Factory Methods

工厂方法

eryar@163.com

摘要 ***Abstract***：本文主要是对《***API Design for C++***》中 ***Factory Methods*** 章节的翻译，若有不当之处，欢迎指正。
关键字 ***Key Words***：***C++***、***Factory Pattern***、

一、概述 ***Overview***

工厂方法是创建型模式，允许在不指定需要创建对象类型的情况下创建出对象。本质上来说，工厂方法就是一个通用的构造函数。***C++***中的构造函数有以下几种限制：

- 无返回值（***No return result***）。在构造函数中不能返回一个值。这就意味着：例如当构造失败时不能返回一个 ***NULL*** 作为初始化失败的信号。
- 命名有约束（***Constrained naming***）。构造函数还是很好识别的，因为它的命名必须与类名一样。
- 静态绑定创建的（***Statically bound creation***）。当创建一个对象时，必须指定一个在编译时就能确定的类名。如：***Foo *f = new Foo()***，***Foo*** 就是编译器必须知道的类名。***C++***的构造函数没有运行时的动态绑定功能（***dynamic binding at run time***）。
- 无虚构造函数（***No virtual constructors***）。在 ***C++***中不能声明虚的构造函数，必须指定在编译时能确定的类型。编译器据此为指定的类型分配内存，然后调用基类的默认构造函数，再调用指定类的构造函数。这就是不能定义构造函数为虚函数的原因。

相反地，工厂方法（***factory methods***）突破了以上所有的限制。工厂方法的基本功能就是一个可以返回一个类的实例的简单函数。但是，它通常与继承组合使用，派生的类可以重载工厂方法以返回派生类的实例。使用抽象基类（***Abstract Base Classes***）来实现工厂很常见，也很有用。

二、抽象基类 ***Abstract Base Classes***

抽象基类就是包含一个或多个纯虚函数（***pure virtual methods***）的类，这样的类不是具体类且不能用 ***new*** 来实例化。相反地，它是作为其它派生类的基类，由派生类来具体实现那些纯虚函数。例如：

```
#ifndef RENDERER_H
#define RENDERER_H

#include <string>

///
/// An abstract interface for a 3D renderer.
///
class IRenderer
```

```
{
public:
    virtual ~IRenderer() {}
    virtual bool LoadScene(const std::string &filename) = 0;
    virtual void SetViewportSize(int w, int h) = 0;
    virtual void SetCameraPos(double x, double y, double z) = 0;
    virtual void SetLookAt(double x, double y, double z) = 0;
    virtual void Render() = 0;
};

#endif
```

上述代码定义了一个抽象基类，描述了一个相当简单的 3***D*** 图形渲染器（***renderer***）。函数的后缀“＝0”声明这个函数是纯虚函数，表示这个函数必须由其派生类来具体实现。

抽象基类是描述了多个类共有的行为的抽象单元，它约定了所有具体派生类必须遵守的合同。在 ***Java*** 中，抽象基类也叫接口（***interface***），只是 ***Java*** 的接口只能是公用的方法（***public method***），静态变量，并且不能定义构造函数。将类名 ***IRenderer*** 带上“***I***”就是为了表明这个类是接口类（***interface class***）。

当然，抽象基类中并不是所有的方法都必须是纯虚函数，也可以实现一些函数。

当任意一个类有一个或多个虚函数时，通常会把抽象基类的析构函数声明为虚函数。如下代码说明了这样做的重要性：

```
class IRenderer
{
    // no virtual destructor declared
    virtual void Render() = 0;
};

class RayTracer : public IRenderer
{
    RayTracer();
    ~RayTracer();
    void Render(); // provide implementation for ABC method
};

int main(int, char **)
{
    IRenderer *r = new RayTracer();
    // delete calls IRenderer::~IRenderer, not RayTracer::~RayTracer
    delete r;
}
```

三、简单工厂模式 ***Simple Factory Example***

在复习了抽象基类后，让我们在简单工厂方法中使用它。继续以 ***renderer.h*** 为例，声明创建工厂，创建的对象类型为 ***IRenderer***，代码如下所示：

```
#ifndef RENDERERFACTORY_H
```

```
#define RENDERERFACTORY_H

#include "renderer.h"
#include <string>

///
/// A factory object that creates instances of different
/// 3D renderers.
///
class RendererFactory
{
public:
    /// Create a new instance of a named 3D renderer.
    /// type can be one of "opengl", "directx", or "mesa"
    IRenderer *CreateRenderer(const std::string &type);
};

#endif
```

这里只声明了一个工厂方法：它只是一个普通的函数，返回值是对象的实例。注意到这个方法不能返回一个指定类型的 ***IRender*** 实例，因为抽象基类是不能被实例化的。但是它可以返回派生类的实例。当然，你可以使用字符串作为参数来指定需要创建对象的类型。

假设已经实现了派生自 ***IRender*** 的三个具体类： ***IRenderer::OpenGLRenderer***，***DirectXRenderer***、***MesaRenderer***。再假设你不想让使用 ***API*** 的用户知道可以创建哪些类型，他们必须完全隐藏在 ***API*** 后面。基于这些条件，可以实现工厂方法的程序如下：

```
// rendererfactory.cpp
#include "rendererfactory.h"
#include "openglrenderer.h"
#include "directxrenderer.h"
#include "mesarenderer.h"

IRenderer *RendererFactory::CreateRenderer(const std::string &type)
{
    if (type == "opengl")
        return new OpenGLRenderer;

    if (type == "directx")
        return new DirectXRenderer;

    if (type == "mesa")
        return new MesaRenderer;

    return NULL;
}
```

这个工厂方法可以返回 ***IRenderer*** 的三个派生类之一的实例，取决于传入的参数字符串。

这就可以让用户决定在运行时而不是在编译时创建哪个派生类，这与普通的构造函数要求一致。这样做是有很多好处的，因为它可以根据用户输入或根据运行时读入的配置文件内容来创建不同的对象。

另外，注意到实现具体派生类的头文件只在 ***rendererfactory.cpp*** 中被包含。它们不出现在 ***rendererfactory.h*** 这个公开的头文件中。实际上，这些头文件是私有的头文件，且不需要与 ***API*** 一起发布的。这样用户就看不到不同的渲染器的私有细节，也看不到具体可以创建哪些不同的渲染器。用户只需要通过字符串变量来指定种要创建的渲染器（若你愿意，也可用一个枚举来区分类型）。

此例演示了一个完全可接受的工厂方法。但是，其潜在的缺点就是包含了对可用的各派生类的硬编码。若系统需要添加一个新的渲染器，你必须再编辑 ***rendererfactory.cpp***。这并不会让人很烦，重要的是不会影响你提供的公用的 ***API***。但是，他的确不能在运行时添加支持的新的派生类。再专业点，这意味着你的用户不能向系统中添加新的渲染器。通过扩展的对象工厂来解决这些问题。

四、扩展工厂模式 *Extensible Factory Example*

为了让工派生类从工厂方法中解耦，且允许在运行时添加新的派生类，可以去维护包含类型及与类型创建关联的函数的映射（***map***）来更新一下工厂类。可以通过添加几个新的函数用来注册与注销新的派生类。在运行时能注册新的类允许这种类型的工厂方法模式可用于创建可扩展的接口。

还有个需要注意的事是工厂对象必须保存状态，即最好只有一个工厂对象。这也是工厂对象通常是单件的（***singletons***）。为了程序的简单明了，这里使用静态变量为例。将所有要点都考虑进来，新的工厂对象代码如下所示：

```
#ifndef RENDERERFACTORY_H
#define RENDERERFACTORY_H

#include "renderer.h"
#include <string>
#include <map>
///
/// A factory object that creates instances of different
/// 3D renderers. New renderers can be dynamically added
/// and removed from the factory object.
///
class RendererFactory
{
public:
    /// The type for the callback that creates an IRenderer instance
    typedef IRenderer *(*CreateCallback)();

    /// Add a new 3D renderer to the system
    static void RegisterRenderer(const std::string &type,
                                 CreateCallback cb);
    /// Remove an existing 3D renderer from the system
    static void UnregisterRenderer(const std::string &type);
```

```
    /// Create an instance of a named 3D renderer
    static IRenderer *CreateRenderer(const std::string &type);

private:
    typedef std::map<std::string, CreateCallback> CallbackMap;
    static CallbackMap mRenderers;
};

#endif
```

为了程序的完整性，将其.cpp 文件中的代码示例如下：

```
#include "rendererfactory.h"
#include <iostream>

// instantiate the static variable in RendererFactory
RendererFactory::CallbackMap RendererFactory::mRenderers;

void RendererFactory::RegisterRenderer(const std::string &type,
                                       CreateCallback cb)
{
    mRenderers[type] = cb;
}

void RendererFactory::UnregisterRenderer(const std::string &type)
{
    mRenderers.erase(type);
}

IRenderer *RendererFactory::CreateRenderer(const std::string &type)
{
    CallbackMap::iterator it = mRenderers.find(type);
    if (it != mRenderers.end())
    {
        // call the creation callback to construct this derived type
        return (it->second)();
    }

    return NULL;
}
```

使用工厂对象创建派生类的方法如下所示：

```
#include "rendererfactory.h"
#include <iostream>

using std::cout;
```

```
using std::endl;

/// An OpenGL-based 3D renderer
class OpenGLRenderer : public IRenderer
{
public:
    ~OpenGLRenderer() {}
    bool LoadScene(const std::string &filename) { return true; }
    void SetViewportSize(int w, int h) {}
    void SetCameraPos(double x, double y, double z) {}
    void SetLookAt(double x, double y, double z) {}
    void Render() { cout << "OpenGL Render" << endl; }
    static IRenderer *Create() { return new OpenGLRenderer; }
};

/// A DirectX-based 3D renderer
class DirectXRenderer : public IRenderer
{
public:
    bool LoadScene(const std::string &filename) { return true; }
    void SetViewportSize(int w, int h) {}
    void SetCameraPos(double x, double y, double z) {}
    void SetLookAt(double x, double y, double z) {}
    void Render() { cout << "DirectX Render" << endl; }
    static IRenderer *Create() { return new DirectXRenderer; }
};

/// A Mesa-based software 3D renderer
class MesaRenderer : public IRenderer
{
public:
    bool LoadScene(const std::string &filename) { return true; }
    void SetViewportSize(int w, int h) {}
    void SetCameraPos(double x, double y, double z) {}
    void SetLookAt(double x, double y, double z) {}
    void Render() { cout << "Mesa Render" << endl; }
    static IRenderer *Create() { return new MesaRenderer; }
};


int main(int, char **)
{
    // register the various 3D renderers with the factory object
    RendererFactory::RegisterRenderer("opengl", OpenGLRenderer::Create);
```

```
    RendererFactory::RegisterRenderer("directx", DirectXRenderer::Create);
    RendererFactory::RegisterRenderer("mesa", MesaRenderer::Create);

    // create an OpenGL renderer
    IRenderer *ogl = RendererFactory::CreateRenderer("opengl");
    ogl->Render();
    delete ogl;

    // create a Mesa software renderer
    IRenderer *mesa = RendererFactory::CreateRenderer("mesa");
    mesa->Render();
    delete mesa;

    // unregister the Mesa renderer
    RendererFactory::UnregisterRenderer("mesa");
    mesa = RendererFactory::CreateRenderer("mesa");
    if (! mesa)
    {
        cout << "Mesa renderer unregistered" << endl;
    }

    return 0;
}
```

你的 ***API*** 的用户可以在系统中注册与注销一个新的渲染器。编译器将会确保用户定义的新的渲染器必须实现抽象基类 ***IRenderer*** 的所有抽象接口，即新的渲染器类必须实现抽象基类 ***IRenderer*** 所有的纯虚函数。如下代码演示了用户如何自定义新的渲染器，在工厂对象中注册，并叫工厂对象为之创建一个实例：

这里需要注意的一点是我向类 ***UserRenderer*** 中添加了一个 ***Create***()函数，这是因为工厂对象的注册方法需要返回一个对象的回调函数。这个回调函数不一定必须是抽象基类 ***IRenderer*** 的一部分，它可以是一个自由的函数。但是向抽象基类 ***IRenderer*** 中添加这个函数是一个好习惯，这样就确保了所有相关功能的一致性。实际上，为了强调这种约定，可以将 ***Create*** 作为抽象基类 ***IRenderer*** 的一个纯虚函数。

五、结论 ***Conclusion***

Finally, I note that in the extensible factory example given here, a renderer callback has to be visible to the RegisterRenderer() function at run time. However, this doesn't mean that you have to expose the built-in renderers of your API. These can still be hidden either by registering them within your API initialization routine or by using a hybrid of the simple factory and the extensible
factory, whereby the factory method first checks the type string against a few built-in names. If none of those match, it then checks for any names that have been registered by the user. This hybrid
approach has the potentially desirable behavior that users cannot override your built-in classes.