

Birdsoft 公用模組

設計規格文件

|  |  |  |
| --- | --- | --- |
| 撰寫人  Brian Chuang |  |  |

|  |
| --- |
| 目錄  [壹、 LogManager 3](#_Toc180862222)  [一、 概述 3](#_Toc180862223)  [二、 行為驅動需求(BDD) 3](#_Toc180862224)  [三、 系統設計 10](#_Toc180862225)  [四、 單元測試案例 20](#_Toc180862226)  [五、 API介面與系統整合 23](#_Toc180862227)  [六、 版本控制與佈署 29](#_Toc180862228)  [貳、 TaskScheduleManager 35](#_Toc180862229) |

# LogManager

## 概述

LogManager 是一個全域使用的日誌管理系統，支援背景任務日誌的寫入、查詢及顯示。它具有靜態實作，支援日誌事件通知，並允許外部設定保留天數及手動清理日誌。

## 行為驅動需求(BDD)

* + 1. 功能需求行為描述
       1. 記錄日誌
          1. 說明：記錄日誌的行為是 LogManager 的核心功能，支援多種日誌層級（如 Information、Warning、Error），並且日誌寫入後觸發 UI 更新事件。
          2. Gherkin

Scenario： 記錄 Information 級別的日誌

Given LogManager 已初始化

When 我寫入一條 Information 級別的日誌訊息 "系統初始化成功"

Then 日誌應被正確寫入檔案

And 應該觸發 LogWritten 事件通知 UI 更新

Scenario： 記錄 Warning 級別的日誌

Given LogManager 已初始化

When 我寫入一條 Warning 級別的日誌訊息 "警告測試"

Then 日誌應被正確寫入檔案

And 應該觸發 LogWritten 事件通知 UI 更新

Scenario： 記錄 Error 級別的日誌

Given LogManager 已初始化

When 我寫入一條 Error 級別的日誌訊息 "系統發生錯誤"

Then 日誌應被正確寫入檔案

And 應該觸發 LogWritten 事件通知 UI 更新

Scenario： 記錄日誌時發生異常觸發事件通知。

Given LogManager 已初始化

When 我寫入一條 Information 級別的日誌訊息 "記錄失敗測試"。

And 寫入過程中發生 I/O 異常

Then 系統應捕捉異常

And 應觸發一個事件通知訂閱者錯誤發生

Scenario： 記錄日誌時發生異常，但使用者未訂閱事件時拋出錯誤。

Given LogManager 已初始化

When 我寫入一條 Information 級別的日誌訊息 “記錄失敗測試”

And 寫入過程中發生 I/O 異常

Then 系統應捕捉異常

And 拋出一個例外處理

* + - 1. 日誌路徑變更設定
         1. Scenario： 設定日誌儲存路徑在程式相對路徑裡的”TestLog”。

Given LogManager 已初始化

When 我將日誌儲存路徑改為相對路徑”TestLog”

Then 系統應在指定的”C:\TestLog”路徑下產生日誌檔案

* + - * 1. Scenario： 設定日誌儲存路徑在絕對路徑”C:\TestLog”。

Given LogManager 已初始化

When 我將日誌儲存路徑改為絕對路徑”C:\TestLog”

Then 系統應在”C:\TestLog”路徑下產生新的日誌檔

* + - * 1. Scenario： 設定日誌儲存路徑在絕對路徑”C:\”。

Given LogManager 已初始化

When 我將日誌儲存路徑改為絕對路徑”C:\”

Then 系統應該不允許在根目錄下產生日誌檔

And 系統應回歸到相對路徑”Log”(預設值)下產生日誌檔

* + - 1. 日誌保留天數設定
         1. 系統允許設定日誌保留的天數，並且根據設定的天數，自動清理超過保留天數的日誌紀錄。
         2. Gherkin

Scenario： 設定日誌保留天數為 7 天

Given LogManager 已初始化

When 我將日誌保留天數設定為 7 天

Then 系統應保存 7 天內的日誌

And 清除超過 7 天的所有日誌紀錄

* + - 1. 手動日誌清理
         1. 除了自動清理功能外，系統應支援手動觸發日誌清理操作，清理所有超過保留天數的日誌。
         2. Gherkin

Scenario： 手動觸發日誌清理

Given 系統有超過保留天數的日誌

And LogManager 已初始化

When 我手動觸發日誌清理操作

Then 所有超過保留天數的日誌應被移除

And 系統應記錄一條 Information 級別的日誌，說明清理成功

* + - 1. 日誌通知事件
         1. 每當有新日誌寫入時，應觸發 LogWritten 事件，允許 UI 層訂閱該事件並做後續處理。
         2. Gherkin

Scenario： 訂閱 LogWritten 事件

Given LogManager 已初始化

And UI 已訂閱 LogWritten 事件

When 我寫入一條 Information 級別的日誌訊息 "更新狀態"

Then LogWritten 事件應被觸發

And 收到一條 Information 級別的日誌訊息 "更新狀態"

Scenario： 多次寫入日誌並觸發事件

Given LogManager 已初始化

And UI 已訂閱 LogWritten 事件

When 我連續寫入三條日誌訊息

Then LogWritten 事件應被觸發三次

And 收到三條 Information 級別的日誌訊息 "更新狀態"

* + - 1. 非同步日誌寫入
         1. Scenario： 非同步寫入 Information 級別日誌

Given LogManager 已初始化

And 已訂閱 LogWritten 事件

When 我以非同步方式寫入一條 Information級別的日誌訊息

Then 系統應成功寫入日誌而不阻塞主線程

And 觸發 LogWritten 事件通知更新

* + - * 1. Scenario： 同時進行多個非同步日誌寫入

Given LogManager 已初始化

When 我同時觸發多個非同步日誌寫入操作

Then 系統應正確寫入所有日誌

And 觸發 LogWritten 事件對應次數

* + - 1. 日誌儲存路徑規則置換
         1. Scenario： 更換儲存路徑生成類別

Given LogManager 已初始化

When 更換儲存路徑規則類別

And 生成路徑格式為 “Log\YYYYMMDD\_YYYYMMDD”

And 目錄名稱起迄以週為單為

And 起始日期為星期日

And 結束日期為星期六

Then 系統依日期產生新的路徑

And 在路徑下產生日誌檔案

* + - * 1. Scenario： 更換生成不合法路徑的類別

Given LogManager 已初始化

When 更換儲存路徑規則類別

And 生成路徑格式為 “Log\\C:”

Then 系統應略過錯誤路徑並使用預設路徑產生日誌檔案

And 系統應記錄一筆路徑生成錯誤的資訊到日誌中

* + - 1. 日誌名稱規則置換
         1. Scenario： 更換日誌名稱生成類別

Given LogManager 已初始化

When 更換日誌檔名生成類別

And 生成日誌名稱格式為 “TestLogYYYYMMDD”

Then 系統依新的格式產生日誌檔案

And 系統應記錄一筆日誌檔案檔名變更的資訊

* + - * 1. Scenario： 更換生成不合法檔案名稱的類別

Given LogManager 已初始化

When 更換檔案名稱規則類別

And 生成檔案名稱格式為 “Log/\:YYYYMMDD”

Then 系統應略過錯誤檔案名稱並使用預設檔案名稱產生日誌檔案

And 系統應記錄一筆檔案名稱生成錯誤的資訊到日誌中

* + 1. And 在路徑下產生日誌檔案非功能需求行為描述
       1. 多執行緒安全
          1. 系統應保證在多執行緒環境下，日誌寫入和事件觸發的行為是線程安全的。
          2. Gherkin

Scenario： 多執行緒同時寫入日誌

Given LogManager 在多執行緒環境下運行

When 100個執行緒同時寫入日誌

Then 系統應保證所有日誌正確寫入

And 不會發生資料競爭問題

* + - 1. 效能需求
         1. 系統在大規模日誌寫入和清理操作時，應保持良好的效能表現。
         2. Gherkin

Scenario： 大量日誌寫入效能測試

Given LogManager 已初始化

When 我進行大規模日誌寫入測試(例如 1,000 條日誌)

Then 系統應能夠在3秒內完成寫入

And 不會阻塞其他作業

## 系統設計

* + 1. 系統架構概述

LogManager 系統旨在提供一個靈活且可擴展的日誌管理機制，支援多種日誌處理功能，包括日誌寫入、清理、路徑和檔名生成等。系統核心由 LogManager 管理，並透過一組介面和其不同的實作來實現各種操作模式（如 default 和 custom 實作）。

* + 1. 類別圖

以下為 LogManager 系統的類別圖(使用mermaid語法)：

classDiagram

class LogManager {

- static InternalLogger \_logger

- static ILogWriter \_defaultLogWriter

- static ILogWriter \_customLogWriter

- static ILogPathGenerator \_defaultLogPathGenerator

- static ILogPathGenerator \_customLogPathGenerator

- static ILogFileNameGenerator \_defaultLogFileNameGenerator

- static ILogFileNameGenerator \_customLogFileNameGenerator

- static ILogCleaner \_defaultLogCleaner

- static ILogCleaner \_customLogCleaner

+ event LogWritten

+ event LogWriteFailed

+ event CleanupSuccess

+ event CleanupFailed

+ Initialize(ILogWriter writer, ILogPathGenerator pathGenerator, ILogFileNameGenerator fileNameGenerator, ILogCleaner logCleaner)

+ ManualCleanup()

+ WriteLogEntry(LogLevel level, string message, Exception? exception)

}

class ICustomLogger {

<<interface>>

+ string LogDirectory

+ string LogFileFormat

+ string GenerateLogFileName(DateTime date)

+ DateTime? GetDateFromFileName(string fileName)

+ void WriteLogEntry(LogLevel level, string message, Exception? exception)

+ event LogWritten

+ event LogWriteFailed

+ event CleanupSuccess

+ event CleanupFailed

}

class InternalLogger {

- string logDirectory

- string logFileFormat

+ string GenerateLogFileName(DateTime date)

+ DateTime? GetDateFromFileName(string fileName)

+ void WriteLogEntry(LogLevel level, string message, Exception? exception)

+ event LogWritten

+ event LogWriteFailed

+ event CleanupSuccess

+ event CleanupFailed

}

class ILogWriter {

<<interface>>

+ void WriteLog(LogEntry logEntry)

}

class DefaultLogWriter {

+ void WriteLog(LogEntry logEntry)

}

class CustomLogWriter {

+ void WriteLog(LogEntry logEntry)

}

class ILogCleaner {

<<interface>>

+ void Cleanup(int retentionDays)

}

class DefaultLogCleaner {

+ void Cleanup(int retentionDays)

}

class CustomLogCleaner {

+ void Cleanup(int retentionDays)

}

class ILogPathGenerator {

<<interface>>

+ string GetLogPath(DateTime date)

}

class DefaultLogPathGenerator {

+ string GetLogPath(DateTime date)

}

class CustomLogPathGenerator {

+ string GetLogPath(DateTime date)

}

class ILogFileNameGenerator {

<<interface>>

+ string GenerateLogFileName(DateTime date)

}

class DefaultLogFileNameGenerator {

+ string GenerateLogFileName(DateTime date)

}

class CustomLogFileNameGenerator {

+ string GenerateLogFileName(DateTime date)

}

class PathType {

<<enumeration>>

Relative

Absolute

}

class LogEntry {

+ LogLevel level

+ string message

+ Exception? exception

+ DateTime timestamp

}

class LogLevel {

<<enumeration>>

Information

Warning

Error

}

LogManager ..> ILogWriter : uses

LogManager ..> ILogPathGenerator : uses

LogManager ..> ILogFileNameGenerator : uses

LogManager ..> ILogCleaner : uses

LogManager \*-- InternalLogger : composition

ICustomLogger <|.. InternalLogger : implements

ILogWriter <|.. DefaultLogWriter : implements

ILogWriter <|.. CustomLogWriter : implements

ILogCleaner <|.. DefaultLogCleaner : implements

ILogCleaner <|.. CustomLogCleaner : implements

ILogPathGenerator <|.. DefaultLogPathGenerator : implements

ILogPathGenerator <|.. CustomLogPathGenerator : implements

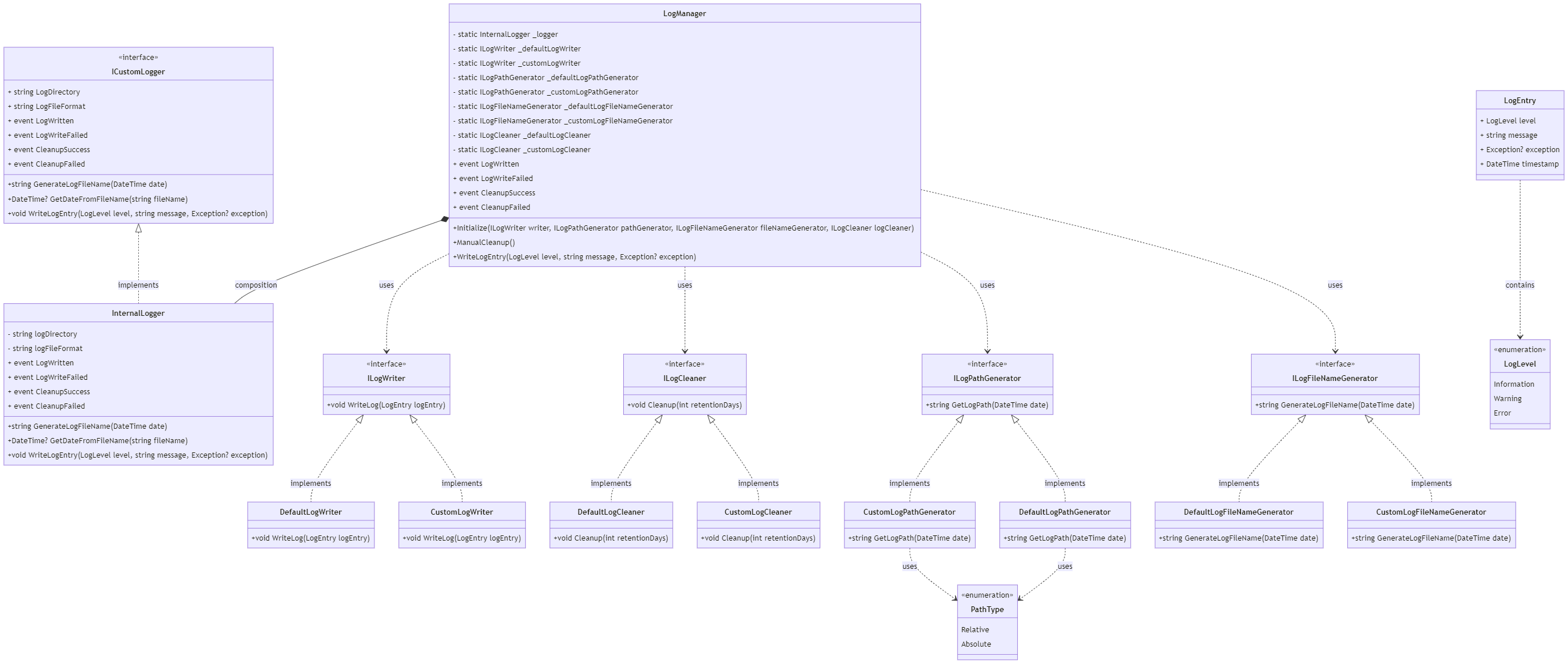
ILogFileNameGenerator <|.. DefaultLogFileNameGenerator : implements

ILogFileNameGenerator <|.. CustomLogFileNameGenerator : implements

DefaultLogPathGenerator ..> PathType : uses

CustomLogPathGenerator ..> PathType : uses

LogEntry ..> LogLevel : contains



* + 1. 類別說明
       1. LogManager

LogManager 是日誌系統的核心，負責管理日誌寫入、路徑生成、檔名生成和日誌清理。它支援多種策略，通過接口進行依賴注入，允許 default 和 custom 的實作。它提供以下功能：

* + - * 1. WriteLogEntry：記錄日誌。
        2. ManualCleanup：手動清理日誌。
        3. 事件：LogWritten、LogWriteFailed、CleanupSuccess 和 CleanupFailed。
      1. ICustomLogger & InternalLogger
         1. ICustomLogger：定義了日誌操作的介面，包括記錄日誌和生成日誌檔名的功能。設計了四個事件，以監控日誌的成功或失敗寫入，以及清理的狀態。
         2. InternalLogger：實作 ICustomLogger，並負責具體的日誌處理邏輯。
      2. ILogWriter, DefaultLogWriter, CustomLogWriter
         1. ILogWriter：接口，用於定義寫入日誌的行為。
         2. DefaultLogWriter：預設的日誌寫入實作，將日誌寫入檔案。
         3. CustomLogWriter：自訂的日誌寫入實作，允許根據特定需求定義自訂的日誌處理方式（如寫入資料庫）。
      3. ILogCleaner, DefaultLogCleaner, CustomLogCleaner
         1. ILogCleaner：接口，定義清理日誌的行為。
         2. DefaultLogCleaner：預設的日誌清理實作。
         3. CustomLogCleaner：自訂的日誌清理實作，可根據需求選擇不同的清理策略（如資料庫清理）。
      4. ILogPathGenerator, DefaultLogPathGenerator, CustomLogPathGenerator
         1. ILogPathGenerator：接口，用於生成日誌存放路徑。
         2. DefaultLogPathGenerator：根據預設規則生成日誌存放路徑。
         3. CustomLogPathGenerator：自訂路徑生成策略，允許使用者設定自己的邏輯。
      5. ILogFileNameGenerator, DefaultLogFileNameGenerator, CustomLogFileNameGenerator
         1. ILogFileNameGenerator：接口，用於生成日誌檔名。
         2. DefaultLogFileNameGenerator：使用預設格式生成日誌檔名。
         3. CustomLogFileNameGenerator：允許使用者自訂日誌檔名生成規則。
      6. LogEntry & LogLevel
         1. LogEntry：日誌條目，包括日誌級別、訊息、異常和時間戳。
         2. LogLevel：枚舉，定義日誌等級（Information、Warning、Error）。
      7. PathType

PathType：用於定義路徑類型的枚舉（相對和絕對），讓路徑生成更具彈性。

* + 1. 擴展性設計

系統通過接口和依賴注入，支援靈活的擴展。未來要新增日誌的存儲策略（如寫入雲端、API）、不同清理策略（如資料庫）、自訂路徑和檔名規則，都可以透過實作相應的接口並將其注入 LogManager 來實現。

* + 1. 整合性測試與模擬

使用 ICustomLogger、ILogWriter、ILogCleaner 等接口，方便進行模擬測試，保證日誌系統在不同情境下的穩定性。可以利用 Mock 工具模擬不同的操作，確保 LogManager 在處理各種異常和擴展場景下表現良好。

## 單元測試案例

* + 1. 測試目標

單元測試的目的是驗證 LogManager 系統中每個組件的獨立功能，確保它們能在各種情境下正確運作。測試涵蓋日誌寫入、日誌清理、路徑和檔名生成，以及異常處理和事件通知。

* + 1. 測試案例分類
       1. 日誌寫入測試 (WriteLogEntry)
          1. 測試案例1.1：成功寫入日誌

Given: LogManager 已初始化，並且 ILogWriter 正常運作。

When: 執行 WriteLogEntry([LogLevel.Information / LogLevel.Warning / LogLevel.Error], ["測試訊息1" / "測試訊息2" / "測試訊息3"], [null, new Exception(“例外測試”)])。

Then: 應觸發 LogWritten 事件，並成功寫入日誌檔案。

* + - * 1. 測試案例 1.2：寫入日誌失敗

Given: LogManager 已初始化，並且 ILogWriter 異常發生。

When: 執行 WriteLogEntry(LogLevel.Error, "錯誤訊息", new Exception("錯誤"))，導致寫入失敗。

Then: 應觸發 LogWriteFailed 事件，並記錄錯誤。

* + - * 1. 測試案例 1.3：非同步寫入日誌

Given: LogManager 已初始化，ILogWriter 支援非同步操作。

When: 非同步執行 WriteLogEntry。

Then: 日誌應成功寫入且不阻塞主線程，LogWritten 事件應被觸發。

* + - 1. 日誌清理測試 (ManualCleanup)
         1. 測試案例 2.1：成功手動清理日誌

Given: LogManager 已初始化，並設置了 ILogCleaner 實例。

When: 執行 ManualCleanup()。

Then: 應觸發 CleanupSuccess 事件，並正確移除過期日誌。

* + - * 1. 測試案例 2.2：日誌清理失敗

Given: LogManager 已初始化，ILogCleaner 運行異常。

When: 執行 ManualCleanup()，導致清理失敗。

Then: 應觸發 CleanupFailed 事件。

* + - 1. 路徑與檔名生成測試
         1. 測試案例 3.1：正確生成日誌檔案路徑

Given: DefaultLogPathGenerator 已初始化，並設置 PathType.Relative。

When: 執行 GetLogPath(DateTime.Now)。

Then: 應返回正確的相對路徑。

* + - * 1. 測試案例 3.2：自訂路徑生成

Given: CustomLogPathGenerator 已初始化，並設置 PathType.Absolute。

When: 執行 GetLogPath(DateTime.Now)。

Then: 應返回符合自訂規則的絕對路徑。

* + - * 1. 測試案例 3.3：生成日誌檔案名稱

Given: DefaultLogFileNameGenerator 已初始化。

When: 執行 GenerateLogFileName(DateTime.Now)。

Then: 應返回符合格式的檔名（例如：Log\_20241018.txt）。

* + - 1. 事件測試
         1. 測試案例 4.1：日誌成功寫入事件

Given: LogManager 已初始化，並訂閱了 LogWritten 事件。

When: 成功執行 WriteLogEntry。

Then: 應確認 LogWritten 事件被觸發。

* + - * 1. 測試案例 4.2：日誌寫入失敗事件

Given: LogManager 已初始化，並訂閱了 LogWriteFailed 事件。

When: 執行 WriteLogEntry，並故意造成寫入失敗。

Then: 應確認 LogWriteFailed 事件被觸發。

* + - * 1. 測試案例 4.3：日誌清理成功事件

Given: LogManager 已初始化，並訂閱了 CleanupSuccess 事件。

When: 執行 ManualCleanup 成功移除過期日誌。

Then: 應確認 CleanupSuccess 事件被觸發。

* + - * 1. 測試案例 4.4：日誌清理失敗事件

Given: LogManager 已初始化，並訂閱了 CleanupFailed 事件。

When: 執行 ManualCleanup，並故意造成清理失敗。

Then: 應確認 CleanupFailed 事件被觸發。

* + 1. 測試環境與工具
       1. 測試框架：xUnit 或其他支援 .NET 的測試框架。
       2. 模擬工具：Moq 或其他 mock 工具，用於模擬介面行為和測試異常情境。
       3. 日誌文件夾：測試環境應設置專用的測試日誌目錄，避免與正式日誌檔案混淆。
    2. 測試覆蓋範圍
       1. 功能測試：覆蓋所有主要功能，包括寫入、清理、路徑生成和檔名生成。
       2. 異常測試：確保在異常情況下系統能正確處理並觸發相關事件。
       3. 邊界測試：針對邊界情況進行測試，例如路徑長度限制、檔名格式等。
    3. 測試報告與分析
       1. 測試報告：每次測試運行後應生成測試報告，列出通過和失敗的案例，以及失敗原因分析。
       2. 覆蓋率分析：確保程式碼覆蓋率達到 80% 以上，特別針對關鍵邏輯進行覆蓋。

## API介面與系統整合

* + 1. 概述

LogManager 提供了一組 API，用於與其他系統進行整合，這些 API 支援日誌的寫入、路徑生成、檔名生成、日誌清理以及事件通知等功能。通過這些 API，開發人員可以輕鬆將日誌功能集成到應用程式中，並靈活地配置自訂的日誌行為。

* + 1. API 說明
       1. LogManager.Initialize
          1. 功能：初始化 LogManager，設置日誌寫入、路徑生成、檔名生成和清理策略。
          2. 方法簽名：

public static void Initialize(ILogWriter writer, ILogPathGenerator pathGenerator, ILogFileNameGenerator fileNameGenerator, ILogCleaner logCleaner)

* + - * 1. 參數：

Writer：用於寫入日誌的 IlogWriter 實例（可選用 DefaultLogWriter 或自訂實作）。

pathGenerator：用於生成日誌存放路徑的 ILogPathGenerator 實例。

fileNameGenerator：用於生成日誌檔名的 ILogFileNameGenerator 實例。

logCleaner：用於清理過期日誌的 ILogCleaner 實例。

* + - * 1. 使用範例：

var logWriter = new DefaultLogWriter();

var pathGenerator = new DefaultLogPathGenerator();

var fileNameGenerator = new DefaultLogFileNameGenerator();

var logCleaner = new DefaultLogCleaner();

LogManager.Initialize(logWriter, pathGenerator, fileNameGenerator, logCleaner);

* + - 1. LogManager.WriteLogEntry
         1. 功能：寫入一條日誌記錄。
         2. 方法簽名：

public static void WriteLogEntry(LogLevel level, string message, Exception? exception = null)

* + - * 1. 參數：

level：日誌等級，支援 Information、Warning 和 Error。

message：日誌訊息內容。

exception：可選，若有異常需記錄，可提供 Exception 對象。

* + - * 1. 使用範例：

LogManager.WriteLogEntry(LogLevel.Information, "系統啟動成功");

LogManager.WriteLogEntry(LogLevel.Error, "發生未預期的錯誤", new Exception("例外詳情"));

* + - 1. LogManager.ManualCleanup
         1. 功能：手動清理過期日誌。
         2. 方法簽名：

public static void ManualCleanup()

* + - * 1. 說明：執行日誌清理操作，根據預設的 ILogCleaner 策略移除過期的日誌。
        2. 使用範例：

LogManager.ManualCleanup();

* + - 1. LogManager 事件

LogManager 支援事件機制，允許外部系統訂閱特定的事件並執行相應的處理：

* + - * 1. LogWritten

功能：當日誌成功寫入時觸發。

事件簽名：

public static event EventHandler<LogEntry> LogWritten;

使用範例：

LogManager.LogWritten += (sender, entry) => Console.WriteLine($"日誌寫入: {entry.Message}");

* + - * 1. LogWriteFailed

功能：當日誌寫入失敗時觸發。

事件簽名：

public static event EventHandler<LogEntry> LogWriteFailed;

使用範例：

LogManager.LogWriteFailed += (sender, entry) => Console.WriteLine($"日誌寫入失敗: {entry.Message}");

* + - * 1. CleanupSuccess

功能：當日誌清理成功時觸發。

事件簽名：

public static event EventHandler CleanupSuccess;

使用範例：

LogManager.CleanupSuccess += (sender, args) => Console.WriteLine("日誌清理成功");

* + - * 1. CleanupFailed

功能：當日誌清理失敗時觸發。

事件簽名：

public static event EventHandler CleanupFailed;

使用範例：

LogManager.CleanupFailed += (sender, args) => Console.WriteLine("日誌清理失敗");

* + 1. 系統整合範例
       1. 在應用程式中使用 LogManager

public class Program

{

public static void Main(string[] args)

{

// 初始化 LogManager

LogManager.Initialize(new DefaultLogWriter(), new DefaultLogPathGenerator(), new DefaultLogFileNameGenerator(), new DefaultLogCleaner());

// 訂閱日誌事件

LogManager.LogWritten += (sender, entry) => Console.WriteLine($"[Log Written]: {entry.Message}");

LogManager.LogWriteFailed += (sender, entry) => Console.WriteLine($"[Log Failed]: {entry.Message}");

// 寫入日誌

LogManager.WriteLogEntry(LogLevel.Information, "應用程式啟動中...");

LogManager.WriteLogEntry(LogLevel.Error, "發生例外", new Exception("例外測試"));

// 手動清理日誌

LogManager.ManualCleanup();

}

}

* + 1. 擴展整合策略
       1. 自訂日誌行為

開發者可根據需求實作 ILogWriter、ILogCleaner、ILogPathGenerator 或 ILogFileNameGenerator 接口，並在 LogManager 初始化時進行設置。

* + - 1. 動態切換策略

可以在程式執行期間動態切換 custom 與 default 策略，例如切換不同的 ILogWriter 來實現日誌的多樣化寫入（如切換到寫入資料庫的自訂實作）。

* + 1. 系統整合建議
       1. 預設設定與自訂配置

將 default 設定作為系統的基礎，確保在沒有特別設定時系統能正常運

* + - 1. 錯誤處理與日誌

建議開發者在應用程式中的關鍵點都進行適當的日誌記錄，以便於問題排查和系統維護。

* + - 1. 事件監聽

對於重要的日誌操作，建議訂閱 LogManager 的事件，以即時掌握系統狀態。

## 版本控制與佈署

* + 1. 版本控制策略
       1. 版本控制工具
          1. 工具選擇

使用 Git 作為版本控制系統，並託管在GitHub版本控制平台上。

* + - * 1. 分支策略

採用 Git Flow 分支策略，確保在不同的開發階段，版本控制保持清晰有序。

* + - 1. 分支規範
         1. main 分支

永遠保持穩定且可用，包含最新的穩定版本。

僅在經過完整測試和驗證的情況下合併，適合佈署至生產環境的版本。

* + - * 1. develop 分支

用於集成開發，包含最新的功能和修復，未經最終驗證的開發版本。

開發人員的特性分支最終都會合併到 develop。

* + - * 1. feature/\* 分支

用於開發新功能，命名格式為 feature/功能名稱。

完成開發並通過測試後，合併回 develop。

* + - * 1. hotfix/\* 分支

用於修正緊急的生產問題，直接從 main 分支拉出並在修正後合併回 main 和 develop。

命名格式為 hotfix/修正描述。

* + - * 1. release/\* 分支

在發布新的穩定版本前，從 develop 拉出，用於最終的測試和驗證。

命名格式為 release/版本號，經過驗證後合併到 main 和 develop。

* + - 1. 版本命名規則

版本號格式：Major.Minor.Patch (例如 1.2.0)

Major：當有重大變更，或不向後相容的更新時提升。

Minor：當有新功能增加，且與現有功能相容時提升。

Patch：當有小修復或改善時提升。

* + 1. 佈署策略
       1. 佈署環境
          1. 開發環境 (Development Environment)

主要用於開發人員日常測試和調試。

使用 develop 分支中的最新代碼。

自動化測試應在開發環境中頻繁運行，確保代碼的即時性和穩定性。

* + - * 1. 測試環境 (Staging Environment)

模擬生產環境的測試部署。

使用 release/\* 分支中的代碼進行最終測試。

測試通過後，代碼才會合併至 main 分支並進行正式佈署。

* + - * 1. 生產環境 (Production Environment)

使用 main 分支中的穩定版本代碼。

僅佈署經過完整測試和驗證的代碼，確保生產系統的穩定運行。

* + - 1. 佈署工具與技術
         1. CI/CD 工具

使用 Jenkins、GitHub Actions 或 GitLab CI/CD 進行自動化的構建、測試和佈署。

* + - * 1. 佈署腳本

佈署腳本應包括以下步驟

構建：使用 dotnet build 編譯 .NET 專案。

測試：使用 dotnet test 執行單元測試。

佈署：自動將經過測試的應用程式佈署至指定的環境。

* + - * 1. 配置管理

使用環境變數或配置文件來管理不同環境的配置差異。

* + - 1. 佈署流程
         1. 開發階段

開發人員在 feature/\* 分支上開發新功能，經過自測後，提交到 develop 分支。

自動化構建和測試將在每次提交後觸發，以檢查代碼的品質。

* + - * 1. 測試階段

當需要發布新版本時，從 develop 分支拉出 release/\* 分支。

在測試環境中進行測試，確保功能穩定，並修復任何測試中發現的問題。

* + - * 1. 發布階段

測試完成後，將 release/\* 分支合併到 main 和 develop 分支。

自動化佈署系統會將 main 分支的最新穩定版本佈署到生產環境。

針對緊急修正的 hotfix/\* 分支，也應該直接合併到 main，並通過自動化佈署至生產環境。

* + 1. 版本記錄與發布說明
       1. 版本記錄

每個版本都應有詳細的變更記錄 (Changelog)，記錄新功能、修正和優化的詳情。

* + - 1. 發布說明
         1. 每次發佈版本應附上發布說明，內容包括新增功能、修復問題、已知問題以及升級指南。
         2. 發布說明應發布在 GitHub Release 或內部的 Wiki 頁面中，方便開發人員和用戶查閱。
    1. 版本回滾機制
       1. 在佈署過程中，應設置版本回滾機制，以應對突發情況或重大問題。當生產環境出現問題時，可以迅速回滾到之前的穩定版本。
       2. 回滾策略
          1. 即時回滾

如果生產環境出現重大問題，CI/CD 工具應支援立即回滾至上次成功部署的版本。

* + - * 1. 回滾驗證

回滾後應重新執行驗證測試，確保系統正常運作。

* + 1. 佈署安全性
       1. 權限控制

僅允許具有相應權限的開發人員和運維人員執行佈署操作。

* + - 1. 日誌監控

佈署過程應記錄詳細日誌，包括構建、測試和佈署的每一步，以便追溯和問題排查。

* + - 1. 備份策略

在每次佈署前應進行數據備份，確保能夠在出現問題時快速恢復。

# TaskScheduleManager