Bit Focus

std::function 基本实现

Posted at 2017-11-25 04:55:23 | Updated at 2017-11-25 04:55:23

std::function 是在 C++11 中新增的一个用于统一包装可调用对象的模板类型. 所谓统一包装, 就是无论被包装的内容的实际类型, 只要符合相应的函数调用签名, 都可以装入一个 std::function 对象中使用. 比如

Code Snippet 0-0

```
#include <iostream>
#include <functional>

// 全局函数
int fn_ptr(int x, int y)
{
    return x + y;
}

// 包含 2 个 int 成员的函数对象类型
struct TwoInts {
    TwoInts(int m_, int n_)
        : m(m_)
```

```
    {}

    int operator()(int x, int y)
    {
        return x + y + m + n;
    }

    int m;
    int n;
};

int main()
{
    // 使用 std::function 类型包装全局函数指针
    std::function<int(int, int)> f(fn_ptr);
    std::cout << f(1, 2) << std::endl; // 输出 3

    // 使用 std::function 类型包装函数对象
    std::function<int(int, int)> g(TwoInts(10, 20));
    std::cout << g(1, 2) << std::endl; // 输出 33

    return 0;
}
```

上面的使用例子中, 两个 `std::function` 对象定义都在栈上. 按照 C++ 的常识, 两个对象一定有相同的尺寸, 即对它们求 `sizeof` 得出的值一定相等. 但用于构造这两个 `function` 对象的材料却有着不同的尺寸, 也就是说 `function` 可以 "捕获" 任何尺寸的可调用对象, 这正是其奇妙之处.

虽然 std::function 是在 C++11 中引入的, 但作为一个基本实现的分析, 本文将排除所有 C++11 的特性以避免不必要的解释. 当然, 这样会产生一个硬伤: 由于可变参数模板特性也是 C++11 中引入的特性, 本文的实现中将不支持**任意多个**模板类型参数, 而是使用返回值类型加上 2 个参数的类型共计 3 个类型作为模板的类型参数列表. 亦即, 在 C++11 中, 下面的用法是可能的

```
std::function<double()> f;          // 只有返回值类型 <double> 的特化
std::function<int(std::string)> g; // 有返回值类型和 1 个参数类型 <int, std::string> 的特化
std::function<void(int, float)> h; // 有返回值类型和 2 个参数类型 <void, int, float> 的特化
// 可以扩展为任意多个参数类型的特化, 这是 C++11 的新特性
```

而本文中要实现的只包含下面这样的形式

Code Snippet 0-1

```
// 默认特化没有实现
template <typename T>
class function;

// 实现有返回值类型和 2 个参数类型的偏特化
template <typename Ret, typename Arg0, typename Arg1>
class function<Ret(Arg0, Arg1)> {
    // ...
};
```

语法上, 类似上面的 function<int(int, int)>, class function<Ret(Arg0, Arg1)> 等类似函数签名的模板特化形式并不常见, 虽然它是

何在内部维护不同类型不同尺寸的可调用对象.

这一功能的实现显然需要堆分配. 也就是说, 在 function 结构内部会有一个指针, 指向某一可调用的对象. 但由于这一可调用对象是编译时不可确定的, 因此又必须包含某种运行时动态的部分. 这样的话方案基本就敲定了: 构造一个抽象基类, 然后 function 对象持有这个基类的指针就行了.

Code Snippet 0-2

```
template <typename Ret, typename Arg0, typename Arg1>
class function<Ret(Arg0, Arg1)> {
    // 可调用的抽象基类
    struct callable_base {
        virtual Ret operator()(Arg0 arg0, Arg1 arg1) = 0;
        virtual ~callable_base() {}
    };

    callable_base* callable_ptr;
public:
    Ret operator()(Arg0 arg0, Arg1 arg1)
    {
        // function 的调用行为就转接到内部指针指向的对象上
        return (*callable_ptr)(arg0, arg1);
    }
};
```

这样一来调用的部分就没问题了, 下面就是构造的部分. 从之前的用法上看, function 需要有一个接受**任意类型**参数的构造函数.换言之, 需要实现下面这个模板构造函数

```
template <typename Ret, typename Arg0, typename Arg1>
class function<Ret(Arg0, Arg1)> {
    // ...
public:
    // ...

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

现在要解决的问题是如何把任意类型的 f 转化为一个 callable_base 的某个子类的对象了. 这一问题的解决方案有固定的模式, 就是编写一个同样接受任意类型模板参数的模板类, 并继承实现 callable_base 抽象基类.

Code Snippet 0-4

```
template <typename Ret, typename Arg0, typename Arg1>
class function<Ret(Arg0, Arg1)> {
    struct callable_base {
        virtual Ret operator()(Arg0 arg0, Arg1 arg1) = 0;
        virtual ~callable_base() {}
    };

    // 此模板类型实现了上述抽象基类
    // 内部复制保存一份函数对象
    template <typename F>
    struct callable
        : callable_base
    {
```

```
            callable(F functor)
                : functor(functor)
            {}

            virtual Ret operator()(Arg0 arg0, Arg1 arg1)
            {
                return functor(arg0, arg1);
            }
        };

        callable_base* callable_ptr;
    public:
        template <typename F>
        function(F f)
            : callable_ptr(new callable<F>(f))
        {}

        ~function()
        {
            delete callable_ptr;
        }

        Ret operator()(Arg0 arg0, Arg1 arg1)
        {
            return (*callable_ptr)(arg0, arg1);
```

```
};
```

这样的实现虽然不能说完善, 但至少可以用来运行本文开头的例子了. 把 std 名字空间拿掉, 如果上述 function 类型定义在全局, 则可用下面的代码跑起来.

Code Snippet 0-5

```
int main()
{
    ::function<int(int, int)> f(fn_ptr);
    std::cout << f(1, 2) << std::endl;

    ::function<int(int, int)> g(TwoInts(10, 20));
    std::cout << g(1, 2) << std::endl;

    return 0;
}
```

说这个实现不完善, 是因为它内部的指针操作不安全. 进行对象复制之后, 就有双重 delete 的危险. 说到底, 上面的代码没有遵循"三法则" (the rule of three) 规约, 在自定义了析构函数的情况下没有正确定义复制构造函数和赋值算符重载.

而实现复制构造函数和赋值算符重载这两个函数的要点在于如何复制内部函数对象. 由于函数对象存入 function 对象之后就剩下抽象基类的指针了, 无法获取其确切的类型信息, 也就不知道该如何调用其复制构造函数. 不过这也并不是特别难以解决: 既然解决如何调用具体的函数对象类型可以通过纯虚函数实现, 那么如何复制具体的函数对象也可以通过纯虚函数实现. 也就是说, 追加下面的函数

Code Snippet 0-6

```
template <typename Ret, typename Arg0, typename Arg1>
class function<Ret(Arg0, Arg1)> {
```

```
        virtual Ret operator()(Arg0 arg0, Arg1 arg1) = 0;
        // 追加用于复制自身的虚函数
        virtual callable_base* clone() const = 0;
        virtual ~callable_base() {}
    };

    template <typename F>
    struct callable
        : callable_base
    {
        F functor;

        callable(F functor)
            : functor(functor)
        {}

        // 复制自身
        virtual callable_base* clone() const
        {
            return new callable<F>(functor);
        }

        virtual Ret operator()(Arg0 arg0, Arg1 arg1)
        {
            return functor(arg0, arg1);
        }
    };
```

```
    callable_base* callable_ptr;
public:
// 复制构造与赋值算符重载使用 clone 函数实现
    function(function const& rhs)
        : callable_ptr(rhs.callable_ptr->clone())
    {}

    function& operator=(function const& rhs)
    {
        delete callable_ptr;
        callable_ptr = rhs.callable_ptr->clone();
    }

    ~function()
    {
        delete callable_ptr;
    }
};
```

这样一个基本可用并可以复制, 甚至还能往 STL 容器里放的 function 类型就实现了. 这一实现也可以看作是对栈空间友好的实现: 在栈上定义一个 function 对象仅需要一个指针的空间! 不过问题就是无论是多大多小的函数对象, 为了包装它们 function 对象都会申请动态空间, 在时间效率上就不够经济了. 有没有办法在被包装的函数对象很小的情况下避免一次空间配置呢? 答案是肯定的, 不过会相应有些许代价.

如果要避免一次 new 又想获得某种程度上的多态行为, 那么抽象基类的做法肯定要换掉. 一个容易想到的替代方案就是使用函数指针代替虚表, 这样一来 function 内部会需要 4 个指针, 如

```
template <typename Ret, typename Arg0, typename Arg1>
class function<Ret(Arg0, Arg1)> {
    // 取代虚表的函数指针
    Ret (* call_fn)(function*, Arg0, Arg1);
    void* (* clone_fn)(function const*);
    void (* destruct_fn)(function*);

    // 之前用于存储可调用对象本身的指针
    // 由于无法知道其具体类型，这里声明为 void* 类型
    void* callable_ptr;

    // 将虚基类的 3 个虚函数分别替换为下面的 3 个函数模板
    // 函数都是 static 定义的，其第一个参数将会是 function 内的可调用对
象
    // 请注意，将可调用对象指针转换为 Functor* 的行为在函数内部通过
static_cast 完成
    // 而参数列表与模板类型 Functor 无关，因此这一组模板函数的特化结果的签
名是相同的
    // 这样就能方便地将特化结果赋值给上面声明的函数指针
    template <typename Functor>
    static Ret call(function* self, Arg0 arg0, Arg1 arg1)
    {
        return (*static_cast<Functor*>(self->callable_ptr))(arg0,
arg1);
    }
```

```
    static void* clone(function const* src)
    {
        new new Functor(*static_cast<Functor const*>(src-
>callable_ptr));
    }

    template <typename Functor>
    static void destruct(function* self)
    {
        delete static_cast<Functor*>(self→callable_ptr);
    }
public:
    // 当 function 以某个可调用对象构造时，以该可调用对象的类型特化各个函
数模板
    // 然后赋值给各个函数指针
    template <typename F>
    function(F f)
        : call_fn(call<F>)
        , clone_fn(clone<F>)
        , destruct_fn(destruct<F>)
        , callable_ptr(new F(f))
    {}

    function(function const& rhs)
        : call_fn(rhs.call_fn)
        , clone_fn(rhs.clone_fn)
        , destruct_fn(rhs.destruct_fn)
```

```
clone 函数复制创建一个可调用对象
    {}

    function& operator=(function const& rhs)
    {
        destruct_fn(callable_ptr);
        call_fn = rhs.call_fn;
        clone_fn = rhs.clone_fn;
        destruct_fn = rhs.destruct_fn;
        callable_ptr = clone_fn(rhs.callable_ptr);
    }

    ~function()
    {
        destruct_fn(callable_ptr); // 析构时以 destruct 函数析构可调
用对象
    }

    Ret operator()(Arg0 arg0, Arg1 arg1)
    {
        return call_fn(callable_ptr, arg0, arg1);
    }
};
```

改造到这一步, 接下来就可以开始优化了. 在上述实现中, function 里存了一个指向函数对象的指针, 这东西挪用一下, 完全可以拿来存一些小对象, 常见的体系结构里一个指针的空间里可以塞 2 个 int 进去呢.

尺寸比较得出的结果是一个编译时可确定的布尔值, 这可以利用 tag dispatching 技巧来分离逻辑. 具体要做的是

Code Snippet 0-8

```
// 定义一个 tag dispatching 模板类
template <bool placement>
struct Placement {};

template <>
class function<Ret(Arg0, Arg1)> {
    // ...
private:
    // 实现两个重载, 分别对应空间足够与不够的情形
    template <typename F>
    void init(F f, Placement<false>)
    {
        // 一个指针尺寸不足以存放可调用对象的初始化过程
        // 与之前的构造函数初始化列表一致
        call_fn = call<F>;
        clone_fn = clone<F>;
        destruct_fn = destruct<F>;
        callable_ptr = new F(f);
    }

    template <typename F>
    void init(F f, Placement<true>)
    {
```

```
            // 各个 _placement 函数实现见后文
            call_fn = call_placement<F>;
            clone_fn = clone_placement<F>;
            destruct_fn = destruct_placement<F>;
            // 使用 placement new 将可调用对象构造在 callable_ptr 所在位置
            // 注意, 需要取得 callable_ptr 的地址然后调用构造函数
            new (&this→callable_ptr) F(f);
        }
    public:
        template <typename F>
        function(F f)
        {
            // 构造函数修改: 根据对象尺寸决定调用哪个 init 函数重载来初始化
            init(f, Placement<(sizeof(F) ⩽ sizeof(callable_ptr))>
    ());
        }
    private:
        // 在不额外申请堆空间的情况下 callable_ptr 的只能不是指向可调用对象,
    而是可调用对象本身
        // 所以 "指向" 可调用对象的指针是 callable_ptr 自身的地址
        // 这里定义两个函数用来进行取地址操作
        static void* addr_of_callable(function* f)
        {
            return &f→callable_ptr;
        }

        static void const* addr_of_callable(function const* f)
```

```
            return &t->callable_ptr;
        }

        // 各个 _placement 模板函数实现
        template <typename Functor>
        static Ret call_placement(function* self, Arg0 arg0, Arg1
arg1)
        {
            // 与非 _placement 版本类似，只是指针换成了 callable_ptr 自身
            return (*static_cast<Functor*>(addr_of_callable(self)))
(arg0, arg1);
        }

        template <typename Functor>
        static void* clone_placement(function const* src)
        {
            // ???
        }

        template <typename Functor>
        static void destruct_placement(function* self)
        {
            // 与非 _placement 版本不同，使用 placement 析构
            static_cast<Functor*>(addr_of_callable(self))-
>~Functor();
        }
};
```

对象. 由于这种情况下不能分配空间而需要在既有的空间上进行 placement new, 按之前的思路和签名实现是不可行的. 所以要对 clone 方法稍作修改

Code Snippet 0-9

```
template <>
class function<Ret(Arg0, Arg1)> {
    // ...

    // 不再用返回值 (返回值类型改为 void), 增加一个参数 function* 作为可调用对象复制的目标
    void (* clone_fn)(function*, function const*);

    // 通过 new 申请额外空间的复制方法, 直接将复制出的可调用对象挂在目标的 callable_ptr 成员上
    template <typename Functor>
    static void clone(function* dst, function const* src)
    {
        dst->callable_ptr = new Functor(*static_cast<Functor const*>(src->callable_ptr));
    }

    // 不通过 new 的复制方式, 在目标的 callable_ptr 成员处调用 placement new 构造
    template <typename Functor>
    static void clone_placement(function* dst, function const* src)
    {
```

```
const*>(addr_of_callable(src)));
    }

    // 另还需要修改复制构造函数和赋值算符重载中使用到 clone_fn 的地方
};
```

这样改头换面的 function 实现就完成了. 下面上一份附例子的完整代码.

Code Snippet 0-10

```
#include <iostream>

template <bool placement>
struct Placement {};

template <typename T>
class function;

template <typename Ret, typename Arg0, typename Arg1>
class function<Ret(Arg0, Arg1)> {
    typedef Ret (* CallFn)(function*, Arg0, Arg1);
    typedef void (* CloneFn)(function*, function const*);
    typedef void (* DestructFn)(function*);

    CallFn call_fn;
    CloneFn clone_fn;
    DestructFn destruct_fn;

    void* callable_ptr;
```

```
    template <typename Functor>
    static Ret call(function* self, Arg0 arg0, Arg1 arg1)
    {
        return (*static_cast<Functor*>(self->callable_ptr))(arg0,
arg1);
    }

    template <typename Functor>
    static void clone(function* dst, function const* src)
    {
        dst->callable_ptr = new Functor(*static_cast<Functor
const*>(src->callable_ptr));
    }

    template <typename Functor>
    static void destruct(function* self)
    {
        delete static_cast<Functor*>(self->callable_ptr);
    }

    static void* addr_of_callable(function* f)
    {
        return &f->callable_ptr;
    }

    static void const* addr_of_callable(function const* f)
    {
```

```
    }

    template <typename Functor>
    static Ret call_placement(function* self, Arg0 arg0, Arg1
arg1)
    {
        return (*static_cast<Functor*>(addr_of_callable(self)))
(arg0, arg1);
    }

    template <typename Functor>
    static void clone_placement(function* dst, function const*
src)
    {
        new (addr_of_callable(dst)) Functor(*static_cast<Functor
const*>(addr_of_callable(src)));
    }

    template <typename Functor>
    static void destruct_placement(function* self)
    {
        static_cast<Functor*>(addr_of_callable(self))-
>~Functor();
    }

    template <typename F>
    void init(F f, Placement<false>)
```

```
            call_fn = call<F>;
            clone_fn = clone<F>;
            destruct_fn = destruct<F>;
            callable_ptr = new F(f);
        }

        template <typename F>
        void init(F f, Placement<true>)
        {
            call_fn = call_placement<F>;
            clone_fn = clone_placement<F>;
            destruct_fn = destruct_placement<F>;
            new (&this->callable_ptr) F(f);
        }
    public:
        template <typename F>
        function(F f)
        {
            init(f, Placement<(sizeof(F) <= sizeof(callable_ptr))>
());
        }

        function(function const& rhs)
            : call_fn(rhs.call_fn)
            , clone_fn(rhs.clone_fn)
            , destruct_fn(rhs.destruct_fn)
        {
```

```
        }

        function& operator=(function const& rhs)
        {
            destruct_fn(this);
            call_fn = rhs.call_fn;
            clone_fn = rhs.clone_fn;
            destruct_fn = rhs.destruct_fn;
            clone_fn(this, &rhs);
        }

        ~function()
        {
            destruct_fn(this);
        }

        Ret operator()(Arg0 arg0, Arg1 arg1)
        {
            return call_fn(this, arg0, arg1);
        }
    };

    int fn_ptr(int x, int y)
    {
        return x + y;
    }
```

```
    TwoInts(int m_, int n_)
        : m(m_)
        , n(n_)
    {}

    int operator()(int x, int y)
    {
        return x + y + m + n;
    }

    int m;
    int n;
};

struct FourInts {
    FourInts(int m_, int n_, int p_, int q_)
        : m(m_)
        , n(n_)
        , p(p_)
        , q(q_)
    {}

    int operator()(int x, int y)
    {
        return x + y + m + n + p + q;
    }
```

```
    int n;
    int p;
    int q;
};

int main()
{
    // 以函数指针构造
    ::function<int(int, int)> f(fn_ptr);
    std::cout << f(1, 2) << std::endl;

    // 以函数对象构造
    ::function<int(int, int)> g(TwoInts(10, 20));
    std::cout << g(1, 2) << std::endl;

    // 复制构造
    ::function<int(int, int)> h(g);
    std::cout << h(3, 4) << std::endl;

    // 赋值算符
    h = f;
    std::cout << h(3, 4) << std::endl;

    // 以超过指针尺寸的函数对象构造的 function 赋值
    h = ::function<int(int, int)>(FourInts(3, 4, 5, 6));
    std::cout << h(1, 2) << std::endl;
```

```
}
```

后记:

上述实现非常粗略, 虽然能囊括 C++11 中的 `function` 的令人眼前一亮的功能, 但也仅仅只是形似的阶段, 离神似还有一段距离. 除了可变参数模板功能上的不足限制了模板参数的灵活度, 缺少完美转发 (perfect forwarding) 也使得在调用过程中可能无法顾及效率方面的问题 (如可能有不必要的参数复制等). 建议读者进一步阅读有关 C++11 的资料. **强力自荐一波** **C++11 有什么以及为什么**.

STL 中的实现稍有不同. 在 STL 中, `clone_fn` 和 `destruct_fn` 合并为了一个函数指针 (`destruct_fn` 较之 `clone_fn` 少一个 `function const*` 参数, 不过这没关系, 在析构时此参数传入空指针即可), 增加一个整数参数, 根据该整数参数的值决定是复制还是析构. 虽然看起来有些混乱, 但很有效地省下了一个指针的空间.

不过, STL 中 `function` 的尺寸通常仍然是 4 个指针大小, 这是因为存储可调用对象的空间有 16 字节 (64 位系统上). 并不是要刻意留出更大的空间, 而是考虑到成员函数指针. C++ 中成员函数指针要存储一些可能与多重继承或虚表有关的信息, 因为比静态函数指针或其他普通指针要大.

Post tags: STL C++11 C++

Comments:

Thank Fly said, at 2020-02-29 07:39:18.263339 (UTC)

cool, 写的很好，学习学习；我再去看看标准库的实现。

Leave a comment:

Your name

Your email (for Gravatar, wont be public)

Your homepage link (placed after your name)