頁xiii

目錄

第 1 篇　基礎知識

01 作業系統執行緒排程

1.1　馮 · 諾依曼系統結構 1-1
1.1.1　概述 1-2
1.1.2　電腦五大組成部分 1-3
1.2　CPU 架構 1-5
1.2.1　CPU 的組成部分 1-5
1.2.2　CPU 邏輯結構 1-7
1.2.3　單核心 CPU 的不足 ...1-9
1.2.4　多核心 CPU 架構 1-10
2.1.2　處理程序與執行緒 2-2
2.1.3　執行緒組 2-3
2.1.4　使用者執行緒與守護執行緒 2-4
2.1.5　並行與並行 2-6
2.1.6　同步與非同步 2-7
2.1.7　共享與獨享 2-8
2.1.8　臨界區 2-8
2.1.9　阻塞與非阻塞 2-8
2.2　並行程式設計的風險 2-9
2.2.1　安全性問題 2-9
2.2.2　活躍性問題 2-11

紅框部分改為「平行」

頁1-16

2. 核心級執行緒的優點

（1）核心級執行緒不會引起整個處理程序的阻塞。當處理程序中的某個執行緒阻塞時，作業系統核心可以排程處理程序中的其他執行緒，也可以排程其他處理程序中的執行緒，不會阻塞處理程序。

（2）作業系統核心將同一處理程序的多個執行緒排程到不同的 CPU 核心上執行，能夠大大提高任務執行的並行度，提高程式的執行效率。

3. 核心級執行緒的缺點

（1）核心級執行緒在執行過程中，如果涉及執行緒的阻塞與喚醒，則可能觸發使用者態和核心態的切換。

（2）核心級執行緒的建立和管理都需要作業系統核心來完成，與使用者級執行緒相比，這些操作比較慢。

> **注意**：如今絕大多數作業系統，例如，Windows、mac OS、Linux 等都支援核心級執行緒。

1.3.3 混合級執行緒

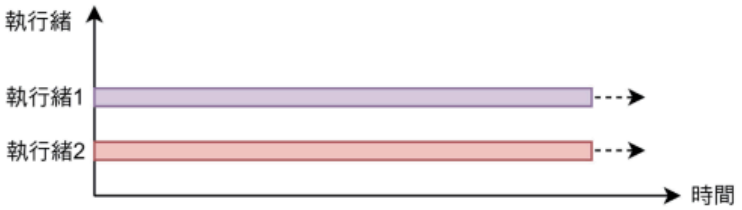
混合級執行緒綜合了使用者級執行緒和核心級執行緒，在使用者空間中建立和管理使用者級執行緒，在核心空間中建立和管理核心級執行緒。可以用圖 1-10 來表示混合級執行緒。

紅框部分改為「平行度」

頁2-6

2.1.5 並行與並行

並行與並行是兩個非常容易混淆的概念。並行指當多核心 CPU 中的一個 CPU 核心執行一個執行緒時，另一個 CPU 核心能夠同時執行另一個執行緒，兩個執行緒之間不會相互先佔 CPU 資源，可以同時執行。圖 2-1 形象地表示了並行的執行流程。



▲ 圖 2-1 並行的執行流程

2-6

紅框處改為「平行」

頁2-7

並行與並行存在如下區別。

（1）並行指多個執行緒在一段時間的每個時刻都同時執行，並髮指多個執行緒在一段時間內（而非每個時刻）同時執行。

（2）並存執行的多個任務之間不會先佔系統資源，並行執行的多個任務會先佔系統資源。

（3）並行只有在多核心 CPU 或者多 CPU 的情況下才會發生，在單核心 CPU 中只可能發生串列執行或者並行執行。

2.1.6 同步與非同步

同步與非同步主要是針對一次方法的呼叫來說的。以同步方式呼叫方法時，必須在方法返回資訊後，才能執行後面的操作。而以非同步方式呼叫方法時，不必等方法返回資訊，就可以執行後面的操作，當完成被呼叫的方法邏輯後，會以通知或者回呼的方式告知呼叫方。

紅框處改為「平行」

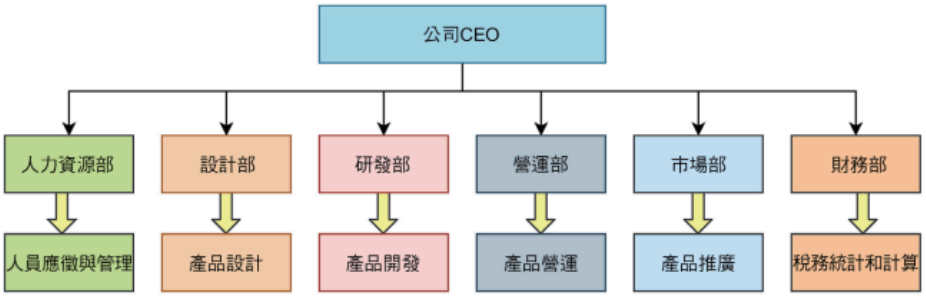
頁2-18

2.4 本章複習

本章主要對並行程式設計進行了概述。首先，簡單介紹了並行程式設計中的基本概念，例如程式、處理程序與執行緒、執行緒組、並行與並行、同步與非同步、臨界區、共享與獨享等。接下來，介紹了並行程式設計中存在的風險，包括安全性問題、活躍性問題和性能問題。最後，簡介了並行程式設計中常見鎖的基本概念，以及在 Java 中的實現方式。

紅框處改為「平行」

頁3-3



▲ 圖 3-2 公司分工後的日常工作

將公司的日常工作分工後，可以發現，各部門之間的工作是可以並行推進的。例如，在人力資源部進行員工的績效考核時，設計部和研發部正在設計和開發公司的產品，與此同時，公司的營運人員正在和設計人員與研發人員溝通如何更好地完善公司的產品，而市場部正在加大力度宣傳和推廣公司的產品，財務部正在統計和計算公司的各種財務報表等。一切都是那麼有條不紊。

所以，在現實生活中，安排合適的人去做合適的事情是非常重要的。映射到並行程式設計領域也是同樣的道理。

紅框處改為「平行」

頁6-25

6.5.2 CPU 重排序

現代 CPU 基本上都支援管線操作，在多核心 CPU 中，為了提高 CPU 的執行效率，管線都是並行的。同時，在不影響程式語義的前提下，CPU 中的處理順序可以和程式的順序不一致，只要滿足 as-if-serial 原則即可。

> **注意**：這裡的不影響程式語義只能保證在程式存在顯性資料依賴關係的情況下，CPU 的處理順序和程式順序一致，不能保證與處理邏輯相關的程式的處理順序和程式順序一致。

CPU 重排序包括指令級重排序和記憶體系統重排序兩部分，如下所示。

（1）指令級重排序指在不影響程式執行的最終結果的前提下，CPU 核心對不存在資料依賴性的指令進行的重排序操作。

（2）記憶體系統重排序指在不影響程式執行的最終結果的前提下，CPU 對儲存在快取記憶體中的資料進行的重排序，記憶體系統重排序雖然可能提升程式的執行效率，但是可能導致資料不一致。

紅框處改為「平行」

頁10-10

10.3 使用 CAS 實現 count++

在前面的章節中，透過 synchronized 實現了執行緒安全的 count++ 操作。本節以另一種 CAS 的方式實現執行緒安全的 count++ 操作。

10.3.1 案例分析

CAS 演算法是一種無鎖程式設計演算法，能夠在無鎖的情況下實現執行緒安全的自動增加操作。在本節的案例中，設定 20 個執行緒並行執行，每個執行緒都透過 CAS 自旋的方式對一個共享變數的資料進行自動增加操作，每個執行緒執行的次數為 500 次，最終得出的結果為 10000。

紅框處改為「平行」

頁11-8

如果所有的轉帳操作都是串列執行的，就會造成只有帳戶 M 向帳戶 N 轉帳完成後，帳戶 X 才能向帳戶 Y 轉帳的問題。在一個銀行的交易系統中，時時刻刻都存在轉帳操作，如果這些轉帳操作都是串列執行的，那麼顯然是不可接受的。

所以，在執行轉帳操作時，最好是讓帳戶 M 向帳戶 N 轉帳與帳戶 X 向帳戶 Y 轉帳這兩個操作能夠並存執行。

11.2.3 發生鎖死

對轉帳操作的程式進一步分析得知，在 transferMoney() 方法中，涉及兩個不同的資源，分別是轉出帳戶 this 和轉入帳戶 targetAccount。試想，如果對轉出帳戶 this 和轉入帳戶 targetAccount 分別加鎖，那麼只有對兩個帳戶加鎖都成功才執行轉帳操作，這樣是不是就能夠使帳戶 M 向帳戶 N 轉帳的操作與帳戶 X 向帳戶 Y 轉帳的操作並存執行呢？

對轉出帳戶 this 和轉入帳戶 targetAccount 分別加鎖的程式如下。

```
/**
 * @author binghe
 * @version 1.0.0
 * @description 轉帳時發生鎖死
 */
public class DeadLockTransferAccount {
    //帳戶餘額
    private long balance;

    public void transferMoney(DeadLockTransferAccount targetAccount, long
transferMoney){
        synchronized (this){
```


紅框處改為「平行」

頁11-9

```
                synchronized (targetAccount){
                    if (this.balance >= transferMoney){
                        this.balance -= transferMoney;
                        targetAccount.balance += transferMoney;
                    }
                }
            }
        }
    }
```

其實，儘管上述程式對轉出帳戶 this 和轉入帳戶 targetAccount 分別進行了加鎖操作，但是這不僅不能使帳戶 M 向帳戶 N 轉帳的操作與帳戶 X 向帳戶 Y 轉帳的操作並存執行，甚至還會引起鎖死的問題。

根據上述程式分析這樣一個場景：假設存在執行緒 A 和執行緒 B 兩個執行緒，執行緒 A 和執行緒 B 分別在兩個不同的 CPU 上執行，執行緒 A 執行帳戶 X 向帳戶 Y 轉帳的操作，執行緒 B 執行帳戶 Y 向帳戶 X 轉帳的操作。

紅框處改為「平行」

頁12-1

在並行程式設計中，鎖能夠有效地保護臨界區的資源。但是，加鎖使得原本能夠並存執行的操作變得序列化，串列操作會降低系統的性能，CPU 對於執行緒的上下文切換也會降低系統的性能。因此，需要在並行程式設計中對使用的鎖進行一定的最佳化。本章對鎖最佳化的一些方案和技巧進行簡單的介紹。

本章相關的基礎知識如下。

- 縮小鎖的範圍。
- 減小鎖的細微性。
- 鎖分離。
- 鎖分段。
- 鎖粗化。
- 避免熱點區域。
- 獨佔鎖的替換方案。
- 其他最佳化方案。

紅框處改為「平行」

頁12-2

12.1 縮小鎖的範圍

縮小鎖的範圍在一定程度上就是縮短持有鎖的時間。最簡單的做法就是將一些不會產生執行緒安全問題的程式移到同步程式區塊之外，尤其要注意的是不會產生執行緒安全問題的 I/O 操作等非常耗時的操作，或者有可能引起阻塞的操作，這些操作儘量放在同步程式區塊之外執行，這樣能夠有效提高程式執行的並行度，從而進一步提升系統的性能。

例如，最佳化前的程式如下。

```
private void callSafeMethod1(){

}
```

紅框處改為「平行」

頁12-3

syncMethod() 方法使用 synchronized 修飾，也就是說，執行緒在執行 syncMethod() 方法時，會對整個方法區塊增加 synchronized 鎖，即使 syncMethod() 方法中存在執行緒安全的程式片段，其他執行緒也只能等待當前執行緒執行完 syncMethod() 方法的所有邏輯並退出 syncMethod() 方法後才能執行 syncMethod() 方法區塊的邏輯。

換句話說，在執行 syncMethod() 方法區塊的邏輯時，整個方法區塊都是以序列化的方式來執行的。這在一定程度上降低了程式的並行度，從而降低了系統的執行性能。

此時，可以縮小 syncMethod() 方法中鎖的範圍，在 syncMethod() 方法區塊中，只對執行緒不安全的 callUsafeMethod() 方法片段加鎖，執行緒安全的 callSafeMethod1() 方法和 callSafeMethod2() 方法會被移到同步程式區塊之外，最佳化後的程式如下。

```
private void callSafeMethod1(){
```

紅框處改為「平行」

頁12-4

}

最佳化後的程式只對執行緒不安全的 callUsafeMethod() 方法片段加鎖，多個執行緒可以同時執行 syncMethod() 方法中的 callSafeMethod1() 方法，而不必等到當前執行緒完全退出 syncMethod() 方法後再執行 syncMethod() 方法的邏輯，提高了程式執行的並行度，進而提升了系統的性能。

12.2 減小鎖的細微性

減小鎖細微性在一定程度上就是縮小鎖定物件的範圍，將一個大物件拆分成多個小物件，對這些小物件進行加鎖，能夠提高程式的並行度，減少鎖的競爭。

如果在某個應用中只存在一個全域鎖，那麼執行緒會以序列化的方式執行被鎖定的同步程式區塊。此時，在並行環境下，很多執行緒會同時競爭這個全域鎖，程式的性能會急劇下降。如果將對這個全域鎖的請求分佈到更多的鎖上，就可以有效降低鎖的競爭程度，此時由於競爭鎖而造成阻塞的執行緒也會更少，從而提高系統的並行度，進一步提升系統的性能。

例如，最佳化前的程式如下。

紅框處改為「平行」

頁12-5

}

可以看到，上述程式中的每個方法都使用了 synchronized 關鍵字修飾，也就是說，對每個方法都增加了 synchronized 鎖，並且鎖物件是當前類別的物件 this。此時，多個執行緒執行同一個類別物件中的任意一個方法都會發生鎖競爭，影響程式的執行性能。

可以分別針對 userSet 集合與 orderSet 集合加鎖，這樣可以縮小鎖的細微性，提升程式的並行度，最佳化後的程式如下。

```
private Set<String> userSet = new HashSet<>();
private Set<String> orderSet = new HashSet<>();

public void addUser(String user){
```

紅框處改為「平行」

頁12-6

```
    }
}
public void removeOrder(String order){
   synchronized (orderSet){
      orderSet.remove(order);
   }
}
```

上述最佳化後的程式在操作 userSet 集合時，就對 userSet 物件加鎖；在操作 orderSet 集合時，就對 orderSet 物件加鎖。這樣，當某個執行緒執行操作 userSet 集合的方法時，其他執行緒在同一時刻能夠執行操作 orderSet 集合的方法，提高了程式的並行度，進一步提升了系統的性能。

12.3 鎖分離

紅框處改為「平行」

頁12-7

在 ConcurrentHashMap 的實現中，使用了一個包含 16 個鎖的陣列，每個鎖保護 1/16 的資料段，其中第 *N* 個資料段交給第 *N*%16（*N* 對 16 取餘）個鎖保護。這樣，當多個執行緒存取不同資料段的資料時，執行緒間就不會發生鎖競爭，從而提高並行存取的效率。

同時，ConcurrentHashMap 將資料按照不同的資料段進行儲存，並為每一個資料段分配一把鎖，當某個執行緒佔有某個資料段的鎖存取資料時，其他資料段的鎖也能被其他執行緒先佔到，即其他資料段的資料也能被其他執行緒存取，提高了程式的並行度，從而提升了性能。

12.5 鎖粗化

紅框處改為「平行」

頁13-7

6. Executors.newWorkStealingPool 方法

當呼叫 Executors.newWorkStealingPool 方法建立執行緒池時，表示建立一個具有並行等級的執行緒池。此方法是 JDK 1.8 新增的方法，能夠為執行緒池設定並行等級，具有比透過 Executors 類別中的其他方法建立的執行緒池更高的並行度和性能。

使用 Executors.newWorkStealingPool 方法建立執行緒池的形式如下。

紅框處改為「平行」

頁20-4

資料庫儲存，單張表的資料分散儲存到多張表中，並且將讀取資料的操作和寫入資料的操作分離。這樣，在架構層面能夠透過增加機器來提升系統的並行處理能力。

紅框處改為「平行」