第 5 章　分布式事务处理

事务（transaction）是数据库系统中保证一致性与可靠计算的基本单元。因此，一旦确定了查询的执行策略，查询就会作为事务执行，并转换为数据库的原子操作。事务确保在对同一数据项进行并发访问（其中至少有一个更新操作）以及发生故障时，数据库可以保持一致性（consistency）和持久性（durability）。

一致（consistent）与可靠（reliable）这两个概念需要更准确地定义。这里首先区分数据库一致性（database consistency）与事务一致性（transaction consistency）这两个概念。

我们说一个数据库处于一致状态，如果该数据库服从在其上定义的所有一致性（完整性）约束，详见第 3 章。修改、插入、删除（这三者统称为更新）都会造成状态的改变。当然，我们要确保数据库不会进入不一致的状态。请注意，在事务执行期间，数据库可能（事实上经常如此）暂时变得不一致，但重点在于当事务执行完毕之后数据库应该是一致的，如图 5.1 所示。

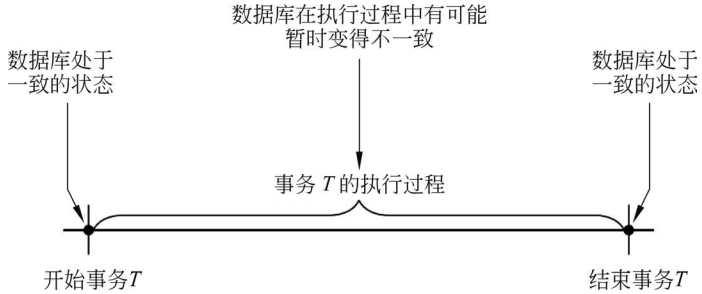

图 5.1　一个事务模型

另一方面，事务一致性指的是并发事务涉及的操作。我们希望数据库能在即使有多个用户同时访问（读或写）的时候保持一致状态。

可靠性是指系统对各种故障的适应（resiliency）能力及其从这些故障中恢复（recover）的能力。一个具有适应能力的系统可以容忍系统故障——即使发生故障，也能继续提供服务。一个具有恢复能力的 DBMS 能够在发生各种类型的故障后恢复到一致状态，即回退到先前的一致状态或前进到新的一致状态。

事务管理的任务是保证数据库始终处于一致状态，即使是在并发访问和故障发生时。并发事务管理是集中式 DBMS 的经典问题，在许多教科书中都有讲解。本章主要探讨分布式 DBMS 环境下的并发事务管理，重点介绍分布式并发控制以及分布式可靠性和故障恢复问题。5.1 节提供一个简单的复习，希望读者熟悉本科数据库课程与教科书中常见的基本事务管理概念和技术，附录 C（在线英文版）将更详细地探讨事务处理的概念。数据复制（data replication）不在本章的考虑范围，这一问题将会留到下一章。DBMS 通常分为在线事务处理（OLTP）或在线分析处理（OLAP）。在线事务处理应用程序，例如机票预订或银行系统，是面向高吞吐事务场景的，因此需要重点解决数据控制与可用性、多用户高吞吐量，以及

可预测的快速响应时间等问题。相比之下，在线分析处理应用程序，例如趋势分析或预测，主要分析来自多个业务数据库的历史汇总数据，需要针对规模较大的关系表处理复杂的查询。大多数 OLAP 应用程序不需要基于最新数据，因此不需要直接访问最新的业务数据。本章主要关注 OLTP 系统，OLAP 系统会在第 7 章进行介绍。

本章的组织如下。5.1 节简要介绍本章中使用的基本概念，并重新审视第 1 章中定义的架构模型，侧重探讨如何为支持事务管理而做修改。5.2 节深入探讨基于可串行化理论（serializability）的分布式并发控制技术。5.3 节介绍基于快照隔离（snapshot isolation）的并发控制技术。5.4 节讨论分布式可靠性技术，包括分布式提交、终结和恢复协议。

5.1　背景与概念定义

本节旨在简要介绍本章所使用的概念和术语。如前所述，本节不会对这些基本概念做深入讲解——这些可以在附录 C（在线英文版）中找到——而是介绍对理解本章其余部分有帮助的基本概念。此外，本节还将讨论如何修改系统架构以适应事务。

如前所述，事务是保证一致且可靠计算的基本单元。每个事务都以 Begin_transaction 命令开始，包含一系列的 Read（读）和 Write（写）操作，并以 Commit（提交）或 Abort（取消）结束。提交操作，一旦被处理就会确保事务对数据库所做的更新是永久的；而取消操作则会撤销事务中的所有操作，因此就数据库而言，就好像该事务从未执行过一样。事务会读取或者写入某些数据，这些数据构成了事务最基本的特性。事务读取的数据项集合构成了它的读集（read set，RS）；事务写入的数据项构成了它的写集（write set，WS）。一个事务的读集和写集的并集构成了它的基集，即$BS = RS \cup WS$。

典型的 DBMS 事务服务提供被称为 ACID 的特性：

（1）原子性（Atomicity）保证事务的执行是原子的，即事务的所有操作要么全被执行，要么就一个都不执行。

（2）一致性（Consistency）是指事务可以正确执行，也就是说，事务的代码能够正确地将数据库从一个一致状态变换到另一个一致状态。

（3）隔离性（Isolation）是指并发的事务之间彼此不可见，直到它们提交为止。隔离性是保证并发事务满足正确性的前提，即多个事务的并发执行不会破坏数据库的一致性。

（4）持久性（Durability）指的是，如果一个事务已经提交，那么它产生的结果就是永久的，并且不受系统故障的影响。

5.2 节介绍的并发控制算法是为了保证隔离性的，即并发的事务会看到一个一致的数据库状态，而且在事务提交后，数据库的状态依旧是一致的。5.4 节介绍的可靠性技术是为了保证原子性和持久性的。一致性（即保证某个事务不会造成数据库出错）通常是由第 3 章介绍的完整性约束保证的。

并发控制算法实现了“正确并发执行”这一概念。最常见的正确性定义是可串行化，它要求由一些事务并发执行生成的历史等价于某个串行（serial）的历史（即这些事务一个接一个地顺序执行）。鉴于事务将一个一致的数据库状态映射到另一个一致的数据库状态，根据定义，任何串行顺序都是正确的。因此，如果并发执行历史等价于任何一个串行顺序，

它也一定是正确的。5.3 节引入了一个更宽松的正确性概念，称为快照隔离（snapshot isolation, SI）。基本上，并发控制算法关注的是如何在并发事务之间有效保证不同的隔离级别。

当事务提交时，其操作需要永久化。这就要求系统管理事务日志，以记录事务的每个操作。提交协议（commit protocols）确保数据库更新和日志保存到持久化存储中，以保证永久保留。另一方面，取消协议（abort protocols）使用日志从数据库中清除已取消事务的所有操作。日志还可以用于系统出现故障时，用来将数据库带回到一致的状态。

在仅包含只读查询的 DBMS 负载基础上引入事务之后，我们需要再对第 1 章中介绍的架构模型做进一步讨论——需要扩展分布式执行监控程序的角色。

分布式执行监控程序包含两个模块：事务管理程序（transaction manager, TM）和调度程序（scheduler, SC）。事务管理程序负责代表应用程序协调数据库中各个操作的执行，而调度程序负责执行特定的并发控制算法，以同步各个操作对数据库的访问。

分布式事务管理中涉及的第三个模块是局部恢复管理程序（local recovery manager, LRM），其功能是在每一个数据库站点中，实现将局部数据库从故障状态恢复到一致状态。

每一个事务都会发起于某一个站点，我们称其为发起站点。发起站点的 TM 负责协调事务中所有数据库操作的执行，因此称发起站点的 TM 为协调者或协调 TM。

事务管理程序为应用程序提供接口，包括之前提到的一系列事务命令 Begin_transaction（事务开始）、Read（读）、Write（写）、Commit（提交）和 Abort（取消）。接下来我们在抽象层次上介绍这些命令在无复制的（non-replicated）分布式 DBMS 中的处理流程。简单起见，本节主要关注 TM 的接口，更多的细节会在后续章节给出介绍。

（1）Begin_transaction（事务开始）：这是给协调 TM 一个信号，表示一个新事务的开始。协调 TM 会在内存日志（称为易失日志，volatile log）中做一些记录，包括事务名称、发起这个事务的应用程序，等等。

（2）Read（读）：如果即将读取的数据项在局部存储，则读取其值并将其返回给事务。否则，协调 TM 定位到这个数据项，并通过恰当的并发控制方法请求存储它的站点返回其值。同时，存储数据项的站点会在易失日志中添加一条日志记录。

（3）Write（写）：如果即将写入的数据项在局部存储，则 TM 在数据处理程序的协助下更新其值。否则，TM 首先定位到这个数据项，进而通过恰当的并发控制方法请求存储它的站点并执行更新。与读的情况类似，存储该数据项的站点会在易失日志中插入一条相应的日志记录。

（4）Commit（提交）：给定一个事务，TM 会协调各个相关站点对数据项执行永久更新。同时，执行写在先日志（write-ahead logging，WAL）协议，将易失日志中的记录迁移到磁盘中的日志（也称稳定日志，stable log）。

（5）Abort（取消）：TM 确保事务的影响不会反映在任何在本事务中产生数据更新的站点上。此时，日志需要执行反做（Undo，也称回滚 Rollback）协议。

为了支持这些服务，事务管理程序（TM）需要与位于同一站点或不同站点的调度程序（SC）和数据处理程序（data processor）进行通信，如图 5.2 所示。

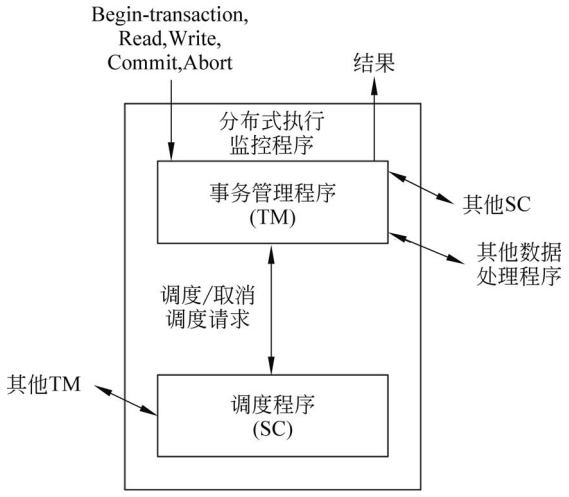

图 5.2　分布式执行监控模型的细节

正如第 1 章指出的，我们介绍的架构模型只是一种用于教学目的的抽象模型。它可以让我们把事务管理中的问题一个个提取出来做单独的讨论。5.2 节将会探讨调度算法，重点关注 TM 与 SC 之间的接口，以及 SC 与数据处理程序之间的接口。5.4 介绍提交和取消两种操作在分布式环境下的执行策略，以及需要为恢复管理程序实现的故障恢复算法。第 6 章将讨论扩展到复制数据库的情况。需要指出的是，这里介绍的计算模型并不是唯一的。人们也提出了其他模型，例如为每个事务使用一个私有工作空间等。

5.2　分布式并发控制

之前提到，并发控制算法可以确保达到特定的隔离级别。本章主要关注并发事务之间的可串行性。集中式数据库的可串行化理论可以直接扩展到分布式数据库的场景。每个站点的事务执行历史被称为局部历史（local history）。如果数据库没有复制，并且每个局部历史都是可串行化的，那么它们的并集，称为全局历史（global history），也是可串行化的。

【例 5.1】 这里举一个非常简单的例子来说明这一点。考虑两个银行账户：x（存储在站点 1）和y（存储在站点 2）。考虑T_1和T_2两个事务：其中T_1将 100 美元从x转到y，而T_2简单地读取x和y的账户余额。

T_1: Read(x)	T_2: Read(x)
$x \leftarrow x - 100$	Read(y)
Write(x)	Commit
Read(y)	
$y \leftarrow y + 100$	
Write(y)	
Commit	

显然，这两个事务都需要在两个站点上执行。考虑以下的两个历史，它们是由两个站

点局部生成的，其中H_i表示第i个站点上的历史，R_j和W_j分别是事务T_j中的 Read 和 Write 操作：

$$H_1 = \{R_1(x), W_1(x), R_2(x)\}$$
$$H_2 = \{R_1(y), W_1(y), R_2(y)\}$$

这两个历史都是可串行化（serializable）的，事实上这它们都是串行（serial）的。因此，它们都是一个正确的执行序列。此外，由于它们的串行顺序都相同的，即$T_1 \rightarrow T_2$。因此，全局历史也是可串行化的，其串行化的顺序也是$T_1 \rightarrow T_2$。

但是，如果两个站点生成的历史如下所示，就会出现问题：

$$H_1' = \{R_1(x), W_1(x), R_2(x)\}$$
$$H_2' = \{R_2(y), R_1(y), W_1(y)\}$$

虽然这两个局部历史都是可串行化的，但是它们的串行顺序不同：H_1'将T_1串行化到T_2之前（$T_1 \rightarrow T_2$），而H_2'将T_2串行化到T_1之前（$T_2 \rightarrow T_1$）。因此，全局历史不满足可串行性。

并发控制协议负责确保事务的隔离性，其目标是保证特定的隔离级别，例如可串行化（serializability）、快照隔离（snapshot isolation）或读已提交（read committed）。并发控制考虑不同的方面或维度。第一个方面显然是考虑算法所针对的隔离级别。第二个方面是考虑协议的目标是防止隔离性被破坏（悲观协议），还是允许隔离性被破坏进而中止冲突事务以保持隔离级别（乐观协议）。

第三个方面是考虑事务如何串行化。这些事务可以根据冲突访问的顺序或预定义的顺序（称为时间戳顺序）进行串行化。前者对应于基于加锁的算法，其中事务根据它们尝试获取冲突锁的顺序进行串行化；而后者对应于根据时间戳对事务进行排序的算法，其中时间戳可以在事务开始时分配（开始时间戳）——对应悲观协议，或者在提交事务之前进行分配（提交时间戳）——对应乐观协议。需要考虑的第四个方面是如何维护更新。一种方法是保留数据的单一版本(这在悲观算法中是可能的)。另一种方法是保留数据的多个版本，该方法应用于乐观算法，但一些悲观算法也使用这种方法来进行故障恢复（基本上保留两个版本，即最新提交的版本和当前未提交的版本）。下一章将讨论考虑复制（replication）的情况。

上述四个方面可以进行不同的组合，其中大多数组合都得到了研究。本节将重点介绍悲观算法、基于加锁的算法（5.2.1 节）、时间戳排序算法（5.2.2 节）和乐观算法（5.2.4 节）的开创性研究工作。理解了这些开创性方法，对学习更复杂的算法大有裨益。

5.2.1 基于加锁的并发控制算法

基于加锁的并发控制算法防止隔离性被破坏的方法是：为每个锁单元（lock unit）维护一个“锁”（lock），并要求每个操作在访问数据项之前先获得数据项上的锁，无论是在读（共享）模式还是在写（排它）模式。操作的访问请求是根据锁模式的兼容性决定的：读锁与另一个读锁兼容，写锁和读锁或写锁都不兼容。系统使用两阶段锁（two-phase locking，2PL）算法管理锁。基于分布式加锁的并发控制算法的一个基础性问题是决定在何处以及如何维护锁，这也通常被称为锁表（lock table）。后续章节将探讨针对这一问题的不同算法。

5.2.1.1　集中式 2PL

2PL 算法可以很容易地扩展到分布式 DBMS 环境，一种方法是将锁管理委派给单个站点负责。这意味着：在所有的站点中，只有一个站点拥有锁管理程序，其余站点的事务管理程序与之通信以获取锁。这种方法被称为主站点 2PL 算法（primary site 2PL algorithm）。

根据集中式 2PL（C2PL）算法，在执行一个事务的时候，站点之间的通信如图 5.3 所示。这一通信是在协调 TM、中心站点的锁管理程序和其他参与站点的数据处理程序（DP）之间进行的。这里的参与站点是指那些存储所需数据项并执行数据库操作的站点。

算法 5.1 给出了集中式 2PL 事务管理算法（C2PL-TM）的概览，算法 5.2 给出了集中式两阶段锁管理算法（C2PL-LM）。算法 5.3 给出了一个高度简化的数据处理程序（DP）算法，在 5.3 节讨论可靠性问题的时候，这一算法还会发生比较大的变化。

这些算法使用五元组表示执行的操作：$Op:\langle Type=\{BT,R,W,\ A,\ C\},arg$:数据项, val:值,tid:事务标识符,res:结果$\rangle$，其中每项的含义如下：

（1）对于操作o:Op,$o.Type\in\{BT,R,W,A,C\}$表示其类型，其中$BT=$ Begin_transaction、$R=$ Read、$W=$ Write、$A=$ Abort、$C=$ Commit；

（2）arg表示操作访问的数据项——只针对读和写操作，对于其他操作，该字段置空；

（3）val表示读和写操作从数据项arg已读取或要写入的值。与之前类似，对于其他操作，该字段置空；

（4）tid表示操作所属的事务，严格说来是事务的标识；

（5）res指定完成数据处理程序所请求的操作的代码，它对可靠性算法是十分重要的。

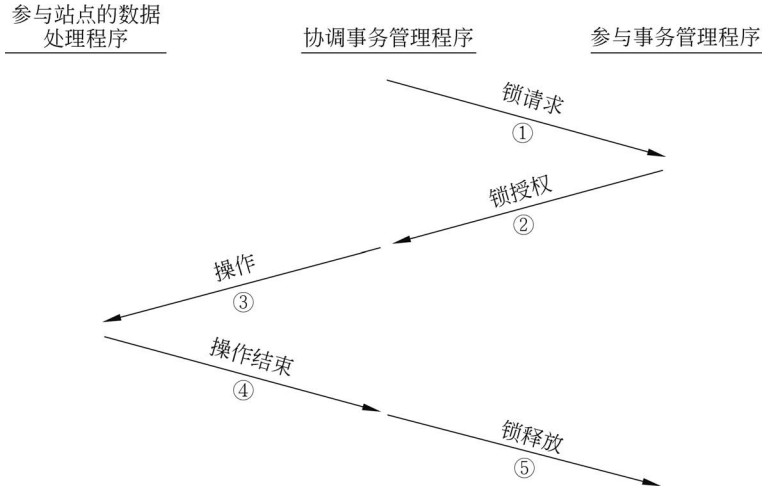

图 5.3　集中式 2PL 的通信结构

算法 5.1　集中式 2PL 事务管理程序（C2PL-TM）

```
Input: msg: a message
begin
    repeat
        wait for a msg
        switch msg do
            case transaction operation do
```

```
                let op be the operation
                if op.Type = BT then DP(op)      {call DP with operation}
                else C2PL-LM(op)                 {call LM with operation}
            end case
            case Lock Manager response do        {lock request granted or locks released}
                if lock request granted then
                    find site that stores the requested data item (say H_i)
                    DP_si(op)                    {call DP at site S_i with operation}
                else                             {must be lock release message}
                    inform user about the termination of transaction
                end if
            end case
            case Data Processer response do      {operation completed message}
                switch transaction operation do
                    let op be the operation
                    case R do
                        return op.val (data item value) to the application
                    end case
                    case W do
                        inform application of completion of the write
                    end case
                    case C do
                        if commit msg has been received from all participants then
                            inform application of successful completion of transaction
                            C2PL-LM(op)          {need to release locks}
                        else
                            record the arrival of the commit message
                        end if
                    end case
                    case A do
                        inform application of completion of the abort
                        C2PL-LM(op)              {need to release locks}
                    end case
                end switch
            end case
        end switch
    until forever
end
```

算法 5.2　集中式 2PL 锁管理程序（C2PL-LM）

```
Input: op:Op
begin
    switch op.Type do
        case R or W do                   {lock request; see if it can be granted}
            find the lock unit lu such that op.arg ⊆ lu
            if lu is unlocked or lock mode of lu is compatible with op.Type then
                set lock on lu in appropriate mode on behalf of transaction op.tid
```

```
                send "Lock granted" to coordinating TM of transaction
            else
                put op on a queue for lu
            end if
        end case
        case C or A do                                  { locks need to be released}
            foreach lock unit lu held by transaction do
                release lock on lu held by transaction
                if there are operations waiting in queue for lu then
                    find the first operation O on queue
                    set a lock on lu on behalf of O
                    send "Lock granted" to coordinating TM of transaction O.tid
                end if
            end foreach
            send "Locks released" to coordinating TM of transaction
        end case
    end switch
end
```

我们将事务管理程序（C2PL-TM）算法实现为一个永远运行的进程，负责等待来自应用程序（带有事务操作）、锁管理程序或数据处理程序的消息。同时，我们将锁管理程序（C2PL-LM）和数据处理程序（DP）算法实现为按需调用的过程。由于这里仅对这些算法做概览性的描述，因此上述区别不会造成什么影响。不过，具体的实现方式可能会大不相同。

C2PL 算法的一个公认缺陷是：中心站点会很快成为整个系统的瓶颈。此外，中心站点的故障或无法访问会造成严重的系统故障，从而让系统变得不可靠。

算法 5.3　数据处理程序（DP）

```
Input: op:Op
begin
    switch op.Type do                       {check the type of operation}
        case BT do                          {details to be discussed in Sect. 5.4}
            do some bookkeeping
        end case
        case R do
            op.res ← READ(op.arg)           {database READ operation}
            op.res ←"Read done"
        end case
        case W do                           {database WRITE of val into data item arg }
            WRITE(op.arg, op.val)
            op.res ←"Wirte done"
        end case
        case C do
            COMMIT;                         {execute COMMIT }
            op.res ←"Commit done"
        end case
```

```
            case A do
                ABORT;                          {execute ABORT}
                op.res ←"Abort done"
            end case
        end switch
    end
```

5.2.1.2 分布式 2PL

分布式 2PL（D2PL）要求每个站点都有锁管理程序。依据分布式 2PL 协议，事务在执行时各协同站点之间的通信如图 5.4 所示。

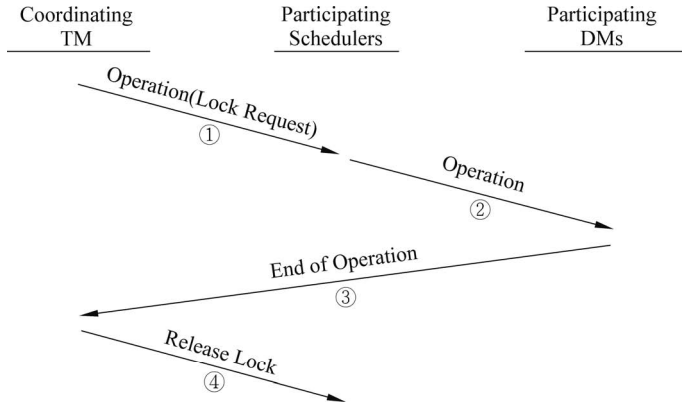

图 5.4 分布式 2PL 的通信结构

分布式 2PL 事务管理算法与 C2PL-TM 类似，但是有两点改动：第一，之前只需要发送给中心站点的锁管理程序的消息，现在需要发送给所有参与站点的锁管理程序；第二，数据操作不再由协调事务管理程序传递给数据处理程序，而是由参与站点的锁管理程序进行传递。这意味着协调事务管理程序不再需要等待“锁请求已批准”消息。另外，关于图 5.4 还有一点需要说明：所有的数据处理程序都需要向协调事务管理程序发出“操作结束”消息。另一种方法是让每一个 DP 只向其锁管理程序发送消息，然后锁管理程序负责释放锁资源并通知协调 TM。这里仅介绍第一种方法，因为它使用了一种与之前的严格 2PL 锁管理程序相同的锁管理算法，因此会使针对提交协议的讨论更简单（参见 5.4 节）。由于上述相似性，这里不再具体给出分布式事务管理和锁管理算法。分布式 2PL 算法已在 R*和 NonStop SQL 数据库中得到使用。

5.2.1.3 分布式死锁管理

基于加锁的并发控制算法可能会导致死锁；在分布式 DBMS 的情况下，由于在不同站点执行的事务存在相互等待的可能性，因此可能会产生分布式（或是全局）的死锁。死锁的检测与解决是分布式环境中管理死锁最常见的方法。等待图（wait-for graph，WFG）可用于检测死锁。WFG 是一个有向图：节点表示活跃的事务，从 T_i 指向 T_j 的边表示 T_i 正在等待 T_j 释放某个数据项上的锁。分布式环境下的 WFG 的形式会更复杂，因为事务是分布式执行的。因此，在分布式系统中，仅在每个站点上维护一个局部等待图（local wait-for graph，LWFG）是不够的，还需要将所有的局部等待图合并，形成一个全局等待图（global

wait-for graph, GWFG），并检查图中是否存在环路。

【例 5.2】考虑 4 个事务 T_1、T_2、T_3、T_4，它们之间的等待关系表示为：$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$。如果 T_1 和 T_2 在站点 1 上运行，T_3 和 T_4 在站点 2 上运行，则这两个站点的 LWFG 如图 5.5(a)所示。请注意，由于死锁是全局的，仅检查两个 LWFG 是不可能检测到死锁的。但如果我们检查 GWFG（站点之间的等待关系用虚线表示），则可以很容易地检测到死锁，如图 5.5(b)所示。

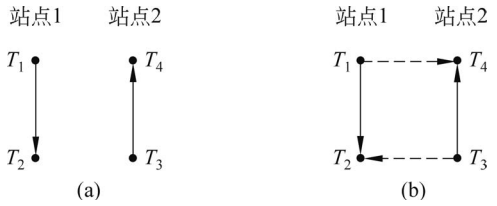

图 5.5　LWFG 与 GWFG 的区别

各种算法在管理 GWFG 的方式上有所不同。目前三种检测分布式死锁的基础算法是集中式、分布式以及层次式死锁检测。下面介绍这三种算法。

1. 集中式死锁检测

在集中式死锁检测方法中，某个站点被选择为整个系统的死锁检测程序。每个锁管理程序定期将其 LWFG 传送给死锁检测程序，然后由后者形成 GWFG 并检查是否存在环路。实际上，锁管理程序只需将其图中变化的部分（即新建或删除的边）发送给死锁检测程序。发送的时间间隔是由系统设计决定的：间隔越小，未被检测到的死锁就会变少，延迟也就越小，但死锁检测和通讯的开销也会越高。

集中式死锁检测很简单，如果并发控制算法是集中式 2PL，这将是一个非常自然的选择。不过，这类方法容易出现故障以及存在较高通信开销。

2. 层次式死锁检测

集中式死锁检测的一个改进是建立死锁检测程序的层次结构，如图 5.6 所示。具体来说，每个站点使用 LWFG 检测局部死锁，进而将其 LWFG 传递给上级死锁检测程序。于是，涉及两个或多个站点的分布式死锁就可以由这些站点的直接上级死锁检测程序检测到。例如，站点 1 的死锁将由站点 1 上的局部死锁检测程序（记为 DD_{21}，第一个数字 2 代表第 2 层，第二个数字 1 代表站点 1）检测出来。如果死锁涉及站点 1 和 2，则会由 DD_{11} 负责检测。如果死锁涉及站点 1 和 4，则会由 DD_{0x} 来检测，其中 x 的值可以是 1、2、3 或 4。

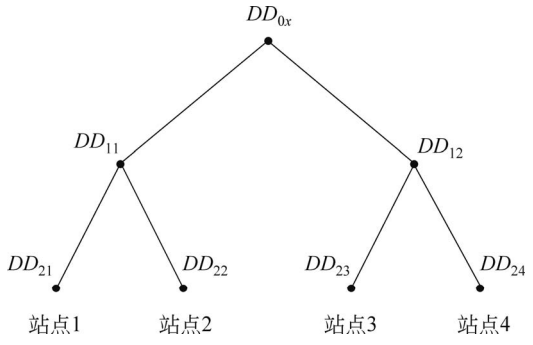

图 5.6　层次式死锁检测