文章编号：2096-1472(2018)-09-18-05
DOI:10.19644/j.cnki.issn2096-1472.2018.09.006

基于闭包封装和序列展开实现“可选用参泛型抽象工厂”

闵 军[1]，覃凤清[2]
(1.宜宾学院图书馆,四川 宜宾 644000；
2.宜宾学院招生就业处,四川 宜宾 644000)

摘 要：C++11、C++14、C++17等新标准发布后，涌现出许多泛型抽象工厂的设计方案。不过，一般泛型抽象工厂的设计，都要求入参列表与用参列表必须一致。本文将在以往设计的基础之上，基于闭包封装和序列展开、使用C++1x的std::integer_sequence、lambda表达式、tuple等新技术，通过数据结构和代码设计的优化，给出一种“可选用参泛型抽象工厂”的实现方式。测试结果显示，该方式复用性更好、更为精简高效，优雅地实现了对产品类型可变、异类组合、参数可变、可选用参等需求的支持。该实现方式及代码具有较强实用价值，可以实际应用到软件项目中。

关键词：闭包封装；序列展开；integer_sequence；可选用参；泛型抽象工厂
中图分类号：TP311.1 **文献标识码**：A

Implementation of *Generic Abstract Factory of Optional Reference Parameter* Based on Closures Encapsulation and Sequence Expansion

MIN Jun[1],QIN Fengqing[2]
(1.*Library*,*Yibin University*,*Yibin* 644000,*China*;
2.*Recruitment and Employment Office*,*Yibin University*,*Yibin* 644000,*China*)

Abstract: After the release of new standards,such as C++11,C++14,and C++17,many designs of generic abstract factory have emerged.However,the design of general generic abstract factory requires the import parameter list and the reference parameter list to be consistent.Based on the previous design,this paper proposes an implementation of *Generic Abstract Factory of Optional Reference Parameter* based on closures encapsulation and sequence expansion.This scheme have implemented a configurable object factory by using new technologies such as C++1x std::integer_sequence,lambda expressions,tuple,optimizing data structures and code design.The test results show that the scheme is more reusable,more simplified and efficient.The scheme can gracefully implements support for variable product types,heterogeneous combinations,variable parameters,and optional reference parameter.This scheme and code have more practical values and can actually be applied in software projects.

Keywords:closures encapsulation;sequence expansion;integer_sequence;optional reference parameter;generic abstract factory

1 引言(Introduction)

抽象工厂模式主要是用于完成“多系列相互依赖的具体产品”的创建工作，避免客户程序和这种“多系列具体产品创建工作”的紧密耦合[1]。随着技术的发展和进步，人们先是使用多态机制和模板编程改进设计模式，随后采用的泛型编程技术是改进抽象工厂传统实现方式的一种有效手段[2]。C++11、C++14、C++17等新标准发布后，涌现出许多泛型抽象工厂的改进方案，比如将具体工厂构造函数存放到STL关联容器中实现自动注册、使用类模板和可变参数模板实现泛型工厂、使用lambda表达式和std::function类模板简化设计等。然而，一般泛型抽象工厂的设计，都要求入参列表与用参列表必须一致，本文将在以往改进的基础之上，基于闭包封装和序列展开、使用C++1x的std::integer_sequence、lambda表达式、tuple等新技术[3]，通过数据结构和代码结构的优化，给出一种“可选用参泛型抽象工厂”的实现方式。

2 基于闭包封装和序列展开实现“可选用参泛型抽象工厂”(Implementation of *Generic Abstract Factory of Optional Reference Parameter* based on closures encapsulation and sequence expansion)

以往泛型抽象工厂的设计，虽然使用了关联容器、可变参数模板等新技术进行优化[4]，但是它们都要求入参列表与用参列表必须一致。下面，本文将在以往设计的基础上进一步改进，介绍“可选用参泛型抽象工厂”的实现方式。

2.1 “可选用参泛型抽象工厂”的结构图

下面给出的图1便是“可选用参泛型抽象工厂”的结构图。

基金项目：国家自然科学基金(61202195)，四川省教育厅科研基金(13ZA0198).

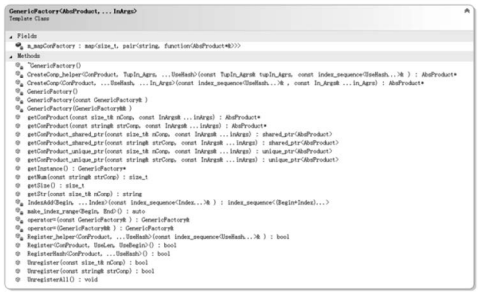

图1 “可选用参泛型抽象工厂”结构图

Fig.1 Structure chart of "Generic Abstract Factory of Optional Reference Parameter"

2.2 可选用参是指：“用参列表”可以任意选用“入参列表”中的参数

本文介绍的可选用参涉及两个概念，“入参列表”和“用参列表”。二者的关系可以通俗地比喻为餐馆菜单(入参列表)和用户点菜(用参列表)之间的关系。

这里的“入参列表”就是具体产品创建函数CreateConp的参数表(参见2.3)，它是在实例化泛型抽象工厂时确定的，定义后就不能改变。对每一个泛型抽象工厂实例来说入参列表都是固定不变的，它规定了用户在创建具体产品时必须输入的参数列表，所以简称为入参列表。入参列表类似于餐馆提供的供应菜单(在实际使用中，入参列表可以使用函数封装、通过函数缺省参数方式进行简化)。

这里的“用参列表”就是具体产品构造函数ConProduct的参数表(参见2.3)，它是在已经定义的泛型抽象工厂实例中注册具体工厂时确定的，它规定了构造具体产品对象时实际使用的参数列表，因而简称为用参列表，这就类似于顾客点菜清单。

以往泛型抽象工厂的设计，都要求这两个参数列表必须一致，本文所说的可选用参，就是希望在注册和使用具体工厂创建具体产品实例时，“用参列表”可以任意选用“入参列表”中的参数，既可以连续选用、也可以散列选用、也可以不选。这就类似于，顾客到餐馆用餐时可以任意点菜，即可以任意点选餐馆供应菜单(入参列表)中的菜品享用，顾客点选的菜单就是(用参列表)。当然，用参列表必须小于等于入参列表，也就是说顾客点菜菜单不能超出餐馆供应菜单的范围。

2.3 具体产品创建函数和具体产品构造函数的不同

在“可选用参泛型抽象工厂”的设计中，请注意区分具体产品创建函数CreateConp(以“入参列表”为参数表，类似餐馆菜单)和具体产品构造函数ConProduct(以“用参列表”为参数表，类似用户点菜)的不同。简单地说，具体产品的创建函数CreateConp包含和封装了具体产品的构造函数ConProduct。数据成员m_mapConFactory里直接存放的是包含lambda表达式funConp的std::function指针；funConp中含有CreateConp；CreateConp通过CreateConp_helper又包含ConProduct，可以参见后面的完整代码。它们之间的包含关系是：

m_mapConFactory std::function<funConp> CreateConp CreateConp_helper ConProduct。

对每一个泛型抽象工厂实例来说，具体产品创建函数CreateConp的类型都是唯一的，其返回值是抽象产品类指针、其参数表就是上面介绍的“入参列表”，入参列表是在实例化泛型抽象工厂时确定的。在同一种泛型抽象工厂实例中，可以注册不同用参列表的具体工厂，每一种具体工厂对应一种具体产品类型定义的构造函数ConProduct，具体产品构造函数的返回值必须是抽象产品类指针、其参数表就是上面介绍的“用参列表”，用参列表可以任意选择入参列表中的参数。

2.4 数据结构和代码优化

在泛型抽象工厂设计中，一般都设置了一个关键数据成员m_mapConFactory，其中存放着具体产品标识与具体产品创建函数之间的对应信息。在本文提供的实现方式中，以lambda表达式作为具体产品的创建函数，关键数据成员m_mapConFactory中要保存的便是以std::function类模板封装的lambda表达式[5]。“可选用参泛型抽象工厂”的代码设计也进行了相应优化，可参见后面的完整代码。

3 通过闭包封装实现类型擦除(To erase type based on the closures encapsulation)

要实现“可选用参泛型抽象工厂”，需要将不同类型具体产品创建函数的数据保存到同一个STL容器之中，我们可以借助类型擦除技术来实现[6]。本文将使用闭包封装技术来实现类型擦除。

简单地说，闭包(closure)就是带状态的函数。闭包封装就是借助闭包来擦除类型，实质上是把一些类型信息保存到闭包之中隐藏起来，从而实现类型擦除的目的。C++11的lambda表达式就是一种闭包类型，可以把具体类型作为lambda表达式的参数隐藏起来，再用std::function类模板封装lambda表达式，就可以通过lambda表达式来实现闭包封装[7]。由于闭包本身的类型是确定的，因此可以把它存放到普通STL容器中，需要时再从闭包中取出其具体类型。

本文介绍的“可选用参泛型抽象工厂”的实现方式，就是通过lambda表达式来实现闭包封装，达到类型擦除的目的。具体实现可参见后面完整代码中的m_mapConFactory容器、Register_helper函数等相关内容。

4 通过序列展开实现可选用参(To implement optional reference parameter based on the sequence expansion)

4.1 利用C++14的std::integer_sequence展开tuple作为函数参数

“可选用参”是“可选用参泛型抽象工厂”的关键特性，我们可以利用C++14的std::integer_sequence展开tuple作为函数参数的方式来具体实现。C++11中新增的std::tuple元组类模板是std::pair的通用扩展，C++14中新增的std::integer_sequence整型序列类模板用于实现编译期整数序列。有了这两样工具，我们便可以借助C++11的可变参数模板和std::integer_sequence的模板参数自动推导机制，实现展开tuple作为函数参数的功能[8]。具体可参见后面完整代码中的Register_helper、CreateConp和CreateConp_helper等函数。

4.2 获取tuple中指定序列作为函数参数

上面我们已经实现了使用C++14的std::integer_sequence展开tuple作为函数参数的功能。但是，这种方式还只能简单地将tuple中的所有元素展开作为函数参数。我们要实现泛型抽象工厂的“可选用参”，就需要实现获取tuple中指定序列的元素组作为函数参数的功能。

为此，一般的整数序列就不能满足要求了，我们需要获取指定起点和终点的整数序列。通过对std::integer_sequence的简单变换，就可以很优雅地生成Begin到End−1的升序序

列，具体实现可参见后面完整代码中的make_index_range和IndexAdd函数。有了make_index_range这一利器，我们便可以轻松实现获取tuple中指定序列的元素作为函数参数的功能。另外，要实现指定tuple中任意散列的元素作为函数参数的功能，可以直接使用可变参数模板std::index_sequence<UseHash...>进行指定即可。借助以上两种方式，便可以设计出“可选用参泛型抽象工厂”。具体实现可参见后面完整代码中的Register、RegisterHash、CreateConp和CreateConp_helper等相关函数。

4.3 本文提供两种任意选择用参序列的方式

为了方便注册，本文介绍的“可选用参泛型抽象工厂”设计了两种具体工厂注册函数Register和RegisterHash，用户可以通过两种方式任意指定用参序列，不过，使用时都需要注意不能越界。具体实现可参见后面完整代码中的Register、RegisterHash。

5 “可选用参泛型抽象工厂”的完整实现代码(Complete implementation code of "Generic Abstract Factory of Optional Reference Parameter")

以下是本文介绍的“可选用参泛型抽象工厂”的完整实现代码。用户需要注意的是，下面提供的代码是基于C++14新标准实现的，需要在支持C++14的编译器中才能正常编译，比如Visual Studio 2015[9]、CodeBlocks 16.01 with GCC 6.1及以上版本[10]。

```
//GenericFactory.h，泛型抽象工厂头文件v3.3.2
//需支持C++14的编译器，如Visual Studio 2015、CodeBlocks 16.01 with GCC 6.1及以上版本
//为了便于编辑排版，以下代码做了一些适当的调整变换
#pragma once
#include <map>
#include <tuple>
#include <memory>
#include <functional>
#include <string>
using std::map;using std::pair;using std::string;using std::function;
using std::index_sequence;using std::make_tuple;using std::get;
using std::make_index_sequence;using std::shared_ptr;using std::unique_ptr;
//泛型工厂类GenericFactory包含2个模板参数：抽象产品类、入参列表
template<typename AbsProduct,typename...InArgs>
class GenericFactory    //AbsProduct为Abstract Product缩写
{
private:
GenericFactory &operator=(GenericFactory&&)=delete;
GenericFactory&operator=(const GenericFactory&)=delete;
    GenericFactory(GenericFactory&&)=delete;
    GenericFactory(const GenericFactory&)=delete;
    ~GenericFactory() {}
    GenericFactory() {}
    map<size_t,pair<string,function<AbsProduct*(const InArgs&...)>>> m_mapConFactory;
    ////获取tuple中指定序列作为函数参数
    //3个模板参数：具体产品类ConProduct，用参列表序列UseHash，入参列表In_Args
    template<typename ConProduct,size_t...UseHash,typename...In_Args>
    static AbsProduct*CreateConp(const index_sequence<UseHash...>&,const In_Args&... in_Agrs)
    {
    return CreateConp_helper<ConProduct>(make_tuple(in_Agrs...),index_sequence<UseHash...>());
    }
    template<typename ConProduct,typename TupIn_Agrs,size_t...UseHash>
    static AbsProduct*CreateConp_helper(const TupIn_Agrs&tupIn_Agrs,const index_sequence<UseHash...>&)
    {
    return new ConProduct(get<UseHash>(tupIn_Agrs)...);
    }
    ////获取tuple中指定序列作为函数参数
    ////用C++14库生成Begin到End-1的升序序列，无VS2017编译时递归层数需小于499的限制
    //例如：make_index_range<2,7>() 在x86时得到struct std::integer_sequence<unsigned int,2,3,4,5,6>
    template<size_t Begin,size_t End>//使用size_t有利于提高代码的可移植性、有效性和可读性
    static auto make_index_range()
    {
        return IndexAdd<Begin>(make_index_sequence<End-Begin>());
    }
    template<size_t Begin,size_t...Index>
    static index_sequence<(Begin+Index)...> IndexAdd(const index_sequence<Index...>&)
    {
    return {};      //返回该函数返回类型的缺省值
    }
    ////用C++14库生成Begin到End-1的升序序列，无VS2017编译时递归层数需小于499的限制
    template<typename ConProduct,size_t...UseHash>
    bool Register_helper(const index_sequence<UseHash...>&)
    {
    auto funConProduct=[](const InArgs&...inArgs)
    { return CreateConp<ConProduct>(index_sequence<UseHash...>(),inArgs...);};
    size_t nConProduct=(0==m_mapConFactory.size()? 0 : ((--m_mapConFactory.end())->first)+1);
```

```
        string strConProduct=typeid(ConProduct).name();
        auto pairReturn=m_mapConFactory.emplace(nConProduct,make_pair(strConProduct,funConProduct));
                return pairReturn.second;
        }
    public:
        static GenericFactory*getInstance()
        {
         static GenericFactory Singleton_Generic_Factory;
                return &Singleton_Generic_Factory;
        }
        //Register有3个模板参数：具体产品类ConProduct(必须是AbsProduct的子类),
        // 用参序列长度UseLen(缺省为所有入参)，用参序列的0基起始序号UseBegin(缺省为0)。注意不能越界
        template<typename ConProduct,size_t UseLen=sizeof...(InArgs),size_t UseBegin=0>
        bool Register()                    //Conp、ConProduct为Concrete Product缩写
        {
        return Register_helper<ConProduct>(make_index_range<UseBegin,UseLen+UseBegin>());
        }
        //RegisterHash有2个模板参数：具体产品类ConProduct(必须是AbsProduct的子类),
        //用参序列UseHash(缺省为空，由0到sizeof...(InArgs)-1之间的正整数构成的散列表，注意不能越界)
        template<typename ConProduct,size_t...UseHash>
        bool RegisterHash()      //Conp、ConProduct为Concrete Product缩写
        {
        return Register_helper<ConProduct>(index_sequence<UseHash...>());
        }
        bool Unregister(const size_t&nConp)
        {
                bool bRet=(m_mapConFactory.erase(nConp)==1);
                return bRet;
        }
        bool Unregister(const string&strConp)
        {
                size_t nConp=getNum(strConp);
                return Unregister(nConp);
        }
        void UnregisterAll()
        {
                m_mapConFactory.clear();
        }
        size_t getSize() const
        {
                size_t nRet=m_mapConFactory.size();
                return nRet;
        }
        string getStr(const size_t&nConp)
        {
                string strRet=(m_mapConFactory.find(nConp)==m_mapConFactory.end())?string():m_mapConFactory[nConp].first;
                return strRet;
        }
        size_t getNum(const string&strConp)const
        {
        string str;
        for (auto mapTemp:m_mapConFactory)
        {
        str=mapTemp.second.first;
        if (strConp==str) {return mapTemp.first;}
        }
        return SIZE_MAX;
        }
        AbsProduct*getConProduct(const size_t&nConp,const InArgs&...inArgs)
        {
        AbsProduct*pRet=(m_mapConFactory.find(nConp)==m_mapConFactory.end()) ? nullptr : m_mapConFactory[nConp].second(inArgs...);
                return pRet;
        }
        AbsProduct*getConProduct(const string&strConp,const InArgs&... inArgs)
        {
        AbsProduct*pRet=getConProduct(getNum(strConp),inArgs...);
        return pRet;
        }
        shared_ptr<AbsProduct>getConProduct_shared_ptr(const size_t&nConp,const InArgs&...inArgs)
        {
        shared_ptr<AbsProduct>ptrRet(shared_ptr<AbsProduct>(getConProduct(nConp,inArgs...)));
        return ptrRet;
        }
        shared_ptr<AbsProduct>getConProduct_shared_ptr(const string&strConp,const InArgs&... inArgs)
        {
        shared_ptr<AbsProduct>ptrRet(shared_ptr<AbsProduct>(getConProduct(getNum(strConp),inArgs...)));
                return ptrRet;
        }
        unique_ptr<AbsProduct>getConProduct_unique_ptr(const size_t&nConp,const InArgs&...inArgs)
        {
        shared_ptr<AbsProduct>ptrRet(unique_ptr<AbsProduct>(getConProduct(nConp,inArgs...)));
```

```
            return ptrRet;
        }
        unique_ptr<AbsProduct>getConProduct_unique_ptr(const string&strConp,const InArgs&...inArgs)
        {
        shared_ptr<AbsProduct>ptrRet(unique_ptr<AbsProduct>(getConProduct(getNum(strConp),inArgs...)));
            return ptrRet;
        }
    };
```

6 “可选用参泛型抽象工厂”的实际使用(Actual use of *Generic Abstract Factory of Optional Reference Parameter*)

6.1 具体产品构造函数的参数可以任意选择

该例中定义了MyShape基类，以及三个子类MyRect、MyLine、MyCircle，借助本文设计的“可选用参泛型抽象工厂”，便可以优雅地实现众多具体工厂的注册和具体产品的创建。MyRect、MyLine两个子类的构造函数都包含三个参数、但顺序不同，MyCircle子类与前两个子类的构造函数参数个数、类型都不相同，其结构图见图2。

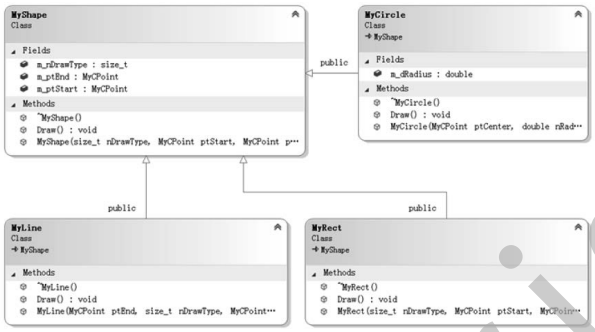

图2 具体产品构造函数的参数可以任意选择

Fig.2 Parameters of concrete factory constructor can be optional

如果使用一般泛型抽象工厂设计，每种不同参数的具体工厂都需要单独注册一个泛型抽象工厂实例，代码繁琐低效[11]。使用本文介绍的“可选用参泛型抽象工厂”，在入参列表一致的情况下，可以将不同用参列表的具体工厂都注册到一个泛型抽象工厂实例之中，然后简单地使用循环方式便可以实现用户业务功能的调用，代码得以大大简化，注册的具体工厂越多，简化得就越为显著。示例代码如下[11]：

```
    //在入参一致的情况下，可将不同用参的具体工厂都注册到一个泛型抽象工厂实例之中
    //模板参数是：抽象产品类、入参列表
    auto pGF=GenericFactory<MyShape,size_t,MyCPoint,MyCPoint,double>::getInstance();
    pGF->Register<MyRect,3>();
//注册具体工厂时，指定用参序列的长度UseLen:3
    pGF->RegisterHash<MyLine,2,0,1>();          //注册具体工厂时，指定用参乱序散列表UseHash:2,0,1
    pGF->RegisterHash<MyCircle,1,3>();          //注册具体工厂时，指定用参散列表UseHash:1,3
    //可用循环方式创建具体产品并调用其Draw函数
    size_t n=pGF->getSize();
    for(size_t i=0;i<n;i++)
    {
    pGF->getConProduct_shared_ptr(i,i,MyCPoint(3,4),MyCPoint(5,6),3.3)->Draw();
    }
```

6.2 具体工厂的异类组合

讨论抽象工厂时，有一个经常用到的案例，不同电脑厂商生产各种电脑部件的例子。假设Dell电脑需要自己生产机箱、主板、内存、显卡等四种部件，联想电脑需要自己生产机箱、主板、内存、显卡、键盘、鼠标等六种部件。借助本文改进的“可选用参泛型抽象工厂”，使用函数封装不同抽象工厂的需求，便能够更为优雅地实现该例中的异类组合[5]。

7 结论(Conclusion)

综上所述，抽象工厂模式的实现方式一直都在不断改进和完善。C++11、C++14、C++17等新标准发布之后，涌现出许多泛型抽象工厂的改进方案。然而，一般泛型抽象工厂的设计，都要求入参列表与用参列表必须一致。本文在以往设计的基础之上，基于闭包封装和序列展开、使用C++1x的std::integer_sequence、lambda表达式、tuple等新技术，通过数据结构和代码设计的优化，给出一种“可选用参泛型抽象工厂”的实现方式。测试结果显示，该方式复用性更好、更为精简高效，优雅地实现了对产品类型可变、异类组合、参数可变、可选用参等需求的支持。该实现方式及代码具有较强实用价值，可以实际应用到软件项目中。

参考文献(References)

[1] Design Patterns:Elements of Reusable Object-Oriented Software[M].USA:Addison-Wesley Professional,1994:99.

[2] Bjarne Stroustrup.The C++ Programming Language Fourth Edition[M].USA:Addison-Wesley Professional,2013:699.

[3] std::integer_sequence[EB/OL].http://en.cppreference.com/w/cpp/utility/integer_sequence,2018-07-06.

[4] Variadic arguments[EB/OL].http://en.cppreference.com/w/cpp/language/variadic_arguments,2018-02-22.

[5] 闵军,罗泓.用lambda表达式和std::function类模板改进泛型抽象工厂设计[J].软件工程,2017,24(09):9-14.

[6] 祁宇.深入应用C++11:代码优化与工程级应用[M].北京:机械工业出版社,2015:267.

[7] Stephen Prata.C++ Primer Plus,Sixth Edition[M].USA:Addison-Wesley Professional,2011:1184-1196.

[8] Invoke the Callable object f with a tuple of arguments [EB/OL].https://doc.bccnsoft.com/docs/cppreference2015/en/cpp/experimental/apply.html,2018-07-02.

[9] Support For C++11/14/17 Features(Modern C++) [EB/OL].https://msdn.microsoft.com/en-us//library/hh567368.aspx,2018-07-02.

[10] C++ Standards Support in GCC [EB/OL].https://gcc.gnu.org/projects/cxx-status.html,2018-08-13.

[11] 闵军,罗泓.C++11实现可变参数泛型抽象工厂[J].软件工程,2017,20(05):18-22.

作者简介：

闵　军(1966-)，男，硕士，研究员.研究领域:C++程序设计，设计模式，计算机网络.

覃风清(1976-)，女，博士，研究员.研究领域:图像处理与信息系统，多媒体通信.