

第25章 完整性管理

本章要点：

实现锁

分析V\$Lock

监控系统中的锁

避免锁：可能的解决策略

锁定与分布式数据库

使用门实现锁

当数据库的容量与系统中的用户数量增加时，对内部资源的争用也变得越来越复杂，这样最终会导致性能出现问题。内部争用通常表现为有足够的空闲内存并且 CPU不是很忙，但是系统中仍然存在性能问题。上述现象使一个初学的管理员对系统中发生的事情感到惊奇。本章的目的就是帮助管理员识别这样的争用问题。本章要讨论的资源是锁与门。

数据库争用可以被分为两大类，稍后在本章中对这两者分别进行详细讨论。

锁——用于限制其他用户对数据的存取。

门——Oracle使用的一个内部机制，用来管理和维护内存锁。

25.1 实现锁

数据一致性——Oracle锁机制保证查询返回的数据与查询开始的时刻的数据一致。因此，即使其他用户正在改动数据，执行查询的用户只会看到“静态”的数据。这提供了数据一致性。

数据并行性——锁用于防止对同一对象进行存取的用户之间破坏性的相互作用。Oracle的锁机制使多个用户能够同时对数据进行安全存取。这提供了数据并行性。

数据完整性——事务在执行一个提交或回滚以前保留锁。这提供了数据完整性。

单用户数据库或许从来不需要锁，但是当你工作在一个多用户环境下，有一个合适地自动满足数据并行性、一致性和完整性的机制是重要的。Oracle通过使用一个内部锁定机制维护数据的完整性、并行性与一致性。Oracle如何维护数据的完整性、并行性与一致性将在随后的小节中讨论。

Oracle通过获得不同类型的锁，代表用户允许或阻止其他用户对相同资源的同时存取并确保不破坏数据的完整性，从而自动满足了数据的完整性、并行性与一致性。资源（resource）这个术语在这里指诸如表与行的用户对象。当实现锁时，要记住的一个关键问题是它不能在系统中形成瓶颈，阻止对数据的并行存取。因此Oracle根据所执行的数据库操作自动地要求不同层次上的锁定以确保最大的并行性。例如，当一个用户正在读取某行中的数据时，其他用户能够向同一行中写数据。然而，用户不允许删除表。这些涉及到使用不同锁定级别的事务能够进行什么级别的对象存取或数据存取的问题将在下节中讨论。

25.1.1 锁的需要

在你实际研究锁定以前，了解一下需要锁定的各种情况。下面是一些阐述了数据一致性与并行性为什么如此重要的实例：

在一个用户正在修改表中数据的同时，另一个用户正试图删除该表。Oracle通过为第一个用户取得一个表级锁定阻止此类事情的发生。第二个想要删除表的用户等待第一个用户释放了表级锁后才能够删除此表。

用户A正试图读取用户B的某个事务中的一些数据，在用户A的事务开始后，该事务由用户B修改和提交。用户A读取用户B提交的数据。这意味着在同一个事务中读取的数据在某个时刻不一致。Oracle保证在一个事务中读取的数据与事务开始时刻保持一致。

Oracle使用系统变化号（SCN）实现事务级的读取一致性。

系统变化号（SCN）是一个数据结构，它定义了一个给定时刻提交的数据库版本。SCN可以被认为Oracle的逻辑时钟。每一次提交它们的数值都要增加。例如，如果某个事务执行了一个UPDATE或提交，该事务将被分配一个假定是500的SCN值，那么下一个提交的事务将得到一个501或更大的SCN值。

某个用户对数据进行修改，另一个用户在第一个用户提交事务以前对同一行进行修改；因此，第一个用户所做的改变丢失了。Oracle通过在该表的行上取得一个行级专有锁并让第二个用户等待的方式来防止此类情况的发生。

一个用户从另一个用户尚未提交的数据中读取数据；也就是说，在用户B的改变提交以前，用户A读取用户B正在修改的行。Oracle通过使用用户A能够读取老数据，也就是说，读取用户B修改以前的数据来防止此类情况的发生。Oracle通过从回滚段中获得数据执行此操作。

25.1.2 锁的概念

Oracle根据前后关系自动决定需要为给定的情况应用什么样的锁以便使用最小的约束提供最大的数据并行性。Oracle有两种锁：共享与专用。

共享锁（share lock）通过数据存取的高并行性来实现。假如获得了一个共享锁，其他用户可以共享相同的资源。许多事务可以获得相同资源上的共享锁，但是对于专用锁来说，情况就不是这样。例如，多个用户可以在相同的时间读取相同的数据。

专用锁（exclusive lock）防止同时共享相同的资源。例如，假如一个事务获得了某一资源上的一个专用锁，那么直到该锁被解除，其他的事务才能够修改那个资源。还有，也允许对资源进行共享。例如，假如一张表被锁定在专用模式下，它并不能阻止其他用户从同一张表中进行选择。

你可以使用锁的级别来实现并行存取并且同时保证数据的完整性。Oracle通过运用最低的约束条件来达到此目的。Oracle会根据需要自动锁定单行或整张表。Oracle使用下述的通用准则决定它需要使用的锁定级别：

一个用户正试图从一行中读取数据时，另一个用户应能够把数据写入同一行中。

一个用户正在更新一行的数据时，其他的用户应能够同时读取同一行的数据。

两个用户正在更新相同的表时，仅当他们试图存取相同的行时才被阻止。

使用锁定机制实现的一个最重要的功能是数据一致性。Oracle在两个层次上保证数据的一

致性：语句层和事务层。

语句层读取一致性（statement-level read consistency）意指任何一条开始执行的语句得到在语句开始执行以前已提交的数据。对数据所做的所有改变，不管是已提交的还是未提交的，在语句执行时对在提交会话以前开始的其他会话来说是不可见的。通过确保其他会话可以得到在语句开始执行时监测到的 SCN 以前提交的数据，Oracle 提供了语句层读取一致性（请参见表 25-1 中的例子）。

表 25-1 其他会话可以得到在语句开始执行时监测到的 SCN 以前提交的数据

时 间	会话-1	会话-2	会话-3
10 00	事务开始		
10 01		事务开始	
10 02	一致性读取 (不能看到会话-2提交的事务)	事务提交与结束事务	
10 03			事务开始
10 04	还是一致性读取		能够看到会话-2提交的数据
10 05			事务提交与结束事务
10 06	一致性读取(不能看到会话-2 与会话-3提交的事务)		

事务层读取一致性（transaction-level read consistency）意指在同一个事务中的所有数据对时间点是一致的。Oracle 通过使用 SCN 机制以及获得所需的专用表锁与行锁来确保事务层读取一致性。

Oracle 通过使用回滚段的数据实现读取一致性。当一条查询进入执行状态时，当前的 SCN 就被确定了。当它开始读数据块的数据时，数据块的 SCN 与监测到的 SCN 进行比较。假如数据块 SCN 的数值大于查询开始时的 SCN，就意味着在该查询开始执行以后，那些数据块中的数据已经变化了。假如从数据块中读出此数据，那么它与查询开始时刻不一致。Oracle 使用回滚段重新构造该数据，因为回滚段总是包含老数据——系统中含有大量修改活动的情况除外，在这种情况下回滚段中的老数据可能被改写。

25.1.3 锁的类型

Oracle 根据操作系统和要在其上获得锁的资源的不同自动获得不同类型的锁。

有四种类型的 Oracle 锁：

数据锁（DML Lock）在表中获得并且用于保护数据，更确切地说是保护数据的完整性。

字典锁（DDL Lock）用于保护对象的结构——例如表和视图索引的结构定义。一个字典锁在 DDL 事务需要它时由 Oracle 自动获得。

内部锁与门（internal lock 和 latch）保护内部数据库结构。门在“使用门实现锁”一节中详细讨论。

分布式锁与并行高速缓存管理（PCM）锁用于并行服务器中（参见第九部分“并行环境与分布式环境”）。

1. 数据锁

（1）DML 锁的用途

DML 锁的用途是保护被多个用户并行存取的数据的完整性。DML 锁可以防止同时进行的

DML操作的破坏性干扰。

例如，DML锁保证一张表中的指定行一次仅能被一个事务修改。它们还保证如果一个未提交的事务包含一个对表的插入操作时，该表便不能被删除。

DML语句使用两种类型的锁结构：

- 1) 事务获得表上的一个共享锁。
- 2) 事务获得它正在改变的每一行上的专用锁。

下面的例子显示了共享与专用类型的锁：

假如有三个用户视图同时对同一行进行修改，他们全都会得到共享的表锁，但是只有最先请求锁的用户得到行锁。

一个正被修改的行总是被专门地锁定以使其他用户不能修改该行，直到包含该锁的事务被提交或回滚为止。

(2) DML锁的模式

下面列举了用于DML的Oracle锁模式。

- 1) 行共享 (RS)：行共享锁是一个共享的表锁。它是一个在被查询行上的专用锁。
- 2) 行专用 (RX)：行专用锁通常表明持有该锁的事务已经对表中的行完成了一次或多次修改。
- 3) 共享锁 (S)：假如一个事务持有一个共享表锁，那么它便防止其他事务在该表中执行DML操作。
- 4) 共享行专用 (SRX)：假如一个事务包含一个SRX表锁，那么它防止其他事务执行DML或SELECT...FOR UPDATE语句。
- 5) 专用 (X)：专用锁是表锁的最具限制性的形式，它允许持有该锁的事务对该表进行独占访问。

注意 你可以通过增加DML_LOCKS与ENQUEUE_RESOURCES参数的值来增加用于一个实例的锁的数量。在V\$LOCK视图中，行锁与表锁分别用Lock Type列中的TX与TM值表示。V\$LOCK视图的Mode列中的SX值针对两种类型的锁。

2. DDL锁 (字典锁)

数据库管理员需要注意数据定义语言 (DDL) 锁以及它们对事务的影响。DDL锁将帮助你诊断并解决锁定问题。当模式对象被一个正在运行的事务引用时，DDL锁能够保护该模式对象的定义。

例如，当用户创建一个过程时，Oracle便自动获得在过程定义中引用的所有对象的DDL锁。

(1) DDL锁的功能

DDL锁使在过程中引用的对象在过程编译结束以前不能被改变或删除。

DDL锁仅影响在DDL操作期间被修改或引用的单个模式对象。它们不锁定整个数据字典。

DDL锁不能被显式地实现。

(2) 不同形式的DDL锁

- 1) 专用DDL锁：当诸如CREATE、ALTER和DROP这样的语句用于一个对象时使用此锁。专用DDL锁的特征：

假如另外一个用户保留了任何级别的锁的话，那么本用户不能得到表中的专用 DDL 锁。例如，假如另一个用户在该表上有一个未提交的事务，那么 ALTER TABLE 语句会失效。

2) 共享DDL锁：当诸如 GRANT 与 CREATE PACKAGE 这样的语句用于一个对象时使用此锁。

共享DDL锁的特征：

一个共享DDL锁不能阻止类似的DDL语句或任何DML语句用于一个对象上，但是它能防止另一个用户改变或删除已引用的对象。

共享DDL锁的另一个特征是在DDL语句执行期间它一直维持，直到发生一个隐含的提交。

3) 可破的分析DDL锁：库高速缓存中的语句或 PL/SQL 对象持有一个用于它所引用的每一个对象的锁。

可破的分析DDL锁的特征：

假如被引用的对象改变了，可破的分析DDL锁检查语句是否应失效。

只要相关的SQL语句仍然在共享池中，可破的分析DDL锁就会持续。假如对象改变了，它检查语句是否应失效。

3. 表锁

事务可以通过发出表 25-1 中的语句获得表锁 (TM)。为了使事务能够保护表中的 DML 存取以及防止表中产生冲突的 DDL 操作，Oracle 获得表锁。例如，假如某个事务在一张表上持有一个表锁，那么它会阻止任何其他事务获取该表中用于删除或改变该表的一个专用 DDL 锁。

表 25-2 显示了不同的模式，当执行特定的语句时，由 RDBMS 获得这些模式的表锁 (TM)。类型列有诸如 TM 的值，模式列的值分别为 2、3 或 6。TM 表示一个表锁；数值 2 表示一个行共享 (RS) 锁，3 表示一个行专用锁 (RX)，6 表示一个专用 (X) 锁。例如，类型列的 TM 值与模式列的数值 3 将被存储在 V\$lock 表中，当发出相应的语句时，发出该语句的会话要用到它们，所以你需要熟悉这些值。在 V\$lock 表中，Oracle 列举了当前由 Oracle 服务器持有的锁，这些锁将在 25.2 节中伴随例子详细讨论。

表 25-2 采用的语句与获得的表锁

语 句	类 型	模 式
INSERT	TM	行专用 (3)(RX)
UPDATE	TM	行专用 (3)(RX)
DELETE	TM	行专用 (3)(RX)
SELECT FOR UPDATE	TM	行共享 (2)(RS)
LOCK TABLE	TM	专用 (6)(X)

当执行一条 INSERT 语句时，V\$lock 表中的类型列的值为 TM，模式列数值为 3，数值 3 意指行专用 (RX) 锁。要想得到一些辅助细节，参考 25.2 节。对于除锁表以外的所有语句，在 V\$lock 表中有两个项目：一个项目对应于表锁 (TM)，另一个项目对应于事务锁 (TX)。对于诸如插入更新的语句来说，删除一个 TM 锁仅用来防止在被锁定对象上的相互冲突的 DDL 操作，这意味着当一个用户正向一张表插入时，他或她会获得一个模式 3 下的 TM 锁 (RX)。假如另一个用户试图删除同一张表，就必须获得模式 6 下的 TM 锁 (专用)。TM (3)(RX) 锁防止 TM (6)(X) 会话获得该锁并且第二个会话仍然要等待。表 25-3 说明了会话获得的锁模式

以及允许的锁模式。

表25-3 允许的锁模式和操作

SQL语句	表锁模式	允许的锁模式
select * from tname...	无	RS,RX,S,SRX,X
Insert Into tname...	RX	RS,RX
Update tname...	RX	RS*,RX*
Delete From tname	RX	RS*,RX*
Select...From tname For Update of...	RS	RS*RX*S*,SRX*
Lock Table In ROW SHARE MODE	RS	RS,RX,S,SRX
Lock Table In ROW EXCLUSIVE MODE	RX	RS,RX
Lock Table In SHARE MODE	S	RS,S
Lock Table In SHARE ROW EXCLUSIVE MODE	SRX	RS
Lock Table In EXCLUSIVE MODE	X	无
RS: Row Share	SRX: 共享行专用	
RX: Row Exclusive	X: 专用	
s: Share		

假如另一个事务已经获得了该行上的一个锁，那么将会发生等待。

事务锁（TX）：当一个事务发出表25-4中的语句时将获得此类型的锁。事务锁总是在行级上获得。TX锁独占地锁住该行并阻止其他事务修改该行，直到持有该锁的事务回滚或提交数据为止。

表25-4 事务锁语句

语 句	类 型	模 式
INSERT	TX	专用(6)(X)
UPDATE	TX	专用(6)(X)
DELETE	TX	专用(6)(X)
SELECT FOR UPDATE	TX	专用(6)(X)

要想获得TX锁，事务首先必须获得该表上的一个TM锁。例如，当发出一条INSERT语句时（参考表25-1），必须获得模式3（RX）中的一个TM锁。在获得了RX模式的TM锁后，事务必须获得专用（X）模式的TX锁（参见表25-4）。假如另一个事务在同一行上有一个TX锁，那么就不能获得TX锁；并且，假如在该表中已经有了一个专用（X）模式下的TM锁，那么也不能获得TM锁。

其他的模式是4（共享）与5（共享行专用），但是这些锁定模式一般不发生在数据库中，因此不必过多讨论。

25.1.4 人工锁定

假如你想要执行一张表中列值的全局更新并且希望事务对该表进行单独存取，以便该事务不必等待其他事务完成该表的操作，那么你可以通过人工锁定该表以防止其他事务获得该表中的锁，从而实现这一点。

示例：LOCK TABLE <Table Name> IN <mode>

你可以明确地在不同模式下锁定表：

行共享。

共享。

共享行。

共享行专用。

行专用。

当用户执行如下的语句时获得共享模式（4）锁：

```
SQL> LOCK TABLE <table name> IN SHARE MODE;  
Table(s) Locked.
```

当用户执行如下的语句时获得共享行专用模式（5）锁：

```
SQL> Lock table <table name> in SHARE ROW EXCLUSIVE MODE;  
Table(s) Locked.
```

现在你可以在不必与其他事务发生争用的情况下修改列值。

实际上，必须使用这些语句的应用很少；因此，同其他每日频繁发生的锁定类型相比，它们被认为是不很重要的。

25.1.5 数据库死锁

死锁（deadlock）是指两个或多个用户都在等待被彼此相互锁定的同一数据而形成的一种局面。

大部分数据库死锁发生在表索引中。来自数据库中一个单行的 SELECT 语句或许会在存储池中放置不止一个锁项目。一个单独的行收到一个锁，但是每一个包含该行数值的索引节点也将得到分配的锁。

Oracle 的共享锁定方案确保维护所有数据库的完整性并且更新不能无意中覆盖以前对数据库的更新。在维护共享锁时有一些不利条件。在 Oracle 中，每一个锁需要 Oracle 实例存储池中的 RAM 存储器的 4 个字节，大型的 SQL SELECT 语句会引起存储器短缺，这可能会损坏整个数据库。例如，假设一条 SELECT 语句把 1000 行检索到缓冲区中，那么就需要 4000 字节的锁定空间。这会引发数据库死锁。

25.2 分析 V\$lock

Oracle 在动态状态表 V\$lock 中存储与数据库中的锁有关的所有信息。本节分析了当资源被锁定时数据库中的各种情况并且检查 V\$lock 表以查看 Oracle 如何报告该锁。

下面是对 V\$lock 结构的描述：

```
ADDR RAW(4)  
KADDR RAW(4)  
SID NUMBER  
TYPE VARCHAR2(2)  
ID1 NUMBER  
ID2 NUMBER  
LMODE NUMBER  
REQUEST NUMBER  
CTIME NUMBER  
BLOCK NUMBER
```

用于分析锁定状态的重要的列如下：

sid——这是会话标识符。

type——这是所获得的或会话等待的锁类型。示例值如下：

TX事务

TM DML或表锁

MR 介质恢复

ST 磁盘空间事务

本章仅涉及TX与TM锁。

lmode/request——本列包含锁的模式。可能的取值如下：

0无

1空

2 行共享 (RS)

3 行专用 (RX)

4 共享 (S)

5 共享行专用 (SRX)

6 专用 (X)

假如lmode列含有一个不是0或1的数值，表明进程已经获得了一个锁。假如 request列含有一个不是0或1的数值，表明进程正在等待一个锁。假如 lmode列含有数值0，表明进程正等待获得一个锁。

下面的select语句是一种用于检查是否有会话正在等待获得任何表中的锁的快速方法。

```
Select count (*)  
From v$lock  
where lmode = 0
```

假如这个select语句返回一个大于0的数值，那么就表明系统中当前有锁会话正等待获得锁。

id1——根据锁类型的不同，此列中的数值有不同的含义。假如锁类型是 TM，那么此列中的数值是将被锁定或等待被锁定的对象的标识。假如锁类型是 TX，那么此列中的数值是回滚段号码的十进制表示。

id2——假如锁类型是TM，那么此列中的数值是0。假如锁类型是TX，那么此列表示交换次数——也就是说，回滚槽重新使用的次数。

使用此信息，你现在可以检查各种情况以及分析不同情况下存储于此表中的数值。下节对数据库中经常发生的锁以及如何从 V\$lock视图中读出此信息进行详细的分析。

25.2.1 案例1：专用锁定的表

为了便于讨论，假设将要被锁定的表是 employee表并且会话标识是 28。假设某个用户发出如下语句：

```
Lock table employee in exclusive mode;
```

此语句将在专用模式下锁定该表。假如一张表被一个用户专用锁定，那么另一个用户能够在该表上使用的唯一的 SQL语句是select语句。在该锁被解除以前，不允许其他用户在此表上进行插入、修改或任何 DDL操作。

使用下面的 select 语句检查 V\$lock 表中的记录：

```
Select sid,
       type,
       lmode,
       request,
       id1,
       id2
From v$lock
where sid = 28;
```

当执行此 select 语句时，得到如下输出结果：

SID	TY	LMODE	REQUEST	ID1	ID2
28	TM	6	0	7590	0

注意如下的监测结果：

V\$lock 表中仅有一项。

所获得的锁是 TM（表）锁。

lmode 列的值为 6，这表明会话已经获得了一个表级专有锁。

id1 列的值为 7590，这是 employee 表的对象标识。正如对 TM 锁所预料的一样，id2 列值为 0。你可以使用 sys.obj\$ 表得到对象描述。你可以使用如下的 select 语句：

```
Select name
      From sys.obj$
     where obj#      =      7590;

      Name
      .....
      Employee
```

request 列的值为 0，这表明该锁已经被此会话获得。

25.2.2 案例2：会话更新专用锁定表的行

在案例2中，当另一个会话正试图修改一张表的某一行时，该表被一个会话专用锁定。这里是由试图修改该表的会话所发出的 SQL 语句：

```
Update employee
Set Name = 'MANISH'
where emp_id      =      '1086';
```

在本例中，在 V\$lock 表中第一个会话的项目与案例 1 中的仍然相同，但是第二个会话的项目非常有趣。假设第二个会话的会话标识是 29。现在假设你执行如下的 select 语句：

```
Select sid,
       type,
       lmode,
       request,
       id1,
       id2
From v$lock
where sid in (28,29);
```

输出结果如下：

SID	TY	LMODE	REQUEST	ID1	ID2
28	TM	6	0	7590	0
29	TM	6	0	7590	0

28 TM	6	0	7590	0
29 TM	0	3	7590	0

下面是本例的监测结果：

会话28的项目仍然与案例1中的相同。

有一个会话29的项目。此项目针对 TM锁。注意lmode列的值为0，request列的值为3。这意味着会话29正等待获得该表中的一个表锁。会话29必须先获得表锁以标记它当前正在使用此表。这将防止其他任何会话在此表上发出 DDL语句。一般，获得此表锁标志着该表被会话使用。这里的id1列再次指出了已经被锁定的表的对象标识。

事务（TX）锁仍然没有被会话29获得；如果没有其他的会话正在存取或修改相同的行，仅在获得TM锁以后，会话29才能够获得TX锁。

此时如果会话28回滚或提交了事务，会话29就能够获得它正等待的锁。假如你在会话28已经提交了事务后在V\$lock表中检查会话29的项目，你可以得到两条记录。输出结果如下：

SID	TY	LMODE	REQUEST	ID1	ID2
29 TM		3	0	7590	0
29 TX		6	0	327680	10834

下面是监测结果：

在V\$lock中会话28的锁项目不再存在，因为它已经提交了事务。

会话29现在能够成功地获得TM锁，request列中的数值3移动到lmode列，表明该锁已被会话获得。

会话29还获得表中的另一个锁；它是一个事务锁或TX锁。在Type列中此选项的值为TX，lmode的值为6，表明是一个事务锁。

id1中的值327680是回滚段的号码。要获得回滚段的名称，你可以使用V\$rollname视图。

下面的select语句将给出回滚段的名称，会话29开始在此回滚段中写入回滚信息。

```

Select name
From v$rollname
where usn      =      trunc(327680/65536)

NAME
-----
RB03

```

因此，会话29开始在回滚段RB03中写入回滚信息。

不管何时，只要会话必须获得一个TX锁，那么它首先必须获得一个TM表锁。在获得该表锁后，只要没有其他会话已经在同一行上获得TX锁，该会话便可以获得TX锁。总之，该会话必须获得两个锁。

25.2.3 案例3：一个会话试图更新另一会话更新过的行

此锁定案例是每日锁定状况中最常见的情况。当两个会话试图同时修改同一行时，首先修改该行的会话得到锁，第二个会话要想得到锁就必须在第一个会话后等待。仅在第一个会话提交或回滚事务时，它获得的锁才能够被释放。

在本例中，会话29（第一个会话）已经把employee表的一条记录中的雇员名字更新为TOM并且还没有提交此修改。会话30（第二个会话）目前试图修改同一行。要分析V\$lock，你可以使用与上一个例子中相同的一条select语句，但是要用相关的sid值替换sid列：

SID	TY	LMODE	REQUEST	ID1	ID2
29	TM	3	0	7590	0
29	TX	6	0	327680	10834
30	TM	3	0	7590	0
30	TX	0	6	327680	10834

基于这个select语句的输出，可以得到如下的监测结果：

会话29（第一个会话）在V\$lock中有两个记录锁：TM锁和TX事务锁。

会话30在此表中有一个对应于TM表锁的项目。在此情况下，会话30能够立刻获得TM锁，因为没有其他会话排它地锁定此表。

会话30有另外一个对应于事务锁（TX）的记录。此项目的request列值为6。这意味着会话30正等待获得一个行专用锁。id1与id2列中的值与对应于会话29的id1与id2列中的值相同。

一个会话锁定了一行，而另一个会话正试图更新该行是最常见的锁定案例。

假如会话29现在回滚或提交与案例2中一样的事务，那么会话30可以获得该锁并且该锁项目的request列中的值将移动到lmode列中，表明已经获得了该锁。

25.3 监控系统中的锁

既然你已熟悉了锁定与相关的系统表的概念，那么很容易用脚本检测锁。在掌握了前一小节中所包含的信息的前提下，用户能够编写他自己的锁检测脚本。尽管如此，这里仍然给出了一些锁监控脚本。本节将包含一些用于检测和监控锁的有用的脚本。

清单25-1中的脚本将帮助你得到有关被锁定在数据库中的对象的数量。它将报告当前锁定的所有对象和它们的锁模式，它还将报告正在等待获得锁的所有会话。

清单25-1 用于检查系统中锁的简单脚本

```

Select s.username,
       s.sid,
       l.type,
       l.id1,
       l.id2,
       l.lmode,
       l.request,
       p.spid PID
From v$lock l,
     v$session s,
     v$process p
Where s.sid = l.sid
And    p.addr = s.paddr
And    s.username is not null
Order By id1, s.sid,request;

```

USERNAME	SID	TY	ID1	ID2	LMODE	REQUEST	PID
MARK	39	TM	4573	0	3	0	19271
MANISH	41	TM	4573	0	2	0	20155
KASTURI	116	TM	4573	0	2	0	19914
ADITI	125	TM	4573	0	2	0	19906
ADITI	95	TM	12547	0	2	0	19906
JYOTI	95	TM	15397	0	3	0	20364
KASTURI	116	TM	15397	0	2	0	19914
ADITI	39	TM	15397	0	2	0	19906
MANISH	41	TX	65548	91271	6	0	20155

ADITI	125 TX	65548	91271	0	6 19906
MARK	39 TX	196626	107701	6	0 19271
JYOTI	95 TX	262156	118264	6	0 20364
KASTURI	116 TX	327699	200758	6	0 19914

所有锁的类型是 TM (2或3) 和 TX (6)。如前所述, 这些锁是在任何数据库中最常见的锁。

快速浏览输出可以得到如下结果:

有3个表对象——它们是4573、12547和15397——当前正被锁定在数据库中。

会话39正把对象4573锁定在行专用模式中。

会话41正把对象4573锁定在行共享模式中。

会话116正把对象4573锁定在行共享模式中。

会话125正把对象4573锁定在行共享模式中。

会话95正把对象12547锁定在行共享模式中。

会话95正把对象15397锁定在行专用模式中。

会话116正把对象15397锁定在行共享模式中。

会话39正把对象15397锁定在行共享模式中。

会话95、116、125、39和41各有一个事务锁 (TX)。最重要的观察结果是会话 125在 request列中有一个非零值, 说明它正在等待一个锁。因为会话 41和125的TX选项上的id1与id2值相匹配, 你可以推断出事务 41阻塞了事务 125。因为事务 125与事务 41之间的公用资源是对象4573, 所以你能够由此推断出会话 41已经锁定了对象4573中的某一行并且会话 125正试图锁定同一行——因此, 它正在等待。

此输出结果容易解释, 因为在会话 125与41之间仅有一个公共资源并且可以很容易地指出正引起锁问题的资源。在大多数情况下, 最常见的是在两个会话之间不止一个公共资源, 当然它们中的一个将被锁定。在这种情况下, 很难使用前面的输出结果解释哪一个资源被锁定。你必须使用 V\$sqltext视图, 它存储了属于系统全局区 (SGA) 中的共享 SQL指针的SQL语句的文本。使用前面的输出结果, 查找正等待获得锁的会话的标识, 然后使用下面的 select语句解释该会话正试图锁定以及正等待哪一个资源:

```
Select sqltext
from v$sqltext a,v$session b
where a.address = b.sql_address
and a.hash_value = b.sql_hash_value
and b.sid = 125
order by piece;
```

上述SQL语句的输出结果如下:

```
SQL_TEXT
-----
update employee set sname = 'JYOTI' where emp_id = '100';
```

这意味着会话 41已经锁定了雇员记录 100, 会话 125正试图锁定同一行。因此, 使用锁监控报告以及SQL语句能够很容易地查明被锁定的资源。

通用SQL脚本

本节将提供几个通用脚本以帮助你确定数据库中的锁信息。

清单25-2是一个通用脚本，它显示了SQL文本、SID以及当前在数据库中持有的锁的对象名。

清单25-2 显示数据库锁的信息

```

Set pagesize 60
Set linesize 132
select s.username username,
       a.sid sid,
       a.owner||'|'||a.object object,
       s.lockwait,
       t.sql_text SQL
from v$sqltext t,
     v$session s,
     v$access a
where t.address = s.sql_address
and   t.hash_value = s.sql_hash_value
and   s.sid = a.sid
and   a.owner != 'SYS'
and   upper(substr(a.object,1,2)) != 'V$'
/

```

上述SQL语句的输出结果如下：

USERNAME	SID	OBJECT	LOCKWAIT	SQL
SHARON	8	MARK.CONFIG	E0034C98	update mark.config set irn=12 where ➤irn=4
KASTURI	12	KASTURI.EMP	E0034A5C	update kasturi.emp set tabno=10 ➤where tabno=5
ADITI	99	DAVID.CHANNEL	E0034D99	update david.channel set c_irn=3 ➤where c_irn=1

清单25-3是一个通用脚本，它产生一个持有（或等候）在数据库中的锁的易读报表。

清单25-3 产生在数据库中持有的锁的报表

```

select B.SID,
       C.USERNAME,
       C.OSUSER,
       C.TERMINAL,
       DECODE(B.ID2, 0, A.OBJECT_NAME, 'Trans-'||to_char(B.ID1)) OBJECT_NAME,
       B.TYPE,
       DECODE(B.LMODE,0,'-Waiting-',
              1,'Null',
              2,'Row Share',
              3,'Row Excl',
              4,'Share',
              5,'Sha Row Exc',
              6,'Exclusive','Other') "Lock Mode",
       DECODE(B.REQUEST,0,' ',
              1,'Null',
              2,'Row Share',
              3,'Row Excl',
              4,'Share',
              5,'Sha Row Exc',
              6,'Exclusive','Other') "Req Mode"
from   DBA_OBJECTS A,
       V$LOCK B,
       V$SESSION C
where  A.OBJECT_ID(+) = B.ID1
and    B.SID = C.SID
and    C.USERNAME is not null
order by B.SID, B.ID2;

```

上述SQL语句的输出结果如下：

Sess ID	USERNAME	Op Sys User ID	TERMINAL	OBJ NAME or TRANS_ID	TY Lock Mode	Req Mode
7	JYOTI	jla	JLA	TRANSMISSION	TA Row Excl	
8	JOHN	jxl	JXL	TABLE_CONFIG	TM Row Excl	
9	BONITA	bog	BOG	DEPT_SCHED	TX -Waiting-	Exclusive
10	MANISH	mdk	MDK	AUDIT	TX Exclusive	
12	MARK	mdb	MDB	INDEX_BLOCKS	TM Row Excl	

另一个有用的视图是 V\$session_wait，它列举了活动会话正在等待的事件。在 V\$session_wait 视图中登记的事件是 enqueue。因此，假如你查询 V\$session_wait 视图寻找一个被锁定的或者你认为被锁定的会话，你会得到清单 25-2 中的输出。

清单 25-4 中的语句是由 SQL*Plus 发出的，用于对 V\$session_wait 的查询输出结果进行格式化。

清单 25-4 使用 V\$session_wait 查询会话中的锁

```
col event format a8
col p2text format a5
col p2 format 999999
col p3text format a5
col p3 format 999999
col wait_time format 999999999
col secs format 99999
col state format a8
col seq# format 99999
set pagesize 24
set verify off

select event, p2text,p2 ,p3text,p3,seq#,wait_time,state,seconds_in_wait secs
from v$session_wait
where sid = 125;
```

上述SQL语句的输出结果如下：

EVENT	P2TEX	P2	P3TEX	P3	SEQ#	WAIT_TIME	STATE	SECS
enqueue	id1	65548	id2	91271	6845	0	WAITING	850

p2与p3列包含id1与id2值，id1与id2是能够从V\$lock中获得的数值。输出结果还报告了会话被锁定的秒数。

注意 当试图解释 V\$session_wait 的输出结果时，应当记住检查 state 列的值，因为它是一个非常重要的参数。此列可能的取值有：

Waiting—表明会话当前正在等待事件的发生。当 state 列中的值为 Waiting 时，seconds_in_wait 列将包含会话用于等待事件所花费的以毫秒为单位的实际时间。

Waited unknown time—当 TIMED_STATISTICS 参数被设置为 false 时，此值被显示出来。

Waited short time—表明会话短期等待。

Waited known time—表明会话已经获得了它正等待的事件。wait_time 列包含了会话实际等待的时间。

另一个重要的视图——V\$sysstat——包含了整个系统的所有重要的统计信息。你可以使用V\$sysstat对每一个会话等待锁定的次数有一个总的了解。

使用如下的SQL语句：

```
select decode(class,1,'User',
              2,'Redo',
              4,'Enqueue',
              8,'Cache',
              16,'OS',
              32,'Par Ser',
              64,'SQL',
              128,'Debug') class1,
       statistic#,name,value
from v$sysstat
WHERE name = 'enqueue waits';
```

上述SQL语句的输出结果如下：

Class	S#N	Name	Value
Enqueue	23	enqueue waits	19989

通过此SELECT语句，你能够推断出系统中有 19989次对锁定的等待。注意当锁被释放时，此计数器被递增；假如会话当前正等待一个锁的话，此值没有被反应在计数器中。假如 enqueue waits 的值太高，那么应用程序需要重新检查——有可能对同时运行存取相同资源的作业进行重新排定。

清单25-5是一个用于产生对等待锁的用户（会话）的报告通用脚本。

清单25-5 产生等待锁的用户的报告

```
column username format A15
column sid       format 9990   heading SID
column type      format A4
column lmode     format 990    heading 'HELD'
column request   format 990    heading 'REQ'
column id1       format 9999990
column id2 format 9999990
break on id1 skip 1 dup
spool tfslckwt.lst
SELECT sn.username,
       m.sid,
       m.type,
       DECODE(m.lmode, 0, 'None',
              1, 'Null',
              2, 'Row Share',
              3, 'Row Excl.',
              4, 'Share',
              5, 'S/Row Excl.',
              6, 'Exclusive',
              lmode, ltrim(to_char(lmode,'990'))) lmode,
       DECODE(m.request, 0, 'None',
              1, 'Null',
              2, 'Row Share',
              3, 'Row Excl.',
              4, 'Share',
              5, 'S/Row Excl.',
              6, 'Exclusive',
              request, ltrim(to_char(m.request,'990'))) request,
       m.id1,
       m.id2
FROM   v$session sn,
       v$lock m
```

```

WHERE (sn.sid = m.sid AND m.request != 0)
      OR (   sn.sid = m.sid
            AND m.request = 0 AND lmode != 4
            AND (id1, id2) IN (SELECT s.id1, s.id2
                              FROM v$sqllock s
                              WHERE request != 0
                              AND s.id1 = m.id1
                              AND s.id2 = m.id2)
      )
ORDER BY id1, id2, m.request;
spool off
clear breaks

```

上述SQL语句的输出结果如下：

USERNAME	SID	TYPE	HELD	REQ	ID1	ID2
SYSTEM	12	TX	Exclusive	None	131087	2328
SCOTT	7	TX	None	Exclusive	131087	2328
AMY	8	TX	Exclusive	None	131099	2332
MANISH	10	TX	None	Exclusive	131099	2332
BOLDT	12	TX	None	Exclusive	131099	2332

清单25-6是一个通用脚本，它显示了锁并给出了 SID 以及要删除的序列号；此脚本产生的报告给出了持有锁的会话的信息，还给出了使用 ALTER SYSTEM KILL SESSION 命令进行删除所需要的信息。

清单25-6 显示持有锁的会话的信息

```

set linesize 132 pagesize 66

break on Kill on username on terminal
column Kill heading 'Kill String' format a13
column res heading 'Resource Type' format 999
column id1 format 9999990
column id2 format 9999990
column lmode heading 'Lock Held' format a20
column request heading 'Lock Requested' format a20
column serial# format 99999
column username format a10 heading "Username"
column terminal heading Term format a6
column tab format a35 heading "Table Name"
column owner format a9
column Address format a18
select  nvl(S.USERNAME,'Internal') username,
        nvl(S.TERMINAL,'None') terminal,
        L.SID||'|'||S.SERIAL# Kill,
        U1.NAME||'|'||substr(T1.NAME,1,20) tab,
        decode(L.LMODE,
                1,'No Lock',
                2,'Row Share',
                3,'Row Exclusive',
                4,'Share',
                5,'Share Row Exclusive',
                6,'Exclusive',null) lmode,
        decode(L.REQUEST,1,'No Lock',
                2,'Row Share',
                3,'Row Exclusive',
                4,'Share',
                5,'Share Row Exclusive',
                6,'Exclusive',null) request
from    V$LOCK L,
        V$SESSION S,
        SYS.USER$ U1,

```

```

SYS.OBJ$ T1
where L.SID = S.SID
and T1.OBJ# = decode(L.ID2,0,L.ID1,L.ID2)
and U1.USER# = T1.OWNER#
and S.TYPE != 'BACKGROUND'
order by 1,2,5 ;

```

上述SQL语句的输出结果如下：

Username	Term	Kill String	Table Name	Lock Hld	Lock Req
SAMUEL	ttyr6	7,5159	SAMUEL.TABLE_CONFIG	Row Excl	
YA-MEI	ttyr5	8,941	SCOTT.SCHEDULE	Row Excl	
BARRETT	ttyq3	6,8885	SYS.CLUSTERS	Exclusiv	
			SCOTT.ACCOUNT	Row Excl	

执行下述的命令，演示如何删除一个会话：

```
ALTER SYSTEM KILL SESSION '6,8885';
```

该命令将删除Barrett的已锁定的会话。

可以在\$ORACLE_HOME/rbms/admin目录中发现另外一个有用的脚本。此脚本被称为utllockt.sql，它能够以树格式打印锁信息。这使得信息容易阅读并且容易发现谁正在锁定资源。

25.4 避免锁：可能的解决策略

本节提供了一些避免锁定的通用练习。

有些应用使用控制表产生后续的序列号。一个典型的例子是在一个排序表中的后续序号。将有一个控制表存储所产生的最后一个序号。任何需要产生新序号的程序将从此表中选择所产生的最后一个排序号，然后递增它。在最后一个序号被递增后，它被更新回控制表中。

在一个多用户环境下，有许多会话试图进入数据库中的一个序列中，此时控制表会成为一个瓶颈。考虑一种情况，其中一个会话从控制表中选择了最后一个序号，在更新了递增的序号以后，由于某些原因在提交以前挂起；整个应用会因为在被挂起的会话后面形成的用户排队而停止。当一个会话把递增的序号更新到控制表，然后在提交以前执行一些辅助的处理时会发生由此类序号引起的另一个瓶颈。控制表记录被锁定的时间将是处理持续的时间。从控制表中捕获下一个序列号以后的辅助处理时间将极大地限制系统的吞吐量。

对于这些问题，有几种可能的解决方案：

减少把最后一个序号更新到控制表中和提交之间的时间；也就是说，一获得下一个序列号，就立刻把它提交给数据库，这样使其他会话能够对记录进行存取。

考虑使用序列。优点有很多。序列的使用极大地简化了应用代码。按照以前的逻辑，需要执行一个选择和一个更新。然而，在从序列中执行了选择以后，序列号自动递增，这样就有了输入/输出的有利条件。锁定瓶颈被完全消除，因为 Oracle在内部自动地为你递增了序列号。缺点是产生的最后一个号码在使用后不能被回滚，因此会引起空洞。

批处理作业固定地从一张正在被处理的表中选择大量的记录，在这些记录上执行一些处理，然后用新值更新它们。在程序执行处理以前锁定这些记录非常重要，因为当另一个会话到来并且更新记录时，批处理作业可能会从该表中选择过时的数值。因此，当批处理作业必须在处理结束时把结果更新回该表中时，它会发现另一个会话已经更新了该记录。要预防此类事情的发生，在选择记录时预先锁定它们。为此编写的 PL/SQL 伪代码如清单 25-7 所示。

清单 25-7 批处理伪代码

```
Begin
  Declare c_emp for
    Select * from emp
    <Where Clause >
  For Update;

  v_emp c_emp%RowType;

  Begin
  Open c_emp;
  Loop;
  Begin

    Fetch c_emp into v_emp;

    --Do some processing --
    Update c_emp
    Set sal = X,
        incr = Y
    Where emp_id = v_emp.emp_id;
    End;
  End Loop;
  Commit;
  Close c_emp;
  End ;
End ;
```

在此代码中，要被处理的整个记录集被锁定；在处理以前需要对记录进行锁定以便当处理这些记录时，另一个会话不能更新它们。此代码的缺陷是即使在代码得到处理后，它们仍然被锁定，直到所有的记录被处理完。假如处理时间很长的话，这些记录被锁定的这段时间会阻止别的会话对它们的并行存取。因为某一给定的时刻或许需要这些记录，所以此锁定方案可能会成为性能的瓶颈。

用来克服上述问题的替换代码在清单 25-8 中给出。

清单 25-8 用于取得最大并行度的批处理伪代码

```
Begin
  Declare c_emp for
    Select rowid from emp
    <Where Clause >;

  v_emp c_emp%RowType;
  r_emp emp%RowType;

  Begin
  Open c_emp;
  Loop;
  Begin

    Fetch c_emp into v_emp;

    Select *
    Into r_emp
    From emp
    Where rowid = v_emp.rowid;
    <Where Clause >
    For Update;
    --Do some processing --
    Update emp
```

```

        Set sal = X,
            incr = Y
        Where rowid = v_emp.rowid
        <Where Clause >;

    Commit;

End;
End Loop;

Close c_emp;
End ;

End ;

```

注意到在清单 25-8 中，外层游标仅选择该表中所有的行。外层游标把值反馈到内层选择中，内层选择使用 for update 子句每次仅锁定一行。清单 25-8 中的提交现在被移动到循环里面；因此，在修改了每一行后，进行提交，这样其他用户能够实现对表的最大并行存取。你或许想在清单 25-7 中的脚本的循环内部移动提交。不幸的是，假如在 Oracle 6.0 中完成此项工作的话，游标会关闭，造成 PL/SQL 循环的中断。只要游标被 open 子句打开，那么表中所有相关的行被锁定。当进行提交时，游标失去它的环境并且显示如下消息：

```
ORA-01002: fetch out of sequence
```

因此按前面显示的式样编写代码是必要的。

当使用含有行级触发器的表时要注意。例如，一张表可以有一个行级触发器，这样当该表的一条记录被更新时，该表中的触发器会使其他的表也得到更新。这会由于一条 update 语句而在系统中造成大量的锁。

用户培训有助于避免锁。有很多次我注意到那些认为他们的进程由于某些原因而挂起的用户为了复位该进程而关掉他们的终端（尤其是在客户 / 服务器应用中）。另一类用户是在让他们的终端处于注册状态时便去吃午饭或离开工作桌。这会锁住系统中的一些表并造成其他用户因为此用户而等待。用户需要明白的是关闭终端不能复位进程，该进程在服务器端仍然处于活动状态。类似地，用户应被要求将非活动的终端退出登录。这有助于增加并行性以及使用户不被锁住。

25.5 锁定与分布式数据库

当一个事务企图同时更新两个分布式数据库时，分布式数据库必须解决固有的更新问题，这被称为两步提交（2PC），可以用如下的 SQL 语句实现：

```

APPLY UPDATE A (DB1)
APPLY UPDATE B (DB2)
IF A = OK AND B = OK
    COMMIT A
    < = At this point deadly exposure occurs.
    COMMIT B
ELSE
    ROLLBACK A
    ROLLBACK B

```

这里，在成功地执行 UPDATE 语句之后，系统将把 COMMIT 语句发布到 A 与 B。当一个失败发生在 A 的 COMMIT 以后和 B 的 COMMIT 以前时会发生一个曝光点。由于没有自动恢复，所以它会造成完整性的重大损失。解决此问题的唯一办法是中断该事务并把更新人工回滚到 A 与 B。

如下的技术能够用来释放数据库中的锁：

使用命令删除持有 ALTER SYSTEM KILL SESSION 'sid ,serial#' 锁的会话;
请求锁持有者提交或回滚。

使用 kill -9 <UserID> 在 UNIX 提示符下删除用户标识。

在一个两步提交事务中, 使用 ROLLBACK FORCE 或 COMMIT FORCE。

从服务器管理器菜单模式或企业管理器中删除用户会话。

详细锁定信息脚本

清单 25-9 是一个详细锁定信息脚本, 该脚本提供了有关当前在数据库中持有的锁的完整译码信息。

清单 25-9 检索数据库锁的译码信息

```

set lines 200
set pagesize 66
break on Kill on sid on username on terminal
column Kill heading 'Kill String' format a13
column res heading 'Resource Type' format 999
column id1 format 9999990
column id2 format 9999990
column locking heading 'Lock Held/Lock Requested' format a40
column lmode heading 'Lock Held' format a20
column request heading 'Lock Requested' format a20
column serial# format 99999
column username format a10 heading "Username"
column terminal heading Term format a6
column tab format a30 heading "Table Name"
column owner format a9
column LAddr heading "ID1 - ID2" format a18
column Lockt heading "Lock Type" format a40
column command format a25
column sid format 990
rem      L.SID||', '||S.SERIAL# Kill,
select
nvl(S.USERNAME,'Internal') username,
      L.SID,
      nvl(S.TERMINAL,'None') terminal,
      decode(command,
0,'None',decode(l.id2,0,U1.NAME||'. '||substr(T1.NAME,1,20),'None')) tab,
decode(command,
0,'BACKGROUND',
1,'Create Table',
2,'INSERT',
3,'SELECT',
4,'CREATE CLUSTER',
5,'ALTER CLUSTER',
6,'UPDATE',
7,'DELETE',
8,'DROP',
9,'CREATE INDEX',
10,'DROP INDEX',
11,'ALTER INDEX',
12,'DROP TABLE',
13,'--',
14,'--',
15,'ALTER TABLE',
16,'--',
17,'GRANT',
18,'REVOKE',

```

```
19, 'CREATE SYNONYM',
20, 'DROP SYNONYM',
21, 'CREATE VIEW',
22, 'DROP VIEW',
23, '- ',
24, '- ',
25, '- ',
26, 'LOCK TABLE',
27, 'NO OPERATION',
28, 'RENAME',
29, 'COMMENT',
30, 'AUDIT',
31, 'NOAUDIT',
32, 'CREATE EXTERNAL DATABASE',
33, 'DROP EXTERNAL DATABASE',
34, 'CREATE DATABASE',
35, 'ALTER DATABASE',
36, 'CREATE ROLLBACK SEGMENT',
37, 'ALTER ROLLBACK SEGMENT',
38, 'DROP ROLLBACK SEGMENT',
39, 'CREATE TABLESPACE',
40, 'ALTER TABLESPACE',
41, 'DROP TABLESPACE',
42, 'ALTER SESSION',
43, 'ALTER USER',
44, 'COMMIT',
45, 'ROLLBACK',
46, 'SAVEPOINT',
47, 'PL/SQL EXECUTE',
48, 'SET TRANSACTION',
49, 'ALTER SYSTEM SWITCH LOG',
50, 'EXPLAIN',
51, 'CREATE USER',
52, 'CREATE ROLE',
53, 'DROP USER',
54, 'DROP ROLE',
55, 'SET ROLE',
56, 'CREATE SCHEMA',
57, 'CREATE CONTROL FILE',
58, 'ALTER TRACING',
59, 'CREATE TRIGGER',
60, 'ALTER TRIGGER',
61, 'DROP TRIGGER',
62, 'ANALYZE TABLE',
63, 'ANALYZE INDEX',
64, 'ANALYZE CLUSTER',
65, 'CREATE PROFILE',
66, 'DROP PROFILE',
67, 'ALTER PROFILE',
68, 'DROP PROCEDURE',
70, 'ALTER RESOURCE COST',
71, 'CREATE SNAPSHOT LOG',
72, 'ALTER SNAPSHOT LOG',
73, 'DROP SNAPSHOT LOG',
74, 'CREATE SNAPSHOT',
75, 'ALTER SNAPSHOT',
76, 'DROP SNAPSHOT',
84, '- ',
85, 'TRUNCATE TABLE',
86, 'TRUNCATE CLUSTER',
87, '- ',
88, 'ALTER VIEW',
89, '- ',
90, '- '.
```

```

91,'CREATE FUNCTION',
92,'ALTER FUNCTION',
93,'DROP FUNCTION',
94,'CREATE PACKAGE',
95,'ALTER PACKAGE',
96,'DROP PACKAGE',
97,'CREATE PACKAGE BODY',
98,'ALTER PACKAGE BODY',
99,'DROP PACKAGE BODY',
command||' - ???') COMMAND,
        decode(L.LMODE,1,'No Lock',
                2,'Row Share',
                3,'Row Exclusive',
                4,'Share',
                5,'Share Row Exclusive',
                6,'Exclusive','NONE') lmode,
        decode(L.REQUEST,1,'No Lock',
                2,'Row Share',
                3,'Row Exclusive',
                4,'Share',
                5,'Share Row Exclusive',
                6,'Exclusive','NONE') request,
l.id1||' - '||l.id2 Laddr,
l.type||' - '||
decode(l.type,
'BL','Buffer hash table instance lock',
'CF','Cross-instance function invocation instance lock',
'CI','Control file schema global enqueue lock',
'CS','Control file schema global enqueue lock',
'DF','Data file instance lock',
'DM','Mount/startup db primary/secondary instance lock',
'DR','Distributed recovery process lock',
'DX','Distributed transaction entry lock',
'FI','SGA open-file information lock',
'FS','File set lock',
'IR','Instance recovery serialization global enqueue lock',
'IV','Library cache invalidation instance lock',
'MB','Master buffer hash table instance lock',
'MM','Mount definition global enqueue lock',
'MR','Media recovery lock',
'RE','USE_ROW_ENQUEUE enforcement lock',
'RT','Redo thread global enqueue lock',
'RW','Row wait enqueue lock',
'SC','System commit number instance lock',
'SH','System commit number high water mark enqueue lock',
'SN','Sequence number instance lock',
'SQ','Sequence number enqueue lock',
'ST','Space transaction enqueue lock',
'SV','Sequence number value lock',
'TA','Generic enqueue lock',
'TD','DDL enqueue lock',
'TE','Extend-segment enqueue lock',
'TM','DML enqueue lock',
'TT','Temporary table enqueue lock',
'TX','Transaction enqueue lock',
'UL','User supplied lock',
'UN','User name lock',
'WL','Being-written redo log instance lock',
'WS','Write-atomic-log-switch global enqueue lock',
'TS',decode(l.id2,0,'Temporary segment enqueue lock (ID2=0)',
            'New block allocation enqueue lock (ID2=1)'),
'LA','Library cache lock instance lock (A=namespace)',
'LB','Library cache lock instance lock (B=namespace)',
'LC','Library cache lock instance lock (C=namespace)',

```

```
'LD','Library cache lock instance lock (D=namespace)',
'LE','Library cache lock instance lock (E=namespace)',
'LF','Library cache lock instance lock (F=namespace)',
'LG','Library cache lock instance lock (G=namespace)',
'LH','Library cache lock instance lock (H=namespace)',
'LI','Library cache lock instance lock (I=namespace)',
'LJ','Library cache lock instance lock (J=namespace)',
'LK','Library cache lock instance lock (K=namespace)',
'LL','Library cache lock instance lock (L=namespace)',
'LM','Library cache lock instance lock (M=namespace)',
'LN','Library cache lock instance lock (N=namespace)',
'LO','Library cache lock instance lock (O=namespace)',
'LP','Library cache lock instance lock (P=namespace)',
'LS','Log start/log switch enqueue lock',
'PA','Library cache pin instance lock (A=namespace)',
'PB','Library cache pin instance lock (B=namespace)',
'PC','Library cache pin instance lock (C=namespace)',
'PD','Library cache pin instance lock (D=namespace)',
'PE','Library cache pin instance lock (E=namespace)',
'PF','Library cache pin instance lock (F=namespace)',
'PG','Library cache pin instance lock (G=namespace)',
'PH','Library cache pin instance lock (H=namespace)',
'PI','Library cache pin instance lock (I=namespace)',
'PJ','Library cache pin instance lock (J=namespace)',
'PK','Library cache pin instance lock (K=namespace)',
'PL','Library cache pin instance lock (L=namespace)',
'PM','Library cache pin instance lock (M=namespace)',
'PN','Library cache pin instance lock (N=namespace)',
'PO','Library cache pin instance lock (O=namespace)',
'PP','Library cache pin instance lock (P=namespace)',
'PQ','Library cache pin instance lock (Q=namespace)',
'PR','Library cache pin instance lock (R=namespace)',
'PS','Library cache pin instance lock (S=namespace)',
'PT','Library cache pin instance lock (T=namespace)',
'PU','Library cache pin instance lock (U=namespace)',
'PV','Library cache pin instance lock (V=namespace)',
'PW','Library cache pin instance lock (W=namespace)',
'PX','Library cache pin instance lock (X=namespace)',
'PY','Library cache pin instance lock (Y=namespace)',
'PZ','Library cache pin instance lock (Z=namespace)',
'QA','Row cache instance lock (A=cache)',
'QB','Row cache instance lock (B=cache)',
'QC','Row cache instance lock (C=cache)',
'QD','Row cache instance lock (D=cache)',
'QE','Row cache instance lock (E=cache)',
'QF','Row cache instance lock (F=cache)',
'QG','Row cache instance lock (G=cache)',
'QH','Row cache instance lock (H=cache)',
'QI','Row cache instance lock (I=cache)',
'QJ','Row cache instance lock (J=cache)',
'QK','Row cache instance lock (K=cache)',
'QL','Row cache instance lock (L=cache)',
'QM','Row cache instance lock (M=cache)',
'QN','Row cache instance lock (N=cache)',
'QO','Row cache instance lock (O=cache)',
'QP','Row cache instance lock (P=cache)',
'QQ','Row cache instance lock (Q=cache)',
'QR','Row cache instance lock (R=cache)',
'QS','Row cache instance lock (S=cache)',
'QT','Row cache instance lock (T=cache)',
'QU','Row cache instance lock (U=cache)',
'QV','Row cache instance lock (V=cache)',
'QW','Row cache instance lock (W=cache)',
'QX','Row cache instance lock (X=cache)',
```

```
'QY','Row cache instance lock (Y=cache)',  
'QZ','Row cache instance lock (Z=cache)', '????') Lockt  
from   V$LOCK L,  
        V$SESSION S,  
        SYS.USER$ U1,  
        SYS.OBJ$ T1  
where  L.SID = S.SID  
and    T1.OBJ# = decode(L.ID2,0,L.ID1,1)  
and    U1.USER# = T1.OWNER#  
and    S.TYPE != 'BACKGROUND'  
order by 1,2,5  
/
```

25.6 使用门实现锁

Oracle使用在以前的小节中提到过的各种技术实现锁。使用门是其中的一种方法。门与锁不同，它在内部获得并由Oracle释放。不能通过发布一条语句明确地获得门。一般，门用于控制对共享代码路径的存取。门是一种能够被快速获得或释放的锁。门用来防止一个以上的进程在一个特定的时间里执行相同的代码段。

25.6.1 门的功能

门用于控制对共享结构的存取。可以在操作系统层上使用信号量实现门。门是在非常短时间内持有的锁。

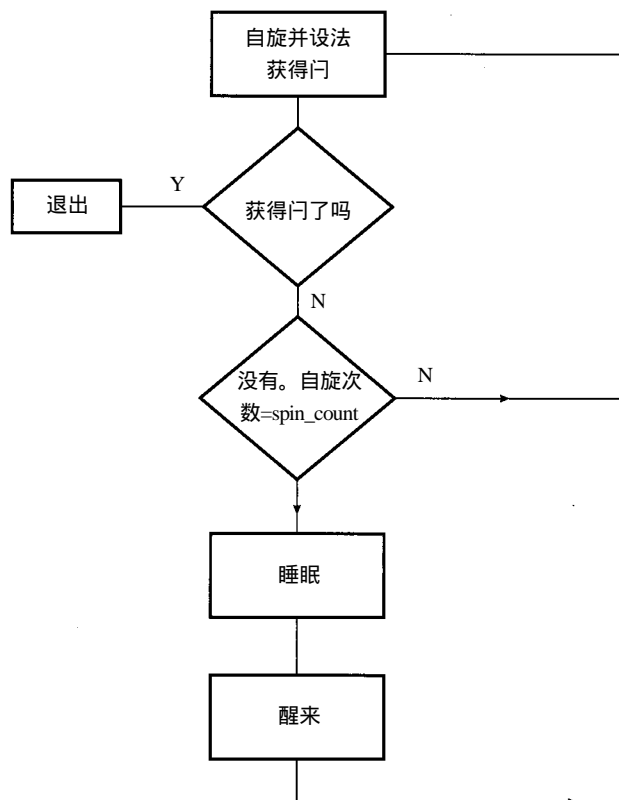


图25-1 在一个门上的进程自旋

一个进程在能够对受门保护的共享结构存取以前，它首先必须获得该门。此门当前可能是空闲的（也就是说，没有其他的进程正在持有该门），在这种情况下此进程立刻能得到该门。此进程将在需要门的时候持有它，然后在不需要时便释放它。这是最简单和典型的情况。

考虑一种情况，在该情况下一个进程已经获得了某个门。另一个进程根据它正试图获得的门的类型，会获取有两个选项的门。有两种与门有关的类型：

立刻。

可以等待。

假如一个进程试图在立刻模式下获得门，而该门已经被另外一个进程所持有，如果该门不能立刻可用的话，那么该进程就不会为获得该门而等待。它将继续执行另一个操作。

在可以等待模式下，假如一个进程在第一次尝试中没有获得该门，那么它会等待并且再尝试一次。如果系统有多个 CPU，此未成功的进程将开始围绕该门自旋并设法获得它。该进程围绕该门自旋的次数在 init.ora 文件的参数 spin_count 中定义。每一次自旋，它都设法获得该门；如果它不能获得该门，那么它就再次自旋并继续下去，直到自旋数达到 spin_count 所规定的数值。然后该进程转入睡眠状态，持续一段指定长度的时间，然后再次醒来，按顺序重复以前的步骤，如图 25-1 所示。

25.6.2 分析与门相关的视图

本节讨论数据字典视图，该视图能够用来获得系统中有关门的更多信息。在这里仅讨论这些视图中的重要列。V\$latch 与 V\$latchholder 是两个与门有关的非常重要的视图：

V\$latch——本视图包含与系统中各种门的性能有关的所有重要的统计数字。表 25-5 给出了对视图 V\$latch 的重要列的描述。

V\$latchholder——假如系统当前存在门争用问题，本视图能够用来确定哪一个会话当前拥有该门。

表25-5 V\$latch

列 名	描 述
immediate_gets	立刻获得的门的成功请求次数（仅限立刻模式）
immediate_misses	没有立刻获得的门的未成功请求次数（仅限立刻模式）
gets	甘愿等待而获得的门的成功请求次数
misses	甘愿等待而获得的门的未成功请求次数
sleeps	一个甘愿等待门的进程请求门的次数并且当该进程没有得到门时，必须转入睡眠状态
spin_gets	不必睡眠，而仅仅自旋就获得门的次数
sleep1~10	每当进程睡眠时，该列中的值就递增——例如，假如一个进程在获得门以前不得不睡眠了3次的话，那么 sleep3 列将增1

25.6.3 门竞争约束

假如系统正面临门竞争问题，那么清单 25-10 和清单 25-11 中的脚本能够用来发现哪一个门受到严重地打击。所有的脚本查询 V\$latch 表以便找到门的细节。下面显示的是对 V\$latch 的描述：

ADDR	RAW(4)
LATCH#	NUMBER
LEVEL#	NUMBER
NAME	VARCHAR2(64)

GETS	NUMBER
MISSES	NUMBER
SLEEPS	NUMBER
IMMEDIATE_GETS	NUMBER
IMMEDIATE_MISSES	NUMBER
WAITERS_WOKEN	NUMBER
WAITS_HOLDING_LATCH	NUMBER
SPIN_GETS	NUMBER
SLEEP1	NUMBER
SLEEP2	NUMBER
SLEEP3	NUMBER
SLEEP4	NUMBER
SLEEP5	NUMBER
SLEEP6	NUMBER
SLEEP7	NUMBER
SLEEP8	NUMBER
SLEEP9	NUMBER
SLEEP10	NUMBER
SLEEP11	NUMBER

下面的脚本将在SQL*Plus中运行。

清单25-10 用于鉴别系统中的门性能脚本

```
col name heading "Name" format a20
col pid heading "HSid" format a3
col gets heading "Gets" format 999999990
col misses heading "Miss" format 99990
col im_gets heading "ImG" format 99999990
col im_misses heading "ImM" format 99990
col sleeps heading "Sleeps" format 99990

select n.name name, h.pid pid, l.gets gets, l.misses misses,
l.immediate_gets im_gets, l.immediate_misses im_misses, l.sleeps sleeps
from v$latchname n, v$latchholder h, v$latch l
where l.latch# = n.latch#
and l.addr = h.laddr(+)
```

上述SQL语句的输出结果如下所示：

Name	HSid	Gets	Miss	ImG	ImM	Sleeps
cached attr list		0	0	0	0	0
modify parameter values		931	0	0	0	0
messages		736117	44	0	0	80
enqueue hash chains		1257100	145	0	0	163
trace latch		0	0	0	0	0
cache buffers lru chains	197351761	2092	66288289	20919	4020	
cache buffer handles		2348	0	0	0	0
multiblock read objects		2139864	700	1	0	900
cache protection latch		0	0	0	0	0
shared pool		1261081	207	0	0	208
library cache		7867103	11803	376	0	11854
redo allocation		1306657	501	0	0	760
redo copy		34	0	463290	71	47

在此输出中，HSID列指示了当前持有门的会话的标识。假如 misses与 gets之比超过1%，那么该门需要引起注意。

使用V\$session wait视图是另外一种用于检查一个长时间挂起的会话是否正在等待一个门的方法，可以使用清单25-11中的查询。

清单25-11 使用V\$session wait视图来鉴别门竞争

```
select event, p1text,p1 ,p2text,p2 ,seq#,wait_time,state
```

```
from v$session_wait
where   sid = '&&1'
and event = 'latch free';
```

上述SQL语句的输出结果如下所示：

EVENT	P1TEXT	P1	P2TEXT	P2	SEQ#	WAIT_TIME	STATE
latch free	address	3	number	11	1181	0	WAITING

假如wait_time列的值为零，表明该会话当前正等待门。

P2列显示了门的数量。可以使用V\$latchname得到门的名称：

```
Select name
From v$latchname
Where latch# = 11;
cache buffers chains
```

清单25-12是一个报告生成脚本，该脚本列举了用于确定一个数据库实例是否正经历门竞争的关键信息。门竞争率应当小于或等于1%。

清单25-12 列举用于确定门竞争的信息

```
tttitle -
      center  'Latch Contention Report'  skip 3

col name form A25
col gets form 999,999,999
col misses form 999.99
col spins form 999.99
col igets form 999,999,999
col imisses form 999.99

select name,
       gets,
       misses*100/decode(gets,0,1,gets) misses,
       spin_gets*100/decode(misses,0,1,misses) spins,
       immediate_gets igets,
       immediate_misses*100/decode(immediate_gets,0,1,immediate_gets) imisses
from v$latch
order by gets + immediate_gets
/
```

上述SQL语句的输出结果如下所示：

Latch Contention Report					
NAME	GETS	MISSSES	SPINS	IGETS	IMISSES
cached attr list	0	.00	.00	0	.00
trace latch	0	.00	.00	0	.00
cache protection latch	0	.00	.00	0	.00
KCL freelist latch	0	.00	.00	0	.00
redo copy	0	.00	.00	0	.00
archive control	0	.00	.00	0	.00
KCL name table latch	0	.00	.00	0	.00
lock element parent latch	0	.00	.00	0	.00
loader state object freelist	0	.00	.00	0	.00
process queue	0	.00	.00	0	.00
error message lists	0	.00	.00	0	.00
query server freelists	0	.00	.00	0	.00
query server process	0	.00	.00	0	.00
virtual circuits	0	.00	.00	0	.00

virtual circuit queues	0	.00	.00	0	.00
virtual circuit buffers	0	.00	.00	0	.00
global tx hash mapping	0	.00	.00	0	.00
device information	0	.00	.00	0	.00
parallel query alloc buffer	0	.00	.00	0	.00
parallel query stats	0	.00	.00	0	.00
process queue reference	0	.00	.00	0	.00
global transaction	0	.00	.00	0	.00
global tx free list	0	.00	.00	0	.00
cost function	0	.00	.00	0	.00
instance latch	0	.00	.00	0	.00
NLS data objects	1	.00	.00	0	.00
cache buffer handles	42	.00	.00	0	.00
latch wait list	70	.00	.00	0	.00
multiblock read objects	124	.00	.00	0	.00
library cache load lock	326	.00	.00	0	.00
sort extent pool	867	.00	.00	0	.00
ktm global data	867	.00	.00	0	.00
process allocation	680	.00	.00	680	.00
sequence cache	2,128	.00	.00	0	.00
user lock	2,676	.00	.00	0	.00
session switching	4,120	.00	.00	0	.00
modify parameter values	5,005	.00	.00	0	.00
cache buffers lru chain	41,770	.00	.00	31,613	.00
dml lock allocation	116,700	.00	.00	0	.00
transaction allocation	177,154	.00	.00	0	.00
undo global data	206,727	.00	.00	0	.00
enqueue hash chains	271,775	.00	.00	0	.00
messages	592,041	.00	.00	0	.00
shared pool	860,906	.00	.00	0	.00
enqueues	1,151,391	.00	.00	0	.00
list of block allocation	1,558,054	.00	.00	0	.00
session allocation	1,688,085	.00	.00	0	.00
system commit number	1,859,249	.00	.00	0	.00
redo allocation	2,009,258	.01	.00	0	.00
row cache objects	2,501,694	.00	.00	42	.00
session idle bit	4,167,221	.00	.00	0	.00
library cache	13,734,618	.00	.00	41	.00
cache buffers chains	43,694,786	.00	.00	6,832,121	.00

清单25-13是一个给出门的各种睡眠率的脚本。

清单25-13 检索门睡眠率

```
col name form A18 trunc

col gets form 999,999,990
col miss form 90.9
col cspins form A6 heading 'spin|s106'
col csleep1 form A5 heading 's101|s107'
col csleep2 form A5 heading 's102|s108'
col csleep3 form A5 heading 's103|s109'
col csleep4 form A5 heading 's104|s110'
col csleep5 form A5 heading 's105|s111'
col Interval form A12
set recsep off

select a.name,
       a.gets gets,
       a.misses*100/decode(a.gets,0,1,a.gets) miss,
       to_char(a.spin_gets*100/decode(a.misses,0,1,a.misses),'990.9')||
       to_char(a.sleep6*100/decode(a.misses,0,1,a.misses),'90.9') cspins,
       to_char(a.sleep1*100/decode(a.misses,0,1,a.misses),'90.9')||
       to_char(a.sleep7*100/decode(a.misses,0,1,a.misses),'90.9') csleep1,
```

```

to_char(a.sleep2*100/decode(a.misses,0,1,a.misses),'90.9')||
to_char(a.sleep8*100/decode(a.misses,0,1,a.misses),'90.9') csleep2,
to_char(a.sleep3*100/decode(a.misses,0,1,a.misses),'90.9')||
to_char(a.sleep9*100/decode(a.misses,0,1,a.misses),'90.9') csleep3,
to_char(a.sleep4*100/decode(a.misses,0,1,a.misses),'90.9')||
to_char(a.sleep10*100/decode(a.misses,0,1,a.misses),'90.9') csleep4,
to_char(a.sleep5*100/decode(a.misses,0,1,a.misses),'90.9')||
to_char(a.sleep11*100/decode(a.misses,0,1,a.misses),'90.9') csleep5
from v$latch a
where a.misses <> 0
order by 2 desc
/

```

The output from the SQL statement above is as follows:

```

spin  sl01  sl02  sl03  sl04  sl05
NAME                                GETS  MISS  sl06   sl07  sl08  sl09  sl10  sl11
-----
cache buffers chain  43,696,090  0.0    0.0 ##### 0.0  0.0  0.0  0.0
                        0.0    0.0  0.0  0.0  0.0  0.0
library cache        13,738,029  0.0    0.0  97.1  0.0  2.9  0.0  0.0
                        0.0    0.0  0.0  0.0  0.0  0.0
session idle bit     4,167,650  0.0    0.0 ##### 0.0  0.0  0.0  0.0
                        0.0    0.0  0.0  0.0  0.0  0.0
row cache objects    2,502,963  0.0    0.0 ##### 0.0  0.0  0.0  0.0
                        0.0    0.0  0.0  0.0  0.0  0.0
redo allocation      2,010,038  0.0    0.0 ##### 0.0  0.0  0.0  0.0
                        0.0    0.0  0.0  0.0  0.0  0.0
system commit numb   1,859,423  0.0    0.0 ##### 0.0  0.0  0.0  0.0
                        0.0    0.0  0.0  0.0  0.0  0.0
session allocation   1,688,269  0.0    0.0 ##### 0.0  0.0  0.0  0.0
                        0.0    0.0  0.0  0.0  0.0  0.0
enqueuees            1,152,139  0.0    0.0 ##### 0.0  0.0  0.0  0.0
                        0.0    0.0  0.0  0.0  0.0  0.0
messages              595,209  0.0    0.0 ##### 0.0  0.0  0.0  0.0
0.0    0.0  0.0  0.0  0.0  0.0
enqueue hash chain   272,103  0.0    0.0  0.0  0.0 ##### 0.0  0.0
0.0    0.0  0.0  0.0  0.0  0.0
cache buffers lru     41,778  0.0    0.0 ##### 0.0  0.0  0.0  0.0
                        0.0    0.0  0.0  0.0  0.0  0.0

```

调整一些重要的门

在清单 25-14 中显示的所有门中，可以采用几个重要的和正确的步骤来调整它们。本节中将要讨论的门有 cache buffers lru chain、redo allocation、redo copy 和 library cache。

清单 25-14 列举所有的门

LATCH#	NAME
0	latch wait list
1	process allocation
2	session allocation
3	session switching
4	session idle bit
5	cached attr list
6	modify parameter values
7	messages
8	enqueuees
9	enqueue hash chains
10	trace latch
11	cache buffers chains
12	cache buffer handles
13	multiblock read objects
14	cache protection latch

```
15 cache buffers lru chain
16 system commit number
17 archive control
18 redo allocation
19 redo copy
20 KCL freelist latch
21 KCL name table latch
22 instance latch
23 lock element parent latch
24 loader state object freelist
25 dml lock allocation
26 list of block allocation
27 transaction allocation
28 sort extent pool
29 undo global data
30 ktm global data
31 sequence cache
32 row cache objects
33 cost function
34 user lock
35 global tx free list
36 global transaction
37 global tx hash mapping
38 shared pool
39 library cache
40 library cache load lock
41 virtual circuit buffers
42 virtual circuit queues
43 virtual circuits
44 NLS data objects
45 query server process
46 query server freelists
47 error message lists
48 process queue
49 process queue reference
50 parallel query stats
51 parallel query alloc buffer
52 device information
```

- 高速缓存缓冲区最近最少使用链 高速缓存缓冲区最近最少使用链负责保护指向高速缓存中的数据库块缓冲区的存取路径。由 init.ora文件的db_block_buffers参数定义在SGA中的高速缓存的容量并且包含了已保存的从数据文件中读出的数据拷贝。当进程需要读数据时，由于某些进程读入的数据而使此缓冲区中存在数据，这使得读操作变成非常有效率。

高速缓存分为两个列表：脏列表与LRU列表。脏列表（dirty list）包含那些已经被修改但还没有被写入到磁盘中的缓冲区。LRU列表由被钉住的缓冲区、还没有被移动到磁盘中的脏缓冲区和空闲缓冲区组成。被钉住的缓冲区（pinned buffer）是指那些当前被其他进程存取的缓冲区。脏缓冲区（dirty buffer）包含那些没有被写入到磁盘中的缓冲区，这些缓冲区随后移动到脏列表中。空闲缓冲区（free buffer）是指那些可供使用的缓冲区。

当一个进程需要从磁盘中读取已经不在高速缓存中的数据时，它需要一个空闲缓冲区用于读取新数据。它搜索LRU列表以寻找空闲缓冲区。假如有过多的对高速缓存中的空闲缓冲区的需求，那么对LRU列表的大量存取会引发对高速缓存缓冲区LRU链的争用。

对门的争用可以通过使用 init.ora参数db_block_lru_latches来减到最小，此参数可以在Oracle 8中使用。通过增加db_block_lru_latches参数的值，可将对门的争用减到最小。此参数

的最大值是CPU数量的两倍。

对此争用的基本原因是大量地请求空闲缓冲区。你可以优化此 SQL 语句以把对空闲缓冲区的高请求减到最小或增加 db_block_buffer 参数的值以增加系统中可用的空闲缓冲区的数量。

注意 注意SGA必须装入实内存的一个连续块中，所以说，如果增加缓冲区高速缓存的话，你必须确保系统中有足够的连续可用内存。

- 重做分配和重做拷贝 重做分配 (redo allocation) 和重做拷贝 (redo copy) 控制对重做日志缓冲区的写访问。当某个进程需要写重做日志缓冲区时，该进程将获得这两种门中的一个。假如写入到重做日志缓冲区的重做日志信息的容量小于 log_small_entry_max_size 参数的值，那么该进程将使用重做分配门。假如信息的容量大于该值，那么使用重做拷贝门拷贝该进程。

一种用于检查重做日志缓冲区中是否存在争用的快速方法是检查是否存在与写重做日志缓冲区有关的任何等待。这可以通过使用系统视图 V\$sysstat 来实现：

```
Select name, value
From v$sysstat
Where name = 'redo log space requests';
```

上述SQL语句的输出结果如下所示：

Name	Value
redo log space requests	12

假如等待数量太高的话，必须增加重做日志缓冲区的容量。

- 争用重做分配门 在一个多CPU系统中通过强迫进程使用重做拷贝门能够减少对重做分配门的争用。因为可以有多个重做拷贝门，所以拷贝会更有效地被执行。重做拷贝门的数量由 init.ora 参数 log_simultaneous_copies 定义。系统中可用门的最大数量是 CPU 数量的两倍。对于一个单 CPU 系统来说，该值为 0 并且将使用重做分配门。假如存在一个对重做分配门的争用，那么可以减小 log_small_entry_max_size 参数的当前值以便使用重做拷贝门。
- 争用重做拷贝门 假如系统正面对重做拷贝门的争用，可以通过增加 log_small_entry_max_size 参数的值（以便使用重做分配门）或者增加 log_simultaneous_copies 参数的值（以便增加可用的重做拷贝门的数量）来减少这种争用。

可以增加 init.ora 文件参数 log_entry_prebuild_threshold 的值以便被写入重做日志缓冲区中的数据被分组或写出。通过增加该参数，可以对许多写操作进行分组以便它们能够用一个操作写出，这样便减少了对这些门的请求，从而减少了争用。

- 库高速缓存门 该门主要与控制对库高速缓存的存取有关。库高速缓存包含共享 SQL 区、专用的 SQL 区、PL/SQL 过程包以及其他的控制结构。共享 SQL 区包含在多个会话之间共享的 SQL 代码。通过增加对这些 SQL 代码的共享，可以避免对此门的争用。

当存在大量的对库高速缓存中空间的请求时会发生对此门的争用。系统中大量的语法分析以及由于进程之间的低共享而产生的大量对打开一个新游标的请求是常见的引发对此门争用的原因。

通过使用可以被多个会话共享的代码可以避免对此门的争用。例如，RDBMS 以不同的方

式处理下面的两条SQL语句：

```
select name from employee where emp_id = 100;  
Select name from employee where emp_id = 100;
```

尽管两条SQL语句看起来相同，但是这两条语句的哈希值不同。上面那条SQL语句的“select”中的“s”是小写字母，而下面那条SQL语句中用的是大写的S。因此，当发出这两条语句时，RDBMS将把它们作为不同的语句分别进行语法分析，从而引起门的负载。通过应用编码标准可以避免此类情况，在编码标准中实现缩进与格式标准，以便所编写的每条SQL语句尽可能有相同的散列值。注意，就连放入更多的空格都会使一条select语句被认为是另一条而引起增加语法分析。

在SQL语句中使用绑定变量能够增加对同一子句的共享。例如，考虑两种不使用绑定变量的情况：

```
Select sal from employee where emp_id = 100;  
Select sal from employee where emp_id = 200;
```

现在，对它们使用绑定变量：

```
Select sal from employee where emp_id := emp_id;
```

通过使用绑定变量，在第一种情况下的前两条select语句都能够共享存储在库缓存中的相同的select语句，这样减少了不必要的语法分析和门的负载。使用绑定变量使在共享池中不致于形成同一条select语句的多个拷贝。

可以在共享池中减少语法分析或者钉住诸如过程与包等频繁使用的对象来减少语法分析。钉住这些对象的优点是这些对象决不会被从共享池中清除并且缩减了随后的语法分析时间。可以通过使用如下的查询识别那些频繁使用的对象：

```
Select name ,executions  
From v$db_object_cache  
Where executions > <threshold limit>  
order by 2 desc;
```

可以使用如下语句将这些频繁使用的对象钉在共享池中：

```
dbms_share_pool.keep('object_name','p');
```

要在共享池中检查没有被钉住的对象，执行如下的查询：

```
Select name,type,kept,sharable_mem  
From v$db_object_cache  
Where kept = 'NO'  
Order by sharable_mem desc;
```

共享池的碎片也会引发大量对这些门的需求。已碎片化的共享池意指共享池中可用的净空闲内存可能很大，但是可用的总的连续内存并不大。因此，一个具有非常高的内存需求的对象（例如大型PL/SQL对象）被定位在共享池中会使已分配内存区的许多小字节块被刷新掉。当一个用户再次请求相关的SQL语句时，这些语句必须再次从语法上得到分析。引起碎片的主要原因是大型PL/SQL对象。通过使用绑定变量以及使用上述提及的技术钉住大型PL/SQL对象以增加SQL语句间的共享来避免碎片的形成。