

## jerry\_fuyi

### 剖析STD::FUNCTION接口与实现

`<functional>` 系列

## 目录

### 前言

#### 一、std::function的原理与接口

[1.1 std::function是函数包装器](#)

[1.2 C++注重运行时效率](#)

[1.3 用函数指针实现多态](#)

[1.4 std::function的接口](#)

#### 二、std::function的实现

##### 2.1 类型系统

[2.1.1 异常类](#)

[2.1.2 数据存储](#)

[2.1.3 辅助类](#)

[2.1.4 内存管理基类](#)

[2.1.5 仿函数调用](#)

[2.1.6 接口定义](#)

### 导航

[博客园](#)

[首页](#)

[联系](#)

[订阅](#) RSS

[管理](#)

### 统计

随笔 - 95

文章 - 0

评论 - 43

阅读 - 12万

### 公告

昵称: [jerry\\_fuyi](#)

园龄: [2年6个月](#)

粉丝: [60](#)

关注: [0](#)

[+加关注](#)

### 搜索

找找看

谷歌搜索

### [2.1.7 类型关系](#)

## [2.2 方法的功能与实现](#)

### [2.2.1 多态性的体现](#)

### [2.2.2 本地函数对象](#)

### [2.2.3 heap函数对象](#)

### [2.2.4 两种存储结构如何统一](#)

### [2.2.5 根据形式区分仿函数类型](#)

### [2.2.6 实现组装成接口](#)

[后记](#)

[附录](#)

## 前言

为什么要剖析 `std::function` 呢？因为笔者最近在做一个 `std::function` 向单片机系统的移植与扩展。

后续还会有 `std::bind` 等标准库其他部分的移植。

## 一、std::function的原理与接口

### 1.1 std::function是函数包装器

`std::function`，能存储任何符合模板参数的函数对象。换句话说，这些拥有一致参数类型、相同返回值类型（其实不必完全相同）的函数对象，可以由 `std::function` 统一包装起来。函数对象的大小

## 常用链接

[我的随笔](#)

[我的评论](#)

[我的参与](#)

[最新评论](#)

[我的标签](#)

## 最新随笔

[1.AVR单片机教程——第三期导语](#)

[2.ATtiny3217 x WS2812B梦幻联动](#)

[3.C++20初体验——concepts](#)

[4.缝合怪的电赛纪实](#)

[5.鼠标修复升级记录（下）](#)

[6.鼠标修复升级记录（上）](#)

[7.自制蓝牙音箱的手册](#)

[8.运气一直好，就不只是运气了——记中学七年](#)

[9.摇摇棒，理工男的择偶权（上）](#)

[10.C++值元编程](#)

## 随笔分类 (116)

[AVR\(49\)](#)

[C++\(30\)](#)

[C++20\(5\)](#)

[MEDS\(4\)](#)

[STM32\(6\)](#)

[数据结构与算法\(20\)](#)

是任意的、不能确定的，而C++中的类型都是固定大小的，那么，如何在一个固定大小的类型中存储任意大小的对象呢？

实际上问题还不止存储那么简单。存储了函数对象，必定是要在某一时刻调用；函数对象不是在创建的时候调用，这个特性成为延迟调用；函数对象也是对象，需要处理好构造、拷贝、移动、析构等问题——这些也需要延迟调用，但总不能再使用 `std::function` 来解决吧？

既然 `std::function` 能存储不同类型的函数对象，可以说它具有多态性。C++中体现多态性的主要是虚函数，继承与多态这一套体制是可以解决这个问题的。[相关资料\[1\] \[2\]](#)中的实现利用了继承与多态，相当简洁。

## 1.2 C++注重运行时效率

利用继承与多态，我们可以让编译器帮我们搞定函数对象的析构。就这种实现而言，这是简洁而有效的方法。然而这种实现需要动态内存，在一些情况下不划算，甚至完全没有必要。C++11引入了lambda表达式，其本质也是函数对象。这个对象有多大呢？取决于捕获列表。你写lambda会捕获多少东西？很多情况下就只是一对方括号而已吧。在这种情况下，lambda表达式的对象大小只有1字节（因为不能是0字节），你却为了这没有内容的1字节要调用动态内存的函数？C++注重运行时效率，这种浪费是不能接受的。

如何避免这种浪费呢？你也许会说检查传入的对象是不是1字节的空类型。且不论这个trait怎么实现，函数指针、捕获一个int的lambda等类型都声称自己是trivial的小对象，也不应该分配到heap中去。

之前说过，`std::function` 的大小是固定的，但是这个大小是可以自己定的。我们可以 在 `std::function` 的类定义中加入一个空白的、大小适中的field，用在存放这些小对象，从而避免这些情况下的动态内存操作。同时，既然有了这片空间，也就不需要看传入的对象是不是1字节的空类型了。

而对于更大的类型，虽然这个field不足以存放函数对象，但足以存放一个指针，这种分时复用的结构可以用union来实现。这种小对象直接存储、大对象在heap上开辟空间并存储指针的方法，称为small object optimization。

在利用继承的实现中，函数对象被包装在一个子类中，`std::function` 中持有一个其父类的指针。然而为了效率，我们需要把空白field和这个指针union起来。union总给人一种底层的感觉，在不确定这个

[维修\(2\)](#)

## 随笔档案 (95)

[2021年2月\(1\)](#)

[2021年1月\(1\)](#)

[2020年12月\(1\)](#)

[2020年10月\(1\)](#)

[2020年9月\(3\)](#)

[2020年8月\(1\)](#)

[2020年6月\(3\)](#)

[2020年5月\(7\)](#)

[2020年4月\(11\)](#)

[2020年3月\(1\)](#)

[2020年2月\(3\)](#)

[2020年1月\(10\)](#)

[2019年12月\(7\)](#)

[2019年11月\(1\)](#)

[2019年10月\(7\)](#)

[更多](#)

## 阅读排行榜

[1. C++类成员默认初始值\(13919\)](#)

[2. 剖析std::function接口与实现\(11390\)](#)

[3. C++11<functional>深度剖析\(6465\)](#)

[4. AVR单片机教程——开发环境配置\(4218\)](#)

union到底存储的是什么时候，当然不能通过其中的指针去调用虚函数。在这样的设计中，多态性不再能用继承体系实现了，我们需要另一种实现多态的方法。

### 1.3 用函数指针实现多态

回想一下虚函数是如何实现的？带有virtual function的类的对象中会安插vptr，这个指针指向一个vtable，这个vtable含有多个slot，里面含有指向type\_info对象的指针与函数指针——对，我们需要函数指针！不知你有没有在C中实现过多态，在没有语言特性的帮助下，比较方便的方法是在struct中直接放函数指针。如果要像C++那样用上vptr和vtable，你得管理好每个类及其对应vtable的内容。你以为这种情况在C++中就有所好转吗？只有你用C++的继承体系，编译器才会帮你做这些事。想要自己建立一个从类型到vptr的映射，恐怕你得改编译器了。（更正：C++14引入了变量模板，请移步[C++值多态：传统多态与类型擦除之间](#)。）

vptr与vtable的意义是什么？其一，每个基类只对应一个vptr，大小固定，多重继承下便于管理，但这点与这篇文章的主题没有关联；其二，当基类有多个虚函数的时候，使用vptr可以节省存储对象的空间，而如果用函数指针的话，虽然少了一次寻址，但继承带来的空间overhead取决于虚函数的数量，由于至少一个，函数指针的占用的空间不会少于vptr，在虚函数数量较多的情况下，函数指针就要占用比较大的空间了。

既然我们已经无法在 `std::function` 中使用vptr，我们也应该尽可能减少函数指针的数量，而这又取决于这些函数的功能，进一步取决于 `std::function` 类的接口。

### 1.4 std::function的接口

虽然C++标准规定了 `std::function` 的接口就应该是[这样](#)，我还是想说说它为什么应该是这样。关于其他的一些问题，比如保存值还是保存引用等，可以参考[相关资料](#)[4]。

最基本的，`std::function` 是一个模板类，模板参数是一个类型（注意是一个类型，不是好几个类型）。我们可以这么写：

```
std::function<int(double)> f;
```

[5. 运气一直好，就不只是运气了——记中学七年\(3706\)](#)

#### 推荐排行榜

- [1. 做个别出心裁的圣诞礼物\(8\)](#)
- [2. 运气一直好，就不只是运气了——记中学七年\(7\)](#)
- [3. 测量C++程序运行时间\(7\)](#)
- [4. C++17结构化绑定\(5\)](#)
- [5. 剖析std::function接口与实现\(5\)](#)

#### 最新评论

[1. Re:AVR单片机教程——数字IO寄存器](#)

很棒，确实是深入浅出，作为一个从STM32到DSP到51的路线，我感觉很贴切，加油

--小白菜大萝卜

[2. Re:AVR单片机教程——开发板介绍](#)

太厉害了

--小白菜大萝卜

[3. Re:AVR单片机教程——EasyElectronics Library v1.3手册](#)

你好，库函数在哪里下载。好像链接不行

--豪杰2021

[4. Re:AVR单片机教程——EasyElectronics Library v2.0手册](#)

`f` 是一个可调用对象，参数为 `double`，返回值为 `int`。你也许会问，这里既规定了参数类型又规定了返回值类型，怎么就成了一个类型呢？确实是一个类型，`int(double)` 是一个函数类型（注意不是函数指针）。

`std::function` 要包装所有合适类型的对象，就必须有对应的构造函数，所以这是个模板构造函数。参数不是通用引用而是直接传值：

```
template <typename F>
function(F);
```

可能是为了让编译器对空对象进行优化。同样还有一个模板赋值函数，参数是通用引用。

每个构造函数都有一个添加了 `std::allocator_arg_t` 作为第一个参数、内存分配器对象作为第二个参数的版本，C++17中已经移除（GCC从未提供，可能是因为 `std::function` 的内存分配无法自定义）。同样删除的还有 `assign`，也是与内存分配器相关的。

另外有一个以 `std::reference_wrapper` 作为参数的赋值函数：

```
template <typename F>
function& operator=(std::reference_wrapper<F>) noexcept;
```

可以理解为模板赋值函数的特化。没有相应的构造函数。

默认构造函数、`nullptr_t` 构造函数、`nullptr_t` 拷贝赋值函数都将 `std::function` 对象置空。当 `std::function` 对象没有保存任何函数对象时，`operator bool()` 返回 `false`，与 `nullptr_t` 调用 `operator==` 会返回 `true`，如果调用将抛出 `std::bad_function_call` 异常。

虽然 `std::function` 将函数对象包装了起来，但用户还是可以获得原始对象的。

`target_type()` 返回函数对象的 `typeid`，`target()` 模板函数当模板参数与函数对象类型相同时返回其指针，否则返回空指针。

作为函数包装器，`std::function` 也是函数对象，可以通过 `operator()` 调用，参数按照模板参数中声明的类型传递。

因为公司需要，开始学习下 AVR，系列博文对我帮助非常大，非常感谢博主

--huyang27

5. Re:模板参数的“右值引用”是转发引用

受益匪浅

--Automata

还有一些接口与大部分STL设施相似，有Rule of Five规定的5个方法、`swap()`，以及 `std::swap()` 的特化等。可别小看这个 `swap()`，它有大用处。

总之，函数对象的复制、移动、赋值、交换等操作都是需要的。对客户来说，除了两个 `std::function` 的相等性判定（笔者最近在尝试实现这个）以外，其他能想到的方法它都有。

## 二、std::function的实现

`std::function` 的实现位于 `<functional>`，后续版本迁移至了 `<bits/std_function.h>`。下面这段代码是GCC 4.8.1（第一个支持完整C++11的版本）中的 `<functional>` 头文件，共2579行，默认折叠，慎入。

⊕ `functional`

这个实现的原理与上面分析的大致相同，使用函数指针实现多态，也使用了small object optimization。

注：标准库的文件的缩进是2格，有时8个空格会用一个tab代替，在将tab显示为4字节的编辑器中缩进会比较乱，我已经把tab全部替换为8个空格；很多人缩进习惯是4格，但如果把2格全部替换成4格也会乱了格式，所以以下摘录自标准库文件的代码全部都是2格缩进。

### 2.1 类型系统

类型之间的关系，无非是继承、嵌套、组合。这个实现中三者都有。

关于继承，你也许会问，我们刚才不是说了这种实现没法用继承吗？实际上没有矛盾。刚才说的继承，是接口上的继承，讲得更具体点就是要继承虚函数，是一种is-a的关系；而这里的继承，是实现上的继承，是一种is-implemented-in-terms-of的关系，在语言层面大多是private继承。

在泛型编程中，还有一个关于继承的问题，就是在继承体系的哪一层引入模板参数。

嵌套，即类中定义嵌套类型，使类之间的结构更加清晰，在泛型编程中还可以简化设计。

组合，在于一个类的对象中包含其他类的对象，本应属于对象关系的范畴，但是在这个实现中，一个类一般不会在同一个scope内出现多个对象，因此我这里就直接把对象组合的概念拿来用了。

### 2.1.1 异常类

首先出现的是 `bad_function_call` 类型，这是一个异常类，当调用空 `std::function` 对象时抛出：



```
1 class bad_function_call : public std::exception
2 {
3 public:
4     virtual ~bad_function_call() noexcept;
5     const char* what() const noexcept;
6 };
```



由于不是模板类（难得能在STL中发现非模板类），实现被编译好放在了目标文件中。虽然GCC开源，但既然这个类不太重要，而且稍微想想就能知道它是怎么实现的了，所以这里就不深究了。

相关的还有一个用于抛出异常的函数：

```
1 void __throw_bad_function_call() __attribute__((__noreturn__));
```

在 `<bits/funcexcept.h>` 中。同样只有声明没有定义。

### 2.1.2 数据存储

有关数据存储的类共有3个：



```
1 class _Undefined_class;
2
3 union _Nocopy_types
4 {
```

```

5  void*      _M_object;
6  const void* _M_const_object;
7  void (*_M_function_pointer)();
8  void (_Undefined_class::*_M_member_pointer)();
9  };
10
11 union _Any_data
12 {
13     void*      _M_access()      { return &_M_pod_data[0]; }
14     const void* _M_access() const { return &_M_pod_data[0]; }
15
16     template<typename _Tp>
17         _Tp&
18         _M_access()
19         { return *static_cast<_Tp*>(_M_access()); }
20
21     template<typename _Tp>
22         const _Tp&
23         _M_access() const
24         { return *static_cast<const _Tp*>(_M_access()); }
25
26     _Nocopy_types _M_unused;
27     char _M_pod_data[sizeof(_Nocopy_types)];
28 };

```



`_Undefined_class`，顾名思义，连定义都没有，只是用于声明 `_Nocopy_types` 中的成员指针数据域，因为同一个平台上成员指针的大小是相同的。

`_Nocopy_types`，是4种类型的联合体类型，分别为指针、常量指针、函数指针与成员指针。

“nocopy”指的是这几种类型指向的对象类型，而不是本身。



`_Any_data`，是两种类型的联合体类型，一个是 `_Nocopy_types`，另一个是 `char` 数组，两者大小相等。后者是POD的，POD的好处多啊，memcpy可以用，最重要的是复制不会抛异常。非模板 `_M_access()` 返回指针，模板 `_M_access()` 返回解引用的结果，两者都有 `const` 重载。

### 2.1.3 辅助类



```
1 enum _Manager_operation
2 {
3     __get_type_info,
4     __get_functor_ptr,
5     __clone_functor,
6     __destroy_functor
7 };
```



`_Manager_operation`，枚举类，是前面所说控制 `std::function` 的函数指针需要的参数类型。定义了4种操作：获得 `type_info`、获得仿函数（就是函数对象）指针、复制仿函数、销毁（析构）仿函数。从这个定义中可以看出，1.4节所说的各种功能中，需要延迟调用的，除了函数对象调用以外，都可以通过这4个功能来组合起来。我们后面还会进一步探讨这个问题。

```
1 template<typename _Tp>
2     struct __is_location_invariant
3     : integral_constant<bool, (is_pointer<_Tp>::value
4                               || is_member_pointer<_Tp>::value)>
5     { };
```

`__is_location_invariant`，一个trait类，判断一个类型是不是“位置不变”的。从字面上来理解，一个类型如果是“位置不变”的，那么对于一个这种类型的对象，无论它复制到哪里，各个对象的底层表示都是相同的。在这个定义中，一个类型是“位置不变”的，当且仅当它是一个指针或成员指针，与一般的理解有所不同（更新：后来改为 `template<typename _Tp> struct`

`__is_location_invariant : is_trivially_copyable<Tp>::type { };` , 这就比较容易理解了)。



```
1 template<typename _Tp>
2     struct _Simple_type_wrapper
3     {
4         _Simple_type_wrapper(_Tp __value) : __value(__value) { }
5
6         _Tp __value;
7     };
8
9 template<typename _Tp>
10     struct __is_location_invariant<_Simple_type_wrapper<_Tp> >
11     : __is_location_invariant<_Tp>
12     { };
```



`_Simple_type_wrapper` , 一个简单的包装器, 用于避免 `void*` 与指针的指针之间类型转换的 `const` 问题。以及 `__is_location_invariant` 对 `_Simple_type_wrapper` 的偏特化。

## 2.1.4 内存管理基类

类 `_Function_base` 定义了一系列用于管理函数对象内存的函数, 这是一个非模板类:



```
1 class _Function_base
2 {
3 public:
4     static const std::size_t _M_max_size = sizeof(_Nocopy_types);
5     static const std::size_t _M_max_align = __alignof__(_Nocopy_types);
6
```

```

7  template<typename _Functor>
8      class _Base_manager;
9
10 template<typename _Functor>
11     class _Ref_manager;
12
13 _Function_base() : _M_manager(0) { }
14
15 ~_Function_base()
16 {
17     if (_M_manager)
18         _M_manager(_M_functor, _M_functor, __destroy_functor);
19 }
20
21 bool _M_empty() const { return !_M_manager; }
22
23 typedef bool (*_Manager_type) (_Any_data&, const _Any_data&,
24                                _Manager_operation);
25
26 _Any_data      _M_functor;
27 _Manager_type  _M_manager;
28 };

```



`_Function_base` 是 `std::function` 的实现基类，定义了两个静态常量，用于后面的trait类；两个内部类，用于包装静态方法；函数指针类型 `_Manager_type` 的对象 `_M_manager`，用于存取 `_Any_data` 类型的 `_M_functor` 中的数据；构造函数，将函数指针置为空；析构函数，调用函数指针，销毁函数对象；`_M_empty()` 方法，检测内部是否存有函数对象。

我们来看其中的 `_Base_manager` 类：



```

1  template<typename _Functor>
2      class _Base_manager
3      {
4      protected:
5          static const bool __stored_locally =
6          (__is_location_invariant<_Functor>::value
7           && sizeof(_Functor) <= _M_max_size
8           && __alignof__(_Functor) <= _M_max_align
9           && (_M_max_align % __alignof__(_Functor) == 0));
10
11         typedef integral_constant<bool, __stored_locally> _Local_storage;
12
13         static _Functor*
14         _M_get_pointer(const _Any_data& __source);
15
16         static void
17         _M_clone(_Any_data& __dest, const _Any_data& __source, true_type);
18
19         static void
20         _M_clone(_Any_data& __dest, const _Any_data& __source, false_type);
21
22         static void
23         _M_destroy(_Any_data& __victim, true_type);
24
25         static void
26         _M_destroy(_Any_data& __victim, false_type);
27
28     public:
29         static bool
30         _M_manager(_Any_data& __dest, const _Any_data& __source,
31                    _Manager_operation __op);
32

```

```

33     static void
34     _M_init_functor(_Any_data& __functor, _Functor&& __f);
35
36     template<typename _Signature>
37         static bool
38         _M_not_empty_function(const function<_Signature>& __f);
39
40     template<typename _Tp>
41         static bool
42         _M_not_empty_function(const _Tp*& __fp);
43
44     template<typename _Class, typename _Tp>
45         static bool
46         _M_not_empty_function(_Tp _Class::* __mp);
47
48     template<typename _Tp>
49         static bool
50         _M_not_empty_function(const _Tp&);
51
52 private:
53     static void
54     _M_init_functor(_Any_data& __functor, _Functor&& __f, true_type);
55
56     static void
57     _M_init_functor(_Any_data& __functor, _Functor&& __f, false_type);
58 };

```



定义了一个静态布尔常量 `__stored_locally`，它为真当且仅

当 `__is_location_invariant` trait 为真、仿函数放得下、仿函数的 align 符合两个要求。然后再反过来根据这个值定义 trait 类 `_Local_storage`（标准库里一般都是根据 value trait 来生成 value）。

其余几个静态方法，顾名思义即可。有个值得思考的问题，就是为什么 `_M_init_functor` 是public的，没有被放进 `_M_manager` 呢？

再来看 `_Ref_manager` 类：



```
1 template<typename _Functor>
2 class _Ref_manager : public _Base_manager<_Functor*>
3 {
4     typedef _Function_base::_Base_manager<_Functor*> _Base;
5
6 public:
7     static bool
8     _M_manager(_Any_data& __dest, const _Any_data& __source,
9               _Manager_operation __op);
10
11     static void
12     _M_init_functor(_Any_data& __functor, reference_wrapper<_Functor> __f
13 };
```



`_Ref_manager` 继承自 `_Base_manager` 类，覆写了两个静态方法。

## 2.1.5 仿函数调用

起辅助作用的模板函数 `__callable_functor`：



```
1 template<typename _Functor>
2 inline _Functor&
3 __callable_functor(_Functor& __f)
4 { return __f; }
```

```

5
6 template<typename _Member, typename _Class>
7     inline _Mem_fn<_Member _Class::*>
8     __callable_functor(_Member _Class::* &__p)
9     { return std::mem_fn(__p); }
10
11 template<typename _Member, typename _Class>
12     inline _Mem_fn<_Member _Class::*>
13     __callable_functor(_Member _Class::* const &__p)
14     { return std::mem_fn(__p); }
15
16 template<typename _Member, typename _Class>
17     inline _Mem_fn<_Member _Class::*>
18     __callable_functor(_Member _Class::* volatile &__p)
19     { return std::mem_fn(__p); }
20
21 template<typename _Member, typename _Class>
22     inline _Mem_fn<_Member _Class::*>
23     __callable_functor(_Member _Class::* const volatile &__p)
24     { return std::mem_fn(__p); }

```



对非成员指针类型，直接返回参数本身；对成员指针类型，返回 `mem_fn()` 的结果（将类对象转换为第一个参数；这个标准库函数的实现不在这篇文章中涉及），并有cv-qualified重载。它改变了调用的形式，把所有的参数都放在了小括号中。

`_Function_handler` 类，管理仿函数调用：



```

1 template<typename _Signature, typename _Functor>
2     class _Function_handler;
3
4 template<typename Res, typename Functor, typename... ArgTypes>

```

```

5  class _Function_handler<_Res(_ArgTypes...), _Functor>
6  : public _Function_base::_Base_manager<_Functor>
7  {
8      typedef _Function_base::_Base_manager<_Functor> _Base;
9
10 public:
11     static _Res
12     _M_invoke(const _Any_data& __functor, _ArgTypes... __args);
13 };
14
15 template<typename _Functor, typename... _ArgTypes>
16 class _Function_handler<void(_ArgTypes...), _Functor>
17 : public _Function_base::_Base_manager<_Functor>
18 {
19     typedef _Function_base::_Base_manager<_Functor> _Base;
20
21 public:
22     static void
23     _M_invoke(const _Any_data& __functor, _ArgTypes... __args);
24 };
25
26 template<typename _Res, typename _Functor, typename... _ArgTypes>
27 class _Function_handler<_Res(_ArgTypes...), reference_wrapper<_Functor>
28 : public _Function_base::_Ref_manager<_Functor>
29 {
30     typedef _Function_base::_Ref_manager<_Functor> _Base;
31
32 public:
33     static _Res
34     _M_invoke(const _Any_data& __functor, _ArgTypes... __args);
35 };
36

```



```

37 template<typename _Functor, typename... _ArgTypes>
38     class _Function_handler<void(_ArgTypes...), reference_wrapper<_Functor>
39     : public _Function_base::_Ref_manager<_Functor>
40     {
41         typedef _Function_base::_Ref_manager<_Functor> _Base;
42
43     public:
44         static void
45         _M_invoke(const _Any_data& __functor, _ArgTypes... __args);
46     };
47
48 template<typename _Class, typename _Member, typename _Res,
49         typename... _ArgTypes>
50     class _Function_handler<_Res(_ArgTypes...), _Member _Class::*>
51     : public _Function_handler<void(_ArgTypes...), _Member _Class::*>
52     {
53         typedef _Function_handler<void(_ArgTypes...), _Member _Class::*>
54         _Base;
55
56     public:
57         static _Res
58         _M_invoke(const _Any_data& __functor, _ArgTypes... __args);
59     };
60
61 template<typename _Class, typename _Member, typename... _ArgTypes>
62     class _Function_handler<void(_ArgTypes...), _Member _Class::*>
63     : public _Function_base::_Base_manager<
64         _Simple_type_wrapper< _Member _Class::* > >
65     {
66         typedef _Member _Class::* _Functor;
67
68         typedef _Simple_type_wrapper<_Functor> _Wrapper;
69         typedef _Function_base::_Base_manager<_Wrapper> _Base;

```

```

69
70 public:
71     static bool
72     _M_manager(_Any_data& __dest, const _Any_data& __source,
73               _Manager_operation __op);
74
75     static void
76     _M_invoke(const _Any_data& __functor, _ArgTypes... __args);
77 };

```

共有6个特化版本：返回值类型为 `void`、其他；函数对象类型为 `std::reference_wrapper`、成员指针、其他。

继承自 `_Function_base::_Base_manager` 或 `_Function_base::_Ref_manager`，提供了静态方法 `_M_invoke()`，用于仿函数调用。有一个覆写的 `_M_manager()`，表面上看是一个偏特化有覆写，实际上是两个，因为返回非 `void` 的成员指针偏特化版本还继承了其对应 `void` 偏特化版本。

## 2.1.6 接口定义

终于回到伟大的 `std::function` 了，但是我们还得更看点别的：

```

1 template<typename _Arg, typename _Result>
2 struct unary_function
3 {
4     typedef _Arg      argument_type;
5
6     typedef _Result    result_type;
7 };
8
9 template<typename _Arg1, typename _Arg2, typename _Result>

```

```

10  struct binary_function
11  {
12      typedef _Arg1      first_argument_type;
13
14      typedef _Arg2      second_argument_type;
15
16      typedef _Result     result_type;
17  };

```



`std::unary_function` 与 `std::binary_function`，定义了一元和二元函数的参数类型与返回值类型。



```

1  template<typename _Res, typename... _ArgTypes>
2      struct _Maybe_unary_or_binary_function { };
3
4  template<typename _Res, typename _T1>
5      struct _Maybe_unary_or_binary_function<_Res, _T1>
6      : std::unary_function<_T1, _Res> { };
7
8  template<typename _Res, typename _T1, typename _T2>
9      struct _Maybe_unary_or_binary_function<_Res, _T1, _T2>
10     : std::binary_function<_T1, _T2, _Res> { };

```



`_Maybe_unary_or_binary_function` 类，当模板参数表示的函数为一元或二元时，分别继承 `std::unary_function` 与 `std::binary_function`。

现在可以给出 `std::function` 类定义与方法声明：



```

1 template<typename _Signature>
2     class function;
3
4 template<typename _Res, typename... _ArgTypes>
5     class function<_Res(_ArgTypes...)>
6     : public _Maybe_unary_or_binary_function<_Res, _ArgTypes...>,
7       private _Function_base
8     {
9         typedef _Res _Signature_type(_ArgTypes...);
10
11         template<typename _Functor>
12             using _Invoke = decltype(__callable_functor(std::declval<_Functor>(),
13                                                         (std::declval<_ArgTypes>()...) ));
14
15         template<typename _CallRes, typename _Res1>
16             struct _CheckResult
17                 : is_convertible<_CallRes, _Res1> { };
18
19         template<typename _CallRes>
20             struct _CheckResult<_CallRes, void>
21                 : true_type { };
22
23         template<typename _Functor>
24             using _Callable = _CheckResult<_Invoke<_Functor>, _Res>;
25
26         template<typename _Cond, typename _Tp>
27             using _Requires = typename enable_if<_Cond::value, _Tp>::type;
28
29     public:
30         typedef _Res result_type;
31
32         function() noexcept;
33

```

```
34     function(nullptr_t) noexcept;
35
36     function(const function& __x);
37
38     function(function&& __x);
39
40     // TODO: needs allocator_arg_t
41
42     template<typename _Functor,
43             typename = _Requires<_Callable<_Functor>, void>>
44     function(_Functor);
45
46     function&
47     operator=(const function& __x);
48
49     function&
50     operator=(function&& __x);
51
52     function&
53     operator=(nullptr_t);
54
55     template<typename _Functor>
56     _Requires<_Callable<_Functor>, function&>
57     operator=(_Functor&& __f);
58
59     template<typename _Functor>
60     function&
61     operator=(reference_wrapper<_Functor> __f) noexcept;
62     void swap(function& __x);
63
64     // TODO: needs allocator_arg_t
65     /*
```

```

66     template<typename _Functor, typename _Alloc>
67         void
68         assign(_Functor&& __f, const _Alloc& __a);
69     */
70
71     explicit operator bool() const noexcept;
72
73     _Res operator()(_ArgTypes... __args) const;
74
75 #ifdef __GXX_RTTI
76     const type_info& target_type() const noexcept;
77
78     template<typename _Functor>          _Functor* target() noexcept;
79
80     template<typename _Functor> const _Functor* target() const noexcept;
81 #endif
82
83 private:
84     typedef _Res (*_Invoker_type)(const _Any_data&, _ArgTypes...);
85     _Invoker_type _M_invoker;
86 };
87
88 template<typename _Res, typename... _Args>
89     inline bool
90     operator==(const function<_Res(_Args...)>& __f, nullptr_t) noexcept;
91
92 template<typename _Res, typename... _Args>
93     inline bool
94     operator==(nullptr_t, const function<_Res(_Args...)>& __f) noexcept;
95
96 template<typename _Res, typename... _Args>
97     inline bool

```

```

98  operator!=(const function<_Res(_Args...)>& __f, nullptr_t) noexcept;
99
100 template<typename _Res, typename... _Args>
101  inline bool
102  operator!=(nullptr_t, const function<_Res(_Args...)>& __f) noexcept;
103
104 template<typename _Res, typename... _Args>
105  inline void
106  swap(function<_Res(_Args...)>& __x, function<_Res(_Args...)>& __y);

```

前面说过，`std::function` 类的模板参数是一个函数类型。一个函数类型也是一个类型；`std::function` 只在模板参数是函数类型时才有意义；因此，有用的 `std::function` 是一个特化的模板，需要一个声明。标准库规定没有特化的声明是没有定义的。

`std::function` 继承自两个类：公有继承模板类 `_Maybe_unary_or_binary_function`，私有继承非模板类 `_Function_base`。

前者是公有继承，但实际上没有继承虚函数，不属于接口继承，而是实现继承，继承的是基类定义的类型别名。因为这些类型别名是面向客户的，所以必须公有继承。这个继承使 `std::function` 在不同数量的模板参数的实例化下定义不同的类型别名。继承是实现这种功能的唯一方法，SFINAE不行。（这是本文第一次出现SFINAE这个词，我默认你看得懂。这是泛型编程中的常用技巧，如果不会请参考[这篇文章](#)或Google。）

后者是私有继承，也属于实现继承，继承了基类的两个数据域与几个静态方法。

`_Signature_type` 是一个类型别名，就是模板参数，是一个函数类型。

`_Invoke` 是一个别名模板，就是仿函数被按参数类型调用的返回类型。如果不能调用，根据SFINAE，S错误不会E，但这个别名只有一个定义，在使用的地方所有S都E了，编译器还是会给E。

`_CheckResult` 是一个trait类，检测第一个模板参数能否转换为第二个。另有第二个参数为 `void` 的偏特化，在类型检测上使返回类型为 `void` 的 `std::function` 对象能支持任何返回值的函数对象。

`_Callable` 也是一个trait类，利用上面两个定义检测仿函数类型与 `std::function` 模板参数是否匹配。

`_Requires` 是一个有趣的别名模板，如果模板参数中第一个value trait为 `true`，则定义为第二个模板参数，否则未定义（是没有，不是 `void`），使用时将交给SFINAE处理。它大致上实现了C++20中 `require` 关键字的功能。实际上concept在2005年就有proposal了，一路从C++0x拖到C++20。我计划在C++20标准正式发布之前写一篇文章完整介绍concept。

`result_type` 是模板参数函数类型的返回值类型，与基类中定义的相同。

在类定义最后的私有段，还定义了一个函数指针类型以及该类型的一个对象，这是第二个函数指针。

其余的各种函数，在1.4节都介绍过了。

### 2.1.7 类型关系

讲了这么多类型，你记住它们之间的关系了吗？我们再来自顶向下地梳理一遍。

一个 `std::function` 对象中包含一个函数指针，它会被初始化为 `_Function_handler` 类中的静态函数的指针。`std::function` 与 `_Function_handler` 类之间，可以认为是组合关系。

`std::function` 继承自 `_Maybe_unary_or_binary_function` 与 `_Function_base`，两者都是实现继承。

`_Function_base` 中有 `_Base_manager` 与 `_Ref_manager` 两个嵌套类型，其中后者还继承了前者，并覆写了几个方法。两个类定义了一系列静态方法，继承只是为了避免代码重复。

`_Function_base` 含有两个数据域，一个是函数指针，`_Function_base` 与两个嵌套类型之间既是嵌套又是组合；另一个是 `_Any_data` 类型对象，`_Function_base` 与 `_Any_data` 之间是组合关系。

而 `_Any_data` 是一个联合体，是两部分相同大小数据的联合，分别是 `char` 数组和 `_Nocopy_types` 类型对象，后者又是4种基本类型的联合。

其余的一些类与函数，都是起辅助作用的。至此，对 `std::function` 定义的分析就结束了。



## 2.2 方法的功能与实现

### 2.2.1 多态性的体现

之前一直讲，`std::function` 是一个多态的函数对象包装器，其中的难点就在于多态。什么是多态？你能看到这里，想必不是初学者，不知道多态是不可能的。Wikipedia对polymorphism的定义是：In programming languages and type theory, polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

可以说，我们要在 `std::function` 中处理好多态，就是要处理好类型。类型当然不能一个个枚举，但可以分类。这里可以分类的有两处：接口类型，即组成模板参数的类型，以及实现类型，即绑定的仿函数的类型。下面，我们就从这两个角度入手，分析 `std::function` 是如何实现的。

### 2.2.2 本地函数对象

先根据仿函数类型分类，可以在 `std::function` 对象内部存储的，无需heap空间的，在这一节讨论。相关的方法有以下3个：



```
1 template<typename _Functor>
2     static void
3     _Function_base::_Base_manager<_Functor>::
4     _M_init_functor(_Any_data& __functor, _Functor&& __f, true_type)
5     { new (__functor._M_access()) _Functor(std::move(__f)); }
6
7 template<typename _Functor>
8     static void
9     _Function_base::_Base_manager<_Functor>::
10    _M_clone(_Any_data& __dest, const _Any_data& __source, true_type)
11    {
12        new (__dest._M_access()) _Functor(__source._M_access<_Functor>());
13    }
14
15 template<typename _Functor>
```

```

16  static void
17  _Function_base::_Base_manager<_Functor>::
18  _M_destroy(_Any_data& __victim, true_type)
19  {
20      __victim._M_access<_Functor>().~_Functor();
21  }

```



`_M_init_functor` 用于初始化对象，在空白区域上用placement `new` 移动构造了函数对象。

`_M_clone` 用于复制对象，在目标的空白区域上用placement `new` 拷贝构造和函数对象。

`_M_destroy` 用于销毁对象，对函数对象显式调用了析构函数。

### 2.2.3 heap函数对象

然后来看函数对象存储在heap上的情况：



```

1  template<typename _Functor>
2  static void
3  _Function_base::_Base_manager<_Functor>::
4  _M_init_functor(_Any_data& __functor, _Functor&& __f, false_type)
5  { __functor._M_access<_Functor*>() = new _Functor(std::move(__f)); }
6
7  template<typename _Functor>
8  static void
9  _Function_base::_Base_manager<_Functor>::
10 _M_clone(_Any_data& __dest, const _Any_data& __source, false_type)
11 {
12     __dest._M_access<_Functor*>() =
13     new _Functor(*__source._M_access<_Functor*>());
14 }
15

```

```

16 template<typename _Functor>
17     static void
18     _Function_base::_Base_manager<_Functor>::
19     _M_destroy(_Any_data& __victim, false_type)
20     {
21         delete __victim._M_access<_Functor*>();
22     }

```



`_M_access<_Functor*>()` 将空白区域解释为仿函数的指针，并返回其引用，实现了这片区域的分时复用。前两个方法都比前一种情况多一层间接，而销毁方法则直接调用了 `delete`。

#### 2.2.4 两种存储结构如何统一

尽管我们不得不分类讨论，但为了方便使用，还需要一个统一的接口。不知你有没有注意到，上面每一个方法都有一个未命名的参数放在最后，在方法中也没有用到。前一种情况，这个参数都是 `true_type` 类型，而后一种都是 `false_type` 类型。这个技巧称为tag dispatching，在调用时根据类型特征确定这个位置的参数类型，从而通过重载决定调用哪一个。



```

1 template<typename _Functor>
2     static void
3     _Function_base::_Base_manager<_Functor>::
4     _M_init_functor(_Any_data& __functor, _Functor&& __f)
5     { _M_init_functor(__functor, std::move(__f), _Local_storage()); }
6
7 template<typename _Functor>
8     static bool
9     _Function_base::_Base_manager<_Functor>::
10    _M_manager(_Any_data& __dest, const _Any_data& __source,
11              _Manager_operation __op)
12    {
13        switch (__op)

```

```

14     {
15     #ifdef __GXX_RTTI
16         case __get_type_info:
17             __dest._M_access<const type_info*>() = &typeid(_Functor);
18             break;
19     #endif
20         case __get_functor_ptr:
21             __dest._M_access<_Functor*>() = _M_get_pointer(__source);
22             break;
23
24         case __clone_functor:
25             _M_clone(__dest, __source, _Local_storage());
26             break;
27
28         case __destroy_functor:
29             _M_destroy(__dest, _Local_storage());
30             break;
31     }
32     return false;
33 }

```



这个版本的 `_M_init_functor()` 只有两个参数，加上第三个参数委托给重载版本处理，这第三个参数是一个 `_Local_storage` 类的对象，它根据 `__stored_locally` 而成，为 `true_type` 与 `false_type`，从而区分两个重载。

`_M_manager()` 方法，同样地，利用tag dispatching把另两组方法统一起来。它通过第三个枚举类型参数来确定需要的操作。

但是，这个方法的返回值是 `bool`，怎么传出 `type_info` 与函数对象指针呢？它们将返回值写入第一个参数所指向的空间中。说起利用参数来传递返回值，我就想起C中的指针、C++中的引用、RVO、Java中的包裹类型、C#中的 `out` 关键字.....这里的处理方法不仅解决了返回值的问题，同时也使各个操作的参数统一起来。

一个值得思考的问题是为什么不把 `_M_init_functor()` 也放到 `_M_manager()` 中去？答案是，调用 `_M_init_functor()` 的地方在 `std::function` 的模板构造或模板赋值函数中，此时是知道仿函数类型的；而其他操作被调用时，主调函数是不知道仿函数类型的，就必须用函数指针存储起来；为了节省空间，就引入一个枚举类 `_Manager_operation`，把几种操作合并到一个函数中。

实际上这一层可以先不统一，就是写两种情况的 `_M_manager`，然后到上一层再统一，但是会增加代码量。

除此以外，还有一种简单的方法将两者统一：



```
1 template<typename _Functor>
2     static _Functor*
3     _Function_base::_Base_manager<_Functor>::
4     _M_get_pointer(const _Any_data& __source)
5     {
6         const _Functor* __ptr =
7             __stored_locally? std::__addressof(__source._M_access<_Functor>())
8                               : __source._M_access<_Functor*>();
9         return const_cast<_Functor*>(__ptr);
10    }
```



三目运算符的条件是一个静态常量，编译器会优化，不浪费程序空间，也不需要运行时判断，效果与前一种方法相同。至于另外两个方法（指函数）为什么不用这种方法（指将两种情况统一的方法），可能是为了可读性吧。

## 2.2.5 根据形式区分仿函数类型

在下面一层解决了不同存储结构的问题后，我们还要考虑几种特殊情况。

`_M_not_empty_function()` 用于判断参数是否非空，而不同类型的判定方法是不同的。这里的解决方案很简单，模板方法重载即可。



```
1 template<typename _Functor>
2     template<typename _Signature>
3         static bool
4             _Function_base::_Base_manager<_Functor>::
5             _M_not_empty_function(const function<_Signature>& __f)
6             { return static_cast<bool>(__f); }
7
8 template<typename _Functor>
9     template<typename _Tp>
10         static bool
11             _Function_base::_Base_manager<_Functor>::
12             _M_not_empty_function(const _Tp*& __fp)
13             { return __fp; }
14
15 template<typename _Functor>
16     template<typename _Class, typename _Tp>
17         static bool
18             _Function_base::_Base_manager<_Functor>::
19             _M_not_empty_function(_Tp _Class::* const& __mp)
20             { return __mp; }
21
22 template<typename _Functor>
23     template<typename _Tp>
24         static bool
25             _Function_base::_Base_manager<_Functor>::
26             _M_not_empty_function(const _Tp&)
27             { return true; }
```



在调用时，普通函数对象、`std::reference_wrapper` 对象与成员指针的调用方法是不同的，也需要分类讨论。



```
1 template<typename _Res, typename _Functor, typename... _ArgTypes>
2     static _Res
3     _Function_handler<_Res(_ArgTypes...), _Functor>::
4     _M_invoke(const _Any_data& __functor, _ArgTypes... __args)
5     {
6         return (*_Base::_M_get_pointer(__functor)) (
7             std::forward<_ArgTypes>(__args)...);
8     }
```



对于普通函数对象，函数调用没什么特殊的。注意自定义 `operator()` 必须是 `const` 的。

对于 `std::reference_wrapper` 对象，由于包装的对象存储为指针，因此存储结构与普通函数对象有所不同，相应地调用也不同。



```
1 template<typename _Functor>
2     static void
3     _Function_base::_Ref_manager<_Functor>::
4     _M_init_functor(_Any_data& __functor, reference_wrapper<_Functor> __f)
5     {
6         _Base::_M_init_functor(__functor, std::__addressof(__f.get()));
7     }
8
9 template<typename _Functor>
10     static bool
11
12     _Function_base::_Ref_manager<_Functor>::
13     M manager( Any data& dest, const Any data& source,
```

```

13         __Manager_operation __op)
14     {
15         switch (__op)
16         {
17 #ifdef __GXX_RTTI
18             case __get_type_info:
19                 __dest._M_access<const type_info*>() = &typeid(_Functor);
20                 break;
21 #endif
22             case __get_functor_ptr:
23                 __dest._M_access<_Functor*>() = *_Base::_M_get_pointer(__source);
24                 return is_const<_Functor>::value;
25                 break;
26
27             default:
28                 _Base::_M_manager(__dest, __source, __op);
29         }
30         return false;
31     }
32
33 template<typename _Res, typename _Functor, typename... _ArgTypes>
34 static _Res
35 _Function_handler<_Res(_ArgTypes...), reference_wrapper<_Functor> >::
36 _M_invoke(const _Any_data& __functor, _ArgTypes... __args)
37 {
38     return __callable_functor(**_Base::_M_get_pointer(__functor))(
39         std::forward<_ArgTypes>(__args)...);
40 }

```





碰到两个星号是不是有点晕？其实只要想，一般情况下存储函数对象的地方现在存储指针，所以要获得原始对象，只需要比一般情况多一次解引用，这样就容易理解了。

对于成员指针，情况又有一点不一样：



```
1 template<typename _Class, typename _Member, typename... _ArgTypes>
2     static bool
3     _Function_handler<void(_ArgTypes...), _Member _Class::*>::
4     _M_manager(_Any_data& __dest, const _Any_data& __source,
5                 _Manager_operation __op)
6     {
7         switch (__op)
8         {
9 #ifdef __GXX_RTTI
10             case __get_type_info:
11                 __dest._M_access<const type_info*>() = &typeid(_Functor);
12                 break;
13 #endif
14             case __get_functor_ptr:
15                 __dest._M_access<_Functor*>() =
16                     &_Base::_M_get_pointer(__source)->__value;
17                 break;
18
19             default:
20                 _Base::_M_manager(__dest, __source, __op);
21         }
22         return false;
23     }
24
25 template<typename _Class, typename _Member, typename _Res,
26         typename... _ArgTypes>
27     static _Res
```

```

28 _Function_handler<_Res(_ArgTypes...), _Member _Class::*>::
29 _M_invoke(const _Any_data& __functor, _ArgTypes... __args)
30 {
31     return std::mem_fn(_Base::_M_get_pointer(__functor)->__value)(
32         std::forward<_ArgTypes>(__args)...);
33 }

```



我一直说“成员指针”，而不是“成员函数指针”，是因为数据成员指针也是可以绑定的，这种情况在 `std::mem_fn()` 中已经处理好了。

`void` 返回类型的偏特化本应接下来讨论，但之前讲过，这个函数被通过继承复用了。实际上，如果把这里的 `void` 改为模板类型，然后交换两个 `_Function_handler` 偏特化的继承关系，效果还是一样的，所以就在这里先讨论了。

最后一个需要区分的类型，是返回值类型，属于接口类型。之前都是非 `void` 版本，下面还有几个 `void` 的偏特化：



```

1 template<typename _Functor, typename... _ArgTypes>
2     static void
3     _Function_handler<void(_ArgTypes...), _Functor>::
4     _M_invoke(const _Any_data& __functor, _ArgTypes... __args)
5     {
6         (*_Base::_M_get_pointer(__functor))(
7             std::forward<_ArgTypes>(__args)...);
8     }
9
10 template<typename _Functor, typename... _ArgTypes>
11     static void
12     _Function_handler<void(_ArgTypes...), reference_wrapper<_Functor> >::
13     _M_invoke(const _Any_data& __functor, _ArgTypes... __args)

```

```

14  {
15      __callable_functor(**_Base::_M_get_pointer(__functor)) (
16          std::forward<_ArgTypes>(__args)...);
17  }
18
19 template<typename _Class, typename _Member, typename... _ArgTypes>
20 static void
21 _Function_handler<void(_ArgTypes...), _Member _Class::*>::
22 _M_invoke(const _Any_data& __functor, _ArgTypes... __args)
23 {
24     std::mem_fn(_Base::_M_get_pointer(__functor)->__value) (
25         std::forward<_ArgTypes>(__args)...);
26 }

```



`void` 只是删除了 `return` 关键字的非 `void` 版本, 因此 `void` 返回类型的 `std::function` 对象可以绑定任何返回值的函数对象。

## 2.2.6 实现组装成接口

我们终于讨论完了各种情况, 接下来让我们来见证 `std::function` 的大和谐: 如何用这些方法组装成 `std::function`。



```

1 template<typename _Res, typename... _ArgTypes>
2     function<_Res(_ArgTypes...)>::
3     function() noexcept
4     : _Function_base() { }
5
6 template<typename _Res, typename... _ArgTypes>
7     function<_Res(_ArgTypes...)>::
8     function(nullptr_t) noexcept
9     : _Function_base() { }

```

```
10
11 template<typename _Res, typename... _ArgTypes>
12     function<_Res(_ArgTypes...)>::
13     function(function&& __x) : _Function_base()
14     {
15         __x.swap(*this);
16     }
17
18 template<typename _Res, typename... _ArgTypes>
19     auto
20     function<_Res(_ArgTypes...)>::
21     operator=(const function& __x)
22     -> function&
23     {
24         function(__x).swap(*this);
25         return *this;
26     }
27
28 template<typename _Res, typename... _ArgTypes>
29     auto
30     function<_Res(_ArgTypes...)>::
31     operator=(function&& __x)
32     -> function&
33     {
34         function(std::move(__x)).swap(*this);
35         return *this;
36     }
37
38 template<typename _Res, typename... _ArgTypes>
39     auto
40     function<_Res(_ArgTypes...)>::
41     operator=(nullptr_t)
```

```
42  -> function&
43  {
44      if (_M_manager)
45      {
46          _M_manager(_M_functor, _M_functor, __destroy_functor);
47          _M_manager = 0;
48          _M_invoker = 0;
49      }
50      return *this;
51  }
52
53  template<typename _Functor>
54      auto
55      function<_Res(_ArgTypes...)>::
56      operator=(_Functor&& __f)
57      -> _Requires<_Callable<_Functor>, function&>
58      {
59          function(std::forward<_Functor>(__f)).swap(*this);
60          return *this;
61      }
62
63  template<typename _Res, typename... _ArgTypes>
64      template<typename _Functor>
65          auto
66          function<_Res(_ArgTypes...)>::
67          -> function&
68          operator=(reference_wrapper<_Functor> __f) noexcept
69          {
70              function(__f).swap(*this);
71              return *this;
72          }
73
```

```
74 template<typename _Res, typename... _ArgTypes>
75     void
76     function<_Res(_ArgTypes...)>::
77     swap(function& __x)
78     {
79         std::swap(_M_functor, __x._M_functor);
80         std::swap(_M_manager, __x._M_manager);
81         std::swap(_M_invoker, __x._M_invoker);
82     }
83
84 template<typename _Res, typename... _ArgTypes>
85     function<_Res(_ArgTypes...)>::
86     operator bool() const noexcept
87     { return !_M_empty(); }
88
89 template<typename _Res, typename... _ArgTypes>
90     function<_Res(_ArgTypes...)>::
91     function(const function& __x)
92     : _Function_base()
93     {
94         if (static_cast<bool>(__x))
95         {
96             _M_invoker = __x._M_invoker;
97             _M_manager = __x._M_manager;
98             __x._M_manager(_M_functor, __x._M_functor, __clone_functor);
99         }
100     }
101
102 template<typename _Res, typename... _ArgTypes>
103     template<typename _Functor, typename>
104     function<_Res(_ArgTypes...)>::
105     function(_Functor __f)
```

```

106     : _Function_base()
107     {
108         typedef _Function_handler<_Signature_type, _Functor> _My_handler;
109
110         if (_My_handler::_M_not_empty_function(__f))
111         {
112             _My_handler::_M_init_functor(_M_functor, std::move(__f));
113             _M_invoker = &_My_handler::_M_invoke;
114             _M_manager = &_My_handler::_M_manager;
115         }
116     }
117
118 template<typename _Res, typename... _ArgTypes>
119     _Res
120     function<_Res(_ArgTypes...)>::
121     operator()(_ArgTypes... __args) const
122     {
123         if (_M_empty())
124             __throw_bad_function_call();
125         return _M_invoker(_M_functor, std::forward<_ArgTypes>(__args)...);
126     }
127
128 template<typename _Res, typename... _ArgTypes>
129     const type_info&
130     function<_Res(_ArgTypes...)>::
131     target_type() const noexcept
132     {
133         if (_M_manager)
134         {
135             _Any_data __typeinfo_result;
136             _M_manager(__typeinfo_result, _M_functor, __get_type_info);
137             return *__typeinfo_result._M_access<const type_info*>();

```

```
138     }
139     else
140         return typeid(void);
141 }
142
143 template<typename _Res, typename... _ArgTypes>
144     template<typename _Functor>
145         _Functor*
146         function<_Res(_ArgTypes...)>::
147         target() noexcept
148         {
149             if (typeid(_Functor) == target_type() && _M_manager)
150             {
151                 _Any_data __ptr;
152                 if (_M_manager(__ptr, _M_functor, __get_functor_ptr)
153                     && !is_const<_Functor>::value)
154                     return 0;
155                 else
156                     return __ptr._M_access<_Functor*>();
157             }
158             else
159                 return 0;
160         }
161
162 template<typename _Res, typename... _ArgTypes>
163     template<typename _Functor>
164         const _Functor*
165         function<_Res(_ArgTypes...)>::
166         target() const noexcept
167         {
168             if (typeid(_Functor) == target_type() && _M_manager)
169             {
```



```

170         __Any_data __ptr;
171         __M_manager(__ptr, __M_functor, __get_functor_ptr);
172         return __ptr._M_access<const __Functor*>();
173     }
174     else
175         return 0;
176 }
177
178 template<typename __Res, typename... __Args>
179 inline bool
180 operator==(const function<__Res(__Args...)>& __f, nullptr_t) noexcept
181 { return !static_cast<bool>(__f); }
182
183 template<typename __Res, typename... __Args>
184 inline bool
185 operator==(nullptr_t, const function<__Res(__Args...)>& __f) noexcept
186 { return !static_cast<bool>(__f); }
187
188 template<typename __Res, typename... __Args>
189 inline bool
190 operator!=(const function<__Res(__Args...)>& __f, nullptr_t) noexcept
191 { return static_cast<bool>(__f); }
192
193 template<typename __Res, typename... __Args>
194 inline bool
195 operator!=(nullptr_t, const function<__Res(__Args...)>& __f) noexcept
196 { return static_cast<bool>(__f); }
197
198 template<typename __Res, typename... __Args>
199 inline void
200 swap(function<__Res(__Args...)>& __x, function<__Res(__Args...)>& __y)
201 { __x.swap(__y); }

```



我们从 `swap()` 开始入手。`swap()` 方法只是简单地将三个数据成员交换了一下，这是正确的，因为它们存储的都是POD类型。我认为，这个实现对函数对象存储在本地的条件的限制太过严格，大小合适的可trivial复制的函数对象也应该可以存储在本地。

在 `swap()` 的基础上，拷贝构造、移动构造、拷贝赋值、移动赋值函数很自然地构建起来了，而且只用到了 `swap()` 方法。这种技巧称为[copy-and-swap](#)。这也就解释了为什么 `std::function` 需要那么多延迟调用的操作而表示操作的枚举类只需要定义4种操作。

`swap()` 还可以成为异常安全的基础。由于以上方法只涉及到 `swap()`，而 `swap()` 方法是不抛异常的，因此两个移动函数是 `noexcept` 的，两个拷贝函数也能保证在栈空间足够时不抛异常，在抛异常时不会出现内存泄漏。

其余的方法，有了前面的基础，看代码就能读懂了。

## 后记

写这篇文章花了好久呀。这是我第一次写这么长的博客，希望你能有所收获。如果有不懂的地方，可以在评论区留言。如果有任何错误，烦请指正。

我是从实现的角度来写的这篇文章，如果你对其中的一些技巧（SFINAE、tag dispatching）不太熟悉的话，理解起来可能有点困难。[相关资料\[8\]](#)介绍了 `function` 类的设计思路，从无到有的构建过程比较容易理解。[相关资料\[9\]](#)分析了另一个版本的 `std::function` 实现，可供参考。

文章内容已经很充实了，但是没有图片，不太直观。有空我会加上图片的，这样更容易理解。

另外，在我实现完自己的 `function` 类以后，还会对这篇文章作一点补充。自己造一遍轮子，总会有更深刻的感受吧。

## 附录

相关资料:

- [1] [Naive std::function implementation](#)
- [2] [How is std::function implemented?](#)
- [3] [std::function - cppreference.com](#)
- [4] [The space of design choices for std::function](#)
- [5] [How true is "Want Speed? Pass by value"](#)
- [6] [C++奇淫巧技之SFINAE](#)
- [7] [What is the copy-and-swap idiom?](#)
- [8] [std::function 基本实现](#)
- [9] [std::function 源码分析](#)

分类: [C++](#)

好文要顶

关注我

收藏该文



jerry\_fuyi

关注 - 0

粉丝 - 60

[+加关注](#)

5

推荐

0


反对

« 上一篇: [C++11<functional>深度剖析](#)

» 下一篇: [一个低电平引发的思考](#)

posted on 2019-07-29 12:02 jerry\_fuyi 阅读(11391) 评论(1) 编辑 收藏 举报

[刷新评论](#) [刷新页面](#) [返回顶部](#)

 登录后才能查看或发表评论, 立即 [登录](#) 或者 [逛逛](#) 博客园首页

[【推荐】百度智能云 2022 新春嘉年华：云上迎新春，开心过大年](#)

[【推荐】园子的不务正业：开始做华为云代理业务，期待您的支持](#)

#### 编辑推荐：

- [记一次 .NET 某智能交通后台服务 CPU爆高分析](#)
- [从0到1用故事讲解「动态代理」](#)
- [如何写好一篇技术型文档？](#)
- [妙用滤镜构建高级感拉满的磨砂玻璃渐变背景](#)
- [公司内部一次关于kafka消息队列消费积压故障复盘分享](#)



百度智能云 企业级云服务器305元

立即购买

#### 最新新闻：

- [这届年轻人，年夜饭都买预制菜了](#)
- [欧洲首台超5000量子位元的量子计算机在德国启动](#)
- [春节不回家，我真的好快乐](#)
- [新加坡会是蔚来的第二个合肥吗？](#)
- [疯狂的预制菜：都在炒，没人吃？](#)
- » [更多新闻...](#)

Powered by:

[博客园](#)

Copyright © 2022 jerry\_fuyi

Powered by .NET 6 on Kubernetes