# 第十三章 交易管理和並行控制

資料庫系統設計理論

李紹綸 著

# 本章內容

- 交易管理 (Transaction Management)
  - 交易的 ACID 四大特性
  - 交易狀態 (Transaction States)
- 錯誤回復 (Failure Recovery)
  - 錯誤的種類 (Types of Failures)
  - 錯誤回復的處理 (Process of Failure Recovery)
- 為何需要並行控制
- 排程的循序性 (Serializability of Schedules)
  - 如何測試非序列排程的正確性
  - 優先次序圖 (Precedence Graph)
- 並行控制的方法 (Methodology of Concurrency Control)
  - 鎖定法 (Locking)
  - 時間戳記法 (Timestamp Ordering)

# 本章內容

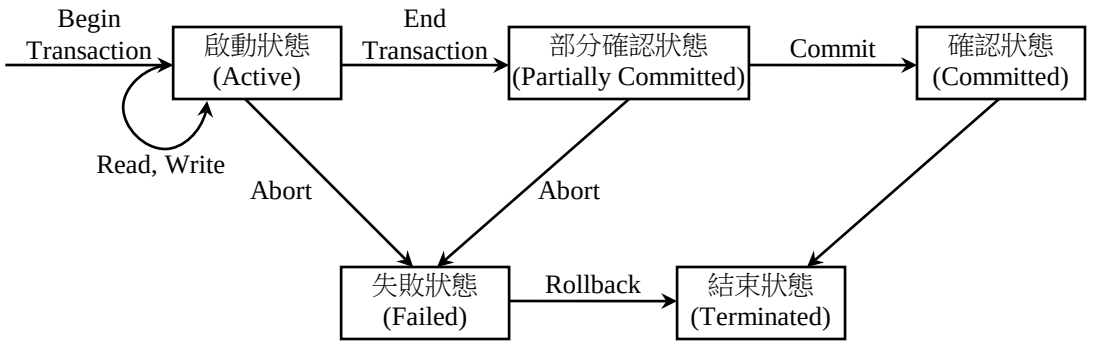
- SQL交易指令
  - BEGIN TRANSACTION
  - COMMIT TRANSACTION
  - COMMIT WORK
  - ROLLBACK TRANSACTION
  - ROLLBACK WORK
  - SAVE TRANSACTION
  - 交易的架構
  - 巢狀交易 (Nested Transaction)
  - 分散式交易 (Distributed Transaction)
  - 交易的隔離等級 (Isolation Level)
  - 資料鎖定 (Lock)
  - 鎖定的死結問題

# 交易的 ACID 四大特性

- 對於交易管理而言，為了維持資料庫的正常運作，每個交易都必須遵守 ACID 的特性，現分述如下：
  - 單元性 (Atomicity)
  - 一致性 (Consistency)
  - 隔離性 (Isolation)
  - 永久性 (Durability)

# 交易狀態 (Transaction States)

- 可以將整個交易的過程再分成下列數種狀態：
  - 啟動狀態 (Active State)
  - 部分確認狀態 (Partially Committed State)
  - 確認狀態 (Committed State)
  - 失敗狀態 (Failed State)
  - 結束狀態 (Terminated State)
- 交易狀態圖



# 錯誤的種類 (Types of Failures)

- 交易錯誤 (Transaction Failures)
  - 邏輯錯誤 (Logical Error)
  - 局部錯誤 (Local Error)
  - 並行控制強制錯誤 (Concurrency Control Enforcement Error)
- 系統錯誤 (System Failure)
  - 電腦錯誤 (Computer Failures)
  - 天災 (Catastrophes)
- 儲存媒體錯誤 (Storage Medium Failure)
  - 磁碟錯誤 (Disk Failure)

# RAID 技術

- RAID 技術可以分為七個等級，現分述如下：
  - RAID 0：又稱為「資料分解」(Data Striping)
  - RAID 1：又稱為「磁碟鏡射」(Disk Mirroring)
  - RAID 2：又稱為「位元交錯」(Bit-Interleaving)
  - RAID 3：又稱為「位元組交錯」(Byte-Interleaving)
  - RAID 4：又稱為「交錯傳輸區段」(Interleaves Transfer Blocks)
  - RAID 5：又稱為「磁碟分解」(Disk Striping)
  - RAID 6：此種方法與 RAID 5 做法相同

# 錯誤回復的處理 (Process of Failure Recovery)

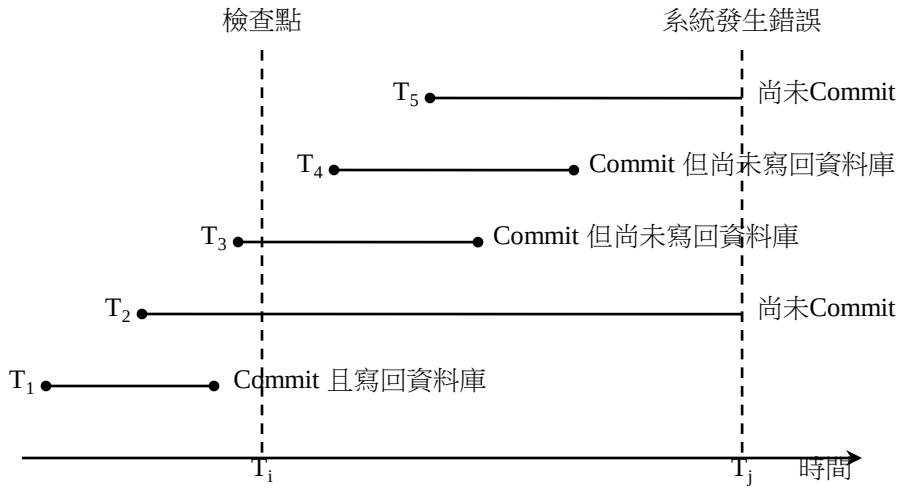
- 「系統日誌」(System Log)中通常會包括下列幾種不同的記錄：
  - [Start_Transaction, T]：記錄著交易 T 開始被執行。
  - [Write_Item, T, X, Old_value, New_value]：記錄著交易 T 將資料庫中項目 X 的值從原先的 Old_value 更改為新的 New_value。
  - [Read_Item, T, X]：記錄著交易 T 從資料庫中讀取項目 X 的值。
  - [Commit, T]：記錄著交易 T 已經成功執行完成，而且會永久地 (Permanently) 記錄在資料庫中。
  - [Abort, T]：記錄著交易 T 已經被中止執行。

# 錯誤回復的處理 (Process of Failure Recovery)

- 「交易記錄」通常會包括下列幾項資訊：
  - 交易編號 (Transaction ID)。
  - 記錄編號 (Transaction Log Record ID)。
  - 指令形式 (Action Type)：包括 Begin Transaction、Commit、Abort、Insert、Delete 和 Update 等。
  - 更新前的資料值 (Old Value)：提供不做交易 (Undo) 時使用。
  - 更新後的資料值 (New Value)：提供重做交易 (Redo) 時使用。

# 錯誤回復的處理 (Process of Failure Recovery)

- 假設系統中依序有 T1、T2、T3、T4 和 T5 等五個交易正在執行，最後一次「檢查點」發生在時間 Ti，之後系統就在 Tj 時間點發生錯誤，在系統發生錯誤之前的狀態圖如下：



# 錯誤回復的處理 (Process of Failure Recovery)

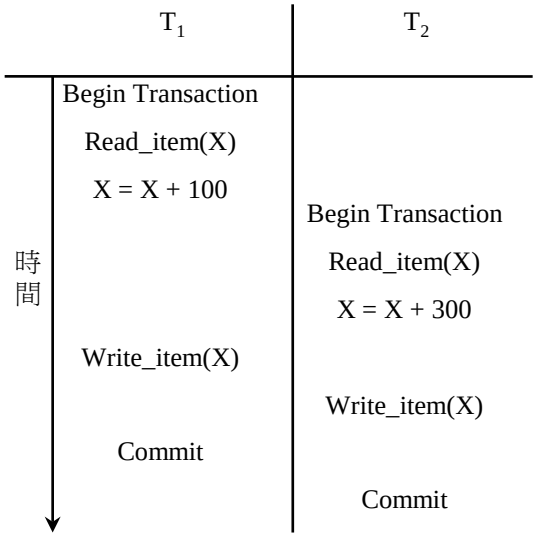
- 「回復處理」的步驟如下：
  - 將 UNDO-List 和 REDO-List 設成空集合。
  - 在「檢查點記錄」中，由最後一個「檢查點」往前面時間搜尋，看看是否存在某交易 T 有開始被執行 [Start_Transaction, T]，但是卻沒有 [Commit, T] 或 [Abort, T]，如果有則將交易 T加入 UNDO-List 中。
  - 在「檢查點記錄」中，由最後一個「檢查點」往後面時間搜尋，看看是否存在某交易 T 有開始被執行 [Start_Transaction, T]，如果有則將交易 T 加入 UNDO-List 中。
  - 在「檢查點記錄」中，由最後一個「檢查點」往後面時間搜尋，看看是否存在某交易 T 有被確認 [Commit, T]，如果有則將交易 T 從 UNDO-List 中刪除，再加入 REDO-List 中。
  - 對於在 UNDO-List 中的所有交易，「不做」(Undo) 所有交易，讓系統回復到之前狀態。
  - 對於在 REDO-List 中的所有交易，「重做」(Redo) 所有交易。

# 為何需要並行控制

- 當多個交易在並行執行時，最常出現的三種問題：
  - 「遺失更新問題」(Lost Update Problem)
  - 「暫時更新問題」(Temporary Update Problem)
  - 「錯誤加總問題」(Incorrect Summary Problem)

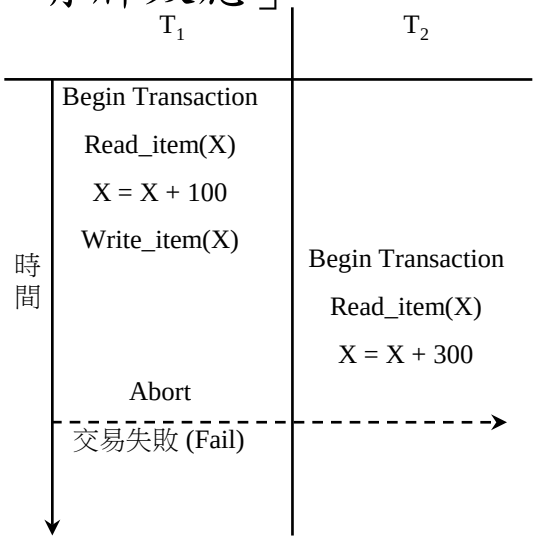
# 「遺失更新問題」(Lost Update Problem)

- 兩個交易交錯地存取了資料庫中相同的項目，並且執行更新的動作，因而造成前一個交易的資料被後來交易的資料所覆蓋 (Overwritten) 而遺失了原有的資料。



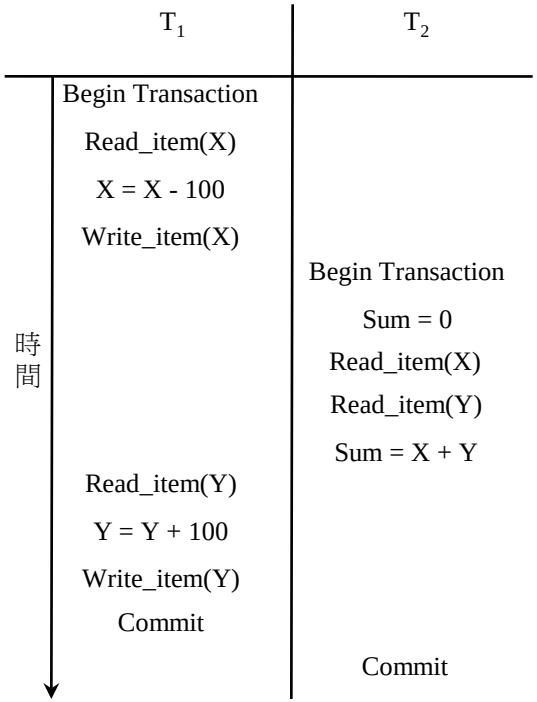
# 「暫時更新問題」(Temporary Update Problem)

- 又稱之為「骯髒讀取問題」(Dirty Read Problem)。
- 這個問題發生在並行中的兩個交易 $T_1$ 和 $T_2$，當交易 $T_2$ 所讀取交易 $T_1$ 中項目 X 的值，是一個未經確認的值 (Uncommitted Value)，當交易 $T_1$ 執行失敗時，交易 $T_2$ 所讀取項目 X 的值便是一個不存在的值，因而會造成交易 $T_2$ 產生不正確的結果。
- 此種讀取未經確認的值通常會造成連續性「中止」交易的現象，我們稱之為「連鎖撤回」或「骨牌效應」。



# 「錯誤加總問題」(Incorrect Summary Problem)

- 當一個交易在處理加總作業時，另一個交易同時存取了資料庫中相同的項目，並且執行更新的動作，因而造成正在處理加總作業的交易會得到不正確的結果。



# 排程的循序性 (Serializability of Schedules)

- 通常是一個交易在讀取 (Read) 某一資料時，另一個交易卻要寫入 (Write) 該資料，此時這兩個交易便會發生衝突 (Conflict)。
- 為了提升系統的執行效能，我們當然不希望這些交易只能依序逐一被執行，但是如果並行執行這些交易卻可能造成交易之間彼此衝突，因此 DBMS 就必須做好「並行控制」，讓所有的交易能夠交錯地 (Interleaving) 執行。

# 「排程」(Schedule)

- 當多個交易同時在執行時，這些交易所有指令執行的先後順序，我們稱之為「排程」(Schedule)。
- 如果排程不會破壞每個交易原來指令執行的先後順序，則此排程是一個「合法的排程」(Legal Schedule) 。
- 否則便是一個「不合法的排程」(Illegal Schedule)，「不合法的排程」其執行結果可能會產生錯誤。

# 「序列排程」V.S.「非序列排程」

- 每個排程通常包含了許多的交易，如果這些交易中的所有指令全部集中在一起以無交錯 (Non-Interleaving) 方式執行，則此排程稱之為「序列排程」(Serial Schedule) 。
- 這些「序列排程」都是合法的排程。
- 為提升系統的執行效率，DBMS 通常會允許交易中的所有指令以並行和交錯方式執行，在此種排程中，各個交易指令會以交錯方式執行，所以這種排程被稱為「非序列排程」(Nonserial Schedule)。
- 「非序列排程」其執行結果可能會產生錯誤，因此並不是每個非序列排程都是「合法的排程」。

# 如何測試非序列排程的正確性

- 如果多個交易並行執行的結果與這些交易按照某一順序個別執行的結果一樣時，我們稱這些交易的排程具有「循序性」(Serializability)。
- 符合「循序性」條件的排程一定會保持資料的一致性，且其執行結果也一定正確。
- 「合法的排程」不一定要具有「循序性」。
- 因為在「非序列排程」的所有組合方式中屬於「合法的排程」的數目遠遠大於「序列排程」具有「循序性」的數目。

# 「衝突相等」(Conflict Equivalent)

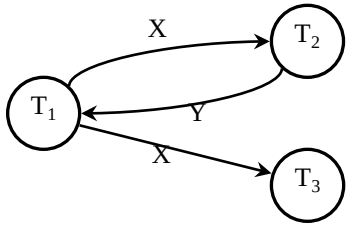
- 我們通常可以利用「衝突相等」(Conflict Equivalent) 的觀念來辨別「非序列排程」的正確性。
- 在同一個排程中，對於同一個項目 X 而言，每個交易中 Write_Item(X) 指令都會和其他交易的 Write_Item(X) 和 Read_Item(X) 指令相衝突。
- 當兩個排程是「衝突相等」時，在這兩個排程中所有相衝突的指令執行順序都會是一樣的。
- 對於含有 n 個交易的「非序列排程」而言，如果該排程與 n! 中某一個「序列排程」是「衝突相等」時，則該排程是一個「合法的排程」，而且此排程稱之為「可衝突相等序列排程」或「具循序性排程」。

# 優先次序圖 (Precedence Graph)

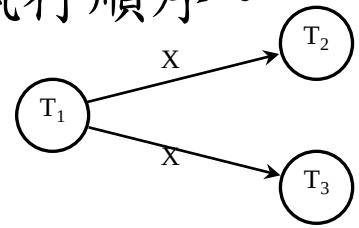
- 要測試一個排程是否具有「循序性」，我們通常會利用有方向性的「優先次序圖」(Precedence Graph) 來測試。
- 每條有向邊 $T_1 \rightarrow T_2$，代表交易 $T_1$ 必須在交易 $T_2$ 之前先執行。
- 當所畫出的「優先次序圖」中含有迴路 (Cycle) 時，則表示該排程不具有「循序性」。
- 如果「優先次序圖」中不含迴路時，則表示該排程具有「循序性」。

# 優先次序圖 (Precedence Graph)

- 含有迴路的「優先次序圖」
  - 因為含有迴路，所以該排程不具有「循序性」。

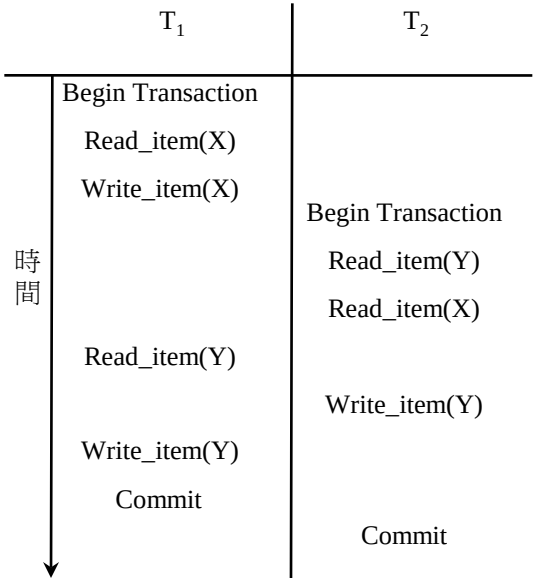



- 不含迴路的「優先次序圖」
  - 因為不含迴路，所以該排程具有「循序性」，而且其「可衝突相等序列排程」可以是 $T_1 \rightarrow T_2 \rightarrow T_3$ 或者是 $T_1 \rightarrow T_3 \rightarrow T_2$ 的執行順序。

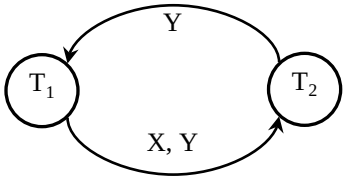


# 優先次序圖 (Precedence Graph)

- 包含 $T_1$ 和 $T_2$ 兩個交易的排程




- 「優先次序圖」



## 並行控制的方法 (Methodology of Concurrency Control)

- 優先次序圖 (Precedence Graph)
- 鎖定法 (Locking)
- 時間戳記法 (Timestamp Ordering)

# 鎖定法 (Locking)

- 兩個並行的交易如果發生衝突 (Conflict) 通常必須具備下列兩個條件：
  - 同時要鎖定同一資料。
  - 至少包含一個「寫入鎖定」。
- 「鎖定法」便是利用交易對於要存取的資料先進行鎖定，以避免交易之間彼此產生衝突，達到交易的「並行控制」。
- 當交易 T 想要存取某一資料時，則必須先將該資料予以「鎖定」(Lock)，如果該資料已經被其他交易鎖定時，則交易 T 便需要等待，直到該資料已經被其他交易「解除鎖定」(Unlock) 為止。

# 二位元鎖定 (Binary Locks)

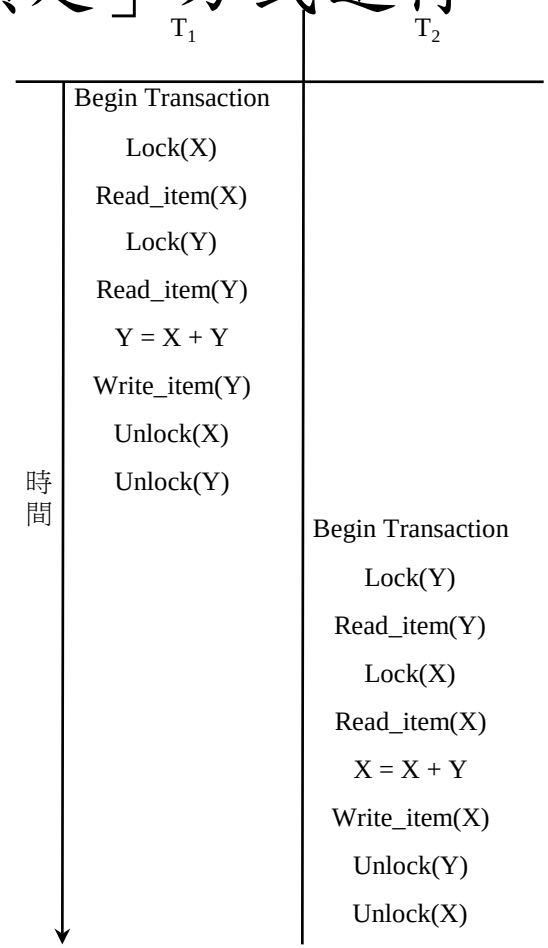
- 在「二位元鎖定」中，通常會包括「鎖定」(Lock) 和「解除鎖定」(Unlock) 等兩種基本操作。
- 每個交易在存取資料 X 之前，必須對資料 X 先申請「鎖定」，如果該資料已經被其他交易鎖定 (即 Lock(X) = 1 時)，則該交易便需要等待，如果資料 X 沒有被其他交易鎖定 (即 Lock(X) = 0 時)，則該交易便可以存取資料 X，並且要將 Lock(X) 的值改設為 1，表示資料 X 已經被此交易鎖定。
- 當該交易存取資料 X 完畢之後，便要將 Lock(X) 的值改設為 0，表示資料 X 已經被「解除鎖定」，如果此時還有其他交易還在等待申請「鎖定」，鎖定管理子系統 (Lock Manager Subsystem) 便會喚醒等待中的交易進行執行。

# 二位元鎖定 (Binary Locks)

- 採用「二位元鎖定」時，所有的交易都必須是一個「良好格式」(Well-formed)，都必須滿足下列規則：
  - 在讀取或寫入資料的內容之前都必須先申請「鎖定」。
  - 在讀取或寫入資料完畢之後都必須「解除鎖定」。
  - 已經被「鎖定」的資料便不能再次申請「鎖定」。
  - 尚未被申請「鎖定」的資料便不能「解除鎖定」。

# 二位元鎖定 (Binary Locks)

- 使用「二位元鎖定」方式進行「並行控制」



# 共享 / 互斥鎖定 (Shared / Exclusive Locks)

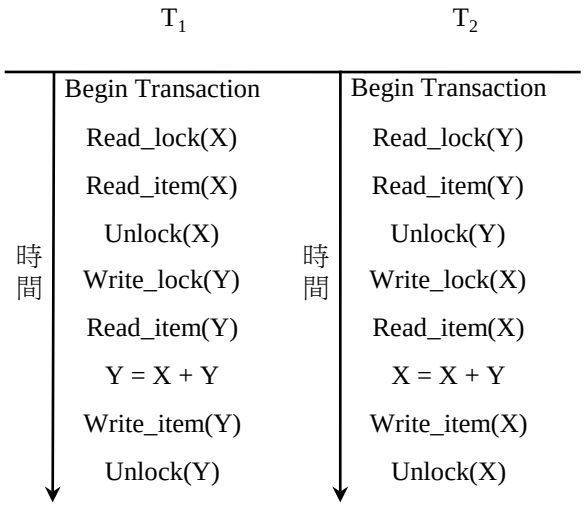
- 「共享 / 互斥鎖定」一般也稱為讀取 / 寫入鎖定 (Read / Write Locks)。
- 通常會包括「讀取鎖定」(Read Lock)、「寫入鎖定」(Write Lock) 和「解除鎖定」(Unlock) 等三種基本操作。
- 「共享 / 互斥鎖定」將鎖定模式分為下列兩種：
  - 共享模式 (Shared Mode)
  - 互斥模式 (Exclusive Mode)
- 一旦交易讀取或寫入資料完畢之後，均可以使用「解除鎖定」，移除對於資料的「讀取鎖定」或「寫入鎖定」。

# 共享 / 互斥鎖定 (Shared / Exclusive Locks)

- 採用「共享 / 互斥鎖定」時，所有的交易都必須滿足下列規則：
  - 在讀取資料的內容之前都必須先申請「讀取鎖定」或「寫入鎖定」。
  - 在寫入資料的內容之前都必須先申請「寫入鎖定」。
  - 在讀取或寫入資料完畢之後都必須「解除鎖定」。
  - 已經被「讀取鎖定」或「寫入鎖定」的資料便不能再次申請「讀取鎖定」。
  - 已經被「讀取鎖定」或「寫入鎖定」的資料便不能再次申請「寫入鎖定」。
  - 尚未被申請「讀取鎖定」或「寫入鎖定」的資料便不能「解除鎖定」。

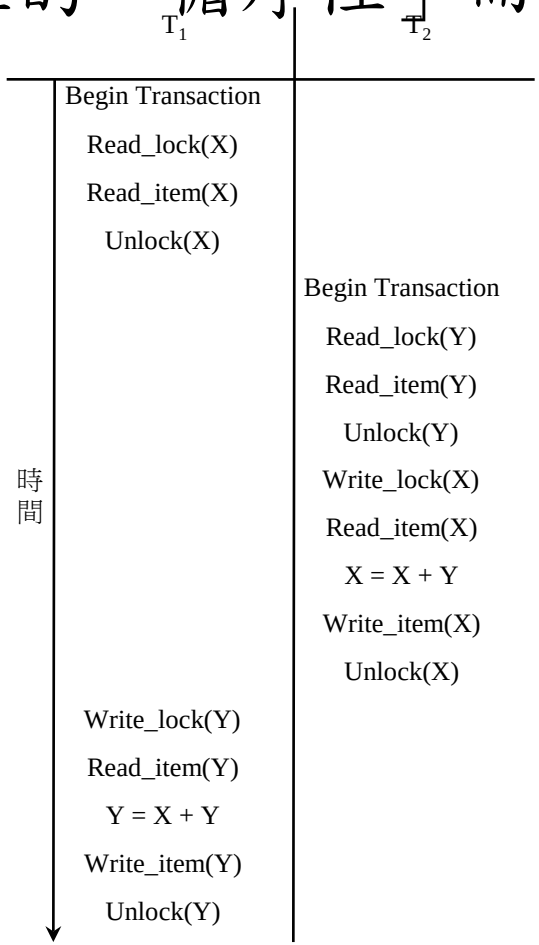
# 共享 / 互斥鎖定 (Shared / Exclusive Locks)

- 使用「共享 / 互斥鎖定」方式進行「並行控制」的兩個交易



# 共享 / 互斥鎖定 (Shared / Exclusive Locks)

- 因無法確保排程的「循序性」而產生錯誤的結果

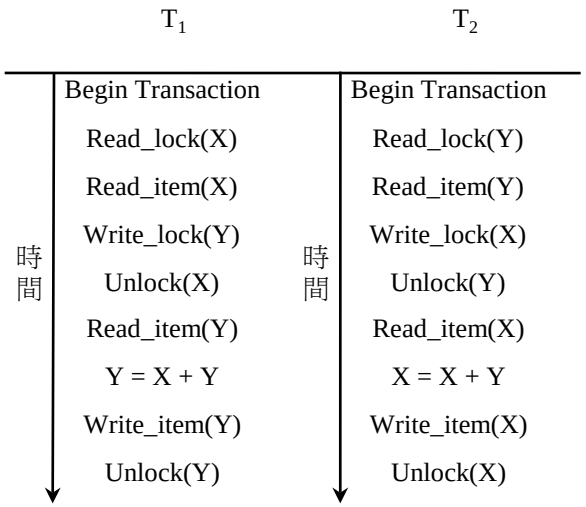


# 兩階段鎖定法 (Two Phase Locking)

- 兩階段鎖定協定 (Two Phase Locking Protocol) 會要求每個交易都必須分別在兩個不同階段來「申請鎖定」(Locking) 和「解除鎖定」(Unlocking)，這兩個階段分別為：
  - 成長階段 (Growing Phase)：或稱為鎖定階段 (Locking Phase)，在此階段的交易只能「申請鎖定」而不能夠「解除鎖定」。
  - 收縮階段 (Shrinking Phase)：或稱為解除鎖定階段 (Unlocking Phase)，在此階段的交易只能「解除鎖定」而不能夠「申請鎖定」。

# 兩階段鎖定法 (Two Phase Locking)

- 使用「兩階段鎖定」方式進行「並行控制」的兩個交易



# 「兩階段鎖定」的版本

- 「兩階段鎖定」協定其實又可分為三種不同的版本：
  - 「基本的兩階段鎖定」(Basic Two-Phase Locking)：可以確保排程的「循序性」，但是卻無法避免「死結」(Deadlock) 和「連鎖撤回」(Cascading Rollback) 現象的發生。
  - 「保守的兩階段鎖定」(Conservative Two-Phase Locking)：交易在開始執行之前，就必須先將所有需要的資料予以「鎖定」才可以執行交易，如此便可避免「死結」現象的發生。
  - 「嚴格的兩階段鎖定」(Strict Two-Phase Locking)：要求所有的交易必須等到已經確認 (Commit) 或是中止 (Abort) 之後，才可以放棄鎖定 (Unlock)，如此便可避免「連鎖撤回」現象的發生。

# 發生死結的條件

- 同時執行多個交易並不一定都會發生「死結」現象，通常只有在多個交易之中會產生「遺失更新問題」時，才會造成「死結」。
- 交易必須滿足下列四個條件，「死結」才會發生：
  - 「彼此互斥」(Mutual Exclusion)
  - 「鎖定且等待」(Lock and Wait)
  - 「不得強佔」(No Preemption)
  - 「循環等待」(Cyclic Waits)
- 想要避免死結發生，只有讓交易不滿足「循環等待」的條件。

# 死結預防 (Deadlock Prevention)

- 「死結預防」是在執行交易之前就先檢查交易執行中的所有狀態，以確認是否會導致死結。
- 第一種做法是採用「保守的兩階段鎖定」(Conservative Two-Phase Locking)，此種做法並不允許死結的發生。
- 另外一種做法則是使用交易時間戳記(Transaction Timestamp) TS(T) 的技術來預防死結的發生。

# 交易時間戳記 (Transaction Timestamp)

- 使用交易時間戳記法通常有兩種演算法可以預防死結的發生：
  - 「等待死亡」(Wait-die) 。
  - 「傷害等待」(Wound-wait) 。

# 「等待死亡」(Wait-die)

- 假設 $TS(T_1) < TS(T_2)$，則表示交易 $T_1$ 是發生在交易 $T_2$ 之前，那麼當交易 $T_1$ 要使用交易 $T_2$ 已經鎖定的資料時，則交易 $T_1$ 允許繼續等待。
- 相反的，假設交易 $T_2$ 要使用交易 $T_1$ 已經鎖定的資料時，則交易 $T_2$ 必須立即中止執行而死亡，之後再以相同的交易時間戳記重新啟動執行，以避免飢餓 (Starvation) 問題產生。

# 「傷害等待」(Wound-wait)

- 假設 $TS(T_1) < TS(T_2)$，則表示交易 $T_1$ 是發生在交易 $T_2$ 之前，那麼當交易 $T_1$ 要使用交易 $T_2$ 已經鎖定的資料時，則交易 $T_2$ 必須立即中止執行 (交易 $T_1$ 傷害交易 $T_2$)，之後再以相同的交易時間戳記重新啟動執行，以避免飢餓問題產生。
- 相反的，假設交易 $T_2$ 要使用交易 $T_1$ 已經鎖定的資料時，則交易 $T_2$ 必須繼續等待。

# 死結偵測 (Deadlock Detection)

- 不同於「死結預防」的做法，「死結偵測」是允許死結發生的。
- 「死結偵測」通常使用「等待圖」(Wait-for Graph) 進行檢查正在等待的交易是否形成一個迴圈，如果迴圈存在，表示這些正在等待的交易會發生死結，那麼系統會選擇其中一個交易當作犧牲者 (Victim)，強迫該交易進行撤回 (Rollback) 動作，並重新開始執行。
- 「等待圖」是以各個交易為節點，如果交易 $T_i$ 正在等待交易 $T_j$ 已經鎖定的資料時，我們通常會以 $T_i \rightarrow T_j$ 符號表示，如果「等待圖」中形成一個迴圈時，就表示已經發生死結了。

# 飢餓問題 (Starvation Problem)

- 「飢餓」(Starvation) 問題是指其他的交易都可以正常執行的狀況之下，唯獨某一交易卻因持續性的等待 (Wait) 或不斷地執行回復 (Rollback) 動作而一直無法順利地執行，造成該交易處於「飢餓」狀態。
- 生「飢餓」的原因是因為在解決「死結」問題的過程中，可能一直選擇同一個交易作為犧牲者，因而造成該交易一直執行回復動作，而無法順利地執行。
- 交易時間戳記法中的「等待死亡」(Wait-die) 或「傷害等待」(Wound-wait) 兩種演算法除了可以預防死結的發生，同樣也可以避免「飢餓」問題的產生。

# 解決「飢餓」問題的方法

- 解決「飢餓」問題其方法有兩種，現說明如下：
  - 先申請鎖定者擁有較高的優先權
  - 等待愈久者擁有愈高的優先權

# 時間戳記法 (Timestamp Ordering)

- 「時間戳記」(Timestamp) 是由 DBMS 所用來建立識別每一個交易的唯一識別碼。
- 當交易開始交付 (Submit) 給系統執行時，系統便依序賦予每個交易一個「時間戳記」，記載著交易的開始時間或順序。
- 使用「時間戳記法」的交易並不是藉由「鎖定」的方式達成並行控制，因此不會發生「死結」。

# 交易衝突

- 當交易 $T_1$ 要求讀取某一資料，而該資料曾經被另一個較年輕 (即時間戳記較大) 的交易 $T_2$ 更改過時
- 當交易 $T_1$ 要求寫入某一資料，而該資料曾經被另一個較年輕 (即時間戳記較大) 的交易 $T_2$ 讀取或更改過時
- 則我們稱此交易 $T_1$ 和交易 $T_2$ 發生衝突 (Conflict)
- 如果發生衝突時，解決衝突的方式是先將交易 $T_1$ 中止並且將交易 $T_1$ 賦予新的時間戳記之後，重新交付 (Resubmit) 再執行一次，所以不會有撤回 (Rollback) 的狀況發生

# 時間戳記法 (Timestamp Ordering)

- 在「時間戳記法」中，對於每一資料 X 皆有兩種不同的「時間戳記」，分別為：
  - Read_TS(X)：代表資料 X 的讀取時間戳記 (Read Timestamp)，裡面記載著所有讀取過資料 X 的交易中，最年輕 (即時間戳記最大) 交易的時間戳記，亦即 Read_TS(X) = TS(T)，其中 TS(T) 為交易 T 的時間戳記。
  - Write_TS(X)：代表資料 X 的寫入時間戳記 (Write Timestamp)，裡面記載著所有寫入資料 X 的交易中，最年輕 (即時間戳記最大) 交易的時間戳記，亦即 Write_TS(X) = TS(T)，其中 TS(T) 為交易 T 的時間戳記。

# 時間戳記法 (Timestamp Ordering)

- 當交易 T 開始交付系統執行時，系統便依序賦予該交易一個時間戳記 TS(T)，並且將每個資料 X 利用讀取時間戳記 Read_TS(X) 和寫入時間戳記 Write_TS(X)，分別記載著所有讀取和寫入該資料的所有交易中最大的時間戳記。
- 當交易 T 要存取某一資料 X 時，就先比較交易 T 的時間戳記 TS(T) 和 Read_TS(X) 或 Write_TS(X) 的值，如果交易 T 的時間戳記 TS(T) 比 Read_TS(X) 或 Write_TS(X) 的值來得大時，那麼交易 T 便可以執行。
- 然後再將資料 X 的讀取時間戳記 Read_TS(X) 或寫入時間戳記 Write_TS(X) 更新為交易 T 的時間戳記 TS(T) 。
- 否則必須將交易 T 另外領取一個較大的時間戳記之後，隨即重新交付再執行一次。

# BEGIN TRANSACTION

- BEGIN TRANSACTION 指令敘述格式如下：

```
BEGIN { TRAN | TRANSACTION }
[ { transaction_name | @tran_name_variable } [ WITH MARK [ 'description' ] ] ]
[ ; ]
```

# COMMIT TRANSACTION

- COMMIT TRANSACTION 指令敘述格式如下：

```
COMMIT { TRAN | TRANSACTION }
[ transaction_name | @tran_name_variable ]
[ ; ]
```

# COMMIT WORK

- COMMIT WORK 指令敘述格式如下：

```
COMMIT [ WORK ]
[ ; ]
```

# ROLLBACK TRANSACTION

- ROLLBACK TRANSACTION 指令敘述格式如下：

```
ROLLBACK { TRAN | TRANSACTION }
[ transaction_name | @tran_name_variable | savepoint_name | @savepoint_variable ]
[ ; ]
```

# ROLLBACK WORK

- ROLLBACK WORK 指令敘述格式如下：

```
ROLLBACK [ WORK ]
[ ; ]
```

# SAVE TRANSACTION

- SAVE TRANSACTION 指令敘述格式如下：

```
SAVE { TRAN | TRANSACTION }
{ savepoint_name | @savepoint_variable }
[ ; ]
```

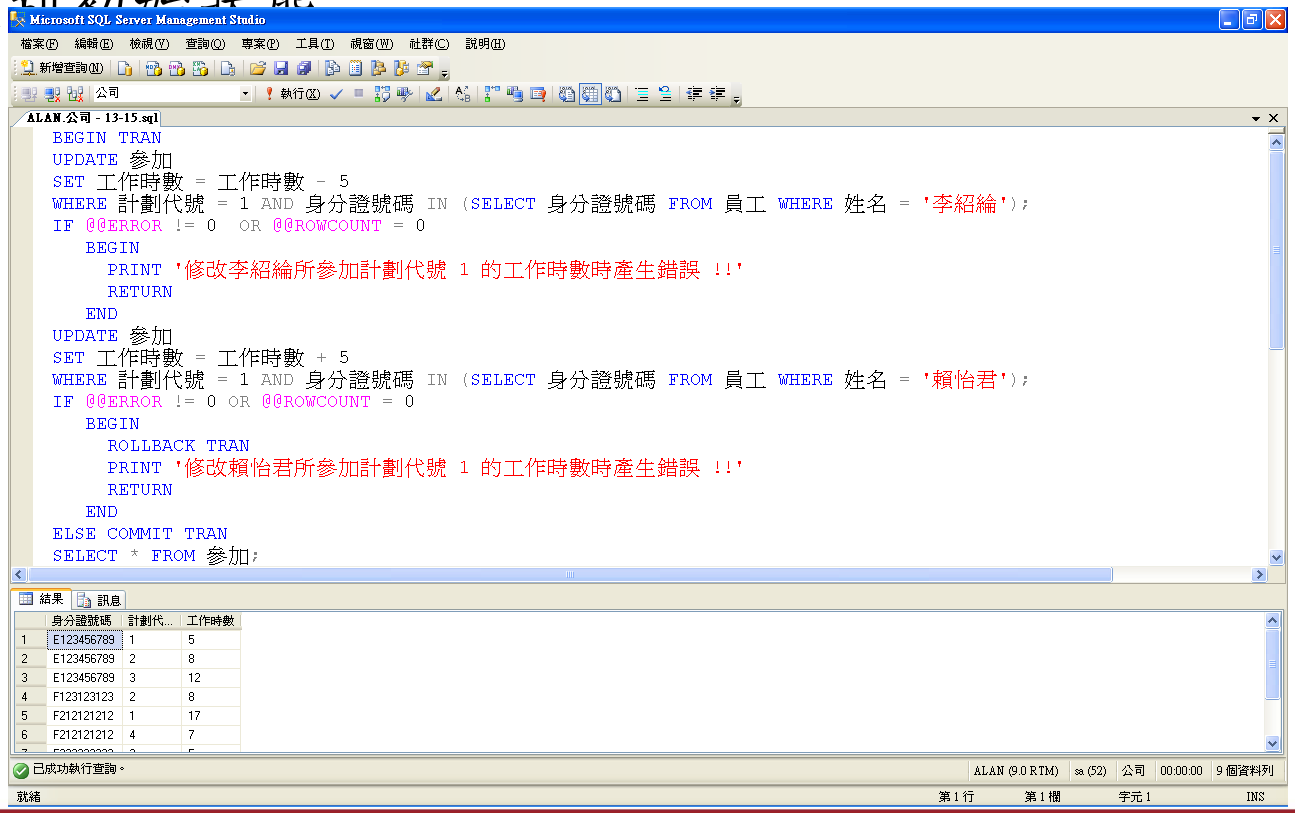
# 交易的架構

- 一個完整交易的架構圖

```
BEGIN TRANSACTION
……
SAVE TRANSACTION savepointname
……
IF (@@ERROR != 0)
  BEGIN
     ROLLBACK TRANSACTION  /* 或 ROLLBACK TRANSACTION savepointname */
     ……                   /* 交易失敗時的後續處理 */
  END
ELSE   COMMIT TRANSACTION
……                        /* 交易結束後的指令敘述 */
```

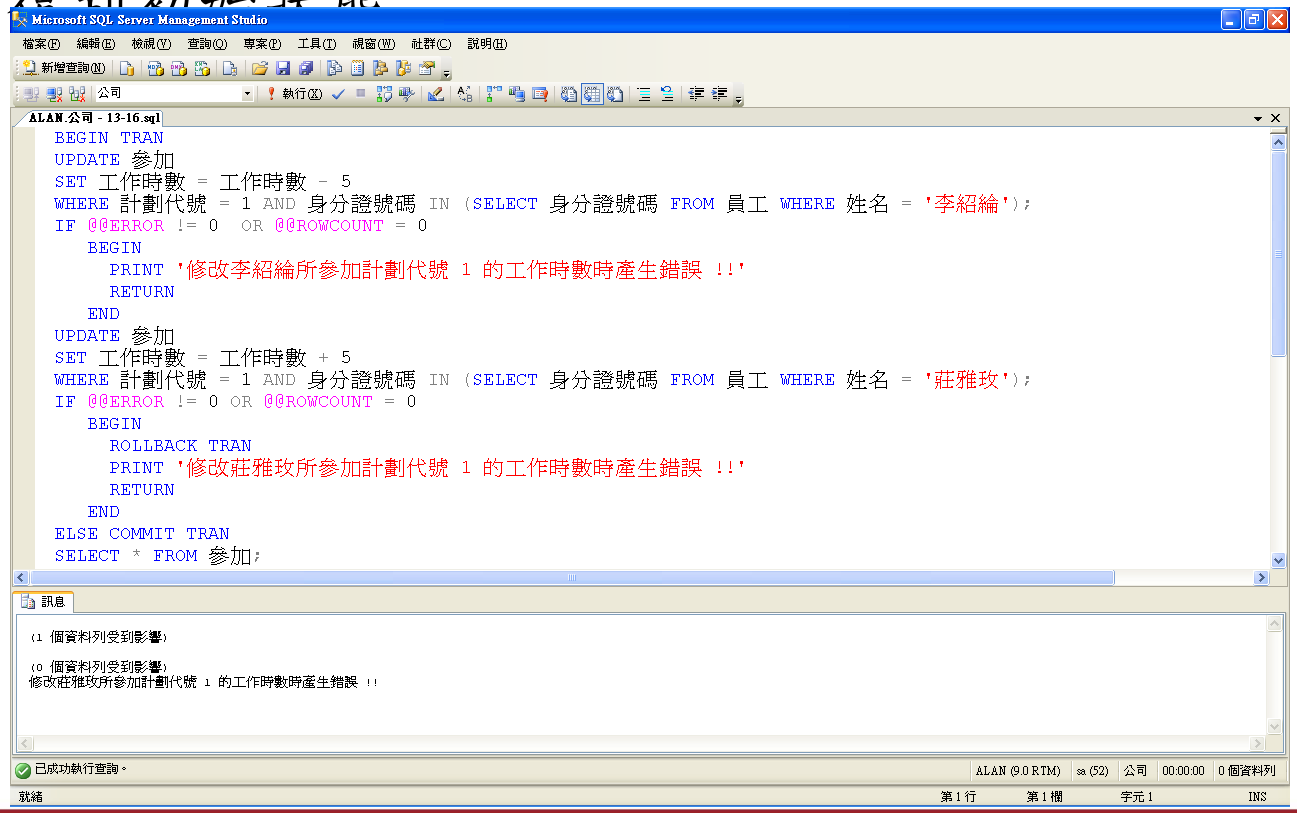
# 【範例 13.1】

- 請在“公司”資料庫中將員工李紹綸所參加計劃代號 1 的 5 小時工作時數轉交由賴怡君這名員工來執行，請用一個完整交易的架構來執行，如果交易失敗時，則必須將交易回復到初始狀態



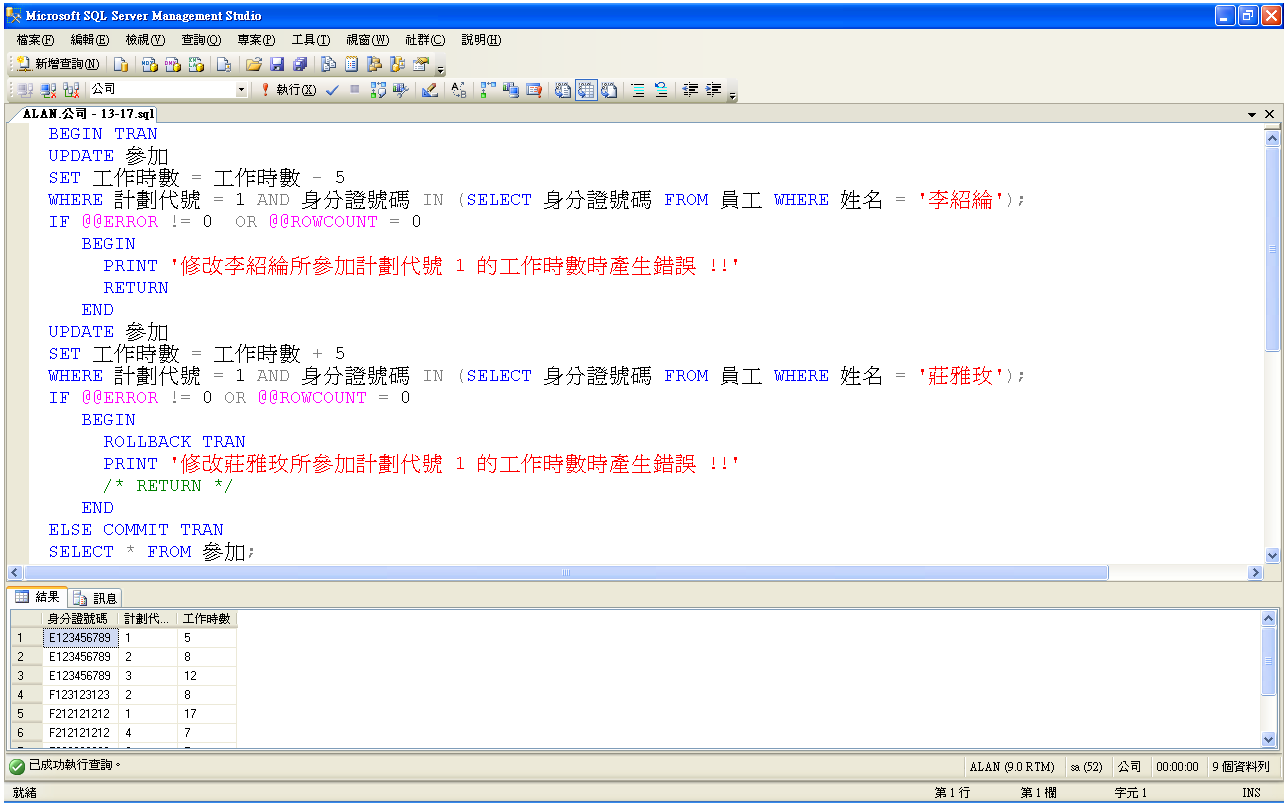
# 【範例 13.2】

- 請在“公司”資料庫中將員工李紹綸所參加計劃代號 1 的 5 小時工作時數再轉交由莊雅玫這名員工來執行，請用一個完整交易的架構來執行，如果交易失敗時，則必須將交易回復到初始狀態



# 範例 13.2

- 採用完整交易的架構來執行 SQL 指令結果 (不含 RETURN)

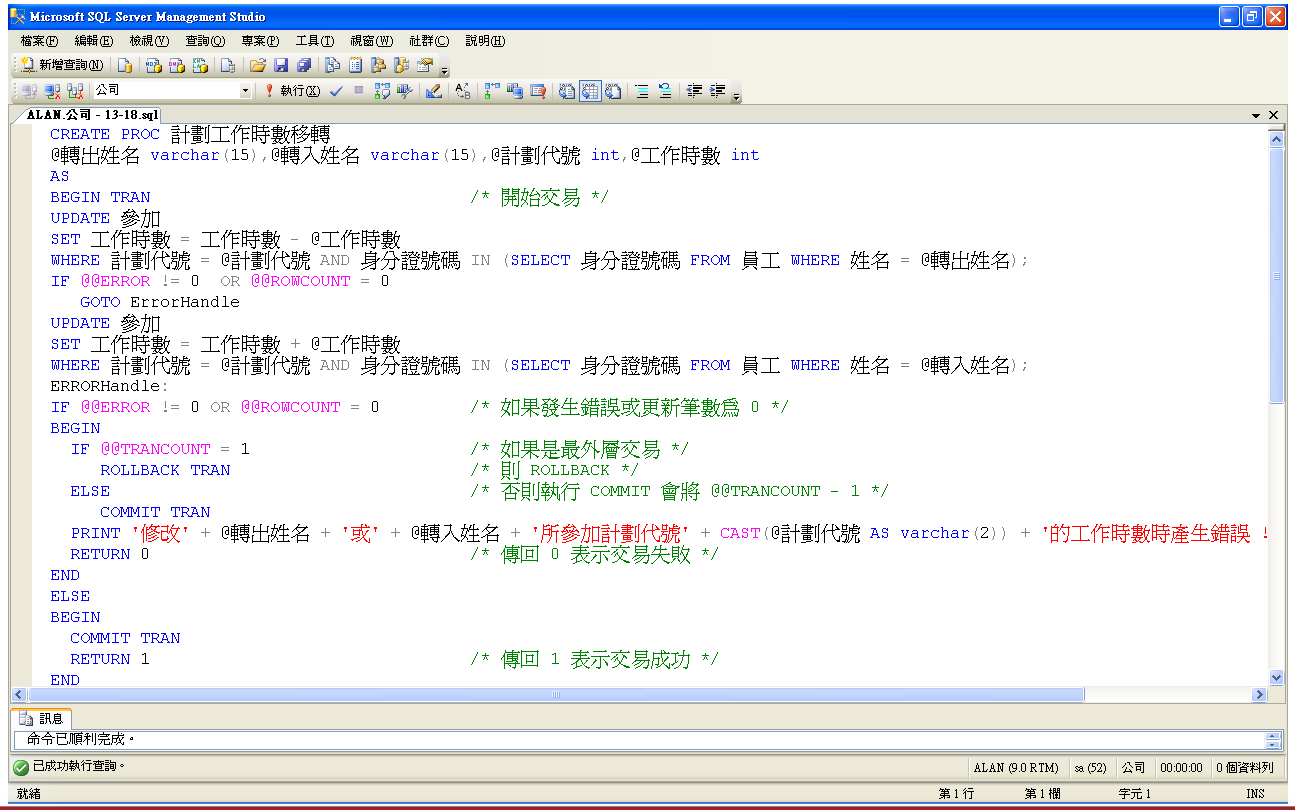


# 巢狀交易 (Nested Transaction)

- 如果交易中還有交易，則稱為「巢狀交易」。
- @@TRANCOUNT 這個「交易計數」是用來計算目前是在巢狀交易的第幾層。
- BEGIN TRANSACTION 敘述會將 @@TRANCOUNT 自動加 1 。
- ROLLBACK TRANSACTION 敘述則會將 @@TRANCOUNT 設定為 0 。
- ROLLBACK TRANSACTION savepoint_name 敘述則不會影響 @@TRANCOUNT 的值。
- COMMIT TRANSACTION 敘述則會將 @@TRANCOUNT 自動減 1。

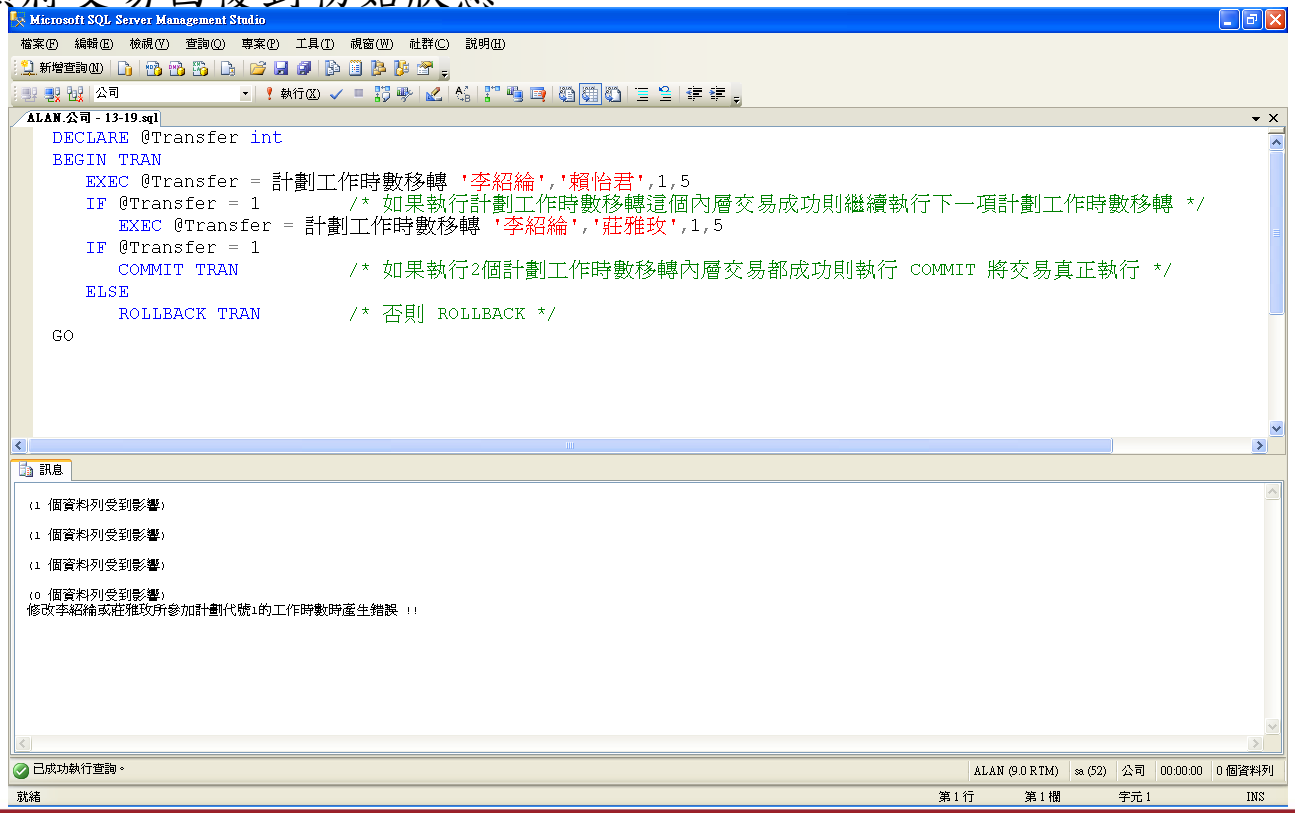
# 【範例 13.3】

- 請在“公司”資料庫中以一個完整交易的架構方式建立一個名為“計劃工作時數移轉”的預存程序，將某位員工所參加的某個計劃代號的某些小時工作時數轉交給另外一位名員工來執行。這個預存程序必須輸入“轉出姓名”、“轉入姓名”、“計劃代號”和“工作時數”等四個參數，並且會回傳一個值，如果回傳 1 表示交易成功，如果回傳值是 0 表示交易失敗，則必須將交易回復到初始狀態



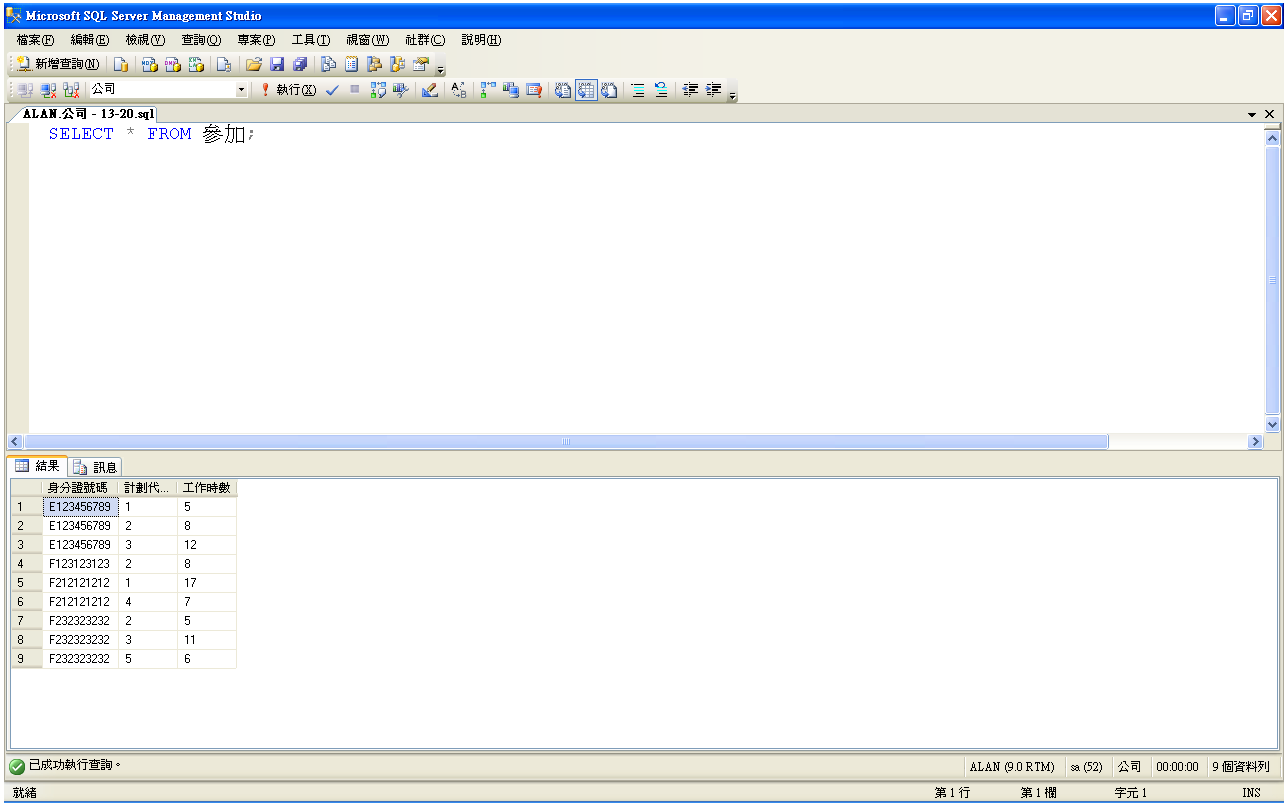
# 【範例 13.4】

- 請在“公司”資料庫中將員工李紹綸所參加計劃代號 1 的 5 小時工作時數轉交由賴怡君這名員工來執行，另外將員工李紹綸所參加計劃代號 1 的 5 小時工作時數再轉交由莊雅玫這名員工來執行，而且這兩次「計劃工作時數移轉」必須視為同一筆交易，請用一個完整交易的架構來執行，如果交易失敗時，則必須將交易回復到初始狀態



# 【範例 13.4】

- 因為第四個 UPDATE 指令執行失敗而 ROLLBACK 回初始狀態



# 分散式交易 (Distributed Transaction)

- 在交易中必須存取多個資料伺服器中的資料或是執行各個伺服器中的預存程序時，就必須使用「分散式交易」。

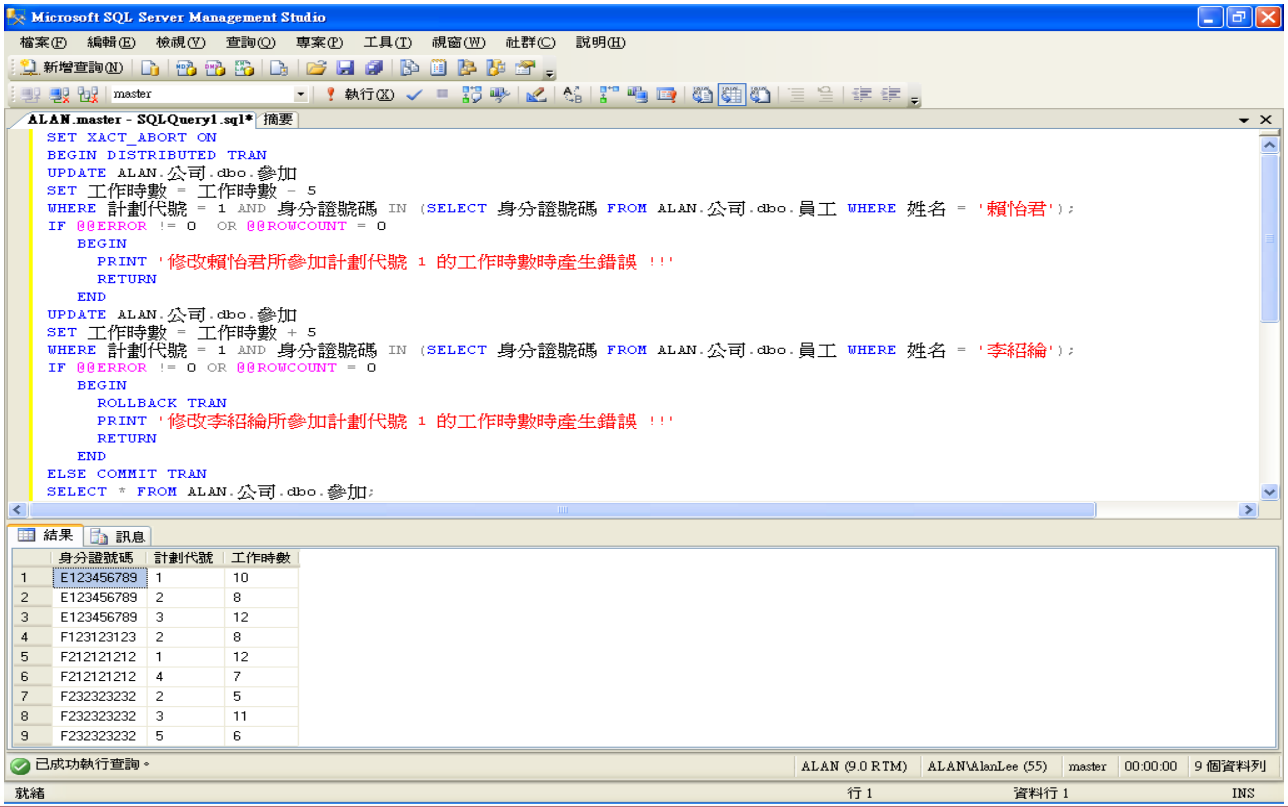
# BEGIN DISTRIBUTED TRANSACTION

- BEGIN DISTRIBUTED TRANSACTION 指令敘述格式如下：

```
BEGIN DISTRIBUTED { TRAN | TRANSACTION }
[ transaction_name | @tran_name_variable ]
[ ; ]
```

# 【範例 13.5】

- 請使用分散式交易方式存取在遠端伺服器 ALAN 中的“公司”資料庫，將員工賴怡君所參加計劃代號 1 的 5 小時工作時數轉交由李紹綸這名員工來執行，請用一個完整交易的架構來執行，如果交易失敗時，則必須將分散式交易回復到初始狀態



# 交易的隔離等級 (Isolation Level)

- 在交易的 ACID 四大特性中，「隔離性」(Isolation) 目的是用來確保交易執行的結果，不會受到其他同時進行的交易所影響。
- 在交易中所使用的資料，會和其他同時進行的交易適度地隔離，因此交易通常會在存取資料之前先將該筆資料予以鎖定，以免受到其他交易的干擾。
- 但是採用「鎖定法」的缺點，除了會使得這些要存取該資料的交易必須排隊等待，因而降低了資料的「並行性」(Concurrency) 之外，還可能造成死結 (Dead Lock)，導致交易的停擺。
- 隔離等級越高能夠確保所讀取資料的正確性，但是其「並行性」就越差。
- 隔離等級越低其「並行性」就越高，但是卻會降低所讀取資料的正確性。

# 5 種交易的隔離等級

- SQL Server 中支援 5 種交易的「隔離等級」，其隔離程度由低到高可分為下列 5 種等級：
  - READ UNCOMMITTED
  - SNAPSHOT
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE

# 設定交易的隔離等級

- SET TRANSACTION ISOLATION LEVEL 指令敘述格式如下：

```
SET TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED
  | READ COMMITTED
  | REPEATABLE READ
  | SNAPSHOT
  | SERIALIZABLE
}
[ ; ]
```

# 交易隔離等級

- 五種交易隔離等級比較圖

| 隔離等級 | 保證不會讀取到別人修改修改中的資料 | 保證已讀取的資料不會被更改 | 保證使用到的資料表不會被更改 |
|---|---|---|---|
| Read Uncommited | ╳ | ╳ | ╳ |
| Snapshot | ○ | ╳ | ╳ |
| Read Committed | ○ | ╳ | ╳ |
| Repeatable Read | ○ | ○ | ╳ |
| Serializable | ○ | ○ | ○ |

# 資料鎖定 (Lock)

- 「鎖定」(Lock) 是將特定的資料暫時鎖住供使用者使用，以防止資料被其他使用者讀取或修改。
- 「鎖定」可以避免一個交易會讀取到其他交易正在更改中的資料，以防止多個交易同時變更同一筆資料。
- SQL Server 可以透過「鎖定」來確保交易的完整性 (Integrity) 和資料庫的一致性 (Consistency)。

# 樂觀和悲觀的並行控制

- 我們可以經由二種觀點來進行「並行控制」(Concurrency Control)：
  - 樂觀的並行控制 (Optimistic Concurrency)
    - 假設會發生資料存取衝突的機率很小
    - 例如 Read Committed 便是採取這種方式
    - SQL Server 預設是使用「樂觀鎖定」
    - 具有較高的「並行性」，且可以讓系統擁有較佳的執行效能
  - 悲觀的並行控制 (Pessimistic Concurrency)
    - 假設會發生資料存取衝突的機率很大
    - 例如 Repeatable Read 便是採取這種方式

# 鎖定的對象

- SQL Server 可以針對不同層級的資源進行鎖定，其鎖定的對象是：

| 資源 | 說明 |
|---|---|
| RID | 以記錄 (Row) 為單位來鎖定 |
| Key | 已設定為索引的欄位 |
| Page | 資料頁或索引頁 (8KB 大小的分頁) |
| Extent | 8 個連續的分頁 Page (SQL Server 配置記憶體給資料頁時的單位) |
| Table | 整個資料表 (包含其中所有的資料和索引) |
| FILE | 資料庫檔案 |
| APPLICATION | 應用程式指定資源 |
| METADATA | 中繼資料鎖定 |
| ALLOCATION_UNIT | 配置單位 |
| DB | 整個資料庫 |

# 鎖定的方法

- 我們還可以採取不同的鎖定方式：
  - 獨占式鎖定 (Exclusive Lock)
    - 獨占式鎖定可以禁止其他交易對資料進行存取或鎖定工作。
  - 共享式鎖定 (Shared Lock)
    - 共享式鎖定可以將資料設成唯讀，並禁止其他交易對該資料進行獨占式鎖定，但是卻允許其他交易對資料再進行共享式鎖定。
  - 更新式鎖定 (Update Lock)
    - 更新式鎖定可以和共享式鎖定共存，但是不可以和其他的更新式鎖定或獨占式鎖定並存。

# 意圖式鎖定

- 意圖式鎖定 (Intent Lock) 表示只想要鎖定物件中某部份的底層資源。
- 意圖式鎖定又可以分為三種：

| 種類 | 說明 |
| --- | --- |
| Intent Shared | 想要共享式鎖定並讀取指定物件中的部份資源。 |
| Intent Exclusive | 想要獨占式鎖定並更改指定物件中的部份資源。 |
| Shared with Intent Exclusive | 想要共享式鎖定並讀取指定物件中的全部資源，同時還要獨占式鎖定並更改指定物件中的部份資源。 |

- 當使用者要獨占式鎖定一個資料表時，SQL server 必須檢查該資料表中的所有資源，包括資料頁和記錄等，查看是否已經被鎖定，以決定該資料是否可以進行獨占式鎖定。
- 由於這是一項非常耗時的工作，所以 SQL server 通常會使用意圖式鎖定來標示某物件中已經有部份資源被鎖定，好讓 SQL server 可以在檢查時直接用來判斷資源是否已經被鎖定。

# 各種鎖定的共存性

- 各種鎖定的共存性比較圖

| 要求進行的鎖定 | 已經存在的鎖定 | | | | | |
|---|---|---|---|---|---|---|
| | IS | S | U | IX | SIX | X |
| Intent Shared (IS) | ○ | ○ | ○ | ○ | ○ | × |
| Shared (S) | ○ | ○ | ○ | × | × | × |
| Update (U) | ○ | ○ | × | × | × | × |
| Intent Exclusive (IX) | ○ | × | × | × | × | × |
| Shared with Intent Exclusive (SIX) | ○ | × | × | × | × | × |
| Exclusive (X) | × | × | × | × | × | × |

# 鎖定的死結問題

- 由於 SQL Server 會定時偵測眾多交易之中是否有死結發生，因此我們並不需要擔心死結問題。
- 如果真的發生死結時，SQL Server 會在這些交易之中選擇一個交易當作犧牲者 (Victim)，強迫該交易進行撤回 (Rollback) 動作，並傳回編號 1205 的錯誤訊息，然後釋放該交易所有鎖定的資源，讓其他交易得以繼續完成交易。
- 如果我們已經知道某些交易並不是很重要或不是很緊急時，那麼我們便可以把這些交易設定為優先的犧牲者。

# SET DEADLOCK_PRIORITY

- SET DEADLOCK_PRIORITY 指令敘述格式如下：

```
SET DEADLOCK_PRIORITY
{ LOW | NORMAL | HIGH | -10 | -9 | -8 | ?| 0 | ?| 8 | 9 | 10 | @deadlock_var | @deadlock_intvar }
```

# 如何避免發生死結

- 以下是幾個在撰寫程式時的技巧：
  - 使用相同的順序來存取資料：假設每個要存取 A、B、C 三個資料表的所有交易都是以 A → B → C 的順序進行存取，那麼就不會發生死結了。
  - 儘量縮短交易的時間：交易的時間越短，佔用資源的時間也越短，而發生死結的機率自然也會減少。
  - 儘量使用較低的隔離等級：採用較低隔離等級的資料鎖定可以提供給更多人同時存取，因此比較不容易發生死結。