组件系统设计文档

# 概要

组件系统是一套可视化脚本，一般在商业引擎中扮演资源管理，AI脚本编辑，剧情过场编辑等角色（诸如cryengine的flow editor和unreal的粒子编辑器）。它的作用是将脚本指令和数据驱动图形化，并提供定制的，基于对话的编辑方式。通过图形化使数据和数据之间的关系更加明朗，通过UI使参数调试更为直观，定制的数据检查也可以减少配置数据出错的概率。

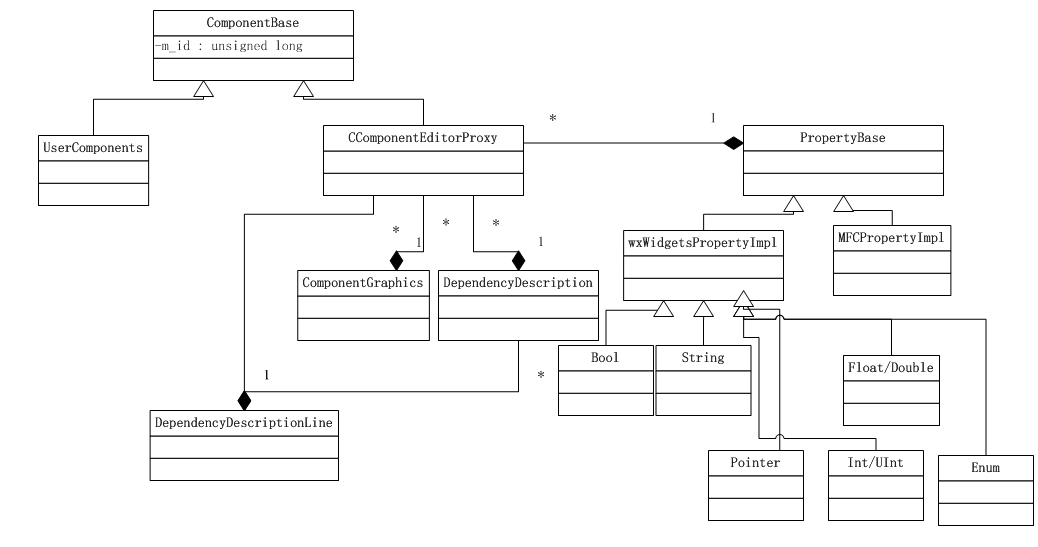
# 设计理念

1. 通用性：作为第三方库可以迅速的应用于任何游戏项目。
2. 扩展性：支持用户自己扩展系统功能，包括自定义新组件，自定义新属性类型，自定义UI库（可以用任意的UI库实现指定的接口）。
3. 可视化：所有的操作均为视窗操作，并能达到和直接修改数据同样的功能。用图形进行数据结构建模，支持模块之间的相互引用。
4. 自动化：建模完成以后，对于任务的增加，均无需做更多操作，数据可视化映射、数据导入导出、组件创建销毁均为自动完成。
5. 基于对话：对于每一个可使用的属性都有对应的描述或者提示，对于操作失败有明确的提示信息。
6. 数据加速：用二进制数据代替文本配置，用模板+数据代替目前的重复文本结构配置。

# 目标

1. 通用的数据驱动解决方案，对任何项目的数据驱动需求均适用。
2. 高效自动化的数据驱动方式，任何需求变动均不需要修改现有代码。
3. 图形化数据以及数据关系，将数据更直观的呈现给用户。
4. 友好的人机交互，高效智能的引导用户解决问题。
5. 时刻遵循对修改关闭，对扩展开放原则。使项目最大程度的保持简洁易懂。

# 系统架构



1．ComponentBase

组件化的根基类，维护最基本的组件需求，譬如用于交互的m\_id以及GUID等接口。这个类相当于一个基本协议用于将数据交互于宿主项目和editor。它只包含一个32位成员m\_id。

2. UserComponents

宿主项目所继承的自定义组件。

3. CComponentEditorProxy

Editor所使用的一个代理类，它可以把任何一个UserComponent用图形化的方式表达出来，它同时负责了在Editor中进行绘制和数据导入导出。

4. DependecyDescriptionLine

用于绘制组件之间的逻辑连线，以及响应Editor对连线的操作。它同时存在于CComponentEditorProxy和DependencyDescription中，原因是：考虑如下情况，一根DependecyDescriptionLine同时连接了两个CComponentEditorProxy，当我删除任意一个Proxy时，这根Line都需要被删除，所以这里两个地方都需要能引用到这根line。

1. DependecyDescription

用于描述组件之间的依赖关系，同时也带有一部分用于绘制时的逻辑数据。

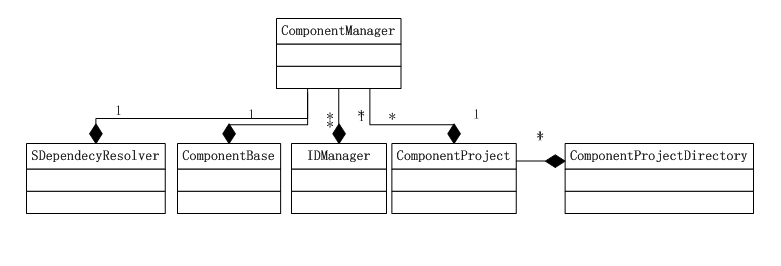
1. ComponentGraphics

用于绘制组件的模块。

1. PropertyBase

用于抽象组件的一个成员变量，其中包含了用于绘制或者数据交互的基本信息，如：类型，变量名，显示名，颜色等等。同时它包含了一个void\*成员用于指向该变量真正的值。

总结：以上架构为整个系统的数据架构，用组件来模拟类，用propertybase来模拟变量，用dependecy来模拟类之间的复合关系。



1. ComponentManager

对整个组件系统进行管理和控制的类，不论是Editor还是宿主工程，几乎任何操作都要通过该类直接或者间接的进行。它包含了所有组件系统操作所需要的数据，也提供了所有的操作接口。

对于宿主工程，需要利用该类进行数据的导入或者导出。

对于Editor，出了需要利用该类进行数据导入导出以外，还需要利用它提供的接口完成特定的操作，诸如：创建一个全新的组件实例（涉及到ID分配，组件实例以及属性的克隆），查找组件的位置，查询组件的父子关系，解决ID冲突，维护数据升级或者更新等等。

1. ComponentProject

用于抽象当前的组件工程。对整个系统遵循 工程->文件 的架构。它包含了以下功能点：

1. 因为包含了整个工程的信息，所以可以通过它查询到某一个或者一类组件的相关信息，我们称之为静态记录。
2. 其中包含的CComponentProjectDirectory用于记录其文件目录结构，为Editor实现MVC结构提供了便利。
3. 记录了数据维护的信息，当用新的代码结构去加载老的XML数据时，Editor会根据加载中的不匹配提示用户是否进行重新绑定，而用户的决定就保存在这里，这样下次再遇到同样的不匹配时，会自动根据这里的信息进行自动操作。

打个比方：假设代码中有个成员变量是bool m\_bTest，此时Editor会将该信息保存到XML中，当我在代码中将这个变量重命名为m\_bNewTest时，为了让Editor将之前保存的m\_bTest重新映射到m\_bNewTest上，Editor会进行提示，用户可以选择是否重定位该属性，如果是，则将XML中的值重新关联到m\_bNewTest，如果选否，则遗弃XML的值。

1. IDManager

主要用于创建新的组件时，为它分配全局唯一的一个值。同样的，在删除一个组件时，要及时的将这个ID进行回收再利用。

1. ComponentBase

组件的基类实例。

1. SDependencyResolver

用于连接组件之间逻辑关系。只为宿主工程所使用。

比如Editor导出了如下信息：组件A有一个对于组件B的依赖（有一根DependencyDescriptionLine连线从A到B，这个信息保存在组件A之上）。当宿主工程加载组件A时，读取到该依赖信息，由于依赖信息需要同时存储在所连接的两个组件之上（参考之前的DependencyDescriptionLine），但此刻非常有可能B还没有被加载，所以这个时候我们先将这个依赖信息抽象化到SDependencyResolver中保存，等所有的组件都加载完了，我们再调用CComponentMananger::ResolveDependency来读取之前保存的依赖信息进行导入。

# **数据映射流程**

对于整个系统，组件是基本数据单位;对于一个组件，属性是基本数据单位。

我们将属性分为以下类型：

1. 基本类型: bool, int, uint, string/wstring, float, double.他们都是最基本的数据单位，只需要维护其本身的值即可。
2. 枚举类型：enum。本质上来说它就是一个int数据，但是它具有可穷举值以及显示字符串的概念。因为其特殊性，必须特别的处理该数据的序列化和反序列化。
3. 列表类型：list。它是一个数据容器，能包含同种任意的类型数据，并且能删除或者添加一个新的数据实例。
4. 指针类型：ptr。它是一个抽象数据，可以被实例化成任意一个其他类型的数据或者组件。
5. 依赖： 一个指向其他组件的指针成员变量。
6. 依赖列表

我们针对上面四种类型的数据做了对应的四种映射流程：

1. 针对基本类型，列表类型，指针类型，使用宏：

#define DECLARE\_PROPERTY(serializer, property, editable, color, displayName, catalog, tip, parameter) DeserializeVarialble(property, &serializer);

参数含义：

serializer: 用于接收该数据映射信息的数据流，一般在导出信息的时候由外部通过组建基类构造函数传入。

property: 该数据变量。我们通过这里获取映射所需要的所有变量信息，如类型、变量名、数据大小。

editable: 用于指出是否允许用户在Editor中对该数据进行修改。

color: 用于指定该数据在Editor中显示的颜色。

displayname: 用于指定该数据在Editor中显示的字符串名称。不能为NULL。

catalog: 用于指定该数据在Editor中显示的类目。可以为NULL。

tip: 用于指定该数据在Editor中被选中时显示的提示信息。可以为NULL。

parameter: 用于指定该数据的扩展参数，这个地方是为未来扩展而定义的一个字符串协议。格式为 关键字:值，关键字：值……。支持的关键字存储在PropertyPublic.h的EUIParamterType中。在CPropertyDescriptionBase::AnalyseUIParameter中被解析。

宏通过调用模板函数来对基本类型，列表类型和指针类型进行区分，并进行对应的特殊处理。

1. 针对枚举类型，使用宏:

#define DECLARE\_PROPERTY\_ENUM(serializer, enumVariable, count, selfDefineStrArray, enumType, editable, color, displayName, catalog, tip, parameter)

参数含义：

其中除了count,selfDefineStrArray和enumType外其他的参数含义与上雷同。

count: 指定该枚举的值数量，用于之后解析传入的selfDefineStrArray数组。

selfDefineStrArray: 指定该枚举用于在Editor上显示的名称字符串数组，可以为NULL,意味着将使用枚举本身作为显示的字符串。

enumType: 枚举类型。用于导出给Editor标示。

1. 针对依赖类型，使用宏:

#define DECLARE\_DEPENDENCY(serializer, ptrProperty, propertyType, displayName, dependencyType)

其中除了dependencyType外其他的参数含义与上雷同。

dependencyType: 可以填入eDT\_Weak 或者 eDT\_Strong, 分别表示弱依赖和强依赖。弱依赖的意思是这个指针可以为NULL，强依赖的意思是这个指针必须不为NULL。

1. 针对依赖列表类型，使用宏:

#define DECLARE\_DEPENDENCY\_LIST(serializer, ptrProperty, propertyType, displayName, dependencyType)

与依赖类型一模一样，之所以要用不同的宏，纯属是为了在代码书写的时候更加明确。

属性映射原理：

1. 对于基本类型：

需要导出的信息有：

1. 一个bool标示符，表示导出的信息是变量还是依赖。

之所以需要这个操作是因为，宿主工程在导出和导入信息的时候，顺序是一致的。由于在Editor这边，属性和依赖分别被抽象成了PropertyBase和DependencyDescription，并保存在两个独立的列表里，当Editor导出信息时，必须要还原之前宿主工程导出信息的顺序，否则宿主工程无法正确导入。于是在此通过导出这个bool值，让Editor加载到CComponentEditorProxy::m\_pSerializerOrder之中，这样在导出的时候通过这个记录就可以恢复数据顺序。

1. 当前变量的类型标示符，以及具体的值。

利用宏REGISTER\_PROPERTY来生成了多个模板函数，以此来使类型匹配到一个具体的枚举值（标示符），并利用serializer将属性的值读入。

1. 当前变量或者依赖的信息的总尺寸大小。将这次对该变量或者依赖的导出数据尺寸记录，以方便后期验证和查错，以及发现错误后可以通过强制修正读取位置进行纠错。
2. 是否可以编辑，颜色，显示名，类别名，提示，解析参数，这几类信息。
3. 对于枚举类型：
4. 一个bool标示符，见上。
5. 枚举标示符。
6. 当前的枚举值。
7. 枚举值类型字符串。

用于后期在editor中与枚举信息进行匹配。

1. selfDefineString数组的个数， 用于后面遍历读入自定义字符串。
2. 自定义字符串，用于editor上自定义显示枚举值。
3. 当前变量或者依赖信息的总尺寸大小，见上。
4. 是否可以编辑，颜色，显示名，类别名，提示，解析参数，这几类信息。
5. 对于列表类型：

和基本类型一致，唯一不同的是，用的宏是REGISTER\_PROPERT\_TEMPLATE1，在读入变量类型的时候，还必须要把列表所能包含的子节点类型一起导入。如std::vector<size\_t> m\_listMember，就必须要导入列表类型和无符号整型类型两个信息。

1. 对于指针类型：

和基本类型一致，唯一不同的是，在读入变量类型的时候，会验证指针所指向的类型是否是组件子类，如果是则将其guid一并导入。

1. 对于依赖以及依赖列表：
2. 一个bool标示符，见上。
3. 一个bool标示符，用于明确是依赖还是依赖列表。因为CDependencyDescription可以同时表示依赖和依赖列表。
4. DependencyType，表明是强依赖还是弱依赖。
5. 依赖的组件GUID,用于在Editor中进行类型检查。
6. 显示名
7. 变量名

组件映射原理：

通过三个宏实现组件映射，分别是：

START\_REGISTER\_COMPONENT

REGISTER\_COMPONENT(ComponentType, DisplayName, CatalogName)

END\_REGISTER\_COMPONENT

这三个宏实质上是一个全局结构体，全局变量可以拿到项目代码启动之前的时机，于是我们可以在构造函数里做一些基本的导入导出工作。

START\_REGISTER\_COMPONENT

在宿主工程导出数据时，主要负责导出一些公用基本信息，比如版本号，文件头，包含的组件个数等。

在宿主工程导入时，负责调用launch函数。

REGISTER\_COMPONENT(ComponentType, DisplayName, CatalogName)

在宿主工程导出数据时，需要导出以下信息：

* + - 1. 一个bool标示符，用于表示该类是否是一个抽象类。

之所以要把抽象类信息也导入到editor主要是考虑到这个功能：如果组件A有一个对组件B的依赖，意味着，这个地方可以与B和B的子类进行连线建立依赖关系，除此以外的组件是无法建立依赖关系的。为了在EDITOR中分别哪些组件和依赖可以建立关系，我们必须将继承体系数据化。在这个继承体系数据化的过程中，不可避免的需要将中间的抽象类也进行数据建模。

在操作上，Editor读入组件信息时，如果发现是抽象类，那么只需要获取跟继承信息相关的数据：GUID。

* + - 1. 整个组件导出数据的大小，用于后期的查错纠错。
      2. 代表该组件的唯一32位标示符。
      3. 代表该组件父类的唯一32位标示符，用于Editor中的继承关系查询。
      4. 组件类名字符串，我们称之为ClassNameString
      5. 组件在editor中的显示名和目录名
      6. 包含的属性个数和依赖个数，用于editor做循环解析。
      7. 组件具体的属性和依赖信息，见上。

在第七步中，为了自动智能的统计一个组件中所有包含的属性和依赖数量，用了一个技巧，首先在serializer的userpointer是一个叫extradata的东西。因为serializer被传入组件的构造函数中，所以在构造函数里对每一个属性和依赖进行数据导出的时候，可以直接修改这个extra data中的counter。

在宿主工程导入时，直接根据ComponentType生成一个组件模板，以供之后加载文件的克隆操作。

END\_REGISTER\_COMPONENT

在宿主工程导出数据时，导出之前的组件总数量，并保存为文件

在宿主工程导入时，直接调用CComponentManager::Import函数

# 组件数据交互流程

数据交互大概分以下几个阶段：

1. 从宿主工程中利用宏将基本的结构信息导出。
2. Editor读入信息，并在这个基础上将工程文件加载出来。
3. 在editor上对工程文件进行修改。
4. 将工程数据导出。
5. 宿主工程读入工程数据，开始工作。

第一步之前已经讨论过了。

第二步，通过CComponentManager::DeserializerTemplateData和CComponentManager::OpenFile来完成。

对于从宿主工程导出的基本信息，我们称之为Template。之后加载文件时，根据文件创建各种不同的组件，我们称之为Instance。Instance是通过Tempalte的clone方法得来的。

Tempalte加载方式：

1. 验证文件头，版本。
2. 获取组件数量，开始遍历导入。
3. 获取当前组件的属性数量和依赖数量，开始遍历导入。
4. 根据数据中的datasize记录进行验证，纠错。

Instance加载方式：

Instance存在于具体的组件文件中（以XML保存）。基本的加载方法既通过调用CComponentManager::OpenFile读取XML的内容。首先通过类型匹配找到对应的Template，再调用其Clone方法获得一个新的Instance实例，再继续根据XML内容初始化属性和依赖数据。

第三步，用户在Eidtor上可以执行操作。

一些基本操作：创建，删除，修改属性和依赖，查询。

第四步，通过调用CComponentManager::Export导出数据。

属性，依赖，组件都必须实现Serialize接口，用于导出。在ptr和list属性上的复杂度比较高。

第五步，通过调用CComponentManager::Import导入数据。

分成三步：读入数据，解决依赖，呼叫初始化请求。因为每个组件都支持带serializer参数的构造函数，所以读入数据非常简单。

# 工作方式

1. 用注册自己的属性类。

REGISTER\_PROPERTY(bool, CBoolPropertyDescription)

REGISTER\_PROPERTY(float, CFloatPropertyDescription)

REGISTER\_PROPERTY(double, CFloatPropertyDescription)

REGISTER\_PROPERTY(int, CIntPropertyDescription)

REGISTER\_PROPERTY(std::string, CStringPropertyDescription)

REGISTER\_PROPERTY(size\_t, CUIntPropertyDescription)

REGISTER\_PROPERTY(CReflectBase\*, CPtrPropertyDescription)

REGISTER\_PROPERTY\_TEMPLATE1(std::vector, CListPropertyDescription)

REGISTER\_PROPERTY\_ENUM(CEnumPropertyDescription)

为了保证库的扩展性，所以这里将数据类型和我们自定义的propertybase子类进行关联。此处支持任意数据类型。  
尚未解决的问题：如何让带一个模板参数的类型和枚举类型共同使用REGISTER\_PROPERTY宏。

2. 注册组件

属性必须是依附在组件上存在，所以我们必须注册自己的组件，通过选中组件，可以查看组件包含的属性。

START\_REGISTER\_COMPONET

REGISTER\_COMOPNENT(CTestComponent, \_T("测试组件用例"), \_T("Test\\Components"))

REGISTER\_COMOPNENT(CTestComponent2, \_T("测试组件用例2"), \_T("Test\\Components"))

REGISTER\_COMOPNENT(CTestComponent3, \_T("测试组件用例3"), \_T("Test\\Components"))

END\_REGISTER\_COMPONET

以上是注册了三个我们自己写的组件用于显示在组件编辑器。

注释:

关于m\_value的管理。

CPropertyDescriptionBase中保存了一个组件成员变量的指针->XML上的值

CPropertyDescriptionBase的子类有一个m\_value->保存的反序列化后的值

CPropertyDescriptionBase的wxPGProperty中包含一个defaultvalue->保存的反序列化后的值

CPropertyDescriptionBase的wxPGProperty中包含一个value->当前的值（未保存）

根据wxPGProperty产生的wxPGProperty和wxPGProperty完全一致。

流程：

1. 宏分析语法得到默认值，如果宏没有设置默认值，则与当时变量的值一致。将值设置给m\_value， variableAddr
2. 反序列化的时候一般会首次创建m\_pPGProperty.此时将得到的值设定到m\_value， varialbeAddr，m\_pPGProperty的defaultvalue。
3. 然后是XML path阶段，这里得到的值，直接设置到variableAddr和m\_pPGProperty的value
4. UI修改阶段设置到m\_pPGProperty的value

其实对于CPropertyDescriptionBase来说m\_value是默认值 variableAddr是当前值

对于wxPGProperty来说defaultvalue是默认值value是当前值

判断是否未保存（variableAddr和wxPGProperty）

判断是否和默认值一样（m\_value和wxPGPRoperty.defaultvalue）



3. 反射组件里的属性

DECLARE\_PROPERTY(property, editable, color, displayName, catalog, tip, parameter)

DECLARE\_PROPERTY\_ENUM(enumType, enumVariable, editable, color, displayName, catalog, tip, count, enumstr)

目前只针对Enum使用了特殊的宏，但以后重构可以统一接口。

# 工作原理

1. 通过宏收集程序员对属性反射的需求数据
2. 组件被注册到一个全局变量中，所以在程序启动的时候，所有注册过的组件都会被创建一次，这样也就有机会将构造函数中的数据处理并保存到一个二进制文件中。
3. 当组件都被创建完毕之后，二进制文件创建完成，此时组件编辑器启动，并解析二进制文件，将组件列表构建完成。
4. 当选中组件列表中的组件时，编辑器会查询该组件所包含的属性信息，并将数据传送给property进行处理。这些处理包括数据传输上的和UI上的创建。
5. 在UI上的操作会保存到另外一个数据文件中。
6. 程序会根据数据文件是否更新来自动重新加载数据。