ChatGPT

RimWorld Def 類別的 Class 屬性使用教學

在 Def XML 中指定自訂類別的標準寫法與位置

在 RimWorld 的 XML 定義檔（Def）中，可以透過 **Class 屬性** 來指定使用自訂的 C# 類別，以取代預設的遊戲邏輯類別 [1]。這通常以特定的標籤名稱出現在 Def XML 中，例如 `<thingClass>`、`<workerClass>`、`<giverClass>` 等。這些標籤一般放置在 `<defName>` 等基本定義之後，作為 Def 的一部分。例如：

```
<ThingDef ParentName="BuildingBase">
  <defName>MySpecialLamp</defName>
  <label>special lamp</label>
  <!-- 指定此物件使用自訂的 C# 類別 -->
  <thingClass>MyModNamespace.Building_MySpecialLamp, MyModAssembly</thingClass>
  ...（其他屬性）...
</ThingDef>
```

上例中，`<thingClass>` 指定此 Thing 使用我們自訂的 `Building_MySpecialLamp` 類別。通常寫法需**包含命名空間與類別名**，並確保該類別所在的程序集（如上例的 `MyModAssembly.dll`）已隨模組載入 [2]。如果類別不在核心遊戲中，最好附加程序集名稱（逗號後）以避免衝突。總之，務必要提供**完整的 Namespace.Class 名稱**，並確保模組的 DLL 已提供該類別 [2]。

常見 Def 類型的 class 屬性名稱及意義

許多常見的 Def 類型支援自訂的 class 屬性，以下列出幾個主要例子：

- **ThingDef**：使用 `<thingClass>` 標籤來指定物件所對應的 C# 類別。預設情況下，不同類型的 ThingDef 會對應遊戲內預設的類別，例如建築物通常對應 `Building` 或其子類、植物對應 `Plant` 類別等。透過指定自訂 thingClass，可讓該物件在遊戲中採用自訂的互動行為或邏輯。 [1]

- **RecipeDef**：使用 `<workerClass>` 標籤來指定製造/手術配方所使用的 **RecipeWorker** 類別 [3]。預設為 `RecipeWorker` 基類，若指定自訂類別（繼承自 `RecipeWorker`），即可改變該配方在執行時的特殊處理（例如檢查條件、產出計算等）。

- **WorkGiverDef**：使用 `<giverClass>` 標籤來指定**工作指派者**所使用的 C# 類別 [4]。WorkGiver 決定了殖民者在何種條件下執行某工作，例如清掃、運輸等。自訂 giverClass（繼承自 `WorkGiver` 或其子類）可調整 AI 尋找工作的邏輯，改變工作執行的條件或順序。

- **JobDef**：使用 `<driverClass>` 標籤來指定**工作驅動**類別（JobDriver） [5]。JobDriver 控制殖民者執行特定工作的行為流程。自訂 driverClass 可改寫工作執行的步驟（例如新增額外動作或改變執行順序）。

- **HediffDef**：使用 `<hediffClass>` 來指定健康附加狀態（Hediff）的類別 [6]。例如狀態效果（疾病、藥物效果等）預設使用 `HediffWithComps` 或其他基類，自訂 hediffClass 可實現特殊的健康狀態邏輯。

- **IncidentDef**：使用 `<workerClass>` 來指定事件的 **IncidentWorker** 類別。IncidentWorker 控制隨機事件（如襲擊、天氣改變）的具體執行方式。透過自訂類別可以改寫事件發生時的邏輯。（例如，遊戲內 Eclipse 事件的 GameConditionDef 就透過 `conditionClass` 指定使用 `GameCondition_Eclipse` 類別；若想改寫其行為，可自訂繼承類別並修改 conditionClass 指向它[7]。）

以上各類 Def 的 class 屬性名稱各異，但用途皆在於**將該 Def 綁定到特定的 C# 類別**[1]。透過這種機制，模組可以用自訂程式碼取代原有邏輯。例如，BiomeDef 使用 `workerClass` 綁定一個 `BiomeWorker` 子類，決定該生物群系在世界生成時的表現[1]；ThoughtDef 使用 `workerClass` 綁定 `ThoughtWorker` 類別，決定該心情思緒何時生效等等。總之，當你在 Def 中看到 `thingClass`、`workerClass`、`giverClass`、`driverClass` 等字樣時，就表示可以指定一個自訂類別來擴充或改寫遊戲行為。

注意： 若要使用自訂類別，必須在 XML 正確地引用類別名稱，並確保對應的類別已在模組的 DLL 中定義且隨遊戲載入[2]。否則遊戲在載入 XML 時將找不到類別，出現錯誤（例如「has null thingClass」或「Could not find class ...」的錯誤）。正確使用方式如上所示，完整命名空間加類別名，必要時包含程序集名稱。

撰寫自訂繼承類別以控制遊戲邏輯

下面我們透過幾個實例，說明如何撰寫自訂類別繼承遊戲的基礎類別，並配合在 Def XML 中的 class 屬性設定，達到改寫遊戲邏輯的目的[8]。每個情境都會提供完整的 XML 片段、對應的 C# 類別實作，以及重點註解說明。

範例 1：自訂 ThingClass 改變物件互動邏輯

情境： 我們希望製作一個特殊的建築物（例如一個會閃爍燈光的燈具）。透過自訂 ThingClass，我們可以改變此建築物的行為，例如每隔一定時間自動執行某動作。

XML 定義（ThingDef）：

```
<ThingDef ParentName="BuildingBase">
  <defName>BlinkingLamp</defName>
  <label>blinking lamp</label>
  <description>A lamp that blinks periodically.</description>
  <!-- 使用自訂的 Thing 類別替代預設的 Building -->
  <thingClass>MyMod.Building_BlinkingLamp</thingClass>
  <comps>
    <li Class="CompPowerTrader"/> <!-- 例如讓此燈具有用電的Comp -->
  </comps>
</ThingDef>
```

在上述 XML 中，`<thingClass>` 指向我們自定義的類別 `MyMod.Building_BlinkingLamp`（位於我們模組的命名空間 MyMod）。這個類別需要繼承 RimWorld 中適當的基類。由於我們的物件是一個建築物，**通常**應繼承自 `Verse.Building` 或其子類（很多可互動建築繼承自 `Building` 或 `Building_WorkTable` 等）。

C# 類別實作（Building_BlinkingLamp）：

```
using RimWorld;
using Verse;

namespace MyMod
{
  // 繼承自 Building 類別，表示這是一個建築物
  public class Building_BlinkingLamp : Building  // 自訂燈具建築
  {
    private int tickerInterval = 250;  // 每250 tick觸發一次（約4秒遊戲時間）

    public override void Tick()
    {
      base.Tick();  // 呼叫基底實作，確保正常的建築物行為
      if (this.IsHashIntervalTick(tickerInterval))
      {
        // 每隔一定tick執行一次：切換燈光或播放效果
        // 這裡我們簡單地讓燈每次切換開關狀態
        bool currentlyLit = this.HasPower;
        // (假設使用CompPowerTrader決定HasPower供電狀態)
        if (currentlyLit)
        {
          // 如果有電且燈亮，則關閉（僅示意，實際可透過Comp實現）
          this.GetComp<CompPowerTrader>()?.PowerOff();
        }
        else
        {
          // 如果燈關，則打開
          this.GetComp<CompPowerTrader>()?.PowerOn();
        }
        // 我們也可以在這裡添加其他效果，例如產生光影或訊息。
        Log.Message("BlinkingLamp toggled at " + this.Position);
      }
    }
  }
}
```

上述程式碼中，我們定義了 `Building_BlinkingLamp` 類別，繼承自 `Building`。透過 override `Tick()` 方法，我們在每隔固定時間執行自訂的邏輯（在這裡讓燈開關閃爍）。`this.IsHashIntervalTick(250)` 是 RimWorld 提供的輔助函式，可用來每隔一定 tick 執行代碼，而不必每個 tick 都檢查。這樣的自訂 ThingClass 使我們能改變物件的互動行為，例如週期性動作、特殊的被使用反應等。只要在 XML 中正確引用，遊戲就會在生成此物件時使用我們的類別，進而執行我們定義的邏輯。

註：實際上控制燈光的開關通常透過 Comp 或其他機制實現；此處為教學簡化示意。重點在於透過自訂類別，可以 override 原本在 `Thing` / `Building` 中的虛擬方法（如 Tick、Destroy、InteractionCell 等）來改變行為。

範例 2：自訂 WorkGiverDef 的 Worker（giverClass）改變工作 AI 行為

情境： 我們希望新增一種特殊工作，例如讓殖民者自動去填充一種自定義容器（比如前述的 VodkaBarrel 酒桶）。我們可以透過新增一個 WorkGiverDef 並指定自訂的工作給予者類別，來改變 AI 的工作指派邏輯 [4]。以下展示一個自訂 WorkGiver 的例子。

XML 定義（WorkGiverDef）：

```
<WorkGiverDef>
 <defName>FillVodkaBarrel</defName>
 <label>fill vodka barrels</label>
 <giverClass>MyMod.WorkGiver_FillVodkaBarrel</giverClass>
 <workTypeDef>Hauling</workTypeDef>
 <priority>50</priority>
</WorkGiverDef>
```

上述 XML 定義了一個新的工作給予者，其 `defName` 是 FillVodkaBarrel，並將 `<giverClass>` 指向我們的 `MyMod.WorkGiver_FillVodkaBarrel` 類別（該類別繼承自 RimWorld 的 `WorkGiver` 基類）。我們也指定了這個 WorkGiver 屬於搬運（Hauling）類的工作類型，優先度為50（僅作為示例數值）。

C# 類別實作（WorkGiver_FillVodkaBarrel）：

```
using System.Collections.Generic;
using RimWorld;
using Verse;
using Verse.AI;

namespace MyMod
{
  // 繼承自 WorkGiver_Scanner，這是大多數需要搜尋目標的工作給予者基類
  public class WorkGiver_FillVodkaBarrel : WorkGiver_Scanner
  {
    // 定義此 WorkGiver 尋找的目標類型，例如只尋找我們的 VodkaBarrel 物件
    public override ThingRequest PotentialWorkThingRequest
      => ThingRequest.ForDef(DefDatabase<ThingDef>.GetNamed("VodkaBarrel"));

    // 可選：限制搜尋範圍，本例不限定全局掃描
    public override PathEndMode PathEndMode => PathEndMode.Touch;

    // 判斷Pawn是否有可執行此工作的條件
    public override bool ShouldSkip(Pawn pawn, bool forced = false)
    {
      // 若該Pawn無法搬運或其他條件不符，則跳過
      return !pawn.health.capacities.CapableOf(PawnCapacityDefOf.Manipulation);
    }

    // 核心：判斷指定Pawn對某個物件是否有工作要做
    public override bool HasJobOnThing(Pawn pawn, Thing t, bool forced = false)
    {
```

```
        // 檢查目標是否為我們的VodkaBarrel，且尚未滿（需要填充）
        if (t is Building_VodkaBarrel barrel && !barrel.IsFullyFilled)
        {
            // 檢查Pawn是否有可用的填充材料等（略）
            return true;
        }
        return false;
    }

    // 核心：給出具體的工作(Job)指派
    public override Job JobOnThing(Pawn pawn, Thing t, bool forced = false)
    {
        // 讓Pawn執行我們自訂的 FillVodkaBarrel 工作（JobDef）
        // 假設我們有對應的 JobDef 和 JobDriver 來處理實際行為
        JobDef jobDef = DefDatabase<JobDef>.GetNamed("FillVodkaBarrel");
        return JobMaker.MakeJob(jobDef, t);
    }
  }
}
```

在上述實作中，我們的 `WorkGiver_FillVodkaBarrel` 繼承自 `WorkGiver_Scanner`（這是 RimWorld 中用來掃描地圖上目標物的工作給予者基類）。主要覆寫了幾個方法：

- `PotentialWorkThingRequest`：告訴遊戲此工作感興趣的目標是哪些。這裡我們限定為尋找 `ThingDef` 名為 "VodkaBarrel" 的物件（假定我們有對應的酒桶物件 Def）。
- `ShouldSkip`：在掃描前檢查 Pawn 是否應跳過此工作（例如不具備所需能力時跳過）。
- `HasJobOnThing`：判斷給定 Pawn 對特定物件是否有工作可做。在此我們檢查該物件是否為我們的目標酒桶且需要填充，以及 Pawn 是否能執行。
- `JobOnThing`：當確定有工作時，生成實際的 Job。這裡假設我們已定義了 JobDef `"FillVodkaBarrel"` 及相應的 JobDriver 來執行填充行為。

透過自訂 WorkGiver 類別並在 XML 中引用，我們改變了 AI 分配工作的條件和內容。這樣殖民者在遊戲中就會將填充酒桶視為一項可執行的工作，並按照我們定義的邏輯去尋找酒桶並執行填充。原本遊戲並沒有 FillVodkaBarrel 這種工作，經由這種方式就能新增新的工作類型及其 AI 行為 [8]。(上例靈感來源於社群範例，其中 WorkGiverDef 定義了 `<giverClass>Neutroamine_Distilling.WorkGiver_FillVodkaBarrel</giverClass>` 並對應 C# 類別 [4]。)

提示： WorkGiver 通常和 JobDef、JobDriver 搭配使用。XML 定義的 WorkGiverDef 指向自訂 WorkGiver 類別，而該 WorkGiver 類別中會指派某個 JobDef（可使用 `<driverClass>` 綁定自訂 JobDriver）。例如上面代碼中 `JobMaker.MakeJob(jobDef, t)` 會讓Pawn執行我們在 JobDef `"FillVodkaBarrel"` 中指定的 driverClass（假設該 JobDef 使用我們的自訂 `JobDriver_FillVodkaBarrel` 類別）[5]。這樣即可全方位地改變從工作搜尋->指派->執行的整個流程。

範例 3：自訂 RecipeDef 的 WorkerClass 改變製作流程或產出

情境： 我們希望修改一個製造配方或手術的行為，例如讓某個配方在完成後返還部分材料，或者在手術成功時有特殊效果。透過自訂 RecipeWorker 類別並在 RecipeDef 中指定 `<workerClass>`，可以實現上述目的。

XML 定義（RecipeDef）：

```
<RecipeDef>
 <defName>MakeAdvancedComponent</defName>
 <label>craft advanced component</label>
 <jobString>crafting advanced component.</jobString>
 <!-- 指定使用自訂的 RecipeWorker 類別 -->
 <workerClass>MyMod.RecipeWorker_MakeAdvancedComponent</workerClass>
 <ingredients>
  ...（配方所需材料定義）...
 </ingredients>
 <products>
  ...（產出物定義，例如先進元件）...
 </products>
</RecipeDef>
```

在這個 RecipeDef 中，我們將 `workerClass` 指向 `MyMod.RecipeWorker_MakeAdvancedComponent`。預設情況下，大多數配方使用的是 RimWorld 提供的 `RecipeWorker` 或其內建子類。自訂一個繼承自 `RecipeWorker` 的類別，可以覆寫其中的方法來改變配方的效果。例如，我們可以覆寫材料消耗或產出生成的邏輯。

C# 類別實作（RecipeWorker_MakeAdvancedComponent）：

```
using System.Collections.Generic;
using RimWorld;
using Verse;

namespace MyMod
{
  // 繼承自 RecipeWorker，控制配方執行的特殊行為
  public class RecipeWorker_MakeAdvancedComponent : RecipeWorker
  {
    // 覆寫配方完成後產出生成的方法
    public override IEnumerable<Thing> ConsumeIngredient(Thing ingredient, RecipeDef recipe,
Map map)
    {
      // 假設此配方有一項關鍵原料需要部分返還
      if (ingredient.def.defName == "AdvancedCircuit" && ingredient.stackCount > 0)
      {
        // 取出一些材料不消耗，作為返還（例如每個材料返還一半）
        int returnCount = ingredient.stackCount / 2;
        if (returnCount > 0)
        {
          Thing returned = ThingMaker.MakeThing(ingredient.def);
          returned.stackCount = returnCount;
          // 將返還的材料掉落在工作桌附近
          GenPlace.TryPlaceThing(returned, ingredient.Position, map, ThingPlaceMode.Near);
        }
      }
      // 呼叫基底實作進行正常的材料銷毀
      //（注意：基類 RecipeWorker 的 ConsumeIngredient 預設是將材料 Destroy）
```

```
            base.ConsumeIngredient(ingredient, recipe, map);
            // 沒有產出要額外返回，因此這裡直接 yield break;
            yield break;
        }

        // 可根據需要覆寫其他方法，例如檢查配方可用性、產生產物等
        // public override IEnumerable<Thing> MakeRecipeProducts(...) { ... }
    }
}
```

上面的 `RecipeWorker_MakeAdvancedComponent` 繼承自 `RecipeWorker`，我們示範性地覆寫了 `ConsumeIngredient` 方法。這個方法在每個原料被消耗時呼叫。透過自訂邏輯，我們檢查特定原料（假設叫做 AdvancedCircuit），將其中一半數量作為**返還**：生成相同的物品掉落在地上，而其餘部分才真正消耗掉[9]。最後調用 `base.ConsumeIngredient` 來執行默認的銷毀操作。

我們也可以選擇覆寫 `MakeRecipeProducts` 方法以改變產出。例如，可以在產出清單中額外增加一個副產品，或依照製作者技能調整產量。又或者對於手術類配方，可以覆寫 `ApplyOnPawn` 來增減成功率或副作用。總之，自訂 RecipeWorker 能讓配方的執行流程更加靈活。

應用： 例如在某些大型模組中，他們透過改寫配方的 workerClass 來實現複雜效果。例如有人想**無條件移除角色的任何部位**，可以將 **RecipeDef** 指向自訂的 `RecipeWorker_RemoveBodyPart` 類別（繼承自 `RecipeWorker` 或更特殊的基類），然後在其中override判定條件，忽略原先感染等限制[8]。這種方法需在 XML 中將 `<workerClass>` 指向新的類別，並確保該類別繼承自原本 RecipeDef 使用的基類（如從 `Recipe_Surgery` 衍生）[8]。總之，**XML 指向新的 workerClass，再撰寫對應C#類別並繼承原有類別，最後override需要的方法，即可改變配方行為**[8]。

註：使用自訂 RecipeWorker 時，如果配方是手術（對 Pawn 執行），可能需要繼承 `RecipeWorker` 之外更專門的類別，如 `RecipeWorkerSurgery`，並留意遊戲內對手術的一些特定處理。一般製造配方直接繼承 RecipeWorker 即可。

模組初始化與自訂類別的載入（Loader/Decorator 模式整合）

當我們在 Def XML 中引用了自訂類別後，遊戲載入時會經由反射自動抓取並使用這些類別。但在更大型的模組架構中，我們可能希望主動控制或記錄 Def 與其關聯類別的載入過程。例如，我們可以設計一個**資產載入器**（`IAssetLoader<T>`）介面來統一處理各類 Def 的初始化，把自訂類別的額外邏輯也一併註冊。還可以運用 **Decorator** 模式來包裝載入器，插入日誌紀錄等功能。

假設我們有以下介面定義，用於載入各種 Def 資源：

```
public interface IAssetLoader<T>
{
    IEnumerable<T> Load(); // 載入並返回一系列 T 資源（這裡T可以是ThingDef等）
}
```

我們可以為 RimWorld 的 Def 實作一個具體的載入器。例如 `ThingDefLoader` 透過遊戲的 `DefDatabase` 來取得所有 ThingDef，並對每個 ThingDef 做一些初始化處理：

```
using System.Collections.Generic;
using RimWorld;
using Verse;

public class ThingDefLoader : IAssetLoader<ThingDef>
{
  public IEnumerable<ThingDef> Load()
  {
    // 從 DefDatabase 獲取已載入的所有 ThingDef
    foreach (ThingDef def in DefDatabase<ThingDef>.AllDefsListForReading)
    {
      // 如果 ThingDef 指定了自訂 ThingClass，進行額外處理
      if (def.thingClass != null && def.thingClass != def.GetTypeDefaultThingClass())
      {
        // 例如：透過反射確保該類別的靜態建構子執行（如有必要）

System.Runtime.CompilerServices.RuntimeHelpers.RunClassConstructor(def.thingClass.TypeHandle);
        // 或登記到我們模組的某資料結構中，方便後續使用
        Log.Message($"[Loader] Registered ThingDef {def.defName} with custom class
{def.thingClass.Name}");
      }
      yield return def;
    }
  }
}
```

說明： 上述 `ThingDefLoader` 對每個 ThingDef 檢查其 `thingClass`。如果發現使用了自訂類別（且不同於預設類別），我們就透過 `RuntimeHelpers.RunClassConstructor` 強制執行該類別的靜態初始化，以防止有些類別的靜態內容尚未載入（選擇性步驟）。同時也輸出日誌，以示我們「註冊」了這個自訂類別。最後將 def 返回。實際環境中，可根據需要對類別進行更多初始化或將其存入集合，供模組其他部分使用。

接下來，我們可以使用 **Decorator** 模式來為任何載入器增加功能。例如，我們定義一個 `LoggingLoaderDecorator<T>` 來包裝 `IAssetLoader<T>`，在載入過程前後及每項資源時記錄日誌：

```
public class LoggingLoaderDecorator<T> : IAssetLoader<T>
{
  private readonly IAssetLoader<T> innerLoader;
  public LoggingLoaderDecorator(IAssetLoader<T> inner) => innerLoader = inner;

  public IEnumerable<T> Load()
  {
    Log.Message($"[Loader] Starting to load assets of type {typeof(T).Name}...");
    foreach (T asset in innerLoader.Load())
    {
      Log.Message($"[Loader] Loaded: {asset}");
      yield return asset;
    }
    Log.Message($"[Loader] Finished loading {typeof(T).Name}.");
```

```
  }
}
```

這個 `LoggingLoaderDecorator` 在開始和結束時各記錄一筆訊息，並對每個載入的項目輸出訊息。接下來，我們將它運用到前面的 `ThingDefLoader`：

```
// 在模組初始化時執行
IAssetLoader<ThingDef> baseLoader = new ThingDefLoader();
IAssetLoader<ThingDef> loaderWithLogging = new
LoggingLoaderDecorator<ThingDef>(baseLoader);

// 執行載入 ThingDef 的流程，並自動記錄日誌
foreach(var def in loaderWithLogging.Load())
{
  // 可以在這裡對載入的 def 進一步處理或驗證
}
```

當模組初始化時，我們將 `ThingDefLoader` 用 `LoggingLoaderDecorator` 包裝，然後調用其 `Load()`。這將觸發我們設計的載入過程：**載入所有 ThingDef** 並對有自訂類別的做註冊處理，同時透過 decorator 輸出詳細的日誌。日誌中會清楚顯示哪些 Def 被載入，以及它們綁定的自訂類別名稱，讓我們瞭解載入流程 10。若有任何一個 Def 的類別未找到或出錯，也能更快定位問題。

類似地，我們可以為 RecipeDef、WorkGiverDef 等實作各自的 Loader，或撰寫更通用的 Loader 來涵蓋多種類型 Def。透過介面和裝飾器模式，模組的初始化程式可以靈活組合需要載入的資源類型，並附加不同的功能（例如記錄、錯誤檢查等）。

最後，將這些載入流程整合到模組啟動流程中。例如，可以在模組的主類別（繼承自 `Verse.Mod`）的建構子或初始化方法中執行上述代碼。或者使用遊戲的靜態構造器機制：

```
[StaticConstructorOnStartup]
static class MyModInitializer
{
  static MyModInitializer()
  {
    // 於遊戲啟動時自動執行載入器
    IAssetLoader<ThingDef> loader = new LoggingLoaderDecorator<ThingDef>(new
ThingDefLoader());
    foreach (var def in loader.Load()) { /* ... */ }
    // 如有需要，也可對其他類型 Def 執行類似流程
  }
}
```

透過 `[StaticConstructorOnStartup]` 標記，當模組的 DLL 載入時，這段初始化程式會在遊戲Def資料庫加載完成後自動執行。我們的載入器就能取得所有 Def 並進行自訂類別的註冊與日誌。這種模式可以確保**Def 所參照的自訂類別**在模組初始化時就被考慮進去，避免遺漏任何需要特殊處理的邏輯。

(注意：以上程式碼主要作為示範。RimWorld 本身已會載入並初始化絕大多數 Def 資訊，自訂載入器更多是為了架構上的組織與擴充。例如記錄載入流程、進行額外驗證或與模組其他系統對接。如果沒有特別需求，模組不一定要自行遍歷 DefDatabase。)

VSCode 開發與 DLL 編譯的模組整合流程

採用 Visual Studio Code 進行 RimWorld 模組開發，我們通常需要手動設置專案來編譯 DLL。以下是基於前述內容的一般開發流程，以及模組目錄結構的說明，以便將自訂類別和 Def XML 串接在一起：

1. **建立模組檔案夾結構**：在 RimWorld 的 Mods 資料夾下創建新模組目錄，例如 `MyMod/`。其子目錄結構如下（示意）：

```
MyMod/
├── About/
│   ├── About.xml            # 模組描述檔，包含名稱、作者、版本等資訊
│   └── Manifest.xml         # (可選) 模組清單，RimWorld 1.1+ 可在此列出DLL
├── Assemblies/
│   └── MyMod.dll            # 編譯產出的 C# 程式集
├── Defs/
│   ├── ThingDefs/
│   │   └── MyThings.xml           # 定義 ThingDef，包含 thingClass 等
│   ├── RecipeDefs/
│   │   └── MyRecipes.xml          # 定義 RecipeDef，包含 workerClass 等
│   └── WorkGiverDefs/
│       └── MyWorkGivers.xml       # 定義 WorkGiverDef，包含 giverClass 等
└── Source/
    ├── MyMod.csproj          # VSCode 專案檔（C# 工程檔）
    └── MyMod                 # 原始碼資料夾（可多層級命名空間）
        ├── Building_BlinkingLamp.cs
        ├── WorkGiver_FillVodkaBarrel.cs
        ├── JobDriver_FillVodkaBarrel.cs
        ├── RecipeWorker_MakeAdvancedComponent.cs
        └──── (其他 .cs 原始碼檔)
```

在 `About.xml` 中，確認有 `<Assemblies>` 節點列出 `MyMod.dll`（或在 Manifest.xml 中列出），以確保遊戲載入模組時會讀取我們的 DLL。如：

```
<Assemblies>
  <li>MyMod.dll</li>
</Assemblies>
```

RimWorld 啟動時會自動載入 Assemblies 資料夾下的 DLL 檔案 [2]。

1. **設置 VSCode 專案**：透過建立 `.csproj` 檔案（如上面的 `MyMod.csproj`），包含對 RimWorld 所需函式庫的引用，例如：
2. `Assembly-CSharp.dll`（核心遊戲邏輯）
3. `UnityEngine.dll` 和 `UnityEngine.UI.dll`（Unity 引擎）

4. `Verse.dll`（RimWorld 基礎命名空間，在 Assembly-CSharp 中，其實引用 Assembly-CSharp 即可獲得 Verse、RimWorld 等）
5. 以及 .NET Framework 3.5/4.x 的必要引用 (RimWorld 基於 .NET Framework 4.7.2左右)。

設定專案的輸出路徑（OutputPath）指向 `..\Assemblies\MyMod.dll`，這樣每次建置時會自動覆蓋模組 Assemblies 下的 DLL 檔案 [10]。在 VSCode 中安裝 C# 擴充套件，載入專案後，可利用命令或 build task 進行編譯。如果使用了 VSCode RimWorld 模組範本，按 **F5** 甚至可以自動編譯並啟動 RimWorld 進行測試 [10]。

1. **撰寫程式碼與 XML**：按照前述各範例，將自訂類別的程式碼寫入 `.cs` 檔案中，命名空間和類別名稱需對應在 XML 中填寫的字串。確保類別宣告為 `public`，以便 RimWorld 的反射機制可以存取。接著在 Def 資料夾的對應 XML 檔中加入/修改定義，插入 `<thingClass>`、`<workerClass>` 等標籤連結到我們的類別。

例如，在 `MyThings.xml` 中加入我們的 `<ThingDef defName="BlinkingLamp">` 定義，內容如範例所示；`MyWorkGivers.xml` 中加入 `<WorkGiverDef defName="FillVodkaBarrel">` 定義等。確保 defName 唯一且不要與遊戲原生或其他模組衝突。

1. **編譯與測試**：使用 VSCode 編譯專案，產生 `MyMod.dll`。啟動 RimWorld，勾選我們的模組。如果專案配置正確，遊戲將載入 XML 並嘗試定位我們 DLL 中對應的類別。進入遊戲後，可以打開開發者模式觀看日誌，或查看控制台輸出，以確認沒有出現「類別未找到」之類的錯誤。此前我們實作的 LoggingLoaderDecorator 也會在日誌中列出載入的 Def 與類別資訊 [10]，方便確認一切正常。

2. **驗證功能**：在遊戲中嘗試使用我們定義的物件和功能。例如生成 **BlinkingLamp** 建築，看其是否按預期閃爍；檢查殖民者是否會執行 **填充酒桶** 工作；執行 **先進元件製作** 配方，觀察材料返還情況。若行為不如預期，利用開發者模式的日誌和斷點（可透過VSCode附加除錯）調試修正。

透過上述步驟，我們將 **XML Def** 與 **C# 程式碼** 成功整合：XML 決定了遊戲中物件、配方、工作的**靜態數據**和關聯類別，而 C# 類別則提供**動態行為邏輯**。VSCode 提供了輕量的環境來編輯與編譯這些程式碼，透過適當的專案設定，可以一鍵編譯並啟動遊戲進行測試，極大地方便了開發流程 [10]。

最後，請記得遵循 RimWorld 社群建議：**僅在需要改變遊戲預設行為時才使用自訂 Def 類別**。對於單純新增數據或輕微改動，優先考慮 DefModExtension、Comp 等更簡單且相容性高的方式 [11] [7]。若確實需要覆寫底層邏輯，自訂 class 屬性是不錯的途徑，但也要注意可能與其他修改相衝突（例如兩個模組若試圖修改同一 Def 的 class，將無法相容 [12]）。在充分理解原版類別運作的前提下運用上述技術，將能為您的 RimWorld 模組增添獨特且強大的功能！

參考來源：

- RimWorld Modding Tutorial – XML 與 C# 連結： [1] [2]
- RimWorld Modding Wiki – 常見 Def 可用的 class 屬性（如 conditionClass 範例） [7]
- Ludeon 官方論壇 – 模組開發討論（如 WorkGiverDef 和 RecipeDef 自訂類別的建議） [4] [8]
- RimWorld Modding Files – HediffDef 範例及註解（hediffClass 用法） [6]
- VSCode RimWorld Mod 開發範本 – 專案結構與編譯說明 [10]

---

[1] [2] [11] [12] Modding Tutorials/Linking XML and C - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/Linking_XML_and_C

[3] User:Alistaire/Tag:RecipeDef - RimWorld Wiki
https://rimworldwiki.com/wiki/User:Alistaire/Tag:RecipeDef

4 5 ThingDef for ThingRequest, custom WorkGiver
https://ludeon.com/forums/index.php?topic=25565.0

6 9 Hediffs.xml
https://github.com/RimWorldMod/RimworldModdingFiles/blob/caeb934fca694c90dc645955f15b4a7e9551e21a/Defs/HediffDefs/Hediffs.xml

7 Modding Tutorials/Modifying defs - RimWorld Wiki
https://rimworldwiki.com/wiki/Modding_Tutorials/Modifying_defs

8 Help a newbie mod creator understand something.
https://ludeon.com/forums/index.php?topic=48712.0

10 GitHub - Rimworld-Mods/Template: Rimworld Mod Template for Visual Studio Code.
https://github.com/Rimworld-Mods/Template