期末考題重點筆記

一、Prolog 與啟發式搜尋

1. 狀態空間表示

在 Prolog 中建模問題通常需定義狀態(space)與後繼關係(successor relation)。常見的寫法是 `s(X,Y,C)`，其中 `X`、`Y` 為狀態，`C` 為從 `X` 移至 `Y` 的成本。這種關係可透過事實或規則描述，並搭配搜尋演算法實現問題求解。為讓程式能自動探索可能狀態，我們也會定義目標狀態檢查，例如 `goal(State)`。

實際例子：八宮格問題

```
% 定義狀態轉換
s(State1, State2, 1) :-
    move(State1, State2).

% 定義移動規則
move(State1, State2) :-
    append(Left, [0|Right], State1),
    append(Left, [X|Right], State2),
    member(X, [1,2,3,4,5,6,7,8]).

% 定義目標狀態
goal([1,2,3,4,5,6,7,8,0]).
```

可能考題：

1. 請實作一個 Prolog 程式來解決八宮格問題，使用 A* 演算法。
2. 說明如何定義狀態空間和後繼關係，並解釋為什麼這樣定義是合理的。

2. A* 演算法與評估函式

A* 演算法結合了成本累積函式 `g(n)` 與啟發式函式 `h(n)`，形成評估函式 `f(n) = g(n) + h(n)`。其中 `g(n)` 代表從起點到目前節點 `n` 的實際耗費，而 `h(n)` 則預估從 `n` 到目標的成本。若 `h(n)` 永遠不高於真實成本，則稱為「可容許」(admissible)，此時 A* 搜尋可以保證找到最短路徑。

實際例子：八宮格啟發式函式

```
% 曼哈頓距離啟發式
h(State, H) :-
    manhattan_distance(State, H).

manhattan_distance(State, H) :-
    findall(D, (nth1(Pos, State, X), X \= 0,
                goal_position(X, GoalPos),
                manhattan_dist(Pos, GoalPos, D)), Distances),
    sum_list(Distances, H).
```

```
% 計算兩個位置間的曼哈頓距離
manhattan_dist(Pos1, Pos2, D) :-
    X1 is (Pos1-1) mod 3,
    Y1 is (Pos1-1) // 3,
    X2 is (Pos2-1) mod 3,
    Y2 is (Pos2-1) // 3,
    D is abs(X1-X2) + abs(Y1-Y2).
```

可能考題：

1. 請解釋為什麼曼哈頓距離是一個可容許的啟發式函式。
2. 比較不同啟發式函式（如曼哈頓距離、錯位方塊數）的優缺點。
3. 實作一個結合多個啟發式的函式，並證明其可容許性。

3. 啟發式設計與可容許性

以八宮格(8-puzzle)為例，常見的啟發式包含：

- 曼哈頓距離：計算每個方塊離目標位置的水平距離與垂直距離之和。
- tile out of place：統計錯位方塊數量。
- 結合多項評估：例如將曼哈頓距離加上順序分數(sequence score)等。

實際例子：結合多個啟發式

```
% 結合曼哈頓距離和順序分數的啟發式
h_combined(State, H) :-
    manhattan_distance(State, H1),
    sequence_score(State, H2),
    H is H1 + 3 * H2.

% 計算順序分數
sequence_score(State, Score) :-
    findall(S, (nth1(Pos, State, X), X \= 0,
                sequence_score_at(Pos, X, S)), Scores),
    sum_list(Scores, Score).

% 計算特定位置的順序分數
sequence_score_at(Pos, X, 1) :-
    Pos = 5.  % 中心位置
sequence_score_at(Pos, X, 2) :-
    edge_position(Pos),
    \+ proper_successor(Pos, X).
```

可能考題：

1. 證明上述結合啟發式是可容許的。
2. 分析不同啟發式組合對搜尋效率的影響。
3. 設計一個新的啟發式函式，並證明其可容許性。

4. IDA* 與其他變形

當搜尋空間龐大時，A* 需要儲存大量節點，容易導致記憶體爆炸。IDA*(Iterative Deepening A*) 透過逐漸提高 `f` 上限來控制展開深度，重複以深度優先方式搜尋，比傳統 A* 更省空間。

實際例子：*IDA 實現*

```
% IDA* 基本實現
ida_star(Start, Solution) :-
    f(Start, F),
    ida_star_search([], Start, F, Solution).

ida_star_search(Path, State, Bound, Solution) :-
    f(State, F),
    F =< Bound,
    (goal(State) ->
        Solution = [State|Path]
    ;   findall(Next, (s(State, Next, _), \+ member(Next, Path)), Children),
        ida_star_search_children(Children, [State|Path], Bound, Solution)
    ).

ida_star_search_children([], _, _, _) :- fail.
ida_star_search_children([Child|Rest], Path, Bound, Solution) :-
    (ida_star_search(Path, Child, Bound, Solution) ->
        true
    ;   ida_star_search_children(Rest, Path, Bound, Solution)
    ).
```

可能考題：

1. 比較 A* 和 IDA* 的優缺點。
2. 在什麼情況下 IDA* 會比 A* 更有效率？
3. 實作一個結合 IDA* 和 Alpha 剪枝的搜尋演算法。

5. 排程問題的 A*

在多處理器排程問題中，狀態可以描述為每個處理器當前完成時間及剩餘任務集合。啟發式 `h(n)` 可能採用「所有剩餘任務平均分配到處理器後預估的最終完成時間」，進一步與目前最大完成時間 `Fin` 取差值。

實際例子：排程問題啟發式

```
% 排程問題的啟發式函式
h_schedule(State, H) :-
    current_finish_time(State, Fin),
    remaining_tasks(State, Tasks),
    total_workload(Tasks, TotalWork),
    num_processors(N),
    FinAll is TotalWork / N,
    H is max(FinAll - Fin, 0).

% 計算當前最大完成時間
current_finish_time(State, Fin) :-
```

```
    findall(Time, processor_finish_time(State, Time), Times),
    max_list(Times, Fin).

% 計算剩餘工作總量
total_workload(Tasks, Total) :-
    findall(Duration, task_duration(Tasks, Duration), Durations),
    sum_list(Durations, Total).
```

可能考題：

1. 證明上述排程問題啟發式是可容許的。
2. 分析不同排程策略對啟發式函式設計的影響。
3. 實作一個考慮任務優先級的啟發式函式。

二、併發程式理論

1. 進程、同步與通訊

論文《Concepts and Notations for Concurrent Programming》總結了併發程式的核心概念：

- **進程(Process)**：獨立的執行單元，可視為「執行緒」或「任務」。
- **通訊(Communication)**：進程之間交換資料的方式，可透過共享變數或訊息傳遞。
- **同步(Synchronization)**：約束進程執行順序的手段，避免競爭或確保條件被滿足。

實際例子：Go 語言中的進程與通訊

```
// 使用 goroutine 和 channel 實現進程間通訊
func producer(ch chan<- int) {
    for i := 0; i < 10; i++ {
        ch <- i  // 發送資料到通道
        time.Sleep(time.Millisecond * 100)
    }
    close(ch)  // 關閉通道表示結束
}

func consumer(ch <-chan int, done chan<- bool) {
    for num := range ch {  // 從通道接收資料
        fmt.Printf("Received: %d\n", num)
    }
    done <- true  // 通知完成
}

func main() {
    ch := make(chan int)     // 無緩衝通道
    done := make(chan bool)  // 同步通道

    go producer(ch)
    go consumer(ch, done)

    <-done  // 等待消費者完成
}
```

可能考題：

1. 解釋 Go 語言中 goroutine 和 channel 的關係。
2. 比較共享記憶體和訊息傳遞兩種通訊方式的優缺點。
3. 實作一個使用 channel 的生產者-消費者模式。

2. Busy-Waiting 與其缺點

早期的同步策略多利用忙等，即進程不斷輪詢共享變數以等待條件。這種方式雖易於理解，卻導致 CPU 空轉。

實際例子：Busy-Waiting vs Channel

```
// Busy-Waiting 方式
var flag bool

func busyWait() {
    for !flag {  // 持續檢查
        // 空轉
    }
    // 執行後續操作
}

// Channel 方式
func channelWait(done <-chan struct{}) {
    <-done  // 阻塞等待
    // 執行後續操作
}
```

可能考題：

1. 分析 Busy-Waiting 的效能問題。
2. 說明為什麼 channel 是更好的同步機制。
3. 實作一個避免 Busy-Waiting 的同步機制。

3. Semaphores 與範例

Semaphore 提供 `P` (嘗試進入) 與 `V` (釋放) 兩操作。若 semaphore 值為 0，執行 `P` 的進程會被阻塞，直到其他進程 `V`。

實際例子：Go 中的 Semaphore 實現

```
type Semaphore struct {
    ch chan struct{}
}

func NewSemaphore(n int) *Semaphore {
    return &Semaphore{
        ch: make(chan struct{}, n),
```

```
    }
}

func (s *Semaphore) P() {
    s.ch <- struct{}{}  // 獲取信號量
}

func (s *Semaphore) V() {
    <-s.ch  // 釋放信號量
}

// 使用範例
func main() {
    sem := NewSemaphore(2)  // 允許兩個並發

    for i := 0; i < 5; i++ {
        go func(id int) {
            sem.P()
            defer sem.V()

            fmt.Printf("Worker %d started\n", id)
            time.Sleep(time.Second)
            fmt.Printf("Worker %d finished\n", id)
        }(i)
    }

    time.Sleep(time.Second * 3)
}
```

可能考題：

1. 使用 semaphore 實作一個讀寫鎖。
2. 解釋 semaphore 和 mutex 的區別。
3. 實作一個限制並發數的資源池。

4. 死鎖與公平

良好設計的同步機制需避免死鎖 (deadlock)。例如兩個進程同時持有對方所需的資源，若不釋放就會互相等待。

實際例子：死鎖檢測與預防

```
// 可能導致死鎖的程式
func deadlockExample() {
    var mu1, mu2 sync.Mutex

    go func() {
        mu1.Lock()
        defer mu1.Unlock()

        time.Sleep(time.Millisecond * 100)
```

```
        mu2.Lock()
        defer mu2.Unlock()
    }()

    go func() {
        mu2.Lock()
        defer mu2.Unlock()

        time.Sleep(time.Millisecond * 100)

        mu1.Lock()
        defer mu1.Unlock()
    }()
}

// 避免死鎖的版本
func safeExample() {
    var mu1, mu2 sync.Mutex

    // 確保鎖的獲取順序一致
    go func() {
        mu1.Lock()
        defer mu1.Unlock()

        mu2.Lock()
        defer mu2.Unlock()
    }()

    go func() {
        mu1.Lock()
        defer mu1.Unlock()

        mu2.Lock()
        defer mu2.Unlock()
    }()
}
```

可能考題：

1. 分析上述程式中的死鎖問題。
2. 實作一個死鎖檢測機制。
3. 設計一個避免死鎖的資源分配策略。

5. 訊息傳遞模式

論文也討論以訊息交換 (message passing) 實作同步，例如 CSP (Communicating Sequential Processes)。

實際例子：CSP 風格的 Go 程式

```go
// 使用 channel 實現 CSP 風格的程式
type Process struct {
    in  <-chan int
    out chan<- int
}

func (p *Process) Run() {
    for x := range p.in {
        // 處理資料
        result := x * 2
        p.out <- result
    }
    close(p.out)
}

func main() {
    // 建立處理管道
    ch1 := make(chan int)
    ch2 := make(chan int)

    // 建立處理程序
    p1 := &Process{in: ch1, out: ch2}
    p2 := &Process{in: ch2, out: nil}

    // 啟動處理程序
    go p1.Run()
    go p2.Run()

    // 發送資料
    for i := 0; i < 5; i++ {
        ch1 <- i
    }
    close(ch1)
}
```

可能考題：

1. 比較 CSP 和 Actor 模型的異同。
2. 實作一個基於 channel 的管道處理系統。
3. 設計一個分散式系統的訊息傳遞機制。

三、Go 語言併發特性

1. Goroutine 基本概念

Goroutine 是由 Go 執行時管理的輕量級執行緒。啟動 goroutine 時只需在函式呼叫前加 `go`，不必像傳統執行緒般預先配置棧大小。

實際例子：Goroutine 使用

```
// 基本 goroutine 使用
func main() {
    // 啟動多個 goroutine
    for i := 0; i < 3; i++ {
        go func(id int) {
            fmt.Printf("Goroutine %d started\n", id)
            time.Sleep(time.Second)
            fmt.Printf("Goroutine %d finished\n", id)
        }(i)
    }

    // 等待所有 goroutine 完成
    time.Sleep(time.Second * 2)
}

// 使用 WaitGroup 同步
func main() {
    var wg sync.WaitGroup

    for i := 0; i < 3; i++ {
        wg.Add(1)
        go func(id int) {
            defer wg.Done()
            fmt.Printf("Goroutine %d started\n", id)
            time.Sleep(time.Second)
            fmt.Printf("Goroutine %d finished\n", id)
        }(i)
    }

    wg.Wait()
}
```

可能考題：

1. 解釋 goroutine 和作業系統執行緒的區別。
2. 實作一個使用 goroutine 的並行處理系統。
3. 分析 goroutine 的記憶體使用和效能特性。

2. 通道(Channel) 與同步

Channel 是 goroutine 溝通的核心工具。創建通道常用 `make(chan T)`，亦可給定緩衝大小 `make(chan T, n)`。

實際例子：Channel 使用模式

```
// 無緩衝通道
func unbufferedChannel() {
    ch := make(chan int)

    go func() {
```

```
        ch <- 1  // 發送會阻塞直到接收
    }()

    x := <-ch  // 接收會阻塞直到發送
    fmt.Println(x)
}

// 緩衝通道
func bufferedChannel() {
    ch := make(chan int, 2)

    ch <- 1  // 不會阻塞
    ch <- 2  // 不會阻塞
    // ch <- 3  // 會阻塞，因為緩衝已滿

    fmt.Println(<-ch)  // 1
    fmt.Println(<-ch)  // 2
}

// 關閉通道
func closeChannel() {
    ch := make(chan int)

    go func() {
        for i := 0; i < 5; i++ {
            ch <- i
        }
        close(ch)  // 關閉通道
    }()

    // 使用 range 接收直到通道關閉
    for x := range ch {
        fmt.Println(x)
    }
}
```

可能考題：

1. 比較無緩衝和緩衝通道的異同。
2. 實作一個使用通道的並行處理管道。
3. 設計一個基於通道的任務排程系統。

3. 選擇(select) 與多路複用

`select` 陳述式允許 goroutine 同時監聽多個通道，根據就緒情況選擇執行。

實際例子：Select 使用模式

```
// 基本 select
func basicSelect() {
    ch1 := make(chan int)
```

```
    ch2 := make(chan int)

    go func() {
        time.Sleep(time.Second)
        ch1 <- 1
    }()

    go func() {
        time.Sleep(time.Second * 2)
        ch2 <- 2
    }()

    select {
    case x := <-ch1:
        fmt.Println("Received from ch1:", x)
    case x := <-ch2:
        fmt.Println("Received from ch2:", x)
    }
}

// 超時控制
func timeoutSelect() {
    ch := make(chan int)

    go func() {
        time.Sleep(time.Second * 2)
        ch <- 1
    }()

    select {
    case x := <-ch:
        fmt.Println("Received:", x)
    case <-time.After(time.Second):
        fmt.Println("Timeout!")
    }
}

// 非阻塞操作
func nonBlockingSelect() {
    ch := make(chan int)

    select {
    case x := <-ch:
        fmt.Println("Received:", x)
    default:
        fmt.Println("No data available")
    }
}
```

可能考題：

1. 使用 select 實作一個超時控制機制。

2. 設計一個基於 select 的並行處理系統。
3. 實作一個使用 select 的負載平衡器。

4. WaitGroup 與工作同步

當需要等待多個 goroutine 結束時，可使用 `sync.WaitGroup`。

實際例子：WaitGroup 使用模式

```
// 基本 WaitGroup
func basicWaitGroup() {
    var wg sync.WaitGroup

    for i := 0; i < 3; i++ {
        wg.Add(1)
        go func(id int) {
            defer wg.Done()
            fmt.Printf("Worker %d started\n", id)
            time.Sleep(time.Second)
            fmt.Printf("Worker %d finished\n", id)
        }(i)
    }

    wg.Wait()
    fmt.Println("All workers finished")
}

// 錯誤處理
func errorHandlingWaitGroup() {
    var wg sync.WaitGroup
    errChan := make(chan error, 3)

    for i := 0; i < 3; i++ {
        wg.Add(1)
        go func(id int) {
            defer wg.Done()
            if err := doWork(id); err != nil {
                errChan <- err
            }
        }(i)
    }

    // 等待所有工作完成
    go func() {
        wg.Wait()
        close(errChan)
    }()

    // 收集錯誤
    for err := range errChan {
        fmt.Println("Error:", err)
```

```
    }
}
```

可能考題：

1. 使用 WaitGroup 實作一個並行處理系統。
2. 設計一個帶錯誤處理的 WaitGroup 模式。
3. 實作一個使用 WaitGroup 的任務池。

5. Mutex 與共享變數

若必須在多個 goroutine 間共享資料，可使用 `sync.Mutex`。

實際例子：Mutex 使用模式

```go
// 基本 Mutex
type SafeCounter struct {
    mu    sync.Mutex
    count int
}

func (c *SafeCounter) Increment() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.count++
}

func (c *SafeCounter) GetCount() int {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.count
}

// 讀寫鎖
type SafeMap struct {
    mu    sync.RWMutex
    data  map[string]int
}

func (m *SafeMap) Get(key string) int {
    m.mu.RLock()
    defer m.mu.RUnlock()
    return m.data[key]
}

func (m *SafeMap) Set(key string, value int) {
    m.mu.Lock()
    defer m.mu.Unlock()
    m.data[key] = value
}
```

可能考題：

1. 比較 Mutex 和 RWMutex 的異同。
2. 實作一個線程安全的資料結構。
3. 設計一個使用 Mutex 的快取系統。

6. 記憶體模型與同步

現代 CPU 可能將寫入暫存於快取，導致不同核心觀察到的資料不一致。

實際例子：記憶體模型問題

```
// 可能出現記憶體問題的程式
func memoryModelProblem() {
    var x, y int

    go func() {
        x = 1
        y = 1
    }()

    go func() {
        r1 := y
        r2 := x
        fmt.Println(r1, r2)
    }()
}

// 使用同步原語確保記憶體一致性
func memoryModelSolution() {
    var x, y int
    var mu sync.Mutex

    go func() {
        mu.Lock()
        x = 1
        y = 1
        mu.Unlock()
    }()

    go func() {
        mu.Lock()
        r1 := y
        r2 := x
        mu.Unlock()
        fmt.Println(r1, r2)
    }()
}
```

可能考題：

1. 解釋 Go 的記憶體模型。
2. 分析並修復記憶體一致性問題。
3. 設計一個保證記憶體一致性的並行系統。

7. Goroutine 洩漏與取消

若 goroutine 因無人接收通道而永久阻塞，就會形成 goroutine 洩漏。

實際例子：Goroutine 生命週期管理

```
// 使用 done 通道控制生命週期
func lifecycleManagement() {
    done := make(chan struct{})

    go func() {
        for {
            select {
            case <-done:
                return
            default:
                // 執行任務
                time.Sleep(time.Millisecond * 100)
            }
        }
    }()

    // 停止 goroutine
    close(done)
}

// 使用 context 控制生命週期
func contextManagement() {
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()

    go func() {
        for {
            select {
            case <-ctx.Done():
                return
            default:
                // 執行任務
                time.Sleep(time.Millisecond * 100)
            }
        }
    }()
}
```

可能考題：

1. 分析並修復 goroutine 洩漏問題。

2. 實作一個可取消的並行處理系統。
3. 設計一個使用 context 的任務管理系統。

8. 綜合範例：併發爬蟲

結合前述技巧，可建立一個具限速與取消功能的並行網頁爬蟲：使用計數 semaphore 控制同時下載連結的數量，並在達到深度限制或收到取消信號時關閉通道，確保所有 goroutine 能順利退出。此例展示了 channels、goroutines、select 與 WaitGroup 的協同運用。

四、範例解答

1. A* 演算法完整實現

```
% A* 演算法的完整實現
solve(Start, Path, Cost) :-
    astar([[Start,0]], [], Path, Cost).

astar([[State,G]|_], _, [State], G) :-
    goal(State).

astar([[State,G]|RestOpen], Closed, Path, Cost) :-
    findall([Next,G1],
        (s(State,Next,C),
         \+ member(Next,Closed),
         G1 is G+C),
        Children),
    append(RestOpen, Children, Open1),
    sort(2, @=<, Open1, OpenSorted),
    astar(OpenSorted, [State|Closed], PathRest, Cost),
    Path = [State|PathRest].

% 八宮格問題的完整實現
s(State1, State2, 1) :-
    move(State1, State2).

move(State1, State2) :-
    append(Left, [0|Right], State1),
    append(Left, [X|Right], State2),
    member(X, [1,2,3,4,5,6,7,8]).

goal([1,2,3,4,5,6,7,8,0]).

% 曼哈頓距離啟發式
h(State, H) :-
    manhattan_distance(State, H).

manhattan_distance(State, H) :-
    findall(D,
        (nth1(Pos, State, X),
         X \= 0,
         goal_position(X, GoalPos),
         manhattan_dist(Pos, GoalPos, D)),
```

```
        Distances),
    sum_list(Distances, H).

manhattan_dist(Pos1, Pos2, D) :-
    X1 is (Pos1-1) mod 3,
    Y1 is (Pos1-1) // 3,
    X2 is (Pos2-1) mod 3,
    Y2 is (Pos2-1) // 3,
    D is abs(X1-X2) + abs(Y1-Y2).

% 使用範例
?- solve([2,8,3,1,6,4,7,0,5], Path, Cost).
```

2. 併發程式範例解答

2.1 生產者-消費者模式

```
// 完整的生產者-消費者實現
type Producer struct {
    ch chan<- int
    done chan<- bool
}

func (p *Producer) Run() {
    for i := 0; i < 10; i++ {
        p.ch <- i
        time.Sleep(time.Millisecond * 100)
    }
    p.done <- true
}

type Consumer struct {
    ch <-chan int
    done <-chan bool
    wg *sync.WaitGroup
}

func (c *Consumer) Run() {
    defer c.wg.Done()
    for {
        select {
        case x := <-c.ch:
            fmt.Printf("Consumed: %d\n", x)
        case <-c.done:
            return
        }
    }
}

func main() {
    ch := make(chan int, 5)  // 緩衝通道
```

```
    done := make(chan bool)
    var wg sync.WaitGroup

    producer := &Producer{ch: ch, done: done}
    consumer := &Consumer{ch: ch, done: done, wg: &wg}

    wg.Add(1)
    go producer.Run()
    go consumer.Run()

    wg.Wait()
}
```

2.2 讀寫鎖實現

```
// 使用 semaphore 實現讀寫鎖
type RWLock struct {
    readers int
    mu      sync.Mutex
    writer  chan struct{}
}

func NewRWLock() *RWLock {
    return &RWLock{
        writer: make(chan struct{}, 1),
    }
}

func (l *RWLock) RLock() {
    l.mu.Lock()
    l.readers++
    if l.readers == 1 {
        l.writer <- struct{}{}  // 第一個讀者獲取寫鎖
    }
    l.mu.Unlock()
}

func (l *RWLock) RUnlock() {
    l.mu.Lock()
    l.readers--
    if l.readers == 0 {
        <-l.writer  // 最後一個讀者釋放寫鎖
    }
    l.mu.Unlock()
}

func (l *RWLock) Lock() {
    l.writer <- struct{}{}  // 獲取寫鎖
}

func (l *RWLock) Unlock() {
```

```
    <-l.writer  // 釋放寫鎖
}
```

2.3 任務池實現

```
// 完整的任務池實現
type Task struct {
    ID     int
    Data   interface{}
    Result interface{}
    Error  error
}

type TaskPool struct {
    tasks    chan *Task
    results  chan *Task
    workers  int
    wg       sync.WaitGroup
}

func NewTaskPool(workers int) *TaskPool {
    return &TaskPool{
        tasks:    make(chan *Task),
        results:  make(chan *Task),
        workers:  workers,
    }
}

func (p *TaskPool) Start() {
    for i := 0; i < p.workers; i++ {
        p.wg.Add(1)
        go p.worker()
    }
}

func (p *TaskPool) worker() {
    defer p.wg.Done()
    for task := range p.tasks {
        // 處理任務
        result, err := processTask(task)
        task.Result = result
        task.Error = err
        p.results <- task
    }
}

func (p *TaskPool) Submit(task *Task) {
    p.tasks <- task
}

func (p *TaskPool) Close() {
```

```
    close(p.tasks)
    p.wg.Wait()
    close(p.results)
}

// 使用範例
func main() {
    pool := NewTaskPool(3)
    pool.Start()

    // 提交任務
    for i := 0; i < 10; i++ {
        task := &Task{ID: i, Data: i}
        pool.Submit(task)
    }

    // 收集結果
    go func() {
        for result := range pool.results {
            fmt.Printf("Task %d completed: %v\n", result.ID, result.Result)
        }
    }()

    pool.Close()
}
```

3. 綜合範例：併發爬蟲

```
// 完整的併發爬蟲實現
type Crawler struct {
    visited     map[string]bool
    mu          sync.RWMutex
    sem         chan struct{}
    maxDepth    int
    ctx         context.Context
    cancel      context.CancelFunc
}

func NewCrawler(maxWorkers, maxDepth int) *Crawler {
    ctx, cancel := context.WithCancel(context.Background())
    return &Crawler{
        visited:  make(map[string]bool),
        sem:      make(chan struct{}, maxWorkers),
        maxDepth: maxDepth,
        ctx:      ctx,
        cancel:   cancel,
    }
}

func (c *Crawler) Crawl(url string, depth int) {
    if depth > c.maxDepth {
```

```
        return
    }

    c.mu.RLock()
    if c.visited[url] {
        c.mu.RUnlock()
        return
    }
    c.mu.RUnlock()

    select {
    case c.sem <- struct{}{}:  // 獲取信號量
    case <-c.ctx.Done():
        return
    }
    defer func() { <-c.sem }()  // 釋放信號量

    // 下載頁面
    resp, err := http.Get(url)
    if err != nil {
        return
    }
    defer resp.Body.Close()

    c.mu.Lock()
    c.visited[url] = true
    c.mu.Unlock()

    // 解析連結
    doc, err := goquery.NewDocumentFromReader(resp.Body)
    if err != nil {
        return
    }

    // 處理找到的連結
    doc.Find("a").Each(func(_ int, s *goquery.Selection) {
        if href, exists := s.Attr("href"); exists {
            if absURL, err := url.Parse(href); err == nil {
                go c.Crawl(absURL.String(), depth+1)
            }
        }
    })
}

func (c *Crawler) Stop() {
    c.cancel()
}

// 使用範例
func main() {
    crawler := NewCrawler(5, 3)  // 最多5個並發，深度3

    go crawler.Crawl("http://example.com", 0)
```

```
    // 等待一段時間後停止
    time.Sleep(time.Second * 30)
    crawler.Stop()
}
```

4. 記憶體模型問題解答

```
// 記憶體一致性問題的解決方案
type SafeCounter struct {
    mu    sync.Mutex
    count int
}

func (c *SafeCounter) Increment() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.count++
}

func (c *SafeCounter) GetCount() int {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.count
}

// 使用 atomic 操作
type AtomicCounter struct {
    count atomic.Int64
}

func (c *AtomicCounter) Increment() {
    c.count.Add(1)
}

func (c *AtomicCounter) GetCount() int64 {
    return c.count.Load()
}

// 使用 channel 同步
type ChannelCounter struct {
    count int
    ch    chan struct{}
}

func NewChannelCounter() *ChannelCounter {
    c := &ChannelCounter{
        ch: make(chan struct{}, 1),
    }
    c.ch <- struct{}{}  // 初始化通道
    return c
}
```

```
func (c *ChannelCounter) Increment() {
    <-c.ch  // 獲取鎖
    c.count++
    c.ch <- struct{}{}  // 釋放鎖
}

func (c *ChannelCounter) GetCount() int {
    <-c.ch  // 獲取鎖
    count := c.count
    c.ch <- struct{}{}  // 釋放鎖
    return count
}
```

四、複習與建議

1. **動手實作八宮格或任務排程**：撰寫完整 A* 搜尋程式，並嘗試不同啟發式。實際運行後觀察節點展開差異，增進理解。
2. **閱讀論文與章節**：將《Concepts and Notations for Concurrent Programming》第一章及 3.1~3.2 詳讀並整理重點，尤其是 semaphore 範例與忙等缺點。
3. **撰寫小型併發程式**：如聊天室、檔案伺服器或計算密集工作，以熟悉 goroutine、channel、mutex 等工具。確保所有 goroutine 都會結束，以避免洩漏。
4. **自我測驗**：列出關鍵名詞（如 critical section、race condition、RWMutex 等），試著用自己的話解釋並寫下程式範例。
5. **維持規律**：每天安排固定時間複習並實作，搭配休息與自我檢驗。透過口頭講解或寫作，可更牢記概念。

結語

本筆記以超過兩千字的篇幅涵蓋了期末考指定內容：包括 Prolog 的狀態空間與 A* 搜尋、論文中的同步機制、Go 語言的併發模式，以及多種範例與實作要點。建議讀者在考前反覆練習程式和演算法推導，並確保掌握每個概念背後的理由與使用情境。熟悉這些主題後，相信能在期末考中順利發揮。

五、補充：Monitor 與訊息系統

除了 semaphore，論文也概述了 Conditional Critical Regions、Monitors 與 Path Expressions 等高階同步方法。Monitor 結合了資料與操作，只有透過 monitor 的程式碼才能存取內部資源，互斥與條件等待都由語言或執行環境隱含處理。Go 的 `sync.Cond` 與 `sync.Mutex` 可以模擬 monitor 風格，雖然語法上沒有顯式的 monitor 關鍵字，但透過封裝與方法呼叫即可達到同樣目的。

在訊息傳遞方面，論文介紹了同步、非同步通道，以及遠端程序呼叫(RPC)與交易 (Atomic Transaction) 觀念。Go 的 `net/rpc` 套件提供 RPC 機制，而更常用的 `net/http` 搭配 goroutine 也能實作分散式服務。若再加入 context 或自訂超時機制，便能處理複雜的網路錯誤情境。

六、進一步閱讀建議

1. **深入理解 A***：可參考 AI 教科書，如 Russell & Norvig 的《Artificial Intelligence: A Modern Approach》，其中對 A* 的分析更為詳盡，並討論一致性、一致啟發式等進階概念。

2. **CSP 與 Go**：欲瞭解 Go 通道背後的理論基礎，可研讀 Hoare 的《Communicating Sequential Processes》。雖然語法與 Go 不同，但核心思想相通。
3. **併發程式設計模式**：Go 社群有大量實作範例，如 worker pool、fan-in fan-out、pipeline 等。熟悉這些模式有助於實戰應用。

七、可能考題與解答

一、Prolog 與啟發式搜尋相關考題

1. 八宮格問題與 A* 演算法

考題敘述：請實作一個 Prolog 程式來解決八宮格問題，使用 A* 演算法。程式需包含狀態表示、移動規則、啟發式函數，以及完整的 A* 搜尋實現。

解答：

```
% 狀態表示：使用列表表示 3x3 棋盤，0 表示空格
% 例如：[2,8,3,1,6,4,7,0,5] 表示：
% 2 8 3
% 1 6 4
% 7 0 5

% 定義狀態轉換
s(State1, State2, 1) :-
    move(State1, State2).

% 定義移動規則
move(State1, State2) :-
    append(Left, [0|Right], State1),
    append(Left, [X|Right], State2),
    member(X, [1,2,3,4,5,6,7,8]).

% 定義目標狀態
goal([1,2,3,4,5,6,7,8,0]).

% 曼哈頓距離啟發式
h(State, H) :-
    manhattan_distance(State, H).

manhattan_distance(State, H) :-
    findall(D, (nth1(Pos, State, X), X \= 0,
                goal_position(X, GoalPos),
                manhattan_dist(Pos, GoalPos, D)), Distances),
    sum_list(Distances, H).

% 計算兩個位置間的曼哈頓距離
manhattan_dist(Pos1, Pos2, D) :-
    X1 is (Pos1-1) mod 3,
    Y1 is (Pos1-1) // 3,
    X2 is (Pos2-1) mod 3,
    Y2 is (Pos2-1) // 3,
    D is abs(X1-X2) + abs(Y1-Y2).
```

```
% A* 演算法實現
solve(Start, Path, Cost) :-
    astar([[Start,0]], [], Path, Cost).

astar([[State,G]|_], _, [State], G) :-
    goal(State).

astar([[State,G]|RestOpen], Closed, Path, Cost) :-
    findall([Next,G1],
        (s(State,Next,C),
         \+ member(Next,Closed),
         G1 is G+C),
        Children),
    append(RestOpen, Children, Open1),
    sort(2, @=<, Open1, OpenSorted),
    astar(OpenSorted, [State|Closed], PathRest, Cost),
    Path = [State|PathRest].
```

2. 啟發式函數設計

考題敘述：請解釋為什麼曼哈頓距離是一個可容許的啟發式函式，並比較不同啟發式函式（如曼哈頓距離、錯位方塊數）的優缺點。

解答： 曼哈頓距離的可容許性證明：

1. 在八宮格問題中，每個方塊每次只能移動一格（水平或垂直）
2. 因此，從當前位置到目標位置的最短距離至少等於曼哈頓距離
3. 這意味著曼哈頓距離永遠不會高估實際成本，滿足可容許性條件

不同啟發式函式比較：

1. 曼哈頓距離：
    - 優點：可容許、計算簡單、考慮了方塊的實際移動距離
    - 缺點：可能低估實際成本，因為沒有考慮方塊移動時的相互影響
2. 錯位方塊數：
    - 優點：計算極其簡單、容易理解
    - 缺點：可能嚴重低估實際成本，因為沒有考慮方塊需要移動的距離
3. 結合啟發式：
    - 優點：可以結合多個啟發式的優點，提供更準確的估計
    - 缺點：計算複雜度增加，需要確保可容許性

3. IDA* 與 A* 比較

考題敘述：比較 A* 和 IDA* 的優缺點，並說明在什麼情況下 IDA* 會比 A* 更有效率。

解答： A* 與 IDA* 比較：

1. A* 優點：
    - 保證找到最優解
    - 展開節點數通常較少
    - 適合有足夠記憶體的環境
2. A* 缺點：
    - 需要儲存所有展開的節點
    - 記憶體使用量可能很大
    - 在搜尋空間大時可能導致記憶體不足
3. IDA* 優點：
    - 記憶體使用量固定且較小
    - 適合記憶體受限的環境
    - 實現相對簡單
4. IDA* 缺點：
    - 可能重複展開相同的節點
    - 在搜尋空間大時可能較慢
    - 不適合有大量相同 f 值的情況

IDA* 更有效率的情況：

1. 記憶體受限的環境
2. 搜尋空間較小
3. 啟發式函數能有效剪枝
4. 目標節點在較淺層

二、併發程式理論相關考題

1. 生產者-消費者模式

考題敘述：實作一個使用 channel 的生產者-消費者模式，需考慮同步、錯誤處理和資源管理。

解答：

```
// 完整的生產者-消費者實現
type Producer struct {
    ch    chan<- int
    done  chan<- bool
}

func (p *Producer) Run() {
    for i := 0; i < 10; i++ {
        p.ch <- i
        time.Sleep(time.Millisecond * 100)
    }
    p.done <- true
}
```

```
type Consumer struct {
    ch    <-chan int
    done  <-chan bool
    wg    *sync.WaitGroup
}

func (c *Consumer) Run() {
    defer c.wg.Done()
    for {
        select {
        case x := <-c.ch:
            fmt.Printf("Consumed: %d\n", x)
        case <-c.done:
            return
        }
    }
}

func main() {
    ch := make(chan int, 5)  // 緩衝通道
    done := make(chan bool)
    var wg sync.WaitGroup

    producer := &Producer{ch: ch, done: done}
    consumer := &Consumer{ch: ch, done: done, wg: &wg}

    wg.Add(1)
    go producer.Run()
    go consumer.Run()

    wg.Wait()
}
```

2. 讀寫鎖實現

考題敘述：使用 semaphore 實作一個讀寫鎖，並解釋 semaphore 和 mutex 的區別。

解答：

```
// 使用 semaphore 實現讀寫鎖
type RWLock struct {
    readers    int
    mu         sync.Mutex
    writer     chan struct{}
}

func NewRWLock() *RWLock {
    rw := &RWLock{
        writer: make(chan struct{}, 1),
    }
```

```
    rw.readerCond = sync.NewCond(&rw.mu)
    return rw
}

func (l *RWLock) RLock() {
    l.mu.Lock()
    defer l.mu.Unlock()

    for len(l.writer) > 0 {
        l.readerCond.Wait()
    }
    l.readers++
}

func (l *RWLock) RUnlock() {
    l.mu.Lock()
    defer l.mu.Unlock()

    l.readers--
    if l.readers == 0 {
        l.readerCond.Broadcast()
    }
}

func (l *RWLock) Lock() {
    l.writer <- struct{}{}
    l.mu.Lock()
    for l.readers > 0 {
        l.readerCond.Wait()
    }
}

func (l *RWLock) Unlock() {
    l.mu.Unlock()
    <-l.writer
    l.readerCond.Broadcast()
}
```

Semaphore 和 Mutex 的區別：

1. 功能範圍：
    - Mutex：二元信號量，只有鎖定和解鎖兩種狀態
    - Semaphore：可以有多個計數值，允許多個進程同時訪問
2. 使用場景：
    - Mutex：適合互斥訪問共享資源
    - Semaphore：適合控制並發數量、實現生產者-消費者模式等
3. 實現方式：
    - Mutex：通常由作業系統提供，效率較高

- Semaphore：可以基於 Mutex 實現，功能更靈活

3. 死鎖檢測與預防

考題敘述：分析並修復死鎖問題，實作一個死鎖檢測機制。

解答：

```
// 死鎖檢測機制
type Resource struct {
    id     int
    locked bool
    owner  int
}

type Process struct {
    id        int
    resources map[int]*Resource
    mu        sync.Mutex
}

type DeadlockDetector struct {
    processes  map[int]*Process
    resources  map[int]*Resource
    mu         sync.RWMutex
}

func NewDeadlockDetector() *DeadlockDetector {
    return &DeadlockDetector{
        processes: make(map[int]*Process),
        resources: make(map[int]*Resource),
    }
}

func (d *DeadlockDetector) DetectDeadlock() bool {
    d.mu.RLock()
    defer d.mu.RUnlock()

    // 建立資源分配圖
    graph := make(map[int][]int)
    for _, p := range d.processes {
        p.mu.Lock()
        for _, r := range p.resources {
            if r.locked {
                graph[p.id] = append(graph[p.id], r.owner)
            }
        }
        p.mu.Unlock()
    }

    // 使用 DFS 檢測循環
    visited := make(map[int]bool)
```

```
    recStack := make(map[int]bool)

    var dfs func(int) bool
    dfs = func(node int) bool {
        visited[node] = true
        recStack[node] = true

        for _, neighbor := range graph[node] {
            if !visited[neighbor] {
                if dfs(neighbor) {
                    return true
                }
            } else if recStack[neighbor] {
                return true
            }
        }

        recStack[node] = false
        return false
    }

    for node := range graph {
        if !visited[node] {
            if dfs(node) {
                return true
            }
        }
    }

    return false
}

// 避免死鎖的資源分配策略
func (d *DeadlockDetector) AllocateResource(processID, resourceID int) bool {
    d.mu.Lock()
    defer d.mu.Unlock()

    // 檢查資源是否可用
    resource, exists := d.resources[resourceID]
    if !exists || resource.locked {
        return false
    }

    // 模擬分配
    resource.locked = true
    resource.owner = processID

    // 檢查是否會導致死鎖
    if d.DetectDeadlock() {
        // 如果會導致死鎖，回滾分配
        resource.locked = false
        return false
    }
```

```
    return true
}
```

三、Go 語言併發特性相關考題

1. Goroutine 與 Channel

考題敘述：解釋 goroutine 和作業系統執行緒的區別，並實作一個使用 goroutine 的並行處理系統。

解答： Goroutine 與作業系統執行緒的區別：

1. 記憶體使用：
    - Goroutine：初始棧大小約 2KB，可動態增長
    - 執行緒：初始棧大小約 1MB，固定大小
2. 創建成本：
    - Goroutine：創建和銷毀成本低
    - 執行緒：創建和銷毀成本高
3. 調度方式：
    - Goroutine：由 Go 運行時調度，M:N 模型
    - 執行緒：由作業系統調度，1:1 模型
4. 並發數量：
    - Goroutine：可輕鬆創建數千個
    - 執行緒：受系統資源限制

並行處理系統實作：

```
// 並行處理系統
type Task struct {
    ID     int
    Data   interface{}
    Result interface{}
    Error  error
}

type Worker struct {
    id       int
    tasks    <-chan *Task
    results  chan<- *Task
    done     <-chan struct{}
}

func (w *Worker) Run() {
    for {
        select {
        case task := <-w.tasks:
```

```
            // 處理任務
            result, err := processTask(task)
            task.Result = result
            task.Error = err
            w.results <- task
        case <-w.done:
            return
        }
    }
}

type ParallelProcessor struct {
    workers  int
    tasks    chan *Task
    results  chan *Task
    done     chan struct{}
    wg       sync.WaitGroup
}

func NewParallelProcessor(workers int) *ParallelProcessor {
    return &ParallelProcessor{
        workers: workers,
        tasks:   make(chan *Task),
        results: make(chan *Task),
        done:    make(chan struct{}),
    }
}

func (p *ParallelProcessor) Start() {
    for i := 0; i < p.workers; i++ {
        p.wg.Add(1)
        go p.worker(i)
    }
}

func (p *ParallelProcessor) worker(id int) {
    defer p.wg.Done()
    worker := &Worker{
        id:      id,
        tasks:   p.tasks,
        results: p.results,
        done:    p.done,
    }
    worker.Run()
}

func (p *ParallelProcessor) Submit(task *Task) {
    p.tasks <- task
}

func (p *ParallelProcessor) Stop() {
    close(p.done)
    p.wg.Wait()
    close(p.tasks)
```

```
    close(p.results)
}
```

2. 記憶體模型與同步

考題敘述：解釋 Go 的記憶體模型，並設計一個保證記憶體一致性的並行系統。

解答： Go 記憶體模型要點：

1. 基本原則：
    - 單個 goroutine 中的操作按順序執行
    - 不同 goroutine 間的同步通過 channel 或同步原語實現
    - 編譯器和處理器可能重排指令，但不會影響單個 goroutine 的行為
2. 同步機制：
    - channel 操作：發送和接收操作是同步點
    - sync 包：Mutex、RWMutex、WaitGroup 等提供同步保證
    - atomic 包：提供原子操作

記憶體一致性系統實作：

```go
// 保證記憶體一致性的並行系統
type ConsistentSystem struct {
    mu      sync.RWMutex
    data    map[string]interface{}
    version int64
}

func NewConsistentSystem() *ConsistentSystem {
    return &ConsistentSystem{
        data: make(map[string]interface{}),
    }
}

// 使用 RWMutex 保證讀寫一致性
func (s *ConsistentSystem) Get(key string) (interface{}, int64) {
    s.mu.RLock()
    defer s.mu.RUnlock()
    return s.data[key], s.version
}

func (s *ConsistentSystem) Set(key string, value interface{}) {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.data[key] = value
    s.version++
}

// 使用 atomic 操作保證版本號一致性
```

```go
func (s *ConsistentSystem) GetVersion() int64 {
    return atomic.LoadInt64(&s.version)
}

// 使用 channel 實現事務
type Transaction struct {
    ops     []Operation
    result  chan error
}

type Operation struct {
    Type  string
    Key   string
    Value interface{}
}

func (s *ConsistentSystem) ExecuteTransaction(tx *Transaction) error {
    s.mu.Lock()
    defer s.mu.Unlock()

    // 執行所有操作
    for _, op := range tx.ops {
        switch op.Type {
        case "SET":
            s.data[op.Key] = op.Value
        case "DELETE":
            delete(s.data, op.Key)
        }
    }

    s.version++
    return nil
}
```

3. Context 與取消機制

考題敘述：實作一個可取消的並行處理系統，使用 context 控制生命週期。

解答：

```go
// 可取消的並行處理系統
type CancellableProcessor struct {
    ctx     context.Context
    cancel  context.CancelFunc
    tasks   chan *Task
    results chan *Task
    wg      sync.WaitGroup
}

func NewCancellableProcessor() *CancellableProcessor {
    ctx, cancel := context.WithCancel(context.Background())
```

```
    return &CancellableProcessor{
        ctx:     ctx,
        cancel:  cancel,
        tasks:   make(chan *Task),
        results: make(chan *Task),
    }
}

func (p *CancellableProcessor) Start(workers int) {
    for i := 0; i < workers; i++ {
        p.wg.Add(1)
        go p.worker(i)
    }
}

func (p *CancellableProcessor) worker(id int) {
    defer p.wg.Done()

    for {
        select {
        case task := <-p.tasks:
            // 處理任務
            result, err := processTaskWithContext(p.ctx, task)
            task.Result = result
            task.Error = err
            p.results <- task
        case <-p.ctx.Done():
            return
        }
    }
}

func (p *CancellableProcessor) Submit(task *Task) error {
    select {
    case p.tasks <- task:
        return nil
    case <-p.ctx.Done():
        return p.ctx.Err()
    }
}

func (p *CancellableProcessor) Stop() {
    p.cancel()
    p.wg.Wait()
    close(p.tasks)
    close(p.results)
}

// 帶超時控制的任務處理
func processTaskWithContext(ctx context.Context, task *Task) (interface{}, error)
{
    // 創建帶超時的上下文
    ctx, cancel := context.WithTimeout(ctx, time.Second*5)
    defer cancel()
```

```
    // 使用 select 實現超時控制
    done := make(chan struct{})
    var result interface{}
    var err error

    go func() {
        result, err = processTask(task)
        close(done)
    }()

    select {
    case <-done:
        return result, err
    case <-ctx.Done():
        return nil, ctx.Err()
    }
}
```

八、基礎與中等難度考題

一、Prolog 與啟發式搜尋基礎考題

1. 狀態空間搜尋基本概念

考題敘述：請解釋什麼是狀態空間搜尋，並說明在 Prolog 中如何表示狀態和狀態轉換。

解答： 狀態空間搜尋的基本概念：

1. 狀態空間：
    - 所有可能狀態的集合
    - 每個狀態代表問題在某一時刻的完整描述
    - 狀態之間的轉換代表可能的操作
2. Prolog 中的狀態表示：
    - 使用事實或規則描述狀態
    - 常見的表示方式包括列表、複合項等
    - 例如：`state([1,2,3,4,5,6,7,8,0])` 表示八宮格的一個狀態
3. 狀態轉換：
    - 使用規則描述狀態之間的轉換關係
    - 通常使用 `s(State1, State2, Cost)` 的形式
    - 轉換規則需要考慮問題的約束條件

2. 啟發式搜尋基本概念

考題敘述：請解釋什麼是啟發式函數，並說明為什麼啟發式函數在搜尋過程中很重要。

解答： 啟發式函數的基本概念：

1. 定義：
    - 啟發式函數是對從當前狀態到目標狀態的距離估計
    - 通常記作 h(n)，其中 n 是當前狀態
    - 啟發式函數的值越小，表示越接近目標
2. 重要性：
    - 幫助搜尋演算法選擇最有希望的路徑
    - 減少需要探索的狀態數量
    - 提高搜尋效率
3. 啟發式函數的特性：
    - 可容許性：不會高估實際成本
    - 一致性：滿足三角不等式
    - 單調性：隨著搜尋深度增加，估計值不會減少

3. 搜尋演算法比較

考題敘述：比較深度優先搜尋(DFS)和廣度優先搜尋(BFS)的優缺點，並說明它們各自適合什麼樣的問題。

解答： DFS 和 BFS 的比較：

1. 深度優先搜尋(DFS)：
    - 優點：
        - 記憶體使用量較少
        - 適合搜尋深度較大的問題
        - 可以快速找到一個解
    - 缺點：
        - 不一定找到最優解
        - 可能陷入很深的無效路徑
    - 適用場景：
        - 解空間較大
        - 只需要找到一個解
        - 記憶體受限
2. 廣度優先搜尋(BFS)：
    - 優點：
        - 保證找到最優解
        - 不會陷入很深的無效路徑
    - 缺點：
        - 記憶體使用量較大
        - 在解空間較大時效率較低
    - 適用場景：
        - 需要找到最優解
        - 解空間較小
        - 有足夠的記憶體

二、併發程式基礎考題

1. 併發與並行

考題敘述：請解釋併發(Concurrency)和並行(Parallelism)的區別，並舉例說明。

解答： 併發與並行的區別：

1. 併發(Concurrency)：
    - 定義：多個任務交替執行
    - 特點：
        - 任務可以同時存在
        - 不一定同時執行
        - 適合 I/O 密集型任務
    - 例子：
        - 網頁伺服器處理多個請求
        - 使用者介面響應多個事件
2. 並行(Parallelism)：
    - 定義：多個任務同時執行
    - 特點：
        - 需要多個處理器
        - 真正同時執行
        - 適合計算密集型任務
    - 例子：
        - 多核心處理器同時執行多個執行緒
        - 圖形處理器並行計算

2. 同步機制

考題敘述：請解釋什麼是同步機制，並說明為什麼在併發程式中需要同步。

解答： 同步機制的基本概念：

1. 同步的定義：
    - 控制多個執行緒的執行順序
    - 確保共享資源的正確訪問
    - 避免競爭條件
2. 為什麼需要同步：
    - 避免資料競爭
    - 確保資料一致性
    - 防止死鎖和活鎖
    - 實現執行緒間的通訊
3. 常見的同步問題：

- 競爭條件：多個執行緒同時修改共享資料
- 死鎖：多個執行緒互相等待
- 活鎖：執行緒不斷重試但無法進展
- 飢餓：某些執行緒永遠無法獲得資源

3. 通訊機制

考題敘述：請解釋共享記憶體和訊息傳遞兩種通訊機制的區別，並說明它們各自的優缺點。

解答： 通訊機制的比較：

1. 共享記憶體：
   - 定義：多個執行緒共享同一塊記憶體區域
   - 優點：
     - 通訊效率高
     - 實現簡單
     - 適合緊耦合系統
   - 缺點：
     - 需要同步機制
     - 容易出現競爭條件
     - 除錯困難
2. 訊息傳遞：
   - 定義：執行緒通過發送和接收訊息來通訊
   - 優點：
     - 鬆耦合
     - 更容易擴展
     - 適合分散式系統
   - 缺點：
     - 通訊開銷較大
     - 需要額外的訊息處理
     - 可能出現訊息丟失

三、Go 語言基礎考題

1. Goroutine 基本概念

考題敘述：請解釋什麼是 goroutine，並說明它與傳統執行緒的區別。

解答： Goroutine 的基本概念：

1. 定義：
   - Go 語言的輕量級執行緒
   - 由 Go 運行時管理
   - 使用 `go` 關鍵字創建
2. 與傳統執行緒的區別：

- 記憶體使用：
  - Goroutine：初始棧大小約 2KB
  - 執行緒：初始棧大小約 1MB
- 創建成本：
  - Goroutine：創建和銷毀成本低
  - 執行緒：創建和銷毀成本高
- 調度方式：
  - Goroutine：由 Go 運行時調度
  - 執行緒：由作業系統調度

3. 使用場景：

- 並行處理
- 非阻塞 I/O
- 事件處理
- 並行計算

2. Channel 基本概念

考題敘述：請解釋什麼是 channel，並說明它在 Go 語言中的作用。

解答： Channel 的基本概念：

1. 定義：

- Go 語言中的通訊原語
- 用於 goroutine 之間的通訊
- 使用 `make(chan T)` 創建

2. 主要作用：

- 同步：確保 goroutine 按正確順序執行
- 通訊：在 goroutine 之間傳遞資料
- 控制：實現 goroutine 的生命週期管理

3. 使用方式：

- 發送：`ch <- value`
- 接收：`value := <-ch`
- 關閉：`close(ch)`
- 遍歷：`for v := range ch`

3. 並行控制

考題敘述：請解釋 Go 語言中的 `sync` 包提供了哪些並行控制機制，並說明它們的用途。

解答： Go 語言的並行控制機制：

1. Mutex（互斥鎖）：

- 用途：保護共享資源

- 方法：`Lock()` 和 `Unlock()`
- 適用場景：需要互斥訪問的共享資源

2. RWMutex（讀寫鎖）：

- 用途：允許多個讀者或一個寫者
- 方法：`RLock()`、`RUnlock()`、`Lock()`、`Unlock()`
- 適用場景：讀多寫少的共享資源

3. WaitGroup：

- 用途：等待一組 goroutine 完成
- 方法：`Add()`、`Done()`、`Wait()`
- 適用場景：需要等待多個並行任務完成

4. Once：

- 用途：確保某個函數只執行一次
- 方法：`Do()`
- 適用場景：初始化、單例模式

5. Cond：

- 用途：條件變數，用於 goroutine 間的等待和通知
- 方法：`Wait()`、`Signal()`、`Broadcast()`
- 適用場景：複雜的同步需求