文章编号：2096-1472(2019)-05-18-07
DOI:10.19644/j.cnki.issn2096-1472.2019.05.005

基于C++17的泛型函数容器实现方法研究

闵 军[1]，罗 泓[2]
(1.宜宾学院图书馆，四川 宜宾 644000；
2.宜宾学院文学与新闻传媒学院，四川 宜宾 644000)

摘 要：泛型函数容器的使用可以解耦对象之间的调用关系，有利于实现高内聚、低耦合的软件设计原则。C++标准库中并没有这样的容器，用C++旧标准实现也很困难、很低效。C++1x等新标准发布后，出现了一些更好的实现方式。本文将在已有设计的基础之上，基于C++17新标准，利用if constexpr、fold expression、std::invoke等新技术，提供一种泛型函数容器的实现方式。测试表明该实现方式简洁高效，解决了重载函数和某些特殊函数的注册调用问题，可以显著降低耦合性、提高代码复用性。

关键词：C++17；泛型；函数容器；高内聚；低耦合
中图分类号：TP311.1 **文献标识码**：A

Study on the Implementation of Generic Function Container Based on C++17

MIN Jun[1],LUO Hong[2]
(1.*Library*,*Yibin University*,*Yibin* 644000,*China*;
2.*College of Literature and Journalism*,*Yibin University*,*Yibin* 644000,*China*)

Abstract: The application of generic function containers can decouple the calling relationships between objects,conducive to the realization of high cohesion and low coupling software design principles.There is no such container in the C++ standard library,and it is very difficult and inefficient to implement with the old C++ standard.The release of new standards,such as C++1x,has brought some better implementation methods.This paper provides a generic function container implementation method based on the existing design and the new C++17 standard,via some new technologies such as if constexpr,fold expression,and std::invoke.Test results show that the simple and efficient implementation method effectively solves the problem of registration and calling of overloaded functions and some special functions,significantly reducing coupling and improving code reusability.

Keywords:C++17;generic;function container;high aggregation;low coupling

1 引言(Introduction)

高内聚、低耦合是软件设计的基本原则，泛型函数容器的使用可以解耦对象之间的调用关系，有利于实现软件设计的这一基本原则[1]。作为一种万能函数注册器，泛型函数容器可以将任意类型的函数用一个key进行注册以供其他程序调用，可以注册普通函数、函数模板、成员函数、函数对象、lambda表达式、重载函数和某些特殊函数等。当全局函数或对象之间存在交互调用需求时，比如需要相互调用对方成员函数或者不适合关联的无关对象之间需要调用其他对象的成员函数或者全局函数需要调用成员函数等类似需求时，我们便可以将需要被调用的函数用一个key注册起来以供其他实体调用。函数的调用者不必知道被调用者，二者都依赖于中间的泛型函数容器，借此便可以解耦对象之间的调用关系[2]。

C++标准库中并没有现存的泛型函数容器，用C++旧标准实现也很困难、很低效[3]。C++11、C++14、C++17等新标准发布后，出现了一些更好的实现方式。本文便是在已有设计的基础之上，基于C++17新标准，利用if constexpr、fold expression、std::invoke等新技术，提供一种泛型函数容器的实现方式[4,5]。

2 泛型函数容器的结构设计(Design of generic function container structure)

泛型函数容器的基本结构与C++ STL中的容器类似。在

把各种不同类型的函数存入容器时必须转换为统一的数据结构，通过一个key进行注册，此后用户便可以借助这个key和必要信息提取已注册的记录来执行该函数，从而实现泛型函数容器的基本功能。

2.1　泛型函数容器的结构图

图1给出了泛型函数容器主体部分的结构图。泛型函数容器类function_container包含关键数据成员m_mapInvoker、注册和调用函数等成员，该类的各种构造器、赋值操作都私有化，实现简单的单例模式。

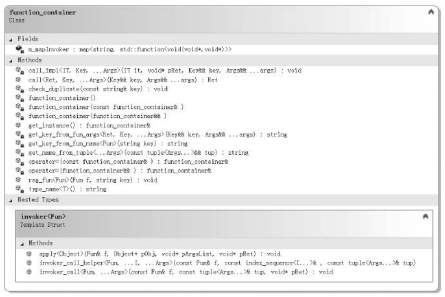

图1 泛型函数容器的结构图

Fig.1 Structure chart of Generic Function Container

2.2　函数存取结构的设计

由于泛型函数容器需要存取各种不同类型的函数，具体包括普通函数、函数模板、成员函数、函数对象、lambda表达式、重载函数和某些特殊函数等，所以需要抽象出能够区分各种不同函数的标志特征：函数签名(Function Signature)。类似于用签名可以识别不同人一样，通过函数签名便可以识别不同的函数[6]。从便于注册和提取的角度，本项目主要关注函数签名的四个部分：函数名f、函数所属对象的指针pObj、参数包指针pArgsList、返回值指针pRet。若是成员函数，pObj必须明确赋值，否则就必须赋值为空指针nullptr。

然后设计一个函数封装类invoker<Fun>，用来保存注册函数的函数签名。Fun是注册函数的具体类型(包含函数签名)，每注册一个函数就创建一个以模板参数Fun区分的该类对象，每个不同对象保存了注册函数的函数签名(包含四个部分)，调用时便可以从中提取函数签名来执行该函数。

2.3　函数注册和提取部分的设计

函数注册时需要完整保存函数签名各个部分的信息，不过调用时并不需要用户提供完整的函数签名。为了简化调用方式，函数调用时用户只需输入函数参数列表和返回值类型即可，若是无返回值函数则只需输入函数参数列表(可以理解为返回值为void类型)。按照以上设计思路，注册函数时需要完整保存函数签名的四个部分，调用函数时用户只需输入两个参数即可。因此，设计注册函数reg_fun时需要有四个参数，设计调用函数call时只需有两个参数即可。具体实现可参见后面完整代码中的reg_fun、call。

3　关键数据成员m_mapInvoker的设计(Design of the key data member m_mapInvoker)

3.1　关键数据成员m_mapInvoker保存的是key与函数的成对记录

我们可以使用std::map容器来保存泛型函数容器的注册数据。在本项目中，关键数据成员m_mapInvoker便是用于保存key与函数成对记录的std::map容器。m_mapInvoker的key字段为std::string类型，是用户指定名称或者返回值及参数类型名称的累加字符串；data字段则是std::function类模版封装的特殊函数类型[7]。

若用户注册时指定了注册名称便以此为该条记录的key，此时的key用std::string类型，不至于产生混淆。不过在调用该注册函数时，若直接使用std::string类型的key，便可能与参数类型产生混淆，因为注册函数的第一个参数也可能是std::string类型。所以，在调用该注册函数时，必须使用key_fun类型的key，才能避免可能发生的混淆。fun_key是只含一个数据成员std::string key的简单封装类型，其主要作用就是在调用时避免与函数参数发生混淆。

若用户注册时未指定注册名称，便使用返回值及参数类型名称的累加字符串作为该条记录的key，提取时也会自动生成返回值及参数类型名称的累加字符串作为key来查询调用该注册函数。这样的设计，同时也解决了参数列表相同、返回值不同的多个函数的注册调用问题。

m_mapInvoker的data字段用于保存函数签名的四个部分，它是以std::function类模版封装的特殊函数void(void*,void*)。该封装函数无返回值，第一个参数为注册函数的参数包指针，第二个参数为注册函数的返回值指针。

具体实现可以参见完整代码中的fun_key类(其结构参见图2)、call_impl、get_key_from_fun_args等函数，以及后面的测试代码。

3.2　封装函数的原始模型

关键数据成员m_mapInvoker的data字段中保存的并非单纯的封装函数，而是通过std::bind绑定到原始函数模型上的间接函数[8]。封装函数的原始模型是invoker<Fun>::apply<Object>(…)，Fun是注册函数的函数类型(包含函数签名)，Object是注册函数所属的对象类型(若是非成员函数则为void)。原始函数invoker<Fun>::apply<Object>(…)有四个参数，分别用于关联注册函数类型（函数签名）的四个部分：函数名f、成员函数的对象指针pObj、参数包指针_1、返回值指针_2。_1、_2为C++11新标准的占位符[9]，是用户在提取记录调用注册函数时需要输入的参数，若注册函数无返回值，则只需输入第一个参数即可。具体可以参见后面的完整代码。

3.3 解决重载函数和某些特殊函数的注册调用问题

在以上设计的泛型函数容器中，若直接注册存在两个以上实例的重载函数，编译时就会报错。解决该问题的思路很简单，就是针对重载函数的多个实例对应地定义多个不同名称和类型的函数指针、并将重载函数赋值给它们，这就相当于将多个重载函数的实例转换成为多个不同名称和类型的新函数指针。使用这些新的函数指针便能在泛型函数容器中成功进行注册和调用。另外，也可以用lambda表达式来消除重载函数的二义性[10]。对某些特殊函数的处理也类似，包括特殊函数1：参数列表相同、返回值不同的多个函数的注册和调用，这在本项目设计中已经解决；特殊函数2：参数列表相同、返回值相同的多个函数的注册和调用，可以用lambda表达式封装该函数，增加一个参数即可。具体请参见后面的测试代码。

4 利用C++17新技术优化代码设计(Optimize code design with C++17 new technology)

在本项目中，使用了C++17的if constexpr[11]新技术在编译期进行判断，去除enable_if_t，合并许多功能类似的函数。包括：合并非成员函数、成员函数注册的两种reg_fun函数；合并有返回值、无返回值的两种call函数；合并Key与fun_key类型相同和不同的两种call_impl、get_key_from_fun_args函数等。

在C++17之前，我们经常用逗号表达式和std::initializer_list将变参依次传入一个函数。用C++17的fold expression折叠表达式代替initializer_list，就要简洁得多[12]。本项目便使用了C++17的这一新技术简化代码设计。

利用C++17的invoke调用器，可以合并非成员函数、成员函数的调用，统一使用std::invoke(f,pArgsList)这种简捷形式进行调用[13]。当然，成员函数调用时，对象实例必须放在pArgsList的首位，作为第一个参数。在本项目中，便使用了C++17的这一新技术简化代码设计。

5 C++17泛型函数容器的完整实现代码(Complete implementation code of C++17 generic function container)

5.1 泛型函数容器的完整实现代码

以下便是本文介绍的泛型函数容器的完整实现代码。用户需要注意的是，以下代码是基于C++17新标准实现的，需要在支持C++17的编译器中才能够正常编译，比如Visual Studio 2017 15.3[14]、CodeBlocks 17.12 with GCC 7.2及以上版本[15]。

```
//function_container.h，泛型函数容器v1.1.5
#include<string>
#include<map>
#include<tuple>
#include<functional>
#include<cassert>
#include "function_traits.h"
struct fun_key {std::string key;};
class function_container
{
public:
    template<typename Fun>
//本项目中，Fun为Function的缩写
    void reg_fun(Fun f,std::string key="")
    {
    using namespace std::placeholders;
    std::string strkey=get_key_from_fun_name<Fun>(key);
    check_duplicate(strkey);
    //用C++17去除enable_if_t，合并非成员函数、成员函数注册两个函数
    if constexpr(std::is_member_function_pointer<Fun>::value){
    using Object=typename function_traits<Fun>::object_type;
    Object* pObj;
    //这样更为通用。避免创建Object对象时必须传入初始参数
//用std::bind绑定成员函数前面要加&。_1、_2为占位符
//将所有注册的函数都分解为4个部分：函数名f、[成员函数的对象指针pObj]、参数包指针_1、返回值指针_2
m_mapInvoker[strkey]={std::bind(&invoker<Fun>::template apply<Object>,f,pObj,_1,_2)};
                }
                else {
m_mapInvoker[strkey]={std::bind(&invoker<Fun>::template apply<void>,f,nullptr,_1,_2)};
                }
    }
    template<typename Ret=void,typename Key,typename...Args>
    Ret call(Key&& key,Args&&...args){
//用C++17合并有返回值、无返回值两个函数
    std::string strKey=get_key_from_fun_args<Ret>(std::forward<Key>(key),std::forward<Args>(args)...);
    auto it=m_mapInvoker.find(strKey);
    assert(it !=m_mapInvoker.end());
```

```
    //若没找到strKey关联的函数，异常报错
        if constexpr(std::is_void<Ret>::value){
        call_impl(it,nullptr,std::forward<Key>(key),std::
forward<Args>(args)...);
                }
                else {
        Ret ret;
        call_impl(it,&ret,std::forward<Key>(key),std::f
orward<Args>(args)...);
                        return ret;
                }
        }
        static function_container& get_instance(){
                static function_container instance;
                return instance;
        }
    private:
        function_container(){};
    //该类的各种构造器、赋值操作都私有化
        function_container(const function_
container&)=delete;
        function_container(function_container&&)=delete;
        function_container&operator=(const function_
container&)=delete;
        function_container&operator=(function_
container&&)=delete;
        //关键数据成员：std::map<用户指定名称或返回
值及参数类型名称累加字符串,封装函数的function指针>
        //封装函数是bind了的invoker<Fun>::apply(...)
        std::map<std::string,std::function<void(void*,
void*)>> m_mapInvoker;
    template<typename IT,typename Key,typename...Args>
        void call_impl(IT it,void*pRet,Key&&
key,Args&&...args){
    if constexpr(std::is_same<Key,fun_key>::value){
    //用C++17合并Key与fun_key类型相同和不同的情况
                        it->second(&std::make_tuple(std
::forward<Args>(args)...),pRet);
                }
                else {
            it->second(&std::make_tuple(std::forward
<Key>(key),std::forward<Args>(args)...),pRet);
                }
        }
        void check_duplicate(const std::string& key){
                auto it=m_mapInvoker.find(key);
                assert(it==m_mapInvoker.end());
    //若找到key关联的函数，说明重复注册，便异常报错
        }
        template<typename Fun>
        //注册函数时使用。这里的key应为string类型
        std::string get_key_from_fun_name(std::string
key){
        if(key.empty()){
        using tuple_type=typename function_
traits<Fun>::bare_tuple_type;
        using return_type=typename function_
traits<Fun>::return_type;
        std::string str="";
        if constexpr(std::is_member_function_
pointer<Fun>::value){
        using Object=typename function_
traits<Fun>::object_type;
        str=type_name<Object*>();
                        }
        key=type_name<return_type>()+str+get_name_
from_tuple(tuple_type{});
                }
                return key;
        }
        //这里的key若为fun_key类型便是函数标识(避免与
普通的参数类型混淆)，若key非fun_key类型便作为普通参数
类型处理
        template<typename Ret,typename
Key,typename...Args>
        std::string get_key_from_fun_args(Key&&
key,Args&&...args)
    //调用已注册函数时使用。
        {
    //用C++17合并Key与fun_key类型相同和不同的情况
        if constexpr(std::is_same<Key,fun_key>::value){
                return key.key;
                }
                else {
                return type_name<Ret>()+get_name_
from_tuple(std::make_tuple(std::forward<Key>(key),
std::forward<Args>(args)...));
                }
```

```
        }
        template<typename... Args>
        //获取各个参数类型易读名称的累加字符串
        std::string get_name_from_tuple(const std::tuple<Args...>&& tup){
        return(...+type_name<Args>());
    //用C++17的fold expression折叠表达式，比initializer_list要简洁得多
        }
        template<class T>
        //获取类型易读名称字符串
        std::string type_name(){
        using TR=typename std::remove_reference<T>::type;
        std::string name=typeid(TR).name();
        if(std::is_const<TR>::value){name+=" const";}
        if(std::is_volatile<TR>::value){name+=" volatile";}
        if(std::is_lvalue_reference<T>::value){name+="&";}
        else if(std::is_rvalue_reference<T>::value){name+="&&";}
                return name;
        }
        //调用器(函数)封装类。每注册一个函数，就创建一个以模板参数Fun区分的该类对象。
        //每个不同对象保存了注册调用器的函数签名，调用时需用function_traits提取该函数签名以执行该函数
        template<typename Fun>
        struct invoker
        {
        //将所有注册的函数都分解为4个部分：函数名f、[成员函数的对象指针pObj]、参数包指针pArgsList、返回值指针pRet
        template<typename Object>
        static inline void apply(Fun& f,Object* pObj,void*pArgsList,void*pRet){
        using tuple_type=typename function_traits<Fun>::bare_tuple_type;
        const tuple_type*pTup=static_cast<tuple_type*>(pArgsList);
        //用C++17合并用于非成员函数、成员函数注册的两个函数
        if constexpr(std::is_member_function_pointer<Fun>::value){
        auto tup=std::tuple_cat(std::tie(pObj),*pTup);
        invoker_call(f,tup,pRet);
                }
                else {
        invoker_call(f,*pTup,pRet);
                }
        }
        template<typename Fun,typename...Args>
        static void invoker_call(const Fun& f,const std::tuple<Args...>& tup,void*pRet)
        {
    //用C++17合并调用无返回值、带返回值的闭包函数
        using return_type=typename function_traits<Fun>::return_type;
        if constexpr(std::is_void<return_type>::value){
        invoker_call_helper(f,std::make_index_sequence<sizeof...(Args)>{},tup);
                }
                else {
        return_type ret=invoker_call_helper(f,std::make_index_sequence<sizeof...(Args)>{},tup);
        if(pRet){*(return_type*)pRet=ret;}
                }
        }
        template<typename Fun,size_t...I,typename...Args>
        static decltype(auto)invoker_call_helper(const Fun& f,const std::index_sequence<I...>&,const std::tuple<Args...>& tup)
        {
        return std::invoke(f,std::get<I>(tup)...);
//C++17的invoke调用器
                }
        };
    };
```

5.2 function_traits<Ret(Args...)>类模板

在泛型函数容器的完整实现代码中，function_traits.h是用于获得函数返回值类型、参数tuple、成员函数指针中对象类型的头文件[16]，其结构见图2中的function_traits<Ret(Args...)>类模板。

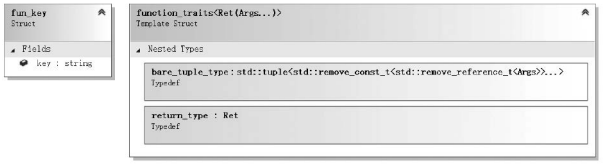

图2 function_traits类模板和fun_key类的结构图

Fig.2 Structure diagram of function_traits class template and fun_key class

6 泛型函数容器的实际使用(Actual use of generic function container)

下面代码测试了泛型函数容器的实际使用。测试可以分为注册函数时输入key和未输入key两大类。每一大类都可以实现无返回值普通函数、带返回值普通函数、函数模板、成员函数、函数对象、lambda表达式、重载函数、某些特殊函数等的注册和调用。

6.1 实际使用的测试代码

```
#include<iostream>
#include "function_container.h"
using namespace std;
void fun1(int a,double b){cout<<a+b<<endl;}
string fun2(int a,unsigned c){return to_string(a+c);}
template<typename T>void fun3(T t){cout<<t<<endl;}
struct Test {void fun4(int a){cout<<a<<endl;} };
struct Fun5{int operator()(int a,int b){return a+b;}};
int main()
{
auto& fc=function_container::get_instance();
//取得泛型函数容器唯一实例
//1、函数注册时不输入key
fc.reg_fun(fun1);
//1.1、无返回值普通函数的注册和调用
fc.call(1,2.5);
//3.5
fc.reg_fun(fun2);
//1.2、带返回值普通函数的注册和调用
cout<<fc.call<string>(1,(unsigned)3)<<endl;
//4
fc.reg_fun(fun3<int>);
//1.3、函数模板的注册和调用
fc.call(9);
//9
Test p2;
fc.reg_fun(&Test::fun4);
//1.4、成员函数的注册和调用。注意：类名和对象名前面必须添加&符号
fc.call(&p2,7);
 //7
fc.reg_fun(&Fun5::operator());
//1.5、函数对象的注册和调用。注意：类名和对象名前面必须添加&符号
cout<<fc.call<int>(&Fun5(),1,7)<<endl;
//8
fc.reg_fun([](int a,char c){return a+c;});
//1.6、lamdba表达式的注册和调用
cout<<fc.call<int>(1,'a')<<endl<<endl;
//98
//2、函数注册时输入key
//注意：注册函数时指定的key为std::string类型，调用该注册函数时必须使用key_fun类型的key
fc.reg_fun(fun1,"a");
//2.1、无返回值普通函数的注册和调用
fc.call(fun_key{"a" },1,2.5);
//3.5
fc.reg_fun(fun2,"b");
//2.2、带返回值函数的注册和调用
cout<<fc.call<string>(fun_key{"b" },1,(unsigned)3)<<endl;
//4
// …
    system("pause");
    return 0;
}
```

6.2 重载函数和某些特殊函数的注册调用测试

前面已经提到，本项目解决了重载函数和某些特殊函数的注册调用问题。具体测试代码如下。

```
#include<iostream>
#include "function_container.h"
using namespace std;
void fun7(int a,double b,int c){cout<<a+b+c<<endl;}
void fun7(int a,int b,int c){cout<<a*b*c<<endl;}
double fun81(double a,double b){return a*b;}
int fun82(double a,double b){return int(a*b);}
int fun83(double a,double b){return int(a-b);}
int main()
{
```

```
    //解决重载函数和某些特殊函数的注册调用问题
    auto& fc=function_container::get_instance();
    //取得泛型函数容器唯一实例
    void(*fun7_idi)(int,double,int)=fun7;
    //重载函数的注册和调用测试
    auto fun7_iii=[](int a,int b,int c){return fun7(a,b,c);};
    fc.reg_fun(fun7_idi); fc.call(1,2.5,4);
    //7.5
    fc.reg_fun(fun7_iii); fc.call(1,2,4);
    //8
    fc.reg_fun(fun81);fc.reg_fun(fun82);
    //特殊函数1的注册和调用测试
    cout<<fc.call<double>(1.3,5.8)<<endl;
    //7.54
    cout<<fc.call<int>(1.3,5.8)<<endl;
    //7
    auto fun83_2=[](double a,double b,int c){return fun83(a,b);};
    //特殊函数2的注册和调用测试
    fc.reg_fun(fun83_2);
    cout<<fc.call<int>(1.3,5.8,0)<<endl;
    //-4
        system("pause");
        return 0;
    }
```

7 结论(Conclusion)

综上所述，泛型函数容器可以将任意类型的函数用一个key进行注册以供其他程序调用，它的使用可以解耦对象之间的调用关系，有利于实现高内聚、低耦合的软件设计原则。C++标准库中并没有这样的容器，用C++旧标准实现也很困难、很低效。C++1x等新标准发布后，出现了一些更好的实现方式。本文便是在已有设计的基础之上，基于C++17新标准，利用if constexpr、fold expression、std::invoke等新技术，提供一种泛型函数容器的实现方式。测试结果表明，该实现方式简洁高效地实现了任意类型函数的注册和调用，并且解决了重载函数和某些特殊函数的注册调用问题，可以显著降低耦合性、提高代码复用性。

参考文献(References)

[1] Ofenbeck G,Rompf T,Püschel M.Staging for generic programming in space and time[C].The ACM SIGPLAN International Conference.ACM,2017:15-28.

[2] Bemardi ML,Cimitile M,Lucca GD.Design pattem detection using a DSL-driven graph matching approach[J].Journal of Software Evolution&Process,2014,26(12):1233-1266.

[3] B Rasool G,Mader P.A customizable approach to design pattems recognition based 011 feature types[J].Arabian Journal for Science&Engineering,2014,39(12):8851-8873.

[4] Chen Yewang,Jiang Zhixiong,Zhao Wenyun,et al.Generic component:a generic programming approach[EB/OL].https://www.computer.org/csdl/proceedings/cit/2007/2983/00/29830087-abs.html,2018 IEEE.

[5] Yallop J.Staging,generic programming[M].New York:ACM,2016:85-96.

[6] 符号修饰(name decoration)与函数签名(function signature)[EB/OL].https://blog.csdn.net/weiwangchao_/article/details/7165467,2011-12-30.

[7] std::function[EB/OL].https://en.cppreference.com/w/cpp/utility/functional/function,2018-06-15.

[8] Bjarne Stroustrup.The C++ Programming Language Fourth Edition[M].USA:Addison-Wesley Professional,2013:967.

[9] std::placeholders[EB/OL].https://en.cppreference.com/w/cpp/utility/functional/placeholders,2018-06-15.

[10] Stanley B,Lippman.C++ Primer 5th Edition[M].USA:Addison-Wesley Professional,2012:572-574.

[11] if statement,attr(optional)if constexpr(optional)(init-statement(optional)condition)statement-true else statement-false[EB/OL].https://en.cppreference.com/w/cpp/language/if,2018-08-21.

[12] Fold expression(since C++17)[EB/OL].https://en.cppreference.com/w/cpp/language/fold,2018-07-19.

[13] std::invoke[EB/OL].https://en.cppreference.com/w/cpp/utility/functional/invoke,2018-07-06.

[14] C++ conformance improvements in Visual Studio 2017 versions[EB/OL].https://docs.microsoft.com/en-us/cpp/cpp-conformance-improvements-2017?view=vs-2017,2018-08-15.

[15] C++ Standards Support in GCC[EB/OL].https://gcc.gnu.org/projects/cxx-status.html,2018-09-30.

[16] 获得函数返回值类型、参数tuple、成员函数指针中的对象类型[EB/OL].https://www.cnblogs.com/ybmj/p/9651227.html,2018-09-15.

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.

罗 泓(1970-)，女，大专，工程师.研究领域：数据分析与处理，电路设计，信息管理系统.