ChatGPT

第四章：DLL 自動載入與 Mod 類別註冊

學習目標

本章將引導您模擬 RimWorld 的模組架構，在 Unity 專案中實現以下目標：

- **模組 DLL 自動載入機制**：瞭解如何讓主程式啟動時自動探索並載入各模組的 **DLL** 檔案，讓每個模組的程式碼可以獨立封裝在自己的組件中執行。
- **動態發現與實例化 Mod 主類別**：學會使用反射掃描 DLL，尋找實作特定介面或繼承特定基底類別（如 `IMod` 或 `ModBase`）的型別，並自動建立其實例。
- **模組類別註冊與初始化**：將動態載入的模組類別註冊進模組管理系統，確保每個模組的主要邏輯類別在遊戲啟動流程中正確初始化，並能夠參與遊戲循環。
- **模組邏輯注入與事件處理**：模擬 RimWorld 的 `Mod` 行為，支援每個模組提供一個主要 Mod 類別，在遊戲啟動或特定時機自動呼叫其初始化方法（例如 `Init()` 或 `OnGameStart()`），以及讓模組程式碼能透過事件監聽或系統註冊等方式影響遊戲。
- **Def 系統整合**：學習如何讓模組 DLL 中的程式碼存取先前章節建立的 Def 資料庫，讀取或修改透過 XML 載入的遊戲資料，使模組程式碼與資料定義相結合。

技術重點

本章將著重講解下列技術要點：

- **目錄掃描與檔案載入**：使用 C# 的檔案系統功能尋找 `Mods` 資料夾中各模組的 `Assemblies` 子目錄，並透過 `System.Reflection.Assembly.LoadFrom` 動態載入 DLL [1]。這使得遊戲可以在執行時擴充行為，而不用預先將所有模組程式碼編譯進主程式。
- **反射與型別篩選**：運用反射 (`Reflection`) 技術，使用 `Assembly.GetTypes()` 列出組件內所有型別，篩選出繼承**基底類別** `ModBase` 或實作**介面** `IMod` 的類別型別。確保排除抽象類別，僅對具體實作進行後續處理。
- **動態實例化**：使用 `Activator.CreateInstance` 將篩選出的 Mod 類別型別實例化。若模組類別定義了帶參數的建構子（例如接受一個 `ModContentPack` 參數），則傳入對應的模組內容物件；否則使用無參數建構子並在事後設定所需屬性。
- **ModContentPack 管理**：引入 `ModContentPack` 類別來封裝每個模組的資訊，例如模組名稱、路徑、載入的組件及對應的 Mod 主類別實例等。這有助於主程式統一管理多個模組，使其各自的資源與邏輯彼此獨立又可被統攝。
- **模組生命周期與事件機制**：設計每個 Mod 主類別的生命周期方法，例如 `Init()`（在資料載入後或遊戲初始化時呼叫）和 `OnGameStart()`（在真正開始遊戲時呼叫），讓模組有機會在正確的時機點插入自訂邏輯。此外，透過全域事件（如自訂 `GameStarted` 事件或 Unity 的事件機制），讓模組能註冊監聽並響應遊戲過程中的各種事件（如每日更新、玩家動作等）。
- **Def 系統整合**：複習前章建立的 Def 資料結構，展示如何讓動態載入的模組程式碼訪問和操作 Def 資料庫。例如，模組可以查詢已載入的 Def（如物品或配方定義）、新增新的 Def，或修改既有 Def 的屬性值，達到修改遊戲內容的效果。

接下來，我們將按照上述重點，一步步實作 RimWorld 模組系統的核心部分：從自動載入 DLL，到發現並初始化各模組的主要類別，最後探討模組如何與資料和事件互動。

模組 DLL 的自動載入機制

在 RimWorld 中，每個模組（Mod）都有自己的資料夾，裡面包含模組描述檔、資源以及可選的 **Assemblies** 資料夾來存放模組的編譯後程式碼（DLL）。我們的 Unity 模擬專案將採用類似結構：假設在遊戲目錄下有一個名為 `Mods` 的資料夾，每個子資料夾對應一個模組。例如：

```
Mods/
├─ Core/                （可能是核心模組，包含基本資源）
├─ ModA/
│   ├─ About/          (模組描述與資訊)
│   ├─ Defs/           (XML 資料定義)
│   ├─ Assemblies/     (模組程式碼 DLL 檔案)
│   └─ ...            (其他資源，如 Textures, Sounds 等)
└─ ModB/
    ├─ About/
    ├─ Defs/
    ├─ Assemblies/
    └─ ...
```

上述結構對應 RimWorld 的模組架構，其中 **Assemblies** 資料夾存放該模組的 DLL 程式集 [1]（例如由 Visual Studio 編譯生成的 `.dll` 檔案）。主程式在啟動時需要自動掃描每個模組資料夾的 Assemblies，將其中的 DLL 檔案載入到遊戲的執行環境中。

步驟1：尋找並載入模組 DLL

我們可以利用 .NET 提供的檔案與反射 API 實現上述功能。在遊戲啟動時，編寫一個模組載入器（`ModLoader`）執行以下動作：

1. **掃描 Mods 資料夾**：取得所有模組資料夾路徑（可以透過 `Directory.GetDirectories("Mods")` 列出）。
2. **尋找 Assemblies**：對每個模組資料夾，組合出其 Assemblies 子路徑，如 `<模組路徑>/Assemblies`，並檢查該目錄是否存在 DLL 檔案。
3. **載入 DLL**：使用 `Assembly.LoadFrom(path)` 將每一個找到的 DLL 檔載入。載入成功後，可選擇將 `Assembly` 物件暫存於對應的 `ModContentPack` 中，以供後續反射使用。

以下是實作上述步驟的範例程式碼：

```
using System.IO;
using System.Reflection;
using UnityEngine;
using System.Collections.Generic;

public class ModLoader {
  public List<ModContentPack> LoadedMods = new List<ModContentPack>();

  public void LoadAllMods() {
    string modsFolder = Path.Combine(Application.dataPath, "../Mods");
    // 假設 Mods 資料夾與遊戲可執行檔在同層（編輯器中可調整路徑）
```

```
        if (!Directory.Exists(modsFolder)) {
            Debug.LogWarning("Mods 資料夾不存在：" + modsFolder);
            return;
        }

        // 掃描每個模組資料夾
        foreach (string modDir in Directory.GetDirectories(modsFolder)) {
            string asmDir = Path.Combine(modDir, "Assemblies");
            if (!Directory.Exists(asmDir)) {
                continue; // 若沒有 Assemblies 資料夾，跳過（可能此模組不含程式碼）
            }

            // 建立 ModContentPack 來記錄此模組資訊
            ModContentPack contentPack = new ModContentPack(modDir);

            // 載入該模組資料夾下所有 DLL 檔案
            foreach (string dllPath in Directory.GetFiles(asmDir, "*.dll")) {
                try {
                    Assembly asm = Assembly.LoadFrom(dllPath);
                    contentPack.Assemblies.Add(asm);
                    Debug.Log($"已載入模組程式集：{Path.GetFileName(dllPath)}");
                } catch (System.Exception e) {
                    Debug.LogError($"載入 DLL 失敗: {dllPath}\n錯誤訊息: {e.Message}");
                }
            }

            LoadedMods.Add(contentPack);
        }
    }
}
```

在上述程式碼中，我們：

- 使用 `Application.dataPath` 取得 Unity 專案的執行路徑，並假設 `Mods` 資料夾位於專案根目錄（或已知位置）。實際應用中，您可以將 Mods 資料夾路徑設為設定值，或放在持久資料夾中。
- 對每個模組建立一個 `ModContentPack` 實例（稍後定義其內容），將模組的基本資訊（例如路徑、已載入的 Assembly 清單）儲存其中。
- 使用 `Assembly.LoadFrom` 載入 DLL。這會把模組的程式碼載入當前 AppDomain。載入成功後，我們記錄日誌並把 `Assembly` 實體保存到 `contentPack.Assemblies` 清單中。
- 加入基本的錯誤處理：如果某個 DLL 載入時拋出異常，透過 `Debug.LogError` 輸出錯誤，但不影響其他模組的載入流程。如此可確保單一模組出錯時不致中斷整體載入。

  技術提醒：在 Unity 編譯為 AOT 平台（如 iOS, WebGL）時，動態載入程式集可能不可行。但在 PC（Mono 或 IL2CPP 框架）環境下，上述方法通常能正常運作。我們這裡以 PC 平台模擬為主。

步驟2：ModContentPack 類別

為了管理模組資訊，我們設計 `ModContentPack` 類別。它可以包含模組的名稱、路徑、載入的組件，以及稍後會建立的 Mod 類別實例等。例如：

```
public class ModContentPack {
  public string FolderPath { get; private set; }  // 模組資料夾完整路徑
  public string PackageId { get; private set; }  // 模組識別名稱（可用資料夾名或About中定義的ID）
  public List<Assembly> Assemblies { get; private set; }
  public ModBase ModInstance { get; internal set; } // 該模組的主要 Mod 類別實例

  public ModContentPack(string folderPath) {
    FolderPath = folderPath;
    PackageId = Path.GetFileName(folderPath);  // 簡單以資料夾名作為ID或名稱
    Assemblies = new List<Assembly>();
  }
}
```

上例中，我們將模組資料夾名稱作為 `PackageId`。在 RimWorld 中，每個模組在 About.xml 裡有定義 `<name>` 以及 `<packageId>`，我們可在進階實作時透過解析 XML 獲取真實名稱與ID。不過在本模擬中，直接使用資料夾名稱作識別即可。`Assemblies` 列表保存所有載入的程式集（通常模組可能只有一個 DLL，但我們一般化處理），`ModInstance` 將在下一步反射出類別時賦值。

完成此步驟後，我們已將所有模組的 DLL 載入記憶體中，接下來要處理的就是**尋找並初始化模組的主要類別**。

掃描 DLL 型別並註冊 Mod 類別

載入 DLL 僅僅是讓模組的程式碼進入了執行環境，下一步我們需要找到模組程式碼中那個用來表示“模組自身”的主要類別，類似於 RimWorld 中每個模組可定義一個繼承自 `Verse.Mod` 的類別。我們將透過反射來達成這個目標：

步驟3：定義 Mod 基底類別 / 介面

首先，我們需要在主程式定義一個模組類別應實作的介面或基底類別，以便識別。這裡我們設計一個抽象基底類別 `ModBase`，模組開發者可以繼承它來編寫自己的 Mod 類別（與 RimWorld 中 `Verse.Mod` 概念對應）。

```
// 位於主程式的 Assembly，供模組程式碼引用
public abstract class ModBase {
  public string ModName { get; private set; }
  public ModContentPack ContentPack { get; private set; }

  // 建構子，允許模組接收自身的 ModContentPack
  protected ModBase(ModContentPack contentPack) {
    ContentPack = contentPack;
    ModName = contentPack.PackageId;
  }

  // 模組可以選擇性地覆寫以下方法來掛接生命周期
  public virtual void Init() { }
```

```
    public virtual void OnGameStart() { }
}
```

`ModBase` 提供： - **建構子**：接受一個 `ModContentPack`，讓 Mod 類別在建立時知道自己對應的模組資料（例如可透過 `ContentPack` 取得自己的設定或資源路徑）。如果模組類別不需要此資訊，也可不定義對應的建構子（我們會在實例化時做對應處理）。 - **屬性**：如 `ModName` 用於識別模組，可從 ContentPack 帶入模組名稱或ID。 - **虛擬方法**：`Init()` 及 `OnGameStart()` 是模組生命周期的兩個關鍵點，稍後會詳細說明其作用。模組類別可選擇 override 這些方法，在對應時機插入自己的邏輯。預設實作為空，表示有默認無特殊行為。

我們也可以將上述結構改為介面（如定義 `IMod` 介面包含 `Init`/`OnGameStart` 方法），但使用抽象類別方便我們提供共用實作（例如保存 ModName 等），因此此處採用基底類別方式。

步驟4：掃描組件內的類別

有了辨識依據（`ModBase` 類別），現在遍歷先前載入的每個 `Assembly`，查找繼承自 `ModBase` 的類別。實現時需要注意過濾條件： - 必須是 **class** 類別型別，而非介面或值類型。 - 必須 **非抽象**（abstract），因為我們要能實例化它。 - 繼承自 `ModBase`（或實作 `IMod` 介面，如果採用介面方案）。

將這些條件應用在反射獲得的型別列表上。我們可以擴充前述的 `ModLoader.LoadAllMods()` 方法，在載入 DLL 後立即進行類別掃描和實例化，也可以在載入所有模組 DLL 之後再統一掃描。為了結構清晰，我們拆分成兩個過程：**載入DLL** 與 **創建Mod類別** 分開。實際上 RimWorld 也是先載入所有模組程式集，再創建 Mod 類別實例 [2]。

以下是掃描並實例化 Mod 類別的程式流程示意：

```
public void InitializeMods() {
  foreach (ModContentPack pack in LoadedMods) {
    foreach (Assembly asm in pack.Assemblies) {
      foreach (Type type in asm.GetTypes()) {
        if (type.IsAbstract || !type.IsClass) continue;
        if (!typeof(ModBase).IsAssignableFrom(type)) continue;

        // 找到繼承 ModBase 的具體類別
        try {
          ModBase modInstance = null;
          // 嘗試使用帶 ModContentPack 參數的建構子
          var ctor = type.GetConstructor(new Type[] { typeof(ModContentPack) });
          if (ctor != null) {
            modInstance = (ModBase)ctor.Invoke(new object[] { pack });
          } else {
            // 使用無參數建構，事後注入 ContentPack
            modInstance = (ModBase)Activator.CreateInstance(type);
            // 若模組類別繼承 ModBase，則可以設定其內的 ContentPack 屬性
            // 這裡假設 ModBase 提供 ContentPack 的 protected setter 或改為 public set
            modInstance.GetType().GetProperty("ContentPack")?.SetValue(modInstance, pack);
          }

          // 保存實例到 ModContentPack，並呼叫初始化方法
```

```
                pack.ModInstance = modInstance;
                modInstance.Init();
                Debug.Log($"已初始化模組：{pack.PackageId} 的 Mod 類別：{type.FullName}");
            } catch (System.Exception ex) {
                Debug.LogError($"模組類別 {type.FullName} 實例化失敗：{ex.Message}");
            }
        }
    }
  }
}
```

在這段程式中，我們對每一個模組的每一個已載入 Assembly 進行如下處理：

- 使用 `asm.GetTypes()` 列出組件中所有定義的型別（類別、結構、列舉、介面等）。對其中每個型別進行判斷：
- 跳過非類別型別以及抽象類別，因為我們無法實例化介面、抽象類別或其他型別。
- 使用 `typeof(ModBase).IsAssignableFrom(type)` 判定該型別是否繼承自 `ModBase`（注意：`IsAssignableFrom` 在給定型別是子類別時返回 true，包括子孫類別，同時可確保型別不同於 ModBase 本身）。
- 一旦確定找到一個模組類別（繼承自 ModBase），便嘗試建立其實例：
- 首先尋找是否存在接受 `ModContentPack` 作參數的建構子。如果有，則呼叫之，傳入當前模組的 `pack`。這對應模組類別宣告如 `public MyMod(ModContentPack contentPack) : base(contentPack) { ... }` 的情況，在 RimWorld 的模組中非常常見 [3]。
- 若沒有找到匹配的建構子，則嘗試使用無參數建構子實例化（`Activator.CreateInstance(type)`）。成功後，再將 `pack` 資訊注入。由於我們的 `ModBase.ContentPack` 是私有或唯讀屬性，這裡為簡化示範，直接嘗試透過 reflection 設定它（**實務上**可將 ContentPack 設為 protected set 或 internal，讓 ModLoader 能夠設定）。
- 將創建出的實例保存回對應的 `ModContentPack.ModInstance` 中，方便日後存取。接著呼叫此實例的 `Init()` 方法，執行模組初始化邏輯。
- 加入異常處理：如果在實例化過程中拋出任何錯誤（例如模組類別的建構子本身拋例外），記錄錯誤不中斷後續模組處理。

透過上述流程，我們就實現了**自動發現並初始化模組主要類別**的功能。當主程式啟動時，`ModLoader.LoadAllMods()` 會載入全部模組 DLL，接著 `ModLoader.InitializeMods()` 掃描並實例化所有模組的 `ModBase` 子類別。至此，每個模組的主類別已經建立，並執行了它們的 `Init()` 方法。

模組類別的初始化與遊戲整合

現在我們來討論模組 `ModBase` 類別在整個遊戲流程中的作用，以及如何讓模組邏輯與遊戲主程式發生互動。

`ModBase.Init()`：模組載入後的初始化

`Init()` 方法會在模組類別實例化後**立即**被呼叫。此時遊戲可能還在啟動階段，我們通常在這裡進行**與資源或資料相關的初始化**。可能的操作包括：

- **日誌紀錄**：輸出模組載入的信息。例如模組作者可以在這裡用 `Debug.Log` 顯示「XXX 模組已載入」等訊息來確認模組成功啟動。
- **模組設定載入**：如果模組有設定檔（例如存在設定值或需要讀取檔案），可以在此讀取。RimWorld 的 `Verse.Mod` 類別允許關聯一個 ModSettings，我們的模擬可在 `Init()` 中手動實作類似行為。

- **事件訂閱**：模組可以在此訂閱遊戲的全域事件或訊息總線。例如，如果主程式有定義事件 `GameEvents.OnDayPassed` 等，模組可在 `Init()` 中註冊監聽器，以便在遊戲進行中得到通知。
- **資料驗證**：若模組需要在遊戲開始前驗證某些資料（例如檢查相依的其他模組是否存在、或校正不合理的 Def 數值），可在此進行。此時 Def 資料可能已載入或尚未載入，視載入順序而定（RimWorld 中，Mod 類別實例化在 Def 加載之前 [2]，但我們可以選擇在 Def 資料完成後再調用 Init，以便模組可使用 Def 資料）。

  注意：初始化順序的安排很重要。如果模組的初始化需要讀取 Def，務必確保 Def 系統已就緒。在 RimWorld 中，由於 mod 類別構造執行早於 Def 載入，所以 mod 在該時點不應馬上操作 Def 資料。而我們的模擬可以彈性處理——例如選擇在載入 XML 後再執行 `InitializeMods()`，或讓 `Init()` 內部推遲使用 Def 資料直到 `OnGameStart()`。

`ModBase.OnGameStart()`：遊戲開始時的模組行為

`OnGameStart()` 顧名思義是在“遊戲正式開始”時呼叫的。所謂正式開始，通常是指玩家進入了遊戲世界，可以開始體驗玩法，例如 RimWorld 中創建或載入殖民地後。我們可以在以下時機觸發所有模組的 `OnGameStart()`： - 當玩家**新開一局遊戲**（世界初始化完畢，進入遊戲地圖時）。 - 當玩家**載入一個存檔**（遊戲世界從存檔讀取完畢時）。

這個時機點確保了絕大多數遊戲內容（包含地形、人物、物件等）都已就緒，模組可以在這裡安全地對遊戲狀態進行操作。例如： - 向地圖中添加模組自定義的物件或事件（例如生成一個特殊建築、觸發一個自訂事件）。 - 調整已載入的遊戲參數（例如修改某些 Def 數值，或應用模組的遊戲規則）。 - 顯示模組相關的 UI（例如開啟一個模組設定介面提示玩家）。

我們需要在主程式相應位置呼叫這些方法。可以設計一個在遊戲開始時迴圈呼叫所有 `ModContentPack.ModInstance.OnGameStart()` 的流程。例如，在主遊戲控制器裡：

```
public void StartNewGame() {
  // ... 遊戲世界創建與初始化邏輯 ...

  // 通知所有模組遊戲已開始
  foreach (ModContentPack pack in ModLoader.LoadedMods) {
    pack.ModInstance?.OnGameStart();
  }
}
```

同理，在載入存檔完成時也調用上述迴圈。這樣每個模組就有機會在遊戲開局時執行特定邏輯。

範例：模組主類別的簡易實作

以下是一個模擬的模組程式碼（位於模組的 DLL 工程中）的範例，展示如何繼承 `ModBase` 並利用上述機制：

```
// 模組DLL中的代碼（假設此檔編譯為 ModA.dll 並放在 ModA/Assemblies/ 下）
using UnityEngine;
using GameName.Defs;   // 假設這是主程式命名空間，包含 Def 系統
using GameName.Core;   // 假設這包含 ModBase, ModContentPack 等定義

public class MyFirstMod : ModBase {
  // 如果需要，可以定義建構子接收 ModContentPack
```

```
    public MyFirstMod(ModContentPack contentPack) : base(contentPack) { }

    // 覆寫 Init，在模組載入後執行初始化
    public override void Init() {
        base.Init();
        Debug.Log($"[{ModName}] 模組初始化中...");

        // 例如：訂閱遊戲每日結束的事件（需主程式有提供此事件）
        GameEvents.OnDayEnd += OnDayEndHandler;

        // 例如：驗證某 Def 數值範圍或存在性
        var sampleDef = DefDatabase<ItemDef>.Get("SampleItemDef");
        if (sampleDef != null && sampleDef.BasePrice < 0) {
            Debug.LogWarning($"[{ModName}] 發現 {sampleDef.defName} 基礎價格為負值，將重置為0。");
            sampleDef.BasePrice = 0;
        }
    }

    // 覆寫 OnGameStart，在真正開始遊戲時呼叫
    public override void OnGameStart() {
        base.OnGameStart();
        Debug.Log($"[{ModName}] 遊戲開始，執行模組啟動邏輯。");

        // 例如：將玩家殖民地中所有木材數量加倍 (演示對遊戲狀態的操作)
        foreach (var thing in FindObjectsOfType<Thing>()) {
            if (thing.defName == "WoodLog") {
                thing.stackCount *= 2;
            }
        }
    }

    // 自訂的事件處理函式
    private void OnDayEndHandler(int dayCount) {
        Debug.Log($"[{ModName}] 第 {dayCount} 天結束，自訂事件處理邏輯執行。");
        // 可以在每日結束時實現一些效果，例如產生一份日報等
    }
}
```

以上**模組程式碼**做了以下事情： - 宣告 `MyFirstMod` 類別繼承 `ModBase`。提供一個建構子調用 `base(contentPack)`，以便主程式在實例化時能傳入 `ModContentPack`。如果主程式找不到這個建構子，也會嘗試無參構造函數（我們有應變機制）。 - 在 `Init()` 中，打印日誌確認載入；假設存在一個 `GameEvents.OnDayEnd` 事件（例如每天遊戲時間結束會觸發並傳遞當前天數），模組將自己的方法 `OnDayEndHandler` 綁定上去，這樣從此每當遊戲日終了，該方法就會被呼叫。接著示範了資料驗證的邏輯：取得某個物品的 Def（假設通過 `DefDatabase<ItemDef>.Get` 方法可以按名稱取得），如果發現不合理的值則更正並發出警告。 - 在 `OnGameStart()` 中，模組執行開局時的邏輯。範例簡單地將所有地圖上木材堆疊量加倍，以展示修改遊戲物件狀態的效果。 - 定義了一個私有方法 `OnDayEndHandler`，作為事件監聽器，在每晚被呼叫時執行（這裡僅打印日誌，可想像延伸為更複雜的行為）。

當主程式按照前述步驟將此模組的 DLL 載入並建立 `MyFirstMod` 實例後：- `MyFirstMod.Init()` 將被呼叫，完成事件訂閱與資料校驗。 - 當玩家開始遊戲或載入存檔時，`MyFirstMod.OnGameStart()` 被呼叫，執行對遊戲狀態的調整。- 每當遊戲日結束，`OnDayEndHandler` 會因訂閱關係而被觸發。

透過這樣的流程，每個模組都能對遊戲做出擴充，且彼此邏輯獨立，不會互相干擾。

整合 Def 系統與模組程式碼

前一章節中，我們建立了 **Def 系統**，透過讀取模組的 Defs 資料夾內的 XML 定義，構造出各種遊戲數據的 Def 實例並存入 `DefDatabase`。現在我們關注**如何讓動態載入的模組程式碼利用這些 Def 資料**。

主要有以下幾種整合方式：

1. **模組新增新的 Def**：模組的 XML 可以定義新的遊戲元素（例如新武器、新生物等），這些會在 Def 加載階段被讀取並加入 DefDatabase。模組的程式碼可以使用自己新增的 Def 來實現功能。例如，上面 `MyFirstMod` 可以有對應的 `SampleItemDef` 定義在 XML 中，程式碼中透過 `DefDatabase<ItemDef>.Get("SampleItemDef")` 獲取它並操作。只要模組的 DLL **引用**了定義 Def 類別的程式集（通常是遊戲主程式 Assembly），它就能使用相同的 Def 類別型別。
2. **存取或修改現有 Def**：模組可以不新增新 Def，而是調整原有遊戲 Def 的屬性。例如修改一項配方的產出數量，或改變建築的耐久度。由於 DefDatabase 持有的是對象的引用，模組程式碼取得後直接修改會影響遊戲行為。例如我們可以：

   ```
   RecipeDef recipe = DefDatabase<RecipeDef>.Get("CookSimpleMeal");
   if(recipe != null) recipe.workAmount *= 0.5f; // 將烹調簡餐所需工作量減半
   ```

   這種修改應在適當時機進行（通常在遊戲開始前即可，或在 OnGameStart，視需要即時性）。
3. **模組程式碼擴充 Def**：如果模組需要新的 Def 類別（例如全新類型的定義數據），可以在其 DLL 中宣告繼承自某個基底 Def 類的子類型。Def 加載系統本身只要能識別該類型並實例化（XML 裡 `Class` 屬性指定到該類完全限定名），就可以創建出來。然而要注意的是，這種新的 Def 子類在未載入對應模組 DLL 前無法被認識。因此順序上應確保**模組 DLL 載入發生在 Def XML 解析之前** [4]。RimWorld 就是在載入模組程式集後才讀取所有 Def 定義，以便支持 mod 定義新的 Def 類別。

根據 RimWorld Wiki 的啟動順序說明，**載入模組 Assembly 與創建 Mod 繼承類實例**發生在讀取 Def 之前 [2]。我們的模擬系統也應遵照這個順序：由 `ModLoader.LoadAllMods()` 完成 DLL 載入，接著（可選地）調用 `InitializeMods()` 創建 Mod 類別，然後再執行先前章節實作的 Def 加載流程。這樣一來，如果模組 XML 中的 `<Defs>` 有 `Class` 指向該模組 DLL 裡的類別，.NET 也能順利找到並實例化它。

> 例：模組 A 的 Defs/ThingDefs 中有 `<ThingDef Class="MyMod.ThingDef_SpecialWeapon"> ... </ThingDef>`，而這個 `ThingDef_SpecialWeapon` 類是在模組 A 的 DLL 定義且繼承自 `ThingDef`。只要我們在解析 XML 前已用 Assembly.LoadFrom 載入了模組 A 的 DLL，則 XML 解析時能找到 `MyMod.ThingDef_SpecialWeapon` 類型並創建其實例，最終存入 DefDatabase。

為了示範模組對 Def 資料的操作，下面假設我們有一個全域的 Def 管理器 `DefDatabase<T>`（類似 RimWorld 的設計），提供一些存取方法。我們在模組代碼中已經展示過 `DefDatabase<ItemDef>.Get(string defName)` 的使用。`DefDatabase` 也許還有 `AllDefs` 屬性可枚舉所有定義。模組可以利用這些 API 來遍歷或檢索自己關心的 Def，進行邏輯處理。

例如，模組可能在 `Init()` 中檢查自己新增的 Def 是否與其他模組的衝突：

```
// 檢查是否有其他模組定義了同名的能力 (AbilityDef)
AbilityDef myAbility = DefDatabase<AbilityDef>.Get("FireballSpell");
if(myAbility != null && myAbility.modContentPack.PackageId != ContentPack.PackageId) {
  // 發現同名的定義且不是本模組的
  Log.Warning($"另一個模組也定義了 FireballSpell 能力，可能造成衝突。");
}
```

假設 `AbilityDef` 有個屬性可以標識來源模組（例如 RimWorld 中每個 Def 通過 `modContentPack` 知道它從何處載入），我們即可做如上判斷。這樣的資料驗證有助於提醒使用者可能的模組衝突問題。

總而言之，**Def 系統與模組程式碼的結合**使得 XML 資料驅動與程式碼邏輯得以協同工作：配置數據由 Def 定義，行為則由程式碼實現，兩者通過一致的識別（如 defName）和類型系統關聯。開發者在撰寫模組時，可以依賴 DefDatabase 來取得遊戲內容資訊，也可以向其中加入新內容，豐富遊戲的可擴充性。

支援多個模組並行運作

模擬環境下可能會同時載入多個模組。我們需要確保前面的機制對每個模組都有效，且彼此不互相干擾。以下是需要注意的幾點：

- **模組隔離**：由於每個模組的 DLL 我們都分別載入，它們各自處於同一應用程式域內，但在程式碼上是獨立的 Assembly。不同模組中如果有相同名稱的類別也不會衝突（因為在不同程式集中）。唯一需要避免的是全域靜態狀態的干擾——例如兩個模組都不應使用同一全域單例去存放不同目的的資料，否則可能相互影響。
- **載入順序**：一般而言，模組載入的順序可以不指定（例如按照資料夾列舉順序）。如果存在模組相依性，可以透過在 About.xml 中定義 `<loadBefore>` / `<loadAfter>` 調整，**但在本模擬中不深入相依關係**。我們預設所有模組平等載入。如果需要特定順序，可以在 LoadedMods 列表排序後再初始化。
- **初始化順序**：通常所有模組的 `Init()` 都會在 Def 加載完成後馬上被呼叫（或者至少在遊戲開始前）。`OnGameStart()` 則更晚，在玩家進入遊戲時同時呼叫。這意味著模組之間在 Init 階段也許可以互相看到對方的某些影響（例如一個模組在 Init 阶段修改了某 Def，另一個模組在 Init 階段讀取該 Def，就能看見修改結果）。這種情況下，開發者需要協調模組順序或在文檔標明。不過，一般模組之間默認不直接通信，除非使用者特意製作互動。
- **錯誤隔離**：正如前述，在載入和初始化每個模組時，都應做好異常捕獲。確保單一模組的錯誤（無論載入失敗或 Init 裡拋錯）都不會導致整個遊戲崩潰或停止載入其他模組。我們已經在程式碼中加入 try-catch 來保護這一點。此外，可以在 UI 上提示使用者哪個模組出現問題。
- **資源命名**：多個模組可能包含相同名稱的資源（如兩個模組都有 Def 名稱 "TestThing"）。通常 Def 是按各自命名空間或模組ID區分的，因此DefDatabase可以區分重名定義（在 RimWorld 中，不同 mod 的 defName 可以相同，但引用時需要指明來源）。在我們的模擬中，可以簡單約定**不允許不同模組出現重複 defName**，或如前所述加強檢查並警告衝突。至於程式類別，由於命名空間和 Assembly 的作用，不太會發生衝突。

經過上述設計，我們的系統應當能**正確初始化多個模組**：每個 ModContentPack 保留各自的 Assembly 列表和 ModInstance，它們的 Init 依序執行，而 OnGameStart 也會各自獨立被呼叫。這樣，各模組可以增加各自的功能，例如一個控制天氣，另一個增加武器，互不干擾地共存於同一遊戲中。

小結

本章我們完成了 RimWorld 模組系統核心部分的模擬實作：自動載入模組 DLL、透過反射發現並初始化模組類別，以及規劃模組在遊戲流程中的行為鉤子。我們強調了與 Def 系統的結合，使模組不僅能帶來代碼邏輯，還

能利用資料驅動的內容。如此一來，整個架構允許開發者編寫獨立的 DLL 來擴充遊戲，而且簡單地將其放入 Mods 資料夾就能被載入，達到**即插即用**的模組效果。

在進入練習題之前，請注意在 Unity 環境中調試動態載入的模組可能會遇到一些問題，例如程式集快取（Assembly Caching）或域重載。通常在 Editor 模式反覆測試時，重啟播放模式會重複載入 DLL，Unity 可能提示重複類別定義的錯誤。為避免這種情況，可以在每次進入播放模式前刪除已載入的模組，或在 Editor 下使用 `Assembly.LoadFile`（不建議）或者建立自訂的域來載入（Unity 預設不支援多 AppDomain）。由於這部分較為進階，本教材暫不深入，開發者可在遇到相關問題時查閱 Unity 對動態載入的支援情況。

接下來，透過一些練習來鞏固本章學到的概念和技巧。

練習題

1. **實作簡單模組並測試載入**：建立一個新的模組資料夾 “TestMod”，在其 Assemblies 裡放入您用 C# 編譯的 `TestMod.dll`。該 DLL 至少包含一個繼承自 `ModBase` 的類別，例如 `TestModClass`，其 `Init()` 中用 `Debug.Log` 輸出一行訊息。啟動遊戲，觀察日誌是否打印出對應訊息，從而驗證模組自動載入與初始化流程的正確性。

2. **擴充模組初始化順序控制**：嘗試修改 `ModLoader.InitializeMods()` 的實作，讓模組的 `Init()` 改在 Def 載入**之後**再執行（提示：可將初始化呼叫移至 Def 加載完成之後的某個時機觸發）。如此可以保證模組在 Init 中能安全地訪問 Def 資料。測試修改是否生效，例如在模組的 Init 中嘗試列出某種類型的全部 Def (使用 `DefDatabase<T>.AllDefs`) 並打印計數。

3. **新增事件鉤子**：為主程式設計一個新的全域事件，例如 `GameEvents.OnNewGameLoaded`（當每次新遊戲世界生成完畢時觸發），並在適當時機調用。修改模組類別，讓其除了 OnGameStart 外，再提供一個方法專門監聽此事件（或直接利用 OnGameStart 代表此事件）。模擬在多個模組同時訂閱時，確保所有模組的對應方法都能被呼叫。思考：如果希望模組能在**遊戲退出或場景切換**時執行清理，需要怎樣的機制？試著為模組類別增添一個例如 `OnGameExit()` 的方法，並在遊戲結束時遍歷呼叫，觀察它的作用。

4. **錯誤處理測試**：製造一個“惡意”模組，其 Mod 類別的 Init 或建構子中故意拋出異常。確認主程式在載入此模組時不會因此崩潰，並能正確記錄錯誤且仍然載入其他正常的模組。這有助於驗證我們的錯誤隔離機制。思考如何進一步加強穩定性，例如對多次連續出錯的模組是否要停用嘗試載入等。

範例檔案與專案架構建議

為了方便開發與測試模組系統，這裡提供 Unity 專案與 Visual Studio 解決方案的架構建議：

- **Unity 專案**：主要包含遊戲本體的程式碼和資源。
- 建立一個目錄（如 `Assets/Scripts/Modding`）放置與模組系統相關的腳本，例如 `ModLoader.cs`, `ModContentPack.cs`, `ModBase.cs`, `DefDatabase.cs` 等。
- 在專案根目錄（與 Assets 同層）建立一個 `Mods` 資料夾（如前所述）供模組存放。注意：Unity 預設不會將非 Assets 內的資料夾打包，所以這適合作為外掛模組的存放位置。若要在編輯器內測試，可以手動將模組 DLL 放入該資料夾。

- **（可選）** 為了在 Editor 模式下方便測試，編寫一個自訂編輯器腳本或在 `Awake()` 中調用 `ModLoader.LoadAllMods()`，確保一進入 Play 模式就載入模組。如果 Mods 資料夾位置特殊，記得在 Editor 下調整路徑（例如 `Application.dataPath` 在 Editor 下指向 Assets，需要定位到專案目錄可使用 `Directory.GetParent(Application.dataPath)`）。

- **Visual Studio 解決方案**：使用 VS 建立**額外的**類庫專案來編寫模組代碼。

- 建立一個 Solution（解決方案），例如命名為 "RimworldLikeMods". 在裡面增加多個 Class Library 專案，每個對應一個模組。比如 "ModAProject", "ModBProject" 等。
- 每個模組專案都需要**引用**遊戲主程式的必要程式集。通常可以將 Unity 提供的 `UnityEngine.dll`、`UnityEngine.CoreModule.dll` 引用上，還有遊戲主程式的腳本 Assembly。如果主程式腳本已編譯成例如 `Assembly-CSharp.dll`（Unity 默認命名），可以從 Unity 專案的 Library/ScriptAssemblies 找到並引用；或者更乾淨的做法是將 `ModBase` 等必要類別提取到一個單獨的可共享的 DLL（如建立一個 "GameModAPI.dll" 給模組專案引用）。 [5]
- 調整模組專案的**輸出路徑**，直接指向 Unity 專案的 Mods 對應模組資料夾下的 Assemblies。例如將 ModAProject 的輸出設為 `<Unity專案路徑>\Mods\ModA\Assemblies\` [6]。這樣，每次在 VS 編譯模組，結果 DLL 會自動覆蓋到 Unity 的 Mods 資料夾中，省去手動複製的麻煩。
- 在模組專案中，定義自己的 Mod 類別繼承 `ModBase`，以及其他所需的類別。撰寫完成後編譯，確認輸出 DLL 出現在正確的位置。

- 返回 Unity，按下 Play 進行測試，查看主控台日誌是否顯示模組載入消息以及執行效果。

- **檔案範例**：舉例來說，您的檔案結構可能如下：

```
UnityProject/
├─ Assets/
│    ├─ Scripts/
│    │    ├─ Modding/
│    │    │    ├─ ModLoader.cs
│    │    │    ├─ ModContentPack.cs
│    │    │    ├─ ModBase.cs
│    │    │    └─ DefDatabase.cs   (以及其他Def相關類別)
│    │    └─ ...（其他遊戲腳本）
│    └─ Resources/ ...（遊戲資源）
├─ Mods/
│    ├─ ModA/
│    │    ├─ About/ ... (模組描述文件)
│    │    ├─ Defs/ ...  (模組的XML定義)
│    │    └─ Assemblies/
│    │         └─ ModA.dll  (VS編譯輸出的模組程式集)
│    └─ ModB/
│         ├─ About/
│         ├─ Defs/
│         └─ Assemblies/
│              └─ ModB.dll
└─ RimworldLikeMods.sln  (VS解決方案，包含 ModAProject, ModBProject 等)
```

  如此安排能方便地在不重新編譯整個遊戲的情況下開發和調整各個模組。您可以隨時新增新的模組專案，或分享 Mods 資料夾給他人試用您的模組。

經由本章學習與以上架構實踐，您應該對 Unity 中實現類似 RimWorld 的模組系統有了清晰的認識。在後續章節中，我們可以進一步探討更多進階主題，例如更複雜的事件系統、模組相依性處理，以及如何運用 Harmony 庫進行更深層的遊戲行為修改等。透過不斷完善，最終能打造出一個靈活且強大的遊戲模組擴充架構。

[1] [2]

---

[1] [5] [6] Rimworld Mod制作教程1 认识Mod结构-CSDN博客
https://blog.csdn.net/qq_29799917/article/details/104692076

[2] [4] Application Startup - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/Application_Startup

[3] 【亲测免费】 RimWorld Modding Guide 教程原创 - CSDN博客
https://blog.csdn.net/gitblog_00051/article/details/141768614