 所示：

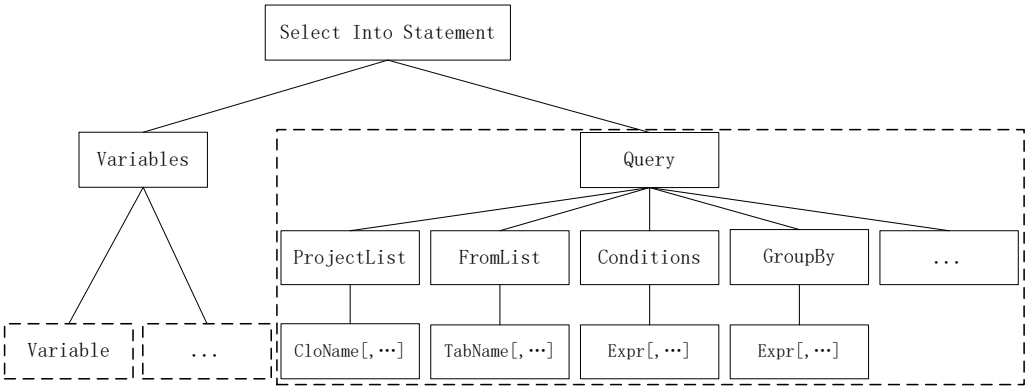


图 3.11 Select Into 语法树

如图 3.11 所展示的语法树中，Select Statement 由一个表示查询的 Query 节点以及一个变量列表组成和 Declare 语句中的变量列表一样的节点。

3.2.4 中间代码生成

在生成语法树过后，解释器直接执行语法树(AST)形式的代码还比较困难，因为在这种树状的数据结构中并不是非常方便使用，因此需要将各种控制结构的语法树转换为另一种更容易理解的数据结构，通常这个过程被称为逻辑计划和物理计划生成。将上面所示的几种语法树封装为以下几种指令：IF_INST、EXPR_INST、CASE_INST、WHILE_INST、SINTO_INST、ASSIGN_INST，DECLARE_INST。

在生成指令序列的过程（逻辑计划和物理计划生成）中不仅仅是简单的把语法树转换为另外一种数据结构表达，如还需要在这一过程中将表达式从中缀形式的语法树转为后缀形式；检查 SQL 语法树中所涉及到的表、字段和表达式等是否有效，并且将 SQL 转为可运算的关系表达式，它包含了一系列的关系演算的基本操作，比如选择、投影、聚集、排序等，本质上，SQL 语句在生成物理计划后是一系列数据操作的有序集合，直接执行并返回数据结果数据。本文中只讨论生成的流程控制结构和后缀表达式的指令数据结构，对 SQL 的逻辑计划和物理计划不做讨论。PL/cedarSQL 中的代码逻辑上是顺序的，反映在语法树中就是孩子节点是有顺序的。

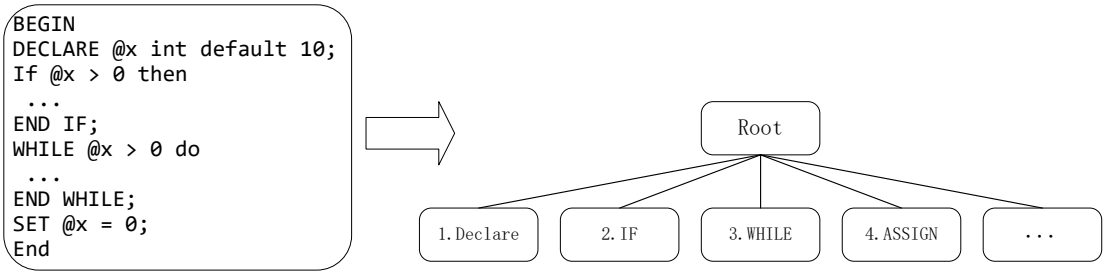


图 3.12 语法树子节点顺序

如图 3.12 所示，在 PL/cedarSQL 语法树根节点的孩子节点从左到右是按照顺序下来的，因此在生成中间代码的时候策略是从根节点的第一个孩子节点开始生成直到最后一个孩子节点。生成的指令按照顺序存储在一个列表中。语法树的结构是递归，因此在生成指令的过程中需要递归的生成。在生成的语法树中很多地方都用到了表达式，而表达式的语法树结构的生成如图 3.13 所示：

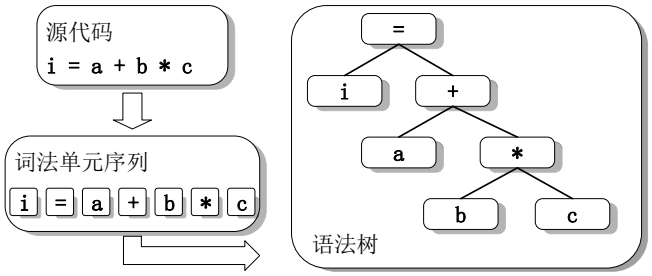


图 3.13 表达式语法树生成

数据库中的表达式和其它语言表达式不太一样，在 SQL 语句中能够支持 LIKE、NOT NULL 等运算符，表达式的结构通常复杂，因为表达式也能够进行嵌套定义。表达式可以是一个变量、常数、一元算符以及二元算符构成，也可以是各种的组合嵌套。在生成逻辑计划时确定表达式语法树中的各个原子表达式的类型和操作符的类型，验证表达式的逻辑正确性，生成物理计划的时候是将中缀表示的表达式转换为后缀表达式，为之后的计算做好准备。后缀表达式的每一项都是 ExprItem 元素，这个数据结构有三个属性分别是：type，data_type，value。其中 type 表示的是何种类型的操作符、函数或元素，data_type 表示如果它不是操作符时所对应元素的数据类型，value 则表示具体值。如图 3.14 所示的表达式生成的后缀表达式。

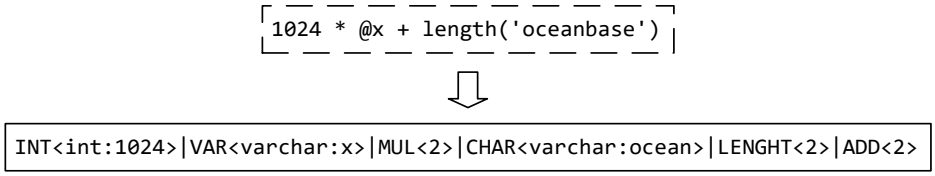


图 3.14 后缀表达式的形式

对于操作符、函数只使用了 value 字段来存储它所需要的操作数数量，而普通类型元素就是使用三个字段来表示，使用后缀方式来表现表达式在计算的时候十分方便。在进行逻辑计划和物理计划生成过后每种结构生成的指令的数据结构如下所示：

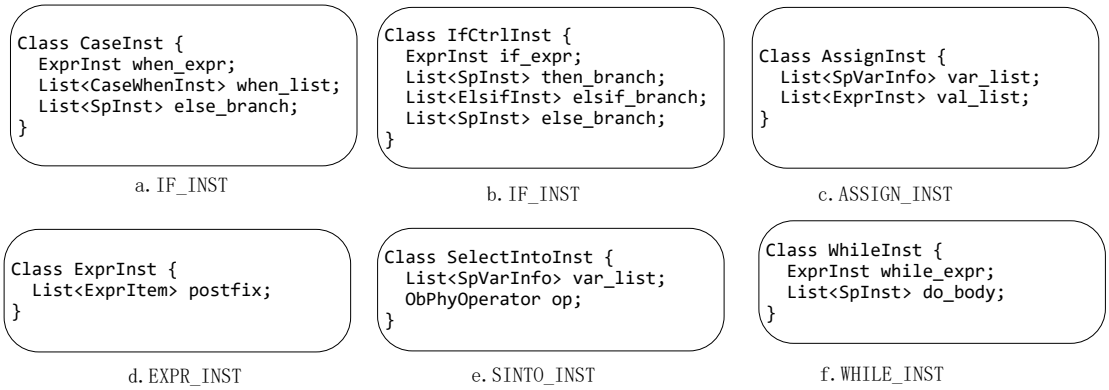


图 3.15 指令数据结构

除图 3.15 所示的指令外，还有一些辅助的指令的数据结构没有列出来，如 CaseWhenInst、ElsifInsty 这两个指令的结构比较简单，都是一个表达式加一个指令列表，DecalreInst 指令是初始化变量的指令，它由一个变量列表，以及每个变量的默认值表达式列表构成。其中 d 是上述的后缀表达式，图 e 中的指令包含一个 ObPhyoperator，这是 SQL 语句生成的物理计划，能够直接 Open 并且通过 get_next_row 获取一行数据。按照语法树根节点的顺序生成，将生成的每一个指令存储在一个列表中，并且递归的生成指令中嵌套的其它指令，最后生成的结构类似下图所示：

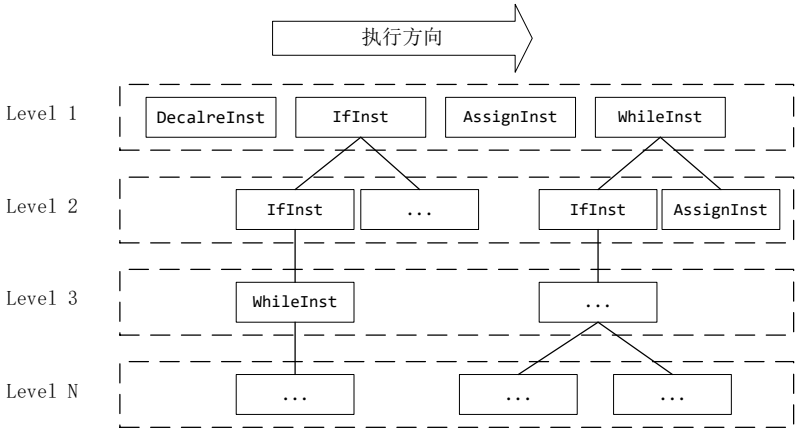


图 3.16 中间代码指令序列

如图 3.16 是语法树在经过逻辑计划和物理计划后生成的指令序列，在 Level 1 层的序列是语法树中根节点下的所有孩子节点，第一层的指令中嵌套的指令出现在 Level 2 层，后面的以此类推，整个指令序列逻辑结构和语法树结构相同。

- 表达式计算

在 PL/cedarSQL 引擎中提供了函数对后缀表达式进行计算，表达式中的操作符除了数学运算符以外还有其它不同类型的函数，并且表达式中的元素除了常量以外还有变量，因此后缀表达式在计算的时候需要解决这两个问题。首先介绍一下基于栈的计算方式，以图 3.17 中的后缀表达式计算为例。

```
输入:INT<int:1024>|VAR<varchar:x>|MUL<2>|CHAR<varchar:ocean>|LENGHT<2>|ADD<2>
```

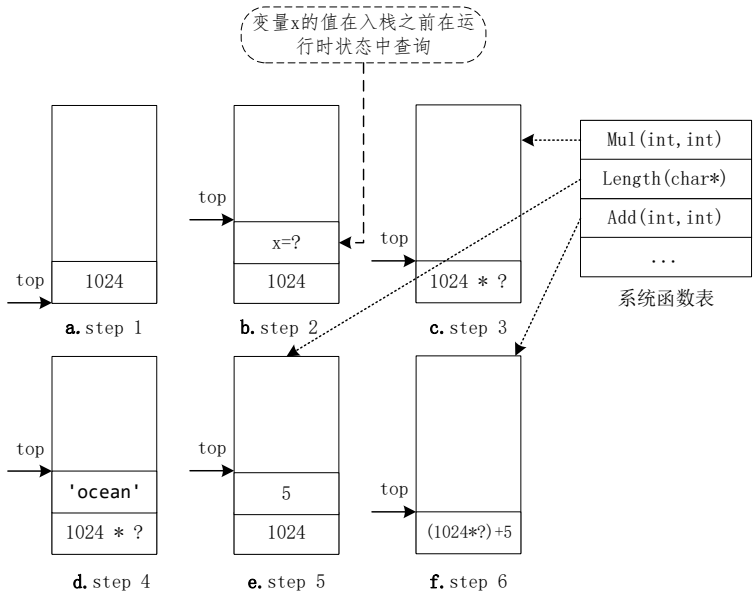


图 3.17 后缀表达式计算

整个计算过程是通过一个堆栈来实现的，具体的步骤如下所示：

1) 1024 是个常数直接入栈；
2) 变量 x 入栈，但是在入栈之前从运行时状态中查询出变量 x 的真实值，用?表示；
3) 遇到 MUL 操作符，它需要 2 个操作数，于是从栈中弹出两个元素调用函数表中的 Mul 函数计算乘积，并将计算出来的结果入栈；
4) 将字符串 ocean 入栈。
5) 遇到 LENGTH 函数，该函数需要 1 个操作数，将 ocean 弹出并且调用函数表中定义的求字符串长度的 Length 函数得到字符长度为 5 并入栈；
6) 遇到 ADD 操作符，它需要 2 个操作数，将从栈中取得两个元素调用 Add 函数计算和值，并将结果入栈；

运算结束后栈中的最后一个元素就是计算结果，需要注意的是在计算的过程中如果使用了变量，但是在运行时状态中没有该变量则会抛出错误。

3.3 解释器实现

PL/cedarSQL 编译器生成的中间代码仅仅是一种逻辑上的结构，并不能直接执行，因此需要一个模块来根据中间代码的含义进行一系列的逻辑操作，这个负责解释执行的模块被称为解释器，它是整个 PL/cedarSQL 引擎的核心。

3.3.1 解释器架构

PL/cedarSQL 的解释执行过程实际上是对生成的中间代码进行深度优先遍历，树型中间代码的逻辑结构对应了结构化的流程控制，在遍历的过程中动态的决定遍历的路径，解释器整体架构如图 3.18 所示：

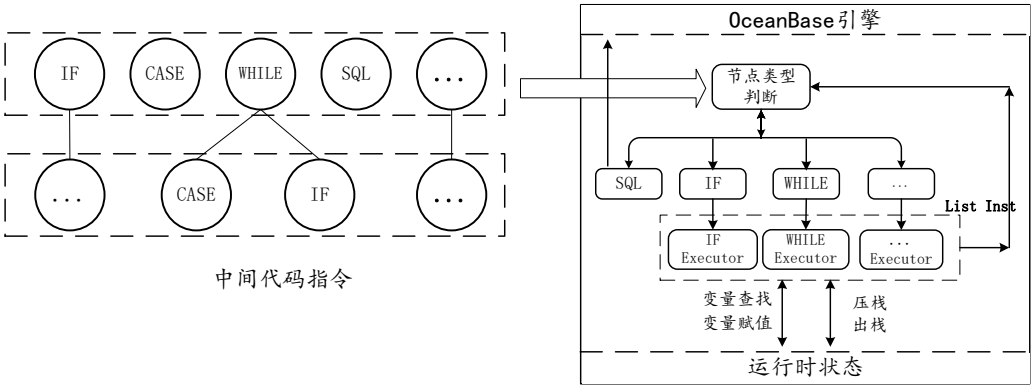


图 3.18 PL/cedarSQL 解释器架构

解释器按照功能可以分为两个模块：一个运行时状态；另外一个是处理函数模块。解释器的输入是中间代码指令序列，解释器为每一种类型指令都有一个对应的处理函数负责执行该指令。在执行过程中需要和运行时状态进行交互，对于 SQL 指令则由 OceanBase 的 SQL 引擎负责执行 SQL 的物理计划，在遍历的过程中遇到嵌套定义的指令就递归的调用相应的处理函数进行执行，直到运行结束。

3.3.2 运行时状态

解释执行的过程中需要查询或者更新变量的值，为此需要为解释器维护一个运行时状态[40]，运行时状态包括一个唯一标识符和一个变量表，唯一标识符用来表示是否执行中有异常，变量表用来存储变量信息。变量表的结构和编译器中使用的符号表结构类似，但是为了在执行过程中能够更加快速的查找或更新变量值，所以没有采用链表结构来存储变量，而是采用了 hash 表来存储变量。

变量可以在不同的语句块中声明，但是在相同作用域内变量不能重名，为了实现对变量的作用域控制，运行时状态的数据结构为一个堆栈，每一个堆栈元素是一个 hash 表，hash 表里通过变量名来获取或设置变量的值。当执行到新一层的指令时就增加一个 hash 表入栈，当前层的指令可以访问从栈顶到栈低的所有变量，当退出该层时将栈顶的 hash 表弹出。

用一个例子来说明在运行过程中运行时状态中的变量是如何变化的，如图 3.19 所示：

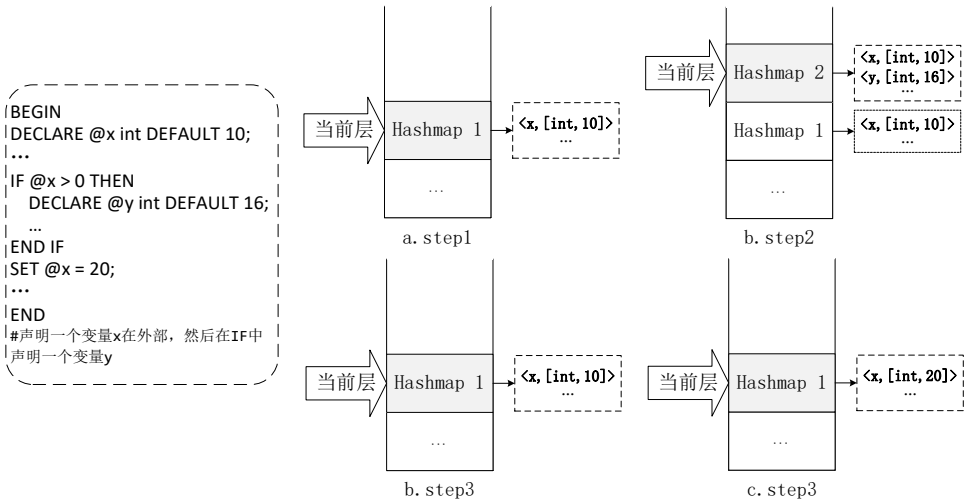


图 3.19 运行时状态

如上图所示，演示了对一段代码的执行过程中运行时状态的变化过程。在执行到第一个声明语句的时候，向当前层的 Hashmap 1 中添加一个表示变量 x 和变量值的键值对，其中变量值包含数据类型和真值两个字段。执行到 IF 语句，表达式判定为真进入新一层执行，新建一个 Hashmap 2 入栈，在遇到第 2 个变量声明语句的时候，向 Hashmap 2 中添加表示变量 y 的键值对。在结束 IF 块的执行后，将栈顶的 Hashmap 2 弹出，遇到一个赋值语句后查找变量 x 的，并将它的值修改为新值。

当声明一个变量时，搜索方向为从当前层开始往栈底搜索，如果在搜索过程中没有找到同名变量，则可以允许声明该变量，否则提示变量已声明。在给一个变量赋值的时候从当前层开始查找变量名，如果当前层没有找到则继续往栈低的方向查找，直到查找到变量或者到栈底，如果查找到了更新对应键所对的的值，否则提示变量不存在。

3.3.3 算法介绍

编译器将源程序生成中间代码指令序列传递到解释器模块后，解释器的主函数遍历指令代码序列，在遍历的过程中根据指令的类型调用对应的指令执行函数解释执行。其中关键的几个算法有，IF 执行函数、DECLARE 执行函数、WHILE 执行函数、CASE 执行函数以及 ASSIGN 执行函数。

- 解释器执行算法

解释器的入口函数称为主函数，主函数遍历中间代码指令序列进行解释执行，解释器的主函数伪代码如下所示：

Algorithm 1 解释器执行算法

```
输入：中间代码指令序列
初始化运行时环境
for all ins in InstructionList do
    if ins is IF then
        调用 IF 执行算法执行该指令;
    else if ins is CASE then
        调用 CASE 执行算法执行该指令;
    else if ins is WHILE then
        调用 WHILE 执行算法执行该指令;
    else if ins is DECLARE then
        调用变量声明算法执行该指令;
    else if ins is ASSIGN then
        调用变量赋值算法执行该指令;
    else if ins is SQL then
        调用数据库引擎执行该指令;
    else if ins is List then
        递归调用自身执行集合指令;
    else
        提示错误信息;
end for
```

如上述代码所示，解释器的输入为中间代码指令序列，在执行前创建一个运行时状态并完成初始化，然后遍历指令序列，依次判断每个指令的类型选择对应的执行算法执行，SQL 指令则直接交给数据库引擎执行，如果不匹配任何类型的话则报告错误。

- IF 执行算法

  IF 选择控制结构指令的执行算法的伪代码如下：

Algorithm 2 IF 指令执行算法

```
输入：IF 指令
  1:  计算表达式的值到临时变量 tmp_flag 里;
  2:  if tmp_flag is true then
  3:      for all ins in then_branch do
  4:          根据指令类型调用对应算法执行指令;
  5:      end for
  6:  else
  7:      for all ins in elsif_branch do
  8:          if 执行指令并且判断是否执行成功 then
  9:              将 need_else 的值设置为 false;
 10:              退出循环;
 11:          end if
 12:      end for
 13:      if exist else branch and need_else is true then
 14:          for all ins in else_branch do
 15:              根据指令类型调用对应算法执行指令;
 16:          end for
 17:      end if
 18:  end if
```

IF 指令执行算法的执行过程如下，首先计算表达式的值，如果为真则遍历 then branch 分支中的指令，每个指令调用对应的处理函数执行。否则遍历执行 elseif branch 中的 Elseif 指令，如果有 Elseif 指令执行成功则结束执行，否则判断是否有 else 分支，如果有的话则遍历 else branch 中的指令，每个指令都调用对应函数执行。流程图如下所示：

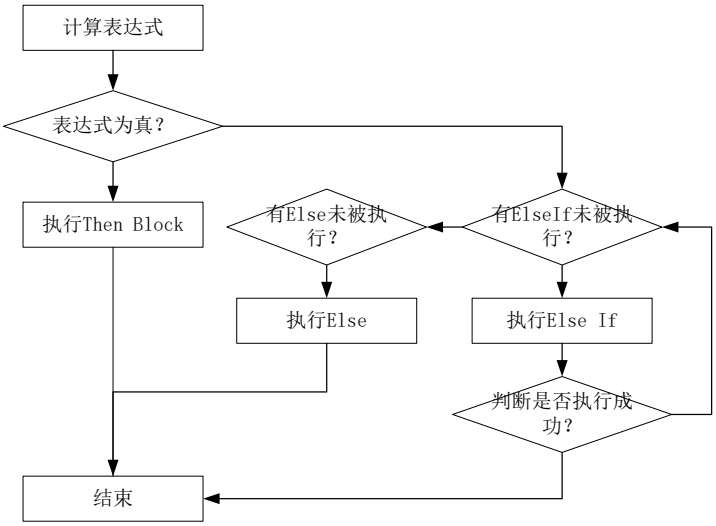


图 3.20 IF 执行流程

- WHILE 执行算法

WHILE 循环结构指令的执行算法伪代码如下所示：

Algorithm 3 WHILE 指令执行算法

输入：WHILE 指令

```
1:  计算表达式的值存储到 flag 中;
2:  while flag is true do
3:      for all ins in do_body do
4:          if ins type is break then
5:              退出循环;
6:          end if
7:          调用对应算法执行指令;
8:          end for
9:          重新计算表达式的值到 flag 中;
10: end while
```

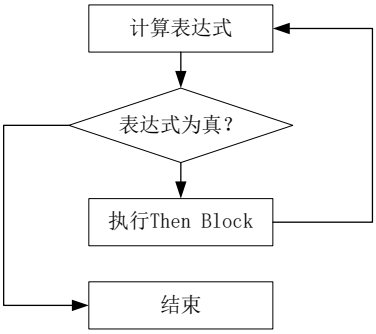


图 3.21 WHILE 执行流程

WHILE 指令执行算法的流程为：首先计算出表达式的值为 α，然后判断 α 的真值，如果为真则遍历执行循环体中的其它指令，然后继续重复这个过程。若 α 为假或者执行到了 break 指令则退出循环。一般来说在循环体内都有一个对表达式值进行修改的操作，否则会造成无法结束循环。

- DECLARE 执行算法

DECLARE 指令的执行算法伪代码如下：

Algorithm 4 变量声明算法

输入：DECLARE 指令

```
1:  if 变量的数据类型合法 then
2:      for all variable in variables do
3:          if 从运行时环境栈顶到栈低搜索到相同名字变量 then
4:              报告重复定义变量的错误;
5:          else
6:              if 默认值不为空 then
7:                  设置变量的初始值等于默认值;
8:              else
9:                  设置变量的初始值等于 null;
10:             end if
11:             将变量以及初始值放入到运行时状态的栈顶;
12:         end if
12:     end for
13: end if
```

对于声明变量的指令，首先在运行时状态中搜索是否有存在相同名字的变量，搜索的方向是从当前层到栈底，如果发现已经声明了相同名字的变量则提示错误。接下来判断变量是否给了初始值，如果没有给初始值，我们将赋一个 null 为初始值。然后将变量名字和初始值放入运行时栈的当前层，也就是最顶层。

- CASE 执行算法

CASE 结构指令的执行算法伪代码如下所示：

Algorithm 5 CASE 指令执行算法

```
输入：CASE 指令
1:  计算表达式的值到变量 c_val 中
2:  设置标志位 need_else 为 true
3:  for all when_ins in when_list do
4:      if c_value = when_ins 表达式的值 then
5:          for all ins in then blocks do
6:          调用对应的算法执行指令
7:          end for
8:          将 need_else 设置为 false
9:      else
10:         do nothing
11:     end if
12: end for
13: if exist else branch and need_else is
14: true then
15:     for all ins in else_branch do
16:         调用对应算法执行指令
17:     end for
    end if
```

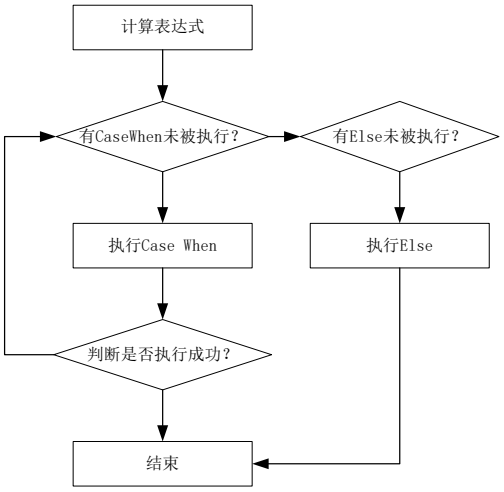


图 3.22 CASE 执行流程

CASE 指令和 IF 指令实现的功能都是进行分支选择，只不过 CASE 的更多的用来进行匹配。CASE 执行的流程如下，首先计算表达式的值 α，遍历 case-when 分支，计算出 case-when 分支中表达式的值 β，然后将 α 和 β 做比较，如果相等则执行 case-when 分支里面的语句，最后判断是否匹配执行了至少一个分支，如果没有则执行 else 分支的语句。

- ASSIGN 执行算法

变量赋值语句的伪代码如下所示：

Algorithm 6 变量赋值算法

输入：ASSIGN 指令

```
1:  for all variable value in variable value list do
2:      if 从运行时环境栈顶到栈低搜索到相同名字变量 then
3:          将运行时状态中该变量的值更改为最新值;
4:      else
5:          报告变量未定义错误;
6:      end if
7:  end for
```

ASSIGN 的操作和 DECLARE 的操作类似，都是对运行时状态进行操作。对于赋值操作来说首先会确定是否声明了该变量，运行时状态的当前层朝向栈底开始搜索，如果没有找到则报错，否则将对应变量的值修改为新值。

- SELECT INTO 执行算法

SELECT INTO 指令实现了将选择的列存储到定义的变量中，该指令的执行算法伪代码如下所示：

Algorithm 7 SELECT INTO 赋值算法

输入：SELECT_INTO 指令

```
1:  将物理计划交给 SQL 引擎执行并得到结果集 result;
2:  if 结果集行数等于一行 then
3:      row = 获取结果集的第一行数据
4:      for all variable value in variable value list do
5:          从行数据中获取列值并更新对应位置的变量的值;
6:      end for
7:  else
8:      报告结果集数据大于一行的错误;
10  end if
```

算法的执行流程如下：首先将 SQL 指令的物理计划打开，它将会发送 scan 或者 get 请求到 ChunkServer 或者 UpdateServer 查询数据然后返回结果集，判断结果集中是否只有一行数据，如果超过一行或者为空需要报告错误，否则循环的将数据取出来赋给对应位置的变量。

3.4 本章小结

本章介绍了 PL/cedarSQL 语言运行环境的实现，介绍了运行环境的架构，将运行环境划分为编译器和解释器两部分。在编译器部分阐述 PL/cedarSQL 的编译技术，对于如何加快编译的速度设计了符号表数据结构，将 SQL 语句和过程语句采用分治法编译充分利用了原有模块，介绍了编译后 PL/cedarSQL 语言中不同控制结构所生成的中间代码结构。解释器部分介绍了解释执行引擎的架构、各种控制结构的执行算法以及运行时状态的数据结构设计，变量作用域的控制方法。

第四章 存储过程管理

PL/cedarSQL 引擎是实现存储过程的核心模块，为实现存储过程机制还需要对存储过程进行必要的管理。本章中将详细讲解存储过程实现原理以及各种操作的流程。首先介绍存储过程管理的架构。其次，介绍存储过程创建的细节和流程以及持久化的方式。然后，介绍存储过程的执行流程和输入输出参数的实现方案和原理。最后，对存储过程一些操作的时间代价进行分析。

4.1 存储过程管理架构

根据存储过程的定义，存储过程实际上就是一段存储在服务器端的一段完成特定功能的程序集，存储过程的执行依赖于宿主语言的编译和运行环境。在 Oceanbase 中我们采用的是 PL/cedarSQL 来作为存储过程的宿主语言，PL/cedarSQL 的语言风格类似 MySQL 数据库的语言标准，用户更习惯这种方式的编程方式，且可移植性很高。在 MergeServer 中通过增加存储过程管理模块来完成：创建、存储、执行、删除等操作，实现对存储过程机制的支持。在增加了 PL/cedarSQL 引擎，以及存储过程管理模块后，MergeServer 的架构如下图所示：

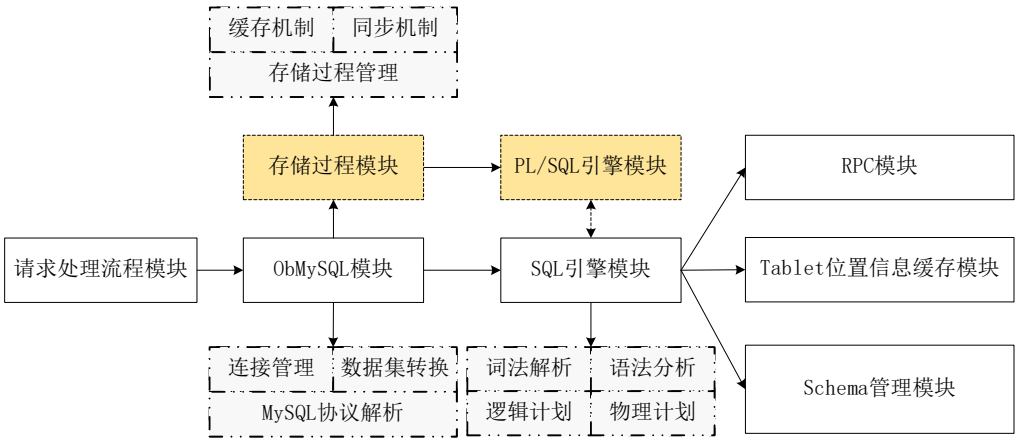


图 4.1 MergeServer 架构图

在收到来自客户端的请求后，请求处理模块把收到的请求交给 ObMySQL 模块，ObMySQL 模块根据 MySQL 协议[16]解析出请求的类型，把普通的 SQL 交给

SQL 引擎处理，把存储过程创建和执行的请求交给存储过程模块进行处理。

存储过程管理模块除了上述的功能外，还有缓存以及同步这两个功能。缓存机制是为了提高存储过程执行速度，让存储过程的执行就像在本地执行一样。而同步机制是为了保证在不同的机器上的缓存一致，这部分的内容将在第五章进行详细的讲解。

设计缓存机制的目的是为了在分布式环境中提高存储过程的性能，按照具体缓存的数据类型可以分为两种：第一种是将系统中所有的存储过程源码都缓存在本地内存，在调用的时候无需从系统表中查询存储过程的源程序。第二种是将存储过程编译后的中间代码缓存在内存中，在执行存储过程的时候遇到已经编译过的存储过程时候可以无需再次编译，直接从缓存中获取中间代码并解释执行。

4.2 存储过程的创建

存储过程的创建实际上就是将存储过程的源码持久化到数据库系统中，但是在持久化之前还有一些需要做一些额外工作，如：验证 PL/cedarSQL 语法是否正确、检查过程是否重复定义等。存储过程管理模块在收到来自客户端创建存储过程的请求后，处理流程如下：

1) 将收到的存储过程创建的源程序解析为两部分，一部分是包含了存储过程名字和参数定义的存储过程声明头，另一部分是由 PL/cedarSQL 代码构成的存储过程体。
2) 根据声明头部分中存储过程的名字在系统表中查询，判断系统中是否已经定义相同名字的存储过程，如果存在则提示存储过程已定义的错误信息。
3) 将由 PL/cedarSQL 代码构成的存储过程体交给 PL/cedarSQL 引擎进行编译，在编译过程中会对 PL/cedarSQL 的语法进行校验，如果出现错误则提示错误信息。
4) 将存储过程源码持久化到系统中，这步包含三部分：将存储过程源码存储到系统表中、将存储过程源码缓存在内存中、将编译生成的中间代码缓存在内存中。

存储过程的创建的流程如下图所示：

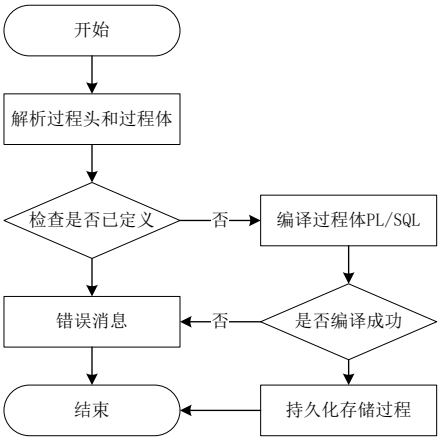


图 4.2 存储过程创建流程

如图 4.2 所示，在存储过程编译成功后，进入到持久化存储过程的步骤。为实现持久化，我们在系统中新增一张名为__all_procedure 的系统表，用来持久化存储过程源码。__all_procedure 表的字段如表 4.1 所示：

表 4.1 __all_procedure 表结构

字段	类型	描述
sp_name	varchar	存储过程名称
arg_number	int	参数个数
args_type	varchar	参数数据类型,使用逗号分隔
args_inout	varchar	参数的输入输出类型
sp_source	text	存储过程源码

在表 4.1 中，将字段 sp_name 设置为主键，这样既可以保证存储过程的唯一性，又可以加快按照存储过程名查询的速度。其中字段 args_number、args_type 和 args_inout 是用来存放存储过程声明头部分的，sp_source 是存放存储过程的创建语句的。

为了在分布式环境下提高存储过程的性能，采用空间换时间的思想，将系统中所有的存储过程源码缓存在 MergeServer 内存中，并将编译后生成的中间代码也根据一定的策略缓存在内存中，执行存储过程的增加缓存命中率加快执行速度。为此在存储过程管理模块中增加两个数据结构，分别是：NameCodeMap 以及 ProcCache。NameCodeMap 是一个存储全局的存储过程名和存储过程源码的 hashmap，它有一个版本号，是用于集群内部同步使用的，将在第五章进行详细

讲解。ProcCache 是用于缓存存储过程名和中间代码的 hashmap，为了减少中间代码占用的内存，ProCache 采用了 FIFO 的方式来进行管理。

4.3 存储过程的删除

在数据库系统中如果有已经不会再使用的存储过程的话，可以通过 Drop 语法将存储过程从系统中移除。存储过程除了持久化在系统表中，还存在于 MS 的缓存中，缓存分为两份：一份是源码的缓存；另一份是中间代码的缓存。所以在删除的时候除了要从系统表中删除对应记录外，还需要将存储过程的源码和中间代码从缓存中删除。具体的流程如图 4.3 所示：

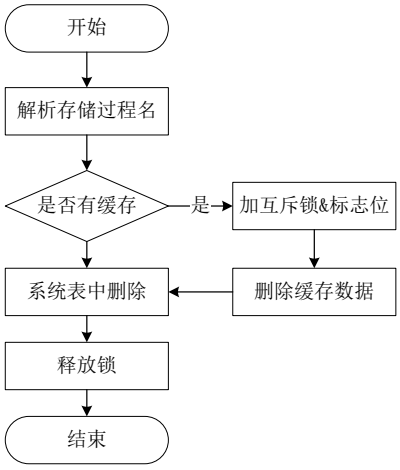


图 4.3 存储过程删除流程

为了防止存储过程在删除过程中被调用而导致系统异常，删除步骤如下：①首先将要删除的存储过程在缓存中的记录加互斥锁和删除标志位；②然后将缓存中的记录移除；③最后将系统表中的记录删除并释放锁。

这些步骤不能交换，否则会出现“幻删”的现象。例如，在删除系统表中存储过程 sp_test 的记录成功后，删除 sp_test 在缓存中的源码和中间代码失败的话，根据 4.4 节中所讲的存储过程的执行流程，此时客户端发起调用存储过程 sp_test 的请求话，会直接从缓存中找到中间代码或源码，能够成功的被执行，用户会认为该存储过程是存在的。但是在系统表中该存储过程的记录已经被删除了，只有 MS 重启丢失缓存数据后，该存储过程才被完全删除，所以这种现象被称为“幻删”。只有先保证成功删除缓存中的数据后再删除系统表中的记录，才可以防止出现上述的现象。

4.4 存储过程执行

存储过程的执行实际上就是存储过程管理模块调用 PL/cedarSQL 引擎解释执行已创建并持久化在数据库系统中的程序。本节介绍了存储过程执行的具体流程步骤，以及如何解决存储过程没有返回值的问题。

4.4.1 执行流程

存储过程调用的请求通过网络达到 MergeServer 后，ObMySQL 模块根据 MySQL 协议解析出调用的存储过程名字以及传递的参数。将过程名和参数传递给存储过程管理模块，在 PL/cedarSQL 解释器真正执行前会进行一系列操作以保证存储过程能够正确执行，所以整个执行流程分为以下几个步骤：1）准备中间代码；2）参数签名检查；3）变量检查；4）复制变量到运行时状态；5）解释执行。整个执行流程如图 4.4 所示：

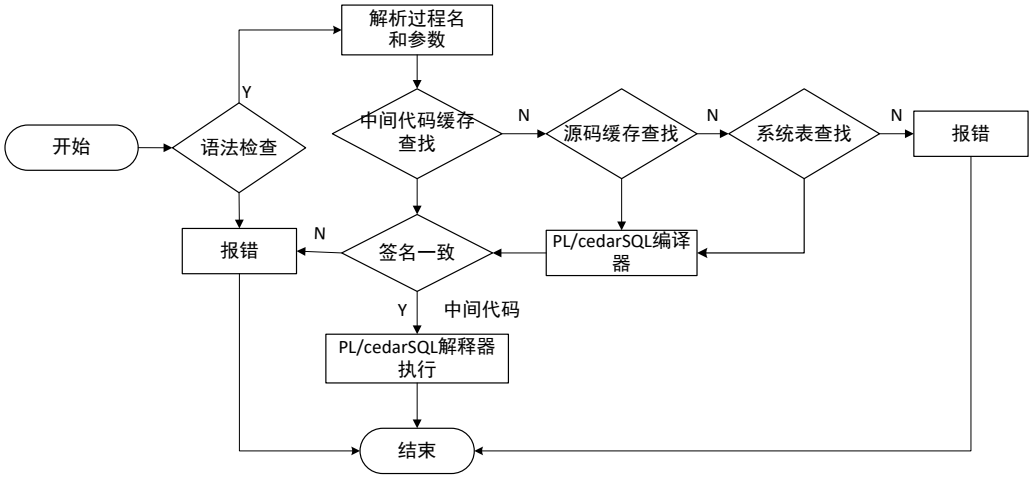


图 4.4 存储过程执行流程图

1) 在执行前需要获取被调用的存储过程中间代码，中间代码获取的策略如下：①从中间代码缓存中查找对应过程名的中间代码；②若上一步没有成功，则在源码缓存中根据存储过程名查找源码，如果有交给 PL/cedarSQL 编译器编译生成中间代码，将中间代码加入缓存；③如果前两步都没有成功，则从系统表中查询该存储过程的源码，将源码交给 PL/cedarSQL 编译器生成中间代码，并将源码加入缓存中，将中间代码加入缓存。

2) 签名一致性检查的作用是为了保证调用的参数个数和类型和存储过程的定

义的形参匹配，具体流程为：首先，检查实参个数和形参个数是否相等；然后，依次检查每个实参的类型和形参的类似是否相容。

3) 存储过程管理模块在完成上述两个准备工作后，将中间代码交给 PL/cedarSQL 解释器执行，在执行中若遇到错误则中断执行并报错。

在从缓存中获取中间代码的时候，需要将中间代码加互斥锁，防止在执行的过程中存储过程被人删除，在执行完成过程将锁释放。在加锁前先检查标志位是否被设置为真，如果标志位已经被设置则返回执行失败，因为此时存储过程正在被删除不可用。在存储过程执行结束后，需要将系统占用的资源进行释放，以及将执行结果反馈到外部去，这是下一小节将要讲述的内容。

4.4.2 存储过程 IN/OUT 参数实现

在定义存储过程的时候可以把形式参数声明为 IN（输入参数）或者 OUT（输出参数）类型，由于存储过程没有返回值，因此可以通过定义 OUT 类型参数传递结果到外部。在声明的时候如果不显示地指定 IN 或者 OUT 则默认的参数类型是 IN，表示传值但是不会再执行结束后把新的值反映到外部，而 OUT 类型参数是表示输出参数，在执行结束过后对 OUT 类型参数的修改会反映到外部去。

在 OceanBase 中用户在外部定义的变量存储在 session 中，而存储过程运行过程中变量时存储在运行时状态栈中，在 session 中定义的变量对于存储过程是不可见的，因此在通过 CALL 调用存储过程的时候如果对应的参数类型是 IN，则在开始执行存储过程的时候将实参的值拷贝到运行时状态栈中的最底层去，执行结束后销毁运行时栈即可。对于 OUT 类型的参数，在存储过程开始执行的时候不需要将 OUT 类型的实参值拷贝到运行时状态栈，只需要拷贝实参的名称和类型，在存储过程运行结束时，将运行时状态栈底层的 OUT 类型参数的值拷贝到用户的 session 空间中去，在外部用户可以通过 select @var 的方式查询到执行结果，这样就可以将存储过程运行的结果反映到外部空间，解决存储过程没有返回值的问题。还有一种参数类型就是 INOUT 类型，这个类型对应的就是 IN 和 OUT 结合，在运行的时候将 INOUT 的参数的值也要拷贝到运行时状态栈底，在结束执行的时候将 INOUT 类型的参数的值拷贝到 session 中去，这样就实现了输入输出

类型的参数。

其他情况，在定义参数为 IN 类型的时候，实参可以是一个变量名也可以是一个表达式（常量值），因为在运行时我们可以通过 session 或者计算表达式得到实参的值，然后将值拷贝到运行时状态栈中，而如果是 OUT 或者 INOUT 类型的话，由于在结束后需要将新值拷贝到 session 空间，需要一个在 session 存在的变量，所以 OUT/INOUT 类型的实参只能是变量名，而不能是表达式，在通过 CALL 调用存储过程的时候，会根据存储过程定义的参数类型和实参的类型进行判断，如果参数类型不满足需求，则提示对应的错误信息。

4.5 时间代价分析

定义 4.5.1. 创建时间代价：假设一个存储过程中除了过程语句外包含 N 条 SQL 语句，节点之间的网络通信时间为 t_{net}，编译 SQL 的时间为 t_{csql}，编译 PL/cedarSQL 其他过程语句的时间为 t_{cpl}，执行一条 Insert 语句的时间为 t_{insert}，同步的时间为 t_{sync}，则存储过程创建的时间代价如下：

$$T_{cost} = 2t_{net} + N * t_{csql} + t_{cpl} + t_{insert} + t_{sync}$$

在创建存储过程的时候，客户端通过网络将创建存储过程的 PL/cedarSQL 代码发送到了 MergeServer 服务器，编译器对存储过程源码进行编译，其中的 SQL 单独调用 SQL 引擎编译，其他的通过 PL/cedarSQL 引擎编译。并且将存储过程源码插入系统表中，以及完成同步操作，最后通知客户端创建成功。按照上面的流程可以得知，创建一个存储过程有 2 次网络通信，然后对其中 SQL 编译的耗时与其中 SQL 数量成正比，最后进行一次数据插入操作。

定义 4.5.2. 执行时间代价：假设一个存储过程中除了过程语句外包含 N 条 SQL 语句，节点之间的网络通信时间为 tnet，编译 SQL 的时间为 tcsql，物理计划执行时间为 tesql，逻辑计算的耗时为 tcompute，则存储过程执行的时间代价如下：

$$T_{sp} = 2t_{net} + N * t_{esql} + t_{compute}$$

若使用 JDBC 完成一个存储过程的任务时间代价为：

$$T_{jdbc} = 2N * t_{net} + N * t_{csql} + N * t_{esql} + t_{compute}$$

注意如果存储过程中没有 SQL 的话，说明这个存储过程直接在本地计算，一条 SQL 的执行需要往返两次，所以有两次网络通信代价

由上面可知使用存储过程和 JDBC 来完成相同的一个任务，两种方式的时间代价差距为：

$$T_{gap} = T_{jdbc} - \ T_{sp} = (2N - 2) * t_{net} + N * t_{csql}$$

可见使用 JDBC 方式要比使用存储过程多了 2N-2 次网络通信的代价，并且还多了 N 次的 SQL 编译时间。由此可见存储过程带来的时间优化是非常明显的。

4.6 本章小结

本章介绍了存储过程管理模块的架构以及各种操作的流程。首先针对存储过程的存储设计了系统表来持久化存储过程的源码，并采用缓存机制将存储过程源码缓存在内存中，提高执行存储过程的速度。其次介绍了存储过程创建的流程，介绍了存储过程模块是如何利用 PL/cedarSQL 引擎解释执行存储过程。然后并通过用户空间和运行时状态数据结构的特点，设计并实现了 IN/OUT 类型的存储过程参数，将执行结果返回到用户空间。最后，定义了存储过程的创建时间代价以及比较存储过程和 JDBC 两种执行方式的时间代价。

第五章 分布式环境下缓存管理

分布式系统为了提高可用性，防止在一部分节点硬盘损坏的时候数据丢失，系统中的数据通常都有多个副本并且分布在不同的节点上，对于数据的更改需要保证不同副本之间数据的一致性。在众多副本中有一个数据副本被称为主副本，数据最先被写到该副本，然后通过复制协议将数据同步到副本。复制协议分为强同步复制以及异步复制[41]，强同步协议需要确保副本数据写入成功才返回成功，而异步复制不需要保证。一致性和可用性是相互矛盾的[31]，强同步虽然保证了副本之间数据的一致性，但是在性能上受到网络、副本可用性等影响。异步方式虽然在性能上有着优势但是不能确保数据的一致性。所以在选择复制协议的时候需要衡量一致性和可用性。

在上一章中通过设计了缓存机制，在执行存储过程的时候无需再次查询系统表获取存储过程源码信息，减少了网络通讯代价，但是带来的问题是集群内各个 MergeServer 上缓存一致性的问题。以三集群[42]（一主二备）下的分布式环境为例，为了提高三集群的利用效率，允许两个备集群上的各 MergerServer 对外提供执行存储过程的服务，在主集群上提供创建、删除和执行存储过程的服务。为了实现所有的 MergerServer 都能够对外提供执行存储过程的服务，必须保证三集群下的各 MergerServer 上对存储过程源码的缓存都是同步的。存储过程的一致性管理，主要由三个部分组成，集群内存储过程源码缓存的一致性管理、集群间存储过程的一致性管理以及宕机、掉线等异常情况下的一致性管理。

集群内部一致性问题需要解决的是各个 MergeServer 在不同情况下如何保持缓存一致的问题。集群间的一致性问题需要解决的是在主机群创建存储过程后，在不同情况下各个备集群中存储过程一致性问题。

5.1 集群内的同步

分布式系统中最关心的两个问题就是分布在不同的节点中的数据是否一致[43]，以及如何保障系统的高可用性。在 OceanBase 集群中为增加可用性，集群

中一般部署两台 RootServer，并且主备之间采用强同步的方式保证数据一致性，使用 Linux HA[44]保障高可用性。主备 RootServer 共享 VIP，当主 RootServer 发生故障时，VIP 能够自动漂移到备 RootServer 所在的机器[45]，备 RootServer 检测到以后切换为主 RootServer 继续提供服务。

在 RootServer 中为了实现将存储过程同步到不同的 MergeServer 中，在 RootServer 维护了一个 NameCodeMap 用来缓存系统中所有的存储过程源码。在主 RootServer 上的 NameCodeMap 是集群中缓存的主副本，为了保证在 RootServe宕机的时候存储过程缓存在主备 RootServer 的强一致性，主备 RootServer 之间的 NameCodeMap 数据使用强同步协议，即所有的操作都需要首先同步到备机，接着修改主机，最后才能返回操作成功。

在集群中除了备 RootServer 以外，所有的 MergeServer 上都有 NameCodeMap 副本，虽然采用强同步协议可以保证数据的强一致性，但是性能会下降。在集群中有多个 MergeServer，如果和 RootServer 一样采用强同步的话，由于强同步协议需要将所有的副本都更新成功后在返回成功，这样如果一个节点失败就会导致更新失败，并且在更新频繁的时候在网络中会同时传输大量的数据，占用大量带宽，降低系统对正常服务的响应时间。因此在 RootServer 和 MergeServer 间采用的异步同步方式，利用这种方式使得 MergeServer 上的数据和 RootServer 的数据保持弱一致性，也就是 MergeServer 上的数据是落后于 RootServer 的，MergeServer 的数据需要一段时间后才能达到最新，所产生的数据不同步时间即不一致的时间窗口[46]。

- 集群内的创建存储过程后的主动同步过程如下图 5.1 所示。

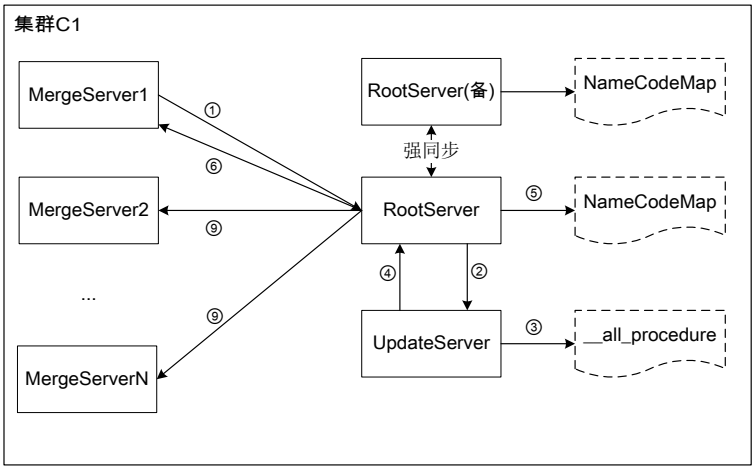


图 5.1 集群内同步流程

图 5.1 中所示的流程步骤如下：

1) 存储过程管理模块在编译完成后将存储过程名字和源码序列化为一个网络包发送到 RootServer。

2) RootServer 在接收到来自 MergeServer 的创建存储过程的网络包后，反序列化出存储过程的名字和源码，检查本地 NameCodeMap 中是否存储相同名字的存储过程，若不存在则构造一个 mutator 发送到 UpdateServer。

3) UpdateSever 接收到来自 RootServer 的 Mutator 后，更新系统表__all_procedure，将新建的存储过程插入到系统表中。

4) RootServer 在接收到 UpdateServer 的更新成功的 response 后，将存储过程名字和源码插入到本地的 NameCodeMap 中，并将存储过程名字和源码(增量)使用强同步协议同步到备 RootServer 的 NameCodeMap 中。

5) RootServe 在和备 RootServer 同步成功后，将存储过程名字和源码序列化为一个网络包通过异步的方式发给所有在 RootServer 上注册的 MergeServer，MergerServer 在接收到网络包之后，将存储过程名字和源码缓存到本地的 NameCodeMap 中。

6) RootServer 在发送完请求给其余 MergeServer 后，返回创建成功包给发送存储过程创建的 MergeServer，MergeServer 在收到成功包后会将存储过程名字和源码缓存到本地的 NameCodeMap 中，并提示用户存储过程创建成功。

由于 RootServer 和其它 MergeServer 之间并没有采用强同步方式，所以不能够保证 MergeServer 上的缓存是最新的。但是这并不会影响该 MergeServer 的正常服务，因为在调用存储过程的流程中，如果在 MergeServer 的缓存中没有查找到该存储过程的源码会发起对系统表的一次查询，由于存储过程源码已经插入到了系统表中，所以能够执行成功。

- 集群内的删除存储过程后的同步过程的流程如下所示。

1) 存储过程管理模块在本地删除存储过程过后，向 RootServer 发送一个删除存储过程的请求，请求中的参数为要删除的存储过程名字。
2) RootServer 在接收到来自 MergeServer 的删除存储过程的请求后，反序列化出存储过程的名字，检查本地 NameCodeMap 中是存在该名字的记录，若存在则构造一个删除请求发送到 UpdateServer。
3) UpdateSever 接收到来自 RootServer 的请求后，更新系统表__all_procedure，将对应记录从系统表中删除。
4) RootServer 在接收到 UpdateServer 的删除成功的响应后，将该存储过程的源码从 NameCodeMap 中删除，并使用强同步协议通知备 RootServer 删除 NameCodeMap 中的对应记录。
5) RootServe 在和备 RootServer 同步成功后，将要删除的存储过程名字序列化后通过异步的方式发给所有在 RootServer 上注册的 MergeServer，MergerServer 在接收到请求之后，从本地的 NameCodeMap 和 ProcCache 中删除该存储过程的相关记录。
6) RootServer 在发送完请求给其余 MergeServer 后，发送消息给原 MergeServer，原 MergeServer 在收到消息包后提示用户存储过程删除成功。

在上述的流程中如果在 RootServer 通知 MergeServer 的时候失败，会导致在 MergerServer 的缓存中仍然有已被删除的存储过程源码，但是由于缓存采用版本号进行控制，通过心跳机制，未收到通知的 MergeServer 在发现自己缓存的版本号和 RootServer 中不一致的时候会主动拉取最新的版本数据，能保证最终删除该存储过程。

5.1.1 延迟生效

存储过程的创建分为多个步骤，因此存储过程并不是立刻就处于可用状态，集群内部同步根据上述的描述可以得到的时间状态如图 5.2 所示：

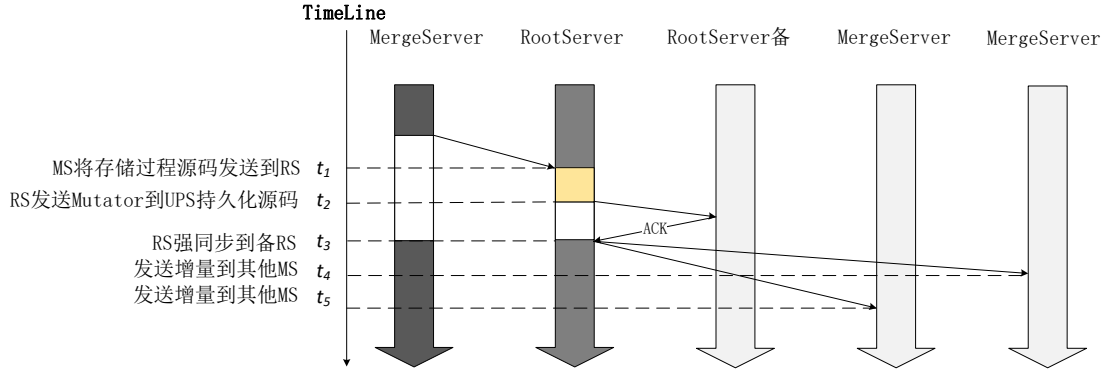


图 5.2 时间状态图

在 MergeServer 在编译了存储过程的源程序过后会经历几个时间状态：t_1 阶段发送源码到 RootServer；t_2 阶段 RootServer 将源码发送到 UpdateServer 插入系统表中；t_3 阶段将增量数据通过强同步方式同步到备 RootServer；t_3 之后开始异步的将增量信息发送到其余的 MergeServer 上。由此可以得知存储过程的生效时间是有延迟的，具体的延迟如图 5.3 所示

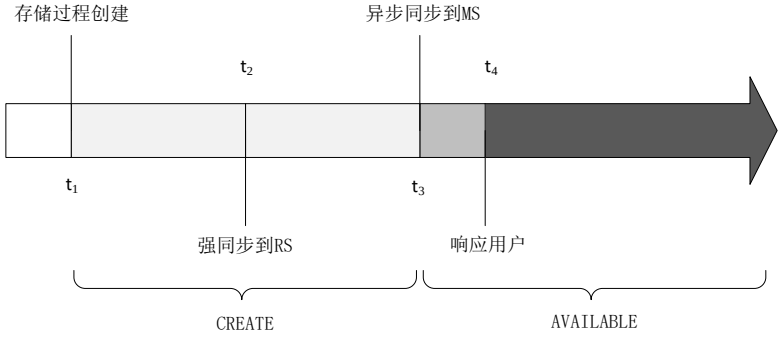


图 5.3 延迟生效图

在 t_3 之后虽然存储过程变得可用，但是在调用存储过程的时候 MergeServer 上面的缓存中还没有更新，此时存储过程管理模块会到系统表__all_procedure 中去查询，此时调用会多一次查询的代价。在 t_4 之后，排除更新异常的 MergeServer，创建的存储过程数据已经通过异步的方式缓存在了 MergeServer 的存储过程缓存模块中。

5.2 集群间的同步

分布式数据库为了增加可用性往往不止部署一个集群，一般情况下会部署两个以上集群甚至异地机房的集群，集群间同步是为了提高可用性，当主机群宕机或者发生异常时能够快速的切换到备集群继续提供服务。

当在集群 C1 中创建存储过程的时候，备集群是无法得知存储过程创建的信息的。集群间的数据一致性是通过同步数据库日志[47]实现的，主集群的操作会产生日志，通过网络同步到备集群，备集群通过对日志进行回放，从而达到数据一致性，集群间日志同步如图 5.4 所示。

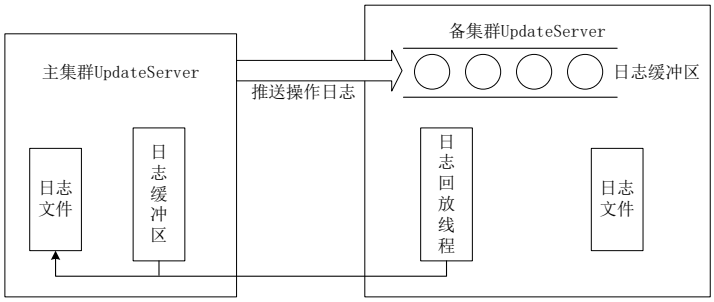


图 5.4 主备集群日志同步

在 OceanBase 的集群之间数据也是通过日志的同步实现最终一致性的，在 OceanBase 中日志同步还有另外一个功能，就是基于日志同步的事件通知机制，工作原理如下：主机群通过在系统表__all_trigger_event 插入一条记录后，备集群在回放日志中遇到对该表的回放操作会回调相应的函数进行回调处理。

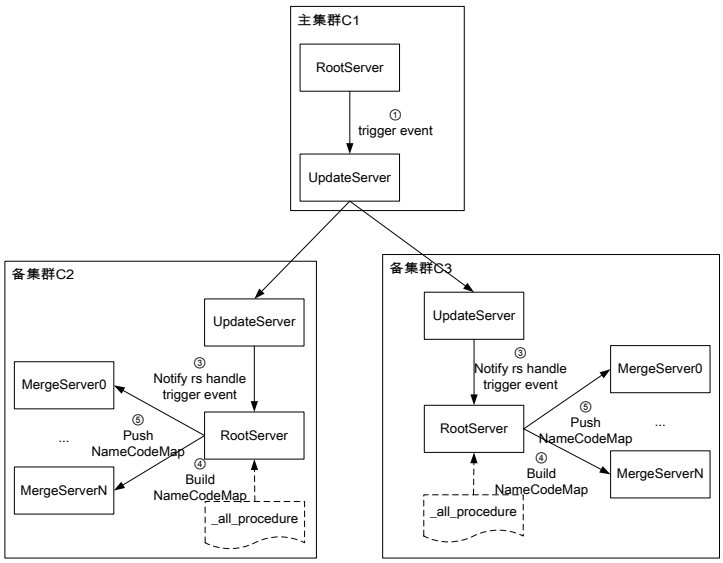


图 5.5 集群间存储过程同步

存储过程创建后集群间同步的流程如下：

1) 主集群 C1 的 RootServer 在发送更新消息给所有的 MergeServer 后，RootServer 产生一个事件通知集群内的 UpdateServer。
2) 主机群通过事件通知机制将同步存储过程的事件通知其它的备集群。
3) 备集群的 UpdateServer 收到来自主机群的事件通知后，通知它所在的集群的 RootServer 处理该事件。
4) RootServer 在收到从 UpdateServer 传来的触发器事件后，解析出类型为同步存储过程的事件，检查自己的角色是否是备 RootServer，于是从所在集群的 __all_procedure 系统表中查询最新的存储过程源码数据，并且更新本地 NameCodeMap 中的数据。
5) 将 NameCodeMap 对象序列化为一个网络包，发送给注册在 RootServer 上的所有 MergerServer，MergerServer 在接收到网络包之后，将其反序列化到本地的 NameCodeMap 中缓存。

在主集群中是先插入记录到__all_procedure 中去，然后在插入记录到__all_trigger_event 中去，在备集群的同步日志的过程中，必然先回放对表__all_procedure 的操作，再回放对__all_trigger_event 的操作，因此在备集群产生了 trigger event 事件的时候，表__all_procedure 已经更新完成，所以在查询备集群的__all_procedure 查询到的数据是最新的。

5.3 异常情况处理

分布式系统的可用性对于用户来说至关重要，用廉价的机器构建的分布式系统中，节点失效和宕机是常事，因此必须要对这些常见的异常情况作出处理。前面两节介绍了正常情况下集群内部和集群间的同步方式。只考虑了正常运行时，主动同步存储过程的缓存，但是没有考虑服务器宕机、掉线、重启等异常情况的处理方法。在本节中将会介绍各种异常情况，以及异常情况的处理方案。

- **情况 1：**MergerServer 掉线、或者尚未在 RootServer 上注册，导致未接收到 RootServer 向 MergerServer 广播的更新请求。

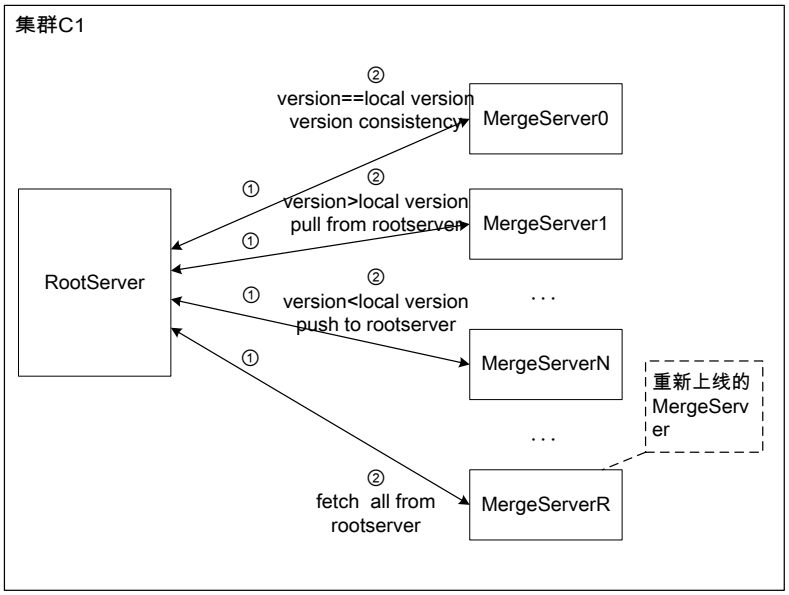


图 5.6 心跳异步更新

虽然在主 RootServer 和备 RootServer 采用了强同步方式可以保证主备同步，但是主 RootServer 和 MergeServer 采用的是异步同步的方式，为了预防 MergeServer 在异步通知过程中更新不成功，除了在创建存储过程时 RootServer 异步推送存储过程的名字和源码到 MergerServer 外，在集群内增加一种心跳异步更新的方式，确保集群内的存储过程的状态是时刻同步的。心跳异步更新的主要原理是：

RootServer 和 MergeServer 之间会一直有一个心跳包检测，目的是为了确保 MergerServer 一直在线。因此，可以利用这个心跳检测，检测 RootServer 上的缓存和 MergeServer 的缓存是否一致。NameCodeMap 有一个 local_version 数据成员，用来记录缓存的版本号[48]，每次对 NameCodeMap 进行更新操作都会使版本号加 1，版本号的初始值为 0。于是在 RootServer 发送给 MergeServer 的心跳包的时候，都增加一个缓存的版本号，MergeServer 在接收到心跳包后，反序列化出其中的版本号与 MergeServer 本地上缓存的版本号做比较。可能出现的三种结果：

1) MergeServer 的版本号与 RootServer 的版本号相等，则说明两者的存储过程缓存已经是一致的，不需要再做任何同步操作。
2) MergeServer 的版本号小于 RootServer 的版本号，则说明 MergeServer 落后了 RootServer，需要重新从 RootServer 上拉取最新的缓存数据。

3) MergeServer 的版本号大于 RootServer 的版本号，则说明 RootServer 宕机重启，将本地的 NameCodeMap 发送给 RootServer。

心跳包中携带缓存的版本号信息后，通过使用版本号检测机制，能够保证在异步方式不成功的情况下保证 MergeServer 中的缓存和 RootServer 上的缓存保持弱一致性，即使 MergeServer 宕机重启，或者掉线。

- **情形 2**：RootServer 重启。

在 RootServer 重启之后，由于 NameCodeMap 是保存在内存中的，所以在重启之后数据会丢失，为了能让 RootServer 重启后提供服务，需要在重启后重新在内存中构造 NameCodeMap。

首先，RootServer 向 MergeServer 会发出 scan 请求，MergeServer 请求 UpdateServer 查询保存在硬盘中的__all_procedure 表，然后返回结果集给 RootServer，RootServer 根据结构重新构建 NameCodeMap。

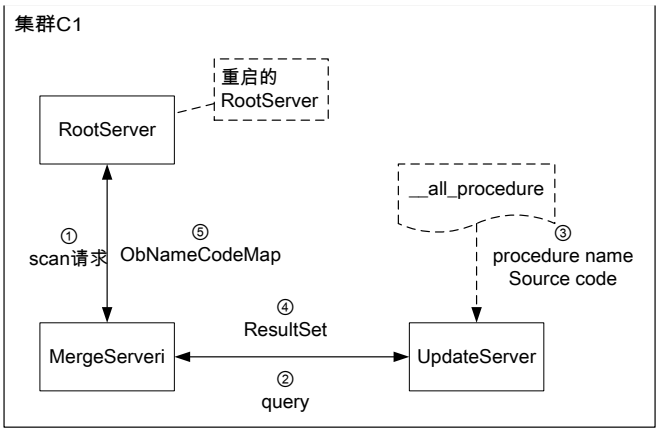


图 5.7 RootServer 重启恢复

- **情形 3**：RootServer 和 MergeServer 都重启。

若 RootServer 和 MergeServer 都宕机，然后重启。这种情形下，RootServer 和 MergeServer 的 NameCodeMap 构建顺序是先 RootServer 后 MergeServer。MergeServer 会启动一个 ObProcedureManager 的线程，等待从 RootServer 上获得 NameCodeMap。而 RootServer 会按照情形 2 所述，构建 NameCodeMap，如图 5.8 所示。

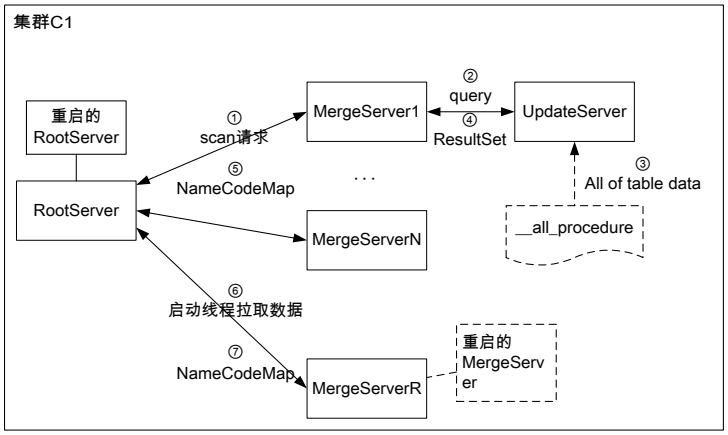


图 5.8 MergeServer 和 RootServer 重启恢复

5.4 本章小结

本章介绍了在分布式环境下通过缓存来优化存储过程执行时的性能，但是带来的问题是不同服务器之间缓存的同步问题，主要分为集群内的同步和集群间的同步方式，以及在异常情况下所采用的方法。对于集群内的同步采用了强同步和异步相结合的方式，对延迟生效的问题进行了分析以及得出了结论。集群间通过日志同步的方式利用事件进行通知更新。

第六章 实验

本文在设计与实现了存储过程模块之后，通过一系列的实验对存储过程的性能进行测试。本章中将使用 OLTPBench 对存储过程的性能进行全面的测试，并对实验结果进行分析和评估。

6.1 实验设置

本文所进行的实验是在由两台服务器上搭建的实验环境中进行的，服务器的硬件配置如表所示。所测试的存储过程是在 OcenBase 开源版本（Version：0.4.2.21）上实现的。

表 6.1 设备配置表

项目	描述
操作系统	CentOS release 6.5
内核	2.6.32431.el6.x86_64
CPU	Intel(R) Xeon(R) CPU E5606@2.13GHz x2
内存	128GB
硬盘	3.6TB
网卡	Broadcom NetXtreme II BCM5709 1000BaseT

OceanBase 是分布式关系数据库，在部署测试系统的时候将 UpdateServer、RootServer 以及 ChunkServer 部署在服务器 A 上，将 MergeServer 部署在服务器 B 上，其中 A、B 两台服务器的硬件都和上表中的配置一样。测试环境部署方案如图 6.1 所示：

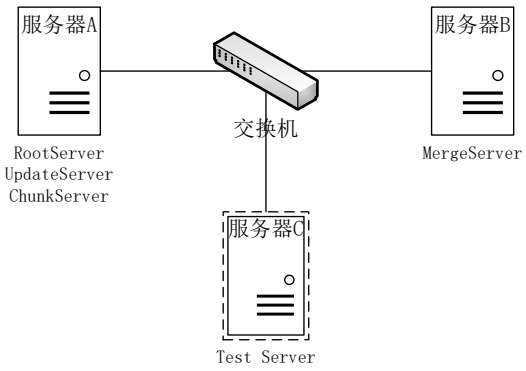


图 6.1 部署架构图

服务器之间采用高速网络连接，服务器 C 负责运行基准测试工具，服务器 B 上的测试程序负责处理测试程序的请求。

6.2 基准测试工具和实验方法

本文为了测试存储过程性能，采用 TPC-C[49]规范进行测试，TPC-C 是专门针对联机交易处理系统(OLTP 系统)的规范，这类系统一般也称为业务处理系统。TPC 委员会只制定了规范但是却没有提供一套测试程序的代码。OLTPBench[58]是一个多线程的负载生成器，它集成了 TPC-C、AuctionMark、CH-benCHmark、SmallBank 等多个的 OLTP 测试框架，可以够用来测试任何支持 JDBC 的关系型数据库性能的 Benchmark 程序，OLTPBench[50]提供了数据收集的功能，例如提供每种类型交易的吞吐量和延迟信息。

TPC-C 测试的所采用的模型是一个大型商品批发供应商，该模型中由多个地理位置相互隔离的仓库(warehouse)组成，每个仓库负责周边多个区域的供货，每个区域负责该区域客户的零售，客户提交订单的中有一定比例的商品是所在区域仓库中没有，需要其他区域的仓库进行供货。为了简化测试过程，本文测试只选用了 TPC-C 标准中给定的 5 种事务处理中的支付事务(Payment)来进行测试，Payment 是模拟了一个交易过程，更新客户余额以反映其订单支付的情况，该事务的过程中会涉及到多张表的查询和更新操作。

本文在测试的过程中设计了两个变量，其中第一个变量是客户端数量，第二个变量是 warehouse 的数量。对比使用不同的变量设置下，使用 JDBC 和存储过程执行 Payment 事务的吞吐量(TPS)和延迟(Latency)性能。其中 OLTPBench 运行在服务器 C 上，然后在服务器 A 和服务器 B 上运行系统监控工具 nmon 采集实验过程中的 CPU、磁盘数据。

其中 JDBC 方式指的是通过应用层编写业务代码完成 Payment 事务对数据库的操作，而存储过程方式是指在存储过程中编写 Payment 事务的业务逻辑，完成对数据库的操作。存储过程的调用也是通过 JDBC 连接实现的。

6.3 实验结果分析

为了测试在不同数量的 warehouse 的情况下，JDBC 和存储过程执行 Payment 事务的 TPS 和 Latency，本文将实验分为两组，两组实验设置不同的 warehouse 数量，分别使用 JDBC 和存储过程（Stored Procedure 简称 SP）执行 Payment 事务。

- 实验设置 1

将 warehouse 的数量设置为 10，客户端数量从 5 开始增加到 80，此时使用 JDBC 和存储过程两种方式执行 Payment 事务的 TPS 和 Latency 如下图所示：

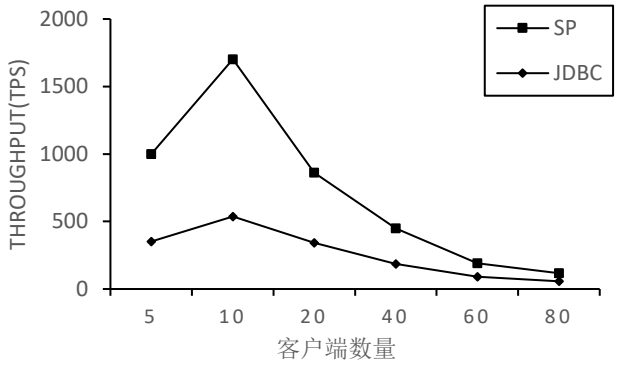


图 6.2 warehouse 为 10 时事务吞吐量

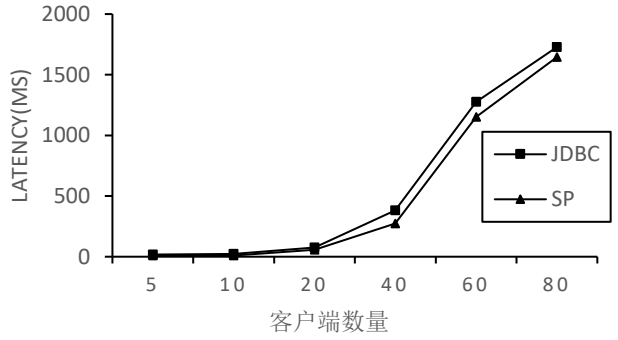
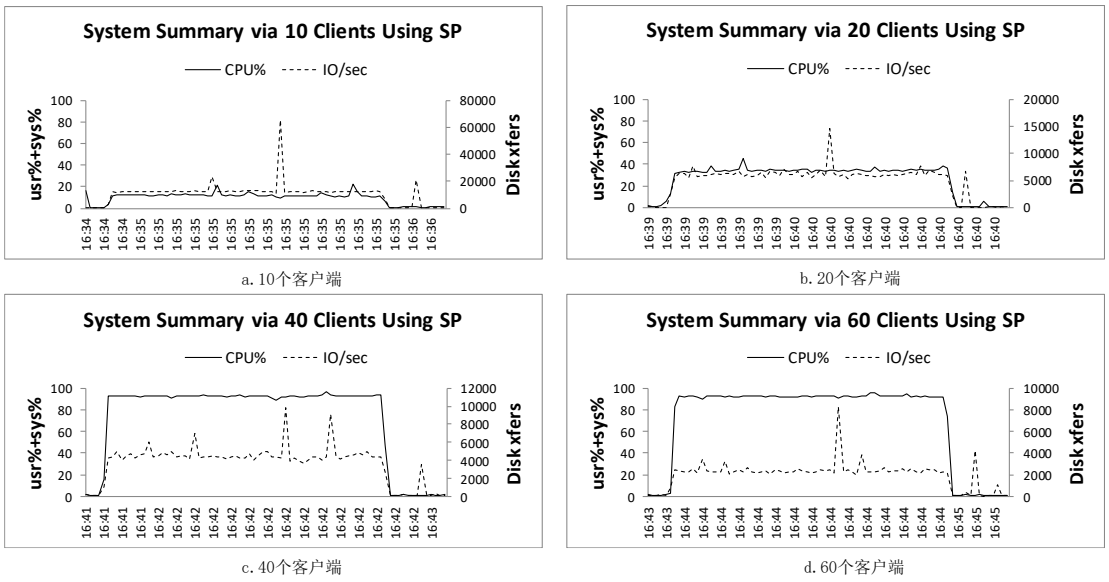


图 6.3 warehouse 为 10 时事务延迟

由图 6.2 可知，在 10 个 warehouse 情况下，随着客户端数量的增加 JDBC 和存储过程的 TPS 都是先增加再降低。在客户端数量为 10 的时候 TPS 达到最高，存储过程的 TPS 大约是 JDBC 的 2 倍左右，在客户端数量为 10 的时候是 JDBC 的 3 倍左右。从图 6.3 中可以得知随着客户端的数量增加，不管是存储过程还是 JDBC，事务的延时都在增加，但是总体来说存储过程的延时低于 JDBC 的。

以存储过程方式为例，当客户数量增加的时候，事务之间的冲突也继续增大，

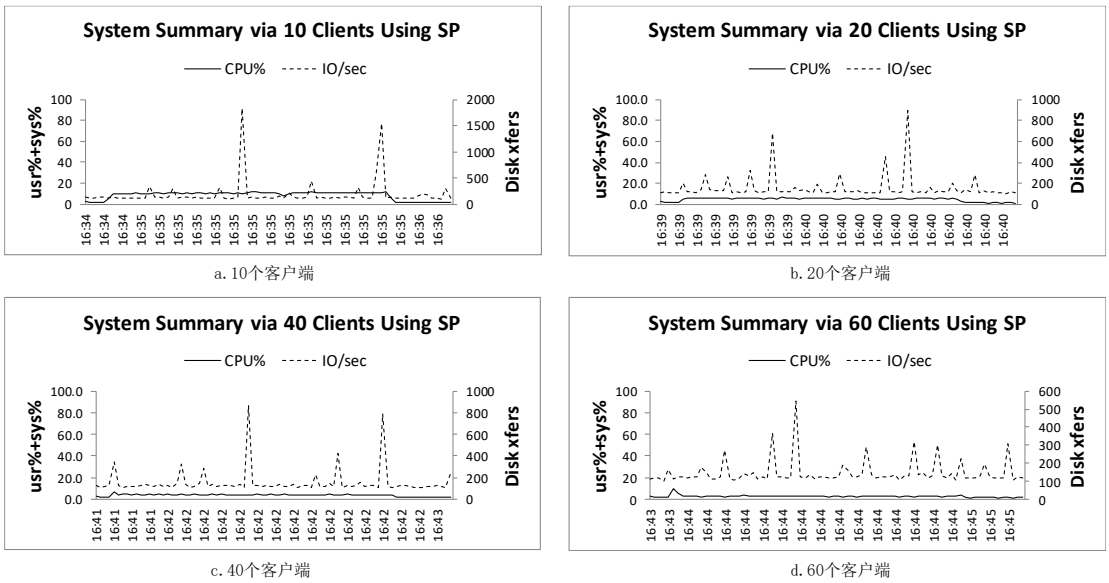
此时的服务器会大量的进行加锁，服务器 A 的系统信息变化如下图所示：



a. 10个客户端　　b. 20个客户端

c. 40个客户端　　d. 60个客户端

图 6.4 执行存储过程在不同客户端数量下服务器 A 资源使用情况



a. 10个客户端　　b. 20个客户端

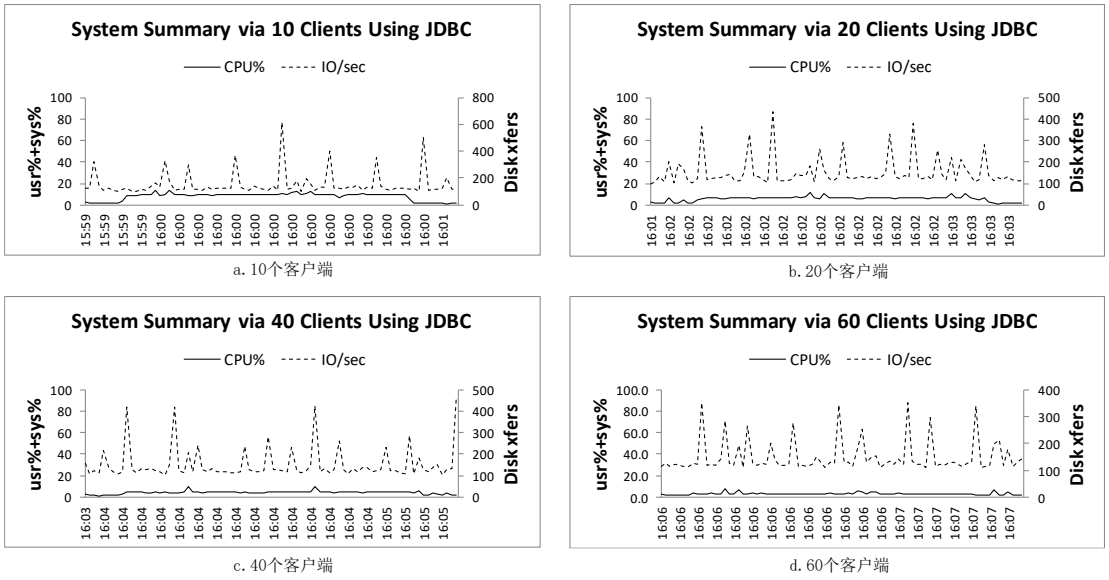
c. 40个客户端　　d. 60个客户端

图 6.5 执行存储过程在不同客户端数量下服务器 B 资源使用情况

由图 6.4 可以看出随着客户端数量的增加，服务器 A 的 CPU 负载不断增大，在客户端 40 的时候 CPU 的负载达到 95%左右，这是由于大量的事务冲突导致的。但是 IO 负载不断减少，这是因为此时需要处理大量的事务的锁竞争导致 CPU 负载增大，提交的事务量降低，导致 IO 降低。图 6.5，对于服务器 B 来说，它接受来自客户端的连接，随着客户端的增加 CPU 负载不断降低，这是因为在客户端数

量较多的时候 TPS 降低，此时服务器 B 的 CPU 负载降低。

使用 JDBC 的时候服务器 A 的变化趋势和图 6.4 类似，也都是在客户端增大的时候 CPU 负载增大，IO 降低，产生这样的原因也都是锁。但是使用 JDBC 的时候服务器 B 的系统信息变化图如下所示：



a. 10个客户端 b. 20个客户端

c. 40个客户端 d. 60个客户端

图 6.6 调用 JDBC 在不同客户端数量下服务器 B 资源使用情况

从图 6.6 中可以看出使用 JDBC 时，服务 B 的 IO 负载和使用存储过程时的 IO 负载不一样，对于使用存储过程来说，大部分时间里 IO 都是很低的，而使用 JDBC 的时候由于需要进行多次的网络通讯，导致服务器 B 的 IO 负载是高于使用存储过程的。

- 实验设置 2

将 warehouse 的数量设置为 40，客户端数量从 5 开始增加到 80，此时使用 JDBC 和存储过程两种方式执行 Payment 事务的 TPS 和 Latency 如图 6.7 所示：

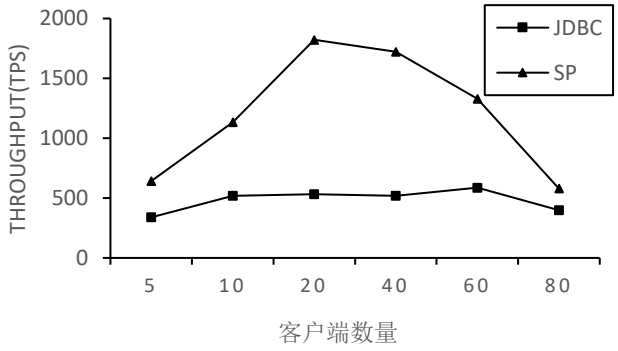


图 6.7 warehouse 为 10 时事务吞吐量

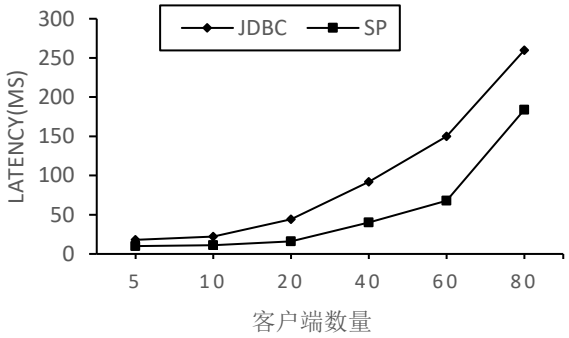


图 6.8 warehouse 为 10 时事务延迟

由图 6.7 可以看出，当设置成 40 个 warehouse 的时候，使用存储过程和 JDBC 的 TPS 都有提升，在客户端数量 20 至 40 左右的时候 TPS 达到顶峰，此时存储过程的 TPS 大约是 JDBC 的 3 倍左右，一般是 JDBC 的 2 倍左右。图 6.8 可以看出在 40 个 warehouse 的时候由于事务冲突减少，所以事务的延时都比较低，可以看出存储过程的延时依然低于 JDBC。

在 40 个 warehouse 的情况下服务器 A 和服务器 B 的性能如图所示：

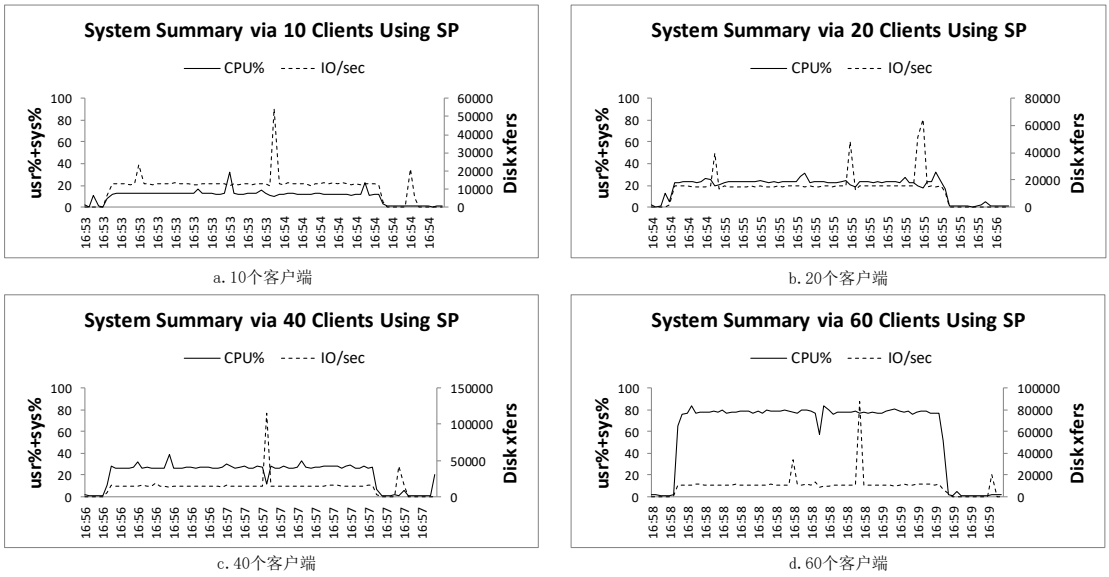
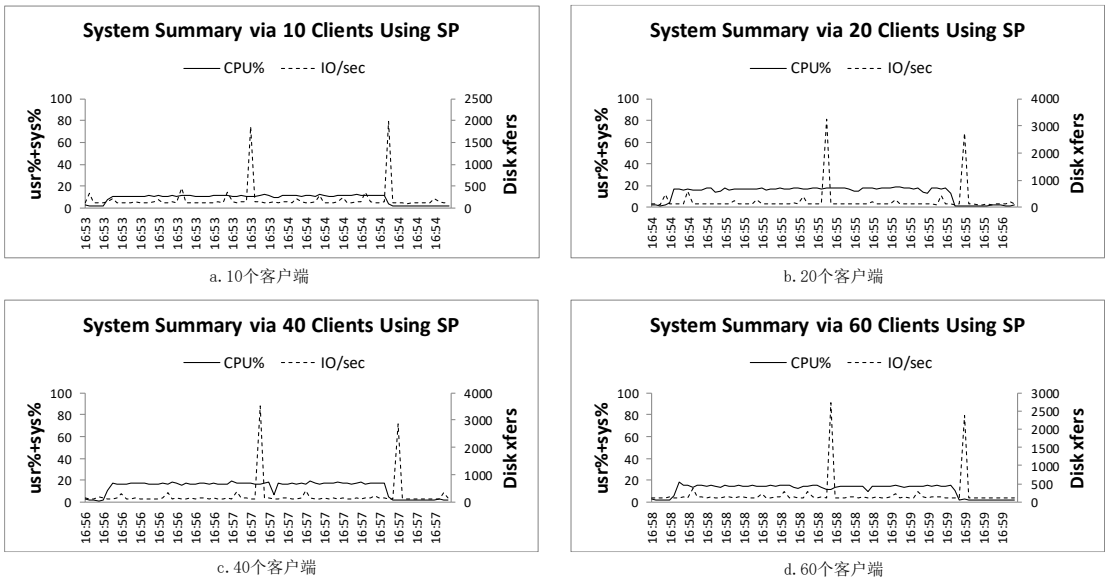


图 6.9 执行存储过程在不同客户端数量下服务器 A 资源使用情况



a. 10个客户端 b. 20个客户端

c. 40个客户端 d. 60个客户端

图 6.10 执行存储过程在不同客户端数量下服务器 B 资源使用情况

由图 6.9 可以看出，服务器 A 的 CPU 负载是在 60 个客户端的时候达到高峰，因为在客户端数量低于 40 的时候事务冲突很少，这时候 CPU 负载比较低，所以这时候 TPS 可以达到高峰，由图 6.10 可以看出在客户端数量 20 到 40 的时候服务器 B 的 CPU 是较高的，当增加到 60 个客户端的时候，有所降低，这是因为事务冲突增加 TPS 降低造成的。

实验中所测量的 Latency 不是单个事务的等待时间，延迟的构成其实可以粗分成以下三个部分：a、Client-Server 之间的网络延迟；b、请求在 Server 上等待处理的时间；c、Server 处理请求需要的时间。（a 是比较固定的一块时间；c 根据方案的不同，处理时间会不同；c 是会随着 Server 负载的增高而增高的。）

所有的客户端都是不间断的发送请求，在不同方案下，能够达到更高吞吐率的设计将会使系统产生更大的负载，负载的增高会导致延迟的增高。

6.4 本章小结

本章中通过使用 OLTPBench 框架在分布式环境下测试了使用存储过程和使用 JDBC 方式完成 TPC-C 中 Payment 事务的性能，通过实验证明存储过程能够有效提高 TPS。根据实验数据可得，在事务冲突较低的情况下存储过程性能最高是 JDBC 方式的 3 倍左右，在事务冲突增加的时候，存储过程方式和 JDBC 方式的性能都急剧降低，分析其原因是由于冲突增多后在 UpdateServer 产生大量的锁操作占用大量 CPU 资源，使得事务处理时间延长从而 TPS 降低。总体来说存储过程对性能的提升是非常明显的。

第七章 总结与展望

7.1 总结

随着互联网的发展，海量数据的存储与处理给数据库领域带来新挑战与机遇，分布式数据库层出不穷，虽然这类数据库解决了海量数据的存储与管理，但是传统数据库中支持的一些特性都还没有这些数据库上实现，比如存储过程、游标、二级索引等。传统企业中在面临海量数据的时候迫不及待的想使用分布式数据库来解决问题，可是由于传统企业中大量的应用使用了数据库所提供的存储过程等技术，而这些传统技术并不是这些新出现的分布式数据库着重解决的问题，所以大部分 NewSQL 数据库都还不支持存储过程。在这样的背景下，许多企业与研究机构都渐渐的开始对这类数据库的功能进行拓展和研究。然而分布式数据与传统的数据库在架构以及其他部分有很多的不同之处，传统数据库中实现这些功能的时候都是没有考虑分布式的情况的，因此这给传统功能在分布式数据库上的实现带来了很大的挑战。本文调研了大量开源数据库中存储过程的实现技术，并着重关注分布式系统中所带来的一些问题，在阿里巴巴集团开发的数据库 OceanBase 在上面设计并实现了存储过程功能，为该数据库的推广和使用奠定了基础。

本文中，首先讲解了存储过程技术出现的背景、存储过程的原理、优势和缺点，并分析传统数据库中存储过程的实现方案的技术点。然后在分布式数据库 OceanBase 上对存储过程实现方案进行研究，根据 OceanBase 数据库架构的特点设计并实现了存储过程的功能，最后在分布式环境下采用缓存的技术进行优化存储过程，探讨了在分布式环境下的对于异常情况的解决办法。存储过程的实现总共分为三部分，第一部分是存储过程的运行时环境；第二部分是存储过程的管理；第三部分是分布式环境下的缓存管理。

本文根据 SQL/PSM 的标准中结合 OceanBase 现有的语法设计了 PL/cedarSQL 语言，该语言是一种过程化 SQL 语言，用来定义存储过程和触发器。为了支持

PL/cedarSQL 语言，设计并实现了该语言的运行时环境，也称为 PL/cedarSQL 引擎。该引擎包含编译器以及解释器这两个模块：编译器将 PL/cedarSQL 的源码进行词法分析、语法分析然后生产语法树形式的中间代码，然后对其中的 SQL 进一步生成逻辑计划以及物理计划；解释器负责在调用存储过程的时候解释执行以及生成好的中间代码。在实现上述 PL/cedarSQL 引擎后为存储过程的实现奠定了基础，在存储过程管理模块中负责对存储过程的创建、存储、执行等进行管理。在分布式环境实现集群内和集群间的同步，考虑各种异常情况下的恢复，保证了存储过程在分布式环境下的性能以及可用性。

最后分布式环境下采用 TPCC 对使用存储过程和使用 JDBC 的方式进行测试，分别在不同的 warehouse 数量以及客户端数量进行测试，分析并比较这两种情况下的 TPS 和 Latency。最后分析得到结论：使用存储过程的确在性能上更有优势。

7.2 未来工作

本文在可扩展数据管理系统 OceanBase 中实现的存储过程只完成了存储过程基本的功能，对于一些复杂的功能支持和性能优化还有所欠缺。在开源版本的 OceanBase 中除了不支持本文所实现的存储过程外还有一些比较重要的功能也未实现，如：触发器、游标等，由于时间和能力有限这些功能还未解决，所以希望在未来完成下面的工作：

1) 存储过程的语法的规范。存储过程的定义语言 PL/cedarSQL 本文参照了 SQL/PSM 的标准，但是还是有一些不一样的地方，因此在未来的工作中对这部分语法的规范化也是很有必要的。规范化后，其它遵循了 SQL/PSM 语法标准的数据库的存储过可以无需修改就可以移植到 OceanBase 系统中。
2) 存储过程的优化。本文对于存储过程的优化仅仅通过增加了缓存机制来进行性能优化，对于有些对性能有很高要求的系统来说是不够的，由于在存储过程的代码中可能会在多个地方同时查询一个表，或者访问相同的表数据，这些数据实际上都是存储在不同的节点，对于这些查询请求可以根据数据的依赖关系，合并一些查询请求也就是所谓的 Group Commit，这样可以减少在 MS 和 CS 以及 UPS 之间的网络通信代价，达到性能的优化。

3) 触发器的实现。触发器是指在数据库状态发生变化的时候执行预定义的指令，触发器的实现需要一个被称为“点火器”的机制，在触发事件的时候执行预先定义的指令，从而实现触发器机制。本文已实现 PL/cedarSQL 引擎能够解释执行指令，所以在以后的工作中实现“点火器”机制来实现触发器功能。

4) 异常处理机制。本文实现的存储过程还未实现异常和错误的处理机制，异常处理机制对于存储过程来说是十分重要的，能够在发生异常的时候捕捉到异常进行处理，防止产生不可预期的后果。

参考文献

[1] 腾讯. 2016 年业绩报告. http://www.tencent.com/zh-cn/content/ir/rp/2016/attachments/201601.pdf. [Online; accessed 20- February-2016]

[2] Wikipedia. Partition(database). https://en.wikipedia.org/wiki/Partition_(database). [Online; accessed 1- August-2016].

[3] Apache. Apache HBase. https://hbase.apche.org. [Online; accessed 1-August-2016].

[4] MongoDB. https://www.mongodb.com/. [Online; accessed 1-July-2016].

[5] Chang F，Dean J，Ghemawat S，et al. Bigtable: a distributed storage system for structured data[C]// Usenix Symposium on Operating Systems Design and Implementation. USENIX Association，2006:15-15.

[6] Stonebraker M. SQL databases v. NoSQL databases[J]. Communications of the ACM，2010，53(4):10-11.

[7] STONEBRAKER M，WEISBERG A. The voltDB main memory DBMS [J]. IEEE Data Eng Bull，2014，36(2):21-27.

[8] Corbett J C，Dean J，Epstein M，et al. Spanner: Google's globally-distributed database[J]. Acm Transactions on Computer Systems，2013，31(3):8.

[9] Alibaba. OceanBase. https://github.com/alibaba/oceanbase/. [Online; accessed 1-September-2016].

[10] Wikipedia. NewSQL. https://en.wikipedia.org/wiki/NewSQL.[Online; accessed 9-June-2016].

[11] Wikipedia. Stored Procedure. http://en.wikipedia.org/wiki/Stored_procedure. [Online; accessed 1-February-2016].

[12] Wikipedia. Oracle database. http://en.wikipedia.org/wiki/Oracle_Database. [On

line; accessed 1-February-2016].

[13]IBM. DB2. https://www.ibm.com/analytics/us/en/technology/db2/. [Online; accessed 12-September-2016].

[14]PostgrepSQL. https://www.postgresql.org/docs/9.5/static/index.html. [Online; accessed 17-April-2016].

[15]彭智勇，彭煜玮. PostgreSQL 数据库内核分析[M]. 机械工业出版社, 2011:187-280.

[16]杨传辉. 大规模分布式存储系统原理解析和架构实战[M]. 北京：机械工业出版社，2013:142-215.

[17]Wikipedia. Cusor. https://en.wikipedia.org/wiki/Cursor_(databases). [Online; accessed 3-August-2016].

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

[19]庞天泽. 可扩展数据管理系统中的高可用实现[D]. 华东师范大学, 2016.

[20]徐俊刚. 分布式数据库系统及其应用[M]. 科学出版社, 2012:4-37.

[21]黄贵，庄明强. OceanBase 分布式存储引擎[J]. 华东师范大学学报(自然科学版)，2014(5):164-172.

[22]AHO A V，SETHI R，ULLMAN J D. Compilers: Principles，techniques，and tools [M]. Boston: Addison-Wesley Publishing Company，1986.

[23]莫利纳 H，厄尔曼 J，怀德姆 J. 数据库系统实现[M].杨冬青，吴愈青，包小源等译. 第 2 版. 北京:机械工业出版社，2010:96-180.

[24]李方超. 基于 NOSQL 的数据最终一致性策略研究[D]. 哈尔滨工程大学, 2012.

[25]朱涛, 周敏奇, 张召. 面向 OceanBase 的存储过程实现技术研究[J]. 华东师范大学学报:自然科学版, 2014(5):281-289.

[26]高仲仪. 编译原理及编译程序构造[M]. 北京航空航天大学出版社，1990:36-142.

[27]陈宇. GKD-PL/SQL 引擎若干关键技术的研究与实现[D]. 国防科学技术大学，2004.

[28]何炎祥，伍春香，王汉飞. 编译原理[M]. 机械工业出版社，2010:15-93.

[29]廖健. 基于虚拟机的存储过程设计与实现[J]. 计算机工程与应用，2004，40(10):172-175.

[30]Wikipedia. SQL:2003. https://en.wikipedia.org/wiki/SQL:2003. [Online; accessed 9-June-2016].

[31]Gilbert S，Lynch N. Brewer's conjecture and the feasibility of consistent，available，partition-tolerant web services[J]. Acm Sigact News，2002，33(2):51-59.

[32]石文渊. PL/SQL 引擎若干关键技术的研究与实现[D]. 国防科学技术大学, 2005.

[33]Sankar S，Viswanadha S，Duncan R. JavaCC: The java compiler compiler [J]. 2008.

[34]Levine J，John L. Flex & Bison[M]. 东南大学出版社, 2010:25-169.

[35]Yacc: Yet Another Compiler-Compiler. http://dinosaur.compilertools.net/yacc/. [Online; accessed 1-June-2016].

[36]Backus–Naur Form. https://en.wikipedia.org/wiki/Backus-Naur_Form. [Online; accessed 1-June-2016].

[37]陈浩. GKD-Base PL/SQL 引擎若干关键技术研究与实现[D]. 国防科学技术大学, 2005.

[38]汪琦. 基于解释器的数据库存储过程研究[D]. 华中科技大学, 2007.

[39]邓晓峰. 基于 Linux 的 PL/SQL 语言编译器[D]. 天津大学, 2005.

[40]高朝瑞. GKD-Base PL/SQL 存储过程和包的研究与实现[D]. 国防科学技术大学研究生院, 国防科学技术大学, 2004.

[41]Gray J, Helland P, O'Neil P, et al. The dangers of replication and a solution[J]. Acm Sigmod Record, 2003, 25(2):173-182.

[42]杨传辉. OceanBase 高可用方案[J]. 华东师范大学学报(自然科学版), 2014 (5):173-179.

[43]David Kroenke and David J Auer. Database concepts. Prentice Hall.2010.

[44]Robertson A. Linux-HA heartbeat system design[C]// Linux Showcase & C onference. 2000:20-20.

[45]周欢，樊秋实，胡华梁. OceanBase 一致性与可用性分析[J]. 华东师范大学学报: 自然科学版，2014 (5):103-116.

[46]张晨东. 可扩展事务处理系统中的日志同步策略[D]. 华东师范大学, 2016.

[47]陈珉，喻丹丹，涂国庆. 分布式数据库系统中数据一致性维护方法研究[J]. 国防科技大学学报, 2002, 24(3):76-80.

[48]Verma S，Mcauliffe M L，Listgarten S，et al. Database management syste m with efficient version control: US，US8010497[P]. 2011.

[49]TPC-C. http://www.tpc.org/tpcc/. [Online; accessed 18-September-2016].

[50]OLTPBenchmark. OLTPBench. https://travis-ci.org/oltpbenchmark/oltpbench. [Online; accessed 18-September-2016].

[51]裴欧亚, 刘文洁, 李战怀,等. 一种面向海量分布式数据库的嵌套查询策略[J]. 华东师范大学学报(自然科学版), 2014(5):271-280.

致谢

时光悄悄的溜走不留声响，眨眼间我即将毕业了，回想当初大学刚毕业的我怀着梦想与激情来到了华东师范大学，而现在的我对学校依依不舍。在这两年半的时间里我收获的不仅仅是愈加丰厚的知识，更重要的是在学习上培养了我灵活的思维方式和广阔的视野。很庆幸这些年来我在数据科学与工程研究院遇到了许多良师益友，无论在学习上、生活上还是工作上都给予了我无私的帮助和热心的照顾，让我在诸多方面都有所成长。

首先，我要感谢周傲英教授领导的数据科学与工程学院。它让我们有机会接触到数据库领域国际一流的研究人员，并且提供给我们国内领先的设备环境进行科学研究。周傲英教授作为导师在科研路上给了我正确的引导，周老师学识渊博每一次的交流都使我受益颇多。

我还要感谢周敏奇老师在科研和生活上的关心与辅导，周老师总是能够抓住问题的核心，让我十分佩服。

我特别感谢钱卫宁教授在论文上给我的悉心指导，让我在一次次的修改中总结并改正自己的不足，教会了我对科研问题思考的方式和认真做好事情的态度。

接下来，感谢在交行工作的期间宫学庆老师给我的帮助，在交行工作期间宫老师是我们的良师益友，特别关心同学的生活和工作，让人感觉到很温暖。

最后，感谢实验室的学弟学妹师兄师姐及我的家人和朋友对我的理解和支持。

感恩之情难以用语言量度，谨以最朴实的话语致以最崇高的敬意！

祝君

二零一六年九月七日

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] 祝君，刘柏众，余晟隽，宫学庆，周敏奇 面向 OceanBase 的存储过程设计与实现[J].华东师范大学学报(自然科学版)，2016.09

[2] 余晟隽，宫学庆，祝君，钱卫宁 基于 Map/Reduce 的分布式数据排序算法分析[J].华东师范大学学报(自然科学版)，2016.09

■ 参与的科研课题

[1] 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，2014-2017，参与人