Hosting the .NET Composition Primitives

陈贞宝，现任西安尤埃信息技术有限公司（http://www.uishell.com）架构师。该公司成立于 2008 年 5 月份，专注于尤埃开放服务平台和尤埃 SaaS 引擎云计算产品开发。

尤埃开放服务平台（XAUI Open Service Platform，UIOSP）是一个移植了 OSGi 规范的动态插件化与模块化平台，支持插件化与模块化、SOA 和模块扩展。

尤埃 SaaS 引擎（XAUI SaaS Engine，XSE）是一个 SaaS 应用商店开放平台。该平台是面向 SaaS 运营商、SaaS 提供商和 SaaS 消费者三个角色的 PaaS 云计算平台，其模式为“SaaS 运营商负责平台运营，SaaS 提供商利用平台提供的开发工具包基于 VS2008SP1 开发 SaaS 应用并上传，SaaS 消费者在应用商店挑选、购买并使用 SaaS 应用”。该平台由应用商店网站、应用开发工具包和应用虚拟运行环境构成。

1 目录

2 介绍

.NET 框架 4.0 版本包括了支持面向组件编程的类。

这些称为组合基元的最底层的类代表了：

- 具有相互组合能力的组件；
- 支持丰富元数据的组件定义；
- 面向组件库的通用查询接口。

在组合基元之上的层次是基于类型的组件基础库，这些库被称为面向特性的编程模型。这个模型包含的功能为：

- 允许将普通.NET 类型标记为组件定义的特性；
- 通过读取、初始化和处理特性类型实现组合基元的类型；
- 从不同地方发现和装载基于特性的组件定义的组件目录。

使用 CompositionContainer，包括.NET 框架，基于特性编程模型是一个完全组合的系统：

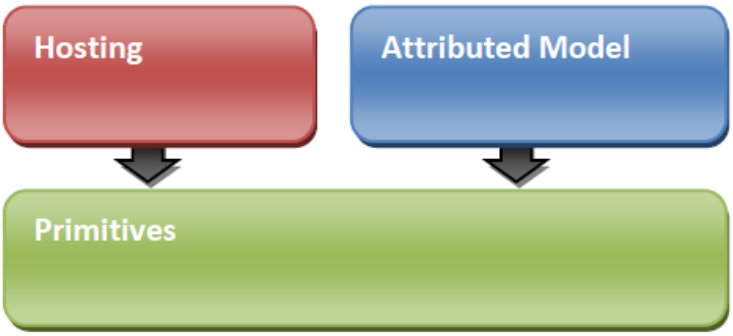

Figure 1 – .NET Composition Architecture

在.NET 框架有很多组合系统，比如：

- 支持解耦应用系统结构的 IOC 容器；
- 初始化一个基于文本对象描述的对象的 XAML 解析器；
- 像使用一序列特有的概念以支持一族场景的 WWF 和 WCF 的领域相关框架；
- 允许被第三方扩展的插件框架；
- 以命令方式组合和处理对象的编程语言。

这些框架的共同点是具有获取独立开发的软件单元并组合他们来实现功能的能力。

在大多数组合框架，这是他们共有的很好的主意，基于特性的编程模型也是如此，而且此外，组合基元以能够不依赖 CompositionContainer 来使用的方式进行编译。

基于特性的编程模型和 CompositionContainer 主要关注于应用程序扩展性，这一般也是插件框架所支持的。

2.1 Unity 例子

这个文档将微软 P&P team 的 Unity IOC 容器作为一个宿主组合系统的例子。Unity 的主要目标是支持解耦式应用系统架构。

宿主组合基元使基于 Unity 的应用程序以和其它.NET 4.0 框架的应用系统一致的方式暴露出扩展点。

这包括：

- 使用 System.ComponentModel.Composition 特性标记的类作为 Unity 组件；
- 使用搜索不同组件源的组件目录——比如，使用 DirectoryCatalog 来发现一个目录中程序集的所有组件。

3 组合基元

组合基元是位于 System.ComponentModel.Composition.Primitives 命名空间的.NET 类型。

组合基元的作用是来指定可以组合在一起的组件来创建有用软件。

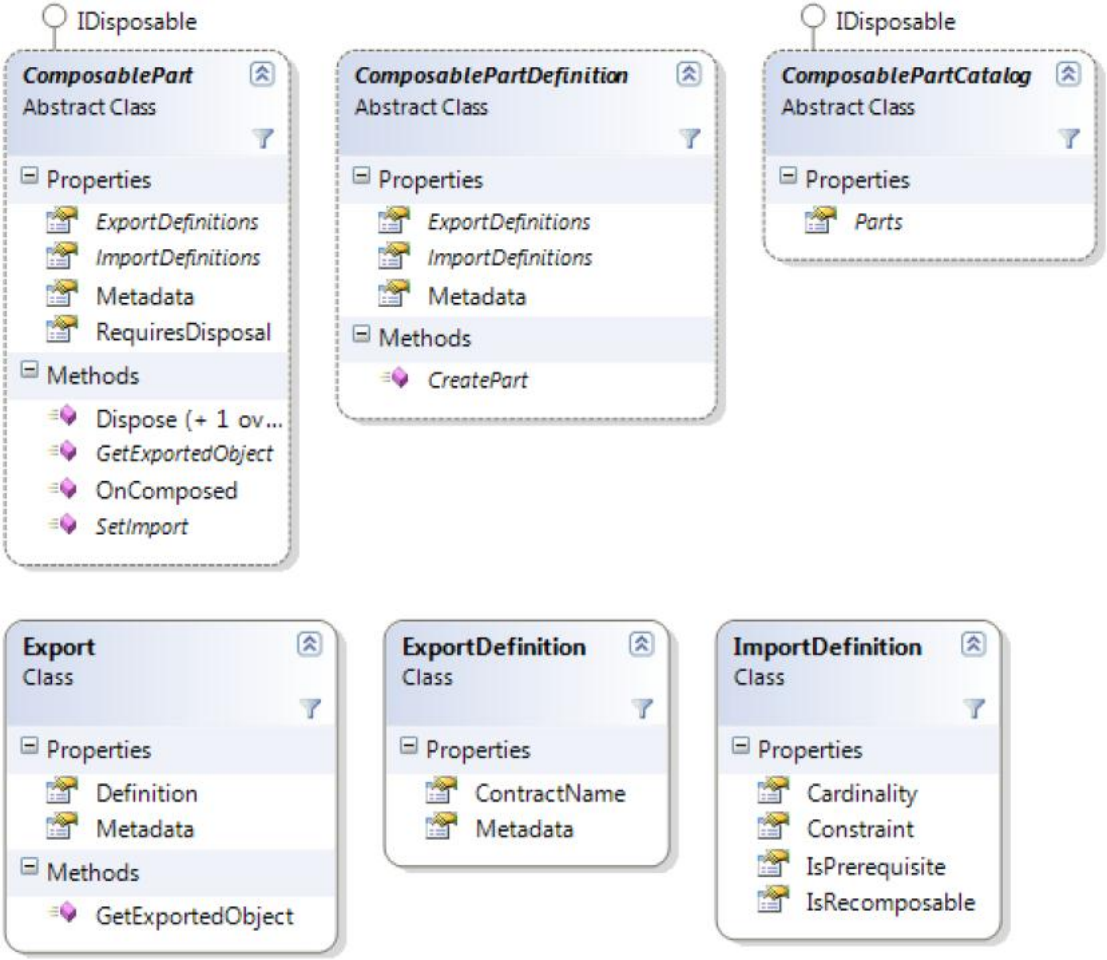

Figure 2 – Composition Primitives Classes

3.1 ComposablePart

组合基元中最关键的类是 ComposablePart。一个 ComposablePart 是一个处于激活状态的、正在执行中的软件组件实例。

ComposableParts 作为 exports 向其它组件提供功能且通过 imports 来使用其它组件的功能。

每一个 ComposablePart 通过它的 ImportDefinitions 和 ExportDefinitions 集合描述了它自己的 imports 和 exports。

imports 和 exports 代表的实际对象可以通过使用 SetImport 和 GetExportedObject 方法与 ComposablePart 交互。

3.2 ExportDefinition

一个 ExportDefinition 是由基于字符串的契约名称和称为 Metadata（元数据）字典形式的额外信息组成的结构体。

每一个附加到一个 ComposablePart 的 ExportDefinition 描述了这个 Part 的一个功能。比如，一个提供了 C#和 VB.NET 源码解析器的 Part 可能通过两个单独的 ExportDefinition 来表示。

在这两个例子，契约名称很可能都是“parser”，但是每一个 ExportDefinition 将使用元数据来进一步指定能够被解析的编程语言的细节。

3.3 ImportDefinition

ImportDefinition 描述了一个依赖。

它包含了一个称为约束的功能，约束基于 Import 的需求来选择 ExportDefinition。比如，一个 C#编译器组件的 import 约束可能是：

(ExportDefinition ed) => ed.ContractName == “parser” && ed.Metadata[“language”] == “C#”

因为约束是一个可在运行时解析的 Linq 表达式，因此一个宿主执行这个表达式来获取需要提供的 exports 并不总是必须的。这个场景将在以下有关 ConstractBasedImportDefintion 类型中讨论。

Linq 是一个指定任意复杂依赖要求的无二义的语言。它可以胜任这个任务是因为需要被完全校验的需求在适当时候约束了其执行。

3.4 ComposablePartDefinition

在给定系统可以被创建的 ComposablePart 的类型是使用 ComposablePartDefinition 来描述的。

一个 ComposablePartDefinition 是一类 ComposablePart 的工厂。一个 ComposablePart 的新实例可以通过使用 CreatePart 方法来创建：

```
ComposablePartDefinition consoleLoggerDefinition = …
var consoleLogger = consoleLoggerDefinition.CreatePart();
```

ComposablePartDefintion 通过 ImportDefinitions、ExportDefinitions 和 Metadata 属性来描述他们创建的 ComposableParts。这些与这个定义能创建的 ComposableParts 的属性相匹配。

3.5 ComposablePartCatalog

ComposablePartDefinitions 在 ComposablePartCatalog 中分类存在，这可以优化通过 IQueryable<ComposablePartDefinition> Parts 属性查询合适定义的性能。

4 编程模型的角色

组合基元可以简单的通过继承上述描述的各个类的方式来使用。直接使用这种模式是笨重的，因此开发人员通常是通过一个编程模型来与这些元素交互。

编程模型主要关注如何使开发人员以一种与领域相关目标相近的方式创建 ComposablePartDefinition。

大部分编程模型都优化了将编程语言的类型映射到一个使用 exports 和 imports 的 ComposablePartDefinition 的任务。

5 基于特性的编程模型

基于特性的编程模型使用特性来将一个普通.NET 类型定义映射到一个 ComposablePartDefinitions 的实例及其相关类型。

5.1 基于特性的定义

一个简单的特性定义可能是：

```
[Export(typeof(IShape))]
public class Square : IShape
{
    public int NumberOfSides { get { return 5; } }
}
```

当这个类被基于特性的编程模型读取时，它将转换为一个 ComposablePartDefinition，这个 ComposablePartDefinition 的 ExportDefinitions 集合里只有一个 ExportDefinition。

这个 ExportDefinition 将有一个与 CompositionServices.GetContractName(typeof(IShape))等价的契约名称。

当这个 ComposablePartDefinition 的 CreatePart 方法被执行，那么一个 Square 的实例将被创建。

当 GetExportObject 被用来获取一个 IShape 的 export，这个 Square 的实例将被返回。

像 Export 这样的元数据在基于特性的编程模型中用于声明 Exports、Imports 和 Metadata 项。

5.2 使用基于特性的编程模型

基于特性编程模型的主要接口是：

- TypeCatalog；
- AssemblyCatalog；
- DirectoryCatalog。

当以上组件目录类型以一种搜索基于特性的定义的方式进行初始化时，所有 ComposablePartDefinition 定义会在 ComposablePartCatalog.Parts 保存。

暴露组合基元的特性和类型将完全封装在组件目录提供的 ComposableDefinition 中。这里强烈不建议自己尝试将类型和 ComposableDefinition 进行映射。

6 高层整合注意点

宿主必须通过 ComposablePartCatalog 类型与组合基元进行交互。

首先需要解决的问题是知道组件目录如何提供给宿主。

可以预料到的是，一个宿主框架的用户一般都系统提供自己的组件库——这将允许他们除了使用基于特性的编程模型之外的其它编程模型。

即使宿主是一个应用程序而不是一个框架，我们也可以通过 ComposablePartCatalog 来与需要的类型进行交互。

以下代码片段演示了配置 Unity 来提供来自 TypeCatalog 的组合片段，并然后通过 Unity 接口使用这些片段。

```
IUnityContainer container = …
Container.AddNewExtension<CompositionPrimitivesIntegration>();
var catalog = new TypeCatalog(typeof(TestPart));
container.Configure<CompositionPrimitivesIntegration>().
    RegisterCatalog(catalog);
var part = container.Resolve<ITestPart>();
```

7 Exports 交互

ComposableParts 使用选择 ExportDefinition 的查询来表示依赖。这些查询是一些附加 ImportDefinition 的断言。

宿主必须使用那些匹配 ImportDefinition 断言的值的来满足 imports。

这些值可以是其它 ComposablePart 的 exports，或者是宿主提供的服务。不管哪一种情况，宿主必须理解一个 ImportDefinition 代表了什么。

7.1 执行一个约束

在一个闭合的或简单的系统，创建一个 ExportDefinition 表示系统每一个可用的 Export 是可能的（假定具有映射 ExportDefinition 到使用的对应 Export 的能力）。

在这些情况，以下代码将查询正确的 exports：

```
var allExportDefs = …
var importDef = …;
var constraint = importDef.Constraint.Compile();
var matchingExportDefs = allExportDefs.Where(ed => constraint(ed));
```

7.2 ContractBasedImportDefinition

很多 import 仅是基于 ExportDefinition 的三个方面的简单查询：

- 相等的契约名称（ContractName）；
- 相等的依赖的类型标识（RequiredTypeIdentity）；
- Metadata 键值的子集。

这些 import 定义可以使用一个 ContractBasedImportDefinition 表示。知道这个类型的宿主可以校验 import 定义并将其转换到特定的类型以访问通用的数据项：

```
var cbid = importDef as ContractBasedImportDefinition;
if(cbid != null)
{
    Console.Write(“ContractName={0}”, cbid.ContractName);
    Console.Write(“RequiredTypeIndentity={0}”, cbid. RequiredTypeIndentity);
    foreach(var requiredKey in cbid.RequireMetadata)
    {
        Console.WriteLine(“Require {0}”, requiredKey);
    }
}
```

7.2.1 契约名称（ContractName）

契约名称（ContractName）是描述一个导出或导入项的任务的唯一字符标识。

在一个 ContractBasedImportDefinition 的 ContractName 属性必须与待校验的 ExportDefinition 的 ContactName 使用字符串相等比较进行匹配。

一个样值可能是“MyCompand-AppName-ToolBarItem”。

单独使用 ContractName 并不确保一个导出和一个导入间的兼容——导入的其它要求可能使用 RequiredTypeIdentity 和 RequireMetadata 值进行表示。

7.2.2 依赖的类型标识（RequiredTypeIdentity）

RequiredTypeIdentity 表示导入的一个需求，即导出的值可以转换为一个指定的类型。

一个 ContractBasedImportDefinition 的约束使用 ExportDefinition 的字符串值 Metadata 中键为 RequireTypeIdentityMetadataName 的字符串值来匹配 RequiredTypeIdentity。

RequiredTypeIdentity 使用 CompositionServices.GetTypeIdentify 方法生成。单独使用这个值还不足以装载所需类的 System.Type 实例。为了映射一个类型，还需：

- 扫描已知所有类型来创建一个从“Type Identity”到类型的 Lookup；或
- 使用 ImportDefinition.Metadata[ImportTypeAssemblyQualifiedName]值作为提示。

7.2.3 RequiredMetadata

RequireMetadata 在这里是用来指定一些必须在 ExportDefinition.Metadata 字典出现的键的字符串。

这些键只对特定的导入有意义。

7.2.4 使用类型和名字交互的特殊情况

很多宿主环境使用一个基于类型、名称或者二者组合的键来定位服务。

当 ContractName 和 RequiredTypeIdentity 为相同字符串值时，这可以认为是一个默认的契约名称：执行过程可以推断仅靠类型就足够了。允许一个‘null’的名字选择器的宿主可能认

为这在组合基元是等价的。

比如，Import 定义包含：
ContractName = “MyNamespace.IMyInterface”
ExportedTypeIdentity = “MyNamespace.IMyInterface”
可由 Unity 集成转换成如下的等价查询：
IUnityContainer container = …
var result = container.Resolve<MyNamespace.IMyInterface>();

当 RequiredTypeIdentity 为 null，则仅与 ContractName 相关。宿主可以返回任何类型的实现，这些类型是指定 ContractName 的规格所允许的。

Unity 一般不支持没有关联类型的命名实例的概念。在此将执行一个基本的映射使得这样的请求是假设一个 System.Object 的注册。

比如，Import 定义包含：
ContractName = “Sender.Timeout”
ExportedTypeIdentity = null
可由 Unity 集成转换成如下的等价查询：
IUnityContainer container = …
var result = container.Resolve<object>(“Sender.Timeout”);

7.3 约束解析或转换

选择导出的一个第三方操作可能应用于现有查询机制的系统：约束可以被转换成一个宿主系统的查询格式。

大多数系统不需要这个能力且它们可以使用 ContractBasedImportDefinition。

7.4 使用宿主提供服务

为了向 ComposablePart 提供不来自其它 Part 的导入值，一个简单的系统可能是创建简单的 ExportDefinition 来表示宿主提供的所有服务，然后反过来在这些项中评估 ImportDefinition.Constraint 断言。

更加复杂的宿主，或那些执行一系列开放可用的服务，一般使用 2 个阶段过程：
A）将 ImportDefinition 转换成宿主内部使用的参数；
B）为所有结果构建 ExportDefinition 并将这些提供给片段（Part）。

8 标准 Part 处理而不是对象图连接

这个文档的前提是不同的片段必须能够做一些关于它们环境具有合理性的假设。

任何来自在任何宿主下相同对象图的给定集合的组件是不必要的。

后者的限制存在是因为：
● 组合基元是为了在开放系统中使用，因此一个作者可以一直不同假设一个片段会收到依赖的特殊配置；
● 使用一个通用 API 集合的扩展（比如基于特性的编程模型），但面向他们将被测试的特定的宿主环境。

9 宿主必须得行为

以下是值得注意的一个宿主必须满足的需求的子集，这是为了成功宿主.NET 组合基元。

9.1 遵从 ImportDefinition 的原则

ImportDefinition.Constratin 是针对于 ExportDefinition 的断言。以下代码应该从不抛出一个异常。

```
public void SetImports(ImportDefinition import, IEnumerable<Export> values)
{
    var predicate = import.Constraint.Compile();
    foreach(Export e in values)
    {
        if(!predicate(e.Definition))
        {
            throw new ArgumentException("Export does not match constraint.");
        }
    }
}
```

9.2 ComposablePart 生命周期

一个 ComposablePart 的生命周期必须遵守以下顺序：
A）通过调用 ComposablePartDefinition.CreatePart 或者使用 new 构建一个具体的实现进行创建；
B）通过调用 SetImports 来满足依赖的先决条件；
C）现在允许通过 GetExportedObject 来访问导出对象，但这些对象必须直到 Part 被激活才可以使用；
D）通过调用 SetImports 来满足非先决条件；
E）通过调用 OnComposed 完成激活过程；
F）来自该 Part 的导出对象现在可以使用了；
G）使用 IDisposable.Dispose 进行清理。
清理是在处理过程中任何地方有效的行为，这是需要的。

9.3 先决依赖的完整组合

如果一个 Import 是一个前提，那么提供给这个 Import 的任何值都必须完全组合。这是一个传递性需求，因此任何依赖的依赖必须被完全组合。

9.4 要求的创建策略

任何有一个为 CreationPolicy.NonShared 的 RequiredCreationPolicy 的 ImportDefinition 必须提供一个唯一的实例。在 ComposablePart 生命周期，没有任何的调用者可以与提供的实例进行交互。

9.5 片段创建策略

如果一个 ComposablePartDefinition S 有一个 CreationPolicy.Shared 的创建策略，那么：

- 非片段实例 p，如果能够访问 S 的一个实例 s，那么它应该能够访问 S 的另一个实例 s’；
- 如果 s 能够访问非片段实例 p，那么 S 的另一个实例 s’也应该能够访问 p。

9.6 重新组合

如果一个导入定义没有标记为可重新组合（IsRecomposable == true），那么 SetImports 仅可能在第一次 OnComposed 被调用之后才能调用。

10 宿主定义的行为

以下是 CompositionContainer 类型的行为功能列表。这些被考虑为宿主的职责且不一定被其它组合基元宿主完全一致的实现。

10.1 懒深对象图初始化

虽然 ComposablePart 对应的接口允许对象图懒深初始化，但并不要求提供给片段的所有依赖都是晚初始化的。

当性能特点不同时，一个立即初始化依赖的实现仍然是正确的。

10.2 循环依赖检测

宿主不要求允许任何类型的循环图。

这里建议，为了简化诊断体验，宿主需要执行循环依赖检测。

10.3 拒绝

Import 访问 Export.GetExportedObject 方法必须总是返回一个值或者抛出一个异常。在这个环境，异常通常是不能恢复的。

宿主必须尽可能的提供导出给不能访问它的片段。如果宿主支持晚组合，处理这个行为必须要求进行深依赖分析。那些不能执行深分析来确定一个导出是否满足的宿主应该期待立刻组合。

10.4 默认值和基数性

当一个导入指定仅需一个实例，宿主可能使用它自己的算法来确定从可能满足依赖的可用实例中选择什么实例。

11 总结

A）组合基元：具有组合能力的组件、组件定义和组件目录
B）基于特性的编程模型：基于特性标记组件定义、处理（读取、初始化和处理）特性实现组合基元的类和组件目录
C）导出定义：约束名称、元数据（RequiredTypedIdentity）
导入定义：约束，是否先决条件，是否可重新组合，基数
组件定义：导出定义，导入定义和元数据；CreatePart 方法
导出：定义，元数据，和 GetExportedObject
组件：导出定义，导入定义，元数据和是否需要清理；GetExportedObject、OnComposed、SetImport 和 Dispose 方法
组件目录：所有组件集合