Darkfall 05:07:58
sora::SoraFunction<void(int)> func2 = sora::Bind(&ins, &t::testFunc2);
func2(1);
Darkfall 05:08:02
boost太巧妙了。。
SC 05:08:04
小米。在798发布的。。囧
Darkfall 05:08:12
这种实现。。
Darkfall 05:08:21
不看他的源代码压根就想不到啊。。
SC 05:08:19
这个是。。。函数对象+绑定器?
Darkfall 05:08:28
模板玩到这种程度。。。
Darkfall 05:08:28
恩
Darkfall 05:08:30
我山寨的
Darkfall 05:08:41
boost::function, boost::mem_fun, boost::bind
Darkfall 05:08:44
简单的山寨。。
Darkfall 05:08:51
机制不可能有他那种程度。。
SC 05:08:53
你是指将函数prototype当类型传给模板参数么?
Darkfall 05:09:16
那个不是关键
Darkfall 05:09:18
关键是在于
Darkfall 05:09:25
你如何把这些封装到一起
Darkfall 05:09:46
任意返回值，任意(不完全)数量的参数的函数
Darkfall 05:09:50
支持隐式转换
Darkfall 05:09:53
支持仿函数
Darkfall 05:09:59
支持C函数和类成员函数
Darkfall 05:10:34
int(int, int)可以转到 function<float(int, int)>
SC 05:10:44
&t::testFunc2
这是嘛语法。。

Darkfall 05:10:50
类成员函数
SC 05:11:09
&和::一起用0 0?
Darkfall 05:11:39
t::testFunc2是个函数
Darkfall 05:11:51

SC 05:12:38
哦。。
&(t::testFunc2)
?
Darkfall 05:12:43
en
SC 05:12:47
.....soga
Darkfall 05:12:50
单个的sora::SoraFunction<void(int)>可以是类成员函数
Darkfall 05:12:52
可以是C函数
Darkfall 05:12:58
可以是operator()(int)仿函数
Darkfall 05:13:05
=_,=
Darkfall 05:13:09
这种封装太鬼畜了
SC 05:13:22

Darkfall 05:13:24
不用bind类成员函数还有另外一种处理方式
Darkfall 05:13:25
```
sora::SoraFunction<void(t*)> func3 = &t::testFunc1;
    func3(&ins);
```
Darkfall 05:13:35
传this
SC 05:13:50
soga...
Darkfall 05:13:57
t::testFunc1(void)实际上被转到testFunc(t*)
Darkfall 05:14:05
啊哈哈哈。
SC 05:14:16
原来C++也是这种机制。。。
Darkfall 05:14:23
就是这两天坐飞机的时候山寨出来的。。
SC 05:14:31

可能oo都是吧。。
Darkfall 05:14:38
恩
Darkfall 05:15:32

SC 05:15:37
话说。。。好多书了。。。我没办法背了。。只能邮寄了。。
Darkfall 05:15:45
stl的mem_fun只能支持1到2个参数
Darkfall 05:16:03
用bind_first和bind_second
SC 05:16:57
恩，boost的。。？
Darkfall 05:17:02
扩展了
Darkfall 05:17:08
支持MAX_BOOST_PARAM个
Darkfall 05:17:10
默认是12
SC 05:17:11
不会是一个数量一个吧。。
SC 05:17:12
。。。。。。
Darkfall 05:17:18
我只山寨到了8
SC 05:17:26
其实够用了。。。
Darkfall 05:17:39
8个对于render4v这种函数都够了
Darkfall 05:17:42
哦，不对
Darkfall 05:17:44
是9
Darkfall 05:17:47
还有个this
SC 05:17:55
。。。
Darkfall 05:18:27
玩了boost之后我感觉我对模板的理解又上了一个层次。。。
SC 05:18:43
最近脑子有点动态语言。。。大不了
func({ arg1, arg2 ..., arg n })
Darkfall 05:18:48
function和type traits是看着boost代码解析弄的。。
Darkfall 05:19:00
memfun和bind是我自己山寨的。。

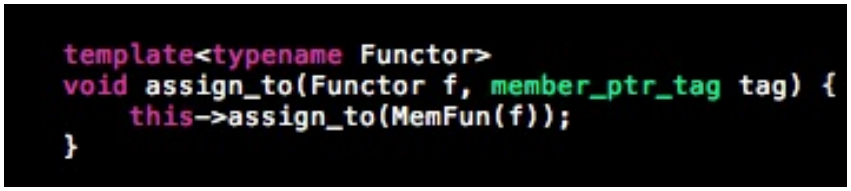
SC 05:19:07
memfun是嘛?
Darkfall 05:19:15
member_func模板
SC 05:19:30
。。看成内存函数 了
Darkfall 05:19:35
针对类成员函数的仿函数
Darkfall 05:19:42
类似的还有stl::ptr_fun
Darkfall 05:19:46
针对c函数
Darkfall 05:19:56
不过stl的都只支持2个参数
SC 05:20:23
不过一直不明白。。。
这样封装。。。的好处。。?
Darkfall 05:20:42
回调，代理，信号等等
SC 05:21:38
为什么不直接。。函数指针。。。
Darkfall 05:21:46
类成员函数要求类类型
Darkfall 05:22:08
也就是你必须手动申明是哪个类的，如果是函数指针
SC 05:22:24
soga。。
Darkfall 05:22:29
而这样一封装
Darkfall 05:22:36
我丝毫不关心是哪个类的
Darkfall 05:22:39
而仿函数
Darkfall 05:22:51
则是把一个任意函数保存为函数对象
Darkfall 05:22:55
我可以储存
Darkfall 05:23:03
可以operator =
Darkfall 05:25:01
没有仿函数有些模板算法都没法实现了
SC 05:25:03
有种你把boost搬进了sora的感觉 0 0
Darkfall 05:25:04
比如for_each之类的
Darkfall 05:25:11

第三个参数都是functor
Darkfall 05:25:24
=_ =学习以为
Darkfall 05:25:28
意味
SC 05:25:31
functor?
Darkfall 05:25:33
我山寨只是为了方便实现一些东西
Darkfall 05:25:37
functor(仿函数)
Darkfall 05:25:48
而又不想带上庞大的boost来编译
SC 05:26:13
so。。
Darkfall 05:26:15
SoraDelegate虽然支持任意返回值
Darkfall 05:26:23
但是只支持一个任意参数作为引用
Darkfall 05:26:37
用来实现signal总觉得有点囧
Darkfall 05:28:47
sorg核心中我山寨的那一部分。。
Darkfall 05:28:55
都可以抽出来一个htl了
Darkfall 05:28:59
hoshizora template library
Darkfall 05:29:01
哈哈哈
SC 05:29:12

Darkfall 05:29:13
其实本来想叫stl的。。
Darkfall 05:29:18
发现重名了= -=
SC 05:29:26
= =
Darkfall 05:29:43
这部分还是打算开源
SC 05:29:52
开源美
Darkfall 05:29:56
有时间了抽出来弄弄
Darkfall 05:30:06
虽然很渣。。
SC 05:30:09

轻量级模板库
SC 05:30:18
好用就行
SC 05:30:20
哈哈
Darkfall 05:30:28
恩
Darkfall 05:30:32
学习意味吧
Darkfall 05:30:41
没有Boost， stl那么复杂
Darkfall 05:30:49
懂template就能看懂
SC 05:30:59
啊。。template。。。
Darkfall 05:31:01
boost::function的实现实际上用了大量#define
Darkfall 05:31:10
一把来说你看他源代码。。
Darkfall 05:31:15
压根就不知道在干啥。。
Darkfall 05:31:25
他的全是宏生成类。。
SC 05:31:47
-~-
Darkfall 05:31:49
不过sora function不支持隐式类型转换=- =
Darkfall 05:31:53
boost可以。。
Darkfall 05:31:58
这点非常令人惊奇。。
SC 05:32:20
隐式类型转换。。。好耳熟的名字。。
Darkfall 05:32:25
因为不管你怎么玩，内部实际上还是函数指针
Darkfall 05:32:40
恩
Darkfall 05:33:02
也就是可以给一个function<float(int)>类型的函数对象附上 int(int)类型的函数。。
SC 05:33:17
哦。。 0 0.。怎么做到的。。
Darkfall 05:33:42
o
Darkfall 05:33:46
SoraFunction也可以
Darkfall 05:33:48

啊哈哈哈
Darkfall 05:33:52
我明白这中间的巧妙了
SC 05:33:55
0 0
Darkfall 05:34:00
sora::SoraFunction<void(int)> func2 = sora::Bind(&ins, &t::testFunc3);
func2(1);
Darkfall 05:34:09
function<void(int)>
Darkfall 05:34:10
void testFunc3(float i) {
std::cout<<i<<std::endl;
}
Darkfall 05:34:13
testFunc3, void(float)
Darkfall 05:34:25
原理就是封装
Darkfall 05:34:39
任何软件开发中的问题，都可以通过增加一个层次来解决
Darkfall 05:34:59
因为对于成员函数
SC 05:35:00
于是。。这个中间层是的样子?
Darkfall 05:35:28
SoraFunction内部实际上调用了SoraMemFun这个仿函数
Darkfall 05:35:42



Darkfall 05:35:50
这个是对于成员函数赋值的函数
Darkfall 05:36:05
Memfun是用于生成SoraMemfun的
Darkfall 05:36:30
这样实际上依据f的类型生成了针对f类型的仿函数
Darkfall 05:36:39
也就是SoraMemfun实际上是void(float)类型
Darkfall 05:36:51
但是他上边的SoraFunction是void(int)
Darkfall 05:37:09

这样SoraFunction的operator()(int)
Darkfall 05:37:19
实际上调用到了SoraMemfun的operator()(float)
Darkfall 05:37:34
也就是实际上是t->testFunc3((float)1);
Darkfall 05:37:46
编译器允许float->int的隐式转换
Darkfall 05:37:53
最多有个warning
SC 05:38:26
所以。。实际上他本来就能做到0 0.。?
Darkfall 05:38:30
恩
Darkfall 05:38:48
因为多了个Memfun蹭
Darkfall 05:38:49
层
Darkfall 05:38:56
编译器就知道怎么隐式转换了
Darkfall 05:39:33
我才知道任何c函数都可以强制转换为void (*)()
SC 05:39:42
但是，就算直接赋过来一个其他prototype的函数指针的时候，也能转换吧
SC 05:39:43
恩
Darkfall 05:40:00
不能
Darkfall 05:40:15
因为模板里你明确申明了函数是void(int)类型
Darkfall 05:40:21
无法匹配void(float)的函数指针
Darkfall 05:41:11
也就是你没办法直接给一个void (t::*)(float)的函数指针
SC 05:41:14
哇。。于是。。把赋值时候的转换移到了调用时 0 0.。
Darkfall 05:41:17
复制上void (t::*)(int)
Darkfall 05:41:19
恩。。
Darkfall 05:41:50
卧槽太巧妙了。。
Darkfall 05:41:56
一环套一环啊。
SC 05:42:06
真hack。。。
Darkfall 05:42:17

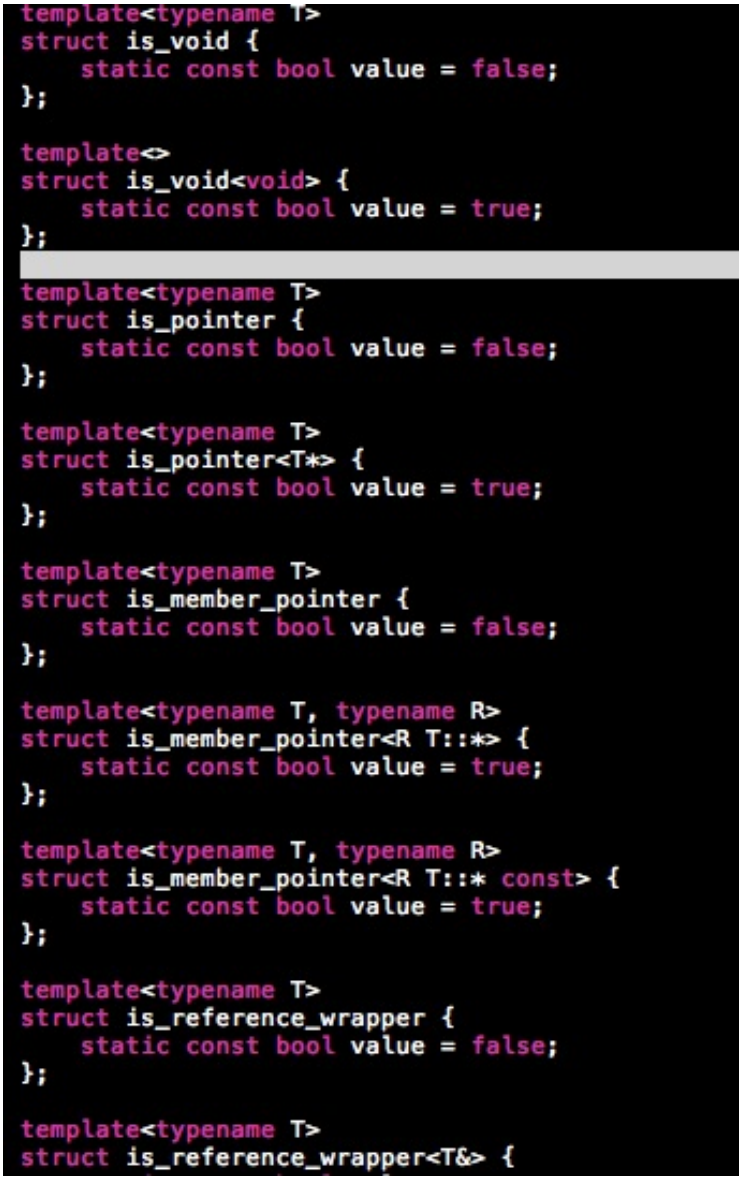
其实最hack的是type traits
SC 05:42:19
太凶逼了
Darkfall 05:42:23
因为可以operator =
Darkfall 05:42:48
你知道怎么判断右值是类成员函数，还是是C函数，还是是仿函数的么。。
Darkfall 05:43:02
因为针对这三个类型的函数要采取不同的赋值策略。。
Darkfall 05:43:26
C函数可以指针保存，但是仿函数涉及到内存了
Darkfall 05:43:38

```
};

struct function_ptr_tag {};
struct function_obj_tag {};
struct member_ptr_tag {};
struct function_obj_ref_tag {};

template<typename F>
class get_function_tag {
    typedef typename if_c<(is_pointer<F>::value),
    function_ptr_tag,
    function_obj_tag>::type ptr_or_obj_tag;

    typedef typename if_c<(is_member_pointer<F>::value),
    member_ptr_tag,
    ptr_or_obj_tag>::type ptr_or_obj_or_mem_tag;

    typedef typename if_c<(is_reference_wrapper<F>::value),
    function_obj_ref_tag,
    ptr_or_obj_or_mem_tag>::type or_ref_tag;

public:
    typedef or_ref_tag type;
};
```

Darkfall 05:43:41
这是最hack的部分..
Darkfall 05:43:47
函数类型判断。。。
SC 05:44:12
if_c是嘛 0 0.。。。。。。。。。
SC 05:44:17
模板?
SC 05:45:08
还有 is_pointer这样的玩意儿0 0.。。是嘛。。。
Darkfall 05:45:19

```cpp
template<bool C, typename T1, typename T2>
struct if_c {
    typedef T1 type;
};

template<typename T1, typename T2>
struct if_c<false,T1,T2> {
    typedef T2 type;
};
```

Darkfall 05:45:30
都是type traits
Darkfall 05:45:36
编译期模板
Darkfall 05:45:44
if_c接受三个模板参数
Darkfall 05:45:48
第一个为bool
Darkfall 05:45:57
如果为true, 则type为t1类型
Darkfall 05:46:05
如果为false, 则type为t2类型
Darkfall 05:46:17
第一个就可以解释为
Darkfall 05:46:22
如果f是指针
Darkfall 05:46:46
那么他是C函数指针或者类成员函数指针
Darkfall 05:47:11
否则他是函数对象
Darkfall 05:47:30
is_pointer那些其实很简单
Darkfall 05:47:33
模板偏特化
SC 05:47:56
我还在看 if_c鬼畜的代码 = =
Darkfall 05:47:56



```cpp
template<typename T>
struct is_void {
    static const bool value = false;
};

template<>
struct is_void<void> {
    static const bool value = true;
};

template<typename T>
struct is_pointer {
    static const bool value = false;
};

template<typename T>
struct is_pointer<T*> {
    static const bool value = true;
};

template<typename T>
struct is_member_pointer {
    static const bool value = false;
};

template<typename T, typename R>
struct is_member_pointer<R T::*> {
    static const bool value = true;
};

template<typename T, typename R>
struct is_member_pointer<R T::* const> {
    static const bool value = true;
};

template<typename T>
struct is_reference_wrapper {
    static const bool value = false;
};

template<typename T>
struct is_reference_wrapper<T&> {
```

SC 05:48:47
soga!
SC 05:48:49
我去!
SC 05:49:46
于是，编译期编译器就选择其中一个模板，然后结果也就定下来了 0 0.。
Darkfall 05:49:50
恩。。
Darkfall 05:49:58
和运行期毛事没有。
Darkfall 05:50:16

```
public:
    typedef or_ref_tag type;
```

Darkfall 05:50:23

```
struct function_ptr_tag {};
struct function_obj_tag {};
struct member_ptr_tag {};
struct function_obj_ref_tag {};
```

这样这个type就是
Darkfall 05:50:26
这四个中的一个
Darkfall 05:50:42
然后把这个type作为第二个参数，利用函数重载。。
Darkfall 05:50:56
在编译期就确定了到底调用哪个赋值函数。。
Darkfall 05:51:15

```
template<typename Functor>
void assign_to(Functor f) {
    typedef typename get_function_tag<Functor>::type tag;
    this->assign_to(f, tag());
}
```

Darkfall 05:51:22
asgin_to原型。
Darkfall 05:51:28
获取函数类型，转到冲在函数
Darkfall 05:51:31
*重载
Darkfall 05:51:40

```
template<typename Functor>
void assign_to(Functor f, function_ptr_tag tag) {
    if(f) {
        typedef typename get_function_invoker1<Functor, R

        mInvoker = &invoker_type::invoke;
        mPtr = make_any_pointer((void (*)())f);
    }
}

template<typename Functor>
void assign_to(Functor f, function_obj_tag tag) {
    if(f) {
        typedef typename get_function_obj_invoker1<Functo

        mInvoker = &invoker_type::invoke;
        mPtr = make_any_pointer(new Functor(f));
    }
}

template<typename Functor>
void assign_to(Functor f, member_ptr_tag tag) {
    this->assign_to(MemFun(f));
}
```

SC 05:51:53
0 0.。
SC 05:52:01
原理类似
Darkfall 05:52:36
不过我这里还没做内存管理
Darkfall 05:52:41

```
mPtr = make_any_pointer(new Functor(f));
```

Darkfall 05:52:44
这里有内存泄露
Darkfall 05:52:51
new之后不会被deleta
Darkfall 05:53:01
因为不管是成员函数还是c函数还是仿函数
Darkfall 05:53:06
底层公用一套储存方式
Darkfall 05:53:17
那个方式依旧是hack...
SC 05:53:30
。。
Darkfall 05:53:37

```
union any_ptr {
    void* obj_ptr;
    const void* const_obj_ptr;

    void (*func_ptr)();

    char data[1];
};
```

Darkfall 05:53:49
=_ =这就是function底层的储存方式
SC 05:54:03
。。。。。。
SC 05:54:17
member function呢?
Darkfall 05:54:35
之前提到过member function实际上储存在一个仿函数内
Darkfall 05:54:44
上边那个new就是申请那个仿函数的内存
Darkfall 05:54:48
然后保存在obj_ptr
SC 05:55:47
so 0 0
Darkfall 05:55:55
然后上边代码里有的。。实际上负责函数调用的invoker..
Darkfall 05:56:04
在获取到这个any_ptr的时候
Darkfall 05:56:10

```
namespace detail {

    template<typename FunctionPtr, typename R, typename T0>
    struct invoker1 {
        static R invoke(any_ptr ptr, T0 arg0) {
            FunctionPtr f = reinterpret_cast<FunctionPtr>(ptr.func_ptr);
            return f(arg0);
        }
    };

    template<typename FunctionObj, typename R, typename T0>
    struct obj_invoker1 {
        static R invoke(any_ptr ptr, T0 arg0) {
            FunctionObj* f = reinterpret_cast<FunctionObj*>(ptr.obj_ptr);
            return (*f)(arg0);
        }
    };
```

Darkfall 05:56:22

做了reinterpret_cast。。
SC 05:56:36
reinterpret_cast..忘了
Darkfall 05:56:38
对于c函数转到目标函数指针...
Darkfall 05:56:49
对于仿函数转到仿函数对象指针。。
Darkfall 05:57:02
然后一切就顺理成章了..
SC 05:58:03
0 0.。就这样全部静态化了。。。
Darkfall 05:58:03
function内部的Invoker。。实际上储存的是一个函数指针
SC 05:58:16
被催下去吃饭了 - -
Darkfall 05:58:17
利用模板特化获取到特定的invoker内部的invoke函数指针...
Darkfall 05:58:19
= =
Darkfall 05:58:26
聊天记录我留下来
Darkfall 05:58:33
作为教程恩恩
SC 05:58:35
恩
SC 05:58:37
。。。哈哈哈
Darkfall 05:58:53
免得时间一长我自己都忘了。。
Darkfall 06:07:28
所以function完全是依赖于模板偏特化。。。
Darkfall 06:07:37
而不是多态