1 什么是对象工厂

面向对象程序通过**继承**和**虚函数**获得了强大的抽象性和多态能力。我们在拥有指向多态对象的基类指针或者引用时，我们可以利用**虚函数**的特性，在调用成员函数时完全确定对象的动态类型从而调用正确的成员函数。例如：

```
class Base
{
public:
    virtual void fa() = 0;
};

class Derived: public Base
{
public:
    void fa(){...}
    void fb(){...}
};

Base *p = new Derived();
```

此时，我们调用 `p->fa()` 所真正调用的是继承类 `Derived` 中的 `fa()` 函数，而调用 `p->fb()` 则是未定义的行为。这说明，在基类接口的限制下，我们能够在执行期动态的调用多态类型的成员函数，这件事情在已经有了一个指向某一个多态对象的基类指针的情况下是显然的。但我们现在要思考一个问题，程序总要在某个地方去创建这么一个多态对象并把它绑定到基类指针上去，而通常多态对象是通过 `new` 来生成的，例如前面的 `Base *p = new Derived();` 但这里出现了 `Derived` 这一确定的类型名称，也就是说，我们必须在编译期就把它确定下来，把它真正的写在我们的程序里。所以，仅仅是利用虚函数的特性我们还是只能在调用成员函数的时候更加的方便，但对于**动态的去产生多态对象**是没有帮助的。

对象工厂就是为了解决这一问题，它的目标就是把所要产生对象的类型真正推迟到执行期再确定。其最基本的功能就是：**在不需要对程序进行任何的修改和重新编译的情况下，根据用户在执行期的不同输入来构造用户想要的不同**

多态对象，并返回其基类指针。工厂模式的作用就是隔离类对象的使用者和具体类型之间的耦合关系。

2 对象工厂的实现

假设我们正在编写一个绘图程序，我们定义了如下的基类：

```
class Shape
{
public:
    virtual void draw() const = 0;
    virtual void rotate(double angle) = 0;
};
```

我们还有一些继承类：

```
class Circle : public Shape
{
public:
    void draw(){...}
    void rotate(double angle){...}
};

class Line : public Shape
{
public:
    void draw(){...}
    void rotate(double angle){...}
};
...
```

对于每个继承类来说，我们都可以定义类似于下面这样的函数：

```
std::unique_ptr<Shape> createCircle()
{
    return std::make_unique<Circle>();
```

```
}
```

这个对象生成函数负责生成一个指向 Circle 对象的智能指针。有了对象生成函数之后，最简单粗暴的方法就是写一个函数，它接受一个身份识别 (ID)，这个 ID 标识着我想要生成哪种多态对象，在函数内部通过 switch 等类似的语句来选则调用哪个对象生成函数。这种方式可以实现我们想要的效果，但是最大的一个缺点就是：如果我又从 Shape 这个基类继承出了一个新的类型，比如 rectangle，那我们就必须找到这个函数在哪定义的，之后在 switch 里面加上一项，这种代码的可拓展性是很差的，我们没有办法在程序中随时随地的修改它。

下面我们来介绍对象工厂是怎么实现我们的目标的。我们所想要的其实就是从继承类的 **ID** 到**对象生成函数**间的一个一一对应关系，我们很自然的可以想到 **std::map** 这个工具。我们可以从一个空的 std::map 开始，每有一个新的继承类，我们就向其中添加一对键和值（即 ID 和对应的对象生成函数），这样我们就可以通过 ID 来得到对应的对象生成函数并用于生成对象。现在我们可以设计出如下的对象工厂：

```
class ShapeFactory
{
public:
    using CreateShapeCallback = std::unique_ptr<Shape> (*)();

private:
    using CallbackMap = std::map<std::string, CreateShapeCallback>;

public:
    //the only way to create a ShapeFactory object
    static ShapeFactory& createFactory()
    {
        static ShapeFactory object;
        return object;
    }

    //return true if registration was successful
    bool registerShape(std::string shapeId, CreateShapeCallback createFn)
```

```
    {
        return callbacks_.insert(CallbackMap::value_type(shapeId, createFn)).
            second;
    }

    //return true if the shapeId was registered before
    bool unregisterShape(std::string shapeId)
    {
        return callbacks_.erase(shapeId) == 1;
    }

    std::unique_ptr<Shape> createShape(std::string shapeId)
    {
        auto it = callbacks_.find(shapeId);
        if (it == callbacks_.end())
        {
            throw std::runtime_error("Unknown Shape ID. ");
        }
        return (it->second)();
    }

private:
    CallbackMap callbacks_;

    struct Object_Creater{
        Object_Creater()
        {
            ShapeFactory::createFactory();
        }
    };

    static Object_Creater obj_crt;


    ShapeFactory() = default;
    ShapeFactory(const ShapeFactory&) = default;
    ShapeFactory& operator=(const ShapeFactory&) = default;
    ~ShapeFactory() = default;
```

```
};

ShapeFactory::Object_Creater ShapeFactory::obj_crt;
```

这是一个**可伸缩对象工厂**的基本设计，每次添加一个新的继承类时，不必修改对象工厂的代码，只需要在程序的任意位置指定好 ID 和对象生成函数并调用 registerShape 注册一下就可以了。当你想要生成一个对象时，你只需要告诉工厂对应的 ID，调用 createShape 就会返回一个指向你想要的对象的基类指针。当你不想让工厂再生产某个多态对象时，只需要在程序的任意位置调用 unregisterShape 删除键值为对应 ID 的映射就可以了。

这个设计里面还加入了 singleton pattern，即全局范围内只能存在一个工厂对象，其他所有的工厂都只能是这个工厂的引用。这一设计是通过把工厂类的构造函数、拷贝构造函数、赋值构造函数都设为 private 的，并且定义一个 public 的 static 成员函数，也就是上例中的 createFactory()，它是能够获取一个工厂对象引用的唯一方式。可以看到这个函数内部定义了一个 static 的工厂对象，这就意味着这个函数只有在第一次调用时会产生这一对象，之后的调用都会直接返回这一对象，所以全局只会存在一个工厂对象。

下面的 Object_Creater 这个结构是用来帮助生成唯一的工厂对象的，可以看到工厂类内部有一个 static 成员变量 obj_crt，在它被定义的时候，就会自动的调用一次 createFactory()，即工厂类在定义完成的时候就已经生成了一个工厂对象，并且这个对象是全局唯一的。

3 泛型工厂

上一节的内容已经能针对 Shape 这个基类很好的实现我们的目标。但从它的实现我们可以看出，如果我有另一种基类和它的继承类，那我们就需要重新写一个对象工厂，但它们其实本质上是没什么区别的。这就是为什么需要泛型工厂。

在上一节里我们可以看到对象工厂具有下面的几个关键要素：

i. 具体产品：即生成的具体对象;

ii. 抽象产品：即返回的基类指针，不带有对象的具体类型;

iii. 产品 ID：用来标识产品的类型;

iv. 产品生产者：生产代表不同具体产品的抽象产品。

泛型工厂就是用来协调这些要素，利用**模板**来给出一个能够用于不同基类的通用对象工厂。在上面的四个要素中，我们实际上只需要确定三个，具体产品的类型是包含在具体的产品生产者中的，所以我们需要的就是一个具有三个模板参数的泛型工厂类。如果假设这里的产品生产者和上一节一样，都采用对象生成函数的话，那它的类型就是返回抽象产品类型指针的函数，也就是说第三个模板参数是可以给一个缺省值的。

我们还需要考虑一个问题，并不是每个基类的构造函数都是不需要参数的，那么考虑上一节中对象工厂的形式，我们没有办法对对象进行带参数构造。如果我们仍然想说我们的泛型工厂能够适用于所有基类，那就要解决这一问题，但不同的基类构造函数需要的参数个数又不一样。这里就需要一种手段叫作**可变长模板**，正如之后代码中 `creatProduct` 函数中展示的，这种手段可以把任意长度的参数完美转发给另一个函数，也就是我们的对象生成函数。下面我们来看泛型工厂的具体实现：

```
template <class AbstractProduct, typename IdType, typename ProductCreator = std
    ::unique_ptr<AbstractProduct> (*)()>
class GenericFactory
{
private:
    using CallbackMap = std::map<IdType, ProductCreator>;

public:
    static GenericFactory& createFactory()
    {
        static GenericFactory object;
        return object;
    }

    //return true if registration was successful
    bool registerProduct(IdType Id, ProductCreator createFn)
    {
```

```
        return callbacks_.insert(typename CallbackMap::value_type(Id, createFn))
            .second;
    }

    //return true if the Id was registered before
    bool unregisterProduct(IdType Id)
    {
        return callbacks_.erase(Id) == 1;
    }

    //support construction
    template<class ...TS>
    std::unique_ptr<AbstractProduct> createProduct(IdType Id, TS && ...args)
    {
        auto it = callbacks_.find(Id);
        if (it == callbacks_.end())
        {
            throw std::runtime_error("Unknown product ID. ");
        }
        return (it->second)(std::forward<TS>(args)...);
    }

private:
    CallbackMap callbacks_;

    struct Object_Creater{
        Object_Creater()
        {
            GenericFactory::createFactory();
        }
    };

    static Object_Creater obj_crt;

    GenericFactory() = default;
    GenericFactory(const GenericFactory&) = default;
    GenericFactory& operator=(const GenericFactory&) = default;
    ~GenericFactory() = default;
```

```
};

template<class AbstractProduct, typename IdType, typename ProductCreator>
typename GenericFactory<AbstractProduct, IdType, ProductCreator>::
    Object_Creater GenericFactory<AbstractProduct, IdType, ProductCreator>::
    obj_crt;
```

我们在使用泛型工厂的时候,只需要实例化为我们想要的对象工厂。比如说现在我们现在有一个基类 Shape_int,它的构造函数需要接受一个整数,那我们就把 AbstractProduct 实例化为 Shape_int,把 IdType 实例化为 std::string,把 ProductCreator 实例化为 std::unique_ptr<Shape_int> (*)(int)。之后在进行对象生成时，调用:

```
GenericFactory<Shape_int, std::string, std::unique_ptr<Shape_int> (*)(int)>
    shapeFactory;
...
std::unique_ptr<Shape_int> p = shapeFactory.createProduct("Circle"/*Shape name
    */, 1/*integer*/);
```

就可以实现我们想要的对象工厂。