ChatGPT

RimWorld 遊戲物件擴充機制教學

本教學將針對 RimWorld 模組開發中的四種**遊戲物件擴充機制**進行逐步講解，並提供完整的範例模組內容。這四種機制包括：

1. **ThingComp：**自訂物件元件，附加於 Thing (例如建築、物品) 以儲存資料或改變行為。
2. **PawnComp：**自訂角色元件 (Pawn 的 ThingComp)，為殖民者/生物等 Pawn 添加暫存狀態，如特殊能力冷卻、士氣值等。
3. **DefModExtension：**自訂 Def 擴充屬性，透過 XML 為各種 Def 添加額外參數（例如武器的元素屬性），並在程式中讀取控制行為。
4. **運行時動態操作：**使用 GameComponent 或 MapComponent 每個遊戲 tick 主動掃描地圖上的特定物件，並根據時間或條件改變其狀態（如動態貼圖、更換產出資源）。

預期讀者為具備 C# 及 RimWorld 模組開發經驗的進階開發者。以下內容將以 Markdown 逐步說明每種機制的設計理念、XML 配置、C# 程式撰寫與測試方法，並提供可編譯執行的範例腳本、模組檔案結構。必要處也加入流程圖和類別架構圖說明，並說明如何在 **moddable-study** 專案（`Script/Extensions` 或 `Script/Logic` 架構）中整合這些機制。

1. ThingComp 物件組件擴充

ThingComp 是 RimWorld 中用於擴充 **ThingWithComps** 之行為與資料的組件類別 [1]。大多數遊戲內的實體（例如**建築**、**物品**、**植物**、甚至 **Pawn**）都繼承自 ThingWithComps，因此能附加多種 ThingComp 來賦予特殊功能 [2]。透過 ThingComp，開發者可以在特定時機（如物件生成時、每 tick、受傷時、被選取時等）執行自訂邏輯，並將所需的變量狀態保存於物件內 [3]。相較於直接修改核心類別，ThingComp 模式具有高度相容性與可存檔性，且可透過 XML **comps** 節點暴露部分功能供配置 [4] [5]。

範例場景：我們將建立一個自訂 ThingComp `CompEnergyAccumulator`，並將其附加到一種建築物件上。該組件會在每個 tick 自動累積「能量」值，且當玩家選取該物件時提供一個 Debug 按鈕輸出當前能量值。此例子展示如何以 ThingComp **儲存屬性**（能量值）、**改變物件行為**（隨時間增加能量）、以及**響應玩家操作**（點選時輸出資訊）。

開發步驟概述

1. **定義組件類別：**創建繼承自 `Verse.ThingComp` 的 C# 類別，如 `CompEnergyAccumulator`，並規劃所需的欄位（如 `storedEnergy`）、覆寫的方法（如 `CompTick` 每 tick 執行邏輯）。
2. **XML 附加組件：**在對應的 Thing 定義（ThingDef）中加入 `<comps>` 節點，指定 `compClass` 為自訂組件類別，將組件附加到該物件上 [5]。如果需要在 XML 中調整組件屬性，可另外定義配套的 `CompProperties` 類別並在 `<comps>` 中使用（本例中不需額外屬性設定）。
3. **實作組件行為：**在組件類別中覆寫 RimWorld 提供的掛鉤方法，例如 `CompTick`（每 tick 被呼叫）進行能量累積，`CompInspectStringExtra`（檢視面板資訊）顯示能量值，`CompGetGizmosExtra` 提供 Debug 按鈕，以及 `PostExposeData` 保存/讀取狀態以支援存檔。
4. **測試與驗證：**編譯並載入模組，透過開發者模式生成該物件，觀察遊戲中每 tick 能量值的變化，點擊 Debug 按鈕是否正確輸出資訊。調整參數及確認組件成功擴充了物件功能。

以下我們將詳細說明上述步驟，並提供相應的 XML 配置與 C# 程式碼範例。

XML 配置：在 ThingDef 中添加自訂組件

首先，在模組的 Defs 資料夾中定義或修改一個 ThingDef，將自訂組件加入其 `<comps>` 列表。假設我們創建一種新的建築物件 **EnergyAccumulator**（能量累積器），在 XML 定義中可如下配置：

```
<ThingDef ParentName="BuildingBase">
 <defName>EnergyAccumulator</defName>
 <label>energy accumulator</label>
 <description>一種用於測試的建築，可逐漸累積能量。</description>
 <graphicData>
  <texPath>Things/Building/EnergyAccumulator</texPath>
  <graphicClass>Graphic_Single</graphicClass>
 </graphicData>
 <thingClass>Building</thingClass>
 <tickerType>Normal</tickerType> <!-- 設置每 tick 更新，使 CompTick 正常運作 -->
 <comps>
  <li>
   <compClass>MyMod.CompEnergyAccumulator</compClass>
  </li>
 </comps>
</ThingDef>
```

上述 XML 透過 `<comps>` 節點將我們的組件類別 `MyMod.CompEnergyAccumulator` 附加到 **EnergyAccumulator** 物件。需要注意的是，我們將 `tickerType` 設為 `Normal`，以確保物件每個遊戲 tick 都觸發更新，從而驅動 `CompTick` 方法執行 [6]。如果省略此設定而物件預設非連續更新，那麼 CompTick 可能不會被呼叫。

（在實際模組中，可使用 XPath PatchOperation 將組件附加到現有的 ThingDef。例如，如果要為現有建築附加組件，應使用 `<PatchOperationAdd>` 插入 comps 節點 [7] [8] 以避免衝突。）

C# 程式碼：定義 ThingComp 類別及其行為

接下來，撰寫 `CompEnergyAccumulator` 類別。在專案的 `Scripts/Logic/` 資料夾中新增一個 C# 檔案 `CompEnergyAccumulator.cs`，內容如下：

```
using Verse;  // Verse 命名空間包含 ThingComp 定義
using UnityEngine;  // 用於使用 Log 輸出 Debug 資訊 (屬於 Unity 引擎部分)

namespace MyMod
{
  /** 能量累積組件：附加於建築，每 tick 增加能量，被選取時提供 Debug 操作 **/
  public class CompEnergyAccumulator : ThingComp
  {
    private float storedEnergy = 0f;  // 累積的能量值

    // 每遊戲 tick 調用：增加能量
    public override void CompTick()
    {
```

```
        base.CompTick();
        storedEnergy += 1f;  // 每 tick 增加1點能量 (可根據需要調整增量或條件)
    }

    // 提供額外的檢視面板資訊：顯示當前能量值
    public override string CompInspectStringExtra()
    {
        return $"Stored Energy: {storedEnergy:F0}";
    }

    // 提供額外的操作按鈕 (Gizmo)：在開發者模式下顯示一個按鈕，點擊時輸出 Debug 資訊
    public override IEnumerable<Gizmo> CompGetGizmosExtra()
    {
        // 僅在開發者模式啟用時顯示，以免影響正常遊戲體驗
        if (Prefs.DevMode)
        {
            yield return new Command_Action
            {
                defaultLabel = "DEBUG: Print Energy",
                defaultDesc = "輸出當前能量值到日誌供除錯。",
                action = () =>
                {
                    Log.Message($"[Debug] {parent.LabelCap} energy = {storedEnergy}");
                }
            };
        }
    }

    // 保存/讀取資料：確保存檔時維持能量值
    public override void PostExposeData()
    {
        base.PostExposeData();
        Scribe_Values.Look(ref storedEnergy, "storedEnergy", 0f);
    }
  }
}
```

上述程式碼實現了以下功能：

- **CompTick**：每 tick 調用一次，將 `storedEnergy` 累加。這模擬一個自動充能裝置的行為。
- **CompInspectStringExtra**：當玩家選取附有此組件的物件時，在資訊檢視面板顯示額外一行字串，內容為目前儲存的能量值。
- **CompGetGizmosExtra**：當物件被選取時提供額外的操作按鈕。這裡我們在開發者模式下添加一個 `Command_Action`，點擊後使用 `Log.Message` 輸出 debug 資訊（物件名稱及當前能量）。
- **PostExposeData**：透過 `Scribe_Values.Look` 將 `storedEnergy` 存檔/讀檔，確保遊戲存檔再載入後能量值保持一致。

RimWorld 在初始化 Thing 時會自動實例化並初始化 comps 列表中指定的組件 [9] [10]。具體流程如下圖所示：

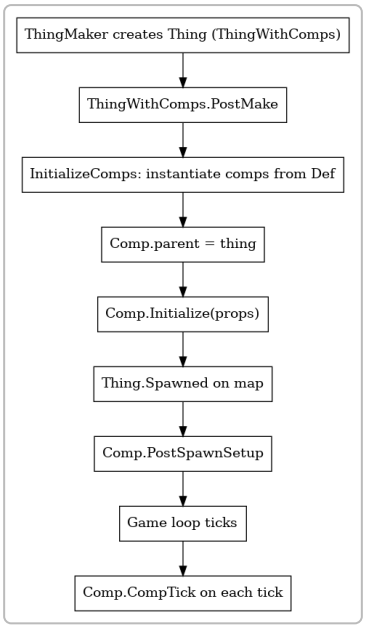

圖 1： RimWorld 中 ThingComp 的初始化與每 tick 執行流程示意圖。Thing 被建立後依序呼叫 `PostMake()`、`InitializeComps()` 初始化組件並設置父物件，再於生成至地圖時調用 `PostSpawnSetup()`。遊戲循環過程中，每個 tick 將調用組件的 `CompTick()`（需物件具備 Normal Ticker）[6]。

接著，我們將組件編譯入模組並測試其效果。

測試與結果驗證

將上述 C# 類別編譯到模組 DLL，並確保 XML 中已正確引用組件類別。載入 RimWorld 並啟用該模組，可以透過開發者模式生成 **EnergyAccumulator** 建築來測試：

- **能量累積：**將遊戲速度調快，可以在檢視面板觀察到 **Stored Energy** 數值隨著遊戲 tick 增加。若需驗證更精確，可在該組件的 `CompTick` 中加入 `Log.Message` 輸出每隔幾秒的能量值變化。
- **Debug 輸出：**選取建築物時，若處於開發者模式，物件介面下方會出現一個 **“DEBUG: Print Energy”** 按鈕。點擊此按鈕，打開 RimWorld 日誌 (開發者模式下按 **` 開啟)，可看到輸出的訊息包含該物件名稱及目前累積的能量值。

透過 ThingComp，我們成功在不修改原生類別的情況下，為建築物件新增了每 tick 自動執行的能量累積行為，以及與玩家互動的除錯訊息輸出功能。此組件完全可以保存於存檔中，卸載模組時只會在讀取存檔時出現一次性錯誤提示（RimWorld 對移除 ThingComp 有此已知非致命問題 [11]），對遊戲進行相容性良好的擴充。

> **整合於專案：**在 **moddable-study** 專案中，可將 `CompEnergyAccumulator` 類別檔案置於 `Script/Logic/` 資料夾，並將 XML 定義新增至模組 Defs 中。這樣可以保持專案結構清晰：**Logic** 類別負責行為實作，**Extensions**（若有）負責 Def 擴充定義。

2. PawnComp – Pawn 專用組件擴充

除了靜態物件，**Pawn**（殖民者或生物）同樣繼承自 ThingWithComps，因此也能透過 ThingComp 進行擴充 [2]。常見需求例如為 Pawn 添加**臨時狀態**欄位（如特殊能力冷卻時間、士氣值、憤怒值等等），並在遊戲過

程中隨事件更新。本節將示範如何建立 Pawn 專用的組件（我們暫稱 **PawnComp**），儲存 Pawn 的暫時狀態並隨遊戲進行而改變，包含：

- **每 tick 更新：**例每秒降低冷卻值或回復士氣值。
- **受傷觸發：**Pawn 受到傷害時更新狀態（如士氣下降）。
- **擊殺觸發：**Pawn 擊殺目標時更新狀態（如士氣提升、重置冷卻等）。

此外，我們將展示如何在 Pawn **初始化**時設置組件初始值，以及在**運作階段**從 Pawn 物件存取該組件與其資料。

範例場景：建立一個 `CompPawnMorale` 組件，專門用於 Pawn 的「士氣值」管理。擁有此組件的 Pawn 具有一個 `morale`（士氣）屬性：初始為 50，範圍 0–100。每 tick 慢慢自然回復（接近日常狀態），當 Pawn **受傷**時士氣降低，當 Pawn **擊殺**其它 Pawn 時（例如擊殺敵人）士氣大幅提升。我們將透過此例演示 PawnComp 的典型用法。

開發步驟概述

1. **定義 PawnComp 類別：**創建繼承自 `ThingComp` 的 C# 類別 `CompPawnMorale`（或取名為更通用的 PawnComp 類別），內含需要的欄位（如 `morale`）和覆寫方法（如 tick 更新、傷害通知）。注意 Pawn 專用組件可以直接繼承 ThingComp，RimWorld 並無內建特別的 PawnComp 基類，使用 ThingComp 即可。
2. **將組件附加到 Pawn：**由於 Pawn 的 ThingDef 定義多為種族設定，通常不直接修改核心人類/動物 Def，我們可以**透過 Trait** 機制來附加組件：定義一個特殊的 Trait，其 comps 列表包含我們的組件類別。當 Pawn 擁有此 Trait 時，遊戲會將對應的 Comp 加入 Pawn身上 [12]。這種方法能有選擇地給特定 Pawn 使用組件。(另一種做法是用**XPath patch** 將組件加入所有 Pawn 的基礎 ThingDef，但範圍較廣，在此不採用)。
3. **實作組件行為：**在 `CompPawnMorale` 中覆寫適當的方法：`CompTick()` 每 tick 自然回復士氣、`PostPostApplyDamage()` 在 Pawn **受到傷害**後調用，用於降低士氣並檢查 Pawn 是否陣亡與擊殺者資訊、`PostSpawnSetup()` 初始化士氣預設值，`PostExposeData()` 保存資料，以及 `CompInspectStringExtra()` 顯示當前士氣值便於觀察。RimWorld 會在 Pawn 受到傷害時自動通知其所有 ThingComp 執行對應的傷害處理方法 [13]（例如 `PostPreApplyDamage` 或 `PostPostApplyDamage`）。我們將利用這點來更新士氣。
4. **資料存取與測試：**透過 `Pawn.GetComp<CompPawnMorale>()` 或 `Pawn.TryGetComp<CompPawnMorale>()` 在程式中取得 Pawn 的組件實例以讀寫士氣值 [14]。在遊戲中，給某個 Pawn 添加我們定義的 Trait（可用開發者指令或在劇情中設定），然後模擬其受傷和擊殺行為，觀察士氣值的變化。例如，使用開發者工具對 Pawn 造成傷害，看士氣是否下降；生成敵人讓 Pawn 將其擊殺，驗證士氣是否上升。

XML 配置：透過 Trait 為 Pawn 附加組件

我們採用 **特質 (Trait)** 來為 Pawn 附加自訂組件。以下是一個 TraitDef 範例，命名為「Berserker」（狂戰士），當 Pawn 擁有此特質時會帶有我們的士氣組件：

```
<TraitDef>
 <defName>Berserker</defName>
 <label>狂戰士</label>
 <degreeDatas>
  <li>
   <degree>0</degree>
   <label>狂戰士</label>
```

```
    <description>戰鬥嗜血，具有額外的士氣屬性。</description>
   </li>
  </degreeDatas>
  <commonality>0.5</commonality> <!-- 出現機率，可調整或設為0僅劇本給予 -->
  <comps>
   <li>
    <compClass>MyMod.CompPawnMorale</compClass>
   </li>
  </comps>
</TraitDef>
```

以上 XML 在 TraitDef 中使用 `<comps>` 節點將我們的組件類別關聯上去。根據 RimWorld 的機制，當一個 Pawn 創建並賦予某特質時，如果該 TraitDef 定義了 comps，Pawn 將獲得這些組件（類似於一些模組對 Trait 增強的實現方式） [12] 。這意味著擁有 “Berserker” 特質的 Pawn 將自動擁有一個 `CompPawnMorale` 實例。

注意： Trait 附加組件的方式相對少見但確實可行 [12] 。如果您希望**所有**某類 Pawn 都有該組件，亦可透過 patch 在其種族 ThingDef 中加入 comps。兩種方法擇其一即可。

C# 程式碼：定義 Pawn 專用組件類別

在 `Scripts/Logic/` 資料夾中新建 `CompPawnMorale.cs`，內容如下：

```
using Verse;
using RimWorld; // 含 DamageInfo 等定義

namespace MyMod
{
  /** Pawn 士氣組件：為 Pawn 記錄士氣值，隨戰鬥狀況變化 **/
  public class CompPawnMorale : ThingComp
  {
    private int morale; // 士氣值 (0–100)

    // Pawn 創建或載入時調用。初始化預設士氣
    public override void PostSpawnSetup(bool respawningAfterLoad)
    {
      base.PostSpawnSetup(respawningAfterLoad);
      if (!respawningAfterLoad)
      {
        morale = 50; // 新建 Pawn 初始士氣50
      }
    }

    // 每tick調用：慢速自然回復士氣（每10秒增加1點至基準值50）
    public override void CompTick()
    {
      base.CompTick();
      if (Find.TickManager.TicksGame % 600 == 0) // 每600個tick (10秒) 檢查一次
      {
        if (morale < 50)
```

```
            morale++; // 略微回升士氣至50
        }
    }

    // 受傷後調用：更新士氣，處理擊殺邏輯
    public override void PostPostApplyDamage(DamageInfo dinfo, float totalDamageDealt)
    {
        base.PostPostApplyDamage(dinfo, totalDamageDealt);
        Pawn pawn = parent as Pawn;
        if (pawn == null) return;

        // Pawn 未死亡且受到傷害時：士氣降低
        if (!pawn.Dead && totalDamageDealt > 0)
        {
            morale = Mathf.Max(morale - 10, 0);
        }
        // 如果此次傷害導致 Pawn 死亡，且有加害者為 Pawn，則加害者士氣提升
        if (pawn.Dead && dinfo.Instigator is Pawn instigator)
        {
            CompPawnMorale comp = instigator.TryGetComp<CompPawnMorale>();
            if (comp != null)
            {
                comp.morale = Mathf.Min(comp.morale + 20, 100);
            }
        }
    }

    // 顯示當前士氣值在檢視面板上
    public override string CompInspectStringExtra()
    {
        return $"Morale: {morale}";
    }

    // 保存/讀取士氣值
    public override void PostExposeData()
    {
        base.PostExposeData();
        Scribe_Values.Look(ref morale, "morale", 50);
    }
  }
}
```

上述組件**僅能附加在 Pawn 上**（例如透過 Trait）。它的關鍵實現包括：

- **PostSpawnSetup**：Pawn 初始化時（非從存檔載入）將 `morale` 設為預設值 50。這確保每個新 Pawn 都有基礎士氣。
- **CompTick**：每隔一定時間緩慢回復士氣。如果士氣低於50，每10秒（600 tick）增加1點，模擬自然恢復狀態。由於 Pawn 本身會每 tick 進行 AI 行動且大多數 Pawn 相關組件不需要每 tick 都緊密更新，所以我們選擇600 tick間隔降低開銷（也可以使用 `CompTickRare` 來每250tick更新一次 [15]，此處用計數條件達成類似效果）。

- **PostPostApplyDamage**：當 Pawn 遭受傷害後執行 [16]。我們在此檢查：若 Pawn **仍活著**且傷害值大於0，則士氣降低（每次受傷-10，最低不低於0）；若該傷害導致 Pawn **死亡**，且 `dinfo.Instigator` 是另一個 Pawn（表示被某 Pawn 殺死），則取得加害者的 `CompPawnMorale`，將其士氣提高20（上限100）。透過 `instigator.TryGetComp<CompPawnMorale>()` 檢索組件，如果不為空則操作 [14]。這實現了**擊殺加士氣**的效果。
- **CompInspectStringExtra**：在選取 Pawn 時於資訊面板顯示目前士氣值，方便觀察測試。
- **PostExposeData**：存檔時保存 `morale` 值，確保重新載入後狀態不丟失。

此組件使 Pawn 能夠感知戰鬥事件並改變自身狀態，但有一個前提：**受害者 Pawn 自身需要具備此組件才能在死亡時觸發加害者加成**。換言之，只有**擁有我們特質/組件的 Pawn 之間**互相擊殺，才會有士氣變化。在本例中，我們假設給玩家殖民者套用了 “Berserker” 特質，而敵人也可以是帶有該特質（測試時可以手動給敵人添加）。否則，若想讓**任意**殺敵行為都影響擁有組件的 Pawn，需要考慮以更泛用的方式攔截擊殺事件，例如 Harmony 對 `Pawn.Kill` 進行前綴 Patch，或使用更高層的訊息系統。由於本教學專注於 RimWorld 提供的組件機制，我們不深入 Harmony 實現，但開發者可視需求選擇適當方案。

Pawn 組件資料的存取與使用

在模組程式的其他部分（例如特殊能力模組、UI 顯示等），我們可能需要讀取或修改 Pawn 組件中的資料。可以使用 RimWorld 提供的擴充方法 `GetComp<T>` 或 `TryGetComp<T>` 來取得 Pawn 的組件 [14]：

```
Pawn somePawn = ...;
CompPawnMorale moraleComp = somePawn.GetComp<CompPawnMorale>();
if (moraleComp != null)
{
  int currentMorale = moraleComp.morale;
  // 基於士氣值執行其他邏輯，例如判斷是否士氣低落觸發特殊思緒等
}
```

`Pawn` 繼承自 `ThingWithComps`，因此擁有 `GetComp<T>` 方法可直接取得指定類型的 ThingComp [14]。若不確定 Pawn 是否有該組件，可使用 `TryGetComp`（在底層也是嘗試轉型）來獲取，為空則表示沒有。開發者應該**先判空**再使用取得的組件。

測試 PawnComp 機制

1. **建立測試 Pawn**：啟動遊戲並使用開發者工具創建一個帶有 “Berserker” 特質的殖民者。例如，使用開發者指令「加入特質」將狂戰士特質賦予某個角色，或在Scenario中預先配置該特質。確認該 Pawn 的檢視面板出現 **Morale: 50** 字樣，表示組件已經附加並初始化。
2. **受傷測試**：使用開發者的「傷害工具」對該 Pawn 造成傷害（例如扣減一定健康）。在其資訊面板應該看到士氣值下降（每次傷害-10）。若傷害過高直接殺死 Pawn，由於 Pawn 已死亡，無法在其面板觀察，但我們可透過日誌或其他方式確認行為。
3. **擊殺測試**：再創建另一個帶有狂戰士特質的測試 Pawn 充當敵人。讓兩者互相戰鬥（可透過心情或直接控制草人進行攻擊）。當一方被擊殺時，檢查存活方的士氣是否上升了20點（若未滿上限）。例如，讓玩家殖民者殺死敵人，應看到玩家殖民者的 Morale 值上升。這驗證了組件在受害者死亡時找出加害者並提升其士氣的作用。

通過以上測試，可以確認 PawnComp 機制正常運作：Pawn 初始化時賦予組件及默認值，每 tick 更新狀態，受傷和擊殺事件均能觸發組件內相應的邏輯修改 Pawn 狀態。這為進階模組（如戰鬥狀態管理、特殊能力CD等）提供了一種**內嵌且可存檔**的實現方案。

整合於專案：在 **moddable-study** 專案中，建議將 Pawn 相關組件類別（如 `CompPawnMorale`）放入 `Script/Logic/` 資料夾。同時，將自訂的 TraitDef 定義加入模組 Defs（例如一個獨立的 Traits_Def 文件）。這樣在專案結構上，**Extensions** 仍保留給 Def 擴充類（DefModExtension），而 PawnComp 與 ThingComp 等邏輯組件皆在 **Logic** 下，清晰區分數據定義與行為實現。

3. DefModExtension 擴充 Def 屬性

除了組件對單一實例提供動態行為，有時我們希望**為某類別的所有物件添加靜態屬性**。RimWorld 提供的 **DefModExtension** 機制，可讓我們透過 XML 為任何 Def（定義檔）添加自訂欄位，並在 C# 中讀取這些欄位 [17]。DefModExtension 的特點是**輕量、相容**，適用於需要擴充 Def 資料而不想修改核心 Def 定義或另創子類別的情況 [18] [19]。

DefModExtension 的原理是：每個 Def 物件都有一個 `modExtensions` 列表，可以存放多個擴充資料對象 [20]。我們可以創建自訂類別繼承 `Verse.DefModExtension`，其中包含所需欄位，然後在 XML 中把此擴充附加到目標 Def。程式中即可使用 `def.GetModExtension<T>()` 獲取該擴充物件並存取欄位 [21] [22]。

範例場景：為武器定義添加**元素屬性**。假設我們想賦予部分武器“火”、“冰”、“電”等元素類型，以便在命中目標時產生特殊效果。我們可建立一個 DefModExtension `ElementalWeaponExtension`，內含一個欄位如 `elementType`（元素類型字串）。然後在特定武器的 ThingDef 中新增此擴充，並填入例如 `Fire`。遊戲中當角色使用該武器攻擊時，我們的程式可讀取其 Def 的 elementType，決定觸發對應的附加效果（例如火焰傷害、冰凍減速等）。

開發步驟概述

1. **定義擴充類別：**創建繼承自 `DefModExtension` 的 C# 類別，包含所需的公共欄位。例如 `ElementalWeaponExtension`，包含 `public string elementType;` 欄位（或用枚舉類型列舉火、冰等）。可將此類別放在 `Scripts/Extensions/` 資料夾下。
2. **在 Def 中附加擴充：**打開對應的 Def（如某武器的 ThingDef）XML，在其中新增 `<modExtensions>` 節點，插入我們定義的擴充類別，並填入欄位值 [23]。可以直接修改模組自己的 Def，或使用 PatchOperationAddModExtension 對原版 Def 進行XPath新增 [24]。
3. **程式中解析應用：**在需要的邏輯處（例如命中事件的函式）中，透過 `ThingDef.HasModExtension<T>()` 判斷某物件的 Def 是否有該擴充，如有則 `GetModExtension<T>()` 取得擴充實例並讀取其中的欄位值，據此控制遊戲行為 [22]。例如，若武器的 `elementType` 是 `"Fire"`，則在造成傷害時附加火焰效果。

C# 程式碼：定義 DefModExtension 類別

在 `Scripts/Extensions/` 資料夾建立 `ElementalWeaponExtension.cs`，內容如下：

```
using Verse;

namespace MyMod
{
  /** 武器元素屬性擴充 **/
  public class ElementalWeaponExtension : DefModExtension
  {
    public string elementType; // 元素類型: 例如 "Fire", "Ice", "Lightning"
```

```
  }
}
```

DefModExtension 非常簡單，僅需繼承自 `DefModExtension` 並定義公開欄位即可 [25]。在此我們只有一個欄位 `elementType`。請注意，DefModExtension 中不宜放複雜的物件邏輯，因為它的生命週期隨 Def 存在，是**靜態的資料容器** [19]。

我們也可以根據需要增加更多欄位。例如，可以加入 `public float bonusDamage;` 讓不同元素給予額外傷害值，或 `public ThingDef spawnedThing` 表示攻擊命中時生成的特殊物件(如火焰)等。這些都能在 XML 進行配置。

XML 配置：將擴充附加到武器 Def

假設我們有一個自訂武器 ThingDef 名為 **FlameSword**，可以這樣在 XML 中添加元素屬性擴充：

```
<ThingDef ParentName="BaseWeapon">
 <defName>FlameSword</defName>
 <!-- ...其他常規屬性定義，例如圖像、傷害等... -->
 <modExtensions>
  <li Class="MyMod.ElementalWeaponExtension">
   <elementType>Fire</elementType>
  </li>
 </modExtensions>
</ThingDef>
```

透過在 `<modExtensions>` 中加入我們的擴充類別和對應值，即可將 `"Fire"` 屬性賦予 **FlameSword** [23]。同理，可以對不同武器附加不同的 `elementType` 或其他欄位值。如需對遊戲已有武器（例如火焰噴射器）添加這種擴充，可使用類似以下的 Patch：

```
<Patch>
 <Operation Class="PatchOperationAddModExtension">
  <xpath>Defs/ThingDef[defName="Gun_FlameThrower"]</xpath>
  <value>
   <li Class="MyMod.ElementalWeaponExtension">
    <elementType>Fire</elementType>
   </li>
  </value>
 </Operation>
</Patch>
```

`PatchOperationAddModExtension` 是 RimWorld 1.1+ 新增的方便用法，能直接將擴充加入目標 Def，而不必擔心目標缺少 modExtensions 節點 [24]。

讀取與應用擴充屬性

定義好擴充類別並在 Def 配置後，我們可以在程式邏輯中使用這些擴充的資訊。 [21] [22]

例如，若要在命中目標時檢查武器的元素屬性：

```
public class WeaponEffectUtility
{
  public static void TryDoElementalEffect(Pawn attacker, Pawn victim, Thing weapon)
  {
    ThingDef weaponDef = weapon.def;
    // 確認此武器Def是否有ElementalWeaponExtension擴充
    if (weaponDef.HasModExtension<ElementalWeaponExtension>())
    {
      string element = weaponDef.GetModExtension<ElementalWeaponExtension>().elementType;
      switch (element)
      {
        case "Fire":
          // 附加火焰傷害或點燃目標
          victim.TryAttachFire(5f); // 目標著火強度5（舉例）
          break;
        case "Ice":
          // 附加冰凍減速效果 (例如通過添加自定義 Hediff)
          HediffDef freezeDef = DefDatabase<HediffDef>.GetNamed("FreezeSlow");
          victim.health.AddHediff(freezeDef);
          break;
        // ...其他元素案例...
      }
    }
  }
}
```

上述偽代碼展示瞭如何取得武器的 Def，並檢查/讀取我們的 `ElementalWeaponExtension` [22]。`HasModExtension<T>()` 用於判斷是否存在某擴充；`GetModExtension<T>()` 則返回擴充對象（若無則返回 null）。取得 `elementType` 後，用 switch 或 if-else 決定不同元素的效果。這裡示範了火元素點燃與冰元素減速的處理邏輯。

藉由 DefModExtension，我們將原本需要硬編碼在武器類別或Def中的資訊延伸出去，使其**可由XML配置**，方便日後調整和擴充。例如，想新增“毒素”元素，只需新增擴充並撰寫相應效果，而無須修改核心 Def 類型結構。

優缺點與適用性

優點：DefModExtension 相當輕量，不會像 ThingComp 那樣在每個物件實例上都佔用額外記憶體或執行頻率 [18]。擴充屬性屬於靜態資料，對**同類所有物件**均適用，且可被存檔（因為 Def 本身存於存檔） [26]。另外，透過內建的 AddModExtension Patch，使對原版 Def 的擴充也相當安全便利 [24]。

限制：擴充屬性資料是全域靜態的，**不能存放會隨物件改變的動態值** [19]。例如不能用 DefModExtension 直接跟蹤武器的耐久度變化等（那應該用 ThingComp）。此外，DefModExtension 本身不知道自己屬於哪個 Def，僅扮演資料載體角色 [27]。若需要在擴充內部區分宿主，可手動加入一個 `public Def parentDef;` 欄位並在附加時填入，但通常直接在用到的程式邏輯裡知道 Def 即可，不需要在擴充中重複此資訊。

總結而言，**DefModExtension** 非常適合用來擴充 **物種 (ThingDef)**、**能力 (AbilityDef)**、**事件 (IncidentDef)** 等的靜態配置數據。例如：為生物添加習性標記、為能力添加冷卻類別、為事件添加自訂參數等，皆可用此方式實現而無需繼承出新的 Def 子類，避開不必要的複雜性和潛在相容性問題 [18] 。

> **整合於專案：**在 **moddable-study** 中，請將 `ElementalWeaponExtension` 這類擴充定義檔案置於 `Script/Extensions/` 資料夾，以區別於一般邏輯類別。模組的 Defs 資料夾則需更新對應 Def，增加 `<modExtensions>` 節點配置。這種分離讓專案更易於維護：Extension 類別純粹描述數據結構，Logic 類別負責行為實現與應用。

4. 運行時動態掃描與狀態變更 (GameComponent/MapComponent)

除了針對單個物件的組件，有時我們需要一個**全域管理者**來在遊戲運行過程中監視並改變遊戲世界的狀態。RimWorld 提供了 **GameComponent** (遊戲級別)、**WorldComponent** (世界級別) 和 **MapComponent** (地圖級別) 這三類全域組件。它們與 ThingComp 類似，可以覆寫特定方法在特定時機執行，但作用範圍更廣 [28] [29] 。

本節聚焦 **MapComponent**（每個地圖各一個的管理組件）或 **GameComponent**（整個遊戲一個的管理組件）來實現**運行時掃描並動態改變物件**的功能。典型的例子如：「每隔一段時間掃描地圖上所有某類建築，根據時間或條件改變其貼圖，或讓其產出資源。」

範例場景：建立一個 `ExampleMapComponent`，每個遊戲 tick 都運行，在一定時間間隔掃描地圖上具有特定 **Def 擴充屬性** 的建築物。如果找到，則對其進行動態處理：本例將示範**定期產出資源**的機制——假設有些建築被標記為每隔一段時間自動產生物品，我們會在 MapComponent 中定時找到這些建築並生成對應資源。類似地，更換貼圖的功能也可在此框架中實現（只是操作變成替換建築的 Graphic 或顯示疊加物件）。

機制與設計

MapComponent 與 GameComponent 的顯著差異在於**範圍**：前者每張地圖有各自實例，後者在遊戲全局僅有一個實例。二者的用法相似，都需要繼承相應基類並**提供一個帶參數的建構子**（分別接受 `Map` 或 `Game` 參數） [30] 。RimWorld 會自動在地圖生成或遊戲開始時**實例化**我們自訂的組件並加入其管理列表，因此無需 XML 定義即可運作 [30] [31] 。

為了實現**定期掃描特定物件**，我們的 MapComponent 可以：

- 在其 `MapComponentTick()` 方法中，使用計數來每隔 N tick 執行一次掃描（避免每tick都全盤掃描造成效率問題）。
- 掃描時，遍歷 `map.listerThings` 提供的物件列表，篩選出我們關心的物件。可以根據 **defName**、**ThingDef**、或前述**DefModExtension 標記**過濾目標。例如，若我們為建築Def附加了一個 `AutoProduceExtension`（含產出資源類型與頻率），就可藉由檢查 `thing.def.HasModExtension<AutoProduceExtension>()` 來識別目標。
- 對篩選出的每個物件執行動作，如：生成物品、改變其貼圖或顏色、輸出日誌等。改變貼圖通常可透過改變 Thing 的 `Graphic` 或其 `GraphicColor` 達成，需要之後刷新地圖渲染（例如調用 `thing.Map.mapDrawer.MapMeshDirty(thing.Position, MapMeshFlag.Things)` 重新繪製）。產出物品則可以使用 `GenSpawn.Spawn()` 在地圖上生成新的 Thing。
- 考慮效能，若目標物件數量眾多，可以優化掃描策略，例如將這些物件集中管理（在其 ThingComp的 `PostSpawnSetup` 中註冊到一個清單，MapComponent 每次直接遍歷該清單而非所有地圖物件）。

C# 程式碼：MapComponent 實現定期物件掃描

在 `Scripts/Logic/` 下創建 `ExampleMapComponent.cs`，內容如下：

```
using System.Linq;
using Verse;

namespace MyMod
{
  public class ExampleMapComponent : MapComponent
  {
    public ExampleMapComponent(Map map) : base(map)
    {
    }

    public override void MapComponentTick()
    {
      base.MapComponentTick();
      // 每 2500 ticks（約遊戲內1小時）執行一次掃描
      if (Find.TickManager.TicksGame % 2500 == 0)
      {
        // 遍歷地圖上所有物件，篩選具有 AutoProduceExtension 擴充的建築
        foreach (Thing thing in map.listerThings.AllThings)
        {
          AutoProduceExtension ext = thing.def.GetModExtension<AutoProduceExtension>();
          if (ext != null)
          {
            // 根據擴充設定產出資源
            ThingDef resourceDef = DefDatabase<ThingDef>.GetNamed(ext.resourceDefName);
            // 將資源生成在建築物的互動格（或其所在格旁邊）
            IntVec3 dropCell = thing.InteractionCell;
            GenSpawn.Spawn(resourceDef, dropCell, map);
          }
        }
      }
    }
  }
}
```

這個 `ExampleMapComponent` 每小時掃描一次地圖中所有物件，若物件的 Def 有我們定義的擴充類別 `AutoProduceExtension`，則生成對應的資源物品在該物件的互動位置（InteractionCell，一般為建築物正前方或鄰近的一格，保證不與建築本身重疊）。`AutoProduceExtension` 是我們假設的擴充類別，可以這樣定義（在 `Scripts/Extensions/` 中）：

```
public class AutoProduceExtension : DefModExtension
{
  public string resourceDefName;  // 每次產出的物品 Def 名稱
```

```
    public int intervalTicks = 2500; // 產出間隔（tick數），本例中暫未直接使用
}
```

並在需要的建築 Def XML 中附加，例如：

```
<ThingDef ParentName="BuildingBase">
 <defName>ResourceGenerator</defName>
 ...
 <modExtensions>
  <li Class="MyMod.AutoProduceExtension">
   <resourceDefName>Steel</resourceDefName>
   <intervalTicks>2500</intervalTicks>
  </li>
 </modExtensions>
</ThingDef>
```

這表示 **ResourceGenerator** 這種建築每 2500 ticks 產出一個鋼鐵 (Steel)。在 `MapComponentTick` 中我們沒有使用 `intervalTicks` 欄位，而是統一採用全域2500的間隔判定；更好的實現可以根據每個 ext.intervalTicks 判定不同建築的時間間隔，避免寫死數值。

關於 MapComponent 的使用注意： RimWorld 會自動建立我們自訂的 MapComponent 實例，每張地圖各一[30]。如果模組在遊戲中途啟用，當前已有的地圖不會自動加上新的 MapComponent，可能需要透過特殊處理載入（例如上文代碼中的 `if (map.GetComponent<MyComponent>() == null) map.components.Add(new MyComponent(map));`[32][33]）。但在大部分情況下，一開始啟用模組再開新遊戲或生成新地圖，均可正常運作。

測試運行時動態行為

1. **啟用模組並生成地圖：**確保 `ExampleMapComponent` 編譯進DLL且模組啟用後，開始一個新遊戲。進入遊戲後，打開日誌介面以觀察可能的 Debug 輸出（本例程式碼未特別輸出每次生成，可在產出處加上 `Log.Message` 確認）。
2. **放置目標建築：**使用開發者工具生成一個具有 `AutoProduceExtension` 標記的建築（如 ResourceGenerator）。可以等待一段時間或加速遊戲。每過約1小時（遊戲內時間），應該會在建築旁掉落對應的資源（例如鋼鐵）。多放置幾個此類建築，可測試 MapComponent 是否對每個都正確產出。
3. **觀察效能：**若地圖上其他物件非常多，2500 tick 扫描AllThings一次仍屬可承受範圍。但我們可以在日誌中查看每小時tick耗時。如有需要，可改進為在建築 `PostSpawnSetup` 時註冊清單，MapComponent 直接迭代該清單，提高效率。

如果我們要實現**根據時間改變貼圖**的效果，也可以在 MapComponentTick 中加入時間判斷，對特定擴充的建築調整外觀。例如：

```
if (ext.changeGraphicAtNight)
{
   int hour = GenLocalDate.HourInteger(map);
   if (hour >= 18 || hour < 6)
      thing.Graphic = GraphicDatabase.Get<Graphic_Single>(ext.nightTexturePath,
ShaderDatabase.DefaultShader, thing.def.graphicData.drawSize, thing.DrawColor);
```

```
    else
        thing.Graphic = GraphicDatabase.Get<Graphic_Single>(ext.dayTexturePath,
    ShaderDatabase.DefaultShader, thing.def.graphicData.drawSize, thing.DrawColor);
    map.mapDrawer.MapMeshDirty(thing.Position, MapMeshFlag.Things); // 刷新地圖上該物件的貼圖
}
```

上述代碼假設 `AutoProduceExtension` 增加了如 `changeGraphicAtNight`、`dayTexturePath`、`nightTexturePath` 等欄位。我們根據當前小時決定使用日間或夜間貼圖，並調用 `MapMeshDirty` 通知遊戲引擎重新繪製該位置的圖層。實際應用中還需考慮只執行必要的切換，避免每tick重複設置貼圖。

GameComponent vs MapComponent

本例我們使用 MapComponent 針對單一地圖執行操作。如果希望一個組件能管理**全遊戲**的狀態（例如跨地圖的計數器、或遊戲總計分機制），可以使用 GameComponent。GameComponent 用法類似，但建構子需要 `Game` 參數 [34] 且取得方式為 `Current.Game.GetComponent<T>()` [35]。其 `GameComponentTick()` 每 tick 執行（無需關聯特定地圖）。WorldComponent 則介於兩者之間，用於模擬整個世界（通常世界地圖事件）相關的狀態。

開發者可根據模組需求選擇合適的全域組件類型：

- **MapComponent**：適用於需要針對**每張地圖分開管理**的情況，如每個地圖的天氣控制、區域管理等。透過 `map.GetComponent<MyMapComponent>()` 獲取 [36]。
- **GameComponent**：適用於**全域唯一**的狀態或管理，如總遊戲時間、全域難度調整等。透過 `Current.Game.GetComponent<MyGameComponent>()` 獲取 [35]。
- **WorldComponent**：適用於隨世界存檔存在的全域狀態，如派系關係全域影響等。透過 `Find.World.GetComponent<MyWorldComponent>()` 獲取 [37]。

測試與整合

透過上述 MapComponent，我們驗證了模組可在**遊戲運行過程**中主動掃描並作用於物件。當玩家在特定時刻沒有直接互動時，模組邏輯仍能自主推進，達到**改變遊戲世界**的效果。這對於許多進階模組很重要，例如定時事件、環境效果、自動化系統等。

在 **moddable-study** 專案中，請將自訂的 Game/MapComponent 類別檔案放入 `Script/Logic/` 下。由於這類組件不需 XML 聲明，自動載入，我們只要確保類別載入遊戲即可（DLL匯入成功）。相關的 Def 擴充類別（如 `AutoProduceExtension`）仍放在 `Script/Extensions/`，對應的 Def 則在模組 Defs 設定妥當。

經過以上四個部分的教學與範例，我們展示了 RimWorld 模組開發中**物件擴充**的主要機制：從 **ThingComp/PawnComp** 提供物件/角色的自定義行為與狀態欄位，到 **DefModExtension** 增廣 Def 的靜態配置，再到 **MapComponent/GameComponent** 實現全域的動態掃描與操控。這些機制彼此並不衝突，甚至可以**結合使用**：例如一個 MapComponent 掃描具有特定 DefModExtension 的建築，並調用其內附的 ThingComp 方法實現更複雜的行為。透徹理解並靈活運用這些擴充方式，將有助於開發者製作功能強大且架構清晰、與其他模組高度相容的 RimWorld 擴充內容。祝各位開發順利，玩轉 RimWorld 模組世界！

參考資料：

- RimWorld Modding Wiki – ThingComp 教學 [1] [4] [5] [6] [13]
- RimWorld Modding Wiki – DefModExtension 教學 [17] [22] [23] [24]
- RimWorld Modding Wiki – GameComponent/MapComponent 教學 [30] [36] [38]

- CSDN 論壇 – 關於 Pawn 特性附加 ThingComp 的討論 [12] (說明 TraitDef 可包含 comps 來賦予 Pawn 組件)

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [13] [14] [15] [16] Modding Tutorials/ThingComp - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/ThingComp

[11] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] Modding Tutorials/GameComponent - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/GameComponent

[12] RimWorld，一个叫无敌的特性，免疫伤害200次，现实一分钟恢复一次免疫伤害，可以突破上限，最高300 - CSDN文库
https://wenku.csdn.net/answer/1ecivraunc

[17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] Modding Tutorials/DefModExtension - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/DefModExtension