d5000 实时库代码解析

version 2.0 (draft)

by 冉攀峰

2012-07-18

目录

引言

本文用于解析 D5000 实时库(不存在二义的前提下，后文中“D5000 实时库”一律简称“实时库”)源码和分析其实现原理。本文共有四章。

第一章:实时库实现机制。重在分析内存映 IO 射和 System V semaphore 机制。实时库采用内存映射 IO 来完成两项工作:一，更快速地存取实时库;二，实现多进程并发(并行)访问实时库。而 System V semaphore 是实时库锁机制实现的基础。

第二章:实时库代码解析。阐述实时库实现原理，主要包括三个方面:

1) context-app-table 三级结构。
2) 锁机制(系统锁，应用锁，表锁)的实现。
3) 索引机制(带洞直接索引)的。

第三章:实时库评价。分析实时库的评价方法，并借此发掘实时库的不足之处，为实时库的改进埋下伏笔。

第四章:实时库改进。在第三章分析的基础上，探讨今后如何改进实时库，以提高性能或更加适合业务场景。

第一章:实时库实现机制

内存映射 IO 和 System V semaphore 机制是实时库实现的基石。表现在：首先，多进程通过内存映射 IO 使 context-app-table 三级结构驻于内存，共享之。context-app-table 三级结构用于管理整个实时库。其次，实时库存取采用内存映射 IO，因为内存映射 IO 比无缓冲 IO 和 C 标准 IO 更加快速。最后，多进程并发访问实时库时，通过锁机制保持同步，锁通过 System V semaphore 实现。

一. 内存映射 IO

内存映射 IO 的接口主要有 mmap 和 munmap 函数(详情参考 manpage)。

内存映射 IO 的原理是将文件的全部或部分映射到进程的虚拟地址空间中，不通过 read/write 和 fread/fwrite 存取文件，而是用 mmap 返回的指向内存映射区域首址的指针，以数组的方式直接操作内存，完成存取。内核自动完成内存映射区域和文件的同步。

多进程通过内存映射并发访问同一文件，是否采用共享内存的方式(如图 1.1.0 所示)，用户无法感知，因此是平台相关的。 参考 Advanced Programming in Unix Environment 和 IEEE Std 1003.1-2008 POSIX Base Specifications， Issue 7 ，均未提及内存映射采用共享内存。

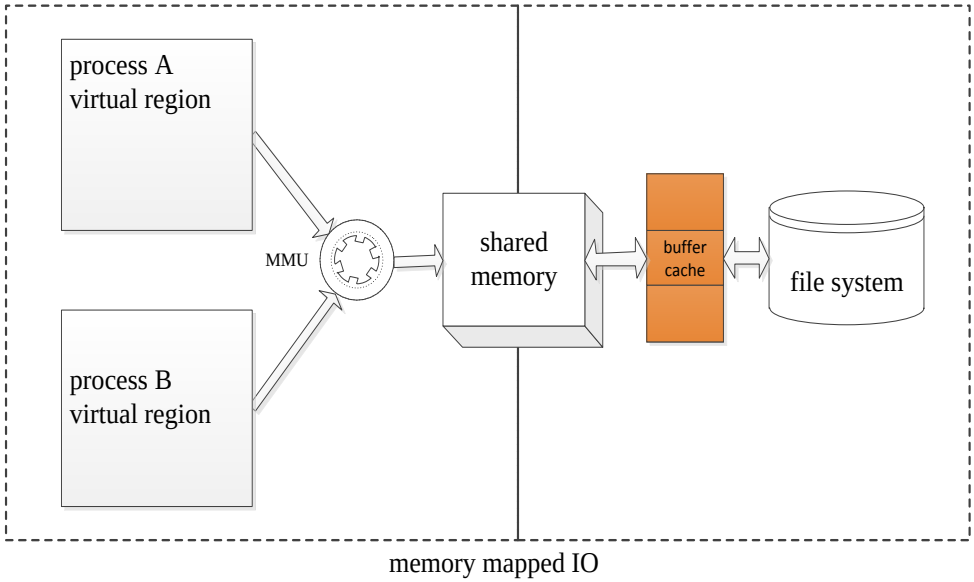

图 1.1.0 多进程并发访问内存映射文件(采用共享内存的实现)

多进程争用对映射文件的访问，会破坏数据的一致性。因此，采用锁机制同步之。

内存映射 IO 有两个特点:假定的文件大小和无保证的内存使用。在第三章中会做进一步阐述。

二. System V semaphore

System V semaphore(不存在二义的前提下，后文中“System V semaphore”简称“semaphore”)的接口有:semget，semctl，semop 等函数(详情参考 manpage)。

semaphore 机制实现了信号量集合，可以原子性地完成对多个资源的申请和释放的控制操作。 实时库 D5000 中锁以此实现。

第二章 实时库代码解析

实时库代码解析主要涉及:context-app-table 三级结构(不存在二义性的前提下，后文中“context-app-table 三级结构”简称“三级结构”)、锁机制和索引机制。 三级结构为实时库的静态布局，而锁机制和索引机制是实时库的动态运行，在分析三者关系的基础上，完成下装案例分析和实时库部分 API 分析。

一. context-app-table 三级结构

实时库三级结构与文件系统中的实时库表文件存放路径一一对应。简言之，实时库有若干个态，每个态控制若干个应用，每个应用控制若干张表。所有态、应用和表在观念上是驻于内存的。实时库的 API 函数中，大多数以记录为存取单位，少数的 API 函数以全表为存取单位。进程存取实时库时，必先选择锁定当前态、当前应用和当前表。**一言以蔽之，三级结构有两大功用：一，管理全体所有的态、应用和表；二，选择锁定当前态、应用和表。**

CTableOp 是实时库基本 API 类，实时库提供其他的 API 是基于 CTableOp 实现的。实时库的设计不具有层次性，CTableOp 涵盖了实时库设计的全部细节。因此，解析实时库代码，实质上，是分析实时库 API 类 CTableOp 的实现。

CTableOp 含有三个成员，类型分别为 COdbSystem、COdbTable 和 COdbFie ld 三个类，三级结构的操作由这三个类控制。图 2.1.1 是 CTableOp 的类图。

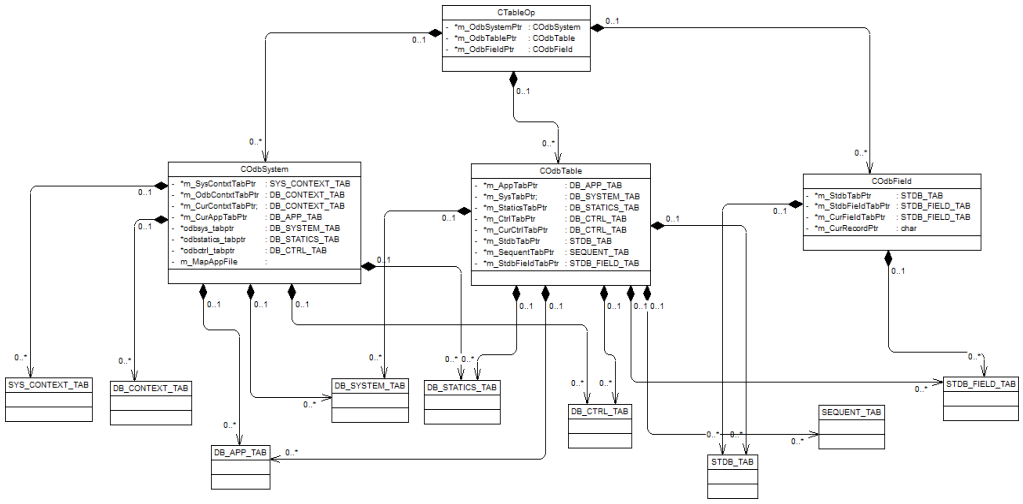

图 2.1.1 CTableOp 的类图

CTableOp::m_OdbSystemPtr：类型为 COdbSystem(源文件为 odb_system.h 和 odb_system.cpp)。其定义如代码 2.1.1 所示：

```
//file:odb_system.h
COdbSystem{
        ... ...
        ... ...
        char     m_SysFile[120];
        char*    m_SysAreaPtr;
        struct   SYS_CONTEXT_TAB* m_SysContxtTabPtr;
        struct   DB_CONTEXT_TAB*  m_OdbContxtTabPtr;
        struct   DB_CONTEXT_TAB*  m_CurContxtTabPtr;

        std::map<string, char*>   m_MapAppFile;
        struct   DB_APP_TAB*      m_CurAppTabPtr;
        struct   DB_CTRL_TAB*     odbctrl_tabptr;

};
```

代码 2.1.1 COdbSystem 类

COdbSystem::m_SysFile 的值为"${D5000_HOME}/data/rtdbms/system.dat"，见 odb_system.cpp 函数 int COdbSystem::InitEnv() 。

COdbSystem::m_SysAreaPtr 指针指向 system.dat 文件的映射内存区域的首址，见 odb_system.cpp 函数 int COdbSystem::SetContxtTabPtr(char* system_file_ptr)。

COdbSystem::m_SysContxtTabPtr 和 COdbSystem::m_OdbContxtTabPtr 分别指向 system.dat 中的 SYS_CONTEXT_TAB 和 DB_CONTEXT_TAB 数组的首部，DB_CONTEXT_TAB 为当前态控制结构。见 odb_system.cpp 函数 int COdbSystem::SetContxtTabPtr(char* system_file_ptr),如图 2.1.2 所示。

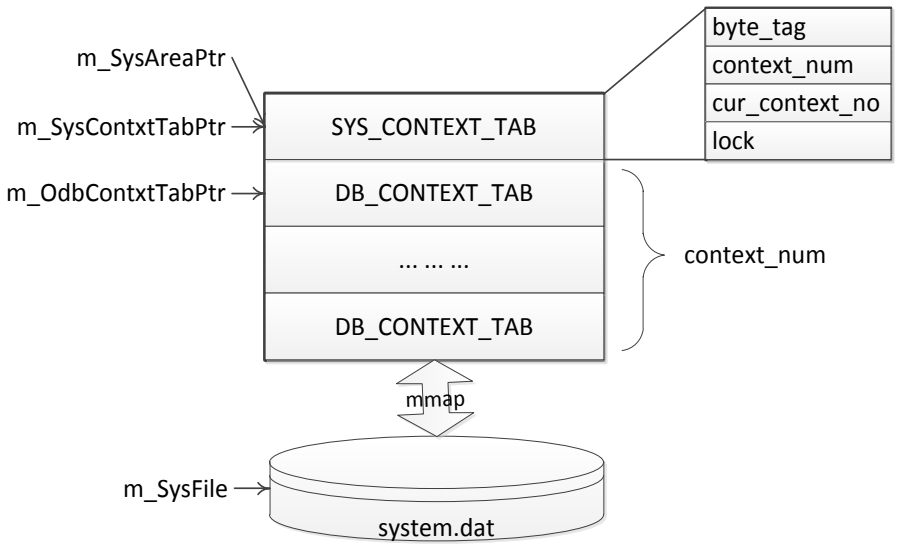

图 2.1.2 system.dat 文件结构

COdbSystem::m_CurContxtTabPtr 指向当前态控制结构，通过 MoveTo(const short context_no, bool read_only)函数切换当前态。m_SysContxtTabPtr->context_num 为系统中态的总数，m_SysContxtTabPtr->cur_context_no 为当前态在 DB_C

ONTEXT_TAB 数组中的索引。所以：m_CurContxtTabPtr=m_OdbContxtTabPtr+ m_SysContxtTabPtr->cur_context_no。

COdbSystem::m_MapAppFile 为 map 数据结构。此 map 的 key 为当前态下的应用名，如实时态有"fes"，"scada"等应用；value 为文件系统中的应用文件映射到内存段首地址，如"scada" 应用对应文件"${D5000_HOME}/data/rtdbms/ context01/scada/scada.dat"映射到内存段首地址。m_MapAppFile 是当前态下的全部应用的控制结构。函数 int COdbSystem::AddApp(const char* all_name, bool read_ only)完成对该结构的设置。应用文件的路径由 int COdbSystem::GetAppFile(char * file_name, const short context_no, const char* all_name) 函数获得。应用文件的结构如图 2.1.3 所示：

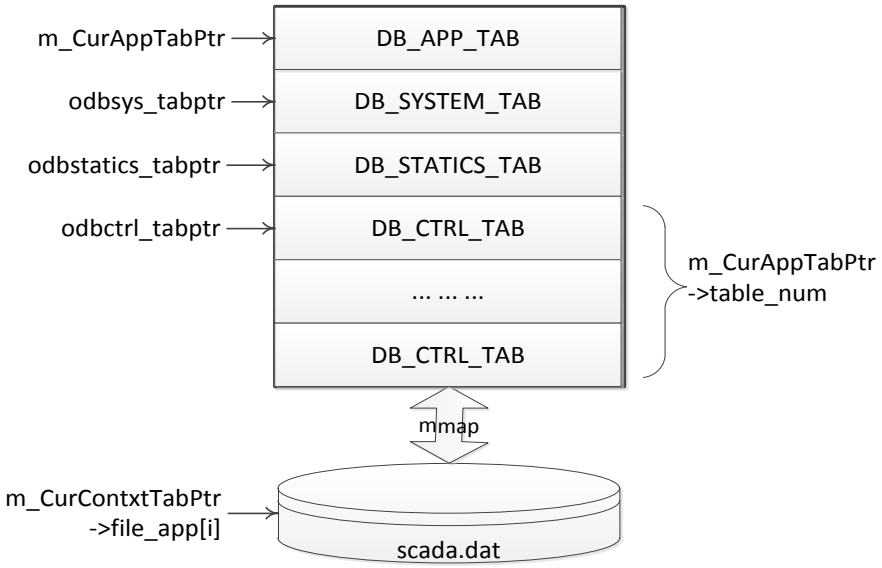

图 2.1.3 应用文件 scada 的文件结构

COdbSystem::m_CurAppTabPtr 指向当前应用控制结构，通过 int COdbSyste m::ShiftTo (const char* all_name, bool read_only)函数可以切换当前应用。COd bSystem::m_CurAppTabPtr=COdbSystem::m_MapAppFile{appname}。应用文件中包含了全体表控制结构 DB_CTTL_TAB，由 COdbSystem::odbctrl_tabptr 指向。见 odb_system.cpp 中 int COdbSystem::SetCurAppTabPtr(char* app_file_ptr)函数。

CTableOp::m_OdbTablePtr：类型为 COdbTable(其源文件为 odb_table.h 和 odb_table.cpp)，其数据成员定义如代码 2.1.2 所示：

```
//odb_table.h
class  COdbTable {
                … …
                … …
                struct DB_APP_TAB*     m_AppTabPtr;       //App info area
                struct DB_SYSTEM_TAB*  m_SysTabPtr;
                struct DB_STATICS_TAB*  m_StaticsTabPtr;
                struct DB_CTRL_TAB*    m_CtrlTabPtr;      //Ctrl info area
                struct DB_CTRL_TAB*    m_CurCtrlTabPtr;   //current table's Ctrl info
                struct STDB_TAB*       m_StdbTabPtr;      //
                struct SEQUENT_TAB*    m_SequentTabPtr;   //
                struct STDB_FIELD_TAB*  m_StdbFieldTabPtr;  //
                char* m_DbFilePtr[MAX_APP_TABLE];
                time_t m_DbOpenTime[MAX_APP_TABLE];
                int  m_MapSize[MAX_APP_TABLE];
                char* m_SequentAreaPtr;                 //current sequent area
                char* m_DataAreaPtr;                   //current data area
                int   m_CurRecordPointer;               //current record position
                char  m_FileName[120];
}
```

代码 2.1.2 COdbTable 类定义

COdbTable::m_AppTabPtr 和 COdbSystem::CurAppTabPtr 含义一样，均指向当前应用控制结构。

COdbTable::m_CtrlTabPtr 和 COdbSystem::odbctrl_tabptr 含义一样，均指向全体表控制结构。

COdbTable::m_CurCtrlTabPtr 指向当前表控制结构，通过 int COdbTable::MoveTo (const int r_table_no)函数切换当前表控制结构。 其中 r_table_no 为表号，此处所谓的表号是表在关系数据库的中编号，以此编号为索引可以从 COdbTable::m_AppTabPtr->table_no 数组中得到当前表控制结构在全体表控制结构中索引。即：COdbTable::m_CurCtrlTabPtr= COdbTable::m_CtrlTabPtr+COdbTable::m_AppTabPtr->table_no[r_table_no]。表文件结构如图 2.1.4 所示。见文件 odb_table.cpp 中函数 int COdbTable::SetTableEnv(const struct DB_CTRL_TAB* const init_ctrl_tab_ptr)。

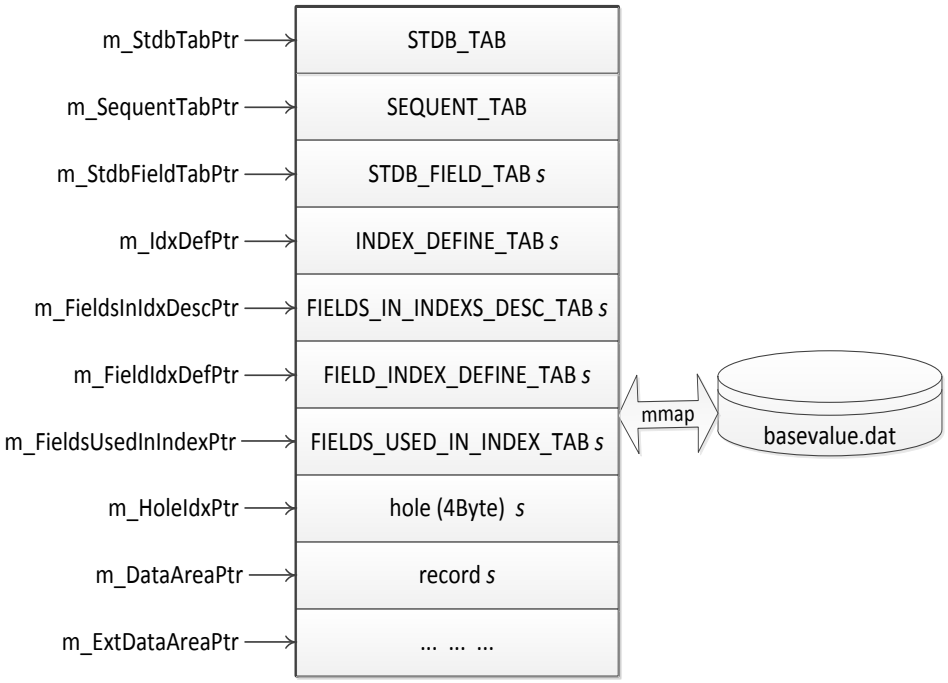

图 2.1.4 表文件 basevalue 的文件结构

COdbTable::m_HoleIdxPtr 指向主键索引(即洞索引)所需的按从小到大排列的溢出区记录号数组的首地址。COdbTable::m_DataAreaPtr 指向表中记录区域的首地址，COdbTable::m_ExtDataAreaPtr 指向表中溢出记录区域的首地址，COdbTable::m_StdFieldTabPtr 指向表的全体域控制结构。

COdbTable::m_DbFilePtr[MAX_APP_TABLE]保持表文件的映射内存段首地址。

CTableOp::m_OdbFieldPtr: 类型为 COdbField（其源文件为 odb_field.h，odb_field.cpp）,COdbField 用于控制存取表中的域。其定义如代码 2.1.3 所示：

```
//odb_field.h
class   COdbField {
                ... ...
                ... ...
                struct STDB_TAB*       m_StdbTabPtr;
                struct STDB_FIELD_TAB*  m_StdbFieldTabPtr;
                struct STDB_FIELD_TAB*  m_CurFieldTabPtr;
                char*  m_CurRecordPtr;
}
```

代码 2.1.3 COdbField 定义

COdbField::m_StdbTabPtr 同 COdbTable::m_StdTabPtr 含义一样，都指向当前表文件的 STDB_TAB 结构。

COdbField::m_StdbFieldTabPtr 同 COdbTable::m_StdTabPtr 含义一样，都指向表文件的全体域控制结构即 STDB_FIELD_TAB 数组。STDB_FIELD_TAB 包

含域的各种属性信息，比如域的数据类型、所占字节大小、偏移量、编号、域名和取值范围等一系列信息。

COdbField::m_CurFieldTabPtr 指向当前域控制结构，其用于存取记录中的某个域。

COdbField::m_CurRecordPtr 指向当前记录数据区域，上文中已经提到 COdbTable::m_DataAreaPtr 指向表中记录区域的首地址，记录区域包含 m_StdbTabPtr->record_sum 条记录，每条记录由记录头部 RECORD_HEAD_STRU 和数据部分组成。通过 m_CurRecordPtr 和 m_CurFieldTabPtr 可以锁定到想要存取的当前记录的当前域在表中实际存储地址。其锁定方法是：m_CurFieldTabPtr=m_StdbFieldTabPtr+m_StdTabPtr->field_no[r_field_no] ， field_value_ptr= m_CurRecordPtr+m_CurFieldTabPtr->offset。可以通过 int MoveTo(const int r_field_no)函数切换当前域。

总结之：三级结构包含实时库全体的控制结构和全体表文件。实时库下装的过程就是将这些数据预先映射到内存，对实时库数据的存取就是对三界结构的操作。图 2.1.5 给出了三级结构的全景视图。

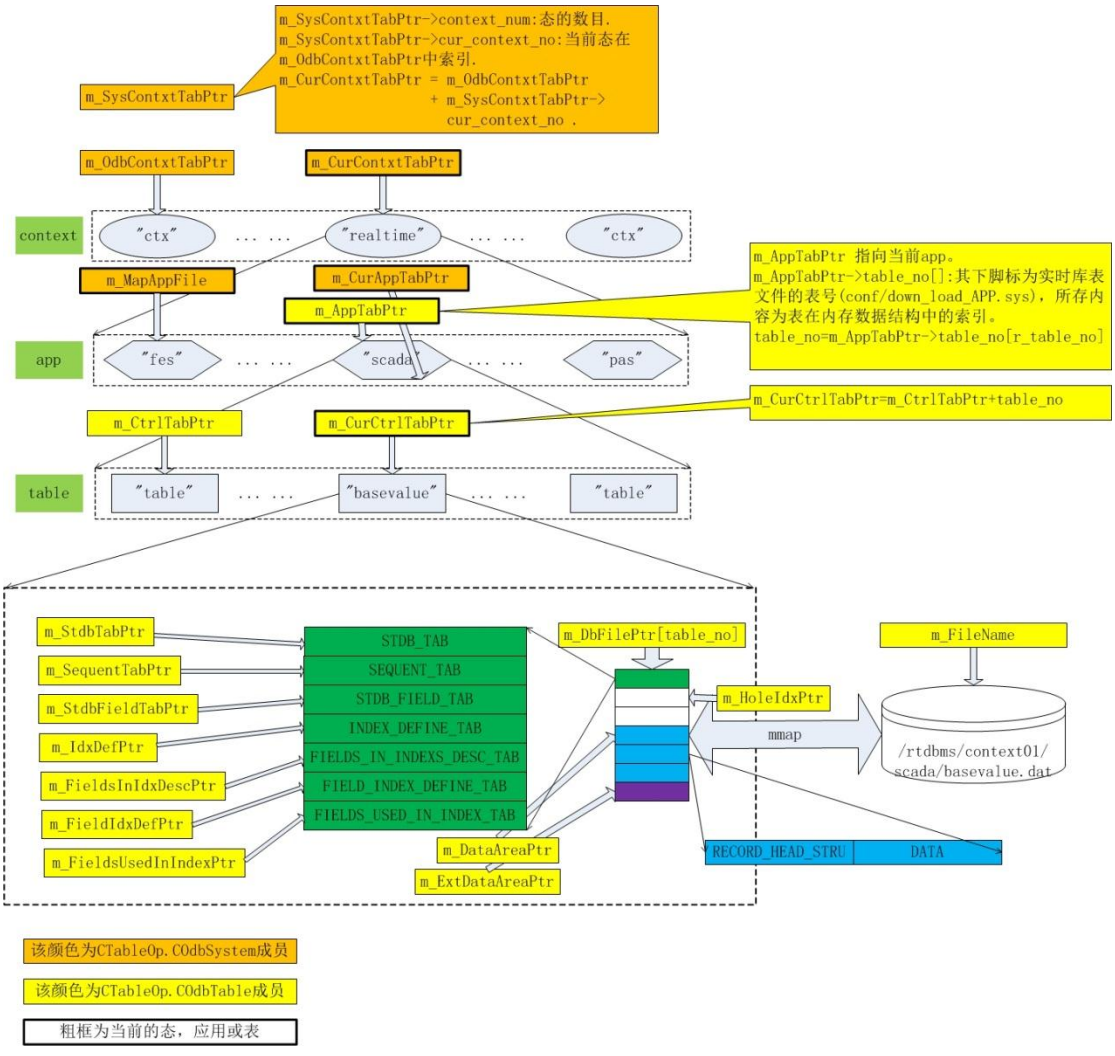

图 2.1.6 context-app-table 三级控制管理结构全景视图

表 2.1.1 给出了三级结构的全体控制结构和当前控制结构的对照比较。

表 2.1.1 context-app-table 的全体和当前控制结构对照表

级别	数据类型	磁盘文件	映射内存段首	全体控制结构		当前控制结构
				数据结构	变量	
context	DB_CONTEXT_TAB	${D5000}/data/rtdbms/system.dat	CTableOp::m_OdbSystemPtr->m_SysContxtPtr	array	CTableOp::m_OdbSystemPtr->m_OdbContxtPtr	CTableOp::m_OdbSystemPtr->m_CurContxtPtr
app	DB_APP_TAB	${D5000}/data/rtdbms/context01/scada/scada.dat	CTableOp::m_OdbSystemPtr->m_MapAppFile["scada"]	map	CTableOp::m_OdbSystemPtr->m_MapAppFile	1.CTableOp::m_OdbSystemPtr->m_CurAppTabPtr 2.CTableOp::m_OdbTablePtr->AppTabPtr
table	DB_CTRL_TAB	${D5000}/data/rtdbms/context01/scada/basevalue.dat	CTableOp::m_OdbTablePtr->m_DbFilePtr[i]	array	1.CTableOp::m_OdbSystemPtr->odbctrl_tabptr 2.CTableOp::m_OdbTablePtr->m_CtrlTabPtr	CTableOp::m_OdbTablePtr->m_CurCtrlTabPtr

二. 实时库下装过程

实时库下装是指将实时库 context-app-table 三级结构和全体表文件从文件系统映射到内存，以备进程访问实时库。${D5000_HOME}/bin/文件夹下的 sys_ctl 和 down_load 命令用于下装实时库。

down_load 为客户端程序，对应服务器端程序 download_server。download_server 在 D5000 系统中，属于"data_srv"应用，其注册的进程名为"down_srv"。两者之间通过 TCP 通信，端口号为 11110。down_srv 代理 down_load 对关系数据库（dameng）进行查询，关系数据库中存储实时库表的元数据，**该元数据是除了创建表中除全部记录之外其余所有部分所需的全部信息，**如表的 STAB_TAB、SEQUENT_TAB、STDB_FIELD_TAB、... …、FIELD_INDEX_DEFINE_TAB 和 FIELDS_USED_IN_INDEX_TAB 部分，主键索引，辅助索引。**总而言之，是除表的记录数据区之外的全部信息。这些元数据封装在 TDownloadOutPara 和 SEQIndex 类中。**

多表下装是对单表下装的迭代，因此，只分析单表下装即可。单表下装的过程如图 2.2.1 所示：

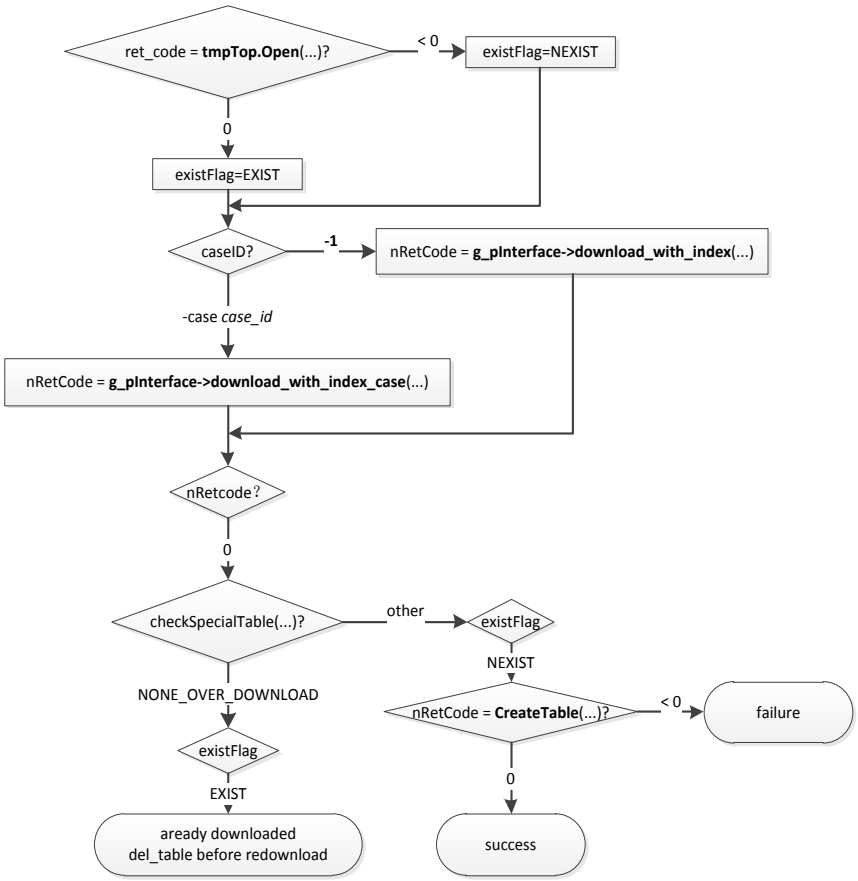

图 2.2.1 单表下装过程

图 2.2.1 中，粗体标识的函数是下装过程中的主要处理，分别为三个函数：

1. CTableOp::Open(…) 函数，该函数部署全体态、应用和表的控制结构和设置当前态、应用和表的控制结构。
2. g_pInterface->download_with_index(…)函数向 down_srv 进程请求实时库表的除记录之外其他全部信息，此信息可以用来创建空表。
3. CreateTable(…)函数调用 CObdTable::CreateTable(…)函数完成表的创建，如果表文件已经存在，则 CObdTable::CreateTable(…)函数映射该文件到内存，如果表文件不存在，创建空表。

总结之，单表下装分成三个阶段：一，设置当前控制结构；二，获取表的元数据；三，创建表。

三. 实时库部分 API 解析

CTableOp 操作表时，先调用 CTableOp::Open(…)函数打开表。实时库下装时也用到了 CTableOp::Open(…)函数，然后这两个函数有不同的函数签名，操作也略有不同：API 中的 Open(…)函数不但完成下装时的 Open(…)函数的全部操作，此外还设置了成员变量 CTableOp::m_CObdFieldPtr，以备随后的记录操作。在第二章.一.小节中已经详细阐述了如何定位某个记录的指定的域。

CTableOp 的函数通过 CObdTable 类实现，如：CTableOp::TableGet(…)通过 CObdTable::GetAllRecords(…)实现，CTableOp::TableWrite(…)函数通过 CObdTable::WriteRecordByKey(…)实现等等。

CTableOp 的函数按照操作表的粒度，可以将分成三类：一，无表操作；二，全表操作；三，记录操作。注意此处的"表"的含义是指表文件的记录数据区域。因此，无表操作实质上是一组读表的某些元数据的函数，此类函数并不会访问表中的数据区。如 GetTableNameByNo，GetTableNoByName 等等。全表操作是指读全表 TableGet，在溢出区插入记录 TableWrite，在溢出区删除记录 DeleteRecord 等操作。实时库表溢出区的主键索引采用顺序索引，因此，表溢出区中的记录按序排列，在表溢出区插入和删除，需要移动此记录之后的全部记录。记录操作指对表中的某个记录进行操作，而不影响表中其他记录。如读记录 TableGetByKey，更新记录 TableUpdate，修改记录 TableModfifyByKey，在非溢出区插入记录 TableWrite，在非溢出区删除记录 DeleteRecord 等操作等。

四. 实时库索引机制

实时库提供索引机制包括主键索引和辅助索引，辅助索引用于 TableOp 中涉及关系查询的 API，主键索引用于按键存取记录的 API。本小节会给出关于主键索引的详细解析。

建立的索引的目的是加快数据存取，数据存取的方式决定索引的取舍，索引对于上层的数据存取操作呈现出数据存储的逻辑结构，并对下层的数据存储形成了物理结构。

实时库的主键索引采用一种叫"带洞直接定位表"的方式。

表中的记录槽数是表最大允许插入记录数目 record_sum 的 2 倍加 1，这些记录槽连续分布，系统将全部记录槽分为前后两个数目相等的记录区，两种记录区之间有一个空槽不存储记录。前区为是直接索引的正常记录区，由 CTableOp::m_OdbTablePtr->m_DataAreaPtr 指向，后区是间接索引的溢出记录区，由 CTableOp::m_OdbTablePtr->m_ExtDataAreaPtr 指向，正常区和溢出区各自是含有 record_sum 个记录的数组。当记录的记录号小于 record_sum，属于正常区记录，否则属于溢出区记录，它们分别存储在正常区和溢出区。

当插入正常区记录时，直接以该记录号作为正常区记录数组的下脚标，修改记录的头部 RECORD_HEAD_STRU 的 exist_tag 字段为 RECORD_EXIST。当删除正常区记录时，与插入类似，将对应记录槽的头部 RECORD_HEAD_STRU 的 exist_tag 字段修改成 RECORD_EMPTY。经过一段时间的插入和删除操作，正常区就会不连续分布的空槽出现。

当插入/删除溢出区记录时，首先要查找索引表，CTableOp::m_OdbTablePtr->m_HoleIdxPtr 指向索引表的首址。溢出区记录按记录号有序地连续地存储在溢出区，索引表的索引项只包含记录号，也是有序地连续地存储在索引表中，记录号相同的记录和索引项各自分别相对于溢出区和索引表开始位置的偏移量是相等的，而该偏移量为此索引项在索引表的脚标。按键存取记录时，取键的低 32 位记为记录号，以该记录号为查询键二分查找索引表，比对索引项中的记录号和查询键，最终返回记录的偏移量或失败。因为表中至多用 2^32 个记录，所以至多查找 32 次。

图 2.4.1 是实时库主键索引的一个示例，定位记录号 30349 的记录的过程为：二分查找记录号 30349（图中沿着绿色结点的路径），找到后，返回索引项的下脚标 2，即记录号 30349 的记录相对于溢出区首址的偏移量为 2。如果查找正常区记录，以记录号做下脚标访问正常区即可。

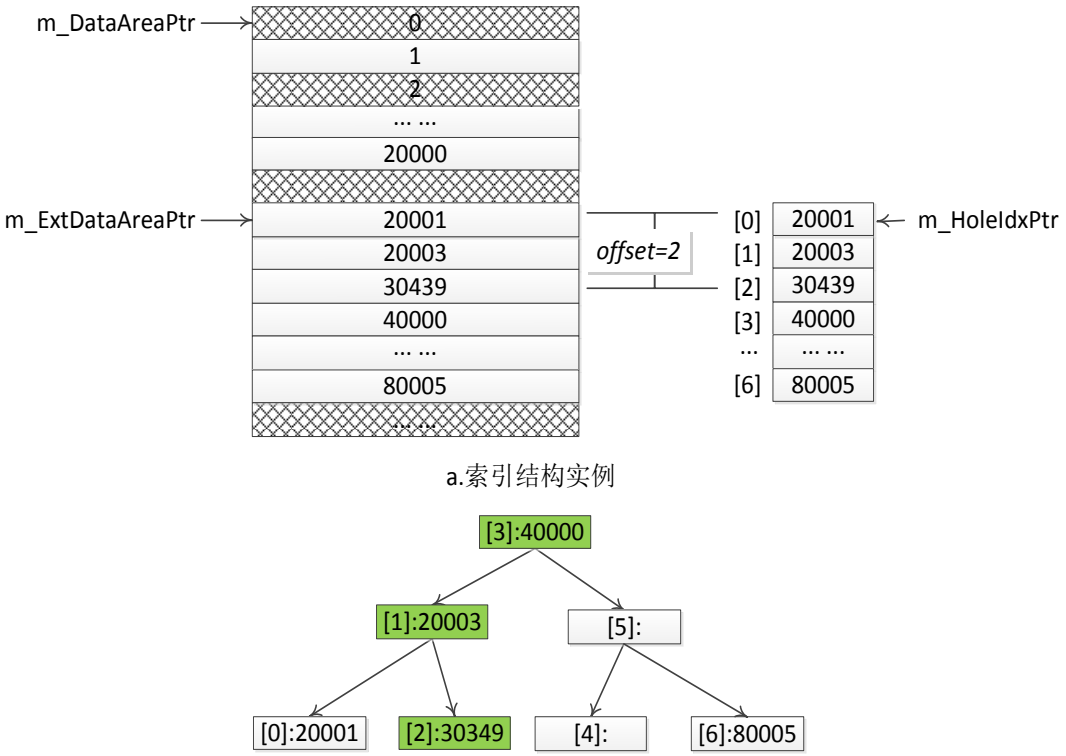

图 2.4.1 实时库主键索引查询实例

其实在同一表中，记录的长度固定且有序排列，因此也可以不必采用顺序表建立索引，直接对表中的溢出区记录进行二分查找。此处采用索引表的主要原因，可能是考虑不同表的记录长度是不相等的，而采用索引表，可以对固定长度的索引项进行二分查找，便于实现。

索引表和溢出区都采用连续且有序地存储，因此，溢出区的空槽出现在溢出末尾且连续分布。插入和删除溢出区记录需要移动该记录之后的所有记录和以及对应的索引，移动这些记录和索引项用 memmove 函数完成。

主键索引采用这种索引方式时，插入/删除正常区记录很高效，插入/删除溢出区记录的效率取决于溢出区记录的数目。考虑到溢出的概率较低而且溢出区的记录数目较少时，插入和删除操作的平均效率是可以接受的。此外，整个插入/删除操作不及更新/读取操作频繁，因此实时库存取有很好的平均性能。显然，这种索引的不足之处是消耗太多的内存，这也说明了一种算法无法同时在时间和空间上达到最优，要么以时间换空间，要么以空间换时间。此外，采用这种索引方式非常有利于按主键顺序读取全表的操作，这类操作在 dbi 和 mmi 中大量存在。如果主键索引采用 hash 实现，结果会如何呢？采用 hash 实现的优点有：一，插入和删除成为了记录操作，而且记录操作有很高的性能；二，便于实现记录锁。而缺点有：一，不方便按主键顺序读取全表的操作，dbi 和 mmi 的响应会变的非常慢；二，占用更多的内存。

五. 实时库锁机制

实时库的锁本质上是互斥量 mutex，通过信号量集合（semaphore）实现。创建新的信号量集合或获得已建信号量集合通过 semget 函数完成。该函数声明如代码 2.5.1 所示，该函数成功调用时，返回信号量集合的 ID。该 ID 在随后的 semctl 和 semop 函数中使用。使用 semget 时，需要传入一个参数 key，key 和 ID 的区别类似文件的路径和文件描述符，通过相同的 key 可以获得同一信号量集合的 ID。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

代码 2.5.1 semget 函数声明

实时库的锁只使用信号量集合中一个信号量，因此需要指明该信号量所在的信号量集合的 key（sem_key）和在信号量集合中的下脚标（sem_no）才能唯一确定一把锁，锁的结构如代码 2.5.2 所示。

```
//odb_prv_struct.h
struct LOCK_STRU
{
        DB_OBJECT_T lock_object;
        DB_LOCK_T   lock_type;
        int         lock_num;
        int         lock_pid;
        int         sem_key;
        int         sem_no;
        int         back_no;
        int         reserved[2];
}
```

代码 2.5.2 实时库锁的结构

实时库的锁按照锁的对象（lock_object）分为系统锁（OBJECT_SYSTEM）、应用锁（OBJECT_APP）和表锁（OBJECT_TABLE），这三种锁分别控制对系统文件(system.dat) 、应用文件（如 scada.dat 等文件）和表文件（如 basevalue.dat 等文件）并发访问。

实时库的锁按照锁的访问类型（lock_type）可以分为读锁（DB_LOCK_READ）和写锁（DB_LOCK_WRITE ），实时库的锁用于解决读者（读实时库的进程或线程）写者（写实时库的进程或线程）问题。

系统锁用于全体态控制结构的管理（见 odb_system.cpp）。增加态（AddContext 函数）和删除态（DelContext 函数）对 system.dat 加写锁，而切换当前态（MoveTo 函数）时对 system.dat 加读锁。增加应用（AddApp 函数）和删除应用（DelApp）时对 system.dat 加写锁。注意：增加和删除应用时，并不修改应用文件，而是修改 system.dat 文件。

应用锁用于全体表控制结构的管理（见 odb_table.cpp）。创建表（CreateTable 函数）和删除表（DeleteTable 函数）时，会修改当前应用文件。因此需要加应用写锁。

表锁用于表文件和索引的并发访问控制（见 odb_tableop.cpp）。如插入记录（TableWrite）和删除记录（DeleteRecords），按键读/修改/跟新记录等操作均采用表锁。

访问实时库时的操作可以分为记录操作和全表操作。对于记录操作采用表锁时，加锁范围和加锁粒度不一致，如读第一条记录和修改最后一条记录的操作是不冲突的，但是，加了表锁之后，这两种操作是无法并发执行。新版的实时库中，表锁做出了一些修改，开发者已经注意到区分全表操作和记录操作，修改了锁的语意。修改之后，全表操作对表的结构进行修改，因此视为对表结构的"写"操作，加写锁；而记录操作不会修改表结构，因此视为对表结构的"读"操作，加读锁。这样读第一条记录和修改最后一条记录就可以同时进行了，然而他们对于读记录的操作并不加锁，而且修改同一条记录可以仍然会产生冲突。因此，修改后的实时库的问题有：一，读脏数据；二，覆盖写。

第三章 实时库评价

实时库的设计首要考虑的问题是一作为存储系统，实时库应该拥有怎样的存取特性。因为存取特性决定存储方法，存储方式决定存取性能。不清楚存取特性的差异，就无法不同存储系统之间的设计差异。

实时库存取特性主要有：

1. 存取是否具有局部性？如果存取有局部性，那么，可以采用连续存储结构，通过预先缓存-延迟写机制可以很好利用局部性。但是，如果存取过程中，随机访问很多，那么局部性就很差，采用连续存储结构并不合理。
2. 支持并发访问。并发引起对共享数据的争用，数据一致性遭到破坏（读脏数据和覆盖写），需要使用并发控制机制（进程或线程同步）使多进

程或多线程互斥的访问共享数据。并发控制是有代价的，会造成系统存取速度降低，更甚至造成死锁。

3. 不同存取操作的出现的频繁性。插入、删除、更新和读取操作中，那种操作更加频繁？那种操作相对不频繁？牺牲不频繁操作的性能以提升频繁操作的性能可以获得非常好的统计平均性能。
4. 存取时间。存取时间过长，影响响应时间和用户体验的满意度。如果需要较短的存取时间，那么可能需要用空间换取时间。
5. 存储的数据是否就有结构性。数据项是无结构的字节序列还是有结构的记录，是定长的记录还是可变长的记录对存储结构都很很大的影响。

描述实时库的存取特性应该使用**更新周期内存取相同表的次数的分布律**和**更新周期内不同存取操作的次数的分布律。**通过这两个随机变量可以量化的分析实时库的性能。从直观上分析，如果更新周期内相同表的访问次数不超过 10 次的概率为 99%，超过 10 次的概率为 1%，那么我们就可以得出实时库中表的并发访问的负载约为 10。此时，我们认为表锁时可以接受，但是当负载超过 100，我们认为是勉强接受，当超过 1000 时，我们认为这样的系统采用表锁时不能接受的。对于跟新周期内不同存取操作次数，我们假设插入/删除操作的概率是 1%，而读取/更新操作的概率是 99%，那么我们尽量提供读取/更新操作的效率。

实时库设计过程中的问题有：

1. 实时库的更新操作不具有局部性。实时库的数据量比较大（仅仅 scada 下的表文件就有 2.6GB），虚拟机的内存较小（2GB），频繁随机更新新导致频繁交换，使性能降低。但是 mmi 和 dbi 所需的读全表的操作就有很好的局部性。
2. 表锁不合理。如果表锁的语意是控制读表文件和写表文件，那么对记录操作加表锁，就无法并发操作同一表的不同记录，而这样的操作并不冲突，应予以支持。如果表锁的语意是控制全表操作和记录操作，则会引起读脏数据和覆盖写的问题。
3. 多进程使用 mmap 函数之间共享表文件，mmap 使用假定的文件大小，而且不保证使用内存。如果表文件最大容纳 2^32 个记录，然后实际容纳记录为 0，那么 mmap 映射此表文件到内存时，需要分配足够容纳 2^32 个记录的内存映射区域，这样以来会占用更多的虚拟地址空间，物理空间和交换区。mmap 无法保证内存使用，它只是能够做到访问映射区域的页时，该页可以交换到内存以供使用。
4. 实时库的实现和上层应用紧密耦合，缺乏层次化设计。

第四章 实时库改进

一.表锁的改进

表锁可以修改成两级锁结构：表锁-记录锁。此前表锁用作区分读写的语意导致锁的粒度和锁的范围不一致，而表锁用作区分全表和记录操作的语意又会导致读脏数据和覆盖写。图 4.1.1 用以说明表锁-记录锁的原理。

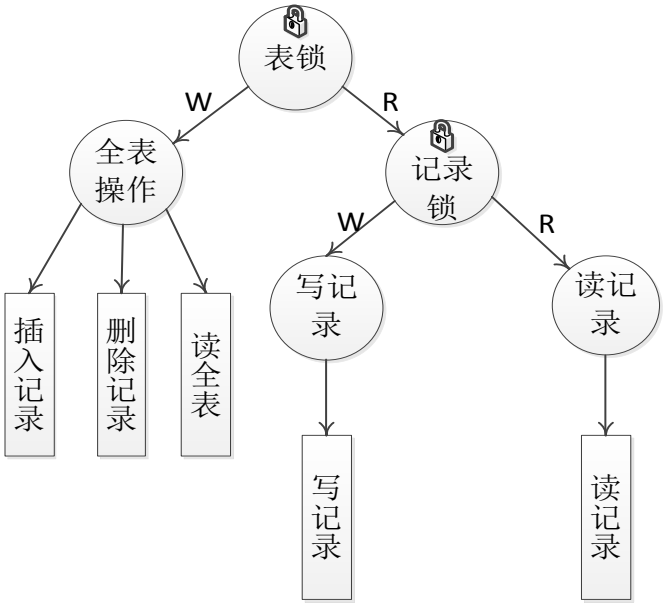

图 4.1.1 表锁-记录锁机制

分析采用表锁-记录锁机制后的并发控制情形：

1. 插入记录和删除记录。插入记录和删除记录都是全表操作，可能搬移移表中若干记录，所以如果不加以互斥，会破坏表的结构。因此插入和删除都申请写表锁，写阻塞写。
2. 插入记录和写记录。类似，这两种操作也不能同时进行，所以前者加写表锁，后者加读表锁。其语意为读阻塞写或写阻塞读。
3. 读记录和读记录。这两种操作允许同时进行，均先申请读表锁，故能同时进入临界区，然后申请读记录锁，故能同时进行操作。其表锁和记录锁的语意均为读允许读。
4. 写记录和读记录。先申请读表锁，两种操作都可以申请下一级记录锁，如果是操作不同记录，则毫无冲突。如果操作相同记录，记录锁控制对对同一条记录的互斥访问，此时记录锁的语意读阻塞写或写阻塞读。
5. 写记录和写记录。和情形 4 类似。

通过以上分析可以看出表锁的语意和记录锁的语意相似，改变表的结构的全表操作视为"写"操作，不改变表的结构的记录操作视为"读"操作。当然以上分析的合理性基于这样的假设—造成实时库的性能瓶颈的主要因素是内存和 swap 区频繁交换。