ChatGPT

ModContentPack 模組資源與程式碼分離教材

第一章：模組封裝架構概述

在本章中，我們將介紹 RimWorld 模組(Mod)封裝架構的概念，以及為何要將**資料**(Defs、資源檔案等)與**程式碼**分離封裝成模組。這將幫助我們在 Unity 專案中模擬類似 RimWorld 的模組系統，為遊戲添加可插拔的內容與功能。

一個 RimWorld 模組本質上是一個獨立封裝的內容包（Content Pack），其中包含了該模組所需的各種資源與程式碼。RimWorld 採用了**ModContentPack**類別來封裝每個模組，實現資料與程式的分離。具體而言，每個模組通常有自己獨立的資料夾，內含：定義檔(Def)資料、編譯後的程式碼(Assemblies)、以及圖像、音效等資源檔案 [1]。這種設計允許遊戲在不修改核心程式的情況下載入或移除模組內容，並確保不同模組之間**互不干擾**。程式碼與資源的隔離可以避免命名衝突，並提高模組開發與維護的便利性。

例如，在 RimWorld 的 Mods 資料夾中，每個模組都放在獨立的子資料夾下，其典型架構如下 [1]：

```
Mods/
 └─ MyModFolder/                ← 模組根目錄（名稱隨意）
     ├─ About/                  ← 模組資訊（必需）
     │   ├─ About.xml           ← 模組描述與定義檔案
     │   └─ Preview.png         ← 模組預覽圖片
     ├─ Assemblies/             ← 編譯後的 DLL 程式碼
     ├─ Defs/                   ← 定義(Def)檔案（XML 等）
     ├─ Textures/               ← 圖片資源
     ├─ Sounds/                 ← 聲音資源
     ├─ Patches/                ← Def 修改檔(XML Patch，可選)
     └─ Languages/              ← 語言翻譯檔（可選）
```

上述架構中，**About** 資料夾包含模組的描述與基本資訊（例如模組名稱、作者、版本相容性等），這讓遊戲可以識別該模組並決定是否載入 [2]。**Defs** 資料夾內則是各種遊戲**定義檔(Defs)**，這些XML定義描述了遊戲中新增的物件、物種、事件等內容。**Assemblies** 則包含模組的C#編譯程式碼（DLL檔案），RimWorld會自動載入此資料夾中的所有DLL [3]。**Textures**與**Sounds**資料夾分別存放該模組專用的圖形和音效資源檔案（例如 `.png` 圖片、`.ogg` 聲音檔）。**Patches**資料夾用於放置XML Patch檔，用以**修改**其他模組或核心遊戲的Def（本章節稍後不深入討論Patch）。**Languages**資料夾則包含多國語言的翻譯文本資源。

如此封裝的好處在於：每個模組的資源和程式碼都在自己的目錄下，彼此獨立，不會直接影響其他模組或遊戲本體。遊戲在載入時可以逐一掃描各模組資料夾，讀取其中的定義、載入其DLL，並將資源整合到遊戲中。

學習目標

- 理解將遊戲內容製作成獨立模組封裝的必要性與好處。
- 知道 RimWorld 模組典型的目錄結構與各子資料夾的用途。
- 了解**資料**與**程式碼**在模組中的分離概念，以及這種架構如何支援多模組共存。

技術重點

- **模組封裝**：掌握 RimWorld 模組（ModContentPack）的封裝概念，一個模組包含自身的Defs、資源及程式碼，以資料夾作為隔離單位 [1] 。
- **目錄結構**：熟悉標準模組目錄中的關鍵資料夾（About, Defs, Assemblies, Textures, Sounds 等）及其功能劃分。
- **內容隔離**：理解不同模組各自管理資源與程式碼的方式，避免因名稱或資源衝突造成的相互影響。

練習題

1. 思考如果沒有模組化架構，將所有擴充內容直接放入遊戲程式會產生哪些問題？
2. 嘗試說明 RimWorld 模組目錄中 **Defs**、**Assemblies**、**Textures** 三個資料夾分別存放何種內容，為什麼要將它們分開？
3. 若你要為自己的 Unity 遊戲設計模組系統，列舉出你認為每個模組資料夾中需要包含的項目，並解釋其用途。

範例檔案架構建議

- 建立一個名為 `Mods` 的根資料夾作為放置所有模組的目錄。在此資料夾下，每個子資料夾代表一個獨立模組，例如 `Mods/MyFirstMod/` 、`Mods/AnotherMod/` 等。
- 在每個模組資料夾中，建立上述介紹的子資料夾架構。例如在 `MyFirstMod` 中包含 `About/` （含有 About.xml）、`Defs/` 、`Assemblies/` 、`Textures/` 、`Sounds/` 等。
- 撰寫一個簡單的 `About.xml` 文件作為模組資訊描述，例如模組名稱、作者、版本需求等。此檔案可在後續章節中由載入程式讀取，用來識別模組。

上述檔案架構將作為後續章節模擬模組載入的基礎。我們會在 Unity 專案的執行時從這些資料夾中載入各模組的內容。

第二章：模組目錄結構與資源隔離

本章我們將更深入探討模組目錄結構中的各個部分，並說明如何透過目錄組織實現**資源**與**程式**在不同模組間的隔離。正確的目錄規劃能避免命名衝突，使多個模組能安全地共存於同一遊戲中。

模組目錄各部分說明

如前章所述，一個標準的模組資料夾包含多個子目錄，以下對主要部分進行說明：

- **About 資料夾**：**唯一必需**的子資料夾，包含模組的元資料。通常有一個 `About.xml` 檔案，定義模組名稱、描述、作者、版本相容性及其他依賴關係等資訊 [2] 。這使得遊戲可以識別模組，並在模組清單中顯示相關資訊。同一資料夾內通常還包含 `Preview.png` （預覽圖片）供玩家在模組列表中預覽模組圖示。

- **Defs 資料夾**：存放模組提供的**各類遊戲定義檔**(XML)。例如物品(Item)、生物(Creature)、建築(Thing)等各種 `Def` 定義都放在此處。如果模組不新增任何遊戲定義，可以沒有此資料夾 [4] 。檔案命名可自由，但通常依據內容類型分類到子資料夾（例如 `ThingDefs/` 、`RecipeDefs/` 等）以方便組織與維護 [5] 。遊戲在載入時會讀取每個模組中 Defs 資料夾下的所有 XML，並解析為相應的遊戲內部物件。

- **Assemblies 資料夾**：存放模組的**程式碼組件**(C# 編譯後的 DLL 檔案)。若模組沒有自訂程式碼，可以省略此資料夾 [6] 。RimWorld在啟動時會自動載入每個模組 Assemblies 資料夾中的 .dll 檔案 [3] 。正確

編譯的DLL須以 RimWorld 或 Unity 提供的API為參考編譯，放入此處後遊戲即可在載入模組時動態載入這些額外的類別與功能。

- **Textures 資料夾**：模組的**圖像資源**(通常為 `.png`)存放處。如果模組不包含自訂圖片可以沒有此資料夾 [7]。RimWorld 的做法是根據圖像檔案的**路徑**來尋找材質，例如 Def 定義中引用某圖片時，會在載入過程中依據相對路徑匹配相應的檔案。需要注意的是，不同模組的紋理檔案路徑若相同，載入時後面的模組將覆蓋前者的圖像 [8]。為避免衝突，建議**在 Textures 下以模組名稱或其他唯一字串建立子資料夾**來命名路徑 [9]。例如，模組 "MyMod" 有一張 `MyIcon.png` 圖片，最好置於 `Textures/MyMod/MyIcon.png`，而非直接放在 Textures 根目錄 [9]。如此可降低不同模組之間圖像檔名相同導致覆蓋的風險。

- **Sounds 資料夾**：模組的**音效資源**(通常為 `.ogg`、可支援 `.wav`/`.mp3`)存放處。同樣地，若無音效可省略 [10]。音效檔的載入與識別也基於路徑。如果兩個模組包含相同路徑名稱的音效檔，後載入的模組會覆蓋前一個的音效 [11]。因此亦建議在 Sounds 下使用模組名稱作為子資料夾或在檔名加前綴，以確保路徑獨一無二 [12]。例如，把音檔放在 `Sounds/MyMod/Explosion.ogg` 而不是直接 `Sounds/Explosion.ogg`。

- **Patches 資料夾**：用於存放XML Patch檔（**可選**）。XML Patch是 RimWorld 提供的一種修改Def的機制，允許模組在不直接複寫原檔的情況下修改其他模組或核心遊戲的定義。若不涉及Def修改則不需要此資料夾 [13]。本系列教程聚焦於模擬模組載入與資源管理，因此Patch細節不深究。

- **Languages 資料夾**：翻譯檔案存放處（**可選**）。模組若要支援多語系，可在此放置不同語言的翻譯XML。

資源與程式碼的隔離策略

透過上述目錄結構，不同模組的內容天然地被**檔案路徑**區隔開。為進一步確保隔離性，我們應採取一些**命名與結構**上的最佳實踐：

1. 資源命名空間：正如前述，圖檔與音檔應放置在以模組識別的子資料夾中，形成獨特的路徑前置字串 [12][9]。例如兩個模組都各自有名為 "Fireball.png" 的圖片時，若它們存放在 `Textures/ModA/Fireball.png` 與 `Textures/ModB/Fireball.png` 路徑下，則載入時透過完整路徑區分 ModA 和 ModB 的圖像，避免衝突。RimWorld 在載入資源時採用「後載覆蓋前載」的策略，如果路徑相同則以最後載入的模組為準 [11]。因此獨特路徑能防止非預期的覆蓋。除了使用子資料夾命名空間，也可以在檔名上加入模組代號前綴（例如 `ModA_Fireball.png`），但使用子資料夾方式在組織上更清晰。

2. 程式碼命名空間：在C#程式碼中，**Namespace**（命名空間）是區分類別的主要方式。每個模組的程式碼建議使用獨立的命名空間（通常以模組名稱作為namespace，例如 ModA 的程式碼放在 `namespace ModA {...}` 中）。這可確保不同模組中即使有類別名稱相同，也不會互相衝突。此外，在編譯DLL時，將每個模組的組件命名為不同的assembly名稱（例如 "ModA.dll" 與 "ModB.dll"）也可避免載入時的混淆。如果兩個DLL包含完全相同的命名空間和類別（極少發生除非惡意），.NET CLR 載入時也會區分不同程式集(Assembly)，因此一般不至於類別衝撞，但**良好的命名空間習慣**仍是必要的。

3. 檔名唯一性：除了資源檔，Def 定義檔(XML)和DLL檔名最好也維持唯一清晰。例如 Def XML 檔的檔名可以包含模組名稱或內容描述，DLL 檔名更需避免通用名稱。雖然 RimWorld 對 Def XML 檔名不嚴格（檔名不影響載入，內容 `<DefName>` 才是關鍵），但為管理方便仍建議合乎模組命名。DLL 則一定要使用獨有名稱，以免玩家手動管理模組檔案時發生覆蓋錯誤。

學習目標

- 理解模組各子資料夾的具體作用，能正確將不同類型的內容分類存放。
- 掌握避免不同模組間資源檔案**命名衝突**的方法，例如使用模組命名空間或前綴。
- 瞭解在C#程式碼層面透過命名空間與組件名稱，實作多模組程式碼的隔離，防止類別定義衝突。

技術重點

- **模組目錄規範**：所有關鍵子資料夾名稱需嚴格按規範命名（區分大小寫），例如 "Defs"、"Textures" 等不可拼寫錯誤，否則遊戲無法識別 [14]。
- **檔案路徑覆蓋規則**：了解遊戲載入多個模組時的資源覆蓋規則——相同路徑的資源以最後載入者為準 [11]。
- **避免衝突策略**：強調在**資源**上使用子資料夾或檔名前綴 [12] [9]、在**程式碼**上使用唯一命名空間/組件名，作為避免衝突的核心手段。

練習題

1. 在假想 scenario 中，有兩個模組都提供了一個檔名為 "Explosion.ogg" 的音效，你會如何安排它們在各自模組的資料夾中，以避免載入時的混淆或覆蓋？
2. 如果兩個模組都需要擴充一個名為 "Fireball" 的遊戲功能（可能各自寫了一個 `Fireball` 類別），說明透過 C# 命名空間可以如何讓兩者共存而不衝突。
3. 觀察上節所建立的模組目錄架構，檢查每個資料夾名稱拼寫是否正確無誤（如 Defs 而非 Def 或其他變體）。思考為什麼大小寫和拼字必須精確一致。

範例檔案架構建議

- **模組範例結構**：建立兩個範例模組資料夾以測試隔離效果。例如 `Mods/ModA/` 與 `Mods/ModB/`：
- 在 `ModA` 的 Textures 資料夾下建立子資料夾 `ModA` 並放入 `Icon.png` 圖檔；在 `ModB` 的 Textures 下建立 `ModB` 資料夾放入同名的 `Icon.png`。藉此模擬兩模組具有同名資源的情況。

- 在 `ModA` 和 `ModB` 各自的 Assemblies 資料夾放入命名不同的 DLL（可為空殼DLL，但命名為 ModA.dll / ModB.dll），各包含一個名稱相同但命名空間不同的類別（例如 ModA.Core.ClassX vs ModB.Core.ClassX）。

- **檔案範例**：準備兩個簡單 Def XML 檔案，皆命名為 `ExampleDef.xml`，分別置於 ModA 和 ModB 的 Defs 資料夾。檔案內容可以只是表示不同模組的Def名稱（例如 `<defName>ModA_Item</defName>` vs `<defName>ModB_Item</defName>`）。這將測試遊戲載入時如何區分兩者。

透過上述範例架構，您可以在後續章節的載入程式中驗證：相同名稱的資源是否因目錄區隔而正確載入，以及模組間程式碼與Def是否各自分開處理。這些準備將為我們進入模組載入機制的實作奠定基礎。

第三章：模組載入機制與 ModContentPack 模擬

有了模組的封裝結構後，下一步就是讓 Unity 專案在執行時能**發現並載入多個模組**。本章我們將設計一個**模組管理器(Mod Manager)**來掃描預先定義的模組資料夾，並利用自訂的 `ModContentPack` 類別模擬 RimWorld 載入模組的流程與登錄機制。

掃描與發現模組

首先，我們需要決定 Unity 專案中模組資料夾的位置。在開發環境下，可以將 Mods 資料夾放在 Unity 專案的 `StreamingAssets`、`Persistent Data Path` 或乾脆與可執行檔同級目錄下，以便在遊戲執行時讀取其內容。為簡化起見，我們假設在遊戲執行時，可以透過一個已知路徑取得 Mods 目錄。例如：

```
string modsPath = Path.Combine(Application.dataPath, "Mods");
// 或 Application.persistentDataPath 等
```

接著，使用 `System.IO.Directory.GetDirectories(modsPath)` 列出該目錄下的所有子資料夾，每個子資料夾即代表一個模組。對於每個發現的模組資料夾，我們將進行以下步驟：

1. **建立 ModContentPack 實例**：我們定義一個類別 `ModContentPack` 來表示模組內容包。當發現一個模組資料夾時，即可初始化一個 `ModContentPack` 物件，並存儲該模組的基本資訊（例如路徑、名稱）。稍後我們將詳細定義該類別的內容。

2. **讀取模組資訊**：載入該資料夾中的 About.xml（如果存在），以取得模組名稱、版本等資訊，存入 `ModContentPack`。如果 About.xml 缺失，可以用資料夾名稱作為模組名稱的後備方案，並給予警告。

3. **載入組件(Assemblies)**：檢查 Assemblies 資料夾，將其中的所有 `.dll` 檔動態載入（使用 `Assembly.LoadFile` 或 `Assembly.LoadFrom`）。這會讓模組的C#類別進入遊戲的應用程式域(AppDomain)，以供稍後使用。載入組件時，要注意**錯誤處理**：若 DLL 不相容或出錯，需捕捉異常並記錄，但不影響其他模組繼續載入（錯誤隔離詳見第八章）。

4. **紀錄模組**：將初始化完成的 `ModContentPack` 實例加入全域的模組清單，例如在一個 `ModManager` 類別中維護一個 `List<ModContentPack> LoadedMods`。

透過上述流程，我們實現了類似 RimWorld 中 `ModLister.RebuildModList()` 和 `LoadedModManager.LoadModContent()` 的步驟[15][16]：即**建立模組列表並載入其程式內容**。在RimWorld中，這過程除了載入DLL外還會排程資源載入等，我們的模擬稍後會涵蓋。

設計 ModContentPack 類別

`ModContentPack` 類別的作用在於封裝一個模組的所有資訊與功能。我們可以在Unity專案中建立一個類別，如下：

```
public class ModContentPack {
  public string FolderPath;  // 模組資料夾的路徑
  public string Name;        // 模組名稱
  public Assembly[] Assemblies; // 載入的組件陣列
  // 可擴充更多欄位，例如版本、作者、Def列表、資源緩存等
  public ModContentPack(string folderPath) {
    FolderPath = folderPath;
    Name = Path.GetFileName(folderPath); // 預設以資料夾名作模組名
  }
}
```

當 ModManager 掃描到一個資料夾時，就 new 一個 `ModContentPack(folderPath)`。之後可調用該物件的方法去載入具體內容，例如：

- `LoadAssemblies()`：掃描 `FolderPath/Assemblies` 下的 dll 檔並使用 `Assembly.LoadFrom` 載入，每載入一個就把 `Assembly` 實例存入 `Assemblies` 陣列中。**注意**：Unity 默認情況下允許載入 managed DLL（在Mono後端下可行），但若使用IL2CPP後端則無法在執行時載入新DLL，因此這裡假設我們在可進行反射的環境下執行（如編輯器或Mono平台）。

- `LoadDefs()`: 掃描 `FolderPath/Defs` 目錄下所有 `.xml` 檔案，利用我們在上一章或之前構建的 Def 解析系統將其轉換為 Def 物件。我們可以將解析出的 Def 物件清單保存到 ModContentPack 中，或直接註冊到全域 Def 資料庫中（可視需求決定）。為了模組間的獨立性，一個策略是**先暫存在各自的 ModContentPack**，待所有模組的Defs都讀取完畢後，再統一合併注入全域（這樣可以處理跨模組引用或依賴，具體見第六章）。

- `LoadResources()`: 掃描 Textures、Sounds 等資料夾，如果我們採用自訂讀取方式，則在此載入資源（或者記錄可用的資源路徑，稍後按需載入）。這部分我們在第五章詳細討論。

此外，可在 ModContentPack 設置一些輔助方法，例如 `GetDef<T>(string defName)` 來在該模組範圍內查詢Def，以利除錯或特殊用途。

Mod 管理器 (LoadedModManager) 流程

在 RimWorld，LoadedModManager 的初始化流程中，會依序呼叫載入組件和建立 Mod 類別實例等操作[16]。我們可以在 Unity 的遊戲啟動腳本（例如一個 GameManager 的 `Start()` 方法）裡撰寫類似的程式碼：

```
void LoadAllMods() {
  string modsPath = Path.Combine(Application.dataPath, "Mods");
  if (!Directory.Exists(modsPath)) return;
  foreach (string modDir in Directory.GetDirectories(modsPath)) {
    try {
      ModContentPack mod = new ModContentPack(modDir);
      mod.LoadAssemblies();
      mod.LoadDefs();
      mod.LoadResources();
      LoadedMods.Add(mod);
      Debug.Log($"Mod loaded: {mod.Name}");
    } catch(Exception e) {
      Debug.LogError($"Failed to load mod at {modDir}: {e.Message}");
      // 這裡捕捉異常以防止單一模組出錯時中斷整個載入流程
    }
  }
}
```

上述邏輯將逐一載入 Mods 目錄下的模組。`try-catch` 區塊確保任何一個模組發生錯誤時不影響後續模組載入（這是**錯誤隔離**的實作之一）。載入過程中使用 `Debug.Log` 提示方便觀察哪個模組已成功載入。

這樣的架構已經搭建起模組系統的骨架：找到模組、封裝模組、並載入其基本內容。接下來的章節，我們將討論如何在載入程式碼後**調用模組自身的初始化邏輯**、如何載入其資源以及整合Def等細節。

學習目標

- 了解**掃描模組資料夾**並建立模組列表的基本流程。
- 能夠設計一個類似 RimWorld `ModContentPack` 的類別，封裝模組的關鍵資訊和載入功能。
- 學習如何在 Unity 中動態載入外部 DLL 組件，以及基本的錯誤處理方式，確保模組載入過程穩健。

技術重點

- **Directory API**：運用 `Directory.GetDirectories` 等方法掃描檔案系統，以發現外部模組。
- **反射載入**：使用 `Assembly.LoadFile/LoadFrom` 在執行期載入外部程式集(僅適用於Mono後端)；理解 Unity 不同後端對動態載入DLL的限制。
- **模組資料結構**：定義 `ModContentPack` 類別，以物件導向方式管理模組資訊（如路徑、名稱、已載入的Def清單等），為後續操作提供便利。
- **例外處理**：在模組載入各環節使用 try-catch，做到單一模組出錯時不會終止整體載入流程，符合模組錯誤隔離的要求。

練習題

1. 撰寫一個簡單的 `ModContentPack` 類別（不需要完整功能），至少包含：模組名稱、資料夾路徑，以及一個方法用於載入該模組的 DLL。嘗試在 Unity 中呼叫此方法載入一個預先編譯的測試 DLL。
2. 如果 `Mods` 資料夾下有三個子資料夾，但其中一個沒有 About.xml 檔案，設計你的載入流程該如何處理這種情況？（提示：可使用資料夾名稱或跳過載入並發出警告）
3. 思考：假如模組之間存在相依關係（例如 ModB 需要在 ModA 之後載入），在目前的載入機制中會出現什麼問題？你能想到如何擴充 ModContentPack 使其包含排序或依賴資訊嗎？

範例檔案架構建議

- **模組管理腳本**：在 Unity 專案的 `Assets/Scripts/Modding/` 資料夾下，建立 `ModManager.cs` 腳本，其中包含 `LoadAllMods()` 方法（如上所示）。這個腳本可以掛載在啟動場景中的一個空物件上，或作為遊戲單例管理器使用。

- **模組內容類別**：在同一資料夾下建立 `ModContentPack.cs`，定義前述的類別結構和方法（如 `LoadAssemblies`, `LoadDefs` 等）。確保引用 `System.Reflection` 和 `System.IO` 命名空間以使用反射和檔案操作功能。

- **測試 DLL**：建立兩個簡單的 C# 類別庫專案對應於前述範例模組 ModA、ModB，各自定義一些簡單的類別或方法（例如輸出一行日誌）。編譯後將生成的 ModA.dll 和 ModB.dll 放入 Unity 專案 `Mods/ModA/Assemblies/` 與 `Mods/ModB/Assemblies/` 中。這將用於測試 `LoadAssemblies()` 是否成功載入模組程式碼。

完成以上準備後，執行遊戲，應能看到 `ModManager` 列印出發現並載入的模組名稱。如果有模組缺少 About.xml 或 DLL 載入失敗，也應在控制台顯示相應錯誤訊息，證明我們的模組載入骨架已經運作。

第四章：模組初始化點設計 (IModInitializer)

載入模組的DLL只是第一步，接下來需要考慮**如何讓模組的程式碼在遊戲啟動時執行**其初始化邏輯。例如，模組可能需要在載入時註冊事件、增加遊戲物件或設定預設值。在 RimWorld 中，當載入一個模組的程式集後，如果該程式集中存在繼承自 `Verse.Mod` 的類別，遊戲會自動建立該類別的實例[17]。模組開發者通常透過這個類別的建構子或覆寫方法來執行初始化工作。

在我們的 Unity 模擬中，我們將引入**IModInitializer 介面**來實現類似的機制。每個模組如果需要在載入時執行程式邏輯，就在它的DLL中實作一個 `IModInitializer` 介面，並在遊戲載入時讓 ModManager 掃描並執行它。

定義 IModInitializer 介面

在 Unity 專案中建立一個介面定義檔（可放在 Modding 資料夾中）：

```
public interface IModInitializer {
  void Initialize(ModContentPack mod);
}
```

這個介面包含一個 Initialize 方法，我們約定：模組若需要初始化，應提供一個類別實作此介面。實作時可以選擇接受 ModContentPack 參數，這樣在初始化時可以知道自己隸屬哪個模組以及存取該模組的相關資訊（如資源路徑等）。

模組開發者在撰寫自己的模組程式碼時，例如可以這樣做：

```
// ModA.dll 內的一個類別
using MyGame.Modding;  // 假設 IModInitializer 定義在此命名空間
public class ModA_Init : IModInitializer {
  public void Initialize(ModContentPack mod) {
    // 例如：登錄一個自訂指令或初始化模組設定
    UnityEngine.Debug.Log($"[{mod.Name}] 模組初始化執行");
    // ...其他初始化邏輯...
  }
}
```

假設 ModA.dll 參考了我們遊戲的 API，其中包含 IModInitializer 介面和 ModContentPack 類別定義。那麼在載入 ModA.dll 並反射出其中的類別時，我們就能找到此 ModA_Init，並調用其 Initialize 方法。

掃描並執行模組初始化

我們需要修改前章的載入流程，在載入組件之後、將模組加入列表之前，增加初始化呼叫步驟：

```
// After mod.LoadAssemblies();
foreach (Assembly asm in mod.Assemblies) {
  foreach (Type type in asm.GetTypes()) {
    if (typeof(IModInitializer).IsAssignableFrom(type) && !type.IsInterface && !type.IsAbstract) {
      // 找到實作 IModInitializer 的具體類別
      try {
        IModInitializer initializer = (IModInitializer)Activator.CreateInstance(type);
        initializer.Initialize(mod);
      } catch (Exception ex) {
        Debug.LogError($"Mod {mod.Name} IModInitializer 啟動失敗: {ex}");
      }
    }
```

```
    }
}
```

此段程式會對模組DLL中每個型別進行檢查：凡是非介面、非抽象類別，且實作了 IModInitializer 介面的，都會被創建實例並調用 `Initialize(mod)`。如此，每個模組都能在載入時執行自己的初始化邏輯。

為避免重覆執行，建議**每個模組DLL**只實作一次 IModInitializer（例如以模組為單位集中在一個初始化類別內）。如果多次實作也不會造成錯，但可能邏輯上不便管理。

與 RimWorld Mod 類別的對應

透過 IModInitializer，我們實現的效果與 RimWorld 中 Mod 類別相似。在 RimWorld，模組DLL中如果有繼承自 `Verse.Mod` 的類別，會在載入時被實例化，其建構子帶有一個 `ModContentPack` 參數供使用 [17]。典型範例如下 [17]：

```
public class MyFirstMod : Mod {
  public MyFirstMod(ModContentPack content) : base(content) {
    Log.Message("My First Mod has been loaded!");
  }
}
```

我們的設計中，IModInitializer.Initialize 方法相當於 RimWorld 的 Mod 類別建構子。不同的是，我們不要求繼承特定基類，只需實作接口即可，靈活度更大一些。另外，在 RimWorld 裡 Mod 類別也常用來提供設定介面等功能，我們此處重點在初始化，因此簡化為一個接口方案。

多模組初始化順序

預設情況下，我們按照模組被發現的順序進行初始化（通常是檔名排序順序）。若存在模組依賴性（例如 ModB 需要在 ModA 之後才能正確初始化），應在 About.xml 或其他配置中提供**LoadAfter**資訊。然而本模擬中未深入依賴管理（可作為進階練習），目前假設模組順序不影響獨立初始化。若需要，可在讀取 About.xml 時解析依賴資訊，並在執行 Initialize 前對 LoadedMods 列表排序。

學習目標

- 學會為模組提供一個**入口點**進行程式初始化，確保模組的程式碼在載入後能參與遊戲運作。
- 瞭解如何使用**反射**尋找並實例化特定接口的實作類別，以呼叫外部模組的程式功能。
- 對比 RimWorld 的 Mod 類別機制，加深對模組初始化時機與作用的理解。

技術重點

- **介面設計**：IModInitializer 的介面定義，以及選擇在 Initialize 傳入 ModContentPack 以提供模組環境資訊。
- **反射操作**：透過 `Assembly.GetTypes()` 列出類型並以 `typeof(IModInitializer).IsAssignableFrom(type)` 判斷接口實作；使用 `Activator.CreateInstance` 建立物件並調用接口方法。
- **錯誤處理**：在初始化呼叫時，同樣需要 try-catch 防護，以防止個別模組初始化邏輯的錯誤影響主程式穩定。

練習題

1. 為前一章製作的測試模組DLL（ModA.dll 或 ModB.dll）添加一個實作 IModInitializer 的類別，例如 `InitClass`，讓它的 Initialize 方法輸出一行包含模組名稱的日誌。重新執行遊戲，確認該日誌有成功顯示。
2. 如果某個模組的 IModInitializer.Initialize 中拋出異常，但我們的ModManager未做防護，會發生什麼情況？如何改進代碼避免這種問題？
3. 思考模組初始化順序的重要性：在沒有特別處理依賴的情況下，假如ModB需要引用ModA在Initialize中創建的資料，會出現什麼問題？你認為可以如何讓ModB等待ModA初始化完成再執行？（提示：或許透過配置依賴順序，或在Initialize中自行檢查依賴模組狀態）

範例檔案架構建議

- **介面與管理程式更新**：在 `Assets/Scripts/Modding/` 下新增 `IModInitializer.cs` 定義介面；修改 `ModManager.cs` 的 `LoadAllMods()` 或相關流程，納入對 IModInitializer 的掃描與執行（如前述代碼範例）。建議將該段邏輯提取為 `InitializeMods()` 函數，以清晰表達步驟。

- **模組代碼調整**：更新之前建立的測試模組專案（ModA, ModB）程式碼，引入我們遊戲專案的 Modding 程式集（引用 IModInitializer 介面和 ModContentPack 類別）。在各自DLL中新增實作 IModInitializer 的類別。例如：

- ModA 專案中新增 `public class ModAInitializer : IModInitializer { ... }`
- ModB 專案中新增 `public class ModBInitializer : IModInitializer { ... }`

並在 Initialize 方法中以 `Debug.Log` 輸出文字（包含模組名稱）。重新編譯 DLL 並覆蓋到 Mods 資料夾下。

- **測試與驗證**：執行 Unity 專案，查看日誌。預期會看到類似「

$$ModA$$

模組初始化執行」和「

$$ModB$$

模組初始化執行」的日誌訊息，順序取決於掃描模組的順序。如此驗證我們的模組初始化機制成功運作。

第五章：載入模組資源並註冊至全域資料庫

在成功載入模組的程式碼和 Def 定義後，遊戲還需要將模組的**圖像、音效等資源**載入，使之能在遊戲中被使用（例如物件顯示正確的圖標，事件播放正確的音效）。本章將討論如何為每個模組實現資源的載入，並將它們註冊到遊戲的全域資源管理中，以便 Def 或程式碼可以存取。

資源載入的時機與方式

根據 RimWorld 的流程，模組的資源（音頻、材質等）實際上是在Def載入之後、遊戲初始化過程的較後階段才真正載入 [18]。這是為了優化啟動時間以及確保Def解析期間不需要馬上用到具體資源。本模擬中，可以選擇兩種方式：

- **立即載入**：在模組載入時馬上讀取所有資源檔案到記憶體。這樣做的好處是實現簡單，載入後即可直接使用資源，但缺點是可能浪費記憶體（如果有些資源其實遊戲過程不會用到）且增加啟動時間。

- **延遲載入**：先記錄資源清單但不馬上讀取，等到資源真正需要使用時（例如物件要顯示圖標時）再讀取。此方式較複雜，需要資源存取有管理（如使用Lazy load或Asset bundle），但能節省初始載入成本。

為教學起見，我們採用**立即載入**實現方案：在模組載入時讀取該模組所有圖像與音效資源至記憶體。

載入圖像資源 (Textures)

Unity 載入外部圖像文件可透過 `Texture2D` 類別實現：

```
public void LoadTextures() {
  string texturesPath = Path.Combine(FolderPath, "Textures");
  if (!Directory.Exists(texturesPath)) return;
  foreach (string file in Directory.GetFiles(texturesPath, "*.png", SearchOption.AllDirectories)) {
    try {
      byte[] imageData = File.ReadAllBytes(file);
      Texture2D tex = new Texture2D(2, 2);
      tex.LoadImage(imageData);
      string assetKey = GetRelativePath(file, texturesPath); // 取得相對於Textures資料夾的路徑作為鍵
      // 將載入的Texture存入全域管理，例如一個字典：GlobalTextures[assetKey] = tex;
      GlobalTextureDatabase.Register(assetKey, tex, this);
    } catch(Exception e) {
      Debug.LogError($"載入圖像失敗: {file} - {e.Message}");
    }
  }
}
```

上述程式碼逐一尋找該模組 Textures 資料夾下的所有 `.png` 檔案，讀取其位元組資料，使用 Unity 提供的 `Texture2D.LoadImage()` 將PNG資料解碼為 Texture2D 物件。然後取得檔案相對於 Textures 根目錄的路徑作為此資源的識別鍵（例如 `MyMod/Icon.png` 或 `MyMod/SubFolder/Icon.png`）。最後，通過一個全域的 `GlobalTextureDatabase`（可以是我們自己建立的靜態類別或管理器）註冊該貼圖。這裡我們還傳入 `this`（ModContentPack），方便記錄該資源來源於哪個模組（如有需要時排除某模組時可用）。

GlobalTextureDatabase 可以是一個簡單的容器，例如：

```
public static class GlobalTextureDatabase {
  private static Dictionary<string, Texture2D> textures = new Dictionary<string, Texture2D>();
  public static void Register(string key, Texture2D tex, ModContentPack mod) {
    // 可以選擇性地在鍵前加上模組識別避免衝突，例如 key = mod.Name + "/" + key;
    if (!textures.ContainsKey(key))
      textures.Add(key, tex);
    else
      textures[key] = tex; // 如鍵已存在，後載入的覆蓋前者
  }
  public static Texture2D Get(string key) {
    textures.TryGetValue(key, out Texture2D tex);
    return tex;
```

```
    }
  }
```

此處採用了簡單的覆蓋策略：當不同模組註冊了相同鍵名的貼圖時，後註冊者覆蓋前者（這與 RimWorld 的行為一致[11]）。不過由於我們前面採取了資源路徑隔離（鍵名通常包含模組名稱前綴），一般情況下撞鍵應該不會發生，除非刻意覆蓋他人資源。

載入音效資源 (Sounds)

音效檔的載入在 Unity 中可使用 `UnityEngine.WWW`（舊版）或 `UnityEngine.Networking.UnityWebRequestMultimedia.GetAudioClip()` 來處理。例如：

```
public IEnumerator LoadSounds() {
  string soundsPath = Path.Combine(FolderPath, "Sounds");
  if (!Directory.Exists(soundsPath)) yield break;
  foreach (string file in Directory.GetFiles(soundsPath, "*.*", SearchOption.AllDirectories)) {
    if (!(file.EndsWith(".ogg") || file.EndsWith(".wav") || file.EndsWith(".mp3"))) continue;
    string url = "file://" + file;
    UnityWebRequest request = UnityWebRequestMultimedia.GetAudioClip(url,
AudioType.UNKNOWN);
    yield return request.SendWebRequest();
    if (request.result == UnityWebRequest.Result.Success) {
      AudioClip clip = DownloadHandlerAudioClip.GetContent(request);
      string assetKey = GetRelativePath(file, soundsPath);
      GlobalSoundDatabase.Register(assetKey, clip);
    } else {
      Debug.LogError($"載入音效失敗: {file} - {request.error}");
    }
  }
}
```

由於音效載入涉及非同步操作（UnityWebRequest），以上以協程方式示意。實務中可在ModManager中統一跑一個協程來載入所有模組音效，以確保載入過程不阻塞主執行緒。`GlobalSoundDatabase` 類似地可以用 Dictionary 存放 `AudioClip`，鍵為路徑鍵。值得注意的是，使用 `AudioType.UNKNOWN` 讓 Unity 自行判斷格式（或依副檔名給對應 AudioType）。這段程式碼會將 `.ogg`、`.wav`、`.mp3` 等支援格式的文件載入為 AudioClip。

如果不希望用協程，也可以對 `.wav` 使用 `WavUtility` 等自寫解析，或限制只使用OGG並採第三方 Ogg 解碼庫，但協程加 UnityWebRequest 是較簡單的方案。需要Unity 2017以上版本支援。

資源的全域註冊與使用

當資源載入完後，全域資料庫中就有了對應鍵的物件。接下來，**如何讓Def或遊戲系統使用這些資源**？

我們有幾種方式：

- **Def 直接參考**：在Def定義中，可能有欄位指定圖檔路徑或音效路徑。例如 Def 裡有 `<iconPath>MyMod/Items/Fireball.png</iconPath>`。在解析Def時或者在Def初始化完成後，我們可以將這個路徑對應的 Texture2D 從 GlobalTextureDatabase 中取出來，賦值給 Def 對象的對

應欄位（例如 `myDef.iconTexture = GlobalTextureDatabase.Get("MyMod/Items/Fireball.png");`）。這種方法需要在Def解析之後執行一次資源綁定操作，我們可以安排在模組全部載入完且Def合併進全域資料庫後進行。

- **遊戲邏輯請求**：遊戲的UI或邏輯在需要時主動請求，比如當需要顯示一個物品圖示時，透過物品的Def來找到圖示鍵再去 GlobalTextureDatabase 取圖（如果Def本身沒有存Texture引用）。這種做法彈性較高但每次使用都查找，效率稍低，可以在Def初始化時先綁定來優化。

建議在**Def初始化階段**就完成資源的綁定（除非某些大型資源想延遲載入），因為這樣後續使用更直接。比如在 `ModContentPack.LoadDefs()` 之後，可以馬上遍歷該模組所有解析出的 Def，將其中包含的資源路徑欄位替換為實際資源對象。這需要你解析Def XML時保留資源欄位值（如 iconPath 的 string），以及對應Def類別有可以存 Texture2D 或 AudioClip 引用的欄位。

模組資源與全域資料庫關聯

我們建立的 GlobalTextureDatabase/GlobalSoundDatabase 是全域共用的，所有模組的資源都放進去。為了管理和可能的**卸載**（如果需要禁用某模組時卸載其資源），我們在註冊時保存了來源模組引用。未來可以增加一個方法，例如 `UnloadByMod(ModContentPack mod)`，遍歷移除所有該 mod 加入的資源以釋放記憶體。當然，本模擬未深入實現卸載，但有了來源資訊就可以做到這一點。

學習目標

- 瞭解如何透過 Unity API 將外部圖像/音頻文件載入為遊戲可用的 `Texture2D` 或 `AudioClip`。
- 學會將載入的資源統一存放到**全域資料庫**中，並以關鍵字索引，以供遊戲各處取得資源。
- 能夠將 Def 定義中對資源的引用（如路徑）與實際載入的資源物件關聯，確保模組新增的內容能在遊戲中正常顯示/播放。

技術重點

- **檔案IO與轉換**：使用 `File.ReadAllBytes` 讀取圖檔，`Texture2D.LoadImage` 轉換為貼圖；使用 `UnityWebRequestMultimedia.GetAudioClip` 讀取音檔轉為音頻片段。
- **非同步載入**：音頻載入範例使用協程實現非同步，以避免主線程卡頓；圖像因資料量通常較小可同步載入，但也要注意過多大型圖可能導致卡頓，可依需要調整策略。
- **資料庫實作**：全域資源資料庫使用靜態類別+字典的簡單實現，鍵值對儲存資源並提供查詢方法。Key的選擇很重要，我們採用了**模組相對路徑**作為key，可滿足區分不同模組又保持與Def引用一致。
- **關聯邏輯**：在適當的時機（如Def解析完成後）遍歷Def把資源鍵替換為資源物件。這部分可在第六章討論，屬於Def系統整合範疇。

練習題

1. 嘗試在 Unity 編輯模式下手動使用 `Texture2D.LoadImage` 載入一張磁碟上的PNG圖片，確定其可行並觀察載入後的 Texture2D 屬性（如寬高）。這將有助於確認載入程式的正確性。
2. 考慮記憶體與效能因素，如果一個模組有大量高解析度圖片，你會建議使用立即載入還是延遲載入？為什麼？如果要延遲載入，你打算怎麼實現？（提示：可能需要在需要時才呼叫 LoadImage，或使用 Unity 的 Addressables/AssetBundle 技術）
3. 設計一個簡單的 GlobalTextureDatabase 測試：在載入模組後，嘗試透過 GlobalTextureDatabase 獲取某個已知鍵值的 Texture2D，然後在遊戲畫面上建立一個物件來顯示該貼圖（例如用一個 UI RawImage 元素或放在一個帶有SpriteRenderer的GameObject上）。觀察模組圖像是否正確顯示，從而驗證整個載入與查詢流程。

範例檔案架構建議

- **資源資料庫類別**：在 `Assets/Scripts/Modding/` 下新增 `GlobalTextureDatabase.cs` 和 `GlobalSoundDatabase.cs`，實作前述的靜態管理功能。也可以將聲音和圖像合併到一個 `GlobalResourceDatabase` 類別中以簡化結構，但分開有助於不同資源型別的專責管理。

- **ModContentPack 資源載入**：擴充 `ModContentPack.cs`，加入 `LoadTextures()` 與（若需要）`LoadSounds()` 方法。對於音效載入，考慮到協程，需要將其設計為協程或返回Task。如果在ModContentPack內不好直接使用Unity的協程，可以由 ModManager 來統一調度。例如ModContentPack 增加一個方法 `GetSoundLoadEnumerator()` 回傳協程IEnumerator，然後ModManager統一在一個Coroutine中執行所有模組的音效載入。

- **Def 資源欄位**：如果Def類別（在前面Def架構模擬教學中）還沒有資源欄位（如 iconTexture, soundClip），可以現在根據需要加入。也就是說，在解析Def XML時把圖像路徑存入Def物件的一個欄位，然後提供一個方法 `ResolveReferences()` 去從Global資料庫取出實際資源並賦值。這類似 RimWorld 在解析完所有Def後的 cross-reference 處理步驟 [19] [20] 。

- **測試用資源**：在先前建立的範例模組資料夾中放入幾個圖片和聲音檔。例如：

- `Mods/ModA/Textures/ModA/testIcon.png` (可以隨意準備小圖片)
- `Mods/ModA/Sounds/ModA/testSound.ogg` (可用任意短音效檔)
  並在 ModA 的某個Def（例如一個 ItemDef）XML中填入對應的 `<iconPath>ModA/testIcon.png</iconPath>`，以模擬Def對資源的引用。

完成以上後，進行遊戲執行：觀察控制台應有資源載入成功或失敗的 log。然後透過腳本或Unity檢視，驗證GlobalTextureDatabase/GlobalSoundDatabase 中是否能取到剛剛那些資源。最終，可以嘗試讓遊戲真的使用它們（如UI顯示或播放音效），確定模組的資源已真正整合進遊戲。

第六章：與 XML Def 系統的整合

前面章節我們處理了模組的程式碼與資源載入，現在關鍵的一步是將模組提供的**Def 定義**整合到遊戲的資料系統中。RimWorld 的 Def 系統允許模組增加新的遊戲內容（透過 Def），因此在我們的 Unity 模擬中，也需要讓每個模組的 Def 被讀取、解析並加入全域資料庫(Def Database)，使遊戲邏輯能感知並使用這些定義。

解析每個模組的 Defs

假設我們已經有一套簡單的 Def 系統（可能在之前的 RimWorld Def 架構模擬教學中已實作）：包含 Def 基類、各種具體 Def 類別，以及一個全域 `DefDatabase<T>` 類別用於存取註冊的 Def。這裡我們重點在**多個來源**的合併。

在 ModContentPack 的 `LoadDefs()`（或 ModManager 集中處理 Defs 的部分）中，應為**每個模組**做以下事：

1. 找到該模組 Defs 資料夾下的所有 XML 檔案（可遞迴子資料夾搜索）。對每個 XML 檔，利用既有的 Def 解析器（例如透過 `XmlSerializer` 或自寫的解析函數）轉換成對應的 Def 物件。由於我們可能不預先知道 Def 的類別，可透過XML根節點或屬性來識別類型，或要求檔案分置於特定子資料夾來區分（如 ThingDefs vs RecipeDefs）。

2. 每創建一個 Def 物件後，**不要立即加入全域資料庫**，而是先暫存在該 ModContentPack 的一個列表中（例如 `mod.LoadedDefs.Add(def)`）。這麼做的原因是，可能會有跨模組的 Def 依賴或引用，或者Def之間需要在全部載入後進行後處理（例如解析引用ID、處理繼承等）。

3. 當所有模組都各自解析完成，然後將各模組的 Defs 統一加入全域的 DefDatabase 中。可以由 ModManager 來執行這步，在 LoadAllMods 最後，加一道流程：

```
foreach(var mod in LoadedMods) {
  foreach(var def in mod.LoadedDefs) {
    DefDatabase.Add(def); // 根據 def 類型加入對應的全域庫
    def.mod = mod; // （如果 Def 基類有記錄來源mod的欄位，可在此設定）
  }
}
```

1. 最後，執行Def系統的後期處理步驟，例如解析跨引用、執行繼承、檢查錯誤等（視我們Def系統的能力而定）。在RimWorld中，這對應 19 20 和後續一系列步驟。我們簡化處理，比如若有Def有父類引用（parentDef），在此時解析並建立繼承關係；若Def有未解解析的鍵串引用其他Def，現在統一解析。

Def 的唯一性與衝突處理

當多個模組載入後，可能出現**Def 重名**的情況。所謂重名，可能指 Def 類型和 defName（或其他唯一識別屬性）相同。例如兩個模組都加入一個 `<ThingDef Name="Steel">`。對於這種衝突，我們需要有策略：

- **覆蓋**：允許後載入的模組覆蓋前者定義（類似資源覆蓋概念）。這可能導致遊戲行為改變，但是有意的覆蓋時有用（例如模組B想修改模組A或原版的某Def）。RimWorld 通常使用 **XML Patch** 來修改，而不直接以重名Def覆蓋，但在沒有Patch的情況下，按載入順序覆蓋也是合理的策略之一 21 。

- **禁止重名**：檢測到衝突就報錯或忽略。這對使用者不太友好，但保守保證資料一致性。如果選擇禁止，那在載入時就需要檢查DefName是否已存在於DefDatabase中，存在則記錄衝突。

為模擬簡化，我們可採**覆蓋策略**：即最後加入的Def將覆蓋之前同名且同類型的Def。在程式上實現就是 DefDatabase.Add 時，如已有相同鍵，則替換並發出警告日誌。

Def 與資源的關聯

在第五章討論的資源綁定需要在這裡實際動手。具體方式依您的Def系統實現而定：如果Def類別例如 ItemDef 有欄位 `public Texture2D iconTexture; public string iconPath;` 那麼在Def創建後、放入DefDatabase前，可以執行：

```
if(def is ItemDef itemDef && !string.IsNullOrEmpty(itemDef.iconPath)) {
  itemDef.iconTexture = GlobalTextureDatabase.Get(itemDef.iconPath);
  if(itemDef.iconTexture == null) {
    Debug.LogWarning($"找不到圖示資源：{itemDef.iconPath} (定義於模組 {def.mod.Name})");
  }
}
```

這樣一來，當遊戲後續需要顯示該物品時，直接使用 `itemDef.iconTexture` 即可，而不必每次查表。對於音效類似，假如有 SoundDef 之類。

整合的結果驗收

經過這步，模組所增加的內容已融入遊戲：DefDatabase　　現在包含來自各個模組的定義物件。遊戲其他系統（AI、UI等）通過尋找 DefDatabase 即可獲取它們。

我們可以進行一些驗證來確保整合正確：

- **計數檢查**：在載入後輸出日誌顯示每種 Def 類型的數量，確認數量符合模組增加的期望值。比如原版有100個ThingDef，ModA加2個，ModB加3個，則應顯示總共105個ThingDef。

- **內容檢查**：嘗試從 DefDatabase 檢索一個模組新增的Def（透過 defName 或其他索引），看看是否成功拿到，且其屬性（包括先前綁定的資源）是否完整。

- **遊戲交互**：如果有功能可以讓玩家在遊戲中創建或查看該Def代表的物件，進一步測試能否正常運作（例如生成一個Mod新增的物品，看其圖示是否顯示正確）。

學習目標

- 學習在多模組環境下批量解析並合併 Def 定義的方法，理解逐模組解析與全域合併的步驟劃分。
- 理解 Def 載入過程中的潛在衝突問題，掌握基本的處理策略（覆蓋或報錯），確保資料一致性。
- 掌握將 Def 與其相關資源（圖像/音效）聯繫起來的實作，確保模組定義在遊戲中可被正確地表現出來。

技術重點

- **XML 解析**：對每個模組重複利用既有的 XML -> Def 解析邏輯，需要注意路徑和編碼等問題（Unity 中可用 `File.ReadAllText` 獲取XML字串，再用XmlDocument或序列化處理）。
- **集合操作**：將不同來源的Def彙總時，對列表、字典的操作要謹慎，尤其處理鍵衝突時。
- **跨引用處理**：如果Def之間有引用（例如一個武器Def引用一個彈藥Def），在所有Def載入前可能無法解析，需在載入後統一處理。可透過兩階段載入：第一階段載入所有Def基本資料；第二階段解析引用（這部分在我們模擬中可簡化或假定無跨模組引用）。
- **調試技巧**：大量Def載入後debug較困難，建議添加一些日誌或使用Unity的ScriptableObject臨時呈現載入結果進行檢驗。

練習題

1. 修改你在前面章節所建立的測試模組 Def XML，在不同模組中使用相同的 `<defName>`，例如ModA和ModB都定義了一個 `ThingDef` defName為 "ExampleItem"。在載入流程中觀察並描述會發生什麼（哪個模組的定義最終留在 DefDatabase 中？是否有警告訊息？）。
2. 如果你希望避免模組Def衝突，在載入時你會選擇哪種策略？試著實現一下：當檢測到重複defName時，在控制台輸出提示衝突並**跳過**後載入的定義。
3. 考慮進階：RimWorld允許XML Def中的 `ParentName` 屬性來繼承另一個Def，模組也可以定義繼承關係。若我們也要支持這種Def繼承，試著描述載入順序上要注意什麼以及可能的處理方式（提示：需要在所有Def載入後再解析Parent引用）。

範例檔案架構建議

- **Def 基礎系統**：在 `Assets/Scripts/Defs/` 資料夾中確保持有Def相關的類別（這可能是前一章教學內容的產出）。確認有一個全域的靜態 DefDatabase 類別或類似機制，以及各種 Def 類型（例如 ThingDef, RecipeDef 等）類別定義，且包含必要欄位（含資源路徑欄位）。如果還沒有，需補充基本的 Def 架構以進行測試。

- **ModContentPack 集成**：擴充 `ModContentPack.LoadDefs()` 實作：讀取XML並解析的邏輯。如果前面有現成工具（例如SimpleXMLParser），在此調用；若無，可在此編寫簡單解析（例如XmlDocument查找節點）。解析時根據XML標記選擇建立對應 Def 類別的實例並填充資料。將生成的 Def 物件暫存於ModContentPack。如 `public List<Def> LoadedDefs = new List<Def>();`。

- **ModManager 合併**：在 `ModManager.LoadAllMods()` 結束時，增加合併步驟，如之前代碼示例。並在合併後，可以調用一次全域的 `ResolveDefReferences()` 靜態方法（需要先行設計），以執行跨引用解析與繼承等（如果不實作，可省略或輸出提示）。

- **測試Def**：使用之前範例模組中的Def XML（ModA和ModB各一個或多個Def），重啟遊戲後，在控制台打印如 `DefDatabase<ThingDef>.Count` 值，或者嘗試透過代碼取得某個已知 defName 的 Def，檢查是否存在。例如可在 ModManager.LoadAllMods() 之後加：
`Debug.Log("Total ThingDefs: " + DefDatabase<ThingDef>.AllDefs.Count);`
同時嘗試：
`var testDef = DefDatabase<ThingDef>.GetNamed("ModA_Item");`
`Debug.Log("ModA_Item exists? " + (testDef!=null));`。

確保日誌結果符合預期，即模組Def已成功載入全域。這標誌著我們的模組Def整合已完成，玩家從此可以在遊戲中體驗模組新增的內容。

第七章：資源動態載入設計探討

在前面章節，我們以**簡化**的方式實現了模組資源的載入（即在載入時一次性將模組所有資源讀入）。本章我們將更深入探討資源動態載入的其他方案，包括 Unity 提供的 Resources 機制、Addressables、以及自製的載入器等，分析各自優劣，為有進一步需求的開發者提供方向。

使用 Unity Resources 機制

Unity 有內建的 `Resources` 資料夾概念：凡放置於 `Assets/Resources/` 路徑下的資源，可在運行時通過 `Resources.Load("path")` 動態載入。這機制的優點是使用簡單且與Unity無縫結合，缺點是**資源必須打包進遊戲內**，不適合真正外部模組。若我們嘗試利用 Resources 來實現模組資源：

- 可以構想在遊戲打包時，把模組的資源也打包進去Resources，並透過命名區分。但這違背模組「外載」的初衷。
- 或者另一種思路是：模組開發者自行將資源做成 AssetBundle 然後放到Mods資料夾，遊戲載入時通過 `AssetBundle.LoadFromFile` 載入，這有點類似Resources的概念但更動態一些。

總之，Resources.Load 適合**靜態內建**資源，不適合我們在不重建遊戲的前提下載入新模組。因此在真正的模組系統中較少直接用它。倒是**AssetBundle/Addressables**可以看做Resources的延伸，適合動態內容。

Unity Addressables 系統

Addressables 是 Unity 推薦的資源管理系統，提供了方便的資源打包、加載和引用方案。其可讓開發者：

- 將特定資源標記為Addressable，打包時Unity會自動生成AssetBundle。
- 運行時通過地址（或資源標籤）來請求載入，Addressables會自動處理AssetBundle的下載（可本地可遠端）與緩存。

若應用於模組系統，我們可以：

1. 要求模組開發者以 Unity 專案製作資源，並使用我們提供的設定將自己的資源標記Addressable，然後build出專屬該模組的AssetBundle（Addressables Groups可以按模組區分）。
2. 遊戲在載入模組時，偵測有無隨附的AssetBundle（可能模組資料夾包含一個bundle文件）。然後用 `Addressables.LoadContentCatalogAsync` 或直接 `AssetBundle.LoadFromFile` 載入。
3. 模組Def中只需要引用Addressable資源的key或地址。載入時可以用 `Addressables.LoadAssetAsync<Sprite>(key)` 等獲取對象。

Addressables 的優勢在於 **高階自動**：內建緩存、依賴管理（Resource依賴）、可整合遠端下載等。對我們模擬系統而言，它比較**複雜**且需要模組在特定環境下製作。因此可能只適合非常專業的模組環境。如KSP、Cities:Skylines那樣有官方支持可以走這路。

自製載入器 vs AssetBundle

我們在第五章所做的是**自製載入器**的一種簡單形式：直接用File IO讀檔並用Unity API解析。這種方法**不需要模組作者做特別處理**（只要提供PNG、OGG文件即可），屬於原始但通用的方案。缺點包括：

- 缺少Unity引擎對格式的優化：例如圖片沒有經過Atlas打包，音效沒有壓縮優化，載入時CPU開銷全部由遊戲負擔。
- 缺少對模型、動畫等複雜資產的支持：我們的簡單載入只對圖片音效，若模組需要帶3D模型或Unity預置體，手動解析相當困難。
- 無法卸載：用AssetBundle可以整包卸載資源，而我們手動載入的Texture2D等需要自己銷毀 (Destroy) 以及小心內存洩漏。

AssetBundle 則是Unity官方提供針對**外部資源**的打包載入機制。我們可以讓模組作者將擴充內容製作成AssetBundle（Unity提供API或Editor工具），然後隨模組一起發布。遊戲端使用 `AssetBundle.LoadFromFile(path)` 載入，之後就可以使用 `bundle.LoadAsset<GameObject>("assetName")` 等取得資源。AssetBundle幾乎可以容納所有類型的Unity資產（包括Prefab、材質、模型等），非常強大。但是：

- 版本適配問題：AssetBundle和Unity版本強相關，不同Unity版本間不兼容。因此如果遊戲開發者更新了Unity版本，需要模組作者重新產出bundle。
- 編輯器要求：模組作者需要使用與遊戲一致的Unity版本，並通過Unity打包資源，這提高了門檻。
- AssetBundle本身也需要管理（例如緩存、卸載、依賴），否則容易出錯。

Addressables在AssetBundle基礎上做了一層封裝管理，簡化了一些，但仍有學習成本。

綜合比較

- **易用性**：自製載入（直接文件）勝，模組作者只需提供文件；Addressables/AssetBundle 需要模組作者有Unity操作。
- **靈活性**：AssetBundle/Addressables 勝，可載各種資產；自製文件載入僅適用有限格式，除非我們也寫自製解析器對模型等（極其困難）。
- **性能**：AssetBundle在載入前經過Build可壓縮資源並優化載入，Addressables更提供異步和內存管理；自製載入則比較粗暴直白。
- **維護**：自製方案完全由我們控制，但需要針對不同格式寫不同代碼；AssetBundle/Addressables利用Unity內建，不用重造輪子，但要跟隨Unity版本和服務更新。

對於**2D類遊戲**或內容相對簡單的擴充（像RimWorld，主要是XML配置+圖片+音效），自製載入器已足夠，而且降低模組創作門檻（只要能畫圖、改XML就能做mod）。這或許也是Ludeon為何採用此模式：RimWorld modder很多並非資深Unity使用者，但會改XML和畫sprite就能貢獻模組。這降低學習成本，壯大了社群 [22]。

如果您的遊戲模組需要**複雜遊戲物件**（例如新增關卡Prefab、新角色3D模型等），那建議引入AssetBundle或Addressables，並為模組作者提供必要的工具或指引。

學習目標

- 了解 Unity 提供的動態資源載入方案（Resources, AssetBundle, Addressables）的基本概念和適用場景。
- 比較自製簡易載入器與官方方案的優缺點，能根據遊戲需求選擇適當的模組資源管理方式。
- 認識到為模組系統選擇技術方案時，需要平衡**易用性**與**功能**：有時簡單可及的方案更利於社群參與。

技術重點

- **Resources vs AssetBundle**：Resources屬靜態，AssetBundle屬動態。AssetBundle在Unity中扮演外部資源容器角色，是大多數Unity遊戲Mod系統繞不開的部分。
- **Addressables**：Unity 2018+ 推出的高層資源系統，本質利用AssetBundle，但省去許多繁瑣細節，值得了解。
- **模組作者體驗**：在選擇技術時，要考慮第三方製作者的技術水平。RimWorld選擇XML+素材文件，很大程度上因為這些對非程序員也友好 [22] [23]。反之，如果要求每個模組都包含DLL+AssetBundle，門檻就高很多。

練習題

1. 針對你的遊戲類型，思考最適合的模組資源載入方式是什麼？列出兩種以上方案並說明理由。例如，你的遊戲是2D像素風格，或是3D大型場景，各自會傾向哪種方式。
2. 查閱 Unity 官方關於 AssetBundle 或 Addressables 的文檔（或教程），總結使用它們來實現模組資源載入的基本步驟。
3. 如果讓你為普通玩家制作模組提供工具，你會怎麼降低使用 AssetBundle/Addressables 的難度？（提示：或許可以做一個Unity Editor擴充，用戶只需放素材點擊一鍵導出mod）。

範例檔案架構建議

本章屬討論性質，沒有新增必需程式碼。但若要嘗試Addressables/AssetBundle，可考慮：

- **Addressables實驗**：在Unity中安裝Addressables套件，設定一些資源為Addressable，然後構建Addressables Content。將生成的 `catalog.json` 和 `.bundle` 文件放入 Mods/<ModName>/ 中。撰寫測試腳本使用 `Addressables.LoadContentCatalogAsync` 指向該catalog路徑，再加載其中的資源。例如：

  ```
  Addressables.LoadContentCatalogAsync("file:///[模組資料夾]/catalog.json").Completed += op => {
    Addressables.LoadAssetAsync<Texture2D>("MyAssetKey").Completed += texOp => { ... }
  };
  ```

  驗證能否成功載入。這練習較進階，但能體驗Addressables在Mod中的運用。

- **AssetBundle實驗**：使用Unity菜單的 **AssetBundle Browser**（需安裝）、或腳本標記資源的AssetBundle 名稱並使用BuildPipeline.BuildAssetBundles輸出一個bundle檔。將其置入Mods資料夾，在遊戲運行時用 `AssetBundle.LoadFromFile` 載入，然後 `LoadAsset` 取出，驗證是否可行。

上述實驗可加深對動態資源載入的理解，但需要一定Unity進階知識。初學者可先了解概念，不必實作。在實際開發中，請根據團隊情況與玩家社群技術能力，決定採用何種方案實現模組資源管理。

第八章：模組錯誤隔離機制

當允許第三方開發的模組接入遊戲時，不可避免地會遇到某些模組**內容有錯**（例如XML格式不正確）或**程式碼異常**導致問題的情況。為提升整體穩定性，我們需要設計錯誤隔離機制，確保單個模組的失敗不至於拖垮整個遊戲。同時，也要給出清晰的錯誤資訊，方便開發者或使用者排查。

載入階段的錯誤處理

在前述載入步驟中，我們已經多次強調在關鍵環節加上 `try-catch`：

- **載入DLL**：使用 Assembly.Load 時 catch ReflectionTypeLoadException 或一般 Exception。如某個DLL無法載入（可能因框架不符或引用缺失），我們記錄錯誤並跳過該模組的程式集載入。這意味著模組的C#功能將不可用，但我們仍可選擇載入其Defs和資源（如果這部分不依賴程式碼）。

- **解析XML**：如果某個 Def XML 文檔格式錯誤（拋出XmlException）或內容無效導致我們的解析代碼錯誤，在 LoadDefs() 中捕捉。可以做的是**跳過該檔案**，並在日誌中指出是哪個模組的哪個檔有問題。繼續解析其他檔案，避免整個模組終止載入。或者，視嚴重程度，終止該模組Def載入但繼續其他模組。

- **資源載入**：如某圖檔損壞或格式不支援，Texture2D.LoadImage 可能返回false或拋出。我們在 LoadTextures() 已加了 try-catch。處理策略通常是**忽略該資源**（比如設置一個佔位Texture或置null），並錯誤日誌提示具體檔案。剩餘資源繼續嘗試載入。

- **初始化執行**：在呼叫 IModInitializer.Initialize(mod) 時也應該 catch 包住。因為模組的初始化邏輯是外部提供，我們無法保證其正確。如果一個模組初始化拋異常，我們可以記錄並**繼續**載入後續模組的初始化。此模組本身可能處於不完全啟動狀態，但遊戲不應因此停下。極端情況下，若初始化失敗導致遊戲關鍵系統受影響，可考慮將該模組標記為失效並提示玩家。

運行時的模組錯誤隔離

載入時的錯誤較易控制，但模組的程式碼在遊戲運行過程中也可能產生錯誤（例如模組監聽某事件時拋出了NullReferenceException）。完全預防這些是不可能的，但我們可以考慮：

- **關鍵系統防護**：在遊戲主迴圈、事件調用等關鍵路徑上，對外部模組的調用進行保護。例如，如果遊戲提供給模組一個事件鉤子 OnDayTick()，那在我們的事件觸發處，可以 `foreach(mod in LoadedMods) { try { mod.OnDayTick(); } catch(Exception e) { LogModError(mod, e); } }`。這樣某模組出錯，不會中斷其他模組和主流程。

- **模組沙盒**：進階的隔離是將模組程式碼放在**獨立的應用程式域**或甚至獨立進程中執行。典型案例如 Kerbal Space Program曾嘗試使用AppDomain載入插件，出錯可回收AppDomain。不過在Unity環境，AppDomain熱切換有限制，Unity 2020之後已不支援自定義AppDomain卸載。此外IL2CPP不支持反射執行這類。故大多數Unity遊戲mod系統停留在**同進程軟隔離**級別，以妥善的程式結構+錯誤處理為主。

- **錯誤日誌與停用**：一旦檢測到某模組頻繁或嚴重出錯，可以考慮**自動停用**該模組的某些功能或者整個模組。例如記錄該模組錯誤次數，如果超過閾值則將其狀態標記為掛起，不再調用它的後續更新。這需要模組管理器維護模組的狀態（Active/Inactive），供遊戲各系統查詢是否要跳過該模組。

向玩家和開發者提供資訊

隔離錯誤的同時，透明地提供錯誤資訊也很重要：

- **日誌分類**：可以將模組錯誤標識出來源。例如在Log輸出時，加上前綴 "[Mod X]" 指出是哪個模組引發錯誤。這讓開發者可以迅速定位問題模組。
- **UI提示**：在遊戲啟動或運行中，如果檢測到模組載入錯誤，可在模組選單或遊戲介面中提示玩家某模組未正確載入。RimWorld本身在主選單mod列表界面就會把有錯的模組標紅並提供錯誤日誌鏈接。作為模擬，我們可以簡單地在控制台打印，或將信息寫入文件供玩家查看。
- **開發模式**：如果遊戲有開發者模式，可顯示更詳細的堆疊資訊甚至中斷點方便調試。但普通玩家模式下應盡量簡潔提示即可，避免嚇到玩家。

例外 vs 中斷

值得注意，在Unity中Unhandled Exception通常不會直接殺死遊戲，但會打出調試日誌。我們的做法是**主動攔截**已知可能出錯的點，將其轉化為我們可控的錯誤資訊。目的是提供友好的反饋和保持遊戲穩定運行。
當然，也要接受一個現實：如果模組直接在Update循環裡寫了個 `throw new Exception()` 沒有被我們捕捉，那仍然會造成Unity日誌錯誤 spam。絕對的安全無法保證，但我們已盡可能在接口層面做了隔離。

學習目標

- 理解為何在模組系統中需要特別重視錯誤隔離，以及失去隔離可能帶來的嚴重後果（如一個模組崩潰導致整個遊戲閃退或卡死）。
- 掌握常見錯誤隔離技巧：使用 try-catch 包裹模組相關操作、追蹤模組錯誤來源、在需要時禁用問題模組。
- 瞭解進階隔離（如AppDomain）的概念，並認識其在Unity環境下的侷限，了解我們主要依賴**軟性隔離**（程式邏輯）而非**硬性隔離**（系統沙盒）的現實。

技術重點

- **錯誤攔截**：分析模組載入和執行過程中的各個環節，在哪些地方容易出錯，逐一添加防護。這需要經驗和細心，也建議多測試惡意或不良模組的行為。
- **模組狀態管理**：引入模組狀態的概念，例如ModContentPack增加一個 `HasLoadError` 或 `IsActive` 屬性。載入時若有重大錯誤，設定狀態以供後續使用（如跳過其Initialize或遊戲事件）。
- **用戶體驗**：設計錯誤資訊的呈現方式。過多技術細節可能讓玩家困惑，過少又不足以讓開發者修復。可以考慮將詳細錯誤輸出到檔案（供開發者）而在UI上僅顯示模組名稱和簡述（供玩家）。

練習題

1. 模擬一個模組載入錯誤：例如手動破壞一個模組的 About.xml（寫入不合法XML），然後執行遊戲。觀察我們的載入流程如何處理這情況，並記錄輸出的錯誤資訊是否清晰指出問題所在。
2. 如果某模組的 IModInitializer.Initialize 內部有錯，但被我們捕捉了，從使用者角度看可能遊戲繼續正常。但該模組實際沒正確初始化。你認為有沒有必要告知玩家或停用該模組？如果有，要如何在介面上反映（例如在模組列表顯示「啟動失敗」）？

3. 思考：我們假設每個模組是獨立的，但現實中模組之間可能互相調用彼此程式碼或資源（即有耦合）。若ModA呼叫了ModB的代碼，ModB出錯掛掉，可能連ModA都功能不正常。這種跨模組錯誤你覺得能防範嗎？如果無法從系統層面防止，你建議如何約束或指導模組開發者避免這類情況？

範例檔案架構建議

- **錯誤處理代碼覆查**：回顧之前實作的各部分（ModManager, ModContentPack等），確保所有可能的Exception來源都有適當的try-catch。特別在反射和IO部分。可以嘗試故意引發錯誤（例如在解析函數裡 `throw`）來驗證防護是否生效。

- **模組狀態標記**：修改 `ModContentPack` 類別，增加類似 `public bool LoadError { get; private set; }`。在捕捉到嚴重錯誤時（如DLL無法載入或整個Defs皆失敗）設定此flag。後續如果遊戲有模組管理UI，這flag可以用來提示用戶。當然，也可以增加 `List<string> ErrorMessages` 留存詳細錯誤。

- **日誌記錄**：實作一個模組錯誤專用的日誌輸出工具。例如 `ModErrorLog.Log(ModContentPack mod, string message)`，它會將訊息格式化包含模組名稱並寫入Unity的Console和一份獨立的日誌檔案（如 "Mods/ModError.log"）。這樣玩家可以將該檔案提供給模組作者調試。

- **測試錯誤場景**：準備一個"BadMod"：

- About.xml 缺失某必要欄位或格式錯。
- Assemblies 裡的DLL刻意編譯錯版本（比如 .NET版本不符，引起載入錯）。
- Defs/XML 有語法錯。
- IModInitializer.Initialize 內寫 `throw new Exception("Init failed");`。
  將此模組放入Mods，跑遊戲，看我們系統對每種錯誤的處理是否各自獨立且遊戲仍能進入主畫面。不斷完善直到這些錯誤都被友善處理。

透過本章完善，我們的模組框架將更健壯，能容忍模組可能的錯誤而不至於影響整體遊戲體驗。這對真實世界的遊戲發行非常重要，因為無法期望所有模組都完美無瑕，有了隔離機制，玩家可以在出問題時只移除或禁用有問題的模組，而非整個遊戲崩潰。

第九章：完整模組架構範例與總結

經過前面各章節的逐步實作與講解，我們已搭建出一個初步完整的 Unity 模組(Mod)系統框架。最後本章將通過一個綜合範例，串聯先前所有部分，並提供最終的教材總結。

綜合範例：載入多個模組並使用其內容

假設我們的 Unity 遊戲專案經過改造，已包含：

- ModManager 模組管理器，可在啟動時載入 Mods 資料夾下的所有模組。
- ModContentPack 類別，封裝模組資訊並提供 LoadAssemblies/LoadDefs/LoadResources 等方法。
- IModInitializer 介面，模組DLL可實作以在載入時執行初始化邏輯。
- GlobalTextureDatabase/GlobalSoundDatabase 等，統一管理資源。
- DefDatabase 等 Def 系統，整合模組定義。
- 錯誤隔離處理，在每步都有防護並記錄問題。

現在，我們準備兩個模組作為驗收：

模組 ModA: - About.xml 定義 name: "ModA", packageId: "com.mygame.moda". - Assemblies 下有 ModA.dll，內含一個類別 ModAInitializer：IModInitializer，在 Initialize 中打印 "ModA initialized"。 - Defs 下有一個 ThingDef 文件 `ThingDefs/ModA_Item.xml` 定義一個新物品，defName="ModA_Item"，iconPath="ModA/Items/ModAItem.png"。 - Textures 下有 `ModA/Items/ModAItem.png` 圖片檔作為該物品圖示。

模組 ModB: - About.xml: name: "ModB". - Assemblies: ModB.dll，含 ModBInitializer：IModInitializer，Initialize 打印 "ModB initialized". - Defs: 一個 ThingDef 檔 `ThingDefs/ModB_Item.xml`，defName="ModB_Item"，iconPath="ModB/ModBIcon.png". - Textures: `ModB/ModBIcon.png` 圖片檔。

將這兩個模組資料夾放入遊戲的 Mods 資料夾後，執行遊戲，**預期結果**：

1. **載入日誌**：控制台應依序顯示：
2. "Mod loaded: ModA"
3. "[ModA] 模組初始化執行" (來自 ModAInitializer)
4. "Mod loaded: ModB"
5. "[ModB] 模組初始化執行" (來自 ModBInitializer)
   這表示兩個模組的DLL均成功載入且初始化被調用。
6. **Def 整合**：我們可以在初始化完後，嘗試使用遊戲提供的指令或測試代碼查看 DefDatabase:
7. 查詢 defName "ModA_Item" 和 "ModB_Item"，應該都能找到對應 Def，且各自帶有 iconTexture 屬性 (已載入的Texture2D)。
   若打印物品列表，可見原本遊戲物品再加上這兩項新物品。
8. **圖示顯示**：如果我們在遊戲UI中特意創建這兩個物品的圖示（例如放在遊戲場景或介面上），應該能看到來自模組的圖像正確顯示，證明資源載入成功且Def關聯正確。
9. **錯誤檢查**：在這兩個模組沒有引發錯誤的情況下，不會有錯誤日誌。但我們也測試過BadMod等，它的不良XML或初始化錯誤都被我們隔離處理，遊戲仍然啟動並能載入其他正常模組（只是在日誌中報告BadMod的問題）。這驗證錯誤隔離機制有效。

系統架構總覽圖 (文字描述)

為方便理解整個系統，以下用文字描述我們模組系統主要組成與流程：

- **模組內容（Mods 資料夾）**：外部資料夾包含多個子資料夾，每個子資料夾 = 一個模組(Mod)。模組內依照固定結構放置 About.xml, Defs, Assemblies, Textures 等內容。
- **遊戲啟動**：Unity 主程式啟動時，`ModManager` 開始運作，定位 Mods 資料夾並列出模組。對每個模組，建立一個 ModContentPack 實例並執行載入。載入包括：
- **讀取描述**：About.xml -> 填入 ModContentPack.Name 等資訊。
- **載入程式碼**：掃描 Assemblies/*.dll -> 反射載入Assembly -> 儲存。
- **載入定義**：解析 Defs/*.xml -> 產生 Def 對象列表 -> 暫存。
- **載入資源**：讀取 Textures/.png, Sounds/.ogg -> 轉為 Unity 物件 -> 註冊至全域庫。
- **執行初始化**：掃描已載入Assembly的類型 -> 找到 IModInitializer -> 實例化並呼叫 Initialize。
- **錯誤處理**：在上述每一步中，如有異常，記錄錯誤並跳過該步驟後續，但繼續下一模組。

- **合併資料**：所有模組的 Def 暫存收集完後，ModManager 將它們合併到遊戲全域 DefDatabase 中。若有命名衝突，後者覆蓋前者並警告。隨後執行一次 Def參照解析（如有需要）。

- **運行時**：遊戲其他系統現在可以從 DefDatabase 獲取包含模組內容的定義，比如物品生成系統能拿到 ModA_Item 的定義去產生該物品；UI能透過 Def引用的 iconTexture 顯示圖標；模組程式碼（IModInitializer可能註冊的事件）也在運作，拓展了遊戲功能。

- **隔離監控**：如果模組在後續運行中拋出錯誤，我們已在主要接口處（如事件分派）做好保護，錯誤會被截住並報告，不會讓遊戲崩潰。

這一整套流程，實現了 RimWorld ModContentPack 系統在 Unity 環境下的一種**模擬**。雖未必與實作細節100%相同，但核心理念是一致的：**透過標準化的資源與程式封裝，讓遊戲能夠動態地加載、整合模組內容，同時保持各模組之間的相對獨立性與穩定性**。

總結

在本教材中，我們從零開始，逐步建立了一個模組系統，包括目錄結構規範、載入管理、模組初始化、資源處理、Def整合以及錯誤隔離等方方面面。這些步驟對於一款支持模組的遊戲而言，都是不可或缺的基礎。

關鍵的心得包括：

- **架構先行**：模組系統需要在遊戲架構階段就納入考量，正如我們所見，許多遊戲子系統（資源管理、內容定義、事件系統）都需因應模組化進行設計或調整 22 。因此，提早規劃能減少日後修改成本。

- **簡化開放**：為了鼓勵更多人製作模組，系統應儘量**簡潔**且**文件齊全**。RimWorld 之所以有繁榮的Mod社群，一部分原因就是XML+素材的模式降低了門檻 22 。我們在設計自己遊戲的模組系統時，也要找到功能與易用的平衡點。

- **安全穩定**：模組是不可控的外來代碼與數據，必須做好隔離和檢查。**永遠不要相信輸入**——這裡的輸入指模組提供的任何內容。我們的示範通過多重try-catch與驗證，最大程度降低了惡劣模組影響。但仍需保持警惕，在發行後根據玩家反饋繼續改進安全性。

- **循序漸進**：可以從支持簡單內容開始（例如僅XML配置的數值mod），逐步擴展到支持劇情事件、再到支持全腳本編寫。一步步擴大模組API的範圍，避免一次設計過度複雜導致難以實現或bug叢生。

最後，希望本教材讓您對 RimWorld 的ModContentPack架構以及一般Unity遊戲如何實現模組支持有了清晰的認識。透過實踐練習，您將能夠將這套框架應用到自己的Unity項目中，打造出擁有豐富社群創作內容的遊戲！

學習目標

- 將之前各章節學到的概念融會貫通，理解整個模組系統的工作流程。
- 能夠親手搭建一個簡易的模組框架並驗證多個模組載入的協同運作。
- 瞭解設計模組系統的要點，包括結構規範、載入順序、錯誤處理以及開發者/使用者體驗等綜合因素。

技術重點

- **系統整合**：各模組子系統（載入、資源、Def）如何交互，例如 Def 載入需要依賴資源庫，模組初始化可能影響 Def 數據等，要處理好先後依賴。
- **全面測試**：一個模組系統涉及檔案IO、反射、序列化、多執行緒(協程)等，需要全面測試不同情況（有無DLL、有無資源、錯誤模組等）。

- **可擴充性**：我們的框架是簡化的，但在實作時應考慮未來擴充，例如增加對AssetBundle的支持、更多類型Def的處理、模組相依關係解決等，保持架構的彈性。

練習題

1. 將本章提供的 ModA 和 ModB 範例模組實際配置並運行，觀察是否達到預期效果。如果不成功，檢查是哪個環節出問題並調試修正。
2. 編寫一個新的模組 ModC，嘗試增添一種新的 Def 類型（假如遊戲原本沒有，例如新增一個技能 SkillDef）。這需要你擴充遊戲的Def系統來支持這類Def，以及讓 ModC 的 XML 定義一個SkillDef。載入後檢查ModC的SkillDef是否進入DefDatabase。
3. 假設要將本模組框架應用到另一種類型的遊戲（比如動作遊戲需要可插入關卡、角色等），列出你認為還需要新增或修改哪些部分。例如增加對模型Prefab的支持（可能透過AssetBundle）、對模組之間通信的支持（例如一個模組能調用另一個的服務）等等。

範例檔案架構建議

- **最終項目結構**：整理Unity項目中與模組系統相關的腳本和資源位置：
- `Assets/Scripts/Modding/`：包含 ModManager.cs, ModContentPack.cs, IModInitializer.cs, GlobalTextureDatabase.cs, GlobalSoundDatabase.cs 等腳本。
- `Assets/Scripts/Defs/`：包含 Def 類別定義和 DefDatabase 管理。
- `Mods/`(在專案根目錄或者PersistentDataPath)：包含測試的 ModA, ModB, (BadMod) 等資料夾，各自有 About.xml, Defs, Assemblies, Textures...
- （如果實作AssetBundle測試則還有AssetBundles資料夾等，不贅述）

- **Documentation**：為你的模組系統寫一份簡短使用說明，供未來的模組開發者參考。內容包括：模組資料夾應放哪、需要哪些必備檔案(如About.xml)、可以添加哪些類型的內容(Defs清單)、如何編譯DLL（目標.NET版本）等。這實際也是模組系統不可缺的一環（RimWorld 就有官方Wiki和大量社群指南）。

完成以上，您就擁有了一個初步完整的Unity遊戲模組化框架！未來可以根據需要繼續優化，但即使保持簡單，這套系統也足以讓您的遊戲具備擴展性，從而延長遊戲壽命並建立起玩家社群，共同豐富您的遊戲世界。祝您在實作自己的模組系統時一切順利，期待看到各種精彩的玩家創造內容！ [1] [11]

---

[1] [2] [3] [5] [8] [9] [11] [12] [21] Mod Folder Structure - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/Mod_Folder_Structure

[4] [6] [7] [10] [13] [14] Mod Folder Basics - RimWorld Modding Wiki
https://rimworldmodding.wiki.gg/wiki/Mod_Folder_Basics

[15] [16] [18] [19] [20] Application Startup - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/Application_Startup

[17] RimWorld Modding Guide 教程-CSDN博客
https://blog.csdn.net/gitblog_00051/article/details/141768614

[22] [23] Creating A Moddable Unity Game
https://www.turiyaware.com/creating-a-moddable-unity-game/