ChatGPT

RimWorld 模組 Harmony Prefix/Postfix Patch 實作深度案例研究

RimWorld 模組開發常透過 **Harmony** 庫來修改遊戲的執行邏輯。Harmony 提供 Prefix（前置）與 Postfix（後置）方法，讓我們在原始方法執行之前或之後插入自訂程式碼，以改寫或擴充原有行為。以下將針對五個不同範疇（AI 行為、角色互動、戰鬥傷害、建築資源、事件系統），分別說明原始遊戲邏輯、使用 Harmony Patch 的動機與實作方法，並提供完整的 Patch 程式碼範例（含中文註解）。同時，每個案例將展示如何結合**模組架構**（如 `ILogicHook<T>` 介面與 `LoggingHookDecorator` 裝飾器）來管理這些 Patch，以及對執行時機、潛在副作用與模組衝突風險的分析。

1. AI 行為與決策：工作分配優化案例

原始遊戲邏輯簡析

在 RimWorld 中，殖民者（Pawn）的**工作分配**主要由其 AI 思考樹中的 `JobGiver_Work` 節點負責。預設邏輯會按照工作優先級順序，掃描每個可執行的工作類型（如打掃、搬運、建造等），一旦找到第一個可行的任務就返回該工作 [1]。這種「優先級優先」的策略有時會導致殖民者忽略距離遠近——例如，他們可能跑到地圖另一端去執行高優先級任務，即使附近有稍低優先級但距離更近的工作可做。**路徑選擇**方面，遊戲使用 `PathFinder` 根據地形和成本計算最佳路徑，但並未在工作選擇階段考慮路程遠近。

Patch 動機與情境

動機： 改善 AI 的工作選擇，使殖民者更傾向於執行距離較近的工作，避免長途奔波造成時間浪費。這類優化可以提高殖民地效率，屬於常見的 QoL（品質改善）模組需求之一。

情境： 假設我們開發一個名為「本地工作優先」的模組。當殖民者找到一項可能的工作時，如果該工作的目標距離過遠，我們希望攔截並過濾掉這項工作，讓 AI 繼續尋找其他候選任務（或乾脆讓角色暫時不接遠處工作）。由於原始 `JobGiver_Work` 只返回第一個發現的任務，我們需要透過 **Harmony Postfix** 來檢查它返回的結果，並在必要時進行修改。

Harmony Postfix 改寫實作

我們選擇對 `RimWorld.JobGiver_Work` 類的 **TryGiveJob** 方法施加 Postfix Patch。此方法返回一個 `Job` 物件（如果找到工作，否則為 null）。透過 Postfix，我們可以取得原始結果（`__result`）並對其進行判斷與二次處理 [2]。以下是完整的 Harmony Patch 程式碼實例：

```
using HarmonyLib;
using RimWorld;
using Verse;
using Verse.AI;

[HarmonyPatch(typeof(JobGiver_Work), "TryGiveJob")]
public static class LocalWorkPriority_Patch
{
```

```
  // 後置方法：在原始 TryGiveJob 執行完畢後觸發
  static void Postfix(Pawn pawn, ref Job __result)
  {
    // 若原本找到的工作任務不為空，檢查其目標距離
    if (__result != null)
    {
      // 取得該工作的目標位置（對於以物件為目標的工作）
      IntVec3 targetPos = __result.targetA.Cell;
      // 簡單計算與殖民者之間的距離（曼哈頓距離或實際路徑長度）
      float distance = pawn.Position.DistanceTo(targetPos);

      // 定義過遠的距離閾值，例如超過100格
      if (distance > 100f)
      {
        // 紀錄日志，說明因距離過遠而放棄該工作
        Log.Message($"[本地工作優先] {pawn.Name} 放棄遠距離工作：{__result.def.defName}，距離
{distance}");
        // 將結果工作置空，表示不接受該工作
        __result = null;
      }
    }
  }
}
```

上述 Prefix 並未使用 (我們讓原方法先執行，再在 Postfix 進行判斷)，這樣可以最大程度保持與原邏輯的相容性 [3]。在 Postfix 中，我們取得 `Pawn`（執行該方法的殖民者）和返回的 `Job` 結果。當發現任務距離超過預設閾值時，將 `__result` 設為 `null` 來**修改原方法的返回值** [2]。這會使殖民者暫不執行該遠距離工作，轉而在下個 AI Tick 再次嘗試尋找其他工作。

> **註：** 此處簡化使用歐幾里得距離或曼哈頓距離作判斷，實際可考慮利用遊戲內路徑尋路成本來判定更精確的路程遠近。

結合模組架構的擴充應用

為了讓這個過濾邏輯更具彈性並易於管理，我們可以將其封裝到自定的模組架構中。例如，定義一個泛型介面 `ILogicHook<T>` 來處理特定邏輯，我們為工作分配創建一個實作：

```
// 自定義介面，用於封裝邏輯處理
public interface ILogicHook<T>
{
  T Process(T input, Pawn pawn);
}

// 本地工作過濾 Hook：實作工作過濾邏輯
public class LocalWorkHook : ILogicHook<Job>
{
  public Job Process(Job input, Pawn pawn)
  {
    if (input != null)
    {
```

```
            float dist = pawn.Position.DistanceTo(input.targetA.Cell);
            if (dist > 100f)
            {
                // 如果太遠，返回 null 表示過濾掉該工作
                Log.Message($"[本地工作Hook] {pawn.Name} 的任務過遠（{dist}格），過濾掉。");
                return null;
            }
        }
        return input;
    }
}
```

接著，我們可以使用 **裝飾器**模式為這個 Hook 添加日誌紀錄等功能。例如，`LoggingHookDecorator` 實作 `ILogicHook<T>`，內部持有實際的 `ILogicHook`，在執行前後進行額外記錄：

```
// 紀錄功能的裝飾Hook
public class LoggingHookDecorator<T> : ILogicHook<T>
{
    private readonly ILogicHook<T> innerHook;
    public LoggingHookDecorator(ILogicHook<T> hook) => innerHook = hook;
    public T Process(T input, Pawn pawn)
    {
        Log.Message($"[Logging] 準備執行 Hook 邏輯: {innerHook.GetType().Name}");
        T result = innerHook.Process(input, pawn);
        Log.Message($"[Logging] Hook 執行完畢，結果: {(result == null ? "null(被過濾)" : "保留任務")}");
        return result;
    }
}
```

在 Harmony Patch 中，我們將實際邏輯委派給這些 Hook：

```
static void Postfix(Pawn pawn, ref Job __result)
{
    // 建立 Hook 實例（可考慮透過依賴注入或單例管理）
    ILogicHook<Job> hook = new LoggingHookDecorator<Job>(new LocalWorkHook());
    __result = hook.Process(__result, pawn);
}
```

如此一來，**邏輯與補丁解耦**：`LocalWorkHook` 專注於決策邏輯，Harmony Patch 僅負責調用 Hook 並將結果回寫。開發者可以輕鬆替換 Hook 實現或在裝飾器中增進功能，而不必修改 Harmony Patch 本身。

執行時機與衝突風險分析

執行時機： `JobGiver_Work.TryGiveJob` 會在 AI 每次尋找新任務時被調用，這可能是每秒多次（每個閒置的 Pawn 每 250 ticks 嘗試尋找工作）。因此本 Patch 的程式碼也將非常頻繁地執行。所幸我們的邏輯相對輕量（簡單距離計算和比較），對效能影響有限。但仍應避免在此處進行過於耗時的運算，以免累積造成卡頓。

可能副作用： 殖民者因過濾掉遠距工作，**短期內可能無所事事**（因為原本找到的任務被置空）。如果沒有其他更近的工作，他們可能會一段時間閒置。這是行為上的改變，需要確保符合玩家預期。例如玩家可能需要手動介入指派遠處工作。如果閾值設定不當，也可能導致某些重要任務被長期忽略。

與其他模組的潛在衝突： 我們的 Postfix 修改了 `TryGiveJob` 的返回值。若另有模組也對此方法做 Patch： - **與其他 Postfix 的相容性：** Postfix 根據 Harmony 設計會全部執行，執行順序則由載入順序決定。我們的 Postfix 不會跳過原方法，因此應與**絕大多數**其他 Postfix 相容 [3]。但是，如果另一個 Postfix 也試圖修改 `__result`（例如將某些工作強制置換），則最後執行的 Postfix 會覆蓋之前的結果。我們應盡量避免與知名模組的衝突，或提供**可配置選項**讓玩家調整此功能。 - **與 Prefix 的相容性：** 假如有其他模組對 `TryGiveJob` 加入了 Prefix，且該 Prefix **返回 false** 跳過了原始邏輯 [4]。那麼我們的 Postfix 仍然會執行，但此時 `__result` 可能是另一模組在 Prefix 中預先設置好的值，或者保持為 null。如果其他模組 Prefix 完全取代了工作分配，我們的過濾邏輯可能無用武之地，甚至在不預期的時間點發生。因此，在說明文件中應提醒可能的相容性問題。特別是 Harmony 的作者也指出：「使用 Prefix 並返回 false（跳過原方法）的做法相較其他方案更具相容性風險」 [5]；我們的模組雖不採用 destructive prefix，但仍需注意與這類模組並存時的行為。 - **與路徑相關模組的互動：** 有些模組可能改變了路徑尋找或加入新計算（例如讓 AI 避開陷阱、火災等），我們的過濾邏輯僅基於距離，未考慮路徑危險度。如果兩者結合，可能出現 AI 因我們過濾機制放棄任務，但實際上另一模組已解決了遠距離問題的狀況。這種情況下，需要協調兩模組的配置，或允許玩家關閉其中一方的功能。

總的來說，此案例展示如何以 Postfix 攔截並修改 AI 的工作決策，達到優化行為的目的，同時也強調了在高頻方法上進行Patch時對效能與相容性的考量。開發者應平衡修改收益與上述風險，並做好模組說明與配置選項供使用者調整。

2. 角色互動：社交與關係行為調整案例

原始遊戲邏輯簡析

角色間的**社交互動**透過 `InteractionWorker` 系列類別來實現。以「浪漫表白」（Romance Attempt）為例，遊戲有一個 `InteractionWorker_RomanceAttempt` 類，其內部定義了當兩名角色進行浪漫互動時，如何決定發生機率及結果。通常決策基於雙方的關係狀態（是否已有伴侶）、好感度、性取向以及隨機機率等。舉例而言，遊戲中如果角色A對角色B有浪漫傾向，`RandomSelectionWeight` 會計算一個權重來表示A嘗試浪漫表白的機率；一旦互動發生，`Interacted` 或相關方法再根據雙方好感度決定成功或失敗並更新關係。

另一個例子，「交配」（繁殖行為）在動物間自動進行，而人類角色的「做愛（lovin’）行為」由 `JobDriver_Lovin` 處理，會在伴侶就寢時隨機觸發，使雙方獲得情緒增益。這些行為的共同點是：**有特定的觸發條件與機率**，遊戲原始邏輯決定了何時發生以及帶來何種效果。

Patch 動機與情境

動機： 提供更合理或可控的角色互動機制。例如： - 避免不恰當的浪漫嘗試：如果角色已經有穩定戀愛關係，我們希望阻止他們頻繁地對他人表白，符合常理。 - 增強特定特質對互動的影響：讓擁有「外貌協會」（美麗、美貌）特質的角色在社交中更具優勢（更高成功率），或者讓「害羞」特質降低他們主動表白的機會。 - 控制醫療或交配行為：例如限定醫療行為只能由特定角色進行，或調整交配的頻率與後果（如增加懷孕機率等）。

情境： 我們以**浪漫表白**為例，開發一個「理性戀愛」模組。此模組旨在讓角色的浪漫行為更貼近人情常理：如果角色已有愛人且對現任伴侶的好感度很高，那麼不會去追求其他人；同時，讓具有外貌特質的角色在表白時機率提高。為達成這些效果，可使用 Harmony Prefix 來攔截表白機率的計算，根據自訂規則調整後再決定是否讓原邏輯繼續。

Harmony Prefix 改寫實作

我們對 `InteractionWorker_RomanceAttempt` 類的 `RandomSelectionWeight` 方法實施 Prefix Patch。這個方法負責計算角色A對角色B進行浪漫互動的相對機率，返回一個浮點數權重。透過 Prefix，我們可以在**原始計算之前**插入自己的邏輯，必要時跳過原計算或修改結果。 6

以下是實作範例：

```
using HarmonyLib;
using RimWorld;
using Verse;
using UnityEngine;  // 可能用於隨機函數

[HarmonyPatch(typeof(InteractionWorker_RomanceAttempt), "RandomSelectionWeight")]
public static class RationalRomanceChance_Patch
{
  // 前置方法：在計算浪漫嘗試機率前執行
  static bool Prefix(Pawn initiator, Pawn recipient, ref float __result)
  {
    // 1. 如果 initiator 已有愛人且對愛人的意見值很高，則不嘗試新戀情
    if (LovePartnerRelationUtility.HasAnyLovePartner(initiator))
    {
      Pawn currentLover = LovePartnerRelationUtility.ExistingLovePartner(initiator);
      if (currentLover != null)
      {
        int opinion = initiator.relations.OpinionOf(currentLover);
        if (opinion >= 25)  // 好感度很高的門檻，例如 +25
        {
          __result = 0f;  // 權重為0，表示不會觸發
          // 記錄日志
          Log.Message($"[理性戀愛] {initiator.Name} 對現任伴侶好感{opinion}，不會追求
{recipient.Name}");
          return false;  // 跳過原始方法，不再計算任何浪漫機率 4
        }
      }
    }
    // 2. 若 initiator 有 "Beautiful"（美貌）特質，增加表白成功率（乘算因子）
    bool hasBeauty = initiator.story?.traits.HasTrait(TraitDefOf.Beautiful) ?? false;
    if (hasBeauty)
    {
      // 調用原始方法計算基礎機率
      // 通過反射或 AccessTools 調用原方法，由於這裡是 Prefix，我們需要自行計算原始值
      float baseWeight = 0f;
      try
      {
        // 透過 AccessTools 調用原本的 RandomSelectionWeight 方法
        baseWeight = (float) AccessTools.Method(typeof(InteractionWorker_RomanceAttempt),
"RandomSelectionWeight")
                        .Invoke(null, new object[] { initiator, recipient });
      }
```

```
        catch { /* 處理反射可能的錯誤 */ }

        // 增加 50% 機率作為美貌增益（舉例）
        __result = baseWeight * 1.5f;
        Log.Message($"[理性戀愛] {initiator.Name} 擁有美貌特質，浪漫機率從 {baseWeight:F2} 提升至 {__result:F2}");
        return false; // 已經自行計算結果，跳過原始計算
      }

      // 3. 其他情況下，不改變機率，允許執行原始方法
      return true;
    }
  }
```

在這段 Prefix 中，我們做了兩級處理： 1. **阻止不理性的表白：** 對於已有伴侶且感情深厚的角色，直接將機率權重設為0並跳過原始計算，確保他們不會去追求他人。 [6] 2. **強化美貌特質影響：** 如果施動者有「美貌」特質，我們希望提高其表白成功率。這裡演示了一種手法：**在 Prefix 中調用原方法的邏輯**來取得基礎值，然後再乘上增益係數。由於 Harmony Prefix 無法輕易取得原結果（原方法尚未執行），我們使用了 `AccessTools` 反射來模擬執行原邏輯（**注意：**這種作法僅作示範，實際中可能需要更嚴謹處理，以避免遞迴調用原 Patch）。設定新的 `__result` 後返回 false，跳過原方法執行，以我們計算的值作為最終結果。

除了上述特殊情況，其餘狀況返回 true，表示不干預，允許遊戲執行原始邏輯計算浪漫互動機率。

結合模組架構的擴充應用

如同案例1，我們可以將社交行為的邏輯調整封裝為 Hook，方便擴充與管理。例如定義 `ILogicHook<InteractionInputs>` 來處理互動發生前的判斷，其中 `InteractionInputs` 可以是一個包含 initiator 和 recipient 的結構。我們實作兩個 Hook： - `RelationshipGuardHook`：檢查施動者現有戀情，決定是否禁止新的浪漫互動。 - `BeautyBonusHook`：檢查特質並調整機率。

我們還可以將結果定義為一個結構 `RomanceAttemptResult` 包含計算出的權重和是否應跳過原方法。簡化起見，這裡概念性展示：

```
public struct InteractionInputs { public Pawn initiator; public Pawn recipient; }

public class RelationshipGuardHook : ILogicHook<InteractionInputs>
{
  public bool ShouldSkip { get; private set; } = false;
  public float weightResult = -1f;
  public InteractionInputs Process(InteractionInputs input, Pawn context = null)
  {
    Pawn initiator = input.initiator;
    Pawn lover = LovePartnerRelationUtility.ExistingLovePartner(initiator);
    if (lover != null)
    {
      int opinion = initiator.relations.OpinionOf(lover);
      if (opinion >= 25)
      {
        // 設定為跳過，並給結果0權重
```

```
            ShouldSkip = true;
            weightResult = 0f;
            Log.Message($"[Hook] 阻止 {initiator.Name} 對他人表白，維持現任關係");
          }
        }
        return input;
    }
}

public class BeautyBonusHook : ILogicHook<InteractionInputs>
{
    public float AdjustWeight(float baseWeight, Pawn initiator)
    {
        if (initiator.story?.traits.HasTrait(TraitDefOf.Beautiful) ?? false)
        {
            float newWeight = baseWeight * 1.5f;
            Log.Message($"[Hook] 美貌特質加成權重: {baseWeight:F2} -> {newWeight:F2}");
            return newWeight;
        }
        return baseWeight;
    }
    public InteractionInputs Process(InteractionInputs input, Pawn context = null)
    {
        // 此 Hook 不直接修改輸入，只提供輔助方法調整權重
        return input;
    }
}
```

然後在 Prefix Patch 中使用這些 Hook：

```
static bool Prefix(Pawn initiator, Pawn recipient, ref float __result)
{
    var input = new InteractionInputs { initiator = initiator, recipient = recipient };
    var guardHook = new RelationshipGuardHook();
    guardHook.Process(input);
    if (guardHook.ShouldSkip)
    {
        __result = guardHook.weightResult;
        return false;
    }
    // 未被阻止則計算原始權重
    float baseWeight = CalculateBaseRomanceWeight(initiator, recipient);
    // 應用美貌加成
    baseWeight = new BeautyBonusHook().AdjustWeight(baseWeight, initiator);
    __result = baseWeight;
    return false;
}
```

上述 `CalculateBaseRomanceWeight` 代表透過反射或其他方式取得原始計算結果的函式。可以看到，**Hook 的運用讓每個邏輯點各司其職**：`RelationshipGuardHook` 專注於關係條件判斷，`BeautyBonusHook` 專注於特質影響。透過這樣的架構，我們還可以方便地加入更多 Hook（例如檢查年齡差距或信仰禁忌等）來豐富行為，同時可用 `LoggingHookDecorator` 對每個 Hook 執行情況進行統一日誌管理。

執行時機與衝突風險分析

執行時機： `InteractionWorker_RomanceAttempt.RandomSelectionWeight` 在遊戲每次評估是否發起浪漫互動時呼叫。這通常發生於社交空閒時刻，頻率不如工作尋找那樣高，但在有多名單身角色時仍可能頻繁觸發。我們採用 Prefix 並在部分情況下跳過原始方法，確保**及早攔截**不想要的行為發生。

可能副作用： - 經此修改後，角色之間的關係發展可能**較慢或不同**於原版。玩家可能注意到角色不再輕易劈腿或亂搭關係，這符合模組預期的理性行為，但也可能讓劇情變得略微平淡。需留意平衡性，確保有趣性不被過度抑制。 - Prefix 在某些條件下使用了反射調用原方法邏輯，此做法可能導致**性能開銷**（儘管浪漫權重計算本身不繁重，但反射頻繁調用不是最佳實踐）。實際實作時，應考慮以更優雅方式取得原始值，例如透過 Harmony 的 `__result` 參考在 Postfix 中調整，或者使用 Transpiler 插入係數。但本例重點在演示 Prefix 操作，因此採用了直觀方式。 - Prefix **跳過原方法**在多條件下發生：我們返回 false 的情況包括感情很好和存在美貌特質。由於我們每次 Prefix 最終都 return false（無論是否修改），實際上**完全取代**了原始計算邏輯。這意味著即使角色不滿足我們列出的特殊條件，我們也通過 `CalculateBaseRomanceWeight` 取得基礎值再返回，未執行原方法本身。這麼做雖然方便我們掌控全流程，但也**提高了與原版差異**的覆蓋範圍。這是一種較具侵入性的修改策略。

與其他模組的潛在衝突： - 其他模組若也修改了浪漫互動的邏輯（如 “Less Stupid Romance Attempt” 等模組），可能採取類似手段禁止某些情況的表白或改變機率。我們的 Prefix **以跳過原方法的方式運作**，因此**兼容性較弱** [5] ——一旦我們的 Prefix 執行並返回 false，後續其他 Prefix (若有) 將不會執行，而 Postfix 仍會執行但只能看到我們設定的結果。這可能導致其他模組的調整失效。例如，另一模組也想降低某些情況表白機率，如果它是 Postfix 只調整結果，那在我們跳過原方法的情況下仍可拿到我們給的結果進行調整（尚算相容）；但若它也是 Prefix，則執行順序先後將決定誰生效。為降低衝突，可以與其他作者協調或在Mod描述中說明衝突情形。 - 本例的反射調用原方法如果與**Harmony 堆疊**產生混用，可能造成不可預期行為（例如調用的實際是已被其他Patch修改過的方法體）。這是另一潛在衝突點。幸運的是，大部分模組傾向於使用 Postfix 來改變結果以保持相容性 [3]。我們的做法相對激進，需要特別標明給進階用戶注意。 - 如果有模組**全面替換了角色社交系統**（例如某些大幅改革關係系統的Mod），我們的Patch可能變得無效甚至出錯——例如對方可能重寫了 `InteractionWorker_RomanceAttempt` 的類或方法，使我們的 Harmony 找不到或Patch錯誤。為此，我們需要在模組發佈說明中列出已知不相容的Mod清單，或採取**檢查防護**，偵測相關類存在再注入Patch，以防止崩潰。

3. 戰鬥與傷害處理：暴擊與傷害修正案例

原始遊戲邏輯簡析

RimWorld 的戰鬥系統中，**命中判定與傷害處理**主要透過 `Verb`（攻擊動作）和 `DamageWorker`（傷害處理器）兩類系統協同完成。流程大致如下： - 攻擊動作（射擊或近戰）由對應的 `Verb` 類執行，例如遠程武器常用 `Verb_LaunchProjectile`，近戰武器用 `Verb_MeleeAttack` 或其子類。這些 Verb 的 `TryCastShot()` 方法會決定攻擊是否命中目標 [7]。對於射擊，系統根據射手技能、武器精度、距離和目標大小等計算命中機率，並可能產生偏移（例如射偏機制）；對於近戰，則考慮攻擊者近戰技能與目標近戰躲避能力等。 - 一旦判定命中，系統創建 `DamageInfo` 結構，裡面包含傷害數值、傷害類型（如穿刺、鈍擊等 `DamageDef`）、攻擊來源（攻擊者Pawn、武器等）以及命中部位等資訊。然後調用被攻擊 `Thing`（通常是 `Pawn`）的 `TakeDamage(DamageInfo)` 方法。 - **傷害處理**：`Pawn.TakeDamage` 內會根據

`DamageInfo` 調用對應的 `DamageWorker`（每種 `DamageDef` 對應一個DamageWorker類別），典型如 `DamageWorker_AddInjury` 處理一般傷害。DamageWorker 決定實際傷害值對各身體部位的分配、是否造成狀態（斷肢、出血等）並返回 `DamageResult`。整個過程自動且封閉，遊戲並未內建所謂「暴擊」機制——傷害大小只由武器和隨機浮動決定。

Patch 動機與情境

動機： 引入**暴擊（Critical Hit）**概念，或其他自訂的戰鬥計算調整，使戰鬥更為多樣化或符合某些模組的世界觀。例如： - 暴擊傷害：攻擊有一定機率造成額外傷害（例如雙倍），模擬致命一擊或要害命中。 - 特殊效果觸發：例如子彈有機率附帶燃燒效果、近戰攻擊有機率擊倒或震懾敵人等。 - 傷害調整：根據攻擊者或目標的屬性（如技能等級、特質）對傷害做動態修正。

情境： 開發一個「暴擊系統」模組，讓每次攻擊有 20% 機率造成雙倍傷害，並在暴擊時給予特別的戰鬥日誌提示或飛濺字體。這裡我們著重於**提高傷害**部分的實現。由於傷害的最終計算發生在 `Pawn.TakeDamage` 階段（此時已確定命中並給出傷害值），我們可以透過 Harmony Prefix 攔截 `TakeDamage`，在傷害實施前修改 `DamageInfo` 的數值。如果我們能在這一刻決定一次攻擊是否暴擊，並放大傷害值，那後續的原始傷害處理流程即可順理成章地造成更重的傷害。

Harmony Prefix 改寫實作

我們對 `Verse.Thing`（或其子類 `Pawn`）的 `TakeDamage` 方法施加 Prefix Patch。`TakeDamage` 接受一個 `DamageInfo` 結構參數，我們可利用 Prefix 來改變此參數內的數值。Harmony Prefix 方法允許我們以 `ref` 方式取得參數，從而直接修改它 [8]。

以下是程式碼範例：

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

[HarmonyPatch(typeof(Pawn), "TakeDamage")]
public static class CriticalHit_Patch
{
  // 前置方法：在 Pawn 接收傷害時觸發
  static void Prefix(ref DamageInfo dinfo, Pawn __instance)
  {
    // 僅針對 Pawn（殖民者或生物）且有攻擊者的情況
    if (__instance != null && dinfo.Instigator is Pawn attacker)
    {
      // 決定是否暴擊：例如20%機率
      if (Rand.Chance(0.20f))
      {
        // 暴擊，將傷害值加倍
        float original = dinfo.Amount;
        dinfo.SetAmount(original * 2);
        // 可選：給予暴擊標記或特效（這裡簡單以日誌表示）

Log.Message($"[暴擊!] {attacker.Name} 對 {__instance.Name} 造成致命一擊，傷害 {original} ->
{dinfo.Amount}");
```

```
            // TODO: 這裡可以加入戰鬥訊息或飛字，例如:
            // MoteMaker.ThrowText(__instance.DrawPos, __instance.Map, "暴擊!",
UnityEngine.Color.red);
        }
    }
  }
}
```

在這個 Prefix 中，我們做了以下處理： - 利用 `ref DamageInfo dinfo` 參數，我們可以直接訪問並修改即將套用的傷害資訊。 [8] - 判定觸發條件：只有當攻擊者存在且是 Pawn 時才考慮暴擊（避免環境傷害或無名來源觸發不必要的計算）。 - `Rand.Chance(0.20f)` 給出20%機率判定暴擊。若觸發，使用 `dinfo.SetAmount()` 方法將傷害值加倍。這會影響後續原始 `TakeDamage` 的處理，使目標受到更高的傷害。 - 加入一行日誌輸出，說明暴擊事件。我們還示範了可以在這裡引發簡單的視覺效果（如 `MoteMaker.ThrowText` 丟出紅色「暴擊！」字樣），強調暴擊發生。

由於我們沒有在 Prefix **return false**，因此**不會跳過原始傷害處理**。原方法將在我們修改過 `dinfo` 後正常執行 [4] ，所以傷害的應用過程（減少生命值、造成傷疤等）一切如常，只是數值變大了。

結合模組架構的擴充應用

像這種戰鬥傷害調整邏輯，也適合封裝進模組的 Hook 系統中。一種做法是定義 `ILogicHook<DamageInfo>` 介面，專門處理傷害信息的過濾或變更。我們可以將暴擊邏輯寫成一個 Hook，甚至分多個以便擴充： - `CriticalHitHook` ：負責決定是否暴擊以及如何調整傷害值。 - 未來還可加入例如 `FirearmJammingHook` （槍械卡彈，不造成傷害）或 `ArmorPenetrationHook` （根據攻擊者武器穿甲值調整傷害）。

以下展示 `CriticalHitHook` 簡化實現：

```
public class CriticalHitHook : ILogicHook<DamageInfo>
{
  private readonly float critChance;
  private readonly float critMultiplier;
  public CriticalHitHook(float chance = 0.2f, float multiplier = 2f)
  {
    critChance = chance;
    critMultiplier = multiplier;
  }
  public DamageInfo Process(DamageInfo dinfo, Pawn attacker)
  {
    if (attacker != null && Rand.Chance(critChance))
    {
      float orig = dinfo.Amount;
      dinfo.SetAmount(orig * critMultiplier);
      Log.Message($"[Hook] {attacker.Name} 暴擊! 傷害 {orig}->{dinfo.Amount}");
      // 可在這裡觸發額外效果，例如標記 dinfo 或通知 UI
    }
    return dinfo;
```

```
    }
}
```

將其與 Logging 裝飾結合：

```
ILogicHook<DamageInfo> hook = new LoggingHookDecorator<DamageInfo>(new CriticalHitHook());
```

在 Prefix Patch 裡使用：

```
static void Prefix(ref DamageInfo dinfo)
{
    Pawn attacker = dinfo.Instigator as Pawn;
    if (attacker != null)
    {
        // 運用 Hook 進行傷害處理，可擴充額外邏輯
        dinfo = hook.Process(dinfo, attacker);
    }
}
```

如此架構下，**增減暴擊概率或倍率**變得簡單，只需調整 Hook 實例參數或替換不同實現；新增其他戰鬥效果也只要實作新的 Hook 並串接。`LoggingHookDecorator` 能紀錄每次傷害Hook的輸入輸出值，在調試平衡性時尤其有用。

執行時機與衝突風險分析

執行時機： `Pawn.TakeDamage` 在每次傷害作用於角色時觸發，包括戰鬥攻擊及其他傷害來源（飢餓、環境傷害等）。戰鬥中這一方法被頻繁呼叫（連發武器每發子彈、近戰每一擊都會各調用一次）。我們的 Prefix 每次都要進行隨機判定與可能的乘法運算，這些操作**開銷很小**，因此對效能影響可忽略不計。然而要注意，如果有其他模組也在此進行大量計算，累積效果可能需要關注。

可能副作用： - **戰鬥平衡改變：** 引入暴擊意味著傷害輸出有波動增大，可能縮短戰鬥時間、增高死亡率。玩家需要調整戰術適應，這符合模組預期（增添隨機性和刺激感），但要小心不要讓暴擊率過高導致遊戲失衡。在模組設置中提供調整暴擊幾率和倍數的選項會是友好的做法。 - **事件影響：** `TakeDamage` 也涵蓋非戰鬥傷害。如果不加區分地套用暴擊，可能出現奇怪情況，例如角色因飢餓掉血時「暴擊把自己餓死」這種不合理結果。因此我們篩選了 `Instigator` 必須是 Pawn 才處理，以避免環境傷害、掉落傷害也暴擊。此外，某些特殊來源（比如陷阱、爆炸裝置）可能 Instigator 不是 Pawn 而是 Thing（如建築物），這些我們就不予暴擊處理，以保持直觀合理性。 - **戰鬥日誌與UI：** 我們增加了日誌與潛在飛字。如果未妥善控制，可能導致資訊過載（每次暴擊都Log.Message可能刷屏）。應考慮使用更合適的反饋機制，如戰鬥日志(CombatLog)或浮動文字，以美觀且不干擾玩家體驗的方式提示暴擊。

與其他模組的潛在衝突： - **傷害數值調整類模組：** 如果另一模組也修改了 `TakeDamage` 或相關流程，例如有模組降低所有傷害以延長戰鬥、或者按特定公式重新計算傷害。我們的Prefix在原始傷害值基礎上乘2，另一模組可能在我們之前或之後又改變數值。Harmony 預設多個 Prefix 會依序執行：若沒有先後強制排序，載入順序決定誰先誰後。如果我們的 Patch 先執行把傷害翻倍，後執行的模組可能再把傷害減半（抵銷我們效果）或做其他調整；反之亦然。這類**數學衝突**有時難以察覺，因此需要與社群協作，或提供可關閉暴擊功能以讓玩家在衝突時取舍。 - **替換攻擊系統的模組：** 某些大型戰鬥改裝Mod可能完全繞過Vanilla的傷害流程，例如自己處理命中和傷害再直接減生命值，而可能不走標準的 `Pawn.TakeDamage`。此時我們的 Patch將不起作用（因為目標方法沒被調用）。雖然不會直接衝突，但等於模組無效。我們應在文檔中註明不相容情形或嘗試兼容（例

如對特定Mod做檢測，用不同Patch機制）。 - **多次傷害Patch重入**：理論上，Prefix 修改 `dinfo` 不影響之後其他 Prefix 能否看到 Instigator 等資訊。但若其他 Prefix對 `dinfo` 也做 `SetAmount` 或甚至更改 `Instigator` 欄位，會導致混亂。為減少問題，各模組作者應遵守一定約定，例如**僅修改自己關心的欄位**。我們的實作僅變更 `Amount`，應盡量避免碰觸其他屬性，以降低互斥風險。

總結來說，此戰鬥案例透過 Prefix 操作展示了如何實現一個簡單但影響顯著的機制（暴擊傷害）。關鍵在於正確選取 Patch 切入點（`TakeDamage`）、小心使用 `ref` 修改參數，以及充分考慮這種底層修改對整體遊戲性的連鎖影響。

4. 建築與資源管理：自動化與維護案例

原始遊戲邏輯簡析

在 RimWorld 中，**建築物**通常具有各種組件（`Comp`）**管理資源**和功能，例如： - `CompPowerTrader`：用於需要電力的建築，包含是否連通電網的狀態、耗電率等邏輯。 - `CompRefuelable`：用於需要燃料（木材、化學燃料等）的建築，管理燃料消耗與補充。 - `CompBreakdownable`：表示建築會隨機故障，需要零件修理。 - `CompHydro` 等（模組中可能加入，用於水力或其他自訂資源）。

這些 Comp 通常每個遊戲刻（`Tick`）都會執行，例如 `CompPowerTrader.CompTick()` 每秒檢查一次電力供應狀況。建築物本身的 `Tick()` 也會調用各個組件的 `CompTick()`。

資源管理體現在比如：電力網每幾秒平衡供耗、燃料發電機每秒消耗燃料單位、蓄電池充放電等等。**自動化**則包含：建築物在特定條件下自動開關或執行行為，如陽光燈天亮時自動關閉、省電；或使用者想要的自動生產、閒置時關機等功能。

Patch 動機與情境

動機： 增強建築物在資源管理和自動化方面的智能。例如： - **自動開關設備**：當電力不足時自動關閉非必要設備，或當有多餘電力時自動開啟某些裝置以提高效率（如夜間自動關燈）。 - **維護提醒或自動維修**：當建築快沒燃料或即將故障時提醒玩家，甚至自動安排補給/維修（若資源足夠）。 - **資源統計與優化**：即時監控電池電量、水塔水量（如果有水模組），在達到臨界值時觸發一些行動，如關閉輸出、防止浪費或危險。

情境： 以**電池管理**為例，設計一個「電力過載保護」模組。當蓄電池充滿電且繼續過度充電（在遊戲中，滿電後再蓄能其實會溢出浪費或造成隱患），我們希望模組能偵測到並採取行動，例如： - 發出警告通知玩家電池過載。 - 自動切斷發電機的輸入（模擬過載斷路，防止浪費或爆炸）。 - 甚至像某些擴充模組設定，讓電池過載時有幾率起火爆炸作為懲罰。

為此，我們可在 `CompPowerBattery` 的運行中檢查電量狀態並注入我們的邏輯。

Harmony Postfix 改寫實作

我們對 `CompPowerBattery.CompTick` 方法加入 Postfix Patch。原方法每 tick 更新電池的充放電量，本身不會對滿電狀態特別處理，只是電滿了就不再增加。透過 Postfix，我們可在每次 tick 結束後檢查電量狀況並實施額外效果。

```
using HarmonyLib;
using RimWorld;
```

```
[HarmonyPatch(typeof(CompPowerBattery), "CompTick")]
public static class BatteryOvercharge_Patch
{
  // 後置方法：在電池每個刻更新後執行
  static void Postfix(CompPowerBattery __instance)
  {
    // 取得電池當前儲存的能量和上限
    float stored = __instance.StoredEnergy;
    float max = __instance.Props.storedEnergyMax;
    // 設定過載臨界比例，例如 0.95（95%）以上算接近滿載
    if (stored >= max * 0.95f)
    {
      // 尚未通知過的情況下發出一次警告（避免每tick狂刷訊息）
      if (!__instance.parent.IsBrokenDown()) // 確認電池沒有故障
      {
        // 發出信件通知玩家
        Find.LetterStack.ReceiveLetter(
          "蓄電池過載臨界",
          $"{__instance.parent.Label} 電量已達 {stored/max:P0}，請注意用電平衡避免損壞。",
          RimWorld.LetterDefOf.NegativeEvent,
          __instance.parent
        );
      }
      // 如果電量超過上限（理論上不會，除非其他mod修改），強制將能量調整為上限
      if (stored > max) __instance.SetStoredEnergyPct(1f);
    }
  }
}
```

在這段 Postfix 中： - `__instance` 參數代表當前執行 `CompTick` 的電池組件。我們取得它的 `StoredEnergy`（當前能量）和 `Props.storedEnergyMax`（最大能量容量）。 - 判斷條件為當前能量達到 95%以上容量。我們用這個作為**過載臨界點**（可調參數）。 - 如果條件滿足，且我們假定電池目前未處於故障狀態，我們執行一次玩家通知。使用 `LetterStack.ReceiveLetter` 發送一封警報信，內容提示哪個電池接近滿溢【注：這裡使用了 RimWorld 的訊息系統，選擇了一個負面事件類型的信件來警示】。 - 為以防萬一，我們也處理了極端情況：如果 `stored` 超過 `max`（可能是其他模組讓電池能超載），我們將電池能量百分比強制設為100%（避免能量值無限制增長）。

透過這個 Patch，玩家在遊戲中會在電池快滿時收到通知，有機會關閉一些發電設備或增加耗電設施來避免能源浪費或危險。

結合模組架構的擴充應用

在模組架構中，我們可以抽象出一個通用的**資源監控 Hook**。例如定義 `ILogicHook<Comp>` 或專門的 `ILogicHook<CompPowerBattery>` 來封裝對資源臨界的檢測和響應。我們將上面的邏輯封裝為一個 Hook：

```
public class BatteryOverchargeHook : ILogicHook<CompPowerBattery>
{
  private bool hasWarned = false;
```

```
  private readonly float threshold;
  public BatteryOverchargeHook(float thresholdPct = 0.95f)
  {
    threshold = thresholdPct;
  }
  public CompPowerBattery Process(CompPowerBattery comp, Pawn pawnContext = null)
  {
    float stored = comp.StoredEnergy;
    float max = comp.Props.storedEnergyMax;
    if (stored >= max * threshold && !hasWarned)
    {
      hasWarned = true; // 確保只警告一次
      string label = comp.parent.LabelCap ?? "電池";
      Messages.Message($"{label} 電量過載臨界({stored/max:P0})！", comp.parent,
MessageTypeDefOf.NegativeEvent);
    }
    // 當電力下降到較低水平，可以重置警告狀態以便未來再次警告
    if (stored < max * 0.5f)
    {
      hasWarned = false;
    }
    return comp;
  }
}
```

這個 Hook 內部保存 `hasWarned` 狀態，確保每次電量超標只提醒一次，避免刷屏，並在電量降下來後重置。還使用了 `Messages.Message` 作即時提示。

在 Patch 中，我們只需：

```
static void Postfix(CompPowerBattery __instance)
{
  ILogicHook<CompPowerBattery> hook =
BatteryHooksManager.GetOverchargeHookFor(__instance);
  // 執行 Hook 處理（可忽略返回或直接信任 Hook 內部處理）
  hook.Process(__instance);
}
```

其中 `BatteryHooksManager.GetOverchargeHookFor` 假設從某管理器取得對應Comp的Hook實例，以便每個電池追蹤自己的warn狀態。`LoggingHookDecorator` 也可包裝這些Hook以記錄何時發出警報。

執行時機與衝突風險分析

執行時機： `CompPowerBattery.CompTick` 大約每 1 秒執行數次（實際tick頻率與遊戲速度相關）。對於有大量電池的基地，我們的 Postfix 也會頻繁執行，但邏輯非常簡單（比較和一次函數呼叫），性能開銷幾乎可以忽略。此外我們在Hook中用了一個狀態變量防止重複通知，確保不會每tick都觸發昂貴的UI行為。

可能副作用： - **信息量增加：** 玩家會受到我們發出的通知。如果基地有多個電池同時滿電，可能收到多封信件或多條訊息。我們已用 `hasWarned` 做了簡單控制，但在多個電池場景下，或許需要更智能的彙總（例如一次

性提示「X個電池已接近過載」）。否則可能出現訊息泛濫，使玩家反感。 - **自動行為**：本例只是提示，若我們進一步自動關停發電機，這可能驚嚇到未預期的玩家。因此若實裝自動控制，須提供介面讓玩家了解並選擇此功能。例如在遊戲中增加建築選項開關「允許模組自動管理此設備」等，以免玩家感到遊戲被無形力量干涉。 - **特殊情況**：如果玩家**刻意**讓電池過載（例如某些挑戰玩法）我們的模組會對此進行干涉，可能與玩家目標相違。但這屬於模組設計取捨，我們假設大多數情況下過載非玩家本意。

與其他模組的潛在衝突： - **電力系統模組：** 有些大型模組（如 RimWorld的 **"電力擴充Power+"** 模組，假設存在）可能改變了電池的Comp類或行為。例如它們可能繼承並替換 `CompPowerBattery` 或更改充電邏輯。如果我們直接Patch `CompPowerBattery.CompTick`，而另一模組用了自己的子類例如 `CompPowerBatteryAdv`，那我們的Patch對新類無效。另外，如果對方也Patch了 `CompPowerBattery.CompTick`，可能也在做相似的事情（比如實現電池串聯、過載爆炸等）。此時玩家同時使用兩模組可能會**重複收到效果**甚至發生邏輯衝突（例如另一模組已經在電池滿時爆炸處理，而我們又彈信件）。解決方式可以是：在我們的模組中偵測這些模組的存在，**自動停用**相應功能或嘗試與其API協作。此外在文檔中清楚列出不相容或需要特定設定的模組。 - **資源通知類模組：** 假如存在通用的「狀態提醒」模組，它可能已覆蓋了電力/水力不足、過載等各種提醒。我們的模組功能和它部分重疊，導致玩家可能收到兩次提醒。這不會造成遊戲錯誤，但體驗不佳。我們可以提供選項關閉我們這邊的通知，只保留其他模組的，或反之。 - **低級衝突：** 我們的 Postfix 邏輯相對獨立，**不改變原方法行為**，因此與其他大部分修改電力計算的模組**技術上相容**[1]——例如別的模組改了充電速率，我們的檢查依然有效；別的模組改了最大容量，我們讀到的新容量也跟著適應。但如果另一模組**transpiler**修改了 `CompTick` 使其完全不一樣（例如改成每小時才更新一次），我們假設依然每tick Postfix，就可能誤判（因為StoredEnergy變化頻率變了）。這屬於深層相容性問題，很難全面預見，只能透過測試發現並特殊處理。

5. 事件系統與任務觸發：動態事件擴充案例

原始遊戲邏輯簡析

RimWorld 的**事件系統**（Incident）與**任務/劇情系統**（Quest）為遊戲增添隨機性與長線目標： - **Incident（隨機事件）**：由遊戲內的「故事講述者」（Storyteller）按照一定機率和間隔觸發。例如襲擊、交易商到訪、天氣異常等。每個事件由一個 `IncidentWorker` 類別處理其生成邏輯。常用模式是 `IncidentWorker_Xxx.TryExecuteWorker(IncidentParms parms)` 返回是否成功執行事件。例如 `IncidentWorker_RaidEnemy` 會根據參數生成敵對襲擊者並入侵地圖。 - **Quest（任務）**：較複雜的情景，可能包含多步驟和獎勵。資料上定義為 `QuestScriptDef` 及其組成部分。遊戲中用 `QuestManager` 來管理當前進行的任務。Quests 利用**信號（Signal）**機制來監控條件，例如擊敗所有敵人時發出信號來完成任務[9]。`SignalManager.SendSignal(Signal)` 方法負責在整個遊戲中傳播這些信號，相關任務部件（QuestPart）監聽到信號後執行對應行為（如完成任務、給予獎勵等）。

Patch 動機與情境

動機： 給予開發者更大彈性來**改寫隨機事件的結果**或**豐富任務流程**。例如： - 調整事件難度或形式：讓襲擊事件生成的敵人數量根據玩家武器科技自適應，而不僅依賴 storytellers 的點數計算。 - 在事件執行後追加額外效果：如襲擊後屍體變成殭屍（結合殭屍模組），或隕石坠落事件後在坑洞生成特定資源等等。 - 自訂任務行為：攔截任務完成信號，加入額外獎勵或後續連鎖任務；或者在特定信號發出時觸發另一些世界變化（如玩家完成某任務後，引發派系聲望改變）。

情境： 我們以**襲擊事件**為例，製作一個「援軍防禦」模組。當敵人襲擊發生時，我們希望： - 在原本敵人產生後，為玩家陣營生成一批臨時盟友援軍，協助防禦（類似於有時任務給你的友軍，但這裡我們讓隨機襲擊也有機率出現友軍來幫忙）。 - 這批援軍可能由鄰近友好派系派出，事後（倖存者）會離開地圖。 - 需要控制此機率避免每次襲擊都有援軍，以保持挑戰性。

為達成此效果，我們可以在 `IncidentWorker_RaidEnemy.TryExecuteWorker` 執行完後（敵人生成完畢）介入，添加我們的援軍生成邏輯。

Harmony Postfix 改寫實作

我們對 `IncidentWorker_RaidEnemy` 的 `TryExecuteWorker` 方法實施 Postfix Patch。該方法返回 bool 表示事件是否成功執行（通常只要能生成襲擊者就返回 true）。我們在 Postfix 中檢查如果襲擊成功執行且我們的條件符合，就生成援軍。

```
using HarmonyLib;
using RimWorld;
using Verse;
using System.Linq;

[HarmonyPatch(typeof(IncidentWorker_RaidEnemy), "TryExecuteWorker")]
public static class RaidReinforcement_Patch
{
  // 後置方法：在襲擊事件執行之後調用
  static void Postfix(bool __result, IncidentParms parms)
  {
    // 僅在原事件成功觸發（__result 為 true）時繼續
    if (!__result) return;
    Map map = (Map)parms.target;
    // 機率判定：例如只有 50% 的襲擊會觸發援軍
    if (Rand.Chance(0.5f))
    {
      // 選擇一個友好派系作為援軍來源（同主地圖同盟派系中隨機選一）
      Faction allyFaction = Find.FactionManager.AllFactions
                  .Where(f => !f.IsPlayer && f.RelationKindWith(Faction.OfPlayer) ==
FactionRelationKind.Ally)
                  .RandomElementWithFallback(null);
      if (allyFaction != null)
      {
        // 使用跟襲擊者數量類似的參數生成援軍
        IncidentParms allyParms = new IncidentParms
        {
          target = map,
          faction = allyFaction,
          points = parms.points * 0.5f, // 援軍強度約為襲擊者的一半
          spawnCenter = parms.spawnCenter, // 在相近地點產生
          spawnRotation = parms.spawnRotation
        };
        // 嘗試生成友軍襲擊（實際為友軍來協助作戰）
        IncidentDef raidFriendly = IncidentDef.Named("RaidFriendly");
        if (raidFriendly.Worker.TryExecute(allyParms))
        {
          Messages.Message($"援軍抵達！盟友派出了部隊協助防禦襲擊。",
MessageTypeDefOf.PositiveEvent);
        }
      }
```

```
    }
  }
}
```

解說： - `__result` 為原 `TryExecuteWorker` 的結果，我們僅在其為 true 時執行，表示敵人確實生成了[10]。若事件因故未成功（例如沒有可生成的點），我們不做任何事。 - 使用 `parms.target` 拿到本次事件的地圖對象 `Map`。 - 以 50% 概率決定是否派出援軍，以避免每次都出現。 - 在遊戲的派系列表中，找到**與玩家同盟**(`Ally`)的一個派系作為援軍來源。若沒有同盟派系（例如玩家與所有人交惡），則不觸發援軍。 - 構造一個新的 `IncidentParms` 作為友軍襲擊事件的參數。這裡我們使用了一個遊戲內建的事件 `RaidFriendly`（通常是任務中出現的友軍增援事件）。我們給它的點數（敵人強度）設為原襲擊點數的一半，確保援軍數量適中，不至於完全代替玩家作戰。 - 調用 `raidFriendly.Worker.TryExecute` 手動執行這個事件，生成友軍。如果成功，我們發出一條正面消息通知玩家援軍已抵達。

透過這個 Postfix Patch，我們在**不改變**原襲擊事件本身的前提下，追加了我們想要的遊戲劇情：有一定機率獲得援軍相助的驚喜，增加遊戲變化。

結合模組架構的擴充應用

將事件和任務的擴充邏輯模組化，可以考慮建立**事件處理 Hook**系統。我們可以定義 `ILogicHook<IncidentParms>` 來處理事件參數，也可以更具體如 `IRaidReinforcementHook` 介面針對襲擊事件擴充。

比如實現一個 `ReinforcementHook`：

```
public class ReinforcementHook : ILogicHook<IncidentParms>
{
  private readonly float triggerChance;
  public ReinforcementHook(float chance = 0.5f)
  {
    triggerChance = chance;
  }
  public IncidentParms Process(IncidentParms parms, Map mapContext)
  {
    if (parms.incident == IncidentDefOf.RaidEnemy && Rand.Chance(triggerChance))
    {
      Faction ally = Find.FactionManager.AllFactions
                .Where(f => !f.IsPlayer && f.RelationKindWith(Faction.OfPlayer) ==
FactionRelationKind.Ally)
                .RandomElementWithFallback(null);
      if (ally != null)
      {
        IncidentParms allyParms = new IncidentParms
        {
          target = mapContext,
          faction = ally,
          points = parms.points * 0.5f,
          spawnCenter = parms.spawnCenter,
          spawnRotation = parms.spawnRotation
        };
```

```
            if (IncidentDefOf.RaidFriendly.Worker.TryExecute(allyParms))
            {
              Messages.Message($"[Hook] {ally.Name} 援軍抵達協助防禦！",
MessageTypeDefOf.PositiveEvent);
            }
          }
        }
        return parms;
      }
    }
```

此 Hook 透過檢查 `parms.incident` 來判斷是否為襲擊事件，若是則以內部邏輯執行援軍生成。注意它返回 `IncidentParms`（此處未改動原 parms，但介面定義如此以統一結構）。

在 Postfix 中，調用 Hook：

```
static void Postfix(bool __result, IncidentParms parms)
{
  if (!__result) return;
  ILogicHook<IncidentParms> hook = new ReinforcementHook();
  hook.Process(parms, (Map)parms.target);
}
```

可見 Hook 的引入讓我們能更輕鬆地**配置觸發概率或行為**（比如將 0.5f 提取為配置），甚至可以有多個 Hook 組合，在一個事件後執行一系列擴充（如援軍、戰利品加成等）。使用 LoggingDecorator 還可以紀錄每次事件Patch執行與否，方便追蹤模組行為。

執行時機與衝突風險分析

執行時機： `IncidentWorker_RaidEnemy.TryExecuteWorker` 在故事講述者決定觸發襲擊時執行，一般而言頻率較低（視難度和劇情進展，可能幾天一次或更久）。我們的 Postfix 在每次襲擊事件**結束後**執行，確保在敵人已生成的前提下運行擴充邏輯。這意味著我們不影響事件本身能否發生，只在它**成功發生**後追加內容，所以不會干擾故事機制的平衡，只增加隨機性。

可能副作用： - **遊戲難度平衡：** 援軍的加入降低了玩家防禦壓力，特別是當玩家有多個盟友派系時，可能每次襲擊都來幫忙（如果我們沒有控制機率或加其他限制）。我們用 50% 概率和援軍強度減半來平衡，但仍可能讓遊戲變易。如果需要，可以更精細地隨機（如隨機事件牌堆概念）或根據玩家狀況決定援軍是否出現，例如玩家實力弱時盟友更可能幫忙，而玩家強大時減少援軍干涉。 - **劇情一致性：** 每次襲擊都叫援軍可能不合常理，而且盟友派系無償助戰過於慷慨。可考慮在任務系統中加入後續：比如戰後降低盟友好感度或要求報酬，以維持合理性。這超出本Patch範圍，但值得在設計上注意。 - **任務聯動：** 如果某襲擊本身是任務的一部分（比如玩家接受了一個任務，其中襲擊是挑戰之一），我們這額外援軍可能**意外降低**任務難度。甚至某些任務劇情期望玩家孤立無援地撐過難關。如果我們無差別地套用援軍機制，可能壞了這類劇本體驗。有必要偵測當前是否有相關任務進行，或此襲擊是否由任務引發，進而**選擇性停用**援軍邏輯。實務中可透過 `parms.quest` 屬性（如果存在的話）或 `parms.questTag` 來識別。

與其他模組的潛在衝突： - **事件修改模組：** 如果有其他模組也Patch了 `IncidentWorker_RaidEnemy.TryExecuteWorker`：- 若對方是 Prefix 修改了襲擊敵人生成的行為（比如改變敵人種類或數量），我們的 Postfix不沖突，仍然會在事後執行，援軍照樣生成。我們的邏輯對原始敵人怎樣不敏感，這種情況**相容**。 - 若對方也有 Postfix並加入自己的內容，例如每次襲擊後改變天氣或掉落物資。

我們的和對方的 Postfix 都會執行，順序則不定（取決於模組載入順序）。兩者通常井水不犯河水，但需注意**執行順序**可能導致預期不同：比如如果另一Postfix在我們之前，改變了 `parms` 的某些屬性，可能影響我們後面使用它（在本例中我們只用 parms 中幾個值，不太會被改）。總之，多個 Postfix 一起運作通常沒有大問題[3]，我們的附加行為獨立性高，不易和別人的邏輯衝突。 - **任務系統模組：** 某些模組可能增加新的 Quest 或改寫信號機制。如果我們想**攔截信號**來觸發額外內容，需要注意信號的識別。例如完成任務的信號通常以 `"QuestFinished"` 之類形式存在於 `QuestPart_Delay` 等模組中。如果攔截 `SignalManager.SendSignal`（這將影響**所有**信號），雖可監視我們關心的事件，但風險較高，因為其他Mod也可能Patch信號系統。我們選擇較具體的切入點（襲擊事件）來避免大範圍干擾。 - **全局行為改變：** 一些模組（如故事襲擊改變）可能不使用 `IncidentWorker_RaidEnemy` 來產生襲擊，而是自己寫機制。這種情況下我們Patch的點就失效了。但這類改動屬於大型總輯模組，我們或許無需兼容，而將其視為互斥的玩法選擇。

總體而言，此事件/任務案例展示了如何在**事件發生點**注入自訂邏輯，用 Postfix 增強遊戲劇情深度。同時提醒我們注意劇情合理性與平衡，並在與任務、其他事件改變Mod並存時，做好條件檢查以避免相衝。透過模組架構Hook的引入，我們也能較從容地管理多種事件擴充規則，使代碼維護更便利。

以上五個案例涵蓋了 RimWorld 模組開發中 Harmony Prefix/Postfix Patch 的廣泛應用範圍。我們從AI行為、社交互動、戰鬥、建築資源到事件任務，各自展示了原始邏輯、修改動機與具體實作方法。通過這些實例，我們也反覆強調了幾項**重要原則**：

- **謹慎選擇 Patch 切入點：** 了解原始程式的運作流程，選用 Prefix 或 Postfix，以最小衝擊實現需求。Prefix 可攔截或修改輸入，Postfix 可觀察並調整輸出，各有適用場景[5][3]。
- **善用 Harmony 功能：** 利用 `__instance`、`__result`、`ref` 參數修改等功能實現對原方法的影響[8][2]。避免不必要的 Transpiler 除非別無他法，因為 Prefix/Postfix 已能涵蓋多數需求且更具相容性。
- **關注相容性與副作用：** Prefix 特別是會跳過原方法的用法，可能對其他 mod **相容性較差**[4]。在開發時要考慮是否有替代方式（例如用Postfix或調整結果而非完全跳過）[3]。同時隨時評估修改對遊戲平衡和體驗的影響，在必要時提供配置或限制來調整。
- **模組化管理：** 將補丁背後的邏輯以 `ILogicHook` 等介面抽象出來，有助於**分離關注點**和**增強維護性**。裝飾器模式的應用（如 `LoggingHookDecorator`）可以方便地為所有 Hook 增加日誌、錯誤處理等橫切關注點，這也是成熟框架常用的設計技巧。

透過深度剖析上述案例，我們不僅學會如何編寫 Harmony Patch 修改 RimWorld 的方方面面，還瞭解在一個大型模組中構建**可擴充、可維護**的程式碼架構的重要性。希望這些內容能作為教材，幫助模組開發者舉一反三，在未來的創作中實現更多新奇有趣的遊戲體驗，同時避免常見陷阱與衝突，打造穩定高品質的 RimWorld 模組。[11][5]

---

[1][2][3][4][5][6][8][11] Modding Tutorials/Harmony - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/Harmony

[7] [1.0][1.1] Removing ForcedMissRadius via xml Patch results in error ...
https://ludeon.com/forums/index.php?topic=51398.0

[9] Prison Labor mod issue : r/RimWorld - Reddit
https://www.reddit.com/r/RimWorld/comments/l52ksh/prison_labor_mod_issue/

[10] bestowner and escort won't appear when the shuttle arrives [1.2 ...
https://steamcommunity.com/app/294100/discussions/0/2800630252900486978/?l=koreana