ChatGPT

Unity 空白專案模擬 RimWorld 的 XML 資料驅動與 Def 定義機制教學

第一章：引言 – 資料驅動設計與 RimWorld Def 架構概述

本章目標

- 了解**資料驅動**（data-driven）設計的概念與優點。
- 認識 RimWorld **Def 系統**在遊戲內容定義中的作用，以及本教材將模擬的目標。

技術重點

- 將遊戲數據配置從程式碼中分離，使用**XML**等外部資料定義遊戲內容。
- RimWorld 的 Def 架構：C# 程式碼負責行為邏輯，XML Def 負責具體數值配置 [1]。
- **Def**（Definition）概念：以資料檔案描述遊戲中的物件、事件等，各 Def 類型對應 C# 類別 [2]。

內容講解

現代遊戲常使用**資料驅動**的方式來管理遊戲內容，即將遊戲的**配置數據**（例如物品屬性、技能效果）從程式碼中抽離，以資料檔案（如 XML、JSON 等）來定義。這種設計讓**遊戲內容的擴充與修改更為容易**：開發者或模組作者無需變動原始程式碼，就能透過新增或編輯資料檔來**新增武器、調整數值**等。RimWorld 則是此設計的典型範例——RimWorld 中絕大多數物品、建築、事件等的參數都定義在 XML 檔案（稱為 Def 檔案）中，而遊戲的 C# 程式碼負責讀取這些定義並執行相應的邏輯。

舉例來說，在 RimWorld 中植物的成長行為由 C# 邏輯控制，但不同種類植物的生長時間、產出物等資料由各自的 Def 定義決定 [1]。也就是說，可以將 Def 視為「藍圖」或「配方」，用來將遊戲中的通用行為套用成特定的具體事物 [1]。C# 程式碼實現了植物如何生長、被收穫的邏輯，而具體「馬鈴薯」或「稻米」植物的生長速度、收穫物產出則由其 Def 資料決定 [1]。這種分離讓開發者可以輕鬆調整任何植物的數值，而無需修改生長邏輯的程式碼。

什麼是 Def？ 在 RimWorld 及其模組系統中，「Def」通常指代一類使用 XML 定義的資料物件，用來描述遊戲中的各種靜態內容。每種 Def 通常對應一個 C# 類別。例如，物品和建築使用 `ThingDef` 類別、天氣使用 `WeatherDef`、研究項目使用 `ResearchProjectDef`，等等 [3] [4]。所有這些 Def 類別最終都繼承自一個通用的基底類 `Def`，內含一些通用屬性（如唯一識別名稱等）。在 XML 中，每個 Def 被表示為特定的節點結構，其中包含該 Def 各項屬性的值。

在 RimWorld 的資料加載流程中，遊戲會讀取**核心**和**各模組**資料夾中的所有 Def 定義（XML 檔案），並載入到遊戲中形成**內容資料庫**。我們將在本教材中模擬類似機制：在 Unity 的空白專案中，不使用 RimWorld 本體，但實作一套簡化的 Def 系統。我們將使用 **XML 檔案**來定義虛構的遊戲內容，創建對應的 C# 類別來承載這些資料，並撰寫讀取流程將 XML 解析為物件。透過此練習，你將學會如何設計一個可擴充的資料驅動框架，使未來若要增加內容，只需新增或修改 XML，無需更動程式碼。

此外，RimWorld 自 Alpha 17 版本後引入了一項強大的功能：**PatchOperation**（資料補丁操作）。PatchOperation 允許模組作者以宣告式（declarative）的方式修改已有的 Def 定義，而不需要完全覆蓋它們。在 Alpha 17 之前，如果兩個模組嘗試修改同一個 Def，只能靠其中一個**覆寫**（overwrite）另一個，後載入的模

組會勝出 [5] 。PatchOperation 則提供了更精細的調整方式，我們也會在後續章節模擬設計一套簡單的 PatchOperation 系統。

本教材將逐步引導你建立以下觀念與技能： - 定義 XML 資料格式來描述遊戲物件（Def），以及 Unity/C# 如何載入該資料。 - 設計對應的 C# 類別結構來表示各種 Def 型別（如物件、能力等），並支援模組化擴充。 - 架設模組資料夾結構，使多個模組的 Def 資料可動態載入，並處理載入**順序**與**覆蓋規則**。 - 實作簡單的 PatchOperation 機制，讓某個模組的 XML 可以**宣告式地修改**另一模組或核心的 Def 資料。 - 最終將所有載入的 Def 建立一個統一的**資料庫**，並驗證多模組情境下覆蓋與補丁的效果。

在開始之前，假設讀者已具備 Unity 與 C# 的基礎知識，瞭解基本的類別、集合與檔案操作。本教材著重在架構與系統設計，因此程式碼會以示範性質呈現，而非完整可直接執行的商業專案代碼。透過每章的範例與練習，你將有機會親手實作並強化對資料驅動架構的理解。

練習題

1. 思考你過去玩過的遊戲中，有哪些內容可能是資料驅動的？列舉一兩個例子，說明該遊戲內容可以如何用外部資料（而非硬編碼）來定義。
2. 在 RimWorld 中，嘗試找出一個實例：遊戲內某項物品或動物的屬性由 Def 檔案決定，而行為由程式碼控制。說明這樣的拆分帶來哪些好處。

第二章：XML Def 定義的結構與載入流程

本章目標

- 瞭解 RimWorld **Def XML** 檔案的基本結構規範，以及如何在我們的 Unity 專案中設計類似的資料格式。
- 學習以 C# 讀取 XML 檔案的流程，為後續的資料解析與載入作準備。

技術重點

- **XML Def 檔案結構**：包含根節點 `<Defs>` 及多個 Def 條目，每個條目對應一個 Def 類型 [2] 。
- XML 節點與 C# 類別、欄位的對應關係：節點名稱對應類別名稱，子節點對應物件屬性 [6] 。
- 基本的 C# **XML 解析**技巧：使用 `System.Xml` 或 LINQ-to-XML (`XDocument`) 載入與遍歷 XML 節點。

內容講解

在 RimWorld（以及我們將實作的系統）中，所有 Def 資料檔案都是 XML 格式，而且每個 XML 檔的**根節點**必須是 `<Defs>` [7] 。在 `<Defs>` 節點之下，可以有多個子節點，每個子節點代表**一個 Def 條目**。下例是一個簡單的 Def XML 結構範例：

```
<?xml version="1.0" encoding="utf-8"?>
<Defs>
  <ThingDef>
    <defName>WoodenSword</defName>
    <label>木劍</label>
    <damage>10</damage>
  </ThingDef>

  <AbilityDef>
```

```
    <defName>Fireball</defName>
    <power>20</power>
    <cooldown>5</cooldown>
  </AbilityDef>
</Defs>
```

在上述範例中： - 根節點 `<Defs>` 表示這是 Def 定義檔案的起始。**所有 Def 檔案都以 `<Defs>` 起頭並結尾** [2] 。 - 其中包含兩個 Def 條目：`<ThingDef>` 和 `<AbilityDef>` 。這些標籤名稱對應我們稍後會定義的 C# 類別（如 `ThingDef` 類別與 `AbilityDef` 類別）。在 RimWorld 中，遊戲會將 `<ThingDef>` 解釋為要建立一個 `ThingDef` 類別的實例，同理 `<AbilityDef>` 對應 `AbilityDef` 類別 [2] 。 - 每個 Def 條目內部有若干子標籤（例如 `<defName>` 、`<label>` 、`<damage>` 等）。**這些子標籤名稱對應 C# 類別的欄位名稱** [6] 。例如稍後我們設計的 `ThingDef` 類別中，會有 `defName` , `label` , `damage` 等欄位。XML 中 `<damage>10</damage>` 會對應到 `ThingDef` 實例的 `damage` 欄位賦值為 10。 - `<defName>` 是非常重要的一個欄位，通常所有 Def 基類都包含此欄位作為**唯一識別名稱**。各個 Def 在同類型中應具有獨一無二的 defName，用於在程式中引用該定義，以及處理覆蓋關係。

除了簡單的數值和字串，Def XML 還可以表現**更複雜的結構**。例如，某些欄位本身是另一種物件或者是一組清單。此時會在 XML 中出現**巢狀子節點**或列表形式。舉例來說，假設我們希望在 `ThingDef` 中添加一個複雜屬性，例如「武器屬性」（WeaponProperties），其中包含射程和類型等資訊，我們可以這樣設計 XML：

```
<ThingDef>
  <defName>WoodenBow</defName>
  <label>木弓</label>
  <damage>8</damage>
  <weaponProps>
    <range>30</range>
    <type>Ranged</type>
  </weaponProps>
</ThingDef>
```

在此 XML 片段中，`<weaponProps>` 本身不是一個簡單值，而是包含兩個子節點。這表示 `ThingDef` 類別裡應該有一個欄位（例如 `weaponProps` ），其型別是一個類別 `WeaponProperties` ，而 `WeaponProperties` 類別有欄位 `range` 和 `type` 。RimWorld 的原始設計正是如此：當某個欄位是複合型別（例如結構體或自訂類別）時，其內容在 XML 中以子標籤形式給出 [6] 。同理，如果某個欄位是一個 **List 列表**（如 `List<string>` ），在 XML 中則通常表示為一組 `<li>` 子節點。例如：

```
<ThingDef>
  <defName>WoodenSword</defName>
  <tags>
    <li>Weapon</li>
    <li>Wooden</li>
  </tags>
  ...
</ThingDef>
```

此例中 `<tags>` 底下的多個 `<li>` 表示這把劍具有兩個標籤 "Weapon" 和 "Wooden"。我們稍後會在程式中處理這類 `<li>` 列表節點，將其解析為 `List<string>` 或其他類型的集合。

理解了 XML 與類別欄位的對應關係後，接下來討論**載入流程**。在 Unity C# 中，我們可以利用 .NET 提供的 XML API 來讀取和解析上述格式的檔案。常見的方式包括：

- `System.Xml` 命名空間下的類別（如 `XmlDocument`、`XmlNode` 等）可以讀取 XML 並以 DOM（文件物件模型）形式遍歷節點。
- **LINQ to XML**（`System.Xml.Linq` 命名空間），提供了 `XDocument`、`XElement` 等類別，更方便地查詢 XML 結構。

以下是載入一個 XML Def 檔案的基本步驟： 1. 從檔案系統讀取 XML 檔案內容。例如在 Unity 中，可將 Def 檔案放置於 `StreamingAssets` 資料夾，然後使用 `System.IO.File.ReadAllText` 或 `FileStream` 讀取文字。 2. 使用 `XDocument.Parse` 或 `XDocument.Load` 將字串內容解析成 XDocument 物件。 3. 獲取 XDocument 的根節點（應該是 `<Defs>`），然後遍歷其子節點。對於每個子節點（即每個 Def 條目），根據節點名稱判斷其類型。 4. **（可選）**：在純載入階段，我們可以先不立即將節點轉換為物件，而是將所有 XML 合併或保存，以便稍後執行 PatchOperation 等操作。我們知道 RimWorld 在背後會將所有模組的 Def XML **合併成一個大的 XML 文件**再解析 [8]。這樣做有利於統一進行修改和檢查。本教學稍後在處理 PatchOperation 時，將採用**先合併 XML、再整體解析**的策略。因此載入時可以先將節點保存到一個總集合中。 5. 後續再逐一將節點轉換為相應的 C# Def 物件（轉換的細節在後面的章節處理）。

為了示範，我們可以先撰寫一段簡單的 C# 代碼展示如何遍歷 Def XML 檔案的內容：

```
XDocument doc = XDocument.Load("Mods/ModA/Defs/ThingDefs/Weapons.xml"); // 假設路徑
XElement root = doc.Root; // 取得 <Defs> 根節點
foreach (XElement defElement in root.Elements())
{
  string defType = defElement.Name.LocalName;
  Console.WriteLine($"發現 Def 條目，類型：{defType}");
  // 您可以在此依 defType 創建對應物件或暫存節點，稍後再處理
}
```

上述程式碼將讀取一個 XML，然後列出每個 Def 條目的類型名稱。例如，如果遇到 `<ThingDef>` 節點，`defElement.Name.LocalName` 就是 `"ThingDef"`。

但是，在實際專案中，我們會有許多 XML 檔散落在各個模組資料夾中。我們需要擴充上述流程，使其： - **遍歷多個檔案**：例如使用 `Directory.GetFiles` 取得某個路徑下所有 `.xml` 檔案，或者遞迴搜尋子目錄中的 XML 檔案。 - **處理合併**：將不同檔案的內容合併。最簡單的方法，是對每個 XML 檔都讀入 XDocument，然後將其 `<Defs>` 子節點收集起來。例如可以建立一個新的 `XDocument`，手動新增一個根 `<Defs>` 節點，然後把每次讀入的 Def 子節點加入此根。這樣就模擬出 RimWorld 合併所有 Def 的動作 [8]。需要注意的是，要確保 XML 結構正確閉合，否則一處格式錯誤可能導致整批資料無法讀取 [8]。

以下是一個示意程式片段，展示合併多個檔案 Def 節點的流程：

```
string modsPath = "Mods"; // 假設 Mods 資料夾在執行目錄下
XDocument combinedDoc = new XDocument(new XElement("Defs")); // 建立根 <Defs>
foreach (string file in Directory.GetFiles(modsPath, "*.xml", SearchOption.AllDirectories))
{
  XDocument doc = XDocument.Load(file);
  XElement fileRoot = doc.Root;
  if (fileRoot != null && fileRoot.Name == "Defs")
```

```
    {
        foreach (XElement defNode in fileRoot.Elements())
        {
            combinedDoc.Root!.Add(defNode); // 加入合併文件的根
        }
    }
}
```

這段程式會搜尋 `Mods` 目錄下所有 `.xml` 檔案（包含子目錄），讀入後，如果根節點名為 `Defs`（符合我們的資料格式），就把其子節點追加到 `combinedDoc` 的根 `<Defs>`。完成後，`combinedDoc` 就包含了所有模組的 Def 條目。**這相當於 RimWorld 在載入時做的事情：將所有模組 Def 合成一個大的 XML** [8]。

接下來的步驟是針對 `combinedDoc` 中的每個 Def 條目，解析成相應的 C# 物件。不過，在轉為物件之前，我們還需要了解 Def 類別如何設計，以及如何根據 XML 建構物件。因此在正式將資料轉成物件並建立資料庫之前，我們需要先設計 C# 類別結構，這將在下一章詳述。

練習題

1. 實際撰寫一個簡單的 XML 檔案（可延續上述範例修改）。例如定義一個新的 `<ThingDef>` 或 `<AbilityDef>`，增加幾個屬性，確保 XML 結構正確。用您熟悉的工具（瀏覽器、XML 編輯器或自寫程式）嘗試載入它，檢查是否有語法錯誤。
2. 嘗試撰寫程式碼來讀取一個 XML 檔並輸出所有 `<defName>` 值。這將要求你在遍歷 XML 節點時，找到 `<defName>` 子節點並取值。此練習有助於熟悉 XML 的遍歷與節點存取。

第三章：自訂 Def 系統的 C# 類別設計

本章目標

- 設計符合我們 XML 結構的 C# 類別層級，包括 **Def 基底類別**以及各種特定 Def 類別（如 ThingDef, AbilityDef）。
- 瞭解如何對應 XML 標籤與類別欄位（含基本類型、物件引用與列表），為後續 XML 資料解析成物件做準備。

技術重點

- **Def 基底類別**：具備通用欄位（如 defName、label、description）及輔助函式。
- **具體 Def 類別**：根據不同遊戲概念定義欄位，如 `ThingDef` 定義物件屬性，`AbilityDef` 定義能力屬性等。
- **欄位對應**：如何在類別中聲明對應 XML 的欄位，包括簡單型別（int, float, string）、複合型別（class）、列表型別（List<T>）等。

內容講解

在撰寫資料解析程式之前，我們需要先建立相應的**類別結構**來承載 XML 中的資料。首先，我們可以定義一個抽象的基底類別 `Def`，讓所有具體的 Def 類別繼承。基底類別的目的是提供一些共通欄位和功能，例如每個 Def 都有一個 `defName` 作為唯一識別，以及可選的 `label`（給玩家顯示的名稱）和 `description`（描述）。此外，我們日後可能在 `Def` 基類中加入一些輔助方法，例如自動註冊或查找功能。

以下是一個可能的 `Def` 基底類別定義：

```
public abstract class Def
{
  public string defName;    // 唯一識別名稱（必填）
  public string label;      // 顯示名稱（選填）
  public string description; // 說明或描述（選填）
}
```

接著，我們設計具體的 Def 類別。例如，我們想模擬**物品/物件**的定義，可創建 `ThingDef` 類別；模擬**能力/技能**的定義，創建 `AbilityDef` 類別。這些類別將繼承自 `Def`，並增加各自特有的欄位。

假設在我們的簡化遊戲中： - `ThingDef` 代表遊戲中的一種物件或道具，可能包含傷害值、重量等屬性，甚至可能有複雜的子屬性（如前述的 weaponProps）。 - `AbilityDef` 代表角色能使用的一種技能，可能包含威力、冷卻時間等。

我們可以這樣定義這兩個類別：

```
public class ThingDef : Def
{
  public int damage;       // 傷害或攻擊力
  public float weight = 1f;  // 重量（有預設值）
  public WeaponProperties weaponProps; // 複合子屬性 (可為 null 表示無額外武器屬性)
  public List<string> tags = new List<string>(); // 標籤列表
}

public class AbilityDef : Def
{
  public int power;
  public float cooldown;
  public string requiredItem;  // 需要特定物品才能使用？（示例）
}
```

另外，我們定義 `WeaponProperties` 類別來說明 `ThingDef.weaponProps` 的結構：

```
public class WeaponProperties
{
  public int range;
  public string type;  // 例如 "Melee" 或 "Ranged"
}
```

這樣的類別設計基本覆蓋了我們在上一章 XML 範例中展示的結構： - `ThingDef` 有 `damage`、`weight` 等對應 `<damage>`、`<weight>` 標籤；`tags` 列表對應 `<tags>` 裡的 `<li>`；`weaponProps` 對應 `<weaponProps>` 子節點。 - `AbilityDef` 有 `power`、`cooldown` 對應 `<power>`、`<cooldown>` 等。

值得注意的是，在 RimWorld 中所有 Def 類別都繼承自一個共同的 `Verse.Def` 基類，而 `Verse.Def` 定義了一些通用欄位如 defName 等 [9] [10]。XML 的 `<defName>` 節點會填入基類的欄位 [6]。我們的 `Def` 基類正是模擬這個概念。

現在，讓我們驗證這些類別與 XML 的對應關係，以 `ThingDef` 為例：

```
<ThingDef>
  <defName>WoodenSword</defName>
  <label>木劍</label>
  <damage>10</damage>
  <weight>1.2</weight>
  <tags>
    <li>Weapon</li>
    <li>Wooden</li>
  </tags>
  <weaponProps>
    <range>1</range>
    <type>Melee</type>
  </weaponProps>
</ThingDef>
```

對應 `ThingDef` 類別：- `<defName>` ⇒ `defName = "WoodenSword"` - `<label>` ⇒ `label = "木劍"` - `<damage>` ⇒ `damage = 10` - `<weight>` ⇒ `weight = 1.2`（若未提供則使用類別的預設值 1.0） - `<tags>` 裡兩個 `<li>` ⇒ `tags = ["Weapon", "Wooden"]` [11]（字串清單，透過收集所有 `<li>` 節點值填充） - `<weaponProps>` 子節點 ⇒ 建立一個 `WeaponProperties` 實例，並對應其內 `<range>` 與 `<type>`：- `weaponProps.range = 1` - `weaponProps.type = "Melee"`

透過這種方式，我們的資料類別完全描述了 XML 裡的信息。接下來的挑戰是如何將 XML 自動映射到這些欄位上。**有幾種方案可以考慮**：

1. **手動解析與設值**：使用先前提到的 XDocument 遍歷節點，針對每個 `<ThingDef>` 創建一個 `ThingDef` 物件，然後逐一讀取其子元素，比對名稱來找到對應欄位，並進行轉型賦值。例如遇到 `<damage>10</damage>` 節點，就找到物件的 `damage` 欄位並賦值 10。這可以利用 C# **反射**（Reflection）根據名稱取得欄位，或直接寫死對不同欄位名稱的處理。
2. **XML 序列化**：利用 .NET 提供的 `System.Xml.Serialization.XmlSerializer` 將 XML 直接反序列化為物件。這要求類別的設計滿足序列化需求，例如提供無參數建構子、使用 `[XmlElement]`、`[XmlArray]` 等屬性註解來匹配 XML 結構。但由於 RimWorld 的 Def XML 常包含多種類別混在一個 `<Defs>`，直接使用內建序列化不太方便（需要額外處理動態型別）。
3. **結合方法**：先將 XML 轉為通用的結構（如字典或中繼資料結構），再根據類別結構填充。這類方法彈性較高，可處理一些特殊情況（例如後續的 PatchOperation 或繼承）。

在本教學中，我們以**方案1（手動解析）**為主進行講解，因為它可以直觀體現資料載入的過程，並方便我們插入自訂邏輯。同時也能了解 RimWorld 在背後做的事情：透過反射，將 XML 節點值賦給 C# 類別的欄位 [6]。舉個例子，在 RimWorld 原始碼片段中，有提到像 `public SomeType someTagName` 會對應 `<someTagName>...</someTagName>` [12]。

接下來我們將在第七章實作實際的解析程式碼，這裡先假想一個流程： - 當讀到 `<ThingDef>` 節點時，創建 `ThingDef` 物件。接著迭代其子節點： - 對每個子節點，例如 `<damage>10</damage>`，取出名稱 "damage"，利用 `typeof(ThingDef).GetField("damage")` 獲得欄位資訊，再將字串 "10" 轉換為 int 賦值給欄位。 - 對 `<weaponProps>` 這種子節點，偵測目標欄位型別是 `WeaponProperties`，因此創建一個 `WeaponProperties` 物件，遞迴解析其子節點 `<range>`、`<type>` 分別賦值，最後把這個物件賦給

`ThingDef.weaponProps`。 - 對 `<tags>` 這種列表節點，發現欄位型別是 `List<string>`，則遍歷 `<tags>` 底下每個 `<li>` 節點，將值加入清單。

- 當讀到 `<AbilityDef>` 節點時，流程類似但針對 `AbilityDef` 類別處理其特定欄位。

我們還需考慮**Def 之間的引用**問題。例如，如果某個 Def 欄位的型別本身是一個 Def（比如 RimWorld 中一個武器 Def 可能有欄位指定其發射的彈藥 `ThingDef`），那麼 XML 中該欄位通常以另一個 Def 的 defName 作為值 [13]。RimWorld 會在載入過程中，根據型別和 defName 去解析引用並關聯實例 [13]。在我們的系統中，若出現類似情況，可以在載入所有 Def 之後進行**第二階段處理**：掃描所有欄位，如果發現是 Def 型別且目前存的值是字串，就在已載入的 Def 資料庫中找到對應 defName 的實例，替換欄位值為實際物件。這部分屬於進階內容，可在未來擴充，目前我們的示範類別中尚未用到這種引用（`requiredItem` 可以被視為需要解析的引用，但我們可先保留字串）。

類別定義完成後，下一步是準備多個模組資料夾結構，然後撰寫程式去載入各模組的 XML 並建立這些物件。我們在進入載入與解析實作前，先討論模組化資料夾的設計，以便清晰管理多個來源的 Def 資料。

練習題

1. 在不查看後續內容的情況下，嘗試自行擴充一個新的 Def 類別。例如設計一個 `SkillDef`（假設用於定義角色技能熟練度）的類別，包含例如經驗值上限、技能描述等欄位。寫出該類別的定義，以及你認為對應的 XML 結構範例。
2. 考慮我們的 `ThingDef` 或 `AbilityDef` 是否需要**繼承**自其他類別（例如 `ThingDef` 是否要繼承 `Def` 或更細分的基類）。在 RimWorld 中，`ThingDef` 實際上繼承自 `BuildableDef`，後者又繼承自 `Def` [14]。這種多層繼承的好處是將共通欄位再細分。但在我們的簡化架構中，可以暫不考慮，或你可以嘗試為共通物件（如可建造物件）引入中間基類。想一想，如果要增加一個 `BuildingDef`（建築定義），哪些欄位可以放在共用的基類中？

第四章：模組化資料夾結構設計

本章目標

- 建立合理的**模組資料夾結構**，讓多個資料來源（模組）能共存且被載入。
- 確保資料夾結構具備**可擴充性**，方便日後增加新模組或新 Def 種類，不需要修改既有程式碼。

技術重點

- **Mods 資料夾**：模擬 RimWorld 的 Mods 資料夾，每個子目錄代表一個獨立模組。
- **Defs 子資料夾**：模組內包含 Defs 資料夾，內部分類存放不同類型的 Def XML（例如 ThingDefs, AbilityDefs）。
- **載入機制**：程式需要遞迴搜尋各模組資料夾下的 Def 檔案，讀取合併。在此結構下增減模組檔案時，程式不需改動即可適應。

內容講解

為了支援**多模組**的資料驅動，我們需要規劃專案的目錄結構。參考 RimWorld 的做法，每個模組有自己獨立的資料夾，裡面至少包含一個 `Defs` 資料夾存放各種定義檔。通常還有一個 `About` 資料夾（內含關於模組的描述，如作者、版本等），但這對我們的資料載入機制沒有直接影響，因此在模擬時可簡化忽略。

我們將在 Unity 工程目錄下創建一個 `Mods` 資料夾，結構可能如下：

```
UnityProject/
└ Assets/
   └ StreamingAssets/         # 使用 StreamingAssets 以便讀取外部檔案
      └ Mods/
         ├ ModA/
         │   ├ Defs/
         │   │   ├ ThingDefs/
         │   │   │   └ Items.xml
         │   │   └ AbilityDefs/
         │   │       └ Abilities.xml
         │   └ Patches/
         │       └ PatchOps.xml
         └ ModB/
             ├ Defs/
             │   └ ThingDefs/
             │       └ Items.xml
             └ Patches/
                 └ MyPatch.xml
```

如上所示： - `Mods/ModA` 和 `Mods/ModB` 代表兩個獨立的模組（例如 ModA 可能是基礎數據，ModB 則是想擴充或修改 ModA 的一個模組）。 - 每個模組都有 `Defs` 資料夾，用來放置定義檔案。為了整齊，我們可以在 Defs 下再按類型分類成子資料夾（如 ThingDefs, AbilityDefs 等）。**實際上，資料夾名稱並不會影響載入**（RimWorld 並不在乎 Defs 底下文件夾的名字，只要是 XML 都會載入） [15] ；分子資料夾純粹是為了模組作者組織檔案方便。因此，你可以自由規劃子資料夾，比如將所有 `ThingDef` 類型的定義放在一處。 - `Patches` 資料夾用來放置 PatchOperation 定義檔（第五章將詳述）。同樣地，其內可以有多個 XML，每個 XML 可以包含多個 Operation。

載入機制需要據此結構做調整。我們在第二章提供的程式碼其實已相對通用了：使用 `Directory.GetFiles(modsPath, "*.xml", SearchOption.AllDirectories)` 可以遍歷 Mods 資料夾下所有 XML 檔案。當我們逐一處理每個檔案時，可以從路徑推斷出它屬於哪個模組（例如包含 `ModA` 或 `ModB` 字樣），但對我們的資料合併來說這不重要——我們關心的只是彙總所有 Def 定義，以及稍後處理 Patch 定義。所以可以採取**兩階段掃描**： 1. 掃描所有 Def XML 檔案並合併。 2. 掃描所有 Patch XML 檔案並記錄（或直接套用，根據設計）。

為簡化實作，我們可以按照以下順序執行： - **載入核心資料（如果有）**：在 RimWorld 中，Core（核心）資料是作為一個基礎模組提供的。我們這裡若有預設的遊戲內建 Def，也可先載入。例如可設一個 `Mods/Core/Defs` 存放基礎定義。但本教學可先忽略，假設 ModA 充當基礎資料集。 - **載入 Mods**：按照某種順序載入。我們需要決定模組的載入順序對 override 覆蓋行為的影響（下一章討論）。在沒有特別指明的情況下，可以默認按資料夾名稱排序或者手動列表定義順序。RimWorld 是根據玩家在模組介面排列的順序來決定誰先誰後。 - **載入 Defs**：對每個模組的 Def XML 進行處理，將節點加入合併 XDocument。我們之前的程式碼片段已經示範了如何合併。需要強調的是，如果**多個模組定義了相同 defName 的 Def**，我們要決定如何處理。一般預期**後載入的模組會覆蓋前面的定義** [16] 。因此，在合併 XML 時或之後，我們得實施一個策略：遇到重複 defName（在同型別下），保留最後出現的版本。實作上，可以在加入節點前先搜尋合併文檔中是否已有相同 defName 的節點，如果有則移除或替換。 - **載入 Patches**：將各模組 Patches 資料夾下的 XML 都讀入，保存其內容（這些內容待會應用在合併的 Def XML 上）。

舉例來說，假設 `ModA/Defs/ThingDefs/Items.xml` 定義了一個 `ThingDef` defName 為 "WoodenSword"，而 `ModB/Defs/ThingDefs/Items.xml` 也定義了一個 defName 同樣為

"WoodenSword"（代表 ModB 嘗試覆蓋/修改木劍）。按照規則，ModB 載入在後，應覆蓋 ModA 的定義[16]。在我們的合併過程中，當處理 ModB 的木劍時，發現合併文檔中已經有一個 defName="WoodenSword" 的 ThingDef（來自 ModA），那麼我們可以選擇： - 移除舊的節點，插入新的（ModB 的）。 - 或者更激進地，直接替換舊節點的內容為新節點內容。

兩種效果類似，重點是**最終合併結果中，每個 defName 只保留一份定義，且是最後載入模組的版本**[16]。我們也可以在覆蓋時輸出提示，以方便檢查（例如 Unity Console 警告 "Def WoodenSword overridden by ModB"）。

下面是一個簡化的覆蓋處理邏輯示例，在前述合併迴圈中加入：

```
foreach (XElement defNode in fileRoot.Elements())
{
  string defName = defNode.Element("defName")?.Value;
  string defType = defNode.Name.LocalName;
  if (defName != null)
  {
    // 查找是否已存在相同類型且 defName 相同的節點
    XElement? existing = combinedDoc.Root!
      .Elements(defType)
      .FirstOrDefault(x => x.Element("defName")?.Value == defName);
    if (existing != null)
    {
      Console.WriteLine($"發現重複 defName，將以後者覆蓋：{defType} defName={defName}");
      existing.Remove();
    }
  }
  combinedDoc.Root!.Add(defNode);
}
```

這段程式在新增新節點前，先嘗試找有無相同類型及 defName 的節點存在。如果有，移除之（相當於舊的被覆蓋）。最後把新的節點加入合併文檔。如此可確保合併結束後，沒有重複 defName 的衝突。

當結構和載入機制確立後，要新增一個模組只需： - 新建一個模組資料夾，在裡面按照結構放置 Def XML（以及選擇性地 Patch XML）。 - 確保在載入順序上你希望它在適當的位置（例如在 Unity，可維護一個模組清單來決定順序，或者名稱排序）。

值得一提的是，我們並沒有深入處理模組**相依性**（dependency）問題。例如一個模組或許需要另一模組的資源才能正確運作。在 RimWorld 中會透過 `About.xml` 宣告依賴並在介面上要求玩家排序。這方面屬於系統使用層面的邏輯，本教材不深究。但開發者在設計載入系統時要留意：**載入順序**非常關鍵，正確的順序能避免缺少依賴或覆蓋錯誤的情況。

練習題

1. 依照上述結構，在您的 Unity 工程的 StreamingAssets/Mods 中創建兩個模組資料夾（可用本章提到的 ModA 和 ModB 命名）。為每個模組添加 Defs 資料夾和至少一個 XML 定義檔（可以複製先前章節的 XML 範例或自行設計簡單內容）。確保檔案和資料夾命名大小寫正確（在某些系統上大小寫不同視為不同路徑）。

2. 思考：如果將來您的遊戲擴充了新的 Def 類型（比如新增 `EnemyDef` 定義敵人角色），您需要在目前的結構或程式中做什麼更動嗎？如何設計才能**不改程式碼**就能支援新的 Def 類型加入？（提示：也許可以靠**反射**自動識別類別，而不需要在程式裡寫死類型列表。）

第五章：模擬 PatchOperation（宣告式修改）的設計與實作

本章目標

- 瞭解 **PatchOperation** 的用途與概念：為何需要 Patch，以及它如何以宣告方式修改 XML 資料。
- 設計一套簡化的 PatchOperation 機制，包括 Patch XML 的格式，以及對應的 C# 應用流程。
- （選擇性）實作一兩種基本的 PatchOperation（如 Add 或 Replace），示範如何套用修改至 Def 資料。

技術重點

- **PatchOperation 背景**：解決多模組覆蓋衝突，只改動目標 Def 的局部資料 [5]。
- **Patch XML 格式**：使用 `<Patch>` 根節點，內含一系列 `<Operation>`，每個 Operation 有類型（Class屬性）及 XPath 定位和修改值等 [17] [18]。
- **XPath**：XML 路徑語言，用於在 XML 文件中查找節點。理解簡單的 XPath 如 `Defs/ThingDef[defName="Wall"]/statBases` [19]。
- **實作**：透過 .NET 的 XML 操作（如 `XDocument` 的 `XPathSelectElements`）來定位節點並修改，或對應撰寫 C# 類別實現。

內容講解

在多模組並存的情況下，如果**兩個模組想修改同一個 Def**，傳統做法是其中一個模組乾脆提供該 Def 的一份完整複本並改動所需欄位。但這會造成**模組衝突**隱患：只有載入順序最後的那個修改會生效，之前的修改全部被覆蓋掉 [5]。為了解決這個問題，RimWorld 引入**PatchOperation**機制，允許模組使用聲明式的方法只修改原始 Def 的某部分。例如，我不需要複製整個武器定義，只需要說「把某武器的傷害值從10改成15」。這樣多個模組即使針對同一 Def 修改不同欄位，也可以**共同作用**，不互相覆蓋。

PatchOperation XML 格式在 RimWorld 中有其規範。在模組資料夾下，通常建立一個 `Patches` 資料夾放置 patch 檔案。每個檔案可寫為：

```
<?xml version="1.0" encoding="utf-8"?>
<Patch>
  <Operation Class="PatchOperationType">
    ... 具體操作內容 ...
  </Operation>
  <!-- 可以有多個 Operation -->
</Patch>
```

- `<Patch>` 是根節點，表示這是一組 Patch 操作定義 [20]。
- `<Operation>` 節點的 `Class` 屬性指明這是哪種操作類型。例如 `PatchOperationAdd`、`PatchOperationReplace`、`PatchOperationRemove` 等等 [21]。我們可以在程式中根據這個類型來決定如何處理。
- 每個 Operation 通常至少包含兩個子節點：

- `<xpath>`：一個 XPath 表達式，用於**鎖定目標節點** [18]。PatchOperation 要修改哪裡，就靠 XPath 找到對應的 XML 節點。例如 `Defs/ThingDef[defName="WoodenSword"]/damage` 會定位到合併後 XML 文件中 defName 為 "WoodenSword" 的 ThingDef 下的 `<damage>` 節點。
- `<value>`：要應用的值或節點。依不同操作而異：對於「Add」操作，value 代表要新增的節點；對於「Replace」操作，value 代表新值/新節點取代舊內容；對於「Remove」，通常沒有 value 只需要定位。
- 某些操作可能有其他輔助標籤，例如 `<Operation Class="PatchOperationAdd">` 可以有 `<order>Append</order>` 或 `<order>Prepend</order>` 來決定插入順序 [22]。

舉一個實際例子：我們有 ModA 定義 WoodenSword，其 `<damage>10</damage>`。現在 ModB 不想完全覆蓋 WoodenSword，而只是想把傷害調高到15，那 ModB 可以撰寫一個 PatchOperation：

```
<Patch>
  <Operation Class="PatchOperationReplace">
    <xpath>Defs/ThingDef[defName="WoodenSword"]/damage</xpath>
    <value>
      <damage>15</damage>
    </value>
  </Operation>
</Patch>
```

解讀： - 這是一個 **Replace** 操作（替換）。它會找出合併 XML 文件中，所有匹配 XPath 的節點進行替換。 - XPath: `Defs/ThingDef[defName="WoodenSword"]/damage` [19]。這將搜尋：在根 `<Defs>` 下的 `<ThingDef>`，篩選條件是其中有 `<defName>` 值為 "WoodenSword"，然後定位其下的 `<damage>` 子節點。 - `<value>` 提供了一個新的 `<damage>` 節點，內有值15。執行替換時，就是用這個新的 `<damage>15</damage>` 去替換掉原本找到的 `<damage>10</damage>` 節點。

如果成功，最終合併的 XML 文件中 WoodenSword 的 damage 會變成15。

除了 Replace，常用的 PatchOperation 還有： - **Add**：在目標節點下增加新的子節點 [23]。例如給某個武器的 `<comps>` 列表新增一個元素。 - **Remove**：刪除找到的節點 [24]。 - **Insert**：插入兄弟節點（通常指定插入在目標的前或後） [23]。 - **Attribute 操作**：修改 XML 屬性而非元素，這裡暫不深入。

RimWorld 還有一些**條件式**的操作如 PatchOperationFindMod、Conditional 等，用來根據其他模組是否存在或節點是否存在而有選擇地執行 patch [25] [26]。在我們的模擬中，可以先不實作這些進階功能，專注於核心的 Add/Replace/Remove 機制。

如何在程式中實作 PatchOperation? 主要步驟如下： 1. **解析 Patch XML**：和 Def 類似，先讀取所有 Patch 檔案，拿到 XDocument 或 XElement 結構。 2. **表示 Operation**：我們可以直接在讀取時處理每個 Operation，也可以建立對應的 C# 類別。例如建一個抽象 `PatchOperation` 類別，和 `PatchOperationAdd`、`PatchOperationReplace` 等子類別，每個類別實現一個 `Apply()` 方法，內部使用給定的 XPath 去修改主 XML。 - 例如 `PatchOperationReplace` 類別持有 `xpath` 字串和一個 `XElement value`。它的 Apply(doc) 方法會用 `doc.XPathSelectElements(this.xpath)` 找到所有匹配的元素，然後對每一個執行 `.ReplaceWith(this.value)` 操作。 - `PatchOperationAdd` 則會對匹配的每一個元素執行 `.Add(this.valueChild)`（可能要區分 Append/Prepend）。 - 這需要使用命名空間 `System.Xml.XPath` 以啟用 XDocument 的 XPath 查詢功能。 3. **應用順序**：通常所有 Def XML 讀入、合併完畢後，再應用所有 PatchOperation 到合併的 XDocument 上 [19]。因此在我們的載入流程中，應該在**轉換為 C# 物件前**執行 patch。否則如果物件已生成，再改就麻煩許多。我們的計劃是： - 第一輪收集並合併 Def XML（上章完成了）。 - 第二輪讀取所有 Patch XML，構造 PatchOperation 列表。 - 將 PatchOperation 列表依序

對合併的 Def XDocument 執行 Apply，使其直接修改 XML 結構。 - 最後再把修改後的 XDocument 解析成物件資料（第七章）。

基於上述思路，讓我們設計對應的類別和示範一段程式碼。首先 PatchOperation 基類和子類：

```
public abstract class PatchOperation
{
  public abstract void Apply(XDocument doc);
}

public class PatchOperationReplace : PatchOperation
{
  public string xpath;
  public XElement value;

  public override void Apply(XDocument doc)
  {
    foreach (var elem in doc.XPathSelectElements(xpath))
    {
      elem.ReplaceWith(new XElement(value));
      // 用value的複本替換（new XElement避免多次Replace同一引用）
    }
  }
}

public class PatchOperationAdd : PatchOperation
{
  public string xpath;
  public XElement value;
  public bool prepend = false;  // 默認追加在後，可選擇prepend

  public override void Apply(XDocument doc)
  {
    foreach (var elem in doc.XPathSelectElements(xpath))
    {
      if (prepend)
        elem.AddFirst(new XElement(value));
      else
        elem.Add(new XElement(value));
    }
  }
}
```

為了將 Patch XML 轉成這些物件，我們需要解析 Operation 節點。例如：

```
List<PatchOperation> operations = new List<PatchOperation>();
foreach (string file in Directory.GetFiles(modsPath, "*.xml", SearchOption.AllDirectories))
{
  if (Path.GetDirectoryName(file)?.EndsWith("Patches") == true)
```

```
    {
        XDocument patchDoc = XDocument.Load(file);
        foreach (XElement opElem in patchDoc.Root.Elements("Operation"))
        {
            string opClass = opElem.Attribute("Class")?.Value;
            string xpath = opElem.Element("xpath")?.Value;
            XElement valElem = opElem.Element("value")?.FirstNode as XElement;
            if (opClass == "PatchOperationReplace")
            {
                var op = new PatchOperationReplace();
                op.xpath = xpath;
                op.value = valElem;
                operations.Add(op);
            }
            else if (opClass == "PatchOperationAdd")
            {
                var op = new PatchOperationAdd();
                op.xpath = xpath;
                op.value = valElem;
                // 檢查有無order
                string order = opElem.Element("order")?.Value;
                if (order == "Prepend") op.prepend = true;
                operations.Add(op);
            }
            // ...其他類型類似處理
        }
    }
}
```

這段程式做了： - 尋找所有檔案路徑中帶有 "Patches" 資料夾的 XML。 - 對每個 Patch XML，遍歷其 `<Operation>` 子節點，根據 `Class` 屬性判斷是哪種操作，提取其中的 `<xpath>` 及 `<value>` 節點（如果有）。 - 針對每種 Class，建立對應的 PatchOperation 物件，填入資料並加入清單。

最後，我們對合併的 Def XDocument 執行所有 Patch：

```
foreach (var op in operations)
{
    op.Apply(combinedDoc);
}
```

經過這一系列處理，我們的 `combinedDoc` 就是已套用所有補丁後的終極 XML 資料。

為了檢查我們的 Patch 系統是否正常，可以在套用前後打印一些關鍵值。例如在套用前，尋找 `WoodenSword` 的 `<damage>` 值，套用 Replace 後再找一次，看是否從10變15。

注意：我們實作的 PatchOperation 是簡化版，假設 XPath 一定能找到東西，且不處理錯誤情況。在實際 RimWorld 中，如果找不到目標節點，預設會拋出錯誤終止載入；開發者可以在 PatchOperation 配置 `<log>` 等標籤控制報錯方式。我們這裡不深究，讀者應明白完善的系統還需有錯誤處理和除錯機制。

練習題

1. 嘗試撰寫一個 PatchOperationAdd 的 XML，例如給某個 Def 增加一個新子節點（比如新增一項額外屬性）。然後思考：如果該屬性原本不存在於原始 Def 類別中，Patch 加上去的資料該如何體現在我們的 C# 類別物件中？（提示：可能需要在類別中添加對應欄位，或者以某種通用結構存未識別的資料。RimWorld 中有 `ModExtension` 可附加額外資料。）
2. 我們在實作 XPath 時直接使用了 `doc.XPathSelectElements` 等便利方法。請查閱 XPath 基礎語法並舉例說明兩個不同的 XPath 查詢：
3. 查詢所有 `ThingDef` 下具有 `<cost>` 子節點的 Def。
4. 查詢特定 DefName 的父節點類型未知，例如找到 defName 為 "Fireball" 的任何 Def 節點。

第六章：Def 載入順序與覆蓋規則

本章目標

- 瞭解多模組環境下 **載入順序** 對最終資料的影響，以及如何決定優先權。
- 掌握 Def **覆蓋**（override）的規則：識別何種條件下發生覆蓋，以及我們的系統如何處理覆蓋。
- 瞭解在載入過程中可能需要提示或避免的衝突情況（例如重名衝突提醒）。

技術重點

- **載入順序**：通常為「先載核心，再載模組；模組之間按順序載入」，後載入的模組可以覆蓋先前的資料 [16]。
- **Def 唯一鍵**：`defName` 作為識別，用於決定是否視為相同 Def。相同類型且 defName 一致則被視為同一定義的覆蓋。
- **覆蓋實現**：在我們的合併與解析中，如何移除或替換已存在的 Def 節點 [16]。在物件層面，最後載入的將覆蓋字典中的舊值。
- **協同設計**：鼓勵模組作者避免無謂的 defName 衝突，並提供載入日志以供除錯。

內容講解

我們在第四章已經部分討論過載入順序與覆蓋。這裡綜合整理，並補充我們系統中如何保障正確的覆蓋邏輯。

1. 載入順序的慣例：在 RimWorld，**模組清單下方的優先級更高**，也就是「後面的覆蓋前面的」 [27] [16]。Core 核心遊戲資料最先載入，其次是前置模組，最後是玩家排在最後的模組。這意味著如果兩個模組定義衝突，後者通常會勝出。這一點我們在前面合併 XML 時的實作已反映：以 Mods 資料夾順序載入，並在碰撞時移除舊節點，保留新的。

在我們的 Unity 模擬中，我們可以**明確控制**載入順序。例如，可以在代碼中先定義一個模組順序列表：

```
string[] loadOrder = new string[] { "ModA", "ModB" };
```

然後遍歷這個順序，依序加載每個模組資料夾（而不是用 `GetFiles` 全域搜尋，因為後者不保證順序或需要自行排序）。這樣我們就能確定 ModA 的內容先合併，再合併 ModB 的。

2. defName 衝突與覆蓋：一旦**同類型**的 Def 出現重複的 defName，幾乎可以確定是要覆蓋（override） [16]。我們的系統通過 `existing.Remove()` 來處理 XML 層的衝突，確保最終只有一個定義。如果我們不是合併 XML 再解析，也可以在**物件層**做覆蓋：例如維持一個 `Dictionary<Type, Dictionary<string,`

`Def>>` 結構，每載入一個 Def 物件就插入此字典；若鍵已存在則覆蓋舊值。兩種方式都可以達成相同效果。在本模擬中，由於要給 PatchOperation 使用，我們採用**XML 合併**的方案，使 patch 可以作用在統一的 XML 上。

覆蓋並不總是**無害**的。模組作者通常會避免使用與他人相同的 defName，除非本意就是為了覆蓋別人的定義（例如修改核心遊戲的某些屬性）。因此重複 defName 可能代表模組間**存在競合**。在我們系統中，可以在偵測到覆蓋時輸出警告，提示開發者或玩家有此情況發生，尤其當並非有意覆蓋時，可能導致難以捉摸的改動。

3. 載入順序與 Patch 的互動：PatchOperation 更加凸顯載入順序的重要。**Patch 並不改變載入順序**的覆蓋邏輯，而是提供額外的修改。如果一個模組使用 Patch 修改另一模組的 Def，那該 Patch 模組必須在載入順序中**排在後者**。比如 ModB 提供 Patch 要改 ModA 的 WoodenSword，那 ModB 就需要在 ModA 後載入，否則它的 Patch 在合併時找不到 WoodenSword（或找到的只是核心定義而不是ModA改過的版本）。RimWorld 的 mod 列表通常也要求依賴於另一模組的 patch 類模組要放在目標模組之後。

4. 特殊覆蓋情況：有時模組可能**刻意**定義與核心一樣的 defName 來達成調整目的。例如 Core 定義了一個武器，模組想完全替換它，於是用相同 defName 提供新的定義。這種情況等同於 override 機制發揮作用，只是沒有用 Patch，而是直接整個取代。這在我們的系統中自然就處理了（Core 的被移除，模組的留下）。但如果**多個模組**都嘗試覆蓋核心同一Def，就會如前述，只有最後一個有效，其他被無聲覆蓋。這時若想多方兼容，只能改用 Patch 或者乾脆協調不要多個都覆寫。

在我們實作的載入流程中，順序和覆蓋的關鍵段落再次說明：

```
foreach (string mod in loadOrder)
{
  // 遍歷模組資料夾中文件 ...
  // 讀取Def，對每個defNode:
  if (defName != null)
  {
    XElement? existing = combinedDoc.Root!.Elements(defType)
                .FirstOrDefault(x => x.Element("defName")?.Value == defName);
    if (existing != null) { existing.Remove(); }
  }
  combinedDoc.Root.Add(defNode);
}
```

這裡的 `loadOrder` 決定了先處理 ModA 再 ModB。如果想調換，改變 `loadOrder` 即可。測試時你可以嘗試交換順序，看覆蓋結果有何不同。

5. 建議與日誌：為了讓開發過程順利，通常我們會記錄載入的重點資訊： - 列出載入了哪些模組、多少個 Def。 - 哪些 Def 被重複定義及覆蓋，誰覆蓋了誰。 - PatchOperation 有無錯誤或成功數。

例如，在 Unity 編輯器播放模式中，可以看到輸出的 Console：

```
Loaded ModA: 10 defs.
Loaded ModB: 3 defs.
Warning: ThingDef defName="WoodenSword" from ModA overridden by ModB.
Applied 2 patch operations from ModB.
```

這些資訊可以幫助我們確認載入順序和結果是否符合預期。

練習題

1. 在範例模組 ModA 和 ModB 中，嘗試製造一個衝突情境：讓 ModA 和 ModB 都定義一個相同 defName 的 Def（比如同一道具）。運行載入流程（可在 Unity 中透過一個腳本執行），觀察最終資料庫中該 defName 的屬性值是取自哪個模組。然後調換兩個模組的載入順序，再次執行，確認結果相反。記錄你的觀察。
2. 想像你增加了第三個模組 ModC，它也覆蓋同一個 def，且ModC放在最後載入。同時ModB對該 def 有一個 PatchOperation 修改。試預測最終結果：ModC 的覆蓋會不會把 ModB 的 Patch 改動蓋掉？（提示：Patch 是在合併後對 XML 修改的，如果 ModC 最後載入直接覆蓋了整個 def，則先前 ModB 的 Patch 作用的修改可能會消失。）此情況要如何避免？（例如確保 Patch 模組順序在最後或者改用另一種兼容方式。）

第七章：讀取 XML 與建立 Def 資料庫的流程

本章目標

- 將前面各部分串聯起來，完成**從 XML 讀取到 C# 物件建立**的整體流程。
- 構建一個統一的 **Def 資料庫**（Def Database），提供查詢功能，以便遊戲其他部分使用載入的定義。
- 驗證資料庫內容的正確性，包括各 Def 數據是否完整載入、覆蓋與 Patch 是否反映在物件上。

技術重點

- **整合流程**：按順序執行 Def 檔案合併、 PatchOperations 應用、物件實例化等步驟，確保時序正確。
- **物件實例化**：以反射或手動對應方式，將最終 XML 節點轉換為 C# Def 物件，處理各種欄位型別的賦值。
- **Def 資料庫**：設計存儲結構（如靜態列表/字典或 ScriptableObject）保存所有載入的 Def，並提供依名稱檢索的功能。
- **參考解析**（如有需要）：在物件建立完畢後，解析欄位中代表其他 Def 的字串引用，轉成實體引用，模擬 RimWorld 通過 defName 解析引用的機制 [13] 。

內容講解

經過前六章的準備，我們已經： - 規劃了 XML 資料格式和檔案結構。 - 定義了對應的 C# 類別（`Def` 基類和子類）。 - 開發了載入合併 XML 的方法，處理了模組順序與覆蓋。 - 設計並實現了 PatchOperation 系統，在 XML 上應用修改。 現在是時候將這一切串聯，將最終的 XML 資料轉換成我們程式內部可用的**Def 物件資料庫**了。

1. 從最終 XML 到物件：我們在第三章討論過如何將 XML 節點對應到類別欄位。現在基於**合併且套用補丁後**的 `combinedDoc` 來進行。高階步驟： - 遍歷 `combinedDoc.Root.Elements()`，對每個 Def 節點： - 根據節點名稱找到對應 C# 類型（如 "ThingDef" -> 類型 `ThingDef`）。 - 創建該類型的實例，例如 `Activator.CreateInstance` 或特定構造。 - 遍歷此節點的子元素 `<...>`： * 尋找類別中同名的欄位。 * 根據欄位型別進行適當的轉換賦值。 - 將實例加入資料庫中。 - 這裡我們採用**反射**以減少硬編碼。如果類別不多，手工寫 if...else 也行，但為了擴展性，用反射根據名稱找類別更佳。

我們可以先建立一個**類型對照**字典，列出目前支持的 Def 類型：

```
Dictionary<string, Type> defTypeMap = new Dictionary<string, Type>
{
  { "ThingDef", typeof(ThingDef) },
```

```
    { "AbilityDef", typeof(AbilityDef) }
};
```

這樣當我們讀到節點名時，就能拿到類型。未來如果新增 Def 類別，只需往字典加一條而不改流程。

接下來是**實例化**與**欄位賦值**，示例程式碼：

```
// 假設已定義 defTypeMap 如上
var defDatabase = new Dictionary<Type, List<Def>>();
foreach (XElement defElem in combinedDoc.Root.Elements())
{
    string defTypeName = defElem.Name.LocalName;
    if (!defTypeMap.TryGetValue(defTypeName, out Type defClass))
    {
        Console.WriteLine($"未知的 Def 種類：{defTypeName}，跳過。");
        continue;
    }
    // 建立實例
    Def defObj = (Def)Activator.CreateInstance(defClass)!;
    // 遍歷子節點填充欄位
    foreach (XElement fieldElem in defElem.Elements())
    {
        string fieldName = fieldElem.Name.LocalName;
        // 利用反射取得欄位資訊
        var fieldInfo = defClass.GetField(fieldName);
        if (fieldInfo == null)
        {
            Console.WriteLine($"類別{defClass.Name}無欄位{fieldName}，跳過賦值。");
            continue;
        }
        object? fieldValue = null;
        Type fieldType = fieldInfo.FieldType;
        if (fieldType == typeof(int))
        {
            fieldValue = int.Parse(fieldElem.Value);
        }
        else if (fieldType == typeof(float))
        {
            fieldValue = float.Parse(fieldElem.Value);
        }
        else if (fieldType == typeof(string))
        {
            fieldValue = fieldElem.Value;
        }
        else if (fieldType.IsSubclassOf(typeof(Def)))
        {
            // 如果欄位本身是一個 Def (引用)，暫存其 defName 字串，稍後解析
            fieldValue = fieldElem.Value; // 先存成字串
        }
        else if (fieldType.IsClass && fieldType != typeof(string))
```

```
    {
        // 處理複合對象，如 WeaponProperties
        object subObj = Activator.CreateInstance(fieldType)!;
        foreach (XElement subElem in fieldElem.Elements())
        {
            var subField = fieldType.GetField(subElem.Name.LocalName);
            if (subField != null)
            {
                // 假設複合物件裡都是簡單型別欄位
                if (subField.FieldType == typeof(int))
                    subField.SetValue(subObj, int.Parse(subElem.Value));
                else if (subField.FieldType == typeof(float))
                    subField.SetValue(subObj, float.Parse(subElem.Value));
                else if (subField.FieldType == typeof(string))
                    subField.SetValue(subObj, subElem.Value);
            }
        }
        fieldValue = subObj;
    }
    else if (fieldType.IsGenericType && fieldType.GetGenericTypeDefinition() == typeof(List<>))
    {
        // 處理列表，如 List<string> 或 List<Def>
        Type itemType = fieldType.GetGenericArguments()[0];
        IList listObj = (IList)Activator.CreateInstance(fieldType)!;
        foreach (XElement li in fieldElem.Elements("li"))
        {
            object? itemVal = null;
            if (itemType == typeof(string))
            {
                itemVal = li.Value;
            }
            else if (itemType.IsSubclassOf(typeof(Def)))
            {
                itemVal = li.Value; // 存下defName字串，稍後解決
            }
            // 其他可能的型別...
            if (itemVal != null) listObj.Add(itemVal);
        }
        fieldValue = listObj;
    }
    // 設定欄位的值
    if (fieldValue != null)
    {
        fieldInfo.SetValue(defObj, fieldValue);
    }
} // 完成所有子欄位賦值

// 將 defObj 加入資料庫
if (!defDatabase.ContainsKey(defClass))
    defDatabase[defClass] = new List<Def>();
```

```
    defDatabase[defClass].Add(defObj);
}
```

上述程式比較長，但邏輯上直觀地涵蓋各種情況： - **基本型別** (int, float, string)：直接解析文字內容成相應型別。 - **Def 引用**：如果欄位型別本身是 Def（例如某個欄位 `SoundDef someSound;`），我們暫時只存其字串值。稍後（在全部 Def 物件創建完畢並填入資料庫後）再解析字串轉成引用。 - **複合類別**：如果欄位是我們定義的類型（如 WeaponProperties），我們遞迴解析其子節點，創建並填值後賦給父物件欄位。 - **List 列表**：偵測 GenericType 是 List，取出 T 型別： - 若 T 是簡單型別，就將每個 `<li>` 的值轉為該型別加入 List。 - 若 T 繼承自 Def（表示列表元素是一些 Def 的引用），則同樣暫存字串。（如 RimWorld 中 `<categories><li>Metallic</li></categories>`，Metallic 是一個 StuffCategoryDef 的 defName [28]）。

2. 構建 Def 資料庫：我們用 `Dictionary<Type, List<Def>> defDatabase` 來存放，每種 Def 類型對應一個清單。其中每個元素是 Def 基類的子類實例。我們也可以為方便，建立一些靜態存取： 例如，在各個 Def 類別中加一個靜態 List：

```
public static List<ThingDef> AllDefs = new List<ThingDef>();
```

然後在上面解析時，如果 defClass 是 ThingDef，就轉型後加入 `ThingDef.AllDefs`。這種做法類似 RimWorld 裡有個 DefDatabase，可通過類型取得全部定義。我們保持簡單，用 dictionary 存。

同時，我們可以為按名稱查找提供一個**輔助字典**： 例如 `Dictionary<string, Def> defByName` 全域地存所有 defName 對應的實例（需注意不同類型可能 defName 相同，所以可以考慮 key 用 "類型+defName" 或結構再區分類型）。為簡化，可假設不同類型 defName 不衝突，直接用 defName 做唯一鍵。但更嚴謹可用 `(Type, defName)` 作鍵。

3. 解析引用：接下來處理先前暫存的字串引用。掃描整個 defDatabase 中的物件欄位： - 如果欄位型別是 Def 或 List<Def> 且裡頭存的是 string，就在 defByName 中找到同名 Def 實例替換。 例如上例的 `fieldType.IsSubclassOf(typeof(Def))` 情況，我們把 `fieldValue` 暫存為 string。可以把那些延遲解析的記錄下來，但更簡單是：在結束後，遍歷所有 Def 物件的所有欄位，再做一次：

```
foreach (var defList in defDatabase.Values)
{
  foreach (Def defObj in defList)
  {
    foreach (FieldInfo fi in defObj.GetType().GetFields())
    {
      if (fi.FieldType.IsSubclassOf(typeof(Def)))
      {
        string? defNameRef = fi.GetValue(defObj) as string;
        if (defNameRef != null && defByName.TryGetValue(defNameRef, out Def targetDef))
        {
          fi.SetValue(defObj, targetDef);
        }
      }
      else if (fi.FieldType.IsGenericType && fi.FieldType.GetGenericTypeDefinition() ==
typeof(List<>))
      {
```

```
            Type itemType = fi.FieldType.GetGenericArguments()[0];
            if (itemType.IsSubclassOf(typeof(Def)))
            {
              IList list = (IList)fi.GetValue(defObj);
              for (int i = 0; i < list.Count; i++)
              {
                string defNameRef = list[i] as string;
                if (defNameRef != null && defByName.TryGetValue(defNameRef, out Def targetDef))
                {
                  list[i] = targetDef;
                }
              }
            }
          }
        }
      }
    }
```

這段針對每個 Def 的每個欄位： - 單一 Def 引用欄位：若值是 string，從 defByName 拿對應 Def 實體賦值。 - 列表欄位：如果列表元素類型是 Def，掃描列表，把裡面每個 string 換成 Def 實體。

完成這步，所有 Def 物件之間的關聯就建立了（如果有的話）。

4. 使用 ScriptableObject（可選）：我們的資料庫目前存在於記憶體的結構中。另一種做法是建立 Unity 的 ScriptableObject 來保存這些資料，以便在編輯器查看或在不同場景間保存。舉例來說，可以製作一個 ScriptableObject 類 `DefDatabaseAsset`，裡面有多個列表欄位對應各種 Def 類型（List<ThingDef> thingDefs 等）。然後在載入完資料後，用 `ScriptableObject.CreateInstance<DefDatabaseAsset>()` 建一個實例，填入各列表，再保存為資源或暫存。 然而這部分屬於 Unity 工具性的強化，不影響核心原理。我們主要專注在運行時能用程式獲取資料即可。如果需要在編輯器檢視，可以在載入後把結果輸出到 Unity Console 或製作自訂的編輯器工具顯示。

5. 簡單驗證：要確認資料庫正確，我們可以做： - 列出每種類型加載了多少個 Def，以及各 defName 是什麼。 - 測試幾個已知的值，例如 WoodenSword 是否 damage 為 15（若有 Patch 修改）或者覆蓋後的屬性符合預期。 - 驗證引用：如果有 Def A 引用了 Def B，那麼看在物件中 A 的欄位是否真的指向 B 物件（而非僅字串）。

可以撰寫一個 Unity MonoBehaviour 腳本，把上述流程放在 `Start()` 內執行，然後打印出結果檢查。由於我們還沒有遊戲邏輯去使用這些 Def，打印檢查是主要驗證手段。

小結：到此，我們的系統已經完整地模擬了 RimWorld 的資料驅動機制： - 讀取多個模組的 XML Def -> 合併 -> 應用 PatchOperation -> 解析成 Def 類別物件 -> 存入資料庫，處理覆蓋和引用。 - 未來在遊戲執行時，腳本可以訪問這個 Def 資料庫，例如 `GetThingDef("WoodenSword")` 來拿到物件進行遊戲內邏輯（比如生成一把武器時讀取其傷害、模型等）。 - 如果要新增新內容，只需添加 XML（或透過更新模組順序改變覆蓋行為），整個系統就會自適應新的資料，而不用改寫C#代碼。

練習題

1. 將整個載入流程在 Unity 中跑通。你可以創建一個空物件，掛上一個腳本，在 `Start()` 調用主載入函數。載入完成後，用 `Debug.Log` 列出例如 ThingDef.AllDefs 裡每個元素的 defName 和一些關鍵屬性。確認 Patch 和 Override 的效果，例如如果 WoodenSword 原本damage 10被改成15，輸出是否正確顯示15。

2. 試著運用資料庫：假設我們要在遊戲中產生一個道具實例，會需要讀取 ThingDef 的資訊。寫一個函數 `SpawnThing(string defName)`，它會根據 defName在資料庫中找 ThingDef，然後實例化一個 Unity GameObject 並應用相關屬性（比如不同道具有不同模型或數值，這裡簡單打印即可）。這有助於鞏固如何從資料庫中取出定義並使用。
3. 思考擴展：如果將來想支持**Def 繼承**（RimWorld XML 支持 Def 屬性繼承機制）或 **抽象 Def**（只作為模板不實例化），你覺得在目前系統上還需要增加哪些步驟或資料？（提示：可能在解析 XML 前，需要先解析 `<ParentName>` 或 `<Abstract>` 屬性，先構造繼承鏈，再填資料。）

第八章：綜合示範與驗收挑戰

本章目標

- 通過一個完整的模擬範例，檢驗整個 Def 系統的正確性和穩健性。
- 引導讀者進行**自主實驗**：增減模組、修改定義、運用 Patch，觀察系統反應，以確信理解並掌握所學知識。

技術重點

- **範例專案**：提供一組範例模組資料（ModA 和 ModB），展示 Def 定義、覆蓋與 Patch 的綜合作用。
- **驗收點**：檢查資料庫載入結果是否符合預期，包括：
- 多個模組資料合併正確。
- defName 衝突依照順序覆蓋。
- PatchOperation 正確地修改了目標值。
- 所有定義均成功轉換為 C# 物件並可查詢。
- **進一步挑戰**：延伸練習，讓讀者在既有框架上做變化以加深理解。

內容講解

為了驗證我們的系統，我們構建以下模擬場景： - **ModA**：作為基礎模組，定義一些遊戲內容。例如一件武器 WoodenSword 和一個技能 Fireball。 - **ModB**：作為擴充/修改模組，對 WoodenSword 進行調整，並增加新的內容。

範例資料設計：

ModA/Defs/ThingDefs/Items.xml

```
<Defs>
  <ThingDef>
    <defName>WoodenSword</defName>
    <label>木製長劍</label>
    <damage>10</damage>
    <weight>1.0</weight>
    <tags>
      <li>Weapon</li>
      <li>Melee</li>
    </tags>
  </ThingDef>
</Defs>
```

（ModA 在此僅示範一個 ThingDef，實際可多個定義在一檔或分檔。）

ModA/Defs/AbilityDefs/Abilities.xml

```
<Defs>
  <AbilityDef>
    <defName>Fireball</defName>
    <power>20</power>
    <cooldown>5</cooldown>
    <requiredItem>WoodenSword</requiredItem> <!-- 需要持有木劍才能施放（虛構的例子） -->
  </AbilityDef>
</Defs>
```

解讀：ModA 定義了 **WoodenSword**（傷害10，重量1.0，標籤Weapon/Melee）以及 **Fireball** 技能（威力20，冷卻5，需要WoodenSword作為引導道具）。注意 Fireball 的 `requiredItem` 引用了 WoodenSword 的 defName，這測試我們的引用解析功能。

ModB/Defs/ThingDefs/Items.xml

```
<Defs>
  <ThingDef>
    <defName>WoodenSword</defName>
    <label>強化木劍</label>
    <damage>12</damage>
    <weight>0.9</weight>
    <tags>
      <li>Weapon</li>
      <li>Melee</li>
      <li>Enhanced</li>
    </tags>
  </ThingDef>
</Defs>
```

解讀：ModB 直接覆蓋定義 WoodenSword（defName 同名）。它修改了 label（名稱）、damage 提高到12、weight 降到0.9，並增加了一個標籤 "Enhanced" 表示這是強化版。根據覆蓋規則，若 ModB 後載入，它的 WoodenSword 應取代 ModA 的版本 [16]。但是 Fireball 的 `requiredItem` 原本指向 ModA 的 WoodenSword，現在應該也能解析到新的WoodenSword物件（因為 defName相同，物件被覆蓋）。

ModB/Patches/PatchOps.xml

```
<Patch>
  <!-- 將 WoodenSword 的傷害再提升到15 -->
  <Operation Class="PatchOperationReplace">
    <xpath>Defs/ThingDef[defName="WoodenSword"]/damage</xpath>
    <value>
      <damage>15</damage>
    </value>
```

```
  </Operation>
  <!-- 為 WoodenSword 增加一個新屬性 specialNote（假設我們在類別中沒定義，看看如何處理） -->
  <Operation Class="PatchOperationAdd">
    <xpath>Defs/ThingDef[defName="WoodenSword"]</xpath>
    <value>
      <specialNote>此物品受到神秘力量的祝福。</specialNote>
    </value>
  </Operation>
</Patch>
```

解讀：第一個 Operation 將 WoodenSword 的 `<damage>` 節點替換為15 [19] 。由於 ModB 定義的 WoodenSword.damage=12 已經覆蓋了 ModA 的10，Patch 再把12改成15。因此最終 WoodenSword 應該是傷害15。 第二個 Operation 嘗試在 WoodenSword 下新增 `<specialNote>` 子節點（此欄位我們的 `ThingDef` 類別中沒有定義）。這屬於**超出類別定義的資料**。在我們簡易實作中，它會被解析流程忽略（因為找不到對應欄位會跳過）。但我們這裡有意展示這情況，用於討論如何處理未知欄位。RimWorld 會將不識別的節點忽略或通過 `ModExtension` 机制處理。我們的實作則在解析時輸出警告並忽略 [6] 。

載入與驗證：

按照正確的順序（ModA 先，ModB 後）執行載入： 1. 載入 ModA WoodenSword, Fireball -> 合併XML。 2. 載入 ModB WoodenSword -> 發現 defName 重複，移除舊的 WoodenSword，加入新的 [16] 。 3. 載入 ModB Patch -> 應用Replace (將WoodenSword.damage改15) 和 Add (增加specialNote節點) [29] [22] 。 4. 解析最終 XML -> 建立物件： - WoodenSword 物件來自 ModB 定義的節點，但 damage 應為15（因為 PatchOperationReplace修改了 XML)；specialNote 節點無對應欄位，會被忽略。 - Fireball 物件保持不變，但其 requiredItem 在物件中是 string "WoodenSword"，稍後解析引用時應指向 WoodenSword 物件。 5. 解析引用： - Fireball.requiredItem 原是 "WoodenSword"，解析後應拿到 WoodenSword 的 ThingDef 實例。 - 其他沒有引用。

1. 構建資料庫：
2. ThingDef 列表包含 WoodenSword 一項。
3. AbilityDef 列表包含 Fireball 一項。

檢查： - WoodenSword.defName = "WoodenSword" - WoodenSword.label = "強化木劍"（ModB override) - WoodenSword.damage = 15（ModB override 12，再被Patch提高到15） - WoodenSword.weight = 0.9（ModB override） - WoodenSword.tags = ["Weapon","Melee","Enhanced"]（ModB override增加 Enhanced） - （WoodenSword.specialNote 未成為欄位，因我們未定義，忽略了。但 XML 加了該節點。如果我們要支持，可以在 ThingDef 類加對應欄位型別，比如 string specialNote。這裡忽略作為示範。） - Fireball.power = 20 - Fireball.cooldown = 5 - Fireball.requiredItem 應該是 WoodenSword ThingDef 實例（驗證其 defName或者比較引用）。

我們可以用一些 Debug.Log 或 Console.WriteLine 列出：

```
ThingDef Loaded: WoodenSword - damage:15, weight:0.9, tags: Weapon/Melee/
Enhanced
AbilityDef Loaded: Fireball - power:20, requiredItem: (ref)WoodenSword
```

並檢查 Fireball.requiredItem 這個引用是不是指向上面同個 WoodenSword 物件。若我們打印 `Fireball.requiredItem.defName` 應得到 "WoodenSword"，且引用相等於資料庫中的 WoodenSword。

可能的擴充測試：

- 改變載入順序：如果 ModB 先於 ModA 載入會怎樣？（答案：ModA 後載會覆寫 ModB 的 WoodenSword，且 ModB Patch 可能在 ModA WoodenSword上執行，也會作用，但整體變成以ModA為主，這不是預期的正常配置但可以看系統表現）。
- 在 ModB 刪除 PatchOperationReplace，看僅覆寫和不補丁的效果（WoodenSword.damage應該就是12而非15）。
- 在 ThingDef 類別中增加 `public string specialNote;` 再重新載入，讓 PatchOperationAdd 的 specialNote 能載入。然後 Fireball沒有該欄位不受影響。我們會看到 WoodenSword.specialNote = "此物品受到神秘力量的祝福。"
- 再新增一個 ModC，放在最後，它用 PatchOperation把 Fireball 的 power 改成30。測試其效果與和諧性。

經過以上的綜合示範，我們確認： - **多模組資料驅動**架構在 Unity 中可行，程式能自動組裝各處XML並生成內容。 - **override規則**運作正常：最後模組的改動勝出 [16]。 - **PatchOperation**成功讓我們不覆蓋整個Def就改變了其中的值 [29]。而且Patch能堆疊在override之上（ModB覆蓋又自己Patch）。 - 資料庫中的物件數據與預期一致，可以供遊戲邏輯使用。

至此，我們已完成 RimWorld 資料驅動機制的模擬實作。通過本教程的學習與實踐，你應該對如何使用 XML 定義遊戲數據、如何利用 C# 反射載入這些數據，以及如何設計模組化的架構有了深入的理解。這種架構不僅適用於 RimWorld，也適用於許多可模組化擴充的遊戲或應用。同學們可以在此基礎上，繼續探究更進階的功能，例如： - Def 的繼承與抽象（讓資料可以減少重覆、共用模板）。 - 更完善的 Patch 操作類型實現。 - 效能優化（大量XML載入時如何減少GC、加速XPath等）。 - 透過Unity編輯工具，自動把定義匯出或檢視，以方便設計人員編輯。

練習題

1. 挑戰：嘗試擴充本系統以支援 **Def 繼承**。具體而言，加入一種屬性例如 `<ParentName>`，允許某個 Def 繼承另一個抽象 Def 的欄位默認值。你需要在 XML 合併後解析繼承關係，然後在實例化物件時先複製父類欄位再覆蓋子類欄位（提示：RimWorld 在 Def 類別中有 `parent`、`Abstract` 等屬性來支持這個機制）。
2. 實驗：新增一個自訂的 PatchOperation 類型，例如 PatchOperationMultiply，作用是將某個數值節點乘上某個倍數（比如把所有藥草藥效乘2）。需要你在 Patch XML 定義 `<Operation Class="PatchOperationMultiply">` 並提供目標xpath和乘數，然後在程式中實現對應的 Apply 邏輯（解析節點值，轉成數字乘法，再寫回去）。測試你的實現是否能正確修改合併XML並反映到最終物件上。
3. 發散思考：資料驅動的理念可以不只用在遊戲內容上，還可以應用在遊戲的**配置**、**關卡**等等。例如你可以用類似Def的方式定義關卡參數、NPC 行為等。試想一個你可以應用本教程所學架構的新場景，描述其資料結構和載入流程。將這當作你未來開發的新起點！

[16] [5] [6]

---

[1] [2] [4] [6] [7] [8] [13] Modding Tutorials/XML Defs - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/XML_Defs

[3] Getting started with RimWorld modding on Linux
https://www.arp242.net/rimworld-mod-linux.html

[5] [15] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [29] PatchOperations - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/PatchOperations

9 10 11 12 14 28 Modding Tutorials/Def classes - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/Def_classes

16 27 Meaning of mod order - what is prioritised when overwriting? : r/RimWorld
https://www.reddit.com/r/RimWorld/comments/3uyb37/meaning_of_mod_order_what_is_prioritised_when/