ChatGPT

第4章：Harmony 前綴 / 後綴 / Transpiler Patch 技術

在前面幾章中，我們介紹了 RimWorld 模組的架構與 Def 系統。本章將進一步探討 **Harmony** 資料庫提供的進階程式碼攔截與修改技術，協助開發者在**執行期**改變遊戲原始程式的行為。Harmony 是目前 RimWorld 模組開發中修改遊戲行為的最佳實踐 [1]。透過 Harmony，我們可以在**不更動**遊戲原始碼的前提下，於方法呼叫前後插入自訂邏輯，甚至直接更改方法內部的執行指令，使多個模組的改動能和平共存 [2]。

學習目標

- 理解 Harmony 的核心概念與運作原理，以及為何在 RimWorld 模組開發中需要使用 Harmony 來攔截/修改遊戲方法。
- 分辨 **Prefix（前綴）**、**Postfix（後綴）**、**Transpiler** 三種 Patch 方法的差異、用法及適用情境。
- 能夠編寫前綴與後綴方法，在原始方法呼叫前後執行自訂邏輯，或改寫原方法的傳回值。
- 初步了解 IL（中介語言）指令，學會使用 **Transpiler** 對原始方法的 IL 進行插入、刪除或替換，以實現更細微的行為修改。
- 將 Harmony Patch 整合進先前建立的模組框架：將撰寫的 Patch 程式碼編譯進模組 DLL，並於模組初始化時自動註冊這些 Harmony Patch。
- 建立一個完整的模組範例，示範如何透過 Prefix/Postfix/Transpiler 修改遊戲邏輯（如攻擊傷害計算）以及模組資料夾結構與必要資源的配置。

技術重點

- **Harmony 攔截原理：** Harmony 利用 runtime 動態替換方法的技巧，在方法執行時注入我們的Prefix/Postfix方法或改寫其IL指令，達到修改遊戲行為的目的 [3] [4]。相較於直接修改遊戲原始碼，Harmony Patch **非破壞性**且**相容性高**：原本的方法仍然保留，修改以附加方式進行，多個模組可同時對同一方法套用Patch而彼此平行運作 [2]。
- **Prefix與Postfix：** Prefix會在原方法**之前**執行，可選擇性地跳過原方法；Postfix在原方法**之後**執行，一定會執行且可讀寫原方法的返回結果 [5]。Prefix允許透過傳回 `false` 來阻止原方法執行（常稱「destructive prefix」），但需謹慎使用，因為這可能中斷其他模組的Prefix及原本邏輯 [6] [7]；Postfix則能安全地在原邏輯完成後執行，適合絕大多數需要增強或調整結果的情境 [8]。
- **Transpiler：** Transpiler 讓我們直接操作方法的 IL 指令序列，相當於在方法編譯完成後、執行前，對其內部實現做出**指令級**修改 [9]。這在 Prefix/Postfix 無法滿足需求時非常有用，例如想移除或改變原方法中間的一部分邏輯。但 Transpiler 技術門檻高，需深入瞭解 CIL 指令結構，撰寫與維護都較困難，因此僅建議在必要時使用 [10]。
- **完整範例實作：** 本章提供一個模組範例，模擬修改遊戲中「攻擊傷害計算」的功能。我們將示範使用 Prefix 讓攻擊傷害加倍（攔截原方法輸入或直接跳過原演算法），使用 Postfix 在原計算結果基礎上再調整傷害，以及使用 Transpiler 將乘算指令插入原方法的 IL 中，達到相同的加倍效果。範例程式碼包含於模組的 DLL 中，並透過 Harmony 在模組初始化時自動套用修改。
- **IL 指令與 CodeInstruction：** 我們將簡介 .NET 平台的中介語言（CIL）運作方式——基於堆疊的指令執行模型——以及 Harmony 提供的 `CodeInstruction` 結構如何封裝 IL 指令。了解如何遍歷原方法的指令列並偵測特定的模式或OpCode，然後以新增、移除或替換的方式修改指令流。例如，我們可以搜尋某個字段賦值指令並在其前插入額外的方法呼叫，以改變原程式的行為 [11]。
- **模組整合與自動註冊：** 我們會將 Harmony Patch 程式碼納入模組的 Assembly，並在模組啟動時自動註冊這些Patch。具體做法是在模組載入時建立 Harmony 實例並呼叫 `PatchAll()`，讓Harmony自動掃描並套用我們用 `[HarmonyPatch]` 標註的所有靜態Patch類別 [12]。為確保 Harmony 庫正確載入，我們不會將 0Harmony.dll 直接放入模組（避免版本衝突），而是利用 RimWorld 提供的**模組相依性**機制，將 Harmony 列為前置需求 [13]。

接下來，我們將按順序詳細說明這些內容。

1. Harmony 技術概念與原理簡介

Harmony 是什麼？為何要使用它？
Harmony 是一個用於 .NET 平台（包含 Unity 引擎）的程式庫，可讓開發者在**執行階段**攔截或修改既有方法的行為 [14]。對於 RimWorld 這類封閉原始碼的遊戲，Harmony 提供了一種**非侵入式（non-destructive）**的方法修改遊戲功能。我們無需直接改動遊戲的 Assembly-CSharp.dll 檔案，只要在自己的模組中編寫 Patch，Harmony 便會在遊戲載入時動態將這些 Patch **鉤入**原始方法的執行流程中。

Harmony 的工作原理可以概括如下：

- **方法攔截**：透過反射與IL操作，Harmony將目標方法替換為由Harmony產生的「替代方法」。當遊戲執行該方法時，實際上執行的是替代方法。
- **執行順序**：在替代方法內，Harmony 會依序呼叫所有對應的 Prefix 方法（若有），再視情況決定是否執行原始方法本身，待原方法完成後，再依序呼叫所有 Postfix 方法。如此一來，我們的 Patch 就能在**原方法執行之前或之後**插入自訂邏輯。
- **非破壞性修改**：Harmony 預設情況下並不移除原始方法的邏輯，只是增加前後綴或改寫部分指令。因此，正確撰寫Patch通常**不會改變**遊戲原有功能的其它部分，也較不會和其他模組牴觸 [2]。例如，兩個不同的模組都可以對同一個遊戲方法附加Postfix，各自的Postfix都會被執行而互不干擾。
- **多重 Patch 協調**：當多個模組 patch 同一方法時，Harmony 會按照**加載順序**將所有 Prefix 都在原方法前執行（除非有 Prefix **攔截**了執行），所有 Postfix 都在原方法後執行。各Patch的執行順序可透過 Harmony 提供的優先權屬性調整，但一般情況下不需要特別設定。

正因為有上述機制，Harmony 成為 RimWorld 模組開發改變遊戲行為的標準方式。簡而言之，**Harmony 允許模組「插手」遊戲程式在執行時的表現**，我們可以藉此修改或替換遊戲功能而不改動遊戲本身的程式碼。

2. 三種 Patch 技術：Prefix、Postfix、Transpiler

Harmony 提供了三種類型的 Patch 方法，對應不同的攔截與修改需求 [3]：

Prefix（前綴方法）

定義與特性： Prefix 是在原始方法**之前**執行的程式碼。它通常用於在原邏輯執行前進行檢查、修改參數、或決定是否中止原方法等。Prefix 方法的簽名可以是 `void` 或 `bool`；若定義為傳回 `bool`，Harmony 會將傳回值用於**控制**原方法的執行 [6]。當 Prefix 傳回 `false` 時，表示**跳過原方法**，也就是說原本的遊戲方法完全不會執行（後續其他帶有副作用的 Prefix 也會被略過） [6]。這種做法讓我們能**完全取代**原方法的功能。然而，需要特別注意的是，**過度或不當地跳過原方法可能導致相容性問題** [6]。例如，如果多個模組都有 Prefix patch 同一函式，其中一個傳回了 false，則其他模組的 Prefix 修改（若有依賴原方法執行結果）將被跳過，可能造成邏輯衝突。因此，Harmony 官方建議**除非必要，避免讓 Prefix 傳回 false** [7]。一般應僅在你要**完全改寫**該方法行為時，才使用這種「破壞性前綴」。相對地，若只需在原方法之前插入少量邏輯但仍讓原方法執行完畢，則 Prefix 應傳回 `true`（或者宣告為 `void` 型別）以保證原方法正常運行。

應用場景： 前綴適合在原方法執行前進行**輸入參數的調整**或**條件攔截**。例如： - 在攻擊計算函式開始處檢查攻擊者或目標的狀態，決定是否要執行原本的傷害計算。 - 攔截某個 AI 決策函式，在特定條件下直接返回自訂結果，跳過預設的 AI 決策流程。 - 修改方法參數：Prefix 可以透過將參數宣告為 `ref` 來更改傳入的參數值，從而影響原方法接下來的行為 [15]。

Prefix 範例： 假設遊戲中有一個 `Building_Trap.KnowsOfTrap()` 方法，原本用於判定角色是否知道陷阱存在。我們希望所有角色**永遠**知道陷阱（無視原本判定）。可以寫一個 Prefix 來達成：將結果強制設為 true，並跳過原方法。這個 Prefix 可能如下：

```
[HarmonyPatch(typeof(Building_Trap))]
[HarmonyPatch(nameof(Building_Trap.KnowsOfTrap))]
static class TrapKnowledgePatch {
  static bool Prefix(ref bool __result) {
    __result = true;            // 強制修改結果為 true
    return false;              // 跳過原始方法，直接使用我們設定的結果
  }
}
```

上例中，Prefix 傳回 false，因此 `KnowsOfTrap` 原方法體將不執行，直接以我們設定的 `__result=true` 作為返回值 16 。如此一來，我們成功**攔截並取代**了原本的判定邏輯。這種技巧非常強大，但請務必確認這麼做不會破壞遊戲的其他機制，並確保沒有其他模組需要依賴原方法執行。正如官方文件所強調的：**若只是小改動或副作用，應盡量使用 Postfix 或 Transpiler，以避免多個實作之間互相衝突** 7 。

Postfix（後綴方法）

定義與特性： Postfix 是在原始方法**執行之後**才觸發的程式碼 8 。與 Prefix 不同的是，無論原方法的執行是否被略過，所有 Postfix **始終**會被執行 17 。這意味著即使某個 Prefix 傳回了 false 跳過原方法，Harmony 保證 Postfix 依然會運行。Postfix 沒有傳回值（必須是 `void`），因為它只是附加在結尾的動作。然而，我們可以在 Postfix 中取得原方法的返回值，方法是以 `ref` 參數的方式來宣告一個與返回類型相符的 `__result` 參數 18 。Harmony 會在呼叫 Postfix 時，將原方法的傳回值傳入 `__result` 參數，讓我們可以讀取或修改它 19 。這種機制讓後綴能**改寫原方法的結果**或利用結果做後續處理。另外，Postfix 也能透過宣告和原方法相同類型與數量的參數名稱，來取得原方法的引數（包括實例方法中的 `this`，或額外使用 `__instance` 參數）供使用。

應用場景： 後綴適合用在需要**在原邏輯之後**執行額外動作的情境，尤其是當我們**不想干擾**原本的執行，只是對結果加以調整或觸發附加效果。例如：- 攻擊傷害計算完成後，額外增加固定值或百分比的傷害（修改返回的傷害值）。- AI 決策函式跑完後，我們在 Postfix 取得AI選擇的結果，然後也許記錄日志或對結果做一些限制（但不改變原結果）。- 某角色狀態更新方法執行後，我們補充一些自訂狀態變化（如額外的效果觸發）。

Postfix 範例： 例如有一個 `OriginalCode.GetName()` 方法會返回某個名稱字串。如果我們希望將特定返回值由 `"foo"` 改成 `"bar"`，可以撰寫 Postfix 如下：

```
[HarmonyPatch(typeof(OriginalCode))]
[HarmonyPatch(nameof(OriginalCode.GetName))]
static class NamePatch {
  static void Postfix(ref string __result) {
    if (__result == "foo")
      __result = "bar";
  }
}
```

在這個後綴中，`__result` 參考了原方法的返回結果[20]。當原方法返回 `"foo"` 時，我們將其改為 `"bar"`。由於 Postfix 總是在原方法之後執行，我們不影響原方法內部邏輯，只是**攔截並更改**了最後的結果。這種方式對遊戲其他部分或其他模組的干擾最小，是**最為相容**的修改方式之一[8]。

Transpiler（IL 編碼替換）

定義與特性： Transpiler 類似一個**編譯後置器**。它並不像 Prefix/Postfix 那樣在運行時直接執行於方法之前或之後；相反，Transpiler 在我們呼叫 `Harmony.Patch()` 時**當下**就對目標方法的 IL 做出修改[9]。簡而言之，Harmony 會將目標方法的 IL 指令提取出來，交給我們的 Transpiler 方法進行處理後，再將修改後的指令序列存回該方法定義中。之後當遊戲執行該方法時，它其實執行的是我們修改過的指令序列。由於 Transpiler 直接改變了方法內部邏輯，它可以實現 Prefix/Postfix 做不到的一些**深度改造**：例如移除原方法中間的某段程式碼，修改常數值，插入額外的運算等[21]。但是，這種強力的修改也意味著我們必須非常熟悉原方法的 IL 細節。一個 Transpiler patch 往往需要針對特定的 IL **模式**進行識別和操作，同時還要避免破壞 IL 的正確性和平衡（例如堆疊平衡、跳轉目標正確等）[22]。開發者需要懂得閱讀和編寫 CIL 指令，並瞭解 Unity/Mono 的執行細節，這讓 Transpiler 成為 **最困難**但也**最靈活**的 Harmony 用法。

應用場景： 當 Prefix 和 Postfix 都無法實現所需改動時，就可能需要考慮 Transpiler。例如： - **細粒度邏輯修改：** 原方法內部有一小段運算我們想改變（例如把公式中的 `*2` 改成 `*3`），如果用 Prefix/Postfix，可能很難精準地在不重寫整個方法的情況下改這一部分，但 Transpiler 可以直接找到乘2的指令改成乘3。 - **移除原有限制：** 遊戲方法中可能有一段 `if` 條件限制了某些行為，我們可以用 Transpiler 找到這段條件相關的 IL 指令並將其移除或修改，從而移除限制。 - **插入新流程：** 想在原方法的中間某個點插入新的方法呼叫或流程。例如在傷害計算中，原本沒有某種檢查，我們可以插入額外的檢查邏輯。

Transpiler 範例： 為了說明 Transpiler 的基本寫法，我們假設有個方法包含將某個欄位 `someField` 賦值的指令，我們希望在**賦值發生之前**插入一個自訂方法呼叫 `MyExtraMethod()`。一個簡化的 Transpiler 可以這樣寫：

```
static FieldInfo f_someField = AccessTools.Field(typeof(SomeType), "someField");
static MethodInfo m_MyExtraMethod = AccessTools.Method(typeof(Tools),
nameof(Tools.MyExtraMethod));

static IEnumerable<CodeInstruction> Transpiler(IEnumerable<CodeInstruction> instructions) {
  bool found = false;
  foreach(var instr in instructions) {
    // 找到 stfld someField 指令
    if(instr.opcode == OpCodes.Stfld && instr.operand == f_someField) {
      // 在它之前插入對 MyExtraMethod 的呼叫
      yield return new CodeInstruction(OpCodes.Call, m_MyExtraMethod);
      found = true;
    }
    yield return instr; // 繼續返回原指令
  }
  if(!found) {
    Log.Error("未找到目標指令，Transpiler未產生作用");
  }
}
```

如上，Transpiler 方法接受原方法的 IL 指令列表（`instructions`）並進行遍歷。我們檢查每一條 `CodeInstruction`，當偵測到符合條件的指令（在此例是 `stfld someField`）時，使用 `yield`

`return` 插入一條新的指令（呼叫我們自定的 `MyExtraMethod`）[23]。接著再 `yield return` 原本的指令，確保原邏輯繼續。這樣，我們就在目標指令之前成功插入了額外的函式呼叫。最後，如果遍歷完畢仍未找到目標指令，不妨記錄錯誤提示方便調試。這僅是一個插入指令的簡單案例；實際上我們可以用類似方式**修改**現有指令（例如改操作碼或操作元）、**刪除**特定指令（跳過 `yield return` 某些instr）等。寫 Transpiler 時的一個建議是：**改動越小越好**，並且儘量以動態、彈性的方式匹配指令，而非依賴硬編碼的指令索引，以便未來遊戲更新或其它 mod 的 Transpiler 也能兼容[21]。

開發與調試工具： 由於撰寫 Transpiler 需要了解原方法的 IL，我們通常需要借助反編譯工具來查看 RimWorld 原始程式的 IL 代碼。常用的工具如 **ILSpy**（可以將C#切換到IL視圖）或 **dnSpy**（功能強大的除錯級反編譯工具）[24]。步驟一般是：用反編譯工具打開 RimWorld 的程序集，找到目標類別的方法，檢視其 IL 指令，據此決定我們的 Transpiler 要搜尋的模式和修改策略。一旦寫好 Transpiler，可以利用Harmony的日誌功能（例如開啟 Harmony 的 DEBUG 模式）來驗證 Patch 是否正確應用。此外，Harmony 提供了一個方便的靜態類 `AccessTools` 來取得私有欄位、方法資訊，以及 `CodeInstruction` 提供了許多靜態方法幫助比對指令（例如上例中的 `instr.opcode == OpCodes.Stfld` 也可以寫成 `instruction.StoresField(f_someField)` 等語法糖[11]），這些都能降低撰寫 Transpiler 的難度。

3. 完整模組範例：修改攻擊傷害的三種 Patch 應用

為了將上述概念串連起來，下面我們通過一個**完整模組範例**來模擬實作 Prefix、Postfix 和 Transpiler 的應用。假設 RimWorld 有一個方法負責計算近戰攻擊傷害，我們以此為目標，展示如何用不同方式進行修改。為簡化說明，我們構造一個假想的方法簽名，如：

```
// 位於某遊戲類別，例如 CombatTools
public class CombatTools {
  // 計算近戰攻擊傷害的原方法
  public static int CalculateMeleeDamage(Pawn attacker, Pawn defender, int baseDamage) {
    int damage = baseDamage;
    // ... 這裡包含原本複雜的傷害計算邏輯 ...
    return damage;
  }
}
```

> **範例目標**：我們希望**加倍**所有近戰攻擊造成的傷害。也就是說，無論原本的計算公式如何，我們的模組要將最終傷害值提高兩倍。

接下來，我們示範三種不同的Patch方案來達成目標。**注意：**以下三種方案是獨立的實現方案，一般情況下在實際模組中只會擇一使用，這裡將它們並列是為了教學演示。

3.1 前綴方案：攔截並替換傷害計算

利用 Prefix，我們可以在 `CalculateMeleeDamage` 執行前先介入邏輯，直接決定其返回值，而**不執行原始方法**。這樣可以完全掌控結果。實現步驟如下：

1. **撰寫 Prefix 方法**：我們的 Prefix 應具備與目標方法相對應的簽名。因 `CalculateMeleeDamage` 是靜態方法，Prefix 也需是靜態；原方法返回 `int`，我們的 Prefix 可以宣告一個 `ref int __result` 參數來設置返回值；我們還想拿到原方法的輸入參數以便計算，因此可以在 Prefix 的參數列表中包含 `int baseDamage` 等。同時為了跳過原方法，我們讓 Prefix 回傳 `bool`。

2. **在 Prefix 中實現加倍邏輯**：我們將 `__result` 設為 `baseDamage * 2`（簡單起見，假設原方法原本應該返回經過某些計算的 damage，但我們決定直接用 baseDamage 的兩倍作為結果）。然後讓 Prefix 回傳 `false`，以跳過原方法後續執行。
3. **標註 [HarmonyPatch]**：使用 `[HarmonyPatch]` 屬性標註一個靜態類，使其表示我們要 Patch 的目標類型和方法。Harmony 會據此找到 `CombatTools.CalculateMeleeDamage` 並套用我們的 Prefix。

實際程式碼如下：

```
using HarmonyLib;
using RimWorld;  // 假定 Pawn 定義在 RimWorld 或 Verse 命名空間
// ... 其他 using 例如 Verse 等, 根據實際需要

[HarmonyPatch(typeof(CombatTools))]
[HarmonyPatch(nameof(CombatTools.CalculateMeleeDamage))]
static class MeleeDamageDouble_PrefixPatch {
  // Prefix 方法：加倍傷害並跳過原計算
  static bool Prefix(int baseDamage, ref int __result) {
    // 將結果直接設定為原本基礎傷害的兩倍
    __result = baseDamage * 2;
    // 跳過原始 CalculateMeleeDamage 的執行
    return false;
  }
}
```

行為分析： 套用此 Prefix 後，當遊戲呼叫 `CombatTools.CalculateMeleeDamage(attacker, defender, baseDamage)` 時，Harmony 會先執行我們的 `MeleeDamageDouble_PrefixPatch.Prefix` 方法。該方法將計算結果直接設為兩倍的基礎傷害，並返回 false，告訴 Harmony **不要執行原始方法**。因此，`CalculateMeleeDamage` 原本內部的複雜公式完全被略過，每次都返回我們設定的值。這滿足了傷害加倍的目標。然而，這種「一刀切」的做法也**取代**了原本所有的傷害計算邏輯，可能忽略了一些遊戲平衡因素（比如攻擊者狀態、防禦者防禦力等原本應體現的影響）。在實務上，除非確定這就是我們想要的效果，否則 Prefix 更常用於在執行原方法**之前**做檢查或輔助工作，或在特定條件下**有選擇地**跳過原方法。例如，我們可以加上條件判斷只有在某種特殊武器時才返回 false 跳過原方法，否則仍然 return true 繼續執行原計算。總之，Prefix 提供了**最高控制權**，但也要謹慎避免破壞遊戲原有的計算平衡或跟其他 mod 的衝突。

3.2 後綴方案：保留原邏輯並調整結果

使用 Postfix，我們可以在原傷害計算執行完成後，再來調整它的結果。這樣做的優點是**保留了原本的所有計算過程**，只是最後改動輸出。因此若遊戲原本考慮了很多因素計算傷害，我們不會丟失那些影響，只是在終點乘以2放大效果。

實現步驟：

1. **撰寫 Postfix 方法**：因為要取得原方法的返回值，我們在參數中宣告 `ref int __result`（型別為原方法返回的 int）。另外，我們也可以列出原方法的參數（如需要攻擊者或防禦者資訊，可一併寫上 `Pawn attacker, Pawn defender, int baseDamage`）供使用。不過在這例子中加倍僅需原結果即可，所以可省略原參數。

2. **實現加倍邏輯**：直接對 `__result` 做 *= 2 的操作，將原本的結果翻倍。由於 Postfix 總是在原方法之後執行，此時 `__result` 已帶有原計算出的 damage 值。
3. **標註 [HarmonyPatch]**：和 Prefix 類似，標註目標類別及方法。本例我們可以用另一個 Patch 類來實現，以區別於 Prefix 範例。

程式碼如下：

```
[HarmonyPatch(typeof(CombatTools))]
[HarmonyPatch(nameof(CombatTools.CalculateMeleeDamage))]
static class MeleeDamageDouble_PostfixPatch {
  // Postfix 方法：在原計算結果基礎上翻倍
  static void Postfix(ref int __result) {
    __result *= 2;
  }
}
```

行為分析： 有了這個 Postfix，遊戲每次計算完傷害後，Harmony 都會將原結果讀入 `__result`，然後執行我們的乘2操作。舉例而言，如果原計算根據各種因素得到傷害值 15，經過我們的 Postfix，最終結果變為 30 並返回遊戲。相較於 Prefix 方案，Postfix **不改變**原方法執行流程，只在最後關頭調整結果，因而**風險較小**且**相容性高** [8]。其他模組若也對這個方法做了 Postfix，彼此都能作用於原結果（執行順序取決於載入順序，但都會執行）。需要注意的是，如果有另一個模組用了 Prefix 且傳回 false 跳過了原方法，那原方法沒跑但我們的 Postfix仍會執行。在這種情況下 `__result` 可能是被其他 Prefix 設定的值。我們的乘2仍會套用在那值上。所以在 Postfix 中也要小心處理，假如遇到某些不期望的值時可以有條件地調整，避免與他人的 Prefix 結果重複作用。不過總的來說，Postfix 非常適合**增強**或**微調**原有功能。

3.3 Transpiler 方案：修改傷害計算方法的內部實現

最後，我們嘗試使用 Transpiler 來達成同樣的傷害加倍效果。設想原始的 `CalculateMeleeDamage` 方法內部某處，最終要將計算出的傷害值返回。可能原程式碼片段（以C#角度想像）是：

```
int damage = ... // 經過一系列計算得到最終傷害
return damage;
```

相應的IL指令（簡化表示）可能是：

```
IL_00xx: stloc.0      // damage 放入本地變數0
IL_00yy: ldloc.0      // 將本地變數0載入計算堆疊
IL_00zz: ret          // 返回
```

如果我們想把返回前的數值*2，只需在 `ldloc.0` 之後插入一條乘2的IL指令。乘2在IL中可以用 `Ldc_I4_2`（載入常數2）加 `Mul`（乘法）兩條指令來實現。Transpiler 允許我們做到這點：

實現步驟：

1. **撰寫 Transpiler 方法簽名**：Transpiler 一律是 `static IEnumerable<CodeInstruction> Transpiler(IL列舉, [ILGenerator], [MethodBase])` 這樣的格式。我們至少要接收

`IEnumerable<CodeInstruction>`，其他兩個參數視需要可選。在此例中不需要額外資訊，可省略 ILGenerator 和 MethodBase。

2. **遍歷原始指令並插入修改**：我們需要偵測出在 IL 中「將計算結果載入堆疊準備返回」的地方。由於我們知道返回類型是 int，通常可尋找 `OpCodes.Ret` 前面對局部變數的載入。比如偵測 `ldloc.0`（假設 damage 編號是0）後面緊跟 `ret` 的模式。在 Harmony 裡，我們可以簡單地檢查每一條指令，如果它的 opcode 是 `OpCodes.Ret`，那前一條可能是我們要的 `ldloc`。但不同編譯器優化可能導致略微不同的 IL，因此為謹慎起見，也可以用更可靠方式：先記錄所有指令，找出最後的 `ret`，往前看一條是否為 `ldloc`，等等。這裡為示範，我們採用簡單策略，直接在遇到 `Ret` 時處理。
3. **插入乘2指令**：一旦定位到Ret，我們就在Ret之前插入兩條指令：載入整數常值2（`Ldc_I4, 2`）和乘法（`Mul`）。由於此時返回值應該已經在堆疊頂端，這兩個指令會把它乘以2並留在堆疊頂端作為新的返回值。
4. **產出新的指令序列**：透過對傳入的 `instructions` 做遍歷並 `yield return`，依序送出我們的新序列。遇到插入點時，多送出我們的指令。

程式碼實現如下：

```
[HarmonyPatch(typeof(CombatTools))]
[HarmonyPatch(nameof(CombatTools.CalculateMeleeDamage))]
static class MeleeDamageDouble_TranspilerPatch {
  static IEnumerable<CodeInstruction> Transpiler(IEnumerable<CodeInstruction> instructions) {
    foreach(var instr in instructions) {
      if(instr.opcode == OpCodes.Ret) {
        // 在 return 前插入常數2和乘法指令
        yield return new CodeInstruction(OpCodes.Ldc_I4, 2);
        yield return new CodeInstruction(OpCodes.Mul);
      }
      yield return instr;
    }
  }
}
```

行為分析： 這個 Transpiler 每迭代到一個指令時，都先檢查是否為 `Ret`。當碰到方法結尾的 `Ret` 指令時，我們先 `yield return` 兩條指令：一條將整數2壓入堆疊，一條將堆疊頂的兩個值相乘。由於在Ret前，原本堆疊頂端正是原方法準備返回的 `damage` 值，插入後執行順序變成：「把原 damage 壓棧 -> 壓入2 -> 相乘 -> Ret返回」。乘法會消耗原 damage 和常數2並推送結果，於是 Ret 就返回了**兩倍**的 damage。經過這個 Transpiler，原方法的IL被永久改寫成帶有乘2的版本。這達到與先前 Prefix/Postfix相同的效果。值得注意的是，如果將此Transpiler和前面的Postfix同時作用在此方法上，結局可能變成四倍傷害（Transpiler已經把值翻倍，Postfix又再翻倍）。因此我們在實作模組時需確保只使用一種 Patch 方法來實現目標，並留意其他 mod 是否對同一方法有類似 Patch。

比較與評估： Prefix、Postfix、Transpiler 三種方案各有優劣： - Prefix 方案**最簡單**直接，但**覆蓋**了原計算過程，在多Mod環境下風險較高（除非透過條件避免過度攔截）。 - Postfix 方案**安全相容**，讓原邏輯與我們的調整**共存**，大多數情況下是首選方案。 - Transpiler 方案**精確**但**複雜**，只有當需要改動原方法內部邏輯（Prefix/Postfix無法辦到）時才考慮使用。而且寫Transpiler要求開發者具備IL知識和充分測試，否則潛在錯誤不易察覺。

實際開發中，應根據需求選擇**最適合的Patch類型**：能用Postfix解決的，就不採用Prefix；非得改內部運算且無法繞過時，再投入時間撰寫Transpiler。另外也要注意性能因素：Prefix/Postfix屬於極輕量的呼叫插入，對性

能影響可忽略不計，而複雜的Transpiler在Patch階段略有開銷（修改IL本身耗時），但Patch完成後執行時效能與修改後的原方法幾乎無異。

4. Transpiler 的 IL 操作基礎

由於 Transpiler 是 Harmony 中相對艱深的主題，本節將對 **CIL（Common Intermediate Language）** 以及 Harmony 提供的 IL 操作接口做一點基礎介紹，幫助初學者理解上一節 Transpiler 範例中到底做了什麼。

IL 簡介： CIL 是 .NET 平台上高階語言（如 C#）編譯後得到的中介語言，也稱為 **MSIL** 或 **IL**。它類似於虛擬機的組合語言指令集，所有 .NET 程式（包括 Unity遊戲的C#程式）在執行前都會被JIT編譯將 IL 轉成機器碼。了解IL的執行模型對寫Transpiler很重要： - IL 是**堆疊導向**的指令集 [25] 。也就是說，大部分指令以**操作堆疊**的值為目標。例如，加法指令會從堆疊彈出兩個操作數，相加後將結果壓回堆疊；方法呼叫指令會彈出所需參數並執行，若有返回則將返回值壓回堆疊。 - IL 指令通常由一個操作碼（opcode）加可選的操作數組成。例如前述的 `Ldc_I4, 2` 是操作碼 `Ldc_I4` 搭配操作數 `2`，表示「將常數4位元整數2推入堆疊」；`Mul` 則是單獨的操作碼，表示從堆疊彈出兩個值相乘。 - 每個方法的IL指令是線性排列的，中間可以有標誌（label）作為跳轉目標。控制流指令（如 `brtrue`, `brfalse`, `jmp` 等）改變執行順序。 - 編譯器有時會對高階語言進行優化，使得 IL 的順序不一定和原始C#直觀順序完全一致。例如常見的情況： `if/else` 可能被編譯成若干跳轉指令、將某些計算提前或延後等。因此在撰寫 Transpiler 時，要對照 IL 和原始碼小心分析，不可想當然地匹配字面C#的順序。

Harmony 的 IL 操作工具： Harmony 透過 `CodeInstruction` 類別來表示單條 IL 指令。我們在 Transpiler 方法中操作的 `IEnumerable<CodeInstruction>` 就是原方法的指令流序列。`CodeInstruction` 除了包含 `opcode` 和 `operand`（操作數）外，還攜帶了標記、分支目標等資訊以供我們在修改時保持正確。常用的方法和屬性包括： - `instruction.opcode` 和 `instruction.operand` 可用來讀取或更改指令內容。 - 靜態屬性/方法例如 `OpCodes.Brfalse`（各種OpCode定義）可用於比較或設置指令操作碼。 - `instruction.Calls(MethodInfo)` / `instruction.LoadsField(FieldInfo)` / `instruction.StoresField(FieldInfo)` 等輔助方法，可快速判斷當前指令是否為對某方法呼叫或對某欄位存取 [11] 。這在尋找目標指令時非常有用。 - `new CodeInstruction(OpCode, operand)` 可建立新指令，我們在前例中即使用它來構造 `Ldc_I4` 和 `Mul` 指令。

Harmony 也提供一些進階特性，例如**匯入匯出本地變數**、**標籤重定位**等，但超出本章範圍。對初學者而言，掌握基本的遍歷和插入即可完成許多簡單任務。

編寫 Transpiler 的一般流程： 1. **取得原方法 IL**：使用 ILSpy/dnSpy 打開遊戲Dll，找到目標方法，切換到 IL 視圖。觀察想修改的部分在 IL 中呈現為哪些指令序列，記錄關鍵的 OpCode 和操作數模式。 2. **撰寫匹配邏輯**：在 Transpiler 中遍歷 `instructions`，尋找步驟1中記錄的模式。可用一連串的 `if` 或更進階的查詢方式匹配連續的指令。例如本例中，我們鎖定最後的 `Ret` 並假設前面一條是我們要修改的值載入。但在更複雜情況下，也許需要匹配多條序列，這可能用到列表窗口或狀態機來實現匹配。 3. **構造新指令**：當找到匹配點時，決定是要插入、刪除還是修改。插入就 `yield return` 額外的CodeInstruction；刪除則跳過對某些原 instruction 的 `yield`；修改則可以改變 `instruction.opcode` 或 `operand` 後再 `yield return`。 4. **保持其餘部分原樣**：沒有命中修改的指令，照常 `yield return` 輸出，確保大部分原邏輯不變。 5. **測試**：啟動遊戲測試功能是否如預期。必要時開啟Harmony的 debug log，或使用 dnSpy 在運行時附加偵錯，檢查方法的最終 IL 是否符合預期。

雖然 Transpiler 複雜，但靈活運用得當可以實現幾乎任意的修改。本節重點在讓讀者對 IL 編碼與 Harmony Transpiler 有初步概念，更多進階內容（如處理泛型方法、try-catch 區塊、優化性能等）可參考 [Harmony 官方文件][HarmonyDoc] 以及 RimWorld 社群的相關討論資源。

5. Harmony Patch 在模組中的整合

了解了如何撰寫各種 Patch，下一步就是將這些 Patch **整合進我們的模組**。本節將說明在 RimWorld 模組架構下，如何組織Harmony相關的程式碼，以及在模組初始化時確保 Patch 正確註冊。

5.1 引入 Harmony 資料庫

首先，我們需要在開發環境中取得 Harmony 函式庫供編譯使用。Harmony 2.x 可從官方 GitHub Release 或 NuGet 取得，也可以直接從 Steam 訂閱 **Harmony** 模組（由官方/社群維護，用於提供 Harmony 執行檔給遊戲） [13] 。重點是：**不要**將 Harmony 的DLL檔直接放進你的模組 Assemblies 資料夾 [13] 。RimWorld 在 1.1 版本之後採用了外部載入Harmony的方式，如果每個模組各帶一份 Harmony.dll，可能因版本不一而導致衝突。正確做法是： - 在你的模組的 About.xml 中宣告對 **Harmony** 模組的相依性。這會確保玩家安裝你的模組時，遊戲也會加載 Harmony 模組並提供 Harmony 函式庫給所有模組使用。 - 編譯時，仍需參考 0Harmony.dll（可以從安裝的 RimWorld\Mods\Harmony 資料夾下找到，或以 NuGet 套件引用），以便使用 HarmonyLib 命名空間的各種功能。但最終編譯出的模組DLL**不需**包含 Harmony 程式碼。

舉例來說，About.xml 可能包含以下片段來指定相依性：

```
<modDependencies>
  <li>brrainz.harmony</li> <!-- Harmony 模組的 packageId，確保其在本模組之前載入 -->
</modDependencies>
```

以上 packageId 需與 Harmony 模組發行者提供的一致（例如 Steam 上 Harmony 模組通常為 **brrainz.harmony**）。有了這個設定，遊戲會在載入本模組前先載入 Harmony 模組，從而保證 HarmonyLib 可用。

5.2 在模組初始化時註冊 Patch

RimWorld 在載入每個模組時，會掃描該模組Assemblies資料夾內的 DLL，載入其中的類型。我們有幾種方法可以讓 Harmony 開始作用： - **靜態建構子註冊法**：利用 RimWorld 提供的 `StaticConstructorOnStartup` 屬性。凡是標記了這個屬性的靜態類，當遊戲啟動時（進入主選單前），其靜態建構子會被自動呼叫 [12] 。我們可以在靜態建構子中建立 Harmony 實例並執行 `PatchAll()`。這種方式簡單且常用在輕量模組中。 - **自訂 Mod 類別註冊法**：如果我們的模組需要一個自訂類繼承 Verse.Mod（例如為了創建設置介面），也可以在該類的建構子裡初始化 Harmony。一樣是呼叫 `harmony.PatchAll()` 或針對特定方法呼叫 `Patch()`。此方法需要我們在 About.xml 中指定 `\<assemblies\>` 引入那個類，通常前幾章應已介紹過如何設置自訂 Mod 類別。

本章範例採用第一種方式，使用靜態建構子來自動套用 Patch。步驟如下：

1. 新增一個專門的初始化類，例如 `HarmonyInit`。將 `[StaticConstructorOnStartup]` 標記附加在類別上。
2. 在該類的靜態建構子裡編寫 Harmony 初始化程式碼。
3. 編譯後，放入模組DLL。RimWorld 讀取模組時就會執行它。

範例程式碼：

```
using HarmonyLib;
using Verse;
```

```
[StaticConstructorOnStartup]
static class HarmonyInit {
  static HarmonyInit() {
    var harmony = new Harmony("com.mymod.example"); // 建立Harmony實例，提供唯一ID
    harmony.PatchAll(); // 自動套用當前Assembly中所有帶有HarmonyPatch的類別
    // Log.Message("Harmony patches applied"); // 可選：輸出日誌確認Patch套用
  }
}
```

這段程式完成以下工作： - 創建了一個新的 Harmony 實例，其中傳入的 ID 字串通常採用**逆URL**格式或類似具有唯一性的識別碼。雖然這個ID只要對您自己的模組保持一致即可，但採用獨特ID可以避免極罕見情況下多模組產生衝突（例如意外用了相同ID的Harmony實例會被視為同一個）。 - 呼叫 `PatchAll()`，Harmony 會掃描我們模組DLL內所有 `[HarmonyPatch]` 類別並按定義自動進行Patch。這就是為什麼我們前面撰寫的 `MeleeDamageDouble_PrefixPatch`、`_PostfixPatch`、`_TranspilerPatch` 等類只要在DLL中，Harmony 就能找到並套用它們。**注意：** 預設情況下，`PatchAll()` 會Patch當前 Assembly 中**所有**標記的類，如果我們有些Patch不想立即啟用，可以先不標 `[HarmonyPatch]`，或使用特定條件控制。 - `StaticConstructorOnStartup` 保證了上述程式在適當時機執行。以 RimWorld 而言，這通常在進入主選單時就執行了（也就是說，只要載入了模組，不需要開始遊戲，就已Patch完成）。如果Patch的目標是遊戲載入過程中就會執行的方法，這種提早初始化能確保攔截成功 [26]。

5.3 經驗與注意事項

- **避免重覆 Patch：** 確保 `PatchAll()` 只執行一次。例如不要同時在靜態建構子和Mod類別裡都呼叫，否則Harmony會嘗試重覆套用相同Patch導致警告或錯誤。
- **版本管理：** Harmony 2向下相容Harmony 1的大多數功能，但最好依據遊戲版本採用正確的Harmony版本。RimWorld 1.1以後官方Harmony模組提供的是Harmony 2.x。[27] [28]（大部分模組開發者現在都使用Harmony 2）。若你的模組需要特別依賴Harmony的新功能，請在文檔中說明。
- **錯誤診斷：** 如果發現某Patch沒有生效，可在開發時使用 `Harmony.DEBUG = true`（HarmonyInstance 2.x中可用 `Harmony.DEBUG` 靜態屬性）來讓Harmony生成詳細日誌。執行遊戲後，到 `%APPDATA%/Locallow/Ludeon Studios/RimWorld by Ludeon Studios/` 目錄下尋找Harmony的日誌檔，可了解每個Patch是否成功，以及衝突情況 [29]。
- **保留良好命名：** 為Patch類和方法命名時，採用有意義的名稱（例如類名含目的，如 `MeleeDamageDouble_TranspilerPatch`）有助於日後維護。即使Harmony允許匿名Patch類，清晰的代碼結構仍是專業開發的要求。
- **其他資源載入：** 如果Patch需要配合一些資料（例如配置值、外部檔案），可以利用 Verse.Mod 類別來載入配置再由Patch使用。但這屬於模組架構整合問題，超出Harmony討論範圍。

6. 範例模組的檔案結構與資源配置

在本章示範中，我們已完成一個修改傷害計算的模組代碼。最後，我們將整個模組應有的檔案結構和相關配置總結如下，以便你將其整合成實際可運行的 RimWorld 模組。

假設我們將此模組命名為 "DoubleDamageMod"，其資料夾結構建議如下：

```
DoubleDamageMod/                 <-- 模組根目錄（名稱自定）
├── About/
│   ├── About.xml                <-- 模組資訊檔，在這裡設定名稱、版本、相依性等
│   └── Preview.png              <-- 模組預覽圖（可選）
```

```
├── Assemblies/
│   └── DoubleDamageMod.dll    <-- 編譯後的模組程式集，包含我們撰寫的Harmony Patch程式碼
├── Defs/
│   └──── (可選) ...            <-- Def 定義檔（如需新增ThingDefs等，可以在此放置XML檔）
└──── (其他資料夾)              <-- 例如Textures/、Patches/等，根據模組需要添加
```

重點說明： - **About.xml**：這是模組定義檔，需填入 `<name>`, `<packageId>`, `<supportedVersions>` 等基本信息。此外，為使用 Harmony，應確保包含相依性宣告。例如：

```
<supportedVersions>
 <li>1.4</li>
</supportedVersions>
<modDependencies>
 <li>brrainz.harmony</li>
</modDependencies>
```

如此，遊戲才能知道在載入本模組前，要先載入Harmony模組。缺少這個步驟可能導致玩家如果沒訂閱Harmony模組時，本模組無法正常工作甚至報錯。 - **Assemblies/DoubleDamageMod.dll**：我們所有的 C# 程式碼（Harmony Patch 類，以及 `HarmonyInit` 初始化類）都會被編譯進這個 DLL。編譯時請確保引用了正確版本的 RimWorld API (Assembly-CSharp.dll, Verse.dll) 和 Harmony dll。將編譯出的DLL放在Assemblies資料夾下，RimWorld會自動載入它。 - **Defs/**：本例模組僅修改現有行為，並未增加新的物件或數值定義，因此 Defs 資料夾可以沒有內容或省略。如果你的模組同時還要透過XML增加新武器、新能力等，則需要在此放置對應的 Def 檔案，Def 系統的部分在之前章節已詳細介紹。 - **Patches/**：請注意，不要混淆 Harmony Patch與RimWorld的XML Patch機制。RimWorld模組也允許在 Patches/ 資料夾裡放置XML定義來批量修改原版Def，稱作**Patch Operations**（這在第二、三章或模組基礎部分或許有提及）。那種XML Patch不同於我們這章講的程式碼Patch。若本模組沒有XML Patch需求，此資料夾可省略。

完成以上結構後，重新啟動 RimWorld，在模組列表中應該能看見我們的模組 "DoubleDamageMod"。將它和 Harmony 模組一起啟用（順序上Harmony應該自動在前，若沒有就手動調整），進入遊戲測試近戰戰鬥。若一切正確，所有近戰攻擊造成的傷害都會明顯增加。為驗證，可以開啟開發者模式觀察戰鬥日誌中的傷害值，或自行撰寫一個小測試場景對比有無啟用模組時敵人承受傷害的差異。

最後提醒，在真實開發過程中，要根據遊戲版本及其他Mod可能的存在情況來調整Patch內容。例如，也許未來版本改變了 `CalculateMeleeDamage` 的內部，使我們Transpiler的匹配需要更新；或另一個Mod也試圖更動傷害，可能需要協調（例如提供設定讓玩家選擇倍率，而非寫死2倍）。這些都是進階課題，而本章學到的Harmony技巧將為你處理複雜情境打下基礎。

練習題

1. **理解Prefix跳過原方法：** 說明當一個 Prefix 傳回 false 時，Harmony 將如何處理該方法接下來的執行流程？對於其他同一目標方法的Prefix和Postfix而言，這有何影響？【提示】試著以你自己的話描述 Harmony 如何協調多個 Prefix 的執行，以及為什麼官方建議謹慎使用「跳過原方法」。
2. **選擇合適的Patch類型：** 假設你想修改遊戲中角色進食的行為：如果角色吃的是某種特定食物，將獲得雙倍的飽食度恢復。你會使用 Prefix、Postfix 還是 Transpiler？簡述你的理由和大致實現思路。
3. **撰寫簡單Postfix：** 選擇 RimWorld 原版遊戲中一個你感興趣的小功能，例如「當玩家建造建築時播放聲音」。假設該功能對應一個方法 `PlayBuildingSound()`, 沒有返回值。請設計一個Postfix，在每

次該方法被呼叫後，在遊戲日誌(console)額外輸出一行 `"建築聲音已播放"` 的訊息（使用 `Log.Message`）。寫出Pseudo-code或簡易的C#實現（不需要完整Patch類，只需描述關鍵部分）。

4. **閱讀IL片段：** 以下是一段假想的 CIL 片段：

```
IL_0000: ldarg.1
IL_0001: ldc.i4.s 10
IL_0003: bge.s IL_0008
IL_0005: ldc.i4.0
IL_0006: br.s IL_0009
IL_0008: ldc.i4.1
IL_0009: stloc.0
IL_000A: ldloc.0
IL_000B: ret
```

請嘗試將這段IL還原成高階語言的邏輯描述（不用精確的C#語法，用易懂的方式描述也可以）。並說明如果想用Transpiler把這段邏輯的返回值取反（0變1，1變0），你可能在IL哪裡下手修改？

檢查點驗收項目

- **原理理解：** 清楚說明 Harmony 在運行時修改方法的機制，以及 Prefix、Postfix、Transpiler 三者的執行時機與作用範圍。如果有人問你「為什麼要用Harmony而不是直接改Dll？」，你能給出合理的解釋（例如避免版權問題、維護性、相容性等）。
- **技術應用：** 能獨立編寫一個簡單的 Prefix 和 Postfix Patch，知道如何透過 `__result` 修改返回值、透過參數名稱獲取原方法引數，以及瞭解 Prefix 傳回 true/false 的差別。能理解Transpiler模板的基本結構，知道在CodeInstructions序列中如何插入或刪除指令。
- **模組集成：** 成功將 Harmony Patch 添加到模組的DLL中，並在模組載入時自動套用。驗證方法包括：遊戲啟動日誌沒有因找不到Harmony或Patch衝突而報錯；目標功能表現符合預期（例如本章範例中傷害確實翻倍）。也可以查看 `\Mods\Harmony\About\About.xml` 確認Harmony模組已被列為依賴載入。
- **程式碼結構：** 模組目錄結構清晰正確，About.xml 信息完整。Patch 類和其他輔助類分門別類放置，命名符合C#慣例並能反映用途。整體程式碼能在 Unity 編譯環境下通過編譯，無語法錯誤。
- **進階思考：** 若遇到需要更複雜改動的需求，知道應該評估使用哪種Patch技術最適合，而不會一股腦全部用Transpiler或全部用Prefix。對於潛在的多模組Patch競合問題，有基本的預判和應對思路（例如透過前綴條件判斷、檢查是否有其他Patch已影響結果等）。

完成以上驗收項目後，表示您已掌握 Harmony 前綴/後綴/Transpiler Patch 技術的基礎知識，並能將其應用於 RimWorld　模組開發中更改遊戲行為的實際場景。在下一章中，我們將繼續探討更多進階的模組開發技巧，進一步拓展您的 RimWorld 模組開發能力。祝您在實驗 Harmony Patch 時一切順利！

---

[1] [2] [3] [4] [5] [6] [8] [10] [12] [13] [14] [16] [19] [26] [29] Modding Tutorials/Harmony - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/Harmony

[7] [15] Patching
https://harmony.pardeike.net/articles/patching-prefix.html

[9] [11] [21] [22] [23] [24] [25] Patching
https://harmony.pardeike.net/articles/patching-transpiler.html

[17] [18] [20] Patching
https://harmony.pardeike.net/articles/patching-postfix.html

27 28 Harmony - the full story : r/RimWorld
https://www.reddit.com/r/RimWorld/comments/fbnm45/harmony_the_full_story/