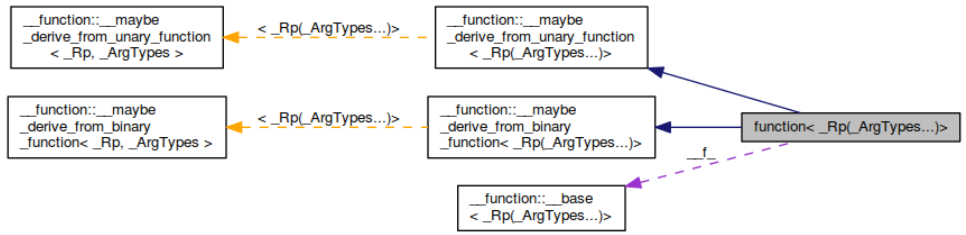
std::function源码分析

原创 htfy96 工作日志 2016/02/09 22:33 阅读数 6.5K

##概览

###std::function




```
template<class _Rp, class ..._ArgTypes>
class function<_Rp(_ArgTypes...)>
    : public __function::__maybe_derive_from_unary_function<_Rp(_ArgTypes...)>,
      public __function::__maybe_derive_from_binary_function<_Rp(_ArgTypes...)>
{
        __base* __f_; //points to __func
        aligned_storage< 3 *sizeof(void *)>::type       __buf_;
        //...
};
```

std::function 最重要的部分就是这个 __base* 指针，及其所指向的存储了实际可调用对象的多态类 __func 。 __base 类充当了 __func 类的接口，定义了 clone 、 operator() 等纯虚函数。

而 __func 对象可能存储的区域**之一**就是自带的**默认缓冲区** __buf_ ,部分MIPS指令集要求指令必须要对齐，所以这里的存储地址也要遵循平台默认的对齐方式。默认的大小是 3*sizeof(void*) ，这是纯经验数据，对大

关于作者

htfy96
程序员
关注 私信 提问

文章	经验值	粉丝	关注
5	296	8	1

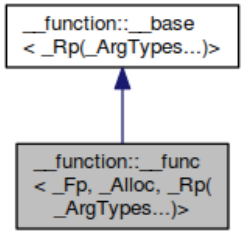
作者的专辑 全部

工作日志 (5)

nononolylyly
文章 1 访问 864

大小应该相对是 4 sizeof(void*)）。但因为可调用对象大小不受约束，所以实际存储的区域**可能也会在新开的堆上。**

std::function 类继承自 __maybe_derive_from_unary_function 与 __maybe_derive_from_binary_function 两个类。这两个类在函数分别满足 ResultT f(ArgT) 和 ResultT f(Arg1T, Arg2T) 形式的时候，分别会特化继承 std::unary_function<ResultT, ArgT> 与 std::binary_function<ResultT, arg1T, arg2T> 。 这两个类是C++11之前对两种特殊可调用对象的静态接口，其内只有 typedef ，在C++11之后已经deprecated，C++17后将移除，这里继承这两个接口只是为了兼容目的。关于C++11之前的 <functional> 分析，详见这篇文章。



###__func

```
template<class _Fp, class _Alloc, class ..._ArgTypes>
class __func<_Fp, _Alloc, _Rp(_ArgTypes...)>
    : public  __base<_Rp(_ArgTypes...)>
{
    __compressed_pair<_Fp, _Alloc> __f_;
    //...
};
```

__func 是实际存储可调用对象的类，其继承了 __base 这个接口。可调用对象与allocator都被存储在一个 __compressed_pair 当中。

###__base

```
template<class _Rp, class ..._ArgTypes>
class __base<_Rp(_ArgTypes...)>
{
    __base(const __base&);
    __base& operator=(const __base&);
```

```
    virtual ~__base() {}
    virtual __base* __clone() const = 0;
    virtual void __clone(__base*) const = 0;
    virtual void destroy() _NOEXCEPT = 0;
    virtual void destroy_deallocate() _NOEXCEPT = 0;
    virtual _Rp operator()(_ArgTypes&& ...) = 0;
#ifndef _LIBCPP_NO_RTTI
    virtual const void* target(const type_info&) const _NOEXCEPT = 0;
    virtual const std::type_info& target_type() const _NOEXCEPT = 0;
#endif  // _LIBCPP_NO_RTTI
};
```

__base 是一个纯虚基类，是 __func 类的接口，对外提供了 clone（复制、移动）、 destroy （析构）、 operator() （调用）等函数。 ##构造 从可调用对象构造出 function 有以下几步：

- 检查该对象是否可调用
- 若缓冲区 __buf_ 不够存放可调用对象，新开内存
- 在 __f_ 指向的内存区域调用placement new，移动构造可调用对象。

###对象是否可调用

```
template<class _Rp, class ..._ArgTypes>
template <class _Fp>
function<_Rp(_ArgTypes...)>::function(_Fp __f,
    typename enable_if
        <
            __callable<_Fp>::value &&
            !is_same<_Fp, function>::value
        >::type*) //使用SFINAE检查该对象是否可调用，并且不是std::function（防止出现function套f

    : __f_(0)
```

在滚到下面之前，先猜一下__callable是怎么实现的。注意以下代码也是合法的，还要考虑 reference_wrapper 、返回值转化等各种形式：

```
{
    void f() { cout << "called" << endl;}
};

int main()
{
    void (A::*mfp)() = &A::f;
    std::function<void(A*)> f(mfp);
    A a;
    f(&a);
}
```

实际上，实现__callable主要依赖于 invoke 的实现， invoke 规定了一个统一的调用方式，将于C++17标准中出现。不论是 f(a,b) 还是 (f.*a)(b) （ f 是可调用对象， a 是成员函数指针）还是 (a->*f)(b) （ a 是可调用对象指针， f 是成员函数指针），都可以以 invoke(f,a,b) 的形式调用。

知道了这个函数，我们只要规定 invoke 可以调用，并且返回值可以转换成 std::function 规定的返回类型的函数就是 callable ：

```
template <class _Fp, bool = !is_same<_Fp, function>::value &&
                            __invokable<_Fp&, _ArgTypes...>::value> //__invokable代表是否
    struct __callable;
template <class _Fp>
    struct __callable<_Fp, true>
    { //如果可以发生调用，继续检查返回值是否可以转换成function的返回值
        static const bool value = is_same<void, _Rp>::value || //实际任何类型的T fun(...)
            is_convertible<typename __invoke_of<_Fp&, _ArgTypes...>::type,
                           _Rp>::value;
    };
template <class _Fp>
    struct __callable<_Fp, false>
    {
        static const bool value = false;
    };
```

> 点帮助：http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4165.pdf

###内存分配与构造

####function 为了保证异常安全。分为两种情况：若自带的 __buf_ 大小够大，且可调用对象的构造函数不抛出异常，则直接构造；否则，则用 unique_ptr 来处理allocator分配出的内存地址，再在上面调用构造函数，这样即使构造函数抛出了异常， unique_ptr 也会自动delete掉指向的内存地址；而如果用裸指针，构造函数抛出异常就会内存泄漏。

```
if (__not_null(__f))
{
    typedef __function::__func<_Fp, allocator<_Fp>, _Rp(_ArgTypes...)> _FF;
    if (sizeof(_FF) <= sizeof(__buf_) && is_nothrow_copy_constructible<_Fp>::value) //缓
    {
        __f_ = (__base*)&__buf_; //__f_指向缓冲区
        ::new (__f_) _FF(_VSTD::move(__f)); //直接构造，间接调用了__func的移动构造函数
    }
    else
    {
        typedef allocator<_FF> _Ap;
        _Ap __a;
        typedef __allocator_destructor<_Ap> _Dp;
        unique_ptr<__base, _Dp> __hold(__a.allocate(1), _Dp(__a, 1)); //__a.allocate(1)分
        ::new (__hold.get()) _FF(_VSTD::move(__f), allocator<_Fp>(__a)); //placement new,
        __f_ = __hold.release(); //安全了，把指针的控制权移交给__f_
    }
}
```

####__func 这个构造函数之中调用了 __func 类的构造函数：

```
__compressed_pair<_Fp, _Alloc> __f_; //__func的的__f_是一个compressed_pair，不是上面的bas

explicit __func(_Fp&& __f, _Alloc&& __a)
    : __f_(piecewise_construct, _VSTD::forward_as_tuple(_VSTD::move(__f)),
                                _VSTD::forward_as_tuple(_VSTD::move(__a))) {}
```

```
struct Null {};
struct Test { int a; };

struct B
{
    Null n;
    Test c;
};

    cout << sizeof(Null) << " "<< sizeof(Test)<<" "<<sizeof(B)<<endl; //1 4 8
```

但这样在有内存对其的时候其实浪费了大量的存储空间，特别是对于 function 这类小对象来说节约空间非常重要。对于空类Null，一个继承自它的类B2，且B2非空类，则B2不会因为Null类的继承而像上例中的内含一样占用空间：

```
struct B1 : private Null
{
};
struct B2 : private B1, private Test
{
};
    cout << sizeof(B1)<<" "<<sizeof(B2) << endl; // 1 4
```

compressed_pair 就用了这种技巧来压缩内存，这种技术在 boost::compressed_pair 当中已经有成熟的库，这里libc++内部也制作了一个自己的 __compressed_pair 。

再来说说这个 piecewise_construct 。一般使用 pair 时，我们都是利用 make_pair(T1(arg1, arg2), T2(arg)) 这样来构造。实际上，发生了以下的步骤：

- 构造出一个 T1 的xvalue(消亡值，属于右值)，匹配上 make_pair(T1&&, T2&&)
- make_pair 把这两个右值引用传递给 pair<T1, T2>(T1&& t1, T2&& t2)
- pair 的构造函数把内部的 first , second 对象在初始化列表中以 first(t1), second(t2) 形式初始化，这个t1,t2都是右值，所以调用了移动构造函数

云券：piecewise_construct 就是为此而生的。使用 pair<T1, T2>(piecewise_construct, tuple<Args...>&& t1, tuple<Args...>&& t2) 这样的形式，最终初始化列表中会直接转化成: first(std::forward<_Args1>(std::get<_I1>(__first_args))...) ，即这些参数会被直接传递给 first，second 对象，直接在 pair 的构造函数内初始化 first second ，而不是先在形成参数时构造出临时对象，再移动过去。这样既有比较好的性能，也不需要具有 first，second 具有复制、移动构造函数。

##复制与移动 复制与移动实际上都是操作内部的 __func 对象。但是，构造函数不具有多态性，怎么根据父类的指针来获得子类的拷贝呢？这是一种常用的技巧：

```
virtual SuperClass* SubClass::clone() { return new SubClass(*this); } //相当于多态new
virtual SuperClass* SubClass::clone(SuperClass* p) { return new (p) SubClass(*this); } //多态
```

###复制构造

```
//.__f_是指向__func对象的指针
template<class _Rp, class ..._ArgTypes>
function<_Rp(_ArgTypes...)>::function(const function& __f)
{
    if (__f.__f_ == 0) //未初始化
        __f_ = 0;
    else if (__f.__f_ == (const __base*)&__f.__buf_) //另一个对象的__func存放在自身的缓冲区内，
    {
        __f_ = (__base*)&__buf_; //自己指向自身的缓冲区
        __f.__f_->__clone(__f_); //相当于new (__f_) __func(另一个__func)，把另一个__func复制到
    }
    else
        __f_ = __f.__f_->__clone(); //放不下了，让它新开一块内存复制到其中，然后自己指过去
}
```

###移动构造

```
template<class _Rp, class ..._ArgTypes>
function<_Rp(_ArgTypes...)>::function(function&& __f) _NOEXCEPT
{
    if (__f.__f_ == 0)
```

```
    {
        __f_ = (__base*)&__buf_; //不能直接指到对方缓冲区去，因为对方__buf会随对象析构销毁掉
        __f.__f_->__clone(__f_); //还是要复制到自己的缓冲区来
    }
    else
    {
        __f_ = __f.__f_; //对方的__func在堆上，直接指过去
        __f.__f_ = 0; //把对方的__f_指空
    }
}
```

##调用

调用的时候先检查内部的 __f_ 指针是否为空，若空则抛异常，否则调用 __f_ 指向的 __func 对象的 operator()：

```
template<class _Rp, class ..._ArgTypes>
_Rp
function<_Rp(_ArgTypes...)>::operator()(_ArgTypes... __arg) const
{
#ifndef _LIBCPP_NO_EXCEPTIONS
    if (__f_ == 0)
        throw bad_function_call();
#endif  // _LIBCPP_NO_EXCEPTIONS
    return (*__f_)(_VSTD::forward<_ArgTypes>(__arg)...); //调用内部__func对象的operator()
}
```

ArgType	forward<ArgType>
T	static_cast<T&&>
T&	static_cast<T&>
T&&	static_cast<T&&>

std::forward 作用如其名，即将参数向前传递。原先的 ArgType = T 时，在调用这个函数时已经复制过了一遍，因此复制过的值可以作为右值， forward<T>(t) 将 t 转成了右值。而对于原先是左值、右值引用的来

```
template<class _Fp, class _Alloc, class _Rp, class ..._ArgTypes>
_Rp
__func<_Fp, _Alloc, _Rp(_ArgTypes...)>::operator()(_ArgTypes&& ... __arg) //完美转发
{
    typedef __invoke_void_return_wrapper<_Rp> _Invoker; //后述，与invoke的特殊语法有关
    return _Invoker::__call(__f_.first(), _VSTD::forward<_ArgTypes>(__arg)...); //__f_.first(
}
```

这里不直接 `return invoke(__f_.first(), ...)` 的原因是，如果 `__f_` 的返回值是 `void` ，但实际可调用对象返回值，就会出错：

```
int foo() { return 42; }
void bar() { return foo(); } //报错,int不能转成void
void bar2() { foo(); } //针对void返回值这样才对
function<void()> f(foo); //合法
```

所以针对 `void` 返回值要特化一下：

```
template <class _Ret>
struct __invoke_void_return_wrapper
{
    template <class ..._Args>
    static _Ret __call(_Args&&... __args)
    {
        return __invoke(_VSTD::forward<_Args>(__args)...);
    }
};

template <>
struct __invoke_void_return_wrapper<void>
{
    template <class ..._Args>
    static void __call(_Args&&... __args)
    {
        __invoke(_VSTD::forward<_Args>(__args)...);
```

仔细思考一下整个调用过程，发现还是具有负担的： 对于形参是T的对象来说，

```
void foo(A) {}
A a;

foo(a); //a被复制构造一次

function<void(A)> f(foo);
f(a); //先被复制构造一次，再被移动构造一次
// 等价于
A b(a); //这个复制发生在function::operator()的形参表里
foo(forward<A>(b)); //发生了移动构造
```

所以在C++11中，移动构造非常重要，如果能够定义移动构造函数请务必定义。否则该例就会退化到两次复制构造，如果在传递大对象时将是不小的负担。

##总结

- `std::function` 是自带的可调用对象适配器。它通过内部 `__f_` 指针调用所指向的 `__func` 类对象的虚方法来实现多态的函数调用、 `new` 与 `placement new` 。其中内带了一个大小是 `3*sizeof(void*)` 的缓冲区，小对象将被分配在缓冲区上，大对象将另外在堆上分配内存存储。
- `__func` 对象利用了 `compressed_pair` 技术来压缩存储的 `可调用对象 - Allocator` 对，并利用 `piecewise_construct` 来就地构造这两个对象，能够处理这两个类没有移动复制构造函数的情况，也提高了性能。
- `std::function` 在形参是非引用时会多发生一次移动构造，可能成为性能的瓶颈。

ctags源码分析(1) - 概要

C++ Boost全库简介

Portable_dev Win32C++开发环境包 mingw+clang+boost+vim

从cTags的vString学习动态字符串

htfy96 博主

> 引用来自“Anthonyhl”的评论
>
> 不错，但对于最后的性能比较，可能点误解。
> foo(forward<A>(b)); 这个过程，并不会移动构造。
> 这一点forward和move一样，只是保证类型正确，不会增加运行时开销。
>
> 另外，sizeof(void*)*3，除了大部分平台都可以放下函数指针外，加上base*那个指针，正好是4个指针长度。

1. forward(b) == b的右值引用，移动构造发生在 利用这个右值引用 初始化foo的形参过程中 2. 是4个指针长度这一点的确没想到，学习了。

2016/02/13 21:57 回复 举报

黄亮Anthony

不错，但对于最后的性能比较，可能点误解。
foo(forward<A>(b)); 这个过程，并不会移动构造。
这一点forward和move一样，只是保证类型正确，不会增加运行时开销。

另外，sizeof(void*)*3，除了大部分平台都可以放下函数指针外，加上base*那个指针，正好是4个指针长度。

2016/02/13 15:02 回复 举报

更多评论