ChatGPT

Def 繼承與跨定義引用

前言

在 RimWorld 的模組系統中，**Def**（Definition）檔案以 XML 格式定義遊戲中的各種元素（物件、配方、聲音等等）。本章節將深入探討 RimWorld **Def 繼承**的機制，以及 **跨 Def 引用** 的處理方式。這是進階主題，適合已具備 Unity 與 C# 開發能力，但尚不熟悉 RimWorld Def 結構的讀者。我們將從原理說明到模擬實作，帶領你了解如何建立共用模板 Def、如何在 Def 間建立引用關係，以及 Unity 專案中如何模擬 RimWorld 的 DefDatabase 架構來解析繼承與引用關係。最後，我們也會提供練習題與驗收檢核，協助你鞏固所學。

RimWorld XML Def 的繼承機制

在大量定義內容的情況下，**繼承**能大幅減少重複資料。RimWorld 採用 XML **屬性**來實現 Def 的繼承。主要有兩個關鍵屬性：

- `Name`：用於宣告一個 Def 節點的**名稱**，可供其他 Def 參考。
- `ParentName`：用於讓某個 Def **繼承**另一個 Def（即父 Def）的內容。

此外還有一個 `<Abstract>` 屬性用於標記 Def 是否為抽象（僅用作模板，不實際生成遊戲內容）。

`<Name>` 與 `<ParentName>` 的用法

當你希望定義一組**共用屬性**供多個 Def 使用時，可以創建一個模板 Def。例如，定義一個基本武器模板：

```
<ThingDef Name="BaseWeapon" Abstract="True">
  <!-- 公共的武器屬性，例如分類、可裝備性等 -->
  <category>Item</category>
  <selectable>True</selectable>
  <!-- 其他共用屬性 -->
</ThingDef>
```

上述 `<ThingDef>` 節點被標記為 `Abstract="True"` 且有名稱 `BaseWeapon`。這表示**BaseWeapon**是一個抽象模板，遊戲讀取後不會直接生成「BaseWeapon」這個物件；它的內容只是提供給其他 Def 來繼承使用 [1] [2]。換言之，**Abstract=true** 的 Def 僅做為繼承用的「幽靈」模板，**本身不會存在於遊戲中** [2]。

當我們要建立具體的武器時，就可以繼承這個模板：

```
<ThingDef ParentName="BaseWeapon">
  <defName>WoodenSword</defName>
  <label>木劍</label>
  <description>用木頭製作的劍，適合初學者使用。</description>
  <!-- 特有屬性，例如傷害值、材質等 -->
```

```
  <damage>5</damage>
</ThingDef>
```

在上述定義中，`ParentName="BaseWeapon"` 表示此 **WoodenSword** 物件**繼承**自先前定義的 **BaseWeapon** 模板。繼承發生時，**WoodenSword** 將自動取得 **BaseWeapon** 中定義的所有子節點內容，不需要在自身重複定義 [3]。我們只需在 **WoodenSword** 中填寫與基本模板不同或新增的部分，例如 `defName`（唯一識別名）和具體屬性。若 **WoodenSword** 定義了與父級相同名稱的節點，則會覆蓋父級對應的值；而未定義的節點則從父級繼承。

注意： 在 XML 中，`ParentName` 是定義在節點標籤上的屬性，而不是子標籤 [4]。例如正確寫法為：

```
<ThingDef ParentName="BaseWeapon"> ... </ThingDef>
```

而不是 `<ParentName>BaseWeapon</ParentName>`。同樣地，`Name` 也是節點的屬性，用於給父級模板命名。

抽象 Def (`Abstract`) 的作用與時機

當一個 Def 僅用作其他 Def 的父級模板，而且自身沒有完整定義或不希望出現在遊戲中時，就應將其標記為 `Abstract="True"` [5]。例如上述的 **BaseWeapon** 模板就是抽象的——它提供了共用屬性但沒有 `defName`（或可以不提供）供遊戲引用實例化。這樣的抽象 Def 在**載入時會被遊戲丟棄**，只保留繼承給子代的內容 [1]。**規則**是：如果這個 Def 的唯一用途是被其它 Def 繼承，就應該標記為 Abstract；反之則不需要 [5]。抽象 Def 不需要 `defName`（遊戲不會嘗試將其生成具體物件，因此不需要唯一識別） [6]。但是其 `Name` 屬性仍然必要，因為子 Def 會透過 `ParentName` 來引用這個名稱。

需要注意的是，**不要繼承一個未定義的父級**。如果 `ParentName` 指向的名稱不存在，或者超出了可繼承的範圍，將導致繼承機制失效或引發錯誤。例如，舊版本的 RimWorld 曾限制繼承作用域在同一模組內，但在 Alpha 15 後已允許跨模組繼承核心或其他模組的抽象 Def [7]。即使如此，**最好在自己的模組內定義所需的抽象模板**，並確保命名不與他人衝突。若兩個模組使用相同名稱的抽象 Def，後載入的模組可能覆蓋前者，導致所有繼承該名稱的 Def 改用新的父內容 [7]。為避免意外衝突，請為自定義抽象 Def 使用獨特的名稱（例如帶上模組或作者前綴）。

繼承中的列表合併與 Inherit 屬性

繼承機制對於一般的單值節點是**覆蓋**行為，但對於列表類節點則是**合併**。也就是說，如果父級 Def 有一個列表節點，而子級也定義了同名的列表節點，預設情況下子級會**保留父級列表的所有元素**，並將自己的元素附加其後。例如：

父級 Def 定義：

```
<weaponTags>
  <li>Weapon</li>
  <li>Melee</li>
</weaponTags>
```

子級 Def 定義：

```
<weaponTags>
  <li>Sword</li>
</weaponTags>
```

繼承後，子級的 `weaponTags` 將包含：`Weapon, Melee, Sword` 三個元素（父級的兩個再加上自己的）[8]。有時這不是我們想要的行為——可能我們希望子級**完全取代**父級的列表。在 RimWorld 的 XML 中，可以透過在子節點上加入 `Inherit="False"` 屬性來實現**不繼承**父列表元素的效果 [9] [10]。例如：

```
<weaponTags Inherit="False">
  <li>Sword</li>
</weaponTags>
```

加入 `Inherit="False"` 後，子級的 `weaponTags` 會**僅保留**自身定義的 `Sword`，而**不會**包含父級的 `Weapon` 或 `Melee` [10]。這個機制對處理列表尤為重要，因為列表預設是累加父項；透過 Inherit 屬性，我們可以細緻地選擇繼承哪些部分、排除哪些部分。請注意，`Inherit="False"` 必須放在**子 Def 的對應節點**上使用，告知解析器停止該節點的繼承。

跨 Def 引用機制

在 Def 系統中，各種定義之間經常需要**互相引用**。例如：

- **物件**（ThingDef）可能需要指定使用的**聲音**（SoundDef），如武器開火聲、物品互動聲等。
- **配方**（RecipeDef）需要定義**產出或材料**為哪種物品（ThingDef）。
- **研究項目**（ResearchProjectDef）可能引用所解鎖的建築或物品 Def，等等。

這些**跨 Def 引用**通常透過在 XML 中填入**目標 Def 的 defName**（或內部識別）來實現。也就是說，一個 Def 並不直接包含另一個 Def 的內容，而是以引用的形式連結。

透過 defName 引用

在 RimWorld 的 XML 中，Def 之間的關聯多以 **defName 字串**表達。遊戲在載入時會將這些字串解析為實際的 Def 物件。例如，有一把手槍定義（ThingDef）包含以下片段：

```
<soundInteract>Interact_Pistol</soundInteract>
```

這表示此手槍物件在被人拿起或放下時，會播放名為 **Interact_Pistol** 的 SoundDef 所定義的音效 [11] [12]。`<soundInteract>` 節點的值 "Interact_Pistol" 對應一個 SoundDef 的 `defName`，而非直接將聲音文件嵌入其中。再如，手槍的射擊動作定義中可能有：

```
<verbs>
  <li>
    <projectileDef>Bullet_Pistol</projectileDef>
    <soundCast>Shot_Pistol</soundCast>
    <soundCastTail>GunTail_Light</soundCastTail>
    ...
```

```
  </li>
</verbs>
```

其中 `<projectileDef>` 的值 "Bullet_Pistol" 引用的是某個子彈 ThingDef 的 `defName`，表示這把槍發射哪種子彈。而 `<soundCast>` 與 `<soundCastTail>` 分別引用了開火音效和餘音效果的 SoundDef [13] [12]。

重要: 這種引用要求**類型**與**名稱**都匹配。也就是說，解析器在處理 `<soundCast>Shot_Pistol</soundCast>` 時，會在 **SoundDef** 類型的資料庫中尋找 `defName` 為 "Shot_Pistol" 的定義；處理 `<projectileDef>Bullet_Pistol</projectileDef>` 時，則會在 **ThingDef** 資料庫中尋找名為 "Bullet_Pistol" 的定義。如果類型不符或名稱不存在，就會發生**引用解析錯誤** [14]。

加載順序與引用解析

由於各 Def 透過名稱相互引用，因此**載入順序**必須確保被引用的 Def 存在。RimWorld 的處理方式是：**先載入所有 Def，再統一解析引用**。也就是說，在初步讀取 XML 階段，引用僅僅記錄為字串，直到所有 Def 都讀入並存入 Def 資料庫後，再進行第二輪解析，把字串替換為實際的 Def 物件。

這種做法意味着：在**同一模組**內，不用擔心 Def 在檔案中的前後順序。即便 A 定義引用了 B，而 B 的定義出現在檔案之後或另一檔案，最終也能解析成功（因為整個模組的 Def 會整體載入） [14]。但在**跨模組**引用時，**模組載入的順序**就變得至關重要。如果模組 X 的 Def 需要引用模組 Y 的 Def，則必須保證**模組 Y 在模組 X 之前載入**，否則模組 X 在解析引用時找不到相應的 Def 會報錯 [15]。在實踐中，這意味着模組 X 應將模組 Y 列為前置或相依模組，透過修改 `About.xml`（例如使用 `<loadAfter>` 或 `<modDependencies>` 等標籤）來**聲明依賴關係**，確保載入順序正確。

引用解析錯誤與日誌

當某個引用找不到對應的 Def 時，RimWorld 會在日誌中報告**"Could not resolve cross-reference..."**的錯誤。 [14] 例如：如果某個 ThingDef 指定了一個不存在的 SoundDef：

```
Could not resolve cross-reference: No SoundDef named Slurp found to give to
ThingDef SomeThing
```

上述錯誤訊息表示：「無法解析跨引用：找不到名為 **Slurp** 的 SoundDef 來提供給 ThingDef **SomeThing**。」 [14]。這通常意味著 XML 中引用了一個拼寫錯誤或缺失的 defName，或是相依的模組未載入。若錯誤發生在列表中的引用，日誌還會顯示 `(wanter=XXX)` 來指明是哪個欄位/節點期待該引用 [16]（例如 `wanter=thingCategories` 表示在解析 `thingCategories` 列表時發生問題）。

實作建議: 開發者在模擬 RimWorld 的 Def 系統時，應在引用解析階段對所有未解析的引用給出適當的錯誤提示，方便調試。例如，可以列出是哪個 Def 的哪個欄位找不到對應名稱。這有助於模組作者迅速定位問題。

XML 與 DefDatabase：載入與解析流程

為了更清晰地瞭解上述繼承與引用的處理，下面我們透過一個完整流程範例來說明 **XML Def 檔案的加載解析** 過程，以及 Unity 模擬專案中 **DefDatabase** 的工作原理。

假設我們有以下三個 Def 定義（為簡化只展示重點）：

```xml
<Defs>
  <!-- 抽象父定義 -->
  <ThingDef Name="BaseMeleeWeapon" Abstract="True">
    <category>Item</category>
    <equipable>True</equipable>
    <damage>10</damage>
  </ThingDef>

  <!-- 繼承抽象父的具體物件定義 -->
  <ThingDef ParentName="BaseMeleeWeapon">
    <defName>SteelSword</defName>
    <label>鋼劍</label>
    <!-- 未特別定義 damage，將繼承父級的值 10 -->
    <damage>12</damage> <!-- 覆寫父級的傷害值為12 -->
    <soundInteract>Sword_SheathSound</soundInteract>
  </ThingDef>

  <!-- 聲音定義，供 SteelSword 引用 -->
  <SoundDef defName="Sword_SheathSound">
    <clipPath>Sounds/Weapon/SwordSheath.ogg</clipPath>
  </SoundDef>
</Defs>
```

上述 XML 包含： - 一個抽象 **近戰武器基底** `BaseMeleeWeapon`，定義了類別、可裝備性，以及預設傷害值等。 - 一個具體的 **鋼劍 SteelSword**，透過 `ParentName="BaseMeleeWeapon"` 繼承了基底武器的屬性。鋼劍改寫了傷害值（從基底的10提至12），其他如 `category` 等則直接繼承。它還定義了 `<soundInteract>` 引用了 **Sword_SheathSound**。 - 一個 **聲音 Def** 定義了 `defName` 為 Sword_SheathSound 的音效（假設這個音效在指定路徑存在）。

現在，我們從載入到解析，逐步說明 DefDatabase 如何處理：

1. **讀取與初步載入** – 遊戲（或模擬系統）首先掃描所有模組的 Def 檔案，將 XML 內容讀入記憶體結構。例如在 Unity 模擬中，我們可能使用解析器將每個 `<ThingDef>`、`<SoundDef>` 節點轉換成對應的 C# 物件實例（如 `ThingDef` 類別、`SoundDef` 類別）。在這一階段，各個 Def 物件的基本欄位（如 defName、label、描述等）已填充，但**尚未處理繼承與引用**。

2. **建立 Def 資料庫** – 將所有載入的 Def 放入 **DefDatabase**（或類似的結構）中，以便後續查找。通常資料庫按類型分類存儲，如 `DefDatabase<ThingDef>` 存放所有 ThingDef，`DefDatabase<SoundDef>` 存放所有 SoundDef。我們可以用 **defName/defID 作為鍵**來暫時識別它們。此時，鋼劍 SteelSword 的物件存在於 DefDatabase 中，BaseMeleeWeapon 也存在（儘管是抽象）。

3. **處理繼承** – 載入所有 Def 之後，開始解析繼承關係。系統會掃描每個 Def 物件，如果發現它具有 `ParentName` 屬性：

4. 取得 `ParentName` 對應的父級名稱（例如 SteelSword 的 ParentName 是 "BaseMeleeWeapon"）。
5. 在 DefDatabase 中找到 **Name**（注意不是 defName）為該值的 Def 物件作父級模板。上述例子中找到的父級就是抽象的 BaseMeleeWeapon。

6. 將父級 Def 的所有內容**複製**或套用到子級 Def 上：子級缺省的屬性由父級填入，子級已經定義的屬性保持不變或覆蓋父級值 [17] 。在我們例子中：
    - BaseMeleeWeapon 提供了 `category=Item`, `equipable=True`, `damage=10` 等；SteelSword 本身未定義 `category` 和 `equipable`，因此繼承父值，`damage` 則由子級的12覆蓋父級的10。
    - 如果有多層繼承（父級本身還繼承自更高父級），則遞歸向上合併所有祖先的內容。
    - 對於列表類節點，如父級或子級含有列表，按前節所述處理合併或過濾（尊重 `Inherit="False"` 設定）。

7. 如果找不到對應的父 Def（例如拼字錯誤或未載入），在模擬實作中應拋出錯誤或警告。RimWorld 本體對缺失父級的情況曾經不直接報錯，而是讓子級使用預設值，這可能導致難以察覺的問題 [18] 。建議我們的系統在偵測到無效的 ParentName 時**立即報錯**，避免遺漏。

8. **移除抽象 Def** – 完成繼承套用後，所有抽象 (`Abstract=true`) 的 Def 便失去存在必要。我們應將它們**從 DefDatabase 中移除**或標記為無效，不讓其參與遊戲運行 [19] 。這步確保後續遊戲邏輯只面對實際可用的定義。同時，也避免玩家在遊戲中看見如 "BaseMeleeWeapon" 這樣不應存在的條目。

9. **解析跨 Def 引用** – 現在，各 Def 內容已最終確定，我們開始將之前僅以字串表示的引用解析為物件：

10. 系統遍歷每一個 Def 的字段，尋找那些應該引用其他 Def 的屬性或列表。例如 SteelSword 的 `soundInteract` 字段類型是 SoundDef，且其值目前是字串 "Sword_SheathSound"。
11. 對每個這類字段，利用 DefDatabase 按類型查找相應 defName。例如在 SoundDef 資料庫中尋找 defName = "Sword_SheathSound" 的 SoundDef。
12. 如果找到，則將 SteelSword.soundInteract 字段賦值為該 SoundDef 物件的引用，表示建立了正確的關聯。對於列表類引用（例如某 Def 含有一個 `requiredItems` 列表，每個元素是另一ThingDef），則對列表中的每個名稱執行同樣的解析，找到對應 Def 後放入列表物件。
13. 如果**找不到**對應的 Def，則記錄錯誤。比如，若 SteelSword 引用的 "Sword_SheathSound" 沒有定義，系統會報告「找不到名為 Sword_SheathSound 的 SoundDef，無法設定給 ThingDef SteelSword」的錯誤 [14] 。透過前述的日誌，我們能迅速定位配置上的問題。

14. 特殊情況：跨模組引用如果未找到，通常是因為目標 Def 所在模組缺失或未載入，這時亦會視作找不到處理。開發者應透過依賴設定來避免此情況。

15. **完成載入** – 經過上述步驟，DefDatabase 中的每一項 Def 都已整合了繼承內容，且所有引用（如果正確）都轉成物件引用。系統現在擁有一套完整且一致的定義數據，可供遊戲各系統使用。此時，我們也可以在 Unity 編輯器或日誌中檢查 DefDatabase 狀態，確認每種 Def 的數量、內容，以及隨機抽查某些 Def 的引用是否已正確連結。

以上流程對應到 RimWorld 本身的加載機制，可以簡要比對：

- **步驟1-2** 相當於 RimWorld **讀入所有模組 Def** 並分類存儲的過程（實際上 RimWorld 是逐個模組讀取合併） [20] 。
- **步驟3** 對應 **繼承解析**：RimWorld 在載入每個模組時即處理繼承，把 ParentName 的內容展開覆蓋 [17] 。
- **步驟4** 是 **棄用抽象 Def**：RimWorld將 Abstract 的定義在內存中移除，不讓它們出現在正式 Def 列表內 [19] 。
- **步驟5** 即 **跨引用解析**：RimWorld 有專門的流程解析所有 def 中的 cross-reference 字串，將其匹配到 Def 物件，並報告未解析的引用 [14] 。
- **步驟6** 最後完成所有模組的載入。

理解這個過程有助於我們在 Unity 專案中實作類似的系統時，正確地排序步驟、處理潛在錯誤。

Unity 專案中的模擬實作

在前一章節中，我們已建立基本的 **ModManager** 和 **DefDatabase** 架構，用於載入 Mods 資料夾下的 XML 定義檔。現在，我們將延續該架構，加入**繼承**與**引用**的支持。

DefDatabase 與 Definition 類別

在模擬實作中，`DefDatabase`（或我們稱之為 `DefinitionDatabase`）是一個核心單例類別，維護著各類 Def 的集合。我們的目標是在這裡實現： - **根據 Type 分類存儲** Def，例如 `DefinitionDatabase.GetDefinition<ThingDef>(string defID)` 可以透過類型和名稱找到特定定義 [21]。 - 提供方法在載入時批量登錄 Def，清除所有 Def 等等 [22]。

`Definition` 應作為所有具體 Def 類別的基類，包含通用欄位如 `defID`（對應 RimWorld 的 defName）、`label`、`description` 等。從我們的實作片段可見，解析器會將 `<defID>` 節點讀取到物件的 `defID` 欄位 [23]。同樣地，未來我們會擴充 `Definition` 基類以包含繼承和引用所需的欄位。例如，可以在 `Definition` 中增加： - `parentName` 字串欄位，儲存 ParentName（如果有）。 - `abstract` 布林欄位，標識是否抽象。 - （如果需要）一個集合，用於暫存還未解析的引用（比如引用其他 Def 的 defID 列表），供稍後解析。

解析器處理繼承與 Abstract

我們可以在 XML 解析階段就捕獲 `ParentName` 和 `Abstract` 資訊。例如：

```
protected virtual void DeserializeCommon(XElement element, T def)
{
  def.defID = element.Element("defID")?.Value;
  def.label = element.Element("label")?.Value;
  def.description = element.Element("description")?.Value;
  // 新增: 捕捉 ParentName 屬性與 Abstract
  XAttribute parentAttr = element.Attribute("ParentName");
  if(parentAttr != null)
    def.parentName = parentAttr.Value;
  XAttribute abstractAttr = element.Attribute("Abstract");
  if(abstractAttr != null)
    def.abstract = (abstractAttr.Value.ToLower() == "true");
}
```

如上，在我們的 `DefinitionDeserializerBase` 共用解析程式中，額外讀取 XML 節點上的屬性值。這樣，每個 Def 物件就會帶有是否抽象、父名稱等元數據。

繼承的套用不建議在第一次讀取時立刻完成，因為父級內容可能在後續文件才出現。因此最佳時機是在**所有 Def 讀入並存入 DefinitionDatabase 後**。可以實作例如 `ModManager.FinishLoading()` 或 `DefinitionDatabase.ResolveInheritance()` 方法，在那裡：

- **遍歷**目前存儲的所有 Def：

- 對於每個 Def，如果 `parentName` 不為空，則執行繼承：
    - 使用 `DefinitionDatabase.GetDefinition<DefType>(parentName)` 找到父物件。其中 DefType 可以通過當前 def 的類型推斷（比如正在處理 ThingDef，就尋找 ThingDef 類型的 parent）。
    - 若找不到父，則記錄錯誤並跳過（或拋出異常，中止載入）。
    - 若找到，將父物件的所有可繼承欄位複製到子物件中。這部分可以利用反射或手動實作複製。例如，可將 Def 類別的屬性（排除 defID/label/description 等識別或描述性欄位）遍歷比較，子物件該屬性為預設值則由父填入，或者更直接地，建立一個 Merge 方法在 Def 類別中專門處理繼承邏輯。
    - 特別處理列表：如果遇到列表屬性，要按照是否 Inherit 來決定合併還是替換。Inherit 屬性的資訊我們需要在解析 XML 時讀取。由於我們的解析架構目前沒有方便地讀取任意節點屬性（如 `<weaponTags Inherit="False">`），一種方案是在 **XML 讀取階段** 直接處理此情況：當 DeserializeSpecific 遇到列表節點時，檢查是否有 Inherit="False"，並將該資訊存入一個臨時結構。實作上，可以將列表讀取的程式擴充，使其返回一個包含列表和一個標記的對象。或者更簡單地，我們可以在**套用繼承**時，如果偵測子 Def 中某列表明確不想繼承（例如我們可以為 Def 類別的列表欄位設計成特殊型別，持有一個 `bool inherit` 屬性），則在合併時跳過父列表。
    - 若父級本身有父，則先遞歸解決父級的繼承（確保祖先鏈條都展開），再套用到子級。要注意避免循環繼承（可以藉由記錄拜訪過的節點來檢測迴圈並報錯）。
- **從資料庫移除 Abstract**：在繼承處理完畢且不再需要父模板後，將所有 `def.abstract == true` 的物件過濾掉，不納入最終可用清單。這樣 `DefinitionDatabase` 裡對外提供查詢的就只有實體 Def 了。我們可以保留一個開發模式開關來檢視Abstract列表，以便調試，但對遊戲功能來說它們已經無意義。

解析跨定義引用

在 Def 完成繼承、內容確定後，就需要解析各種跨引用，使 Def 之間真正連結起來。實作上，可以在與繼承相鄰的流程中一併完成，也可以單獨設置一個 `ResolveCrossReferences()` 步驟。

- **尋找引用欄位**：最直接的方法是透過**約定**：例如我們知道 `ThingDef` 類別裡，所有類型為 `Def` 或其子類的欄位都應該被解析。例如 `ThingDef.soundInteract` 類型是 SoundDef，那麼它的值應該是已載入的 SoundDef 物件，而不是殘留字串。
- 我們可以在 ThingDef 的解析時，暫時將 `<soundInteract>` 的值存在 ThingDef 的某個暫存字串欄位，如 `soundInteractID = "Sword_SheathSound"`。然後在解析引用階段，用這個字串去 `DefinitionDatabase.GetDefinition<SoundDef>("Sword_SheathSound")` 找到對象，賦給 `soundInteract` 欄位。
- 更通用的做法是使用**反射**：掃描所有 Def 物件的公共欄位，凡是類型繼承自 `Definition` 的欄位，都嘗試將目前儲存的字串轉為引用。同理，若欄位是 `List<T>` 且 `T` 繼承 `Definition`，則迭代解析每個元素名為 Def。

- 我們也可以通過**特性（Attribute）標註**的方式，來標記哪些欄位需要進行引用解析，然後在解析時有選擇地處理。

- **解析與賦值**：根據上述定位到的字串，利用 DefDatabase 拿到相應 Def 物件。如果找到則賦值，如果找不到就記錄錯誤。如前所述，請提供足夠的錯誤資訊，例如：「某Def[defID] 的字段 X 無法解析 'Y'，找不到對應的定義。」。

- **清理暫存**：若我們引入了暫存的字串欄位來保存引用名，解析完可以選擇清空或移除，避免佔用記憶體且防止誤用未轉換的值。

完成這一步，Def 之間的關聯就和 RimWorld 遊戲內一致了：我們可以在 SteelSword 的定義物件中直接訪問到 `soundInteract.clipPath` 等屬性，而不是一個字串。這對後續的模擬系統（例如物品生成、音效觸發）都至關重要。

資料夾結構與檔案組織建議

按照 RimWorld 模組的慣例，一個模組中 Def 檔案通常放置於 `Mods/<你的Mod>/Defs/` 資料夾下。為了良好的組織和與未來擴充性的兼容，我們建議：

- **按類型分類 Def 檔案**：可以在 Defs 資料夾下建立子資料夾，如 `ThingDefs/`, `RecipeDefs/`, `SoundDefs/` 等，將對應類型的定義檔分開存放。這不僅清晰，也避免單一巨大的 XML 造成管理困難。
- **抽象模板集中管理**：如果你的模組定義了許多抽象 Base Def，可考慮將它們放在一個專門的檔案（或數個檔案）中，命名上可以加前綴如 `_BaseDefs.xml` 以確保它們相對較早被讀取。雖然我們的系統並非依賴檔案名順序（會整體處理繼承），但這樣做利於人工檢視和維護。
- **命名規範**：defName/defID 最好帶有模組或功能前綴，避免與遊戲核心或其他模組撞名。例如前述 **SteelSword** 可以改為 **MyMod_SteelSword** 作為 defName。抽象 Def 如 BaseMeleeWeapon 也可命名為 **MyMod_BaseMeleeWeapon** 以降低衝突風險。
- **About.xml 相依聲明**：當模組 A 需要引用模組 B 的 Def 時，務必在模組 A 的 About.xml 中加入對模組 B 的依賴宣告（例如 `<li>ModB_PackageId</li>`）。這確保 ModManager 載入順序正確。否則即便我們的解析器足夠智能，也無法處理「未載入不存在的 Def」。

練習題與驗收檢核

為了幫助鞏固您對 Def 繼承與跨定義引用的理解，請嘗試完成以下練習，並對照檢核項目自我評估：

練習 1 – 定義繼承階層：
創建一組武器相關的 Def： 1. 定義一個抽象 `ThingDef` 作為武器基底，例如 **BaseGun**，包含通用屬性（如射程、價值等）。 2. 定義另一個抽象 `ThingDef` **BaseRifle**，使其 `ParentName="BaseGun"` 並額外添加步槍類武器特有的屬性（如更高射程）。 3. 定義一個具體武器 **LaserRifle**，`ParentName="BaseRifle"`，設定其 `defName`、`label`，以及改寫部分屬性（比如射程、更換圖像路徑等）。 4. 問題：在 LaserRifle 中不想繼承 BaseRifle 的某個列表（比如 `weaponTags`），應如何處理？

驗收檢核： - 是否成功建立多層繼承（BaseRifle 繼承 BaseGun，LaserRifle 繼承 BaseRifle）？ - LaserRifle 是否正確取得了來自 BaseGun 和 BaseRifle 的所有必要屬性值？ - LaserRifle 中你有意排除的列表內容是否透過 `Inherit="False"` 如預期地沒有繼承父級項目？

練習 2 – 跨 Def 引用與錯誤測試：
1. 定義一個新的 `SoundDef`，例如 **LaserShotSound**，模擬雷射射擊的音效（不要求真有音檔，可寫個虛擬路徑）。 2. 在前述 **LaserRifle** 的定義中，增加一個欄位（如果不存在則假設有）`<soundCast>LaserShotSound</soundCast>`，表示開火音效引用。 3. 運行你的模擬系統，觀察是否 LaserRifle 的 soundCast 成功解析為 LaserShotSound 的物件。 4. 接著，故意將 LaserRifle 的 soundCast 值改成一個不存在的名稱，例如 `NonExistSound`，再次載入檔案，檢查系統日誌。

驗收檢核： - LaserShotSound 是否正確載入至 SoundDef 資料庫，LaserRifle 的對應欄位引用是否成功解析？ - 刻意的錯誤情境下，系統是否輸出明確的「無法解析跨引用」錯誤，指出找不到 `NonExistSound`？ - 如果 LaserRifle 的某個引用目標在另一模組，當不載入該模組時，系統是否也能給出類似的錯誤提示？

練習 3 – 閱讀與理解日誌：
利用模擬系統載入你創建的 Def，開啟詳細日誌模式（如果有），回答下列問題： - 在繼承處理過程中，系統列出的日誌順序是否與我們描述的步驟一致？例如，是否先看到 "Applying inheritance for BaseRifle -> BaseGun" 等消息，然後才出現 "Resolving cross-references for LaserRifle" 等？ - 當你移除 BaseGun 定義或拼錯名字，再載入時，系統響應如何？有沒有及時報出父類不存在的問題？ - 整體載入完成後，嘗試呼叫 `DefinitionDatabase.GetDefinitions()`（或等價方法）檢視返回的結構，確認其中沒有 Abstract 的定義殘留，且各 Def 數量與你的文件定義數量一致。

完成以上練習並通過檢核項目後，表示你已充分掌握 RimWorld 模組系統中的 Def 繼承與跨定義引用機制。在之後的開發中，請繼續遵循良好實踐，例如合理利用繼承避免重複、謹慎處理跨模組引用與依賴，這將使你的模組架構更健壯且易於維護。祝你在 RimWorld 模組創作之旅中玩得開心、學得愉快！ [1] [14]

---

[1] [2] [3] [5] [7] [9] [10] [17] [19] [20] Modding Tutorials/XML file structure - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/XML_file_structure

[4] [6] Hi, OP here, I'm delivering you my Beginners Guide To Modding. : r/RimWorld
https://www.reddit.com/r/RimWorld/comments/olayyw/hi_op_here_im_delivering_you_my_beginners_guide/

[8] XML Inheritance - RimWorld Modding Wiki
https://rimworldmodding.wiki.gg/wiki/XML_Inheritance

[11] [12] [13] Modding Tutorials/Weapons Guns - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/Weapons_Guns

[14] [16] Modding Tutorials/Troubleshooting - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/Troubleshooting

[15] Help troubleshooting mods / debug log? - Steam Community
https://steamcommunity.com/app/294100/discussions/0/1700542332321973942/

[18] MODDERS PLEASE READ, Abstracts and how they should be used.
https://ludeon.com/forums/index.php?topic=19499.0

[21] [22] DefinitionDatabase.cs
https://github.com/angus945/moddable-study/blob/66d04a5d8bc56de2f5eb0d0c6725365dad9249dd/Assets/Script/ModInfrastructure.Core/Definition/DefinitionDatabase.cs

[23] DefinitionDeserializerBase.cs
https://github.com/angus945/moddable-study/blob/66d04a5d8bc56de2f5eb0d0c6725365dad9249dd/Assets/Script/ModInfrastructure.Core/ModManager/Deserializer/DefinitionDeserializerBase.cs