ChatGPT

RimWorld Harmony 整合教學：在現有專案中實作範例 Patch

在這份教學中，我們將說明如何將 **Harmony** 套件整合進您的 RimWorld 模組專案（以 `angus945/moddable-study` 專案架構為基礎），並在 `Script/HarmonyTest` 資料夾中實作幾個簡單的 **Harmony Patch** 範例。Harmony 是目前修改 RimWorld 遊戲程式執行行為的最佳實踐方案 [1]。透過 Harmony，我們可以在目標函式執行的**之前**或**之後**插入自訂程式碼（分別稱為 Prefix 前置和 Postfix 後置） [2] 來改變或擴充遊戲功能。接下來我們將一步步完成以下內容：

1. **安裝 Harmony 的方式** – 說明如何將 Harmony 套件新增到專案（透過 NuGet、引用 DLL 或編輯 .csproj）。
2. **資料夾結構與檔案擺放** – 在專案的 `Script/HarmonyTest` 資料夾建立相關檔案與說明其用途。
3. **實作範例 Patch** – 三個逐步實作的 Patch 範例：
4. **範例 A**：遊戲開局時印出歡迎訊息（模擬 Mod 載入初始化過程）。
5. **範例 B**：攔截 `Thing` 的建立（透過 Prefix 在物件生成時修改其屬性或輸出訊息）。
6. **範例 C**：攔截簡單的 UI 行為（例如每次開啟任意選單視窗時印出訊息）。
   每個範例將提供：Patch 目標的類別與方法、使用的 Patch 類型（Prefix/Postfix）、完整註解之程式碼，以及執行後的預期行為來驗證效果。
7. **進階封裝與設計** – 說明如何將上述 Patch 邏輯進一步封裝為 `ILogicHook<T>` 介面或使用 Decorator 模式，以提高程式的可測試性與可維護性。

請按照以下步驟逐章節進行，以快速將 Harmony 整合進您現有的模組框架並實作範例功能。

1. 安裝 Harmony 套件

在開始撰寫 Harmony Patch 程式碼之前，首先需要在開發環境中安裝 **Harmony** 程式庫。以下提供三種方式，您可以選擇最適合您的環境者：

- **方法1：使用 NuGet 套件管理安裝** – 如果您使用 Visual Studio 或 Rider 等 IDE 開發 C# 專案，最簡便的方法是透過 NuGet 套件管理器安裝 Harmony。您可以在套件管理器中搜尋 **Lib.Harmony**（Harmony 2.x 的發行套件名稱）並安裝最新版本 [3]。這會自動將 Harmony 程式庫加入專案的參考。安裝完成後，專案引用中會出現 `0Harmony.dll`（Harmony 核心程式庫）。

- **方法2：手動下載 DLL 並引用** – 您也可以從 Harmony 的官方釋出頁面下載對應版本的 **0Harmony.dll** 檔案 [4]。下載後將此 DLL 檔案放入您的專案目錄中（例如 Unity 專案的 `Assets/Plugins` 資料夾，或傳統 .NET 專案的引用資料夾），再將它加入為專案參考。這種方式適合沒有直接使用 NuGet 的情況，例如 Unity 編輯器中使用 .asmdef 的專案。**注意：**如果您計畫發布此 RimWorld 模組到 Steam 工作坊，請**不要**直接將 `0Harmony.dll` 檔案包含在模組的 Assemblies 資料夾中，以免不同模組各自附帶舊版 Harmony 導致衝突；正確作法是讓使用者另外訂閱 Harmony 模組或在您的 About 文件中標註對 Harmony 的相依性 [4]。

- **方法3：編輯 .csproj 檔加入參考** – 若您偏好手動編輯專案檔，您可以直接在專案的 `.csproj` 檔中新增對 Harmony 的引用。一種方式是利用 `<PackageReference>` 節點來引用 NuGet 套件，例如：

```
<ItemGroup>
 <PackageReference Include="Lib.Harmony" Version="2.2.2" />
</ItemGroup>
```

這會在還原套件時自動取得 Harmony 2.2.2 並包含於編譯。或者，您也可直接新增 `<Reference>` 指向本機的 0Harmony.dll 路徑。如果是 Unity 專案，則可以在對應的 .csproj 中新增 `<Reference Include="0Harmony.dll" HintPath="Assets\Plugins\0Harmony.dll" />`。記得在編輯後重新載入專案使設定生效。

完成以上任一種安裝方式後，請確定您的專案可以引用 Harmony 庫：在程式碼中加入 `using HarmonyLib;`（Harmony 2.x 的命名空間為 HarmonyLib）以及 RimWorld 所需的命名空間（如 `using Verse;` 等），以便使用 Harmony 提供的特性（attributes）和類別。

2. 資料夾結構與檔案擺放

安裝 Harmony 後，接下來我們在專案中建立一個專門的資料夾來放置所有 Harmony 相關的 Patch 程式碼。根據專案架構，我們在 `Assets/Script/` 底下新增 `HarmonyTest` 資料夾，用於存放此次教學的示範 Patch 類別及初始化碼。設定完成後的資料夾結構可能如下所示：

```
Assets/Script/
└─ HarmonyTest/
    ├─ HarmonyInit.cs            <- Harmony 初始化與註冊 Patch 的程式碼
    ├─ WelcomeMessagePatch.cs    <- 範例 A：開局歡迎訊息的 Patch 類別
    ├─ ThingCreationPatch.cs     <- 範例 B：攔截 Thing 生成的 Patch 類別
    └─ UIPatch.cs                <- 範例 C：攔截 UI 開啟行為的 Patch 類別
```

HarmonyInit.cs：初始化 Harmony – 在撰寫各種 Patch 之前，我們需要一段初始化程式碼來讓 Harmony 掛載這些 Patch。通常的作法是定義一個帶有 `[StaticConstructorOnStartup]` 屬性的靜態類別，在其靜態建構子中建立 Harmony 執行個體並呼叫 `PatchAll()` [5]。這樣可以在模組載入時自動套用我們定義的所有 Patch。如:

```
using HarmonyLib;
using Verse;  // Verse 是 RimWorld 核心命名空間，提供 Log 等功能

[StaticConstructorOnStartup]
static class HarmonyInit
{
  static HarmonyInit()
  {
    // 創建 Harmony 執行個體，並以本模組唯一ID（例如作者_模組名稱）初始化
    var harmony = new Harmony("com.yourname.moddablestudy.harmonytest");
    // 自動尋找並套用當前程序集中的所有 HarmonyPatch
    harmony.PatchAll();
    Log.Message("[HarmonyTest] Harmony Patches 已初始化");
  }
}
```

上述程式碼在遊戲啟動載入您的模組時就會執行：Harmony 將掃描此程式集內標註了 `[HarmonyPatch]` 的靜態類別並套用我們定義的所有前置/後置函式。`Log.Message` 則用於確認初始化成功（訊息將出現在 RimWorld 的除錯主控台）。

> i **提示：** `[StaticConstructorOnStartup]` 是 RimWorld 提供的特殊屬性，任何帶有此屬性的靜態類別，其靜態建構子會在模組加載時自動執行 [5]。我們利用這點來作為 Harmony 的進入點。如果您沒有直接引用 RimWorld 的 Verse 函式庫，也可改用 RimWorld 的 Mod 類別（繼承自 `Verse.Mod`）的建構子來呼叫 Harmony 初始化程式碼。

完成目錄結構設置和 Harmony 初始化後，我們就可以著手實作各個範例 Patch。在 `HarmonyTest` 資料夾中，每個 Patch 範例各用一個 `.cs` 檔實作，接下來將逐一說明。

3. 實作範例 Patch

本節將透過三個小範例（A、B、C）來示範如何使用 Harmony 撰寫 Prefix/Postfix Patch。每個範例皆包含：**目標類別與方法**（即我們要插入程式碼的遊戲原始函式位置）、**Patch 類型**（使用前置 Prefix 或後置 Postfix）、對應的**完整程式碼**（內含中文註解），以及執行該 Patch 後的**預期行為**，方便您在遊戲中驗證功能是否生效。

範例 A：遊戲開局時印出歡迎訊息（模擬 Mod 載入初始化）

Patch 目標類別與方法位置： `Verse.Game.InitNewGame` 方法（RimWorld 在開始新遊戲時呼叫的方法）
Patch 類型： 後置 Postfix（在原始方法執行完之後插入）

完整程式碼： 以下我們建立一個 Harmony Patch 類別，針對 RimWorld 核心的 `Game.InitNewGame()` 方法進行後置處理。在新遊戲初始化完成後，輸出一條歡迎訊息到 RimWorld 日誌：

```
using HarmonyLib;
using Verse;

[HarmonyPatch(typeof(Game))]
[HarmonyPatch(nameof(Game.InitNewGame))] // 或直接寫 "InitNewGame"
static class WelcomeMessagePatch
{
  // Postfix 後置方法，參數對應原方法簽名（這裡 InitNewGame 無參數）
  static void Postfix()
  {
    // 在遊戲開局後執行歡迎訊息
    Log.Message("歡迎來到 RimWorld 模組世界！ (這是由 Harmony Patch 插入的歡迎訊息)");
  }
}
```

上述程式透過 `[HarmonyPatch]` 屬性標註目標為 `Game.InitNewGame`。當 RimWorld 執行到 **開始新遊戲** 的流程時，原本的 `Game.InitNewGame()` 方法結束後，Harmony 會觸發我們定義的 `Postfix()` 方法。此時我們呼叫 `Log.Message` 列印歡迎字串，模擬模組載入時的初始化訊息。

執行後預期行為： 將此 Patch 編譯進模組並啟動 RimWorld，開始新的遊戲時您應能在開發者主控台或輸出日誌中看到我們剛才定義的歡迎訊息。例如，當您在主選單選擇劇本並點擊「開始」載入新地圖後，日誌中會出

現：`歡迎來到 RimWorld 模組世界！（這是由 Harmony Patch 插入的歡迎訊息）`。這表示 Harmony 已成功在遊戲開局點插入了我們的自訂程式碼。您可以利用此技巧在模組初始化時執行設定或提醒玩家模組已載入。

範例 B：攔截 Thing 物件的生成（使用 Prefix 修改實例屬性或輸出訊息）

Patch 目標類別與方法位置：`Verse.ThingMaker.MakeThing(ThingDef def, ThingDef stuff)` 方法（遊戲中生成物件時呼叫的靜態方法）
Patch 類型：前置 Prefix（在原始方法執行前插入）

完整程式碼：我們將對 RimWorld 的物件工廠類別 `ThingMaker` 下手，攔截其 `MakeThing` 方法呼叫。在原方法執行**之前**（Prefix），取得將要被生成的物件定義 `ThingDef`，可以選擇性地修改參數或進行日誌輸出：

```
using HarmonyLib;
using Verse;

[HarmonyPatch(typeof(ThingMaker))]
[HarmonyPatch(nameof(ThingMaker.MakeThing))]
[HarmonyPatch(new Type[] { typeof(ThingDef), typeof(ThingDef) })] // 指定參數類型以匹配正確的 MakeThing 函式
static class ThingCreationPatch
{
  // Prefix 前置方法，攔截物件生成
  static void Prefix(ThingDef def, ThingDef stuff)
  {
    // 範例：在物件正式建立前，輸出將要生成的 Thing 定義名稱與材質(stuff)資訊
    string thingName = def.defName;
    string stuffName = stuff?.defName ?? "（無材質）";
    Log.Message($"[HarmonyTest] 即將生成物件: {thingName}, 材質: {stuffName}");

    // 您也可以在這裡修改輸入參數或進行其他邏輯
    // 例如：若生成木材(Log)時，把材質參數改為特殊材質 (這只是示意，實際需確保邏輯合理)
    // if (def.defName == "WoodLog" && stuff != null) { ...改變 stuff... }
  }
}
```

在此 Patch 中，`Prefix` 方法會在每次有物件透過 `ThingMaker.MakeThing` 被創建前執行。我們藉由攔截參數來得知**將要產生**的物件是什麼(`def`)以及其材質(`stuff`)，並將資訊打印到日誌。由於這是 Prefix，我們可以在原物件生成前做一些操作，例如根據條件改變傳入的參數或進行資料紀錄。如果我們想完全取代原本的生成行為，也可以讓 Prefix 回傳 `false` 來**跳過原方法**並手動指定 `__result`（進階用法，這裡僅提示其可能性 6）。

執行後預期行為：編譯並載入模組後，當遊戲中有任何物件被建立時（例如生成新角色、掉落戰利品、建造建築等），都將觸發我們的 Prefix。您可以打開開發者模式並觀察日誌，會看到類似訊息：`[HarmonyTest] 即將生成物件: Steel, 材質: （無材質）` 或 `...物件: MealSimple, 材質: Rice` 等等。這代表我們成功攔截了物件生成的呼叫點。例如，開局時地圖生成初始物品或後續製造物件時，都能即時看到這些訊息。透過修改 Prefix 內的程式碼，您甚至可以調整生成行為（但請小心，因為跳過或改變原方法可能影響遊戲平衡，需要謹慎驗證）。

範例 C：攔截簡單 UI 行為（開啟選單時印出訊息）

Patch 目標類別與方法位置：`Verse.WindowStack.Add(Window window)` 方法（遊戲中的視窗堆疊，用於新增並顯示介面視窗）
Patch 類型：前置 Prefix（在原始方法執行前插入）

完整程式碼：此範例展示如何攔截遊戲UI視窗的開啟。我們將針對 RimWorld 的 `WindowStack.Add()` 方法做 Prefix Patch，這個方法在**每次有新視窗介面被打開**時呼叫。我們的 Prefix 會取得將要加入的視窗物件，並將其類別名稱輸出到日誌：

```
using HarmonyLib;
using Verse;
using UnityEngine;  // Window 繼承自 UnityEngine.Object

[HarmonyPatch(typeof(WindowStack))]
[HarmonyPatch("Add")]
[HarmonyPatch(new Type[] { typeof(Window) })]
static class UIPatch
{
  // Prefix 前置方法，在任意新視窗加入時觸發
  static void Prefix(Window window)
  {
    // 輸出正在開啟的視窗類別名稱以供觀察
    string windowType = window.GetType().Name;
    Log.Warning($"[HarmonyTest] 一個新的介面視窗即將開啟: {windowType}");
  }
}
```

在這段程式中，每當遊戲透過 `Find.WindowStack.Add(new WindowType(...))` 打開任何介面對話框或選單時，我們的 Prefix 都會被呼叫 [7] [8]。透過 `window.GetType().Name` 我們可以取得即將開啟的視窗類別，例如**選項設定**對話框（`Dialog_Options`）、**存檔/讀檔**視窗（`Dialog_SaveFileList`）、**建築選單**（`ArchitectCategoryTab`）等，並將名稱以 `Log.Warning` 形式打印出來（使用 Warning 可以在日誌中醒目呈現黃色文字）。

> **註**：上述 `[HarmonyPatch]` 特性的用法指定了目標類別 `WindowStack` 及方法 `"Add"`，並用類型陣列明確限定我們要攔截的方法簽名為 `Add(Window)` [8]。這是為了避免遊戲中可能存在多個同名的 `Add` 方法而導致匹配錯誤。

執行後預期行為：載入此 Patch 後，嘗試在遊戲中執行各種會打開介面視窗的操作，觀察日誌輸出。例如：

- 打開**主選單**（點擊 Escape 或「選項」）時，應看到日誌輸出介面類型，如 `[HarmonyTest] 一個新的介面視窗即將開啟: Dialog_MainMenu` 或 `Dialog_Options` 等。
- 在遊戲中點擊右下角「選單」按鈕（存檔/讀取選單）時，日誌輸出 `Dialog_SaveFileList` 或相關視窗名稱。
- 打開其他 Mods 的設置介面、建造選單、科研選單等，只要有新 Window 類別彈出，都會記錄其類名。

這表示我們的 Harmony Patch 已成功攔截所有視窗的開啟事件。從這個範例可以延伸出許多實用技巧，例如在每次開啟特定介面時執行某些自動化動作，或配合 Prefix/Postfix 改變視窗的部分屬性。(本範例實作靈感參考自 Harmony 官方 Wiki 的 WindowStack.Add 教學 [7] [8]。)

4. 進階：封裝 Patch 邏輯以提升測試性與維護性

以上範例我們直接將需要執行的功能寫在 Harmony Patch 的 Prefix/Postfix 靜態方法中。對於簡單的功能這樣做沒有問題，但隨著模組邏輯變複雜，您可能希望提高程式的**可測試性**（例如在不啟動遊戲的情況下測試邏輯）以及**可維護性**（模組擴充或修改時更容易管理）。這時可以考慮將 Patch 的邏輯從靜態方法中**分離出來**，使用介面或 Decorator 等設計模式來封裝。

一種做法是定義通用的**鉤子介面** (Hook Interface)，例如這裡提到的 `ILogicHook<T>`。假設我們為每種類型的 Patch 定義一個介面，專門負責處理該 Patch 所對應的邏輯，例如：

```
// 定義一個泛型介面，用於封裝特定類型事件的處理邏輯
public interface ILogicHook<in T>
{
  void OnEvent(T data);
}
```

然後我們可以為不同的 Patch 編寫實作類別。例如，針對範例 C（視窗開啟），我們建立一個實作 `ILogicHook<Window>` 的類別來處理視窗開啟時的行為：

```
public class WindowOpenLogger : ILogicHook<Window>
{
  public void OnEvent(Window window)
  {
    // 將原本 Patch 中的邏輯搬移到這裡
    string windowType = window.GetType().Name;
    Log.Message($"(Hook) 開啟視窗: {windowType}");
  }
}
```

接著，我們調整原先的 Harmony Patch 前置方法，改為呼叫這個介面的實作：

```
[HarmonyPatch(typeof(WindowStack))]
[HarmonyPatch("Add")]
[HarmonyPatch(new Type[] { typeof(Window) })]
static class UIPatch
{
  // 透過依賴注入或服務定位獲取介面實例，預設可指向我們的實作類別
  public static ILogicHook<Window> WindowOpenHook = new WindowOpenLogger();

  static void Prefix(Window window)
  {
    // 將事件委派給可替換的邏輯處理實例
    WindowOpenHook.OnEvent(window);
  }
}
```

現在，`UIPatch` 的 Prefix 不直接包含業務邏輯，而是調用 `WindowOpenHook` 介面的方法。我們可以在單元測試中**替換**這個 `WindowOpenHook` 為不同的實作或假物件。例如，在測試環境中傳入一個紀錄呼叫的假實作，來驗證當有 Window 開啟事件時我們的 Hook 是否被正確呼叫，而無需真的啟動 RimWorld。這種利用介面抽象的方式，運用了**依賴反轉原則**（DIP），使得程式的各部分鬆耦合、易於替換。

除了介面之外，您也可以採用 **Decorator 裝飾者模式** 來增強可維護性。Decorator 可以讓我們將額外的功能與主邏輯分開。例如，我們可以實作多個 `ILogicHook<Window>`，一個專門處理核心邏輯，另一個專門記錄日誌，然後用裝飾模式將兩者組合：由記錄日誌的 Hook 裝飾核心 Hook，如此在每次事件發生時，先執行核心處理再記錄日誌，而兩者邏輯彼此獨立。如下：

```
// 核心處理 Hook：例如開啟特定視窗時自動暫停遊戲等
public class WindowOpenCoreLogic : ILogicHook<Window> { ... }

// 裝飾者 Hook：封裝一個內部的ILogicHook，先執行內部邏輯再執行自身邏輯
public class LoggingDecorator<T> : ILogicHook<T>
{
  private readonly ILogicHook<T> inner;
  public LoggingDecorator(ILogicHook<T> innerHook) { inner = innerHook; }

  public void OnEvent(T data)
  {
    // 執行內部核心邏輯
    inner.OnEvent(data);
    // 執行額外的記錄或其他輔助功能
    Log.Message($"[Decorator] {typeof(T).Name} 事件已處理於 {System.DateTime.Now}");
  }
}
```

透過上述設計，我們可以輕鬆組合不同的 Hook 行為，例如：

```
WindowOpenHook = new LoggingDecorator<Window>( new WindowOpenCoreLogic() );
```

使得每次有 Window 開啟事件時，同時執行核心邏輯和日誌紀錄。未來若要修改任一部分邏輯，只需針對相應的類別調整，而不影響 Harmony Patch 的架構。另外，由於 Patch 方法內部相當精簡（僅一行呼叫），**程式碼重複**減少，日後無論遊戲更新或需求變動，都比較容易維護和調試。

綜上，利用 `ILogicHook<T>` 介面和 Decorator 模式，我們達成了**單一職責**與**鬆耦合設計**：Harmony Patch 僅負責與遊戲方法接軌並將事件轉交，真正的模組邏輯則封裝在容易測試和替換的類別中。這種設計讓您的模組架構更具彈性，能夠隨專案成長輕鬆擴充功能，同時確保程式的可讀性與可靠性。希望透過本教學的逐步講解，RimWorld 模組開發新手們可以順利地將 Harmony 整合進現有框架並實現自己的創意功能！ [1] [4]

---

[1] [2] [4] [5] [6] Modding Tutorials/Harmony - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/Harmony

[3] Lib.Harmony 2.2.2 - NuGet
https://www.nuget.org/packages/Lib.Harmony/2.2.2

7 8 Home · pardeike/Harmony Wiki · GitHub
https://github.com/pardeike/Harmony/wiki