第 5 章

创建型模式

从本章开始将系统地讨论设计模式的概念和原理，在了解传统的设计模式（如单例模式、工厂模式、观察者模式等）的基础上，将进一步探索如何使用C++11元编程（Meta Programming）技术实现这些设计模式。

通过比较传统的实现方式和元编程实现方式，将深入了解C++11元编程方式的实现细节及C++11元编程技术的独特优势。此外还将探讨C++11元编程如何使设计模式的实现更加简洁、高效和灵活，以及它如何为开发者提供更多的编程选择和优化空间。

创建型模式提供了一种创建对象的机制，将对象的创建和使用分离以更好地满足系统的变化和扩展性需求，提高代码的可维护性和可扩展性。创建型模式主要关注对象的创建过程，以及如何解耦对象的创建和使用。

创建型模式主要包括工厂模式（Simple Factory Pattern）、抽象工厂模式（Abstract Factory Pattern）、单例模式（Singleton Pattern）、原型模式（Prototype Pattern）和生成器模式（Builder Pattern）。

5.1 工厂模式及其实现

在工厂模式中通常定义一个抽象的工厂类，该类包含一个创建对象的方法。具体的子类工厂可以继承抽象工厂，并实现自己的对象创建逻辑。通过工厂类，客户端代码可以通过调用工厂方法来创建所需的对象，而无须了解具体对象的实现细节。

另外，工厂模式还可以帮助遵循开闭原则，即对扩展开放，但对修改关闭。通过将对象的创建逻辑封装在工厂类中，可以在不修改客户端代码的情况下添加新的对象类型。

5.1.1 工厂模式的传统结构

传统工厂模式首先定义一个抽象的产品类，并在子类中实现具体创建接口，然后定义一个静态的工厂方法实现。这种方法对于传参的类型、数量都不能做到灵活应对，并且需要延迟到业务中进行实现，这就不能在实际开发中将程序员的注意力集中在具体业务上，并且因为要根据具体业务实现静态工厂方法，所以一旦添加新产品就不得不修改工厂逻辑，在产品

类型较多时会造成工厂逻辑过于复杂，不利于系统的扩展和维护。

absProduct 定义工厂方法的接口，根据实际需要定义接口函数，如图 5-1 所示。要求每个接口对应一个特定的应用。cncrtPdtA、cncrtPdtB 是实际实现，通过针对 absProduct 中的接口进行实现，从而创建对象。

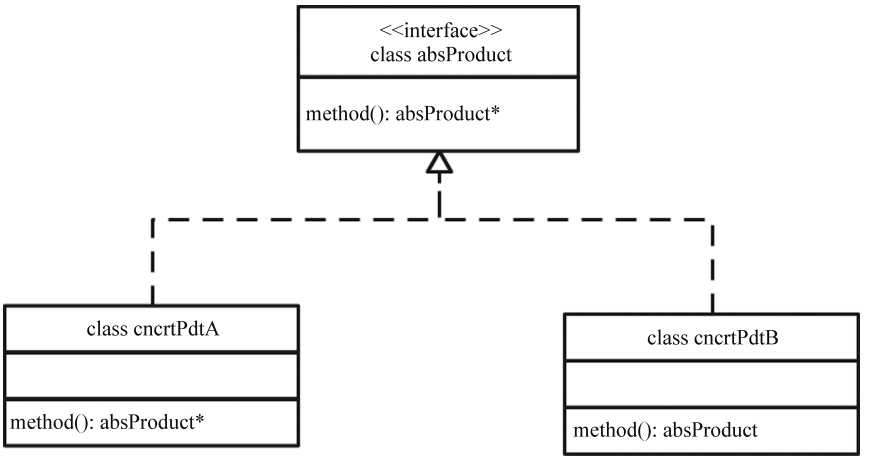

图 5-1　传统工厂模式 UML 简图

C++ 元编程可以将接口定义在编译期完成，可以在具体类型上增加通用性，并可以利用变长模板，将创建过程的参数变化在编译期明确。

传统工厂模式主要通过继承的方式实现接口。在每次用到工厂模式时都需要根据实际情况重新编写接口定义并重新实现具体类。这种方式造成了大量代码需要重写，在不同的项目中需要构造不同的符合工程需要的工厂模式代码，在浪费了大量的开发及测试时间的同时也容易引入新的问题。

5.1.2　使用C++11实现工厂模式的结构

利用C++11 的元编程技术，使用产品类的构造函数可以灵活地应对不同的参数数量和参数类型的产品类的构造。利用模板类可以很方便地实现数据类型的多样性，利用模板函数实现生成对象上的多样性。通过函数模板和类模板配合能够在编译期适应不同数据类型的工厂构造方式。

元编程技术实现的工厂模式放弃了类继承实现接口的方式，使用工厂模板避免了重复编写代码的情况，将工厂模式的实现和具体业务分离开来，并通过可变模板参数的工厂函数支持了不同数量的参数和不同类型的参数传入接口，增加了产品类的灵活性。应用时可以将具体产品继承自产品类，这样便可利用私有化的构造函数，从而避免了通过构造函数生成对象。C++11 工厂模式 UML 的简图如图 5-2 所示。

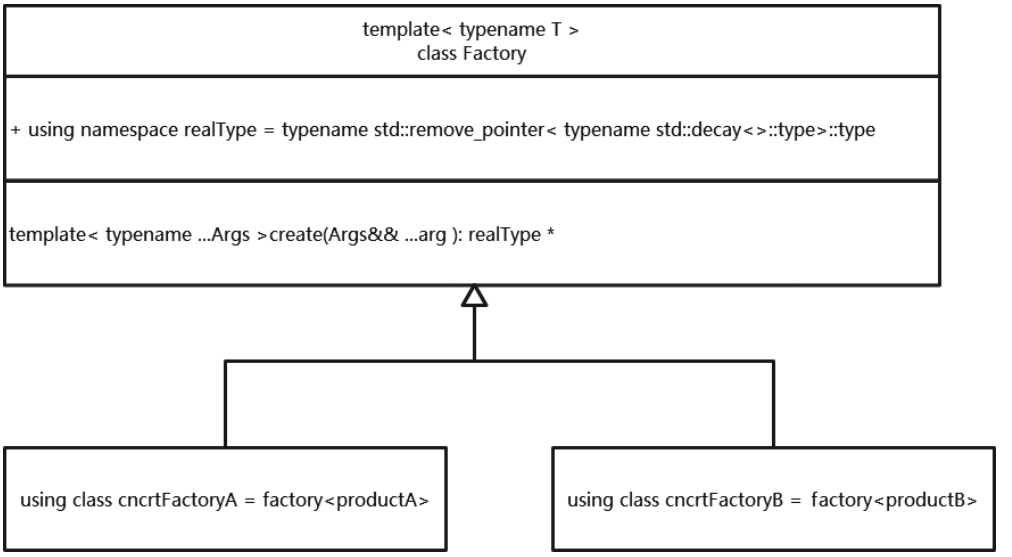

图 5-2 C++11 工厂模式 UML 的简图

5.1.3 工厂模式的实现和解析

在本书中工厂模式的实现采用了两种方式，一种是模板类；另一种是模板函数。

（1）模板类实现方式。通过模板参数实现对不同类型对象的创建。它的目的是提供一种通用的方式来创建对象，避免直接调用构造函数以增加代码的可扩展性和灵活性。

代码中的模板定义是这样的，代码如下：

```
//designM/factory.hpp

#include <type_traits>

template <typename type>
struct product
{
public:
    using realType = typename remove_pointer<
        typename decay< type >::type > :: type;
    static_assert(is_constructible<realType>::value &&
        is_class<realType>::value, "");
  ... ...
};
```

代码中使用元函数 std::is_constructible<T>::value 和 std::is_class<T>::value 进行类型检查，要求产品类必须支持构造函数。通过这种静态的类型检查可以在编译时发现不合要求的类，避免在运行时出现不支持的类型，从而导致错误。

using realType＝typename remove_pointer<typename decay<type>::type>::type;允许通过已有的对象推断目标类型。元函数 std::decay<T>::type 将 T 类型退化成最简单的形式，去掉 const 等修饰，然后使用元函数 std::remove_pointer<T>::type 移除指针类型。将实际的产品类型整理成原始的类类型。通过这种方式能够扩大类的包容性，在后面的各个模块的实现中都采用了这样的处理方式来扩大模块的适用范围，示例代码如下：

```
cncrtPdtA  * a = nullptr;
... ...
... ...
auto * b = product<decltype(a) >::create(...);
```

struct product{ //... };：这是结构体 Factory 的定义部分。在这里可以定义结构体的成员变量和成员函数等。

结构体 product 是一个根据模板参数进行定制化和实例化的工厂类模板。通过这些模板参数，可以在结构体 product 中根据不同的类型相应地进行操作和实现。这样可以减少代码的冗余，并增加代码的灵活性。

接下来定义了 create()方法，用来构造对象，在方法内部仍然使用了 new 操作符实现对象的构造操作，代码如下：

```
//designM/factory.hpp

template <typename... Args>
static realType * create(Args &&... args)
{
    realType * ret = nullptr;
    ret = new realType(std::forward<Args>(args)...);
    return ret;
}
```

使用 new 操作符来动态地分配内存，在分配内存时使用 std::forward<>()将传入的参数表 args 展开并完美地转发给实际要构造的类的构造函数。这样可以将参数以其原本的方式传递给构造函数。

当使用 new 操作符分配内存时可能会抛出 std::bad_alloc 异常，或者在调用构造函数时，构造函数内部可能会抛出模块自己实现的异常，但在上述代码中并没有针对这些情况进行处理，读者在使用时需要在自己的业务代码中处理这些异常情况。另外 create()函数返回的是裸指针，这容易存在内存安全问题，在实际使用时需要注意释放指针。为了能够更加安全地操作内存，在 create_shared()函数中通过返回的智能指针更好地管理内存和对象的生命周期，代码如下：

```
//designM/factory.hpp

template <typename... Args>
```

```
static std::shared_ptr<realType >create_shared(Args &&... args)
{
   auto ret = std::make_shared<realType >(std::forward<Args>(args)...);
   return ret;
}
```

这样通过针对构造函数的封装实现了基本的工厂操作。为了针对构造过程进行控制，可以进一步修改 create_shared()函数，通过传递函数对象将特定初始化操作延迟到函数对象中进行处理。

create_callback()方法提供了用户可定制的错误和后续处理机制，允许用户传入自定义的后续处理函数对象来进一步地处理创建对象后的操作。传统工厂模式通常只返回一个空指针或者抛出异常，而这种定制化的错误处理方式可以根据具体需求采取不同的处理策略，能够增加代码的可扩展性和可定制性，代码如下：

```
//designM/factory.hpp

template <typename... Args>
static std::shared_ptr<realType >
create_callback(
   std::function< void (std::shared_ptr<realType > ptr, Args&&...)>func,
   Args&&... args)
{
   auto ret = std::make_shared<realType >(std::forward<Args>(args)...);
   func(ret, std::forward<Args>(args)...);
   return ret;
}
```

其中，func 的函数原型是将实际创建的对象作为参数 ptr 传递给回调函数，在回调函数中可以进行额外的初始化处理。例如在某些情况下需要使用 std::enable_shared_from_this 模块，并且在要创建的对象中用到 shared_from_this()方法，由于在构造函数中还没有完成整个对象的构造操作，所以调用这种方法会出现问题。这时需要将使用 shared_from_this()函数的初始化工作放到另外一个初始化函数中进行，这个场景可以在回调函数中调用另外的初始化方法，回调函数的原型如下：

```
void func(std::shared_ptr<realType > ptr, Args&&... params);
```

(2) 模板函数实现方式以模板函数进行自动构造生成对象，不需要针对类型进行继承等其他的处理，使用起来会更加方便，代码如下：

```
//designM/factory.hpp

#include <type_traits>
template< typename type, typename Args...>
realType * product(Args&&... args){
```

```
    static_assert(is_class<type >::value&&is_constructible<type >::value, "");
    realType * ret = nullptr;
    ret = new type (std::forward<Args>(args)...);
    return ret;
}
```

在这段代码中使用了 std::is_class<>和 std::is_constructible<>进行编译期类型检查，确保了工厂函数只能用于类类型，并且能够正确地构造对象。这增加了代码的安全性和稳定性。

通过 std::forward<>()和完美转发方式传递参数，保证了传递给产品的构造函数的参数类型和值的完整性，避免了多余的复制操作，提高了代码的效率。

工厂函数相比模板类的工厂模式更加通用和灵活。它不再依赖于特定的产品类或特定的工厂类，而是一种可以根据需要创建不同类型对象的通用工厂函数。这种灵活性使代码更具可扩展性和可维护性。

虽然这种实现方式更加灵活，但是其中并没有实现原始类型的推断部分。在实际使用时需要根据具体情况进行选择。另外程序中没有针对内存分配失败和实际类在初始化过程中对异常情况进行处理，使用时需要在业务代码中针对此情况进行处理。

5.1.4　应用示例

在给出工厂模式的示例之前，先简单介绍 CRTP(Curiously Recurring Template Pattern)方式。在 CRTP 中，派生类(例如 class a)作为模板参数传递给基类模板(例如 product<a>)。基类可以通过派生类的类型信息实现一些特定的逻辑或行为。

在这个例子中，product<a>基类提供了静态函数 create 来创建 a 类的实例。通过 CRTP，派生类 class a 可以访问和继承 product<a>类中的成员和方法。

CRTP 的一个优点是它允许在编译时进行静态绑定和优化，而不需要虚函数的运行时开销；另一个优点是通过 CRTP，基类可以访问派生类的成员和方法，这为实现代码的重用和灵活性提供了机会。

需要注意的是，使用 CRTP 时应该小心并谨慎，因为错误地使用 CRTP 可能导致代码更复杂和难以维护。在使用 CRTP 时，应该权衡使用的利弊，并确保正确理解和适当使用该模式。

(1) 采用 CRTP 方式使用工厂模式。在这个例子中，struct a 通过公有继承方式继承自 product<a>类，product<a>为类 a 提供了工厂功能，这样在类 a 中就不需要另外实现工厂函数了，实现代码如下：

```
struct a : public product< a >
{
  a(int c);
};
a * pa2 = a::create(23);
```

(2) 采用模板类特化方式,直接特化使用,特化后就是针对特定类的工厂类。可以对实际的对象进行封装而不必暴露出来,避免直接使用原类,代码如下:

```
namespace pdt_prvite__
{
    class pdtA{ ... ... };
    class pdtB{ ... ... };
}
using cncrtPdtA = product<pdt_prvite__::pdtA>;
using cncrtPdtB = product<pdt_prvite__::pdtb>;
```

这段代码利用模板类特化的方式,对 product 模板进行重新命名,将具体的产品类类型从私有名字空间中暴露出来,在外部使用时如果不指明 pdt_prvite__名字空间,则产品类的实现就在外部不可见,这样就避免了调用构造函数对产品进行初始化。采用这种方式可以很方便地在已有代码的基础上进行工厂模式改造,而不需要对代码大幅度地进行修改。

(3) 工厂模板函数的使用示例。定义一个类并在其构造函数中完成对象的初始化。实际也是利用构造函数,示例代码如下:

```
class YourClass
{
public:
    YourClass(int value) : m_value(value)
    {
        //在构造函数中完成对象的初始化
    }
    //其他成员函数和成员变量
...
};
```

调用工厂函数创建对象。使用工厂函数来创建对象,传入的参数包括类名和构造函数的参数,代码如下:

```
YourClass* objPtr = product<YourClass>(42);
```

使用对象,可以调用对象的成员函数、访问对象的成员变量等,释放对象的内存。当不再需要对象时,需要手动调用 delete 运算符以释放对象的内存,代码如下:

```
objPtr->someFunction();
std::cout << objPtr->someVariable << std::endl;

delete objPtr;
```

5.2 抽象工厂模式及其实现

抽象工厂模式(Abstract Factory Pattern)提供了一种封装一组相关或相互依赖对象的创建方式而无须指定其具体类。

在软件开发中有时需要创建一组相关的对象,这些对象之间可能存在关联或依赖关系。

如果直接在客户端代码中创建这组对象，则将导致代码耦合度高、难以维护和扩展。在这种情况下使用抽象工厂模式可以很好地解决这个问题。

5.2.1　抽象工厂模式的传统结构

抽象工厂模式通过引入一个抽象工厂接口定义一组用于创建相关对象的方法。每个具体的工厂类都实现了抽象工厂接口，并负责创建一组相关的对象。客户端代码通过调用工厂接口的方法来创建对象，而无须知道具体的工厂类和对象。抽象工厂模式的核心组成部分包括抽象工厂接口(Abstract Factory)，其中定义一组用于创建产品的方法；具体工厂类(Concrete Factory)实现抽象工厂接口，负责创建一组相关的产品的具体逻辑；抽象产品接口(Abstract Product)定义一组产品的通用方法接口；最后是具体产品类(Concrete Product)实现抽象产品接口，并实现具体的产品类创建逻辑。

抽象工厂模式提供了一种封装一组相关对象的创建方式，便于对代码进行组织和管理；将客户端代码与具体工厂类和产品类解耦，从而使客户端代码更加灵活和可扩展。能够支持产品族的创建，可以创建一组相关的产品。传统抽象工厂模式 UML 的简图如图 5-3 所示。

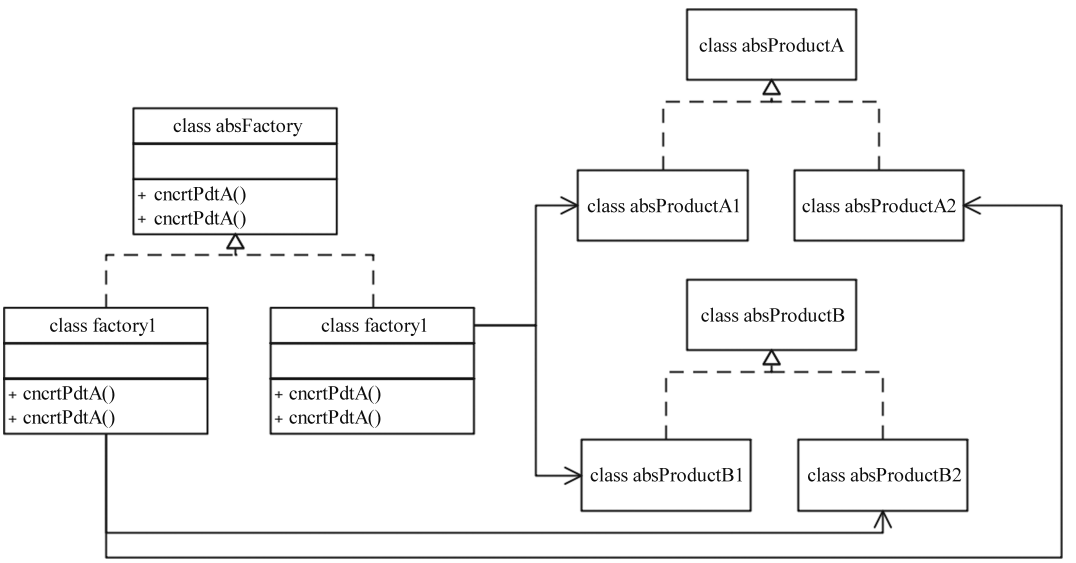

图 5-3　传统抽象工厂模式 UML 的简图

5.2.2　使用C++11实现抽象工厂模式的结构

传统方式的抽象工厂在需要增加产品时需要调整接口，增加新的工厂方法并实现其具体创建逻辑，这些工作都需要延迟到具体业务中。针对不同的业务场景需要进行重写或者修改代码。

通过C++11的元编程技术可以将这项工作独立出来，将创建通用的抽象工厂的相关逻

辑实现在工厂模式的类模板中。这样业务程序员就可以将主要的注意力集中在业务实现上，使设计模式的逻辑和业务逻辑解耦。

使用元编程技术实现在编译期构造抽象工厂的具体类型，并和相关的产品类相关联。这种关联的关系是在编译期实现的，是一种静态的抽象工厂模式，并不会造成运行期的负担。

在具体实现中采用可变模板参数来添加产品类型；生产接口使用可变参数的函数模板，可以很灵活地使用不同类型的参数和参数数量，可以提高代码的灵活性和可复用性。C++11元编程抽象工厂模式 UML 的简图如图 5-4 所示。

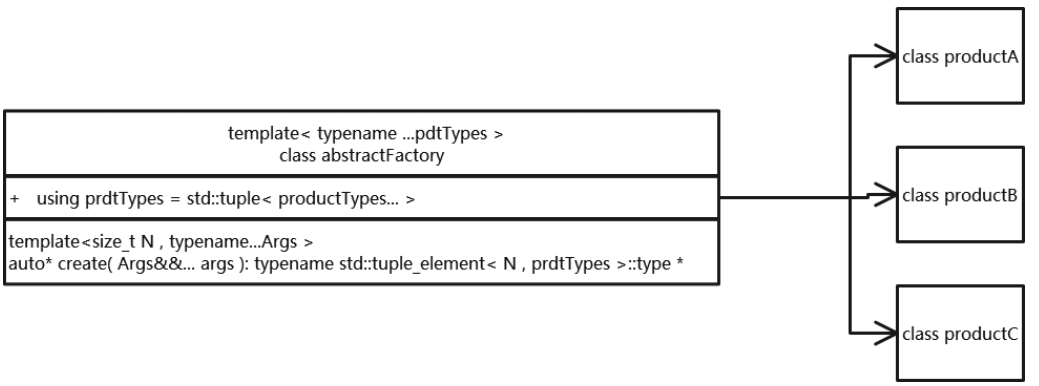

图 5-4 C++11元编程抽象工厂模式 UML 的简图

这种实现方法并没有严格地实现抽象工厂模式的所有内容，但仍符合抽象工厂模式的基本特征，如工厂方法的使用。对于不同类型的对象，使用工厂模式模块中的模板函数 factory()来创建相关的对象，这样客户端代码就不需要知道具体的实现细节了，从而实现了对象的解耦。这也提高了代码的可维护性和可扩展性。

5.2.3 工厂模式的实现和解析

实际实现使用了两种方式，一种是宏；另一种是模板类，其中采用宏实现的方式更加符合抽象工厂的完整概念，但是灵活性相对差一些；使用模板类实现则没有完整实现抽象工厂的内容，但是利用C++11的可变模板参数将生产参数、产品类型和抽象工厂解耦，具有了更好的通用性和灵活性。

(1) 宏实现方式主要通过宏来简化接口和实现两部分的编码方式，并在宏里面封装工厂方法。先看主要代码的第一部分，这一部分中通过宏声明接口类的名称、基础的结构和接口方法定义的宏，在宏 DECLARE_ABSTRACT_FACTORY 的实现中，考虑了因为接口类会作为基类被实现类继承，所以在宏实现中包含一个空的虚析构函数，从而避免在业务中写接口类时忘记编写虚析构函数。宏 ABST_PRODUCT_NAME 用来声明接口方法，这个宏直接将方法声明为纯虚函数，并在每个宏参数前增加“create_”前缀作为新的产品构造方法的名字。在宏中使用可变宏参数，因此一个宏用来声明不同参数表类型的接口。__VA_

ARGS__则是展开的宏参数，在实际使用这个宏时，可变宏参数是实际需要定义接口的参数表的数据类型，代码如下：

```
//designM/absFactory.hpp

#define DECLARE_ABSTRACT_FACTORY(absFactoryName)     \
class absFactoryName {                               \
public:                                              \
    virtual ~absFactoryName(){}

#define END_DECLARE_ABSTRACT_FACTORY() };
//声明接口函数，利用可变宏展开可以很方便地实现任意参数数量的接口声明
#define ABST_PRODUCT_NAME(productName, ...)          \
    virtual productName * create_#productName(__VA_ARGS__) = 0;
```

然后是代码的第二部分，这一部分实际上是针对第一部分的接口类型实现的宏工具，在这一步中利用了 5.1 节中的工厂模式定义了接口的实现部分代码。

宏 START_IMPL_ABST_FACTORY 用来声明接口实现类，宏参数 implFactoryName 是实现类的名字。实现类需要继承接口类 abstFactoryName，也就是使用上面声明接口宏声明的接口类，代码如下：

```
//designM/absFactory.hpp

START_IMPL_ABST_FACTORY(implFactoryName, abstFactoryName)
class implFactoryName: public abstFactoryName{
public:

#define END_IMPL_ABST_FACTORY() };
```

PRODUCT_NAME_XX 这个宏函数的操作是用来实现上面定义的虚函数接口，通过连接、可变宏参数实现灵活的函数声明和代码实现部分。在实际的实现代码中有 11 个相似的实现，它们的主要区别在于将 XX 替代为数字，具体的数字代表着可以使用的参数的数量。例如 0 代表实现的接口函数不需要参数，1 代表需要一个参数。宏参数表里面 productType 是实际需要进行生产的产品类型，baseType 是实际需要返回的数据类型，但是进行内存分配时使用 productType，代码如下：

```
//designM/absFactory.hpp

#define PRODUCT_NAME_0(productType, baseType)    \
virtual baseType * create_#baseType() override    \
{                                                 \
    return factory< productType >();               \
}
```