C++反射实现动态对象创建

这不属于图形学，也不是什么很有技术含量的东西。写这个主要是因为刚好有人问到。另外，我一直不大喜欢语法糖相关的东东，这里尽量降低语法糖依赖。

首先，这个东东，主要适用于什么场景？

我来先假设一个应用场景：做游戏的时候，必然是需要一个场景编辑器/地图编辑器这样一个东东的。编辑器主要给美术和策划用，用于编辑一个漂亮的场景，编辑一些占位，遮挡，编辑一些怪物出现，怪物AI等等。编辑的东西，一般会保存成一个文本文档，例如我以前做的是用的xml保存，可以手动编辑。游戏发包的时候，再把XML编成二进制文件，有需要的话再加密之类的。

脚本里面，一般记录的是对象的名字。例如，有一百种怪物，假设名字分别为Monster1、Monster2……Monster100。那么，程序解释这个脚本的时候，就得到了这个怪物的名字（这里，仅仅以怪物为例，其他的也可以类似的处理）。程序得到这个名字的时候，如何创建这个对象呢？传统的，最简单的，谁都能想到的方法：

Void CreateObject(std::string ObjName)

{

If(ObjName == “Monster1”)

{

New Monster1;

}

Else if(ObjName == “Monster2”)

{

New Monster2;

}

……

}

我相信绝大部分新手，都是从这样的代码开始的，如何让这个代码更优雅一点呢？这个也简单，怪物用一个枚举跟名字对应，用一个map保存对应关系，大概这样：

Enum MonsterID

{

Monster\_1,

Monster\_2,

Monster\_3,

……

Monster\_Max,

}

Std::map<std::string, MonsterID> mMonsterMap;

Void CreateObject(std::string ObjName)

{

// 注意，这里必须先find一下，不能直接下标找

If(mMonsterMap.find(ObjName) == mMonsterMap.end())

{

Return;

}

MonsterID ID = mMonsterMap[ObjName];

Switch(ID)

{

Case Monster\_1:

New Monster1;

Break;

……

}

}

以上代码，确实更加优雅了一些，比起前面一种。但是其实做法是换汤不换药。这类代码有什么问题呢？

1. 如果对象类型太多，需要写太多的分支，例如可能有一百多甚至更多的分支。这类代码其实也不是没见过，即使在UE4这类管理非常严格的代码，也能看到一百几十个分支的写法，这里不讨论对错，只提一下我觉得这是不优雅的地方。
2. 同上，如果分支太多，还会导致包含的头文件太多。因为你new对象的时候，是需要包含头文件的。包含一百几十个头文件，绝对不是什么很好的主意。
3. 如果作为一个框架，这类基本属于核心代码了。核心代码的话，按道理是不该可以轻易改动的，甚至很多小组里，核心代码一般人不该有修改的权限。但是，这里每增加或者删除一种类型，都必须修改这部分核心代码，这其实不大科学。所以这代码其实耦合性也挺高的。

反射机制是如何实现这个功能，避免上面问题的呢？首先，其实有很多个方案。比较常见的，基本都是用模板的。这里，我提供一种我自己比较常用的方案。方案无关对错，一个是个人喜好，二来跟实际需求有很大关系。

我自己比较常用的方案大爱是这样的：

1. 首先，定义一个基类，一切方法全部抽线成虚函数，这样避免了头文件包含的问题。大概这样：

class DynamicObjectBase

{

public:

DynamicObjectBase() {}

virtual ~DynamicObjectBase(){ }

virtual void Initialise(){}

virtual void Release(){}

protected:

private:

};

1. 定义一个工厂类，用于创建对象。由于方法全部都是抽象的，所以不需要包含头文件，只返回Base对象即可，方法里面核心函数有CreateObject和Register，Register的目的，是每增加一个对象的时候，都注册一下。这样，增加对象完全不需要改动代码。大概这样：

class DynamicObjectFactory

{

typedef DynamicObjectBase\*(\_stdcall\* DynamicCreateObject)();

public:

static DynamicObjectFactory\* Instance();

DynamicObjectBase\* CreateObject(*std*::*string* ObjectName);

void Register(*std*::*string* ObjectName, DynamicCreateObject Func);

void UnRegister(*std*::*string* ObjectName);

protected:

DynamicObjectFactory();

~DynamicObjectFactory();

*std*::*map*<*std*::*string*, DynamicCreateObject> mCreateDynamicObjectFuncMap;

};

1. DynamicObjectBase.CPP代码如下，主要是上部分函数的实现：

#include "DynamicObjectBase.h"

DynamicObjectFactory::DynamicObjectFactory()

{

}

DynamicObjectFactory::~DynamicObjectFactory()

{

}

DynamicObjectFactory\* DynamicObjectFactory::Instance()

{

static DynamicObjectFactory Ins;

return &Ins;

}

DynamicObjectBase\* DynamicObjectFactory::CreateObject(*std*::*string* ObjectName)

{

if (mCreateDynamicObjectFuncMap.*find*(ObjectName) != mCreateDynamicObjectFuncMap.*end*())

{

DynamicCreateObject Func = mCreateDynamicObjectFuncMap[ObjectName];

return Func();

}

return nullptr;

}

void DynamicObjectFactory::Register(*std*::*string* ObjectName, DynamicCreateObject Func)

{

if (mCreateDynamicObjectFuncMap.*find*(ObjectName) == mCreateDynamicObjectFuncMap.*end*())

{

mCreateDynamicObjectFuncMap[ObjectName] = Func;

}

}

void DynamicObjectFactory::UnRegister(*std*::*string* ObjectName)

{

if (mCreateDynamicObjectFuncMap.*find*(ObjectName) != mCreateDynamicObjectFuncMap.*end*())

{

mCreateDynamicObjectFuncMap.*erase*(ObjectName);

}

}

1. 以上代码的核心是：每添加一个类型，只需要把这个类型的名字和创建这个类型的函数（其实就是new对象的函数），Register一下即可。这里需要用到函数指针。我这里用的是传统函数的指针，老代码了。新人或者新架构都喜欢用std::function<>，这个我觉得都无所谓。

这里肯定会有人问，你不是需要register一下吗，你不还是得有地方执行register代码吗？你有一百几十中类型，不还是要写一百几十次register吗，你增加修改类型，不还是要修改这部分register的代码吗？

这里，非常巧妙的利用了static变量。这个变量的核心是：初始化的时候，是在main函数之前，而且是无序的，不可控的被执行，不需要你主动去执行。所以，register其实是放在一个static对象的构造函数里面来自动执行的。

1. 这里是register部分的代码，我写成了宏定义。这是常规化操作。一般的大型项目，都有非常多的宏定义。典型的例如UE4，里面有无数的宏定义嵌套，能看到你头昏脑胀，怀疑人生。我没那么叼，就写一些中规中矩的。大概是这样子的：

#define REGISTER\_DYNAMICOBJECT(*name*) \

DynamicObjectBase\* \_stdcall Create\_##name() \

{ \

return new *name*; \

} \

struct Register##name \

{ \

Register##name() \

{ \

DynamicObjectFactory::Instance()->Register(#name, Create\_##name); \

} \

}Register\_##name;

1. 仔细看看，上面很简单，首先是定义了一个new对象的函数，这个函数啥都不干，就new一个对象。下面这个结构体，就是一个非常简单的结构体，核心其实就是定义一个static对象，然后这个对象被初始化的时候，会自动执行构造函数，然后构造函数里面，会调用到register。这里有一个地方要注意的。因为这么搞，会有一百几十个static对象。而这类对象初始化的时候，是没有顺序的。那么如何确保factory对象在所有这些对象构造之前呢？这里用的一个static类函数。这个函数被调用的时候，factory的static对象必然已经被初始化了，所以能保证顺序。
2. 下面是测试代码，代码的实现，只需要放在cpp文件里即可，完全不需要有h文件。注意，每一个类都需要使用一下RegisterObject这个宏定义，就上面这个。保证每个类都被注册了。下面是TestDynamicObject.cpp的代码，大概是这样的：

#include "DynamicObjectBase.h"

#include "Register.h"

class TestDynamicObject : public DynamicObjectBase

{

public:

virtual void Initialise();

virtual void Release();

protected:

private:

};

REGISTER\_DYNAMICOBJECT(TestDynamicObject)

void TestDynamicObject::Initialise()

{

*printf*("TestDynamicObject Initialise\n");

}

void TestDynamicObject::Release()

{

*printf*("TestDynamicObject Release\n");

delete this;

}

class TestDynamicObject1 : public DynamicObjectBase

{

public:

virtual void Initialise();

virtual void Release();

protected:

private:

};

REGISTER\_DYNAMICOBJECT(TestDynamicObject1)

void TestDynamicObject1::Initialise()

{

*printf*("TestDynamicObject1 Initialise\n");

}

void TestDynamicObject1::Release()

{

*printf*("TestDynamicObject1 Release\n");

delete this;

}

class TestDynamicObject2 : public DynamicObjectBase

{

public:

virtual void Initialise();

virtual void Release();

protected:

private:

};

REGISTER\_DYNAMICOBJECT(TestDynamicObject2)

void TestDynamicObject2::Initialise()

{

*printf*("TestDynamicObject2 Initialise\n");

}

void TestDynamicObject2::Release()

{

*printf*("TestDynamicObject2 Release\n");

delete this;

}

1. 下面是main函数里的测试代码，大概这样就好：

int main()

{

DynamicObjectBase\* Obj1 = DynamicObjectFactory::Instance()->CreateObject("TestDynamicObject");

Obj1->Initialise();

Obj1->Release();

Obj1 = DynamicObjectFactory::Instance()->CreateObject("TestDynamicObject1");

Obj1->Initialise();

Obj1->Release();

Obj1 = DynamicObjectFactory::Instance()->CreateObject("TestDynamicObject2");

Obj1->Initialise();

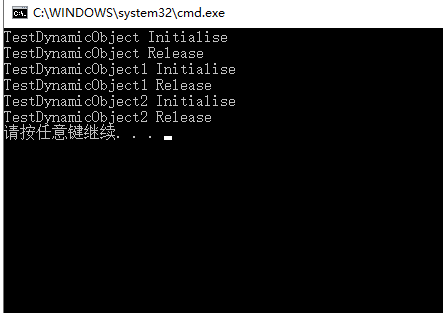
Obj1->Release();

*system*("pause");

return 0;

}

1. 执行效果大概是这样的：



以上就是所有的代码了。

这么设计的好处：

1. 低耦合。每增加或者删除一个类型，只需要增加一个cpp或者删掉一个cpp即可，其他任何代码无需改动。
2. 最大限度的减少头文件。要知道，项目越大，头文件越多，互相之间include越多，很容易出问题，容易出现那种互相include然后导致编译编不过之类乱七八糟的问题。

这么设计也有一些不足之处。例如，没有头文件会降低耦合，但是也会降低自由度。例如，你打算给monster1怪物新增加一个独有的接口，就很麻烦了。这种情况，你需要先在Base类里面增加一个虚函数，然后再在monster1里面把这个函数实现了。所以实际上，根据自己的实际需求设计自己的框架才是最合理的，没有说一套框架包打天下这种好事。

一家之言，讲得不对的，欢迎各位大佬来喷。