第3章

嵌入式数据库系统的关键技术

嵌入式数据库涉及的理论和技术包含了计算机和嵌入式系统的最新成果。同时在嵌入式系统或移动设备中，其资源是很有限的。嵌入式数据库一般和应用系统紧密结合在一起，不同的嵌入式系统对嵌入式数据库的应用有不同的需求。比如，实时数据库和移动数据库既有功能上的重合点，也有自身特殊功能的需要。前者十分强调实时性的处理，后者特别重视移动环境下的数据处理。

本章对嵌入式数据库系统的各项关键技术做了较详细的介绍。本章首先介绍嵌入式数据库的总体结构，其次阐述嵌入式数据库的存储管理策略和访问算法，然后介绍实时事务处理技术和并发控制，接下来介绍嵌入式数据库的恢复与备份技术，并简要介绍嵌入式数据库系统的可定制技术，最后对嵌入式数据库的扩展性体现中的典型代表——XML 技术做了阐述。

3.1 概述

嵌入式数据库不仅需要具有传统数据库普遍具备的关键技术，同时还必须完善解决下面不同于传统数据库的各项问题中的关键技术。综合来说，嵌入式数据库的关键技术如下。

1）系统的高可靠性技术

在无人干预的条件下安全可靠地运行是嵌入式数据库系统的一大特性要求，这对系统的可靠性提出了很高的要求。目前嵌入式数据库研究的一个热点就是如何在有限资源的嵌入式系统中确保系统的高可靠性。

2）系统整体的微型化和内存管理

有限的存储空间是各种嵌入式数据库系统在应用时都需要考虑的问题。因此，实现嵌入式系统的关键就是实现嵌入式数据库系统的微型化和合适的内存管理策略。

嵌入式数据库的微型化主要包括两个方面：数据库的微型化和数据库管理系统的微型化。数据库的微型化是指提高数据存储空间利用率，增加数据库的存储能力。其主要实现方法是对数据的压缩存储优化和关系模式的优化。实现微型化的主要途径之一就是依据系统的具体应用需求，选择系统必需的组件功能，为此系统的微型化是以放弃系统功能完备性为代价的。

嵌入式系统中内存是十分有限的，除了管理系统和数据库本身占用存储空间外，数据处理时也会引起内存的开销，即载入数据后，对更新数据、查询数据或对数据库表的修改和删

除等操作都会造成内存资源消耗。因此，除了精简数据库的内部结构外，一种节省内存资源的重要途径就是选择一个合适的内存管理策略。

3）数据同步/复制技术

嵌入式数据库中的数据通常只是某个数据源或后台主数据库的一个局部数据拷贝，用于满足人们在任意地点、任意时刻访问任意数据的需求。也就是说，为了使信息与嵌入式应用程序有机结合，需要从中心数据库下载数据及上传数据。在嵌入式移动数据库中这种信息双向共享主要是通过装载数据同步/复制技术实现的。

4）事务处理及备份恢复

事务是程序并发控制的基本单位。一个程序可划分为多个事务。一个事务由一系列基本操作所组成。这些操作是一个不可分割的整体。嵌入式数据库系统在事务处理方面，应根据整个应用系统中嵌入式设备或移动计算环境下的设备的计算环境特征来进行事务处理控制。例如，嵌入式实时数据库和嵌入式移动数据库对事务的要求就有一定的差异。

事务处理之后，数据库要进行备份。与大型数据库管理系统相比，嵌入式数据库的备份和恢复不能简单以独立的服务或类似形式进行，而要按照某种特定简化方式完成。

5）系统可移植性与多平台支持

嵌入式系统技术应用领域广泛，发展迅猛。同时，从嵌入式操作系统的种类繁多、系统特点多样和更新速度快来看，嵌入式数据库只有满足支持多平台的性能要求才能适应目前广阔的应用空间。嵌入式数据库具有良好的可移植性也是适应嵌入式设备的迅速更新的一个表现。

6）数据安全性

安全性历来都是数据库研究领域的一个有待解决完善的问题。由于嵌入式设备具有自身体积的局限性、自由移动性、便携性等新特性，故其又出现了新的不安全性，设备中的数据大多关联到一些个人隐私性的数据，一旦被盗、丢失或者有不正常用户访问将带来严重后果。随着嵌入式系统对网络多终端的支持，系统数据的安全性也就变得日益重要。

7）系统可定制能力

当数据系统和应用分离时，即使应用所需求的功能较少，但数据系统仍然包括了所有支持的功能模块，这造成了资源的浪费，对于资源有限的嵌入式系统来说是不合理的。系统可定制能力正好可以解决这个问题，它根据实际应用的需求定制数据库系统的功能，真正做到量体裁衣、物尽其用。

另外，在具体的嵌入式数据库中还有特定的关键技术。比如在嵌入式实时数据库中，高实时性就是必须要考虑的。随着嵌入式设备的移动，如果应用请求的处理时间过长，作业处理完成后可能得到无效的逻辑结果或有效性大大降低。因此，处理的及时性和正确性同等重要。又比如在嵌入式移动数据库中，移动数据广播技术和移动 Agent 技术也是关键技术。移动数据广播技术解决了嵌入式移动数据库系统在移动用户数目增多和上、下行链路之间的差异性问题，效果明显。而移动 Agent 技术所处的背景是移动计算环境下传统的 C/S 计算模型已满足不了移动计算的日益需求，移动 Agent 作为一种灵活的技术创新，一种新的软件范型被广泛地应用到移动计算环境中。

嵌入式数据库的技术评价指标见表 3-1。

表 3-1 嵌入式数据库技术评测指标

特　征	子 特 征	度 量 元
实现技术测试	数据模型	支持数据类型
	存储方式	支持存储方式
	索引结构	支持索引结构
	数据库模式	关系型数据库
	SQL 支持	是否支持 SQL
嵌入式特性	精简性	空间占用大小(KB)
	可管理性	是否实现零管理
	数据容量	最大处理数据量大小
	移植性	支持平台数量或列举主要应用平台
	扩展性	是否支持 XML、API、Java 扩展等
基准性能	可预测性	数据操作时间和存储空间占用情况
	数据处理能力	增、删、改、查基本操作的实时性能
	并发能力	TPC-B Benchmark
	可靠性	错误处理能力
应用综合特性	系统配置	系统配置能力
	支持语言	支持语言或接口数量
	适用平台	支持平台数量
	易用性	难易程度
	使用场合	适用微型、小型、中型或大型系统
	成本	是否开源、免费

3.2 嵌入式数据库存储设备管理策略简介

嵌入式系统针对不同的应用场景会采用相应的存储设备和存储方式进行数据存储。比如针对高速数据处理采用内存进行存储,针对持久化存储采用闪存进行存储,针对大数据量低速存取采用磁盘进行存储,以及将存储设备组成 RAID 阵列进行存储等。这就要求嵌入式数据库能够适应不同的存储设备和方式,隐藏存储细节,并提供统一的数据存储和管理服务,使用户在开发数据库应用程序时能够聚焦于业务逻辑。


视频讲解

3.2.1 嵌入式系统的存储方式

嵌入式系统根据其设计用途和使用场景会采用以下几种存储方式。

(1) 全内存存储:用于对实时性、处理速度有着较高要求的临时数据处理的情况,如交易处理、火控解算等。由于内存相对其他存储器更加昂贵以及掉电后其保存的数据会丢失,这类应用通常具有对处理速度和实时性要求较高、数据量小、临时数据和中间结果较多等特点。

(2) 闪存(Flash)存储:对于嵌入式系统而言,闪存是最广泛使用的存储设备。相对于磁盘而言,首先,闪存体积小、重量轻,非常适合结构紧凑的嵌入式系统。其次,闪存是纯电子设备,没有机械组件,因此具有良好的抗震动性且运行噪声低,可以用于运行环境恶劣的嵌入式系统,如工业控制、航空航天等。最后,闪存的读写速度大大高于磁盘,其容量也在不

断增大，以闪存为基础的固态硬盘的容量已经达到1TB以上。虽然闪存的单位存储价格仍然较为昂贵，但嵌入式系统处理的数据量通常较小，闪存器件的价格占比相对于整个系统而言还是较低的。

由于具有高可靠性、高存储密度、低价格、非易失、擦写方便等优点，Flash存储器取代了传统的EPROM和EEPROM，在嵌入式系统中得到了广泛的应用。Flash存储器可以分为若干块，每块又由若干页组成，对Flash的擦除操作以块为单位进行，而读和写操作以页为单位进行。Flash存储器在进行写入操作之前必须先擦除目标块。

根据所采用的制造技术不同，Flash存储器主要分为Nor Flash和Nand Flash两种。Nor Flash通常容量较小，其主要特点是程序代码可以直接在Flash内运行。Nor Flash具有RAM接口，易于访问，缺点是擦除电路复杂，写速度和擦除速度都比较慢，最大擦写次数约10万次，典型大小为128KB。Nand Flash通常容量较大，具有很高的存储密度，从而降低了单位价格。Nand Flash的块尺寸较小，典型大小为8KB，擦除速度快，使用寿命也更长，最大擦写次数可以达到100万次，但是其访问接口是复杂的I/O口，并且坏块和位反转现象较多，对驱动程序的要求较高。由于Nor Flash和Nand Flash各具特色，因此它们的用途也各不相同，Nor Flash一般用来存储体积较小的代码，而Nand Flash则用来存放大体积的数据。

在嵌入式系统中，Flash上也可以运行传统的文件系统，如ext2等，但是这类文件系统没有考虑Flash存储器的物理特性和使用特点，如Flash存储器中各个块的最大擦除次数是有限的。

为了延长Flash存储器的整体寿命需要均匀地使用各个块，这就需要磨损均衡的功能；为了提高Flash存储器的利用率，还应该有对存储空间的碎片收集功能；在嵌入式系统中，要考虑出现系统意外掉电的情况，所以文件系统还应该有掉电保护的功能，保证系统在出现意外掉电时也不会丢失数据。因此在Flash存储设备上，目前主要采用了专门针对Flash存储器的要求而设计的JFFS2(Journaling Flash File System Version 2)文件系统。

(3) 磁盘存储：随着磁盘技术的进步，相对于之前的磁盘其体积已经大大缩小，价格也进一步降低，因此对于某些需要进行大数据量存储且对访问速度要求较低或者对价格较为敏感的嵌入式系统而言，有使用磁盘进行数据存储的需求，而且通常也不会将这类设备运用到强震动、强辐射、温度剧烈变化的恶劣环境中。

(4) 存储器阵列存储：有些嵌入式系统对于数据存储的并行性或者可靠性有着很高的要求，因此还可能存在使用闪存或者磁盘搭建存储器阵列进行存储的情况。现实情况中，考虑到嵌入式设备的体积，存储器阵列通常使用RAID0或者RAID1的方式搭建。

RAID是英文Redundant Array of Independent Disks的缩写，中文简称为独立磁盘冗余阵列。RAID就是一种由多块硬盘构成的冗余阵列。虽然RAID包含多块硬盘，但是在操作系统下它是作为一个独立的大型存储设备出现。利用RAID技术于存储系统的好处主要有以下三种：通过把多个磁盘组织在一起作为一个逻辑卷提供磁盘跨越功能；通过把数据分成多个数据块(Block)并行写入/读出多个磁盘提高访问磁盘的速度；通过镜像或校验操作提供容错能力。

RAID0又称为Stripe或Striping，它代表了所有RAID级别中最高的存储性能。RAID0提高存储性能的原理是把连续的数据分散到多个磁盘上存取，这样，当系统有数据

请求时就可以被多个磁盘并行执行,每个磁盘执行属于它自己的那部分数据请求。这种数据上的并行操作可以充分利用总线的带宽,显著提高磁盘整体存取性能。

RAID1 是磁盘阵列中单位成本最高的,但提供了很高的数据安全性和可用性。当一个磁盘失效时,系统可以自动切换到镜像磁盘上读写,而不需要重组失效的数据。虽然 RAID1 的性能没有 RAID0 磁盘阵列那样好,但其数据读取速度较单一硬盘来得快,因为数据会从两块硬盘中较快的一块中读出。RAID1 磁盘阵列的写入速度通常较慢,因为数据要分别写入两块硬盘中并做比较。RAID1 磁盘阵列一般支持"热交换",也就是说阵列中硬盘的移除或替换可以在系统运行时进行,无须中断退出系统。

(5) 混合存储:在某些特殊应用场景下,嵌入式系统的应用程序还可能存在同时使用多种存储方式的情况。例如,同时在内存中以及闪存中创建不同的数据库,其中内存数据库保存缓存数据,闪存数据库保存备份数据。

3.2.2 嵌入式数据库存储设备管理策略

基于上述对嵌入式系统的存储方式的分析,为了实现嵌入式数据库对应用程序隐藏存储细节并提供统一的数据管理服务,嵌入式数据库通常采用以下策略对嵌入式设备的存储设备进行管理。

1. 对表或者数据类进行分类

对嵌入式数据库而言,嵌入式系统使用的各类存储设备均可以归结为非持久性存储和持久性存储两类。比如,内存存储属于非持久性存储,其他存储方式属于持久性存储。因此,在使用数据定义语言进行数据库模式设计时,应指定表或者数据类的存储方式,例如:

```
create transient table table1 {[fields]};
create persistent table table2 {[fields]};
```

其中,table1 即表 1 为非持久性表,在数据定义语言中用 transient 表示。table2 即表 2 为持久性表,在数据定义语言中用 persistent 表示。fields 表示表的各个数据项。嵌入式数据库在进行数据处理时,会检查数据所属的表或者数据类的存储属性,从而采用不同的方式进行存取。比如对表 1 中的数据,数据库会调用非持久性存储管理模块进行处理;对于表 2 中的数据,数据库会调用持久性存储管理模块进行处理。这样,嵌入式数据库就能方便地支持包括混合存储方式在内的各种存储方式。

2. 划分逻辑设备

嵌入式数据库根据存储设备的物理特性和访问方式可将其抽象为几种逻辑设备,并且在创建数据库时指定该数据库使用的逻辑设备及其用途,逻辑设备包括以下几种。

1) 普通内存设备

普通内存设备表示直接通过地址访问的非持久性存储空间,一般由应用程序事先指定给嵌入式数据库的一块内存空间。普通内存设备的属性包括:空间入口地址、空间大小等。

2) 共享内存设备

共享内存设备表示多个进程所共享的内存空间。嵌入式数据库的共享内存访问的实现方式根据其基于的操作系统不同而不同,但都是通过一个名称进行访问。共享内存设备的

属性包括：空间名称、空间大小等。普通内存设备和共享内存设备为非持久性设备，这类设备嵌入式数据库可以直接通过地址或者名称访问，设备的空间大小决定了数据库的大小。

3）单文件设备

单文件设备表示一个数据文件，嵌入式数据库将使用该文件进行数据存储。文件保存在持久性存储设备上，嵌入式数据库通过文件系统进行创建、读写和删除操作。该文件的容量有限，当该文件被写满后，数据库就无法继续扩大。单文件设备的属性包括：文件名、文件路径、文件大小上限等。

4）普通多文件设备

普通多文件设备表示一组数据文件，嵌入式数据库将使用这一组文件进行数据存储。在这一组文件中，单个文件的容量有限，当第一个文件被写满后，会再创建一个新的数据文件继续写入。普通多文件设备的属性包括：文件组名、文件组路径、单个文件大小上限、数据库大小上限等。

5）RAID0 设备

RAID0 设备表示采用 RAID0 方式组织的多文件存储方式。根据 RAID0 的定义，嵌入式数据库将通过文件系统使用多个数据文件交替进行数据写入。应用程序通过 RAID0 设备的文件组路径可以将多个数据文件分别放置在不同的持久化存储设备上，从而获得 RAID0 带来的并行性提升效果。

6）RAID1 设备

RAID1 设备表示采用 RAID1 方式组织的多文件存储方式。根据 RAID1 的定义，嵌入式数据库将通过文件系统将同样的数据操作同时写入多个文件。应用程序可以通过 RAID1 设备的文件组路径将多个数据文件分别放置在不同的持久化存储设备上，进一步获得 RAID1 带来的冗余备份效果。

单文件设备、普通多文件设备、RAID0 设备和 RAID1 设备均为持久性存储设备，这类设备嵌入式数据库通过文件系统访问。

根据数据管理的要求，逻辑设备还应指定用途，包括：内存数据、内存缓存、数据文件以及日志文件。其中，非持久性设备可以指定为内存数据或者内存缓存，持久性设备可以指定为数据文件或者日志文件。逻辑设备的用途在创建数据库时指定，其中，创建内存数据库只需要指定一个用途为内存数据的非持久性设备。创建持久化存储数据库则需要指定一个用途为内存缓存的非持久性设备、一个用途为数据文件的持久性设备和一个用途为日志文件的持久性设备。

这样，嵌入式数据库将各类存储设备和方式转换为一种或多种逻辑设备的组合，并针对每种逻辑设备实现其存储管理模块，从而对各种物理存储设备的操作转换成对逻辑设备的操作。然后，每种存储管理模块对上层数据管理模块提供相同的访问接口，从而在嵌入式数据库内部实现了对存储细节的隐藏。

3. 采用统一的数据库创建和操作接口

为了向应用程序隐藏存储细节，嵌入式数据库应向应用程序提供统一的数据库创建接口，各种存储设备上的数据库均通过该接口创建，例如：

```
Database_open(dbname,dev,n_dev);
```

其中,dbname 参数表示数据库名,dev 参数为该数据库使用的逻辑设备配置信息,n_dev 表示用到的逻辑设备的数量。例如,当创建一个不需要多进程共享的内存数据库时,dev 中应配置一个用途为内存数据的普通内存设备,设备数量参数 n_dev 为 1。当创建一个在闪存上的一个文件中保存数据的持久性数据库时,dev 中应配置一个用途为内存缓存的普通内存设备、一个用途为数据文件的单文件设备和一个用途为日志文件的单文件设备,设备数量参数 n_dev 为 3。

统一数据操作接口根据实际情况可采用导航式 API 或者 SQL 语言,应用程序不需要了解存储细节,可直接通过数据操作接口进行操作,嵌入式数据库根据数据库对应的逻辑设备去调用逻辑设备管理接口,进而实现在存储器上的数据存取。

3.3 嵌入式数据库访问算法

数据库访问算法是一个数据库产品的核心,掌握数据库的访问算法原理,也就掌握了这个数据库的精髓。由于嵌入式数据库的种类繁多,关系型数据库和非关系型数据库等均得到广泛应用,因此很难有一种完全覆盖所有嵌入式数据库的访问算法。本节主要介绍 4 种代表性的嵌入式数据库访问算法——B 树、Hash、Queue、Recno 算法。Berkeley DB 数据库对前 4 种算法有良好的支持,SQLite 数据库则采用了存储表数据用 B+树,存储表索引用 B-树。

3.3.1 数据的存储组织

存储组织包括数据表示和存储空间管理两个方面。数据表示是数据库中应用数据的物理存储的表现方式,它受到数据库系统所采取的存储模型的制约。存储空间管理是对存储设备可用存储空间的应用组织策略(3.2 节已经介绍),它的目标有两个:高效利用存储空间和为快速的数据存取提供便利。

在关系型数据库管理系统中,数据字典、索引、应用数据是存储的主要内容。

1. 数据字典

对关系数据的描述存储在数据库的数据字典中。与数据本身相比,数据字典的特点是数据量比较小,使用的频率高,因为任何数据库操作都要参照数据字典的内容。数据字典在网状、层次数据库中常常用一个特殊的文件来组织。所有关于数据的描述信息存放在一个文件中。关系数据库中数据字典的组织通常与数据本身的组织相同。数据字典按照不同的内容在逻辑上组织为若干张表,在物理上就对应若干文件而不是一个文件。由于每个文件中存放数据量不大,所以可简单地用顺序文件来组织。

嵌入式数据库常用系统表来组织数据字典。所谓系统表就是由数据库系统自动建立的表,在数据库装入的时候建立并被初始化。

2. 索引

索引是关系数据库的存取路径。在网状和层次数据库中,存取路径是用数据之间的联系来表示的,因此索引已与数据结合并固定下来。

关系数据库中,存取路径和数据是分离的,对用户是隐蔽的。存取路径可以动态建立、删除。存取路径的物理组织通常采用 B 树类文件结构和 Hash 文件结构。在一个关系上可

以建立若干个索引。有的系统支持组合属性索引，即在两个或两个以上属性上建立索引。索引可以由用户用 CREATE INDEX 语句建立，用 DELETE INDEX 语句删除。

在执行查询时，嵌入式数据库管理系统查询优化模块也会根据优化策略自动地建立索引，以提高查询效率。由此可见，关系数据库中存取路径的建立是十分灵活的。

在嵌入式数据库中，工作版本常驻主存，数据的文件形式组织不是重点，但在主存中，仍然需要构建快速的存取路径才能取得高的存取效率；而且，在数据从外存调入主存时，文件型索引将加快这一过程。

3. 应用数据

关于数据自身的组织，嵌入式数据库管理系统可以根据数据和处理的要求自己设计文件结构，也可以从操作系统提供的文件结构中选择合适的加以实现。目前，操作系统（包括嵌入式操作系统）提供的常用文件结构有顺序文件、索引文件、索引顺序文件、Hash 文件和 B 树类文件等。

数据库中数据组织与数据之间的联系是紧密结合的。在数据的组织和存储中必须直接或间接、显式或隐含地体现数据之间的联系，这是数据库物理组织中主要考虑和设计的内容。

在网状和层次数据库中，常用邻接法和链接法实现数据之间的联系。对应到物理组织方式中，就是要在操作系统已有的文件结构上实现数据库的存储组织和存取方法。例如 IMS 数据库中，操作系统提供的低级的存取方法有 SAM、ISAM、VSAM、OSAM。IMS 数据库管理系统在此基础上设计了 HSAM、HISAM、HDAM、HIDAM 四种数据库的存储组织和相应的存取方法。其中，HSAM 层次顺序存取方法按照片段值的层次序列码的次序顺序存放各片段值。而层次序列码正体现了数据之间的父子和兄弟联系，这是一种典型的按物理邻接方式实现数据之间联系的方法。在这种存储方法中，整个数据库中不同片段型的数据均存储在一个 SAM 文件中。网状数据库中最常用的组织策略是：各记录型分别用某种文件结构组织，记录型之间的联系——SET 用指引元方式实现。即在每个记录型中，增加数据库管理系统控制和维护的系统数据项——指引元，它和用户数据项并存于同一个记录中。

关系数据库中实现了数据表示的单一性。实体及实体之间的联系都用一种数据结构——“表”来表示。在数据库的物理组织中，每一个表通常对应一种文件结构，因此数据和数据之间的联系两者组织方式相同。

在嵌入式数据库中，数据将分为“永久版本”和“临时版本”。数据库在运行的大部分时间里都只关心临时版本，只在系统空闲或显式要求的情况下才将临时版本中的数据更新到永久版本中。这是一种乐观的持久化策略。因此，对嵌入式数据库来说，首先关注的是数据在主存中的高效的存取，其次才会考虑数据的文件组织形式，尽可能地提高数据在内外存之间的调入、调出效率。

3.3.2　B 树访问算法

1. B－树和 B＋树的定义

定义 3.1　一棵 m 阶的 B－树，或为空树，或为满足下列特性的 m 叉树：

树中每个节点至多有 m 棵子树；

若根节点不是叶子节点,则至少有两棵子树;

除根之外的所有非终端节点至少有$\lceil m/2 \rceil$棵子树;

所有的非终端节点中包含下列信息数据$(n, A_0, K_1, A_1, K_2, \cdots, K_n, A_n)$,其中:

$K_i(i=1,\cdots,n)$为关键字,且$K_i < K_{i+1}(i=1,\cdots,n-1)$;$A_i(i=0,\cdots,n)$为指向子树根节点的指针,且指针$A_{i-1}$所指子树中所有节点的关键字均小于$K_i(i=1,\cdots,n)$,$A_n$所指子树中所有节点的关键字均大于$K_n$,$n(\lceil m/2 \rceil - 1 \leqslant n \leqslant m-1)$为关键字的个数(或$n+1$为子树个数)。

所有的叶子节点都出现在同一层次上,并且不带信息(可以看作是外部节点或查找失败的节点,实际上这些节点不存在,指向这些节点的指针为空)。

定义 3.2 B+树是应文件系统所需而出现的一种B-树的变形树。一棵B+树和m阶的B-树的差异在于:

有n棵子树的节点中含有n个关键字;

所有的叶子节点中包含了全部关键字的信息,以及指向含这些关键字记录的指针,且叶子节点本身依关键字的大小自小而大顺序链接;

所有的非终端节点可以看成是索引部分,节点中仅含有其子树(根节点)中的最大(或最小)关键字。

2. B-树的查找过程

由B-树的定义可知,在B-树上进行查找的过程和二叉排序树的查找类似。例如,在图3-1所示的一棵B-树上查找关键字47的过程如下:从根开始,根据根节点指针t找到$*a$节点,因为节点中只有一个关键字,且查找值47>关键字35,则若存在必在指针A_1所指的子树内,顺指针查找到$*c$节点,该节点有两个关键字43和78,又43<47<78,则存在必在A_1所指子树中。顺指针找到$*g$节点,在此节点中顺序查找到关键字47,查找成功。

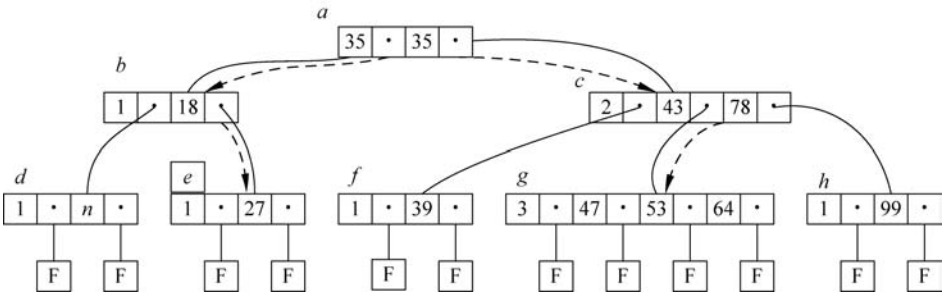


图 3-1 4阶B-树

查找不成功的过程也类似。例如,在同一树中查找23,从根开始,23<35,则顺指针A_0找到$*b$节点,又$*b$中只有一个关键字18且23>18,顺指针A_1找到$*e$节点。同理因为23<27,顺指针A_0向下查找,因为此时指针所指为叶节点,说明B-树中不存在关键字23,查找失败并终止。

3. B-树查找分析

从以上查找算法中可以看出,在B-树中进行查找包含以下两种基本操作。

(1) 在B－树中查找节点。

(2) 在节点中查找关键字。

由于B－树通常存储在磁盘上，则前一查找操作是在磁盘上进行，后一查找操作是在内存中进行，即在磁盘上找到指针 p 所指节点后，先将节点中的信息读入内存，然后再利用顺序查找或折半查找查询等于 K 的关键字。显然，在磁盘上进行一次查找比在内存中进行一次查找的时间消耗多得多，因此，在磁盘上进行查找的次数，即待查找关键字所在节点在B－树上的层次树，是决定B－树查找效率的首要因素。

现在考虑最坏的情况，即待查找节点可能出现在B－树上的最大层次数。也就是含 N 个关键字的 m 阶B－树的最大深度。以一棵3阶的B－树为例，按B－树的定义，3阶的B－树上所有非终端节点至多可有两个关键字，至少有一个关键字。因此，若关键字个数≤2时，树的深度为2(即叶子节点层次为2)；若关键字个数≤6时，树的深度不超过3。反之，B－树的深度为4，则关键字的个数必须≥7，此时，每个节点都含有可能的关键字的最小数目。

下面讨论深度为 $l+1$ 的 m 阶B－树所具有的最少节点数。根据B－树的定义，第一层至少有1个节点；第二层至少有2个节点；因除根之外的每个非终端节点至少有($\lceil m/2 \rceil$)棵子树，则第三层至少有 $2(\lceil m/2 \rceil)$ 个节点，…，以此类推，第 $l+1$ 层至少有 $2(\lceil m/2 \rceil)^{l-1}$ 个节点。而 $l+1$ 层的节点为叶子节点。若 m 阶B－树中具有 N 个关键字，则叶子节点即查找不成功的节点为 $N+1$，由此有：$N+1 \geqslant 2(\lceil m/2 \rceil)^{l-1}$。

4. B－树的插入和删除

B－树的生成是从空树开始，逐个插入关键字而得。但由于B－树节点中的关键字个数必须≥$\lceil m/2 \rceil-1$，因此，每次插入一个关键字不是在树中添加一个叶子节点，而是首先在最底层的某个非终端节点中添加一个关键字，若该节点的关键字个数不超过 $m-1$，则插入完成，否则要产生节点的“分裂”。一般情况下，节点可如下实现分裂。假设 $*p$ 节点中已有 $m-1$ 个关键字，当插入一个关键字之后，节点中含有信息为：$(m, A_0, K_1, A_1, K_2, \cdots, K_m, A_m)$，其中 $K_i < K_{i+1}$，$(1 \leqslant i < m)$，此时可将 $*p'$ 节点中包含信息$(m-\lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil+1}, A_{\lceil m/2 \rceil+1}, \cdots, K_m, A_m)$，与关键字 $K_{\lceil m/2 \rceil+1}$ 和指针 $*p'$ 一起插入到 $*p$ 的双亲节点中。

反之，若在B－树上删除一个关键字，则首先应找到该关键字所在节点，并从中删除之，若该节点为最下层的非终端节点，且其中的关键字数目不少于$\lceil m/2 \rceil$，则删除完成，否则要进行“合并”节点的操作。假若所删除关键字为非终端节点中的 K_i，则可以用指针 A_i 所指子树中的最小关键字 Y 代替 K_i，然后在相应的节点中删去 Y。

如果删除的节点为最下层的非终端节点中的关键字，则需分以下三种情况讨论。

(1) 被删关键字所在节点中的关键字数目不小于$\lceil m/2 \rceil$，则只需从该节点中删去该关键字 K_i 和相应指针 A_i，树的其他部分不变。

(2) 被删关键字所在节点中的关键字数目等于$\lceil m/2 \rceil-1$，而与该节点相邻的右兄弟(或左兄弟)节点中的关键字数目大于$\lceil m/2 \rceil-1$，则需将其兄弟节点中的最小(或最大)的关键字上移到双亲节点中，而将双亲节点中小于(或大于)且紧靠该上移关键字的关键字下移到被删关键字所在节点中。

(3) 被删关键字所在节点和其相邻的兄弟节点中的关键字数目均等于$\lceil m/2 \rceil-1$。假

设该节点有右兄弟,且其右兄弟节点地址由双亲节点中的指针 A_i 所指,则在删去关键字之后,它所在节点中剩余的关键字和指针,加上双亲节点中的关键字 K_i 一起,合并到 A_i 所指兄弟节点中(若没有右兄弟,则合并到左兄弟节点中)。

B+树上的随机查找、插入和删除过程与B-树类似,只是在查找时,若非终端节点上的关键字等于给定值,并不终止,而是继续向下直到叶节点。因此,无论查找成功与否,B+树上的每次查找都是走了一条从根到叶节点的路径。

5. Berkeley DB 的 B 树访问方式

Berkeley DB 的 B 树访问方式采用 B+树的结构,关键字有序存储,并且其结构能随数据的插入和删除进行动态调整。不同于一般的 B+树方式,为了代码的简单,Berkeley DB 没有实现对关键字的前缀码压缩。B 树算法支持高效的数据查询、插入、删除等操作。关键字可以为任意的数据结构。使用这种方式在树中查找、插入和删除的时间复杂度为 $O(\log_b N)$,其中 b 为每个页面保存的键(Key)的数量。B 树访问方式是 Berkeley DB 中最常用的一种数据访问方式。

B 树访问方式将键/值对(Key/Value pairs)记录保存为叶节点页面(leaf pages),把键/子树页地址对(Key/Child page address)保存为内部节点。键(Key)在树中是有序存储,排序方式是由数据库创建时指定的比较函数决定。树的叶节点级页面带有指向兄弟节点页面的指针,这样可以简化遍历操作的复杂度。B 树访问方式支持精确查找和范围查找(大于等于指定键值的范围),还支持插入、删除和遍历所有记录的功能。

当树中所有页面已经填满时,再向数据库中插入新记录就会导致节点页面分裂,这会使得半数左右的键被转移到同级的新页面中。目前大多数的 B+树算法都会在节点分裂后产生这种仅利用了半页空间的现象,而这通常会降低系统性能(因为在插入节点和节点分裂后依然需要保持 B+树的有序性和其他特性)。为了解决这个问题,Berkeley DB 记录了插入的顺序,并且采用非均衡的分裂节点页面的做法来保持页面的高填充率。

删除节点时,空页面会被接合到分裂前的单页上去。这种访问方式并不对其他页面做平衡处理(尽管在 B+树节点插入和删除时会由于节点的分裂和合并而需要对树中其他节点子树做调整来保证树的平衡),也不在页面间移动键来保持树的平衡。尽管这个做法可能会增加节点的搜索时间,但是保持树的完好平衡会增大代码量和复杂程度,从而降低数据库更新效率和可能导致更多的死锁。

同样为了简化操作,Berkeley DB 的 B+树访问方式不对内部节点或叶子节点的键做预比较。Berkeley DB 的 B 树查找函数为__bam_search()。

函数原型:

```
int __bam_search __P(DBC * dbc,                    //数据库游标
db_pgno_t root_pgno,                               //根页面号
const DBT * key,                                   //键对象
u_int32_t flags,                                   //查找参数
int slevel,                                        //查找所在层
db_recno_t * recnop,                               //逻辑记录号指针
int * exactp);
```

3.3.3　Hash 访问算法

1. 哈希(Hash)函数和哈希表及相关定义

定义 3.3　在查找时往往希望不经过任何比较，一次存取便能得到所查记录，那就必须在记录的存储位置和它的关键字之间建立一个确定的对应关系 f，使每个关键字和结构中一个唯一的存储位置相对应。因而在查找时，只要根据这个对应关系 f 就能找到给定值 K 的映像 $f(K)$。若结构中存在关键字和 K 相等的记录，则必定在 $f(K)$ 的存储位置上，因此，进行比较便可直接取得所查记录。这个对应关系 f 则称为哈希(Hash)函数，按照这个思想建立的表就是哈希表。

哈希函数是一个映像，因此哈希函数的设定很灵活，只要使得由任何关键字所得的哈希函数数值都落在表长允许范围之内即可；对不同的关键字可能得到同一哈希地址，即 key1≠key2，而 $f(\text{key1})=f(\text{key2})$，这种现象称为冲突(collision)。具有相同函数值的关键字对该哈希函数来说称作同义词(synonym)。一般说来，冲突只能尽可能地少，而不能完全避免，因为哈希函数是从关键字集合到地址集合的映像。

常用的哈希函数构造方法有直接定址法、数字分析法、平方取中法、折叠法、除留余数法、随机数法等。常用的处理冲突的方法有开放定址法、再哈希法、链地址法、建立一个公共溢出区等。

2. 哈希表的查找

在哈希表上进行查找的过程和哈希造表的过程基本一致。给定 K 值，根据造表时设定的哈希函数求得哈希地址，若表中此位置上没有记录，则查找不成功；否则比较关键字，若和给定值相等，则查找成功；否则根据造表时设定的处理冲突方法查找下一地址，直到哈希表中某个位置为“空”或者表中所填记录的关键字等于给定值时为止。

3. 哈希表查找分析

虽然哈希表在关键字与记录的存储位置之间建立了直接映像，但由于“冲突”的产生，使得哈希表的查找过程仍然是一个给定值和关键字比较的过程。因此，仍需以平均查找长度作为衡量查找效率的量度。查找过程中需和给定值比较的关键字的个数取决于三个因素：哈希函数、处理冲突的方法和哈希表的装填因子。哈希表的装填因子定义为

$$\alpha=\frac{\text{表中填入的记录数}}{\text{哈希表的长度}}$$

α 表示哈希表的装满程度。直观地看，α 越小，发生冲突的可能性就越小；反之，α 越大，表中已填入的记录越多，再填记录时，发生冲突的可能性就越大，则查找时，给定值需与之进行比较的关键字的个数也就越多。

若记录数为 n，哈希表的平均查找长度是 α 的函数，因此不论 n 多大，总可以选择一个合适的装填因子，以便将平均查找长度限定在一个范围内。

需要说明的是，如果要在非链地址处理冲突的哈希表中删除一个记录，则需在该记录的位置上填入一个特殊的符号，以免找不到在它之后填入的“同义词”记录。

4. Berkeley DB 的 Hash 访问方式

Berkeley DB Hash 访问方式的数据结构是扩展线性 Hash 表(Extended Linear

Hashing,也称动态 Hash 表)。当哈希表增大时扩展哈希表会调整哈希函数,以期保证所有的哈希桶不被填满并保持在一个稳定的状态。关键字可以为任意的数据结构。

Berkeley DB 的 Hash 访问算法支持插入、删除和精确查找(不支持范围查找)。应用程序可以遍历所有保存在表中的记录,但它们返回的顺序是不确定的。

Berkeley DB 的 Hash 查找函数为__ham_lookup(),函数原型如下:

```
static int __ham_lookup(DBC * dbc,              //游标指针
const DBT * key,                                //键结构体指针
u_int32_t sought,                               //查找方式
db_lockmode_t mode,                             //锁模式
db_pgno_t * pgnop);                             //页面号指针
```

另外,Berkeley DB 还提供了 4 种不同的哈希函数,分别为__ham_func2()、__ham_func3()、__ham_func4()和__ham_func5()。

3.3.4 Queue 访问算法

1. Queue 定义

Queue(队列)是一种先进先出(First In First Out,FIFO)的线性表。它只允许在表的一端进行插入,而在另一端删除元素。最早进入队列的元素最早离开。在队列中,允许插入的一端称为队尾(Rear),允许删除的一端则称为队头(Front)。假设队列为 $q=(a_1,a_2,\cdots,a_n)$,那么 a_1 就是队头元素,a_n 则是队尾元素。队列中的元素是按照 $a_1,a_2,\cdots,a_n$ 的顺序进入的,退出队列也只能按照这个次序依次退出,也就是说,只有在 $a_1,a_2,\cdots,a_{n-1}$ 都离开队列之后,a_n 才能退出队列。图 3-2 为队列的示意图。

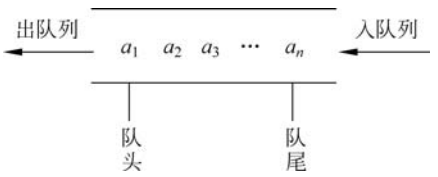


图 3-2 队列示意图

2. Berkeley DB 的 Queue 访问方式

Berkeley DB 的 Queue(队列)访问方式用逻辑记录号来保存定长记录。每条记录都使用一个逻辑号作为键。Queue 访问方式使得用户能够快速在队列尾部插入记录,并能够方便地从队列开头删除记录或者返回记录。

Queue 算法与 Recno 方式接近,只不过记录的长度为定长。数据以定长记录的方式存储在队列中,插入操作把记录插入到队列的尾部,相比之下插入速度是最快的。

当较小编号的记录被添加或被删除时,开发者可以选择让记录自动重新编号,这样,新的键就可以插入到已有键的中间。

这种访问方式由于使用了记录级的锁(Record Level Locking),所以不是太常用。但是当需要在应用程序中对队列使用并发访问时,这种访问方式会获得很高的性能。

Berkeley DB 的查找函数为__qam_c_get(),函数原型如下:

```
int __qam_c_get(DBC * dbc,                      //游标指针
DBT * key,                                      //键对象指针
DBT * data,                                     //数据对象指针
u_int_32_t flags,                               //数据访问参数
db_pgno_t * pgnop)                              //页面指针
```

3.3.5　Recno 访问算法

B 树、Hash 算法是数据库中常用的索引方式。但是，当要存储的记录太庞杂而无法从中提取合适的关键字时，采用 B 树或 Hash 方式存储记录效率就比较低。在这种情况下，采用 Recno 方式能更好地提高数据库存取效率。

Recno 访问算法是在 B 树访问算法的基础上发展而来的，它不仅继承了 B 树方式适合随机或顺序存储数据的特点，而且比 B 树更适合处理各种庞杂的记录。Recno 算法要求每一条记录都有一个逻辑记录号，逻辑记录号由算法本身生成。实际上，这和关系型数据库中逻辑主键通常定义为 int AUTO 型是同一个概念。Recno 访问算法提供了一个存储有序数据的接口。记录的长度可以为定长或不定长。

Recno 模型如图 3-3 所示，它和 B 树模型类似，由两部分组成，上层是 B 树索引，下层所有的叶节点组成一个顺序集。索引树中的关键字只起到指路标的作用，不能由它直接找到记录，它也不一定就存在。当随机查找时，通过索引树找到要查找的记录，从根部开始查找。当顺序查找时，可以顺着链头开始，即先用随机查找的方法找到所要求的记录后，再从这一记录开始顺序查找其他的记录。

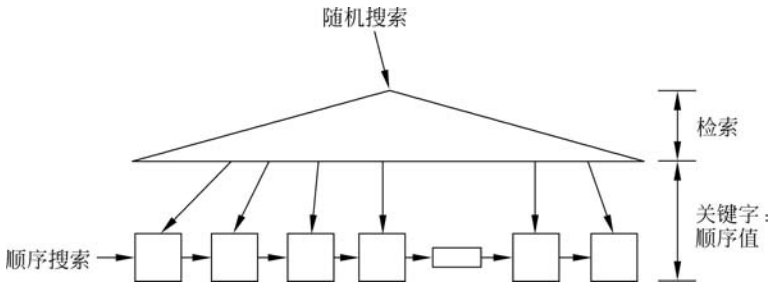


图 3-3　Recno 模型

Recno 的叶节点格式保存了记录的关键字和数据，而中间的节点保存了关键字和子页面地址。Recno 方式以逻辑记录号作为关键字，而数据可以是任意长度的字符串，因此很适合处理各种庞杂的数据。图 3-4 为 Berkeley DB 中的 Recno 节点。

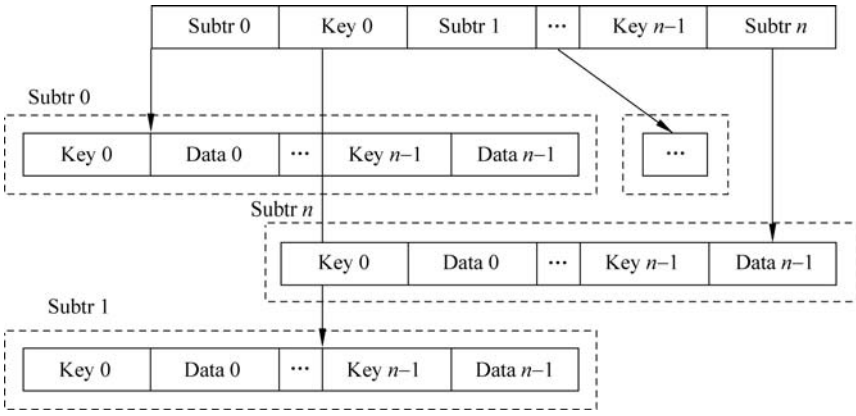


图 3-4　Recno 节点

Recno 访问方式可以自动分配逻辑记录号给每一条记录,并能够靠此记录号查找和更新记录,Recno 允许应用程序通过线性数字进行快速查找。

1. Recno 查找

查找关键字为 K 的记录,首先从根部开始,比较要找的关键字 K 与节点中的 key_0,key_2,…,key_{n-1}。当 $k=key_i$ 时,仍要继续向下查找直到在叶节点找到关键字 K 为止。

若在叶节点仍未找到,则查找的关键字不存在,反之则查找成功;当 $k<key_0$ 时,继续向下在 $subtr_0$ 中查找;当 $key_i<k<key_{i+1}$ 时,顺着 $subtr_{i+1}$ 向下找。当 $k>key_{n-1}$ 时,继续在 $subtr_n$ 中查找。

使用 Recno 方式不仅可以精确地查找某一个关键字,还可以查找在一定范围内的所有关键字。

2. Recno 的插入和删除

前面提到 Recno 算法是从 B 树算法发展而来,Recno 插入算法的基本思想和 B 树算法极为相像。这里仅介绍 Recno 的删除算法。

在 Recno 中删除记录相对比较简单,因为记录一定是叶节点,因此只要找到这个叶节点,将这个节点删除即可。注意,若找到的节点是 Recno 的中间节点,并不需要删除这个节点,因为它不是真正的记录,只是起到分界的作用而已。只有当删除一个关键字使得整个页面为空而需要与其他页面合并时,才有可能需要改动索引树中的关键字。

Recno 中的节点分裂和合并与 B 树中的方法一致。

Recno 的查找函数为__bam_rsearch(),函数原型如下:

```
int __bam_rsearch(DBC * dbc,                    //游标指针
db_recno_t * recnop,                            //记录号指针
u_int32_t flags,                                //数据访问参数
int stop,                                       //停止标记
int * exactp);
```

3.3.6 访问算法的特点

4 种访问算法的特点如下。

1. B 树

关键字有序存储,其结构能随数据的插入和删除进行动态调整。B 树算法支持高效的数据查询、插入、删除等操作。关键字可以为任意的数据结构。

适用于 Key 为复杂类型且有序的情况。

2. Hash

Berkeley DB 中实际使用的是扩展线性 Hash 算法(Extended Linear Hashing),它可以根据哈希表的增长进行适当的调整。关键字可以为任意的数据结构。

适用于 Key 为复杂类型,数据较大且 Key 随机分布的情况。

3. Recno

该算法要求每一个记录都有一个逻辑记录号,逻辑记录号由算法本身生成。相当于关

系数据库中的自动增长字段。Recno 建立在 B 树算法之上，提供了一个存储有序数据的接口。记录的长度可以为定长或不定长。

适用于 Key 为逻辑记录号且非高并发的情况。

4. Queue

与 Recno 方式接近，只不过记录的长度为定长。

适用于 Key 为逻辑记录号、定长记录且高并发的情况。

3.4　实时事务处理技术

当多个用户并发地存取数据库时，会产生多个事务同时存取同一数据的情况。若对并发不加控制就可能会读取或者存储不正确的数据，破坏数据库的一致性和数据的完整性。所以数据库管理系统必须提供并发控制机制，以避免可能出现的错误，如丢失修改、不可重复读、读“脏”数据。并发控制就是要用正确的方式调度事务操作，使一个用户事务的执行不受其他事务的干扰，从而避免造成数据的不一致。但是传统的并发控制策略不能确保嵌入式实时数据库对实时性能的要求，因此，需要在充分挖掘实时数据特征和实时事务特征的基础上，对嵌入式数据库的实时事务调度策略及并发控制策略进行分析，以提高事务执行的并发度。本章 3.4 节和 3.5 节对实时事务和实时并发控制进行分析。

从数据库的角度看，一个实时系统典型地由 3 个紧密耦合的子系统组成：被控系统、执行控制系统、数据系统。被控系统就是所考虑的应用过程，称为外部环境或物理世界；执行控制系统就是监视被控系统的状态，协调和控制它的活动，称为逻辑世界；数据系统有效地存储、操纵与管理实时（准确和及时）信息，称为内部世界；执行控制系统和数据系统称为控制系统。内部世界的状态是物理世界状态在控制系统中的映像，执行控制系统通过内部世界状态而感知外部环境状态，且基于此而与被控系统交互作用，所有这些都与时间紧密相连，因而实时数据库主要在数据和事务上表现出各种不同特点。

3.4.1　数据特征

在实时数据库中，数据只在一定时间内是“流行”的，其值随外部环境状态变化而改变。所以，用户不能只考虑数据库内部状态的一致性，还必须考虑外部状态与内部状态之间的一致性；也不能认为使用数据时，简单地提供其最新的值就是最合适的，还必须考虑它与其他被使用数据间的“时间一致性”。这些就是实时数据库的数据特征。概括起来说，实时数据库的数据一致性包括内部一致性、外部一致性、相互一致性，下面具体给出它们的概念。

一个的数据对象为一个三元组 d：<v，tp，evi>。其中 d_{v}，d_{tp}，d_{evi} 分别为 d 的当前状态值；观测时标（Observation Timestamp），即采样 d 对应的现实世界对象的值的时间；外部有效期（External Validity Interval），即自 d_{evi} 算起 d_{tp} 具有外部或绝对有效性的时间长度。

1. 内部一致性

内部一致性就是传统意义上的数据库内部的一致性。

定义 3.4　一个数据 d 是内部一致的，当且仅当 d_{v} 满足所有对其预先定义的数据库内

部的完整性限制和一致性要求。

所有对数据库的操作都是以事务的形式进行的,传统的数据库可通过可串行化调度来保证并发事务的正确性,通过事务阻塞、回滚和撤销来消除可能出现的数据不一致性问题,因此,在传统数据库中可串行化是数据库一致性的重要标准。

2. 外部一致性

在实时数据库中,数据库依靠逻辑世界和物理世界的频繁交换作用来获取准确、及时的数据,故一个正确的数据库状态必须与物理世界当时的状态一致。即事务使用的数据库中的值 d_v 是在其有效时间范围 d_{evi} 内。

定义 3.5 一个数据 d 是外部一致的,当且仅当$(t_c - d_{tp}) \leqslant d_{evi}$($t_c$ 为当前或检测时间)。数据的外部有效期 d_{evi} 可以通过足够多的物理世界对应参数的取样来获得,外部一致性与时间限制相联。必要时只有先牺牲内部一致性而确保外部一致性,然后再恢复内部一致性。

数据的外部时间一致性要求表明存于数据库中的采样数据滞后于被采样的实际过程数据不得超过一定的时间。

3. 相互一致性

在过程控制中,一组相关数据被使用时,存在着它们之间在时间上的相互(或相对)一致性问题。

定义 3.6 用来做决策或导出新数据的一组相关数据称为一个相互(相对)一致集,每一个这样的数据集 R 都有一个与之相联的相互有效期,记为:R_{mvi}。

定义 3.7 设 R 是一个相互一致集,$d \in R$。D 是相互(或相对)一致的,当且仅当:

$$\forall d' \in R(|d_\varphi - d'_\varphi|) \leqslant R_{mvi})$$

相互一致性保证了用于做决策或导出新数据的一组数据是在公共的有效时间范围内彼此接近地产生的。R_{mvi} 的获得不像 d_{evi} 那么简单,且它们之间存在着相关性,R 中各数据的 d_{evi} 的最小者对 R_{evi} 起着决定作用。

4. 数据库状态正确性

数据库状态正确性意味着内部、外部和相互一致性的全部。

定义 3.8 若一个数据既是外部一致的又是相互一致的,则称它是时间一致的。

定义 3.9 一个数据具有正确的状态,当且仅当它同时是时间一致和内部一致的。若实时数据库中的每一个数据都具有正确的状态,则称该实时数据库满足数据的正确性。

在实时数据库中,数据库的正确性依赖于实时的事务调度策略、并发控制协议以及实现机制。

3.4.2 实时事务特征

实时事务是嵌入式实时数据库实时性的核心体现。在经典数据库理论中,事务具有ACID特性,即“原子性”“一致性”“隔离性”和“持久性”,其特点如下。

A:说明所设计的事务具有“原子性”(Atomicity),也就是某一事务在执行时要么全部执行、要么全部不执行,一定作为一个整体操作。

I：说明所设计的事务具有“隔离性”(Isolation)，也就是在执行所设计的事务时，别的事务在此时不会也执行。

C：说明所设计的事务具有“一致性”(Consistency)，也就是要使数据库的全部数据满足一致性的约束，希望所发生的事务能够使数据库具有一致性。

D：说明所设计的事务具有“持久性”(Durability)，也就是某个事务只要执行成功了，它所涉及数据的变化不应该丢失。

但是，在实时数据库管理系统中事务则具有根本性的区别，实时事务的特性主要表现在如下方面。

(1) 定时限制：事务的执行具有显式的时限，如期限、截止时间等。这是由于控制系统不断跟踪被控制系统而引起的，它要求实时数据库有时间处理机制。

(2) 正确性：事务的正确性不仅在于逻辑结果的正确性，而且要求时间正确性，即必须在给定的截止期内提交。

(3) 可预测性：在有些系统中，事务错失截止期将导致系统的崩溃，因此要求能够预测这些事务的执行时间，以确保事务能够满足截止期，从而保证系统的正常运行。

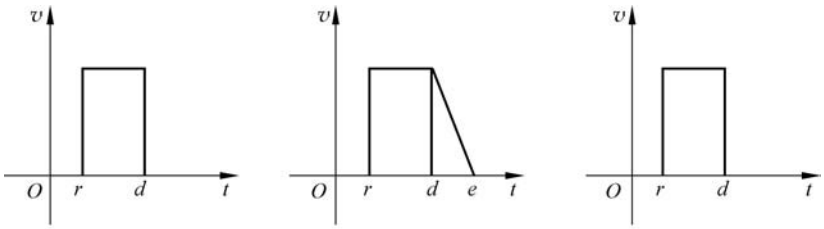
(4) 可恢复性：当事务由于错失截止期而被夭折或者由于数据访问冲突必须重启时，该事务的所有操作都能够被取消并恢复到原来数据库的状态。

(5) 重要性：不同事务的提交会给系统带来不同的价值。为了实现系统的高价值，价值高的事务将具有优先执行权。

因此，传统事务的特性已经不适合实时数据库中的事务。实时数据库中事务最重要的是定时性，即事务的执行具有显式的定时限制，典型表现为截止期。事务(或系统)的正确性不仅依赖于其逻辑结果的正确性，还依赖于逻辑结果产生的时间。

实时事务按关键性可以分为硬实时事务、软实时事务、固实时事务。

图 3-5 中(a)～(c)分别是硬、软、固实时事务的典型例子。其中，v、t 两坐标轴分别为价值函数和时间；d 为截止期；e 为最终有效时间；r 为放行或启动时间，当 $t<r$ 时，$v(t)=0$，表示在事务未准备好以前启动是无价值的。



(a) 硬实时事务　(b) 软实时事务　(c) 固实时事务

图 3-5　实时事务按关键性分类

(1) 硬实时事务：用于安全紧急性活动，超过截止期则价值函数取负值。硬实时事务在错失执行期限后将会导致灾难性的后果。因此，在系统发现事务具有很大的负值时，必须采取预定的紧急措施。

(2) 软实时事务：这类事务在超截止期后，价值函数会不断下降，到被称为最终有效时间的某一时刻时，价值函数取 0。软实时事务允许其执行有一定程度的超期。

(3) 固实时事务：一旦到达截止时间，价值函数取 0，但不为负值。

根据一般实时应用系统的活动(事务)特点，可以确定如下实时事务模型。

General-Tran：一般事务(即传统意义的事务)。

Loop-Tran：循环事务，循环执行的一般事务或结构事务。

Endless-Tran：无终止事务，具有很长执行期的一般事务或结构事务。

Long-life-Tran：长寿事务，循环(Loop)或无终止(Endless)事务。

Nested-Tran：嵌套事务，由子事务以树形结构组成的事务。

Multilevel-Tran：多层事务，由子操作以树形结构组成的事务。

Split-Tran：分裂事务，一个事务分支出另一个事务。

Cooperative-Tran：合作事务。

Joint-Tran：合并事务。

Flat-Coop：平坦通信事务。

Hierarchical-Coop：层次间通信合作的事务。

Complex-Tran：复杂事务，可由上面列出的各事务构成。

3.4.3 实时事务调度

实时事务的调度技术也是实时数据库的关键组成。

1. 优先级

嵌入式数据库的实时事务调度一般都是依据基于优先级的算法进行调度的，因为事务的优先级决定了事务的重要性，比较重要的事务应该首先完成。而优先级的决定取决于多种因素，常用的有事务的释放时间、事务的截止时间、事务的已执行时间、事务的空余时间、事务的关键程度等。大多数嵌入式实时数据库系统用事务的截止时间描述事务的紧迫度。决定一个事务截止期的时间描述有很多，有的直接给出事务的截止期，有的可能只有开始时间的限定或者执行时间的确定。根据不同时间描述可以确定事务时间的松散度，基于松散度和截止时间共同计算事务优先级，有如下一些策略。

1) 最早放行最优先

该策略将最高优先级分派给最早开始执行的事务。这种算法只看事务的开始时间，它可能会先执行一个开始时间早，但要运行很久的事务，而不去执行一个开始时间稍迟，但要求很快执行完的事务，这不能满足实时系统的要求。

2) 截止期最早最优先

截止期最早最优先的策略是最早截止期事务的优先级最高，即根据截止期时间分派优先级，截止期越早，优先级越高。

3) 可达截止期最早最优先

它使具有最早的可达截止期者的优先级最高。所谓一个事务 T 的截止期是当前“可达到”的，指的是 $t+(E-P)\leqslant d$。这里 t 为当前时间，E、P 分别为事务 T 的执行时间估算和已执行时间，d 为 T 的截止期。

4) 空余时间最短最优先

事务 T 的空余时间 $S=d+(t+E-P)$，即根据推迟 T 的执行而仍然满足截止期的可推迟时间量估算。它考虑了当前时间与剩余执行时间估算 $C=t+E-P$，故随事务的停止

执行，其优先级动态上升。而若事务在执行，则因当前时间 t 与事务的已执行时间 P 同步增加，故 S 不变，因而优先级也不变。

5）价值最高最优先

在系统中，一个实时事务的价值可以综合事务的关键性和事务的特征（如硬实时、软实时事务对实时性的不同要求）等因素制定。例如，一个关键的硬实时事务在系统中应具有较高的价值。每个事务都有价值函数，其值最大者最优先。问题是如何合理地构造价值函数，如下是一个例子：

$$V(t)=c(\omega_1(t-t_S)-\omega_2 d+\omega_3 P-\omega_4 S)$$

其中 t、d、P 和 S 的意义同上，c、t_S 分别为 t 的危急度、开始时间，ω 为加权因子。

6）价值密度最大最优先

事务完成时的期望价值与实现该价值所需计算量的比最大者优先级最高。

价值密度函数为

$$\mathrm{VD}=\frac{v(\tau+c)}{c}=\frac{v(\tau+E-P)}{E-P}$$

上述各种优先级分配策略各有优缺点，这里对(1)～(4)时间优先级分析如下。

(1) 最早放行最优先。优点是简单易行，但对于实时数据库来说，其缺点是显而易见的，由于没有考虑事务的截止期，可能会使一个刚到达的紧急事务等待一个先来但并不紧急的事务。在某些情况下，这样的事务处理会给系统带来灾难，通常不用于实时事务处理。

(2) 截止期最早最优先。该策略表达的意思很简单，使“截止期”最早的事务具有最高的优先级，在很多情况下，配上适当的并发控制协议，其处理结果十分理想。它使得最需要处理（截止期最早）的事务首先获得系统资源，但由于没有考虑事务处理所需的时间，可能会将最高优先级分配给一个要过或已过截止期的事务，从而导致有机会能够在截止期内完成的事务也因被推迟而超时，以致事务成功率下降。

(3) 可达截止期最早最优先。通过对事务的执行时间进行预分析来判断事务的截止期是否可达，对可达的事务进行截止期最早的优先调度，不仅考虑了事务的截止时间，还考虑了事务的执行时间，仍以截止时间为标准进行事务优先级的分配，是对第 2）种分配策略的一种改进，有效克服了上述策略的缺点。

(4) 空余时间最短最优先。该策略既考虑了事务的截止时间，又考虑了事务的运行时间。对于正在执行的事务，如果空余时间大于等于 0 则事务处理可在截止期时间前完成，在处理过程中空余时间不变，所以事务优先级也不变；如果空余时间小于 0 则事务处理已经或将会超过截止时间。对于未执行（被挂起）的事务，空余时间是逐渐减小的，故其优先级在增大。但要注意，该策略和第 3）种策略还是有差别的，它针对事务的挂起与执行的变化，确定其优先级的升降。

通过分析可以知道，优先级调度的指导思想都是通过一定的优先级分配策略使实时事务在事务调度队列中排队来等待调度。然而各种优先级分配策略各有优缺点，需要根据不同的应用环境以及应用需求选取合适的优先级分配策略。

2. 实时事务调度

1）静态表驱动调度

它执行静态可调度性分析，且产生一种调度（通常为一时刻表），在运行时用来决定一个

任务何时开始执行。这种方法适用于周期的任务(事务),是高可预报的,但也是高度不灵活的,因为任务及其特征必须事先完全确定,任何改变都可能要求对该表进行全面修改。

2) 优先级驱动可抢占调度

这种调度是在运行时,先按"最高优先级最先"原则指派事务执行,当后来在执行过程中有更高优先级的事务到达时,则抢占原任务的处理机,执行新来的高优先级事务,这有利于处理新到达的高优先级事务(如硬实时或时间紧迫的事务)。

3) 动态计划式调度

该方法实际上是第 1)种方法的动态执行,它同样进行可调度性分析,但在调度分析的时间上和第 1)种不同,第 1)种通常处理的是周期执行、预定义事务,因此它可以做好一个固定的表,而动态计划式调度在运行时对到达随机任务(事务)进行动态可调度性分析或可行性检验,仅当断定它是可执行的时候,才运行执行。

4) 动态尽力式调度

动态尽力式调度是当前许多实时系统中使用的方法。按照这种方法,基于任务(事务)的特征为每一个任务计算出一个优先级值,然后按其优先级来调度多任务。

3.4.4 基于功能替代的实时事务二次调度机制

近年来,实时事务处理技术不断得到发展和进步。随着嵌入式实时数据库对于高实时性和高可靠性的更高要求,基于功能替代的实时事务二次调度机制得以引入和广泛运用。下面对该机制做简要介绍。

1. 功能替代的事务模型的定义

为了便于说明功能替代的事务模型,先做出如下定义。

定义 3.10 一个具有时间限制的应用为一个实时事务,它由若干个任务组成,任务包含了一组功能等价的子事务,称为事务的功能替代集。在一个实时事务中,由每一任务中的一个成员所组成的集合称为该事务的一个功能替代集。

一个实时事务可以表示为一个四元组 $T::=(\mathrm{TS},R,C,<_t)$:

$$
\begin{aligned}
&T::=(\mathrm{TS},R,C,<_t)\\
&\mathrm{TS}::=\{<\mathrm{TK}>\}^*\\
&\mathrm{TK}::=\{<\mathrm{ST}_i \mid \forall i\neq j(\mathrm{ST}_i\leftrightarrow\mathrm{ST}_j),1\leqslant i,j\leqslant n\}^*\\
&\mathrm{ST}_i::=<T>\mid<\mathrm{OP}>(i=1,2,\cdots,n)\\
&\mathrm{OP}::=<\mathrm{OB-OP}>\mid<\mathrm{TM-TP}>\\
&R::=(<\mathrm{DR}>,<\mathrm{PR}>)\\
&<_t::=\text{时序}\\
&C::=\text{一组限制组合}
\end{aligned}
$$

其中,$\{\}^*$ 表示非空集合,$\leftrightarrow$表示"功能等价";ST 表示运算中间量;OB-OP 和 TM-TP 分别表示数据库的对象操作(如插入、删除等)和事务管理操作(BEGIN,COMMIT 等);DR 和 PR 分别表示一个数据资源集合和一个处理资源(如 CPU 时间、缓冲区等)集。

该定义表明,实时事务是一个四元组:任务集 TS、资源集 R、时序$<_t$ 和限制集 C。TS 中的每一个任务由一个非空有限的子任务集组成,其中的子任务或是一个事务,或一个基本

操作(包括对象操作和事务管理操作),它们称为原事务的子事务,所以事务是嵌套结构的,一个任务的子任务/子事务在功能上都是彼此等价的。

R 为执行该事务所需要的系统资源,它由定义在各功能替代集上的一组系统资源信息组成,包括所需要存取的数据(DR)和 CPU 时间以及缓冲区等(PR)。C 是该事务的一组约束,包括时间约束、一致性约束等,它由定义在各功能替代集上的一组约束组成,包括该事务所必须满足的 DR 中数据的完整性限制和 TS 中任务的定时限制。

$<_t$ 为任务集上的一种时序关系,与各功能替代集之间不存在直接的时序关系,$<_t$ 应由定义在各功能替代集上的一组时序组成。

功能替代集是实时事务的组成成员,它能完成所对应的实时事务的功能。在特殊情况下,当实时事务由一个功能替代集组成时,该功能替代集就是一个实时事务。可以说功能替代集也是实时事务,确切地说是实时事务的一个例子。功能替代集具备了实时事务的一些基本特性,如定时限制功能,同构(意味着将所有功能替代集中功能等价的子事务进行有条件的归类处理,能得到以任务为基本单位的实时事务层次结构)等。

功能替代集与所对应的实时事务之间的关系首先是对象与成员之间的关系,但当实时事务以任务为基本单位时,与功能替代集在结构上保持着同构,即实时事务的任务模型结构与功能替代集保持同构,如图 3-6 和图 3-7 描述它们的这种关系。其中,图 3-6 表示一个实时事务,图 3-7 表示该实时事务对应的功能替代集,即执行模型。其次,一个实时事务可以有多个功能替代集,在每一次执行过程中,只有一个功能替代集实施运行,如果由于某种原因此功能替代集夭折,而该实时事务的截止期未到,系统可以选择另外一个功能替代集投入运行,只要有一个功能替代集成功执行,则该实时事务就能提交,只有当所有的功能替代集都失败,才意味该实时事务夭折。

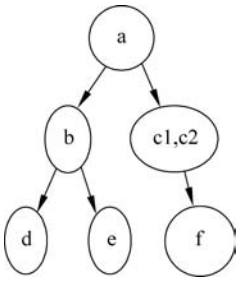


图 3-6　实时事务任务模型

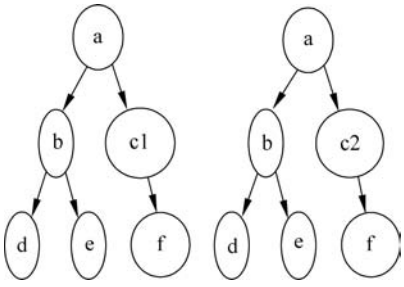


图 3-7　实时事务功能替代集

2. 预分析处理

由于实时事务的成功执行要满足特定的时限,系统通常不可能依次将每个功能替代集都执行一次直到实时事务提交,这样通常会超过截止期。理想的情形是系统能选择最适应运行环境的功能替代集实施运行,使事务的成功率达到最高。所以,事务的功能替代集需要进行预分析,为后续的调度提供依据。功能替代集的引入,使实时事务的调度分为两大步骤。第一步是在实时事务的诸多功能替代集之间进行分析调度,便于选取最佳功能替代集代表实时事务投入下一步调度;第二步是实时事务之间的调度。总的来说,对功能替代集的调度有利于选择最有可能满足各种约束、最有可能得到所需系统资源的功能替代集投入运行,从而提高实时事务的执行效率和成功率。

事务预分析处理的算法流程如下。

输入：实时事务。

输出：T 的一个替代。

步骤如下。

(1) 生成任务树 TK-树，并提取有关结构、行为、资源请求、时限的知识。

(2) 分解 TK-树，产生调度树；/ * 得到各替代 FXi * /。

(3) 进行各调度树的可调度性分析。

(4) 生成弱调度树和强调度树。

(5) 初始调度。

3. 二次调度思想

从调度的角度来说，当事务重启时，如果依然按原路径执行，那么导致其重启的原因可能并没有排除，势必造成再次无效重启。然而，如果该事务在重启时能选择替代集，就有可能避开障碍获得成功，提高该事务的成功率。对用户而言，系统透明地处理了失败的“执行”。

基于功能替代的实时事务调度是将传统的一次性调度分解为多重调度——内部调度和外部调度。内部调度是指在一个实时事务中选取若干功能替代集，剔除不可调度的功能替代集，它实际上包括预分析和预调度两部分。外部调度类似于传统的实时事务调度，若干实时事务竞争系统资源，如图 3-8 所示。当内部调度的结果为空集时，该实时事务不可调度，当事务执行失败时，不能立即夭折，必须重新转入内部调度。内部调度的作用是找出实时事务的可调度集，提高外部调度的效率。

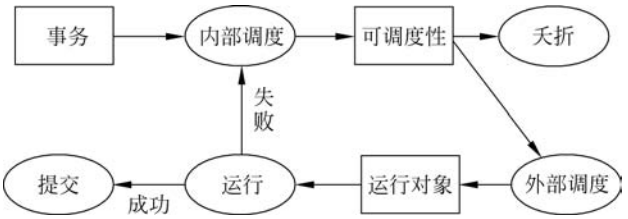


图 3-8　实时事务二次调度示意图

停止此调度活动的原因：事务的截止期到；由特殊操作强迫停止；该事务的所有功能替代集都夭折和有一个功能替代集成功执行并提交。

根据替代之间的性能差异，对该事务的所有可调度的替代做出调度，此调度有别于传统的实时事务调度，称为内部调度。内部调度的目的是在一个实时事务中选取最佳的替代，它实际上包括预分析和预调度两部分；外部调度就是传统的实时事务调度，针对多个实时事务，目的是在多个实时事务中分配系统资源(包括 CPU 数据对象等)。

实时事务及其调度关系如图 3-9 所示。

实时事务 T_1 由功能替代集 g_{11}，g_{12}，g_{13} 组成；实时事务 T_2 由功能替代集 g_{21}，g_{22} 组成。内部调度 SN1 针对实时事务 T_1，从中选取一个合适的对象 g_i，($i \in [1,3]$)，内部调度 SN2 针对实时事务 T_2，从中选取一个合适的对象 g_{2j} ($j \in [1,2]$)，而外部调度 SW1 针对 g_{1i} 和 g_{2j}，实际上是对内部调度结果进行二次调度，经过外部调度的实时事务将正式运行。

设有实时事务集合 $TT=\{T_i \mid 1 \leqslant i \leqslant m\}$，$T_j=\{fx_{ij} \mid 1 \leqslant j \leqslant D(GT_i)\}$，用 ES 和 IS 分

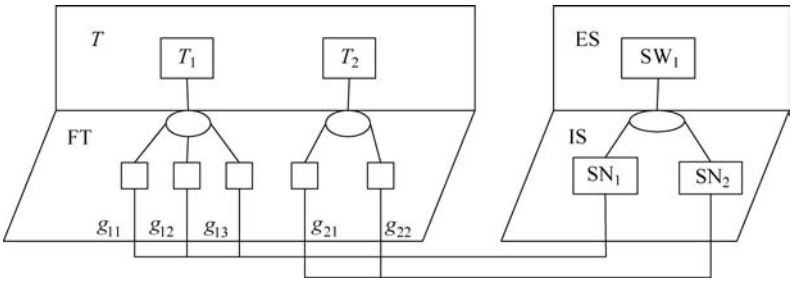


图 3-9　实时事务及其调度关系图

别表示内部调度和外部调度，则对于 T_i 的一个内部调度为

$$\mathrm{IS(TT)}=\mathrm{scheduling}(\mathrm{fx}_{iq1},\mathrm{fx}_{iq2},\mathrm{fx}_{iq3},\cdots,\mathrm{fx}_{iqp}),$$

$$\mathrm{fx}_{iq1}\in T_i,\quad 1\leqslant qp\leqslant D(\mathrm{GT}_i)$$

对应的一个外部调度为

$$\mathrm{ES(TT)}=\mathrm{scheduling}(\mathrm{IS}(T_1),\mathrm{IS}(T_2),\cdots,\mathrm{IS}(T_i),\cdots,\mathrm{IS}(T_m))$$

$$=(\mathrm{fx}_{a1},\mathrm{fx}_{a2},\mathrm{fx}_{a3},\cdots,\mathrm{fx}_{am})$$

内部调度与外部调度紧密相关，内部调度是基础，只有通过内部调度才能选出一个可调度的替代参与外部调度；外部调度是最终目的，只有通过外部调度，实时事务才能真正投入执行。

内部调度机制采用基于优先级的调度策略对事务的可调度集进行调度，相应的优先级分派策略可见 3.4.3 节所示的优先级策略。

3.5　实时并发控制

3.5.1　并发控制概述

并发控制隔离并发执行的事务，确保事务的执行不会相互干扰，保证数据的逻辑一致性。传统的数据库系统中，强调所有事务具有平等的地位和相同的调度机会。实现并发控制协议的技术包括两阶段锁、时间戳、多版本或者乐观的并发控制协议。每种机制基于不同的假设进行设计，但是有一个共同的目标——事务的可串行化。并发控制的基本思想是移出冲突，当一个冲突被检测，或者终止一个冲突的事务或者采用其他方法解决。并发控制协议能从 3 个不同的方面进行分类：冲突检测、冲突解决、串行化规则与顺序。

1. 冲突检测

检测冲突的方法主要有两种：悲观(Pessimistic)和乐观(Optimistic)。悲观的方法总是在存取数据时检测可能的冲突，相应的并发控制协议称为悲观并发控制(PCC)协议；而乐观的并发控制(OCC)允许事务访问所有的数据，只是在事务提交之前检测冲突。锁、时间戳和串行化图都能够用于事务间的冲突检测。

锁机制要求事务在存取数据对象之前必须获得相应的锁，基本的锁类型包括读锁(共享锁)与写锁(互斥锁)。锁机制通常应用于悲观的并发控制协议，也能够用于乐观并发控制协议中来指示事务所存取的数据对象，这种锁称为弱锁(Weak Lock)。

在时间戳(Timestamp)机制中,每个事务 T_i 执行前由系统分配一个时间戳 TS(T_i),如果有一个新的事务 T_j 进入系统,则 TS(T_i)<TS(T_j)。而每个数据对象 x 需要与两个时间戳相关联,即写时间戳 WTS(x)与读时间戳 RTS(x),写时间戳表示成功执行写 x 的所有事务的最大时间戳;读时间戳表示成功执行读 x 的所有事务的最大时间戳。每当有新的读写操作时,数据对象的时间戳就被更新。时间戳协议保证冲突可串行化且无死锁,但是可能产生不可恢复的调度。对协议进行扩展,可以保证调度可恢复且无级联夭折,如通过只在事务末尾执行写操作等方式。

串行化图(Serialization Graph)表示数据库中所有事务的执行历史,并发控制管理器能够利用环检测算法判断这个历史是不是可串行化的。例如,SGT(Serialization Graph Testing)协议就是通过这种方式进行冲突检测的。

每种冲突检测机制都存在一种悲观并发控制协议,如两阶段锁协议(Two-Phase Locking,2PL)、时间戳排序协议(Timestamp Ordering,TSO)与串行图测试协议(Serialization Graph Testing,SGT)。而另一方面,这些机制也都能用于乐观的并发控制中在事务验证阶段检测冲突。

2. 冲突解决

每当检测到一个冲突,必须采用一种冲突解决方法解决这个冲突。常用的方法包括:终止或者重启冲突中的一个事务,或者阻塞请求事务。如果冲突是在事务访问数据之前被检测到,阻塞与终止都是可选的方法。但是,当采用乐观并发控制协议,冲突是在事务访问数据之后才被检测到,只有终止冲突中的某个事务。其他可能的冲突解决方法是允许多版本(Multiversion)与重调整串行化顺序。

3. 串行化规则与顺序

事务的串行化可以基于开始时间、完成时间或者动态导出的顺序,如数据存取时间。

下面的方法经常用来确定串行化顺序。

事务开始时间或者预指定的时间戳:在每个事务执行之前分配一个时间戳,如果基于这个时间戳顺序事务不能通过验证,则被夭折。

事务完成时间:乐观并发控制采用这种方法,事务的验证基于这个时间戳顺序。有研究表明,事务的串行化顺序基于开始时间比基于完成时间可能导致更多的事务夭折。

元组存取顺序(Granule Access Order):两阶段锁采用这种机制,不过如果所有的锁在提交时释放,则串行化顺序与基于事务完成时间相同。

传统数据库强调对事务处理的响应时间和系统的吞吐率,但是实时数据库系统(RTDBS)却强调的是事务的时限和错失率。所以,实时数据库不能沿用传统数据库系统的并发控制协议,实时数据库为了使尽可能多的事务满足它们的截止期,应将事务的时间信息加入到调度中来,事务越紧急,应越早地投入运行。当一个事务正在运行时,如果新到了一个更为紧急的事务,该执行事务应该被抢占。实时数据库中事务的并发执行具有以下特点。

1) 并发执行的事务具有时间约束和依赖性

实时数据库系统中的事务具有显式的时间约束,此外,当它们并发执行时,事务之间可能存在与时间有关的依赖关系。比如,事务 T_1 必须在事务 T_2 开始执行之前(或提交前/后)开始提交;事务在运行过程中可能触发子事务,被触发事务与此触发事务并发执行

等等。

2）满足截止时间要求

在实时数据库中，按截止期特性将实时事务分为硬实时事务和软实时事务。系统必须保证硬实时事务的截止期，否则该事务的价值在截止期后为零甚至对系统产生危害；软实时事务超过截止期后虽然不会对系统产生危害，但该事务的价值会急剧下降，由此，并发控制在保证数据一致性的前提下，为了满足事务的截止期要求，应区别对待不同特性的实时事务。

3）系统资源和 CPU 控制权的抢占

一个紧急事务到达时，为了处理该紧急事务，系统会使正在执行（还未完成）的事务夭折，而去执行该紧急事务。

4）事务原子性

由于必须保证事务的原子性，被抢占的事务夭折重启。

5）时间和逻辑依赖

事务处理的正确性依赖于逻辑结果和逻辑结果产生的时间。

6）数据的实时性

系统宁愿要及时的部分正确的结果，也不要过时的精确的结果。在资源受限的实时数据库系统中，当系统超载时，必然存在某些事务不能在其截止期内完成，对于某些实时应用环境，截止期之后的精确结果是无效的，那么在截止期内，如果能得到近似结果也比没有结果要好。

7）数据一致性标准降低

传统的数据库一致性标准是可串行化，对于实时数据库系统而言，此标准过于严格，为了使尽可能多的事务满足截止期，需要降低正确性标准。评价并发控制协议性能的标准不是反应时间和系统吞吐率，而是满足事务截止期的比率，即：系统的成功率。

下面介绍实时数据库系统的常见的并发控制协议，并对协议的性能进行分析。

3.5.2　实时并发控制协议

实时数据库中的并发控制协议不仅要求保证数据的逻辑一致性，而且必须考虑满足并发事务的截止期。目前，许多面向实时数据库的并发控制协议已经被提出，主要包括基于锁的并发控制协议、乐观并发控制协议、多版本并发控制协议、推测并发控制协议等。

1. 基于锁的并发控制协议

保证可串行性的一种锁协议是两阶段锁（2PL）协议，该协议要求事务分为两个阶段提出加锁与解锁请求。一开始，事务处于增长阶段，事务能够根据需要获得锁。一旦事务释放了锁，它就进入缩减阶段，不能再申请任何新的锁。

对于任何事务，调度中该事务获得其最后加锁的时刻（增长阶段结束点）称为事务的封锁点。这样，多个事务可以根据它们的封锁点进行排序，这个顺序就是事务的一个可串行化顺序。

两阶段锁并不能保证不会发生死锁，并且级联回滚也可能发生。级联回滚可以通过修改两阶段锁为严格两阶段锁来避免，严格两阶段锁要求事务持有的所有排它锁必须在事务提交后方可释放。另一个两阶段锁的变体是强两阶段锁协议，它要求事务提交之前不能释

放任何锁。在强两阶段锁协议下,事务可以按照其提交的顺序串行化。大部分商用数据库系统或者采用严格两阶段锁,或者采用强两阶段锁。

传统的锁式协议像 2PL 对实时数据库是不够的,可能遇到的两个主要问题就是优先级反转和死锁。若请求者 T_j 的优先级比占有者 T_i 的优先级高,则按协议在解决冲突时发生高优先级事务等待低优先级事务完成的情况,这就是“优先级反转”。

这对实时数据库是不合乎要求的。因为它违背了高优先级事务在系统资源使用上要优先于低优先级事务的原则。为此,人们开发了许多策略来解决这些问题。最典型的有 2PL-HP、2PL-WP、2PL-CPI 与 2PL-CR、优先级顶协议、有序共享协议、RTL、基于资源预报的并发控制协议。

基于锁的协议需要考虑封锁粒度的影响。

封锁对象的大小称为封锁粒度(Granularity),封锁粒度与系统的并发度和并发控制的开销密切相关。直观地看,封锁的粒度越大,数据库所能够封锁的数据单元就越少,并发度就越小,系统开销就越小;反之,封锁的粒度越小,并发度较高,但系统开销也就越大。对于封锁粒度的大小要根据具体的应用系统确定。传统数据库获得锁的开销较小,因此通常选用小粒度封锁单位,以增加系统的并行性以及处理的吞吐量,但在冲突较低时,并发能力对吞吐量几乎没有影响。在嵌入式数据库中,事务获得锁的开销与处理数据的开销相当,过小的封锁粒度反而会降低系统的性能,如果将一个事务将细粒度锁换成粗粒度锁,反而会减少开销。并且,由于应用程序比较少,事务的并发度并不是很高。所以,为了在保证并发度的前提下减少事务封锁开销,通常在 EDBMS 中使用锁定整个关系的锁,甚至是整个数据库的锁(在这种情况下相当于事务串行执行,适合于一些特殊场合),而在冲突严重时,使用细粒度的元组锁。

2. 乐观并发控制协议

基于锁的并发控制属于悲观的方法,具有下面内在的缺点。

(1) 锁维护要求一定的系统开销。

(2) 总是假定事务冲突经常发生,而实际上锁只在最坏情形下才是必要的。

乐观并发控制基于相反的假设,事务冲突很少发生,因此允许事务无阻碍地执行直到全部操作完成,然后在提交时进行验证,如果通过了检验就提交,否则夭折。如果系统中事务之间的数据竞争很弱,大部分事务能够通过验证并提交;而如果事务之间的竞争越激烈,越多的事务就将被夭折并重启,从而降低系统资源利用率。

为此,乐观并发控制将事务 T_i 的执行分成以下 3 个阶段。

(1) 读阶段:T_i 要存取的数据都复制到它的工作区,所有的更改操作都在工作区针对这些副本数据进行,当 T_i 的计算全完成时,就进入下一阶段。

(2) 验证阶段:检查 T_i 的操作是否奇偶违反了可串行化的要求,若违反了,则终止事务 T_i,否则就提交并进入下一阶段。

(3) 写阶段:T_i 成功地通过了验证阶段,所有由它变更的数据从工作区中复制到数据库中。

为了验证事务,对每一活动的事务 T_i,要记录它的“读”和“写”数据集合:

$$RS[T_i]=\{x \mid R_i(x)\text{已为乐观并发所允许}\}$$

$$WS[T_i]=\{x \mid W_i(x)\text{已为乐观并发所允许}\}$$

乐观并发策略的关键是如何验证事务，下面为 3 种不同类型的验证方法。

1）OCC-FV

在乐观并发控制中，事务允许不受阻碍地执行，直到它们到达提交点再进行验证。这样，事务执行包括 3 个阶段：读、验证和写。验证基本上分为向后确认和向前确认。向后确认机制针对已提交事务，无法在串行化处理中考虑事务的优先级，不能应用到 RTDBS。向前确认针对当前运行的事务，正被确认的事务或者冲突的动态事务可能重启，所以它有利于 RTDBS。而且，向前机制通常比向后机制探测和解决冲突早，资源和时间浪费较少。这里把使用向前确认的乐观算法称为 OCC-FV。

OCC-FV 不使用事务优先级解决数据冲突。高优先级事务由于低优先级事务提交可能需要重启。已有几种用优先级等待或夭折机制将优先级信息加到 OCC-FV 中的方法。

2）OPT-SACRIFICE

OPT-SACRIFICE 使用优先级驱动夭折来解决冲突。某事务到达确认阶段时，如果一个或多个冲突事务比该事务优先级高，它就夭折；否则，它就提交并且所有冲突的事务立即重启。此机制采用向后确认，在确认事务大部分执行后可以排除夭折，因为冲突的高优先级事务仍然在读阶段。这种机制的问题是无效牺牲：一个事务为之牺牲的是后来被删除的事务。

3）OPT-WAIT

在 OPT-WAIT 机制中，当一个事务到达确认阶段时，如果其优先级在所有的冲突事务中不是最高的，它就等待具有较高优先级的事务完成。这一协议使较高优先级的事务首先满足截止期并且没有无效重启。

实时数据库系统要求具有较高的成功率，并发控制协议致力于将超过截止期的事务减至最少。乐观并发控制和锁式悲观并发控制存在下面两方面问题。

（1）当系统超载时，超过截止期的事务数量又大幅度上升，如果这些事务一开始就不被系统接纳，系统就不会在这些事务上浪费资源，性能会更好。Bestavros 提出了一种可接纳的事务机制对事务进行管理。

（2）乐观的并发控制和锁式并发控制性能差异很大，乐观并发在数据竞争缓和的情况下，协议性能较好。在竞争激烈的情况下，协议性能较差；锁式协议的性能正好相反。两种协议在不同负载条件下都表现出稳定性较差的特点。

3. 多版本并发控制协议

多版本并发控制协议主要包括以下两种。

（1）Multiversion Timestamp Ordering（多版本时间戳 MVTO 协议），该算法能够减少拒绝操作，提高并发度，同时能够保证所有事务读关系的正确性，从而保证其调度是可串行化的。缺点是频繁地拒绝到达时间比较晚的写操作，同时还存在级联夭折。

（2）Two Version-PCP 协议采用两版本方法扩展了 RW-PCP 算法，简称 2VPCP。该算法无死锁，单阻塞，支持可调度性分析，减少了高优先级事务的阻塞时间，可以提供更好的可调度条件。

4. 推测并发控制协议

推测并发控制协议主要包括以下两种。

(1) Speculative Concurrency Control(推测并发控制协议,SCC)是一种更适合实时数据库管理系统的并发控制算法。该算法依靠冗余计算来尽早寻求可串行化的调度,可以减轻悲观协议中的阻塞问题和乐观协议中的重启问题,从而更好地满足事务截止期。

(2) Alternative Version Concurrency Control(替代版本并发控制,AVCC)协议很好地解决了浪费的重启问题和浪费的执行问题,同时通过事务多版本来提高事务满足截止期的机会。缺点是维护事务多版本需要系统提供足够的资源。

3.6 数据库恢复和备份

嵌入式数据库通常运行在嵌入式操作系统和嵌入式硬件平台上,不管是软件还是硬件平台故障,或者是来自外界的物理威胁,如洪水、泥石流等自然灾害,都有可能导致嵌入式数据库管理系统发生故障,造成数据库信息的丢失,造成不可挽回的损失。因此,采用某种方式来保证嵌入式数据库中数据的可靠性就显得非常重要。

应对数据库故障的常用方法是在系统运行时收集冗余数据,一旦发生故障,利用已有的数据将数据库恢复至正确状态。虽然恢复原理简单,然而实现技术比想象的要复杂许多。恢复子系统是数据库管理系统非常重要的组成部分,而且体积相当庞大,恢复子模块代码量常常占一个完整代码的百分之十几。现行商用数据库管理系统如 Oracle、DB2、SQL Sever 等均有完备齐全的恢复和备份方式,嵌入式数据库 BDB、SQLite、Oracle Lite 等也提供一些恢复方法。

3.6.1 数据复制及备份

1. 数据复制

数据复制就是通过在一个或多个物理设备上保存数据库副本的方式来保证拷贝数据库与源数据库相一致,以此来提高数据库中信息的安全性和可用性。采用数据复制技术可以保证当一个物理设备出现故障时,可以由其他设备上的数据库副本继续为用户提供服务。因此,数据复制技术提供了一种有效的容错保护方式。

数据复制技术采用在多个物理设备上建立数据的副本,每个副本即为数据的镜像,每个物理设备上的镜像保存数据库服务器中全部或部分数据副本。当服务器中的数据进行更新时,采用同步或异步的方式对其他站点的数据进行更新和保证数据的同步。数据复制可以通过网络将服务器中的全部或部分数据传送到其他站点,保证同步更新。

根据主数据库和备份数据库更新的时间限制,数据复制可以有两种方式,一种是同步方式,另一种是异步方式。同步数据复制方式是指本地和异地数据库以完全同步的方式进行更新,即当本地更新时,立即更新异地数据库。异步复制则是指当本地数据改变时,并不立即更新异地数据,可以以固定的时间间隔对异地数据进行更新操作。虽然同步复制方式可以保证主数据库和备份数据库之间实时更新,但同步复制方式对带宽提出了更高的要求。异步复制不要求本地数据和异地数据的更新完全同步,且受网络带宽影响较小,但这种方式下,本地和异地数据的更新存在延迟,常用在对实时性要求不高的场合。

2. 数据备份

数据备份是指在某一个时刻存在一个异地数据库,保存和本地数据库相同的数据副本。

当发生存储介质故障时，这种方式可以对数据库进行恢复，是保障数据库安全的一个有效手段。

数据备份同样可以有同步备份和异步备份两种方式。同步备份是指当本地数据库数据发生更改时，立即对异地数据库进行更新。异步备份是指本地和异地数据库的更新不是同时进行的，可以选择一个时间间隔定期对本地数据库进行备份。

根据备份时是备份所有数据库文件还是只备份数据库的改变，数据库备份又有以下两种备份方式。

1）完全备份

完全备份方式要求备份所有的数据库文件。随着数据库使用时间的不断增加，数据文件或关键字文件会一直增大，采用完全备份时需要全部备份这些文件。因此，当数据库文件很大时，完全备份方式的备份效率会比较低。

2）增量备份

增量备份就是每次只备份数据库文件改变的部分，但前提是第一次备份时同样需要将所有的数据库文件备份过去，第一次进行的备份称为基准备份。由于增量备份方式是每次只备份数据库的变化部分，因此，备份效率相对完全备份方式会比较高。

3.6.2　嵌入式数据库备份

数据库备份技术采用某种方式把全部或部分数据库信息备份到其他物理设备上，采用同步或异步的方式更新备份数据库，以确保主数据库与备份数据库相一致。当运行主数据库的物理设备出现故障时，可以由镜像数据库来保障数据库的可靠性，对主数据库进行恢复或提供支持。即使数据库没有出现故障，也可以通过镜像功能来实现多用户对数据库的并发访问，用户不必等待其他用户对数据库释放锁。因此，采用数据库镜像技术对数据库提供一个备份副本是保证数据库可靠性的一个有效措施。

下面给出一种嵌入式数据库的备份方案。

该数据库备份支持数据库的本地备份和异地备份。在实现备份时要求可以以同步和异步两种方式进行；支持主数据库和备份数据库的跨平台备份；支持数据库的远程复制。

图 3-10 所示为典型的嵌入式数据库备份系统结构。

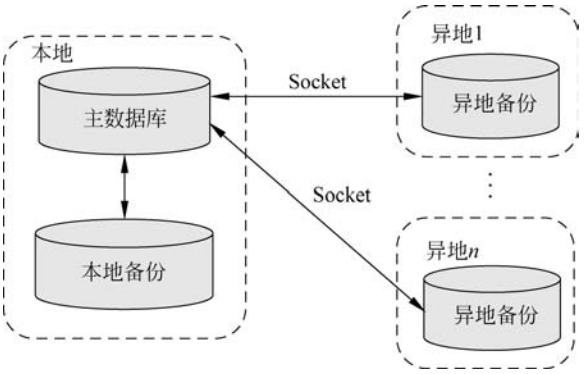


图 3-10　嵌入式数据库备份系统结构

当本地主数据库开始运行时,用户可根据需要选择是否开启异地镜像数据库系统。在开启异地镜像数据库系统时,本地数据库中包含的文件用 Socket 传输到异地相应的镜像路径下。其中,本地数据库和异地数据库文件存放路径可以通过配置文件进行设置。

本地数据库和异地数据库的更新采用事务和日志相结合的方式。本地数据库的更新均在事务内进行操作,异地数据库根据事务文件和日志文件对异地数据库进行更新。

异地数据库可以采用同步或异步的方式进行数据库的更新,从而实现数据库在异地的备份。采用同步还是异步的方式更新异地数据库,用户可以根据调用不同的接口函数进行选择。

镜像文件和镜像事务日志文件是进行备份数据库更新时必需的两个重要文件。当进行备份数据库文件更新时,将镜像文件和镜像事务日志文件通过 Socket 传输到备份数据库目录下,备份数据库根据这两个文件进行备份数据库的更新,以此来保证主数据库和备份数据库的一致性。其中,镜像文件会在备份数据库目录下一直存在,镜像事务日志文件为一个临时性文件,在对备份数据库进行更新结束之后会自动销毁。

为保证主数据库更新的可靠性,对主数据库的更新在事务内进行。在镜像数据库初始化和启动之后,应用程序通过事务对主数据库数据进行操作,在结束事务时,对镜像事务日志文件进行更新,将所有修改过的页面写入镜像日志文件中,同时,在镜像文件中创建一条事务记录。在事务结束时,主数据库系统端通过 Socket 向异地镜像数据库发送一个标志位,当异地镜像数据库收到触发信号后,接收相关文件,对镜像数据库进行更新,确保事务的变化应用于主数据库和镜像数据库中。

3.6.3 嵌入式数据库恢复

数据库恢复技术是每一个数据库系统必须具备的基本特性。由于嵌入式数据库运行环境的复杂性,其对数据库的可靠性提出了更高的要求。在嵌入式数据库中,最常见的数据库的恢复过程依赖于事务操作,通过事务操作保证事务执行前后数据库始终处于一致性状态。

嵌入式数据库与通用数据库一样,会面临事务故障、系统故障和介质故障。恢复子系统必须能完全处理这 3 类故障,保证事务的 ACID 特性和系统可靠性。恢复子系统要保证恢复时的效率,快速地从不一致状态恢复,尽快地响应用户请求,提高系统可用性以及数据库的数据安全。

每个数据库管理系统(DBMS)都有自己的恢复管理器,其工作是合理利用冗余,消除失败事务带来的影响。系统需正确应对以下 3 类故障。

事务故障是指事务没有运行到事务预期的终点而中止。引起事务故障的原因通常有删除操作、违反某些完整性约束、发生死锁而被系统选中进行撤销等。由于异常中止,事务可能使数据库处于不正确状态,且事务故障通常是非预期的,不能由应用程序处理。对于事务故障,处理的方法是进行撤销(Undo)操作,以消除失败事务对数据库的影响。

系统故障对于嵌入式数据库来说,可能发生得最为频繁,因为嵌入式设备可能需要经常进行关闭或重启,因此恢复子系统应能够正确地处理系统故障。系统故障是指使数据库系统无法继续正常运行而必须重启的任何事件,如硬件错误、操作系统故障、DBMS 代码错误、掉电等。由于写磁盘耗时且严重影响性能,DBMS 采用的写盘策略通常是无法预料的。系统故障发生时,已提交的事务所做的更改可能只有部分写入物理数据库中,甚至全部未写回,未提交的事务可能有部分修改已写回磁盘。系统重启后,需自动进行故障恢复处理,将

故障发生时按照所有事务是否提交分为已提交事务集合和中止事务集合。对已提交事务集合进行重做并再次提交；对中止事务集合进行回滚操作，撤销事务的影响，最终将数据库恢复到一致性状态。

介质故障是指外存被损坏而导致数据库文件不再可用，如磁头损坏、磁道损毁、瞬时强磁场干扰、磁介质磁性减弱等。对于嵌入式环境，最易发生的可能是设备的 Flash 损毁、SD 卡失效等故障。然而介质故障一旦发生，带来的破坏性是巨大的。应对介质故障，最有效的方法是定期做备份。故障发生后，在系统管理员的参与下，利用最新备份和活动日志或归档日志，可正确恢复数据库。

嵌入式数据库恢复子系统对性能要求较通用数据库更高，因为嵌入式设备通常要反复重启，且无专一管理员对其进行监管。总体来说，事务恢复通常在毫秒级内完成，系统恢复在秒级完成，而介质故障恢复通常在分钟甚至小时级别内完成，嵌入式数据库处理故障的性能应尽量高效。不同的故障，采取的恢复手段不同，对系统的性能影响也不同。

事务故障发生在事务执行过程中，此时事务的所有信息保存在内存中，Undo 信息或多或少也保存在系统缓冲区中，因此故障恢复时应充分利用访存的速度，尽量避免 Undo 信息在执行过程中刷到外存，以保证故障在毫秒级别完成。

系统故障发生后，内存中所有的数据库信息都丢失，因此必须借助已有的数据库信息，从设备上读取数据重建故障时的数据库状态，此阶段 I/O 的数量是决定恢复速度的关键。因为嵌入式数据库系统通常对内存需求有严格的限制，加之系统故障由于设备的经常置位复位而频繁发生，因此系统故障恢复时，性能是一个核心因素。恢复子系统应该采用合理的算法，以保证系统故障在秒级别完成。

介质故障发生后，需依靠已有数据库副本和日志信息，重建数据库，此阶段依赖于副本的备份频繁程度。由于嵌入式数据库大多应用在移动设备和终端上，且一般嵌入在具体的应用程序中，用户甚至感觉不到它的存在，因此备份方式不易采用过于复杂的方式。数据文件和副本的安全必须要得到保证，备份时尽量避免采用通用数据库的明文方式存放备份信息，减少信息泄露带来的风险。

根据数据库更新策略，通常将恢复技术分为影子页技术和基于日志的恢复技术。更新时前者保存数据页的两个版本以提供冗余信息，而后者通常采用原地更新时记录的冗余信息以提供恢复时所需的信息。

1. 基于影子页的恢复技术

在数据库管理系统中，事务管理模块和恢复管理模块提供对事务原子性和持久性实现的支持，影子页技术是实现事务原子性和持久性的方案之一。影子页技术不采用原地更新策略，它把更新页写入非易失性介质的另一个位置，当数据库在下一次检查点前发生故障时，使用数据库的旧版本恢复数据库，原理如图 3-11 所示。

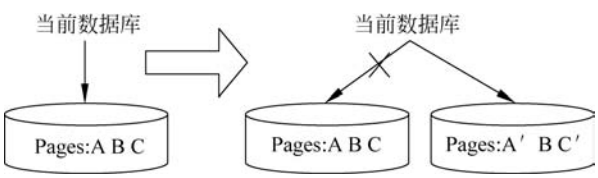


图 3-11　影子页恢复原理

当事务要更新数据库时,首先创建数据库的一个完整拷贝,所有更新都在新拷贝上进行,原数据库不发生任何更改。如果数据库管理系统发生事务中止,则新拷贝被简单地删除,原数据库不受失败事务的影响;当事务成功完成,则进行提交操作,首先确保数据库新拷贝都写到磁盘,然后数据库当前版本指向数据库新拷贝,之前的副本被删除,数据库完成从一个一致性状态到另一个一致性状态的转换。

显然,这个恢复技术原理较简单,实现难度不大,但是粒度较大,实际实现时常采用页面。它最明显的缺点是操作的粒度较大,在高并发系统中不太适用,因为这些系统常常要求粒度更细的访问控制来提高事务的并发处理能力。

2. 基于日志的恢复技术

一个基于日志的恢复子系统中,日志、检验点、重装是提供恢复能力而必须进行的准备工作。

1) 日志

日志文件在数据库恢复中起着非常重要的作用,可以用来进行事务故障恢复和系统故障恢复。日志数据保存数据库的冗余信息,以一种安全的方式记录数据库的更新历史。日志缓冲区专门存放数据库操作的记录,传统的数据库日志记录包括记录名、更新之前记录的旧值、更新之后记录的新值、事务标识、操作类型等。在嵌入式实时数据库系统中,为了减少系统的开销,在日志记录中不包括新旧记录值,对日志记录的写操作只对缓冲区进行,当缓冲区满时,才由磁盘写操作写入日志文件当中。

根据记录的对象不同,日志可分为物理日志和逻辑日志。物理日志记录数据库数据的物理变动,粒度可大至页面,小至字节。逻辑日志关注事务的每一步操作和修改,记录其操作和参数,面向更高层次。

根据数据库修改前后的状态和转换,日志又可分为状态日志和转换日志。状态日志分为前映像和后映像,当需要撤销事务时,利用前映像状态日志;当事务需要重做时,利用后映像状态日志。转换日志记录数据的前后差异,它是利用位的异或达到数据状态转换的目的,当需要数据的最新状态时,利用当前数据异或转换日志得到,反之亦然。

组合以上两种日志,可产生物理状态日志、物理转换日志和逻辑转换日志,然而状态日志和逻辑日志不能进行组合,因为后续状态是逻辑日志执行后才能得到的状态。

在数据库中,根据日志在恢复时的功能,日志又可以分为以下的 3 种方式。

Undo 日志系统:这种方式实现的日志简单,在执行回滚操作时会把前面提交的全部数据丢掉。

Redo 日志系统:这种日志在执行事务的回滚时,会把全部的事务都重新做一遍。这种方式实现的日志,开销比较大。

Undo/Redo 日志系统:以这种方式实现的日志会在日志系统中管理许多关于执行动作的信息,显得非常方便。

对于第一种日志来说,在数据库系统还没有把全部事务修改过的数据回写永久存储介质之前不允许再提交同样的事务;一旦执行了提交操作,系统会马上存入到永久性存储设备中,这样有时会出现读写的瓶颈效应;但是从实现角度来说,简单、安全、传统的日志方式即是这样设计的。

对于第二种日志来说,所设计的数据库系统允许很多数据库对存放在内存中的数据进

行改变，这种设计方式虽然减少了对外设的操作，但是当数据库系统出现异常情况而重启后，会执行大量的恢复工作，很烦琐。

对于第三种日志来说，它是为事务能够交叉操作的系统设计的，这种方式的日志系统，会存储非常多的日志信息，处理起来也十分繁复。

2）检验点

快速的恢复高度依赖于检验点的存在。检验点一方面减少了在数据库系统崩溃后系统重启时所要做的恢复工作量。另一方面，检验点也会影响数据库的处理性能，即数据库系统不希望在检验点执行的时候，停止其他事务的执行。

目前，在嵌入式内存数据库系统中，多采用一种基于模糊检验点的检验点策略。通过一个 last_checkpoint 来记录系统故障时所必须要进行恢复的日志起点，从而不需要从后开始扫描日志以找出检验点记录。在系统中，检验点作为最低优先级运行，从而不会影响事务的运行；检验点过程循环进行，即下一个检验点过程开始于上一个检验点过程结束。

3）重装

系统故障后重装必须保证将数据库恢复到某一个一致性状态。在系统故障时，造成数据库不一致状态的原因只能是已提交的事务对数据库的更新没有写入到外存数据库。因此，在系统故障发生的时候，用户只需要对重新修改了起始位置的日志进行重装操作即可。

3. 恢复策略

数据库发生故障后，恢复时采用的技术与系统本身采用的策略相关，这些策略包括：数据库采用何种更新策略、缓冲区管理策略和系统采用何种检查点。

1）更新策略

数据库版本更新策略分为原子方式（ATOMIC）和非原子方式（NO-ATOMIC）。原子方式要求数据库的脏页集要么全部写入非易失性储存器，要么全部不写入，保证数据库的状态时刻一致，即便发生系统故障也是如此；非原子方式则不要求所有脏页集一次全部写入，页面管理方式更灵活，它采用原地更新方式将页写入磁盘，由于脏页写入时机不一致，因此该策略应对系统故障时会更复杂。

2）缓冲区管理策略

按照页面替换时机可将页面管理策略分为窃取（STEAL）和非窃取（NO-STEAL）方式。采用窃取方式时，事务的脏页可在事务执行的任意时刻写入磁盘，腾出页面缓冲空间供其他页面使用；非窃取方式管理页面非常严格，要求一个事务的所有被修改的页在提交前都存在于数据缓冲区中。采用非窃取方式的优势在于不需要记录日志，但它对缓冲区的尺寸需求是不可预见的；使用窃取方式管理页面非常方便，然而需记录日志应对事务故障和系统故障，采用的恢复技术较复杂。

按照事务完成（End of Transaction，EOT）时的处理方式可将页面管理策略分为强制（FORCE）和非强制（NON-FORCE）方式。强制方式在事务提交时，要求该事务的脏页集必须写入磁盘，对于支持归档日志的系统需记录 Redo 日志，对于无归档要求的系统不用记录，记录 Redo 日志的主要目的是进行全局恢复。非强制方式在事务结束时不触发脏页写盘动作，何时写入没有具体要求，可延迟，因此必须记录 Redo 日志，以确保事务的持久性得到保证。

3) 检查点种类

为减少系统恢复时扫描的日志量,各类数据库管理系统通常使用检查点技术,检查点分为以下 4 种。

(1) 面向事务的检查点(Transaction-oriented Checkpoint,TOC),实际由前面介绍的缓冲区管理策略强制规则所隐含,两者联系紧密。TOC 检查点在事务结束标记 EOT 写入日志文件之前,要求所有脏页被写入非易失性介质。使用 TOC 检查点来减少恢复时的工作量是通过在正常事务处理时施加一些处理开销实现的。该方法的最大缺陷在于,被频繁修改的页在多个事务中均被反复写入,这些多余的写操作与数据缓冲大小有着紧密的联系,且数据页驻留缓冲区越久,它被多个事务修改的可能性就越大。因此,现行的数据库管理系统通常不采用该类检查点,事务提交一次就做一次检查点开销大,对并发度要求较高的数据库系统而言是不可接受的。

(2) 事务一致性(Transaction Consistency Checkpoint,TCC)检查点,可在系统事务执行过程中设置,待系统中无写事务时,触发检查点动作。TCC 实现简单,然而它最大的缺点是需等待系统无活动写事务时才能进行。在事务频繁的系统中,可能导致很长时间无法进行检查点操作,且一旦开始进行检查点,系统无法响应用户新的请求,不能执行新的写事务,对大型的数据库管理系统是不可取的。

(3) 动作一致性(Action Consistency Checkpoint,ACC)检查点。它的原理基于 TCC 检查点,两者间的差别在于划分一致性的粒度不同,TCC 的一致性面向事务,ACC 的一致性面向记录。ACC 把事务看作一系列对数据库进行更新的 DML 操作,检查点设置的时机和 TCC 一样,可在事务处理的任意时刻进行。实际进行检查点操作时,相比 TCC 其粒度更细,不是位于事务之间,而是位于 DML 操作之间。ACC 检查点缩短了事务响应的延迟,然而对于数据库缓冲区很大的系统,由于检查点期间刷脏页耗时较长,仍然会牺牲系统很大性能。

(4) 模糊(Fuzzy)检查点。它要求检查点时,所有在日志缓冲区中的日志均写入日志文件,不要求数据缓冲区中的数据页都写入外存。Fuzzy 只关心日志是否写盘,数据页有可能只有部分甚至全部未写入物理数据库,因此数据库在检查点后所处的状态是“模糊”的,是否一致无法确定。Fuzzy 检查点即便应对缓冲区很大的系统,由于顺序写日志往往可在数次内完成,所以对系统性能影响较小,这是因为 Fuzzy 检查点不关心数据页的写盘操作,因此它又被称为间接检查点。

在设计数据库恢复功能时,可采用更新策略、缓冲区管理策略和检查点策略这 3 种策略的组合,来选择自己想要的架构,不同的结构对应不同的恢复方法。例如,Oracle 采用 No-ATOMIC+STEAL+No-FORCE+ACC 方式,Berkeley DB 本质上也是采用的这种方式,而 System R 则采用 ATOMIC+STEAL+No-FORCE+ACC 方式等。但并不是以上所有的组合都可行,它们之间存在某种约束,如当检查点选 TOC 时,缓冲区管理策略只能使用强制规则。

4. 一个嵌入式数据库的恢复例子

本节介绍一个嵌入式数据库的恢复例子。该数据库系统的恢复子系统的结构如图 3-12 所示。

恢复子系统分为四层:用户层、管理层、缓冲层和文件层。用户层是指提供给应用程序

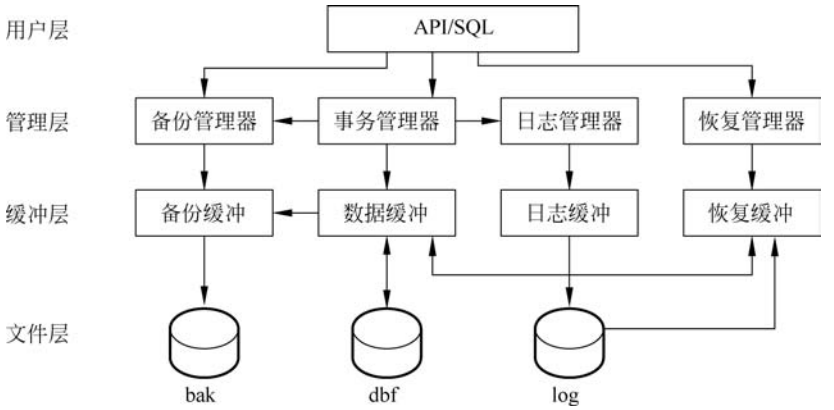


图 3-12　恢复子系统模块结构图

编程的接口 API 或通过终端键入的 SQL 语句。

事务由用户层请求触发，系统运行时，事务管理器管理整个系统的事务，来保证事务的 ACID 特性。事务处理时，会对数据库页面进行更改，当页面不在数据缓冲中时，它负责通知数据缓冲进行页面替换的相关工作。事务修改数据之前，通知日志管理器进行日志记录工作，做到先记日志，后修改数据。日志管理器负责 Redo 日志的生成，当物理事务结束时将收集的日志写入日志缓冲区中，若缓冲区已满时进行刷日志操作。事务若成功提交，将该事务的所有日志均写入磁盘；若事务执行时发生故障，利用该事务的回滚段（在数据缓冲或 dbf 文件中），进行事务故障恢复。

当指定时间点到来或用户发出备份指令时，备份管理器启动备份线程。待数据库中无写事务时，事务管理器通知备份管理器进行备份。备份管理器借助备份缓冲区进行库备份，数据从数据缓冲中获取。备份管理器对数据页面进行过滤和加密后，写入 bak 备份文件中。

系统每次启动时，自动进行系统故障恢复工作。恢复管理器检测 log 日志文件中记录的最近一次检查点信息，确定是否有 Redo 日志供重做。若系统上次是正常关闭，不进行后续恢复工作而直接正常启动系统，供用户使用；若系统上次是非正常关闭，则需要进行系统恢复工作。恢复管理器首先确定有效 Redo 日志范围，之后启动分发线程和多个可配置重做线程进行 Redo 重做，接下来恢复管理器检测回滚段是否处于使用状态，若是则通知事务管理器启动一个新的事务并把该回滚段赋予新事务，交由事务管理器进行事务故障处理工作，待以上操作完成后，系统做一次检查点操作，确立一个数据库一致状态。

5. 主流数据库采用的恢复方案

本节介绍主流数据库的恢复策路，包括嵌入式产品 Berkeley DB、SQLite 和通用产品 SQL Server。

1）Berkeley DB

Berkeley DB 是美国的公司开发的开源的嵌入式数据库程序库，它是一个经典的 C-Library 库。Berkeley DB 支持关系和非关系型数据，是一个高性能的、可伸缩的、模块可插拔的数据库，为多种编程语言提供应用程序接口。

Berkeley DB 提交事务或回滚事务时使用先写日志规则，采用前后映像日志技术处理

应用程序故障、系统故障和磁介质故障。Berkeley DB 支持内存数据库和磁盘数据库两种模式,当工作在内存数据库模式时,要求事务执行期间产生的日志不能超过系统的默认值1MB;当工作在磁盘数据库模式时,事务的日志缓冲区若溢出,则日志会被刷到日志文件中。由于 Berkeley DB 支持事务的高并发,因此日志采用文件组形式,当前活动的日志文件仅有一个。

Berkeley DB 采用检查点定期将脏页写到磁盘中,以减少故障恢复时的工作量。当发生应用程序或操作系统崩溃时,系统恢复时需要往前回滚两个检查点的日志,仅回滚一个检查点是不可取的,因为恢复系统不能确定最近的一次检查点是否正确完成,由此可知,Berkeley DB 采用的是动作一致性检查点。在恢复时,Berkeley DB 可精确地恢复到某个时刻,支持基于时间点的恢复,但是操作起来较复杂,需要数据库管理员事先分析日志,并确定将要恢复到的日志的 LSN。

Berkeley DB 支持非关系型数据,但它不理解数据库模式的概念,因此备份采用的是物理备份方式。

2) SQLite 恢复策略

SQLite 恢复机制基于日志,系统不采用 Redo 日志,不使用检查点,仅 Undo 采用日志。SQLite 中,当事务的锁类型达到 Pending 时,为该事务创建一个回滚日志文件。日志采用影子页技术,在对数据库页面进行任何更改之前,将该页所有数据及一些状态控制信息一并刷入磁盘中,之后该事务的每次对页面的修改均采用此策略。在事务提交前,事务操作过程中所有涉及被修改的页的前映像均存在 Undo 文件中;事务提交时只需先刷所有脏页到磁盘,然后删除回滚日志文件即可。

若在事务执行过程中发生事务故障,则直接从 Undo 日志文件中读取页面数据且覆盖到对应页面即可;在未成功删除日志文件之前出现的系统故障,如应用程序崩溃、系统宕机等,均认为事务未成功执行,下次系统启动后,用 Undo 日志文件中的页数据直接覆盖对应页,便可恢复数据库状态到此次事务执行之前,来保证数据的一致性。

SQLite 日志系统逻辑简单,发生故障时恢复算法也较简单,且易于控制。在一次事务中,若对少数某几个页面进行频繁的修改,则只需对涉及的所有页面记录一次信息。由于采用页面级记录旧映像的策略,因此日志文件增长速度会非常快;恢复系统仅采用 Undo 日志形式,因此在事务过程中可能需要进行反复的刷盘操作。

SQLite 嵌入式数据库支持备份,备份期间对库加共享锁,备份实际采用 UNIX 系统的 cp 命令或者 Windows 系统的 copy 工具来实现,备份完成后,释放共享锁。

3) SQL Server 恢复策略

SQL Server 使用事务日志、检查点、磁盘镜像等技术手段进行故障恢复。事实上,SQL Server 事务日志也有 Undo 和 Redo 之分。SQL Server 服务器为每个数据库创建一个 SYSLOGS 系统表,事务对各个数据库的修改记录都将保存在这个表中,当发生故障时系统利用该系统表数据进行修复。

SQL Server 服务器启动时,会自动执行恢复进程。它先为每个数据库搜索收集其事务日志,再搜索每个数据库的 SYSLOGS 表来决定哪些事务应该进行 Redo 操作或者回滚操作。对于已提交事务而更新未反映到数据库中的事务进行重做,对于未提交事务,则进行回滚操作,以撤销事务对数据库的影响。完成这些操作后,在 SYSLOGS 表中记下一个标志,

以标识当前数据库处于一致性状态。

SQL Server 有两种类型的检查点：固定时间自动执行的检查点和利用 CHECKPOINT 命令设置的检查点。检查点时，SQL Server 首先暂停数据库服务器中所有的事务，然后将事务日志和缓冲区中所有脏页先后分别写盘，随后在日志中记录一个检查点操作，最后恢复事先暂停的事务。

当数据库的物理介质发生破坏时，需要后备副本的支持。SQL Server 提供脱机全备份、联机增量备份、差异备份、数据库文件和文件组备份等多种方式，以保证即使在物理介质损坏的情况下，也可恢复数据。

从以上分析可见，不同数据库产品，采用的恢复技术不同。基于日志的恢复技术被广泛地使用在各类数据系统中，不论是企业级数据库还是嵌入式数据库。

3.7　系统可定制技术

当数据库系统和应用是分离时，即使应用所需的功能很少，数据库系统仍然包括了所有支持的功能模块，对于硬件资源非常有限的手持设备和智能电器来说，这无疑是不合理的。

现有的数据库管理系统通常采用固定的结构体系，系统一旦形成后，很难改变它的数据库特点或对它进行功能修改和扩充，否则就需要对底层的结构和实现作很大修改，按这种方式形成的系统灵活性很差，资源不能重复利用。嵌入式系统内存容量少，CPU 处理能力低，而嵌入式应用实现的功能往往比较单一和具体，对数据的处理和维护能力要求不高，在这种情形下，一个功能完备的数据库管理系统不仅没有必要，而且会使本来就有限的内存资源更紧张。较理想的嵌入式 DBMS 可根据实际应用的需要只包含所需要的数据库功能，这样不仅能达到管理数据的目的，还可以最大限度地利用系统资源。

图 3-13 显示了一种三层结构的嵌入式数据库设计模型。

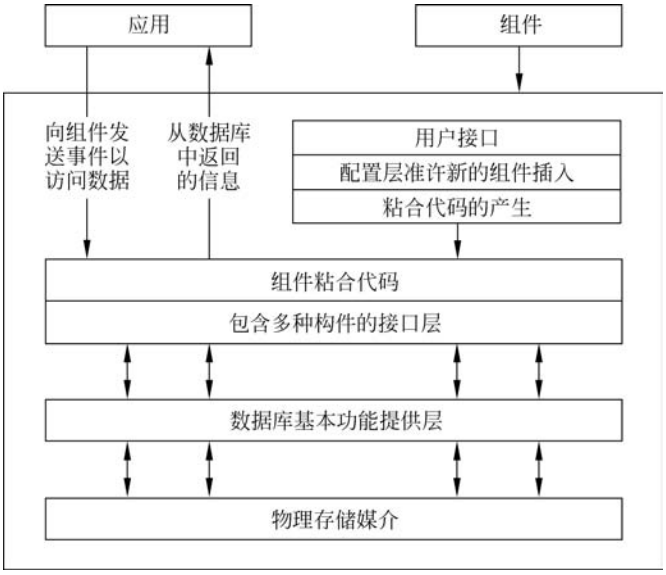


图 3-13　基于组件结构的嵌入式数据库模型图

第一层是数据库的底层,它将提供最接近于底层硬件的数据库服务。这一层提供基本的数据库功能,包含了数据对象的格式、存储以及如何检索这些对象的方法。该层隐藏了用户的硬件部分相关的细节,同时这一层自身也会被作为组件的一部分。这一层通过开放的接口向上一层提供服务,所有的新功能如硬件依赖方法可以通过这层被添加到接口上。

第二层是数据库的中间层,也属于接口层,它为组件机制的运行提供支撑,它们需要被植入系统中以提供特别的服务。一般来说,数据库要能支持传统数据库已有的最基本的服务功能,这些基本的服务将由单独的组件提供,这些组件提供了事务处理、位置依赖查询处理、缓存管理、断开连接操作处理以及协作控制机制。采用这种设计方式的目的就是为了做到向数据库增加和删除功能都可以通过组件的形式来完成,而不需要使用多样的功能管理方式,来达到设计思想统一而简单化。

第三层是最顶层,属于配置层,它提供了向数据库中植入组件的机制。每一个组件通过这一层向上公开它们提供的属性和服务的接口。这部分的数据库设计提供了用户接口,用户可以通过它们访问组件接口中提供的属性和服务。如果组件向数据库增加了新的属性和服务功能,那么同样可以通过用户接口访问到这些属性和服务功能。

在嵌入式系统中采用基于组件的方式,实现对数据库的剪裁、配置。此时,系统不再是一个统一不变的整体,而是由多个独立的功能组件组成,每个组件完成特定的数据库功能。这些组件可以在系统内自由加载或卸载,与此同时,系统允许增加组件支持新的功能。这种基于组件的数据库管理系统的最大特点是能够根据实际需要选择其中的一部分而不是全部子系统,这种方式便于裁减和扩充功能,与常规数据库系统相比具有更大的灵活性,从而可以实现以最小的系统内核提供必需的数据库支持。

3.8 XML

从表 3-1 中可知,嵌入式数据库对于 XML 等的支持程度反映了它的扩展性的优良程度。随着 XML 文档的日益广泛应用,XML 数据的存储以及如何有效地管理大量的 XML 文档是亟待解决的问题。这也是嵌入式数据库在网络化背景和移动计算环境下需要处理的重要问题。

XML 是从标准通用标记语言 SGML 衍生出来的一种开放的、以文字为基础的卷标语言,其目的是让互联网上的数据描述有一个简单可行的标准。它与 HTML 的区别在于:HTML 是用来描述展示页面的方法,且只有单一的固定格式,而 XML 则是用来描述页面的内容,并且具有可扩充的灵活格式。

每一个 XML 文档都包含了逻辑结构和实体结构。逻辑结构包含文件中的元素以及元素之间的顺序。实体结构包含文件中使用的实际数据,这些数据可能是存储在用户计算机内存中的文字,也可能是的一个图形档案等。

XML 具有良好的数据存储格式、可扩展、高度结构化、易于网络传输等优点。XML 可以提供在应用程序和系统之间传输结构化数据的方法。像客户信息、信用卡交易、订单和完成请求等,这类数据都能够转换成并 XML 在应用程序之间共享。

应用程序使用 XML 文档时,需要对其进行解析,将其从文档格式转变为程序中可以直接使用的数据结构,解析的结果包括数据之间的逻辑关系(数据结构)和数据信息。

XML 的编程接口主要包括两种：基于树结构的接口和基于事件驱动的接口，通过这些接口实现应用程序对数据信息的增删、修改及节点遍历等。XML 接口组成如图 3-14 所示。

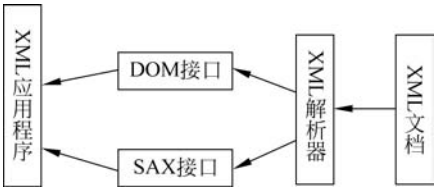


图 3-14　XML 接口组成

简单地说，一个 XML 解析器就是一段可以读入 XML 文档并且分析其结构的代码。XML 文档本身只是一个文本文件，它需要能够识别 XML 格式化信息的解析器来解析并提取其中的内容。根据对文档的不同处理方式，可以划分为基于 DOM 的解析器和基于 SAX 的解析器。前者根据文档的内容建立一个层次的数据结构，提供给用户一个操作文档的接口。后者则由事件驱动，通过串行的方式来处理文档，即当解析器遇到一个开始或者结束标志时，它会向应用程序发送消息，由应用程序决定如何处理。目前常见的解析器有一部分是能同时兼容 DOM 和 SAX 的，如 IBM 的 XML4J 等。

Oracle Berkeley DB XML 是一个可嵌入的开源 XML 数据库，可基于 XQuery 访问存储在容器中的文档，并对其内容进行索引。Oracle Berkeley DB XML 构建于 Oracle Berkeley DB 之上，并继承了其丰富的特性和属性（包括环境、各个级别的事务、Replication 等）。与 Oracle Berkeley DB 一样，它通过应用程序运行，无须人为管理。

Berkeley DB XML 部分包括 XML 索引、XQuery 引擎和 XML 文档解析器。整个 Berkeley DB XML 包含如下组件。

（1）Berkeley DB。BDB XML 继承了 BDB 的可扩展性、缓存、灵活的存储与访问以及事务等特性。这意味着 BDB XML 也可以采用表来存储非 XML 数据。

（2）Xerces C++。Xerces 是 Apache 软件基金会下一个开源的 XML 解析器项目。它提供 C++和 Java 两种语言的支持。它支持的 XML 特性包括名称空间、DTD 和 Schema 验证、SAX 和 DOM 的实现。

（3）Pahtan。Pathan 是一个 XPath 处理器开源项目，它由 DesionSoft、Sleepcat、Data Direct 和 Parthenon Computing 合作开发。Pathan 作为 XPath 功能内置于 Xerces DOM 中。

（4）XQuery。XQuery 包是 BDB XML 的一部分，它为 Xerces DOM 提供了 XQuery 功能。XQuery 是一门功能强大的查询语言，相当于关系数据库中的 SQL。

下面通过一个最简单的查询例子，介绍最基本的 Berkeley DB XML 的编程流程，并介绍 Berkeley DB XML 中的一些基本概念。

本示例程序 query.cpp 使用 C++编写。

```
/*
 * 这是一个最简单的 Berkeley DB XML 程序,描述了如何进行查询和结果处理
 * 这个程序展示了以下几个方面内容:
 * 初始化 Berkeley DB XML
 * 创建 XmlContainer
 * 插入 XML 文档
 * 创建 XQuery 查询和执行查询
 * 在查询中如何使用变量
 * 结果处理
```

```
 * /
#include < iostream >
#include < dbxml/DbXml.hpp >
using namespace DbXml;
int
main(int argc, char * * argv)
{
    // 定义 XmlContainer 的文件名
    std::string containerName = "people.dbxml";
    // 定义 XML 文档内容
    std::string content = "< people >< person >< name > joe </name ></person >< person >< name >
mary </name ></person ></people >";
    // 定义 XML 文档名字,每个存储于 XmlContainer 中的 XML 文档必须有唯一的名字
    std::string docName = "people";

    // 定义 XQuery 查询语句,用来查询姓名等于某个值的 person 节点, 注意 $ name 是一个 XQuery
//变量,可以赋值
    std::string queryString =
        "collection('people.dbxml')/people/person[name = $ name]";
    try {
        // 所有的 BDB XML 程序都需要一个 XmlManager 对象,XmlManager 用于管
        // 理 BDB XML 的各种对象资源
        XmlManager mgr;
        // 检查同名 XML 容器是否已经存在了,存在的话就删除
        if (mgr.existsContainer(containerName))
            mgr.removeContainer(containerName);
        // 用 XmlManager 创建一个 XmlContainer. Berkeley DB XML 把所有的 XML
        // 数据、索引以及其他相关内容存储在 XmlContainer 中,XmlContainer
        // 在磁盘上的表现就是一个.dbxml 文件,当然也可以以其他后缀
        // 作为文件名结尾.我们也可以使用已经创建好的 XmlContainer
        XmlContainer cont = mgr.createContainer(containerName);
        // 修改 container 需要创建一个 XmlUpdateContext 对象
        XmlUpdateContext uc = mgr.createUpdateContext();
        // 插入一个 XML 文档,提供文档名文档内容,和一个 XmlUpdateContext 对象
        // 插入后 XML 文档就会以 Berkeley DB XML 的格式存储于 XmlContainer 中
        cont.putDocument(docName, content, uc);
        // 如果是查询则需要创建一个 XmlQueryContext 对象
        XmlQueryContext qc = mgr.createQueryContext();
        // 可以在 XmlQueryContext 对象中设置需要查询的变量值
        qc.setVariableValue("name", "mary");
        // 接着创建一个 XmlQueryExpression 对象,用来进行查询,用前面创
        // 建的 XmlQueryContext 对象作为参数
        XmlQueryExpression expr = mgr.prepare(queryString, qc);
        // 执行查询,返回 XmlResults 对象
        XmlResults res = expr.execute(qc);
        // 可以通过 XmlQueryExpression::getQuery()方法获得 XQuery 查询语句
        // 通过 XmlResults::size()方法可以知道查询的结果集大小
        std::cout << "The query, '" << expr.getQuery() << "' returned " <<
```

```
            (unsigned int) res.size() << " result(s)" << std::endl;
        // 处理返回的 XmlResults 对象，输出它们的值
        XmlValue value;
        std::cout << "Result: " << std::endl;
        while (res.next(value)) {
            std::cout << "t" << value.asString() << std::endl;
        }
        //异常处理，Berkeley DB XML 的所有对象如果发生异常会抛出 XmlException
        } catch (XmlException &xe) {
            std::cout << "XmlException: " << xe.what() << std::endl;
    }
    return 0;
}
```

本程序的运行结果是：

```
[ying@ying build_unix] $ ./query
The query, 'collection('people.dbxml')/people/person[name = $name]' returned 1 result(s)
Result:
<person><name>mary</name></person>
```

通过这个示例程序可知，Berkeley DB XML 的程序一般有如下几个步骤，创建 XmlManager；创建或打开 XmlContainer；创建 XmlQueryExpression 并执行查询；处理查询结果。读者可以尝试修改 setc.setVariableValue("name","mary")这一句，看查询结果是否有变化。更高级的 Berkeley DB XML 程序可添加事务、环境（Berkeley DB 的 Environment）等功能，在此不再赘述。

3.9　小结

尽管小型嵌入式数据库面临的难题比大型数据库要复杂得多，但是市场需求却是与日俱增，而且将是一个很大的开放市场。在将来，智能移动设备和其他各种装置的嵌入式数据库可以接受连续的数据流，以满足这些装置的应用需求，这是嵌入式数据库系统未来发展基本趋势的一个方面；另一方面，鉴于手持装置使用无线通信技术，这一技术对嵌入式数据库系统的未来发展将产生举足轻重的影响。因此，无线通信技术的不断发展和无线通信业务成本的降低，将是嵌入式数据库系统在未来发展中取得成功的关键因素。同时，制造商将进一步提高各种计算装置的数据处理和存储的能力，从而使得嵌入式数据库的设计变得更加容易，而且在功能上将会得到新的拓展。

本章介绍了嵌入式数据库的各项关键技术的原理和实现方法，限于篇幅，本章不可能将所有重要技术一一介绍，特别是嵌入式数据库是以应用为中心的数据库，应用目的的巨大差异会导致技术上的“抓手”也不同。随着云、物联网等技术的蓬勃发展，嵌入式数据库的关键技术也在不断向前发展，这也需要读者与时俱进，多查阅最新资料，多实践设计，这样更有利于掌握嵌入式数据库。

习题

1. 简述嵌入式数据库的系统结构。
2. 简述嵌入式实时数据库的系统结构。
3. 简述嵌入式移动数据库的系统结构。
4. 请比较 BDB 与 OpenBASE Lite 之间的异同点。
5. 嵌入式数据库的物理存储结构包括哪些?
6. NOR Flash 和 NAND Flash 在性能方面各有什么特点?
7. 典型的内存数据库包括哪些? 简述其主要特点。
8. 提高实时数据库可靠性的方法主要有哪些?
9. 与传统数据库的事务相比,嵌入式实时数据库的事务有哪些特点?
10. 嵌入式数据库的实时并发控制有哪些?
11. 嵌入式数据库的恢复技术主要有哪些?
12. 嵌入式数据库有哪些常用访问算法?