

也谈 C++ 的虚函数表

秦新良

2015 年 1 月 11 日

目 录

1	缘起	2
2	虚函数表的初始化	2
3	结论	7

1 缘起

从C++ [虚函数表解析](#)这篇文章中我们可以了解到，C++ 中的虚函数是通过一张虚函数表来实现的。如图1所示，基类 *Base* 实例化后，其隐含的指针 `*__vptr` 指向了 *Base* 的虚函数表。

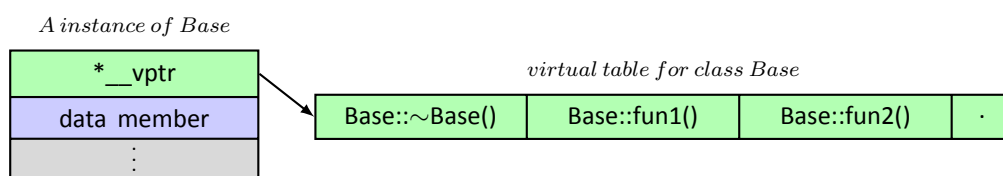


图 1: Base 的虚函数表

当基类 *Base* 被派生类 *D1* 继承并覆盖其虚函数 *fun1* 后，*D1* 实例化后的隐含指针 `*__vptr` 与其虚函数表的关系如图2所示。

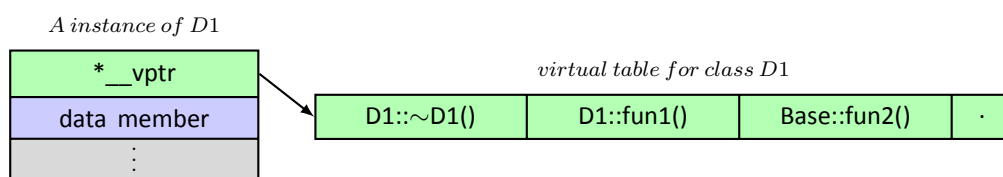


图 2: D1 的虚函数表

类实例化后，其隐含的指针 `*__vptr` 是何时指向了类的虚函数表，类的虚函数表又是在什么时候初始化的？本文着重阐述这两个问题。

2 虚函数表的初始化

类实例化后，其隐含的指针 `*__vptr` 是何时指向了类的虚函数表？答案是编译器¹编译时在类的构造函数中自动插入了代码来做这个初始化的工作，这一点可以从 *Base* 汇编后的构造函数中清楚的看到。*Base* 的定义如代码1所示。代码2为其编译后的汇编代码。

```
1 class Base
2 {
3     public:
```

¹本文验证使用的 GCC 版本为：gcc version 4.7.3 (Ubuntu/Linaro 4.7.3-1ubuntu1)

```

4     Base(){};
5     virtual ~Base(){};
6     virtual void fun1() {};
7     virtual void fun2() {};
8 };
9
10 class D1: public Base
11 {
12     public:
13     D1(){};
14     virtual ~D1(){};
15     virtual void fun1() {};
16 };
17
18 class D2: public Base
19 {
20     public:
21     D2(){};
22     virtual ~D2(){};
23     virtual void fun2() {};
24 };
25
26 int main(int argc, char *argv[])
27 {
28     D1 d1;
29     D2 d2;
30
31     return 0;
32 }

```

代码 1: 类的定义

```

1 Disassembly of section .text:
2
3 ;; Base::Base()
4 080485c2 <_ZN4BaseC1Ev>:
5     80485c2:    55                push    %ebp
6     80485c3:    89 e5            mov     %esp,%ebp
7     80485c5:    8b 45 08        mov     0x8(%ebp),%eax
8     80485c8:    c7 00 d8 87 04 08 movl    $0x80487d8,(%eax)
9     80485ce:    5d              pop     %ebp
10    80485cf:    c3              ret
11
12 ;; Base::~~Base()
13 080485d0 <_ZN4BaseD1Ev>:
14    80485d0:    55                push    %ebp
15    80485d1:    89 e5            mov     %esp,%ebp
16    80485d3:    83 ec 18        sub     $0x18,%esp
17    80485d6:    8b 45 08        mov     0x8(%ebp),%eax
18    80485d9:    c7 00 d8 87 04 08 movl    $0x80487d8,(%eax)
19    80485df:    b8 00 00 00 00  mov     $0x0,%eax
20    80485e4:    83 e0 01        and     $0x1,%eax

```

```

21 80485e7:      85 c0      test    %eax,%eax
22 80485e9:      74 0b      je      80485f6 <
    _ZN4BaseD1Ev+0x26>
23 80485eb:      8b 45 08    mov     0x8(%ebp),%eax
24 80485ee:      89 04 24    mov     %eax,(%esp)
25 80485f1:      e8 6a fe ff ff    call   8048460 <
    _ZdlPv@plt>
26 80485f6:      c9        leave
27 80485f7:      c3        ret
28
29 ;; Base::~~Base()
30 080485f8 <_ZN4BaseD0Ev>:
31 80485f8:      55        push    %ebp
32 80485f9:      89 e5      mov     %esp,%ebp
33 80485fb:      83 ec 18    sub     $0x18,%esp
34 80485fe:      8b 45 08    mov     0x8(%ebp),%eax
35 8048601:      89 04 24    mov     %eax,(%esp)
36 8048604:      e8 c7 ff ff ff    call   80485d0 <
    _ZN4BaseD1Ev>
37 8048609:      8b 45 08    mov     0x8(%ebp),%eax
38 804860c:      89 04 24    mov     %eax,(%esp)
39 804860f:      e8 4c fe ff ff    call   8048460 <
    _ZdlPv@plt>
40 8048614:      c9        leave
41 8048615:      c3        ret
42
43 ;; Base::fun1()
44 08048616 <_ZN4Base4fun1Ev>:
45 8048616:      55        push    %ebp
46 8048617:      89 e5      mov     %esp,%ebp
47 8048619:      5d        pop     %ebp
48 804861a:      c3        ret
49 804861b:      90        nop
50
51 ;; Base::fun2()
52 0804861c <_ZN4Base4fun2Ev>:
53 804861c:      55        push    %ebp
54 804861d:      89 e5      mov     %esp,%ebp
55 804861f:      5d        pop     %ebp
56 8048620:      c3        ret
57 8048621:      90        nop
58
59 ;; D1::D1()
60 08048622 <_ZN2D1C1Ev>:
61 8048622:      55        push    %ebp
62 8048623:      89 e5      mov     %esp,%ebp
63 8048625:      83 ec 18    sub     $0x18,%esp
64 8048628:      8b 45 08    mov     0x8(%ebp),%eax
65 804862b:      89 04 24    mov     %eax,(%esp)
66 804862e:      e8 8f ff ff ff    call   80485c2 <

```

```

_ZN4BaseC1Ev>
67 8048633:      8b 45 08      mov     0x8(%ebp),%eax
68 8048636:      c7 00 c0 87 04 08  movl    $0x80487c0,(%eax)
69 804863c:      c9              leave
70 804863d:      c3              ret
71
72 ;; D1::~~D1()
73 0804863e <_ZN2D1D1Ev>:
74 804863e:      55              push    %ebp
75 804863f:      89 e5           mov     %esp,%ebp
76 8048641:      83 ec 18        sub     $0x18,%esp
77 8048644:      8b 45 08        mov     0x8(%ebp),%eax
78 8048647:      c7 00 c0 87 04 08  movl    $0x80487c0,(%eax)
79 804864d:      8b 45 08        mov     0x8(%ebp),%eax
80 8048650:      89 04 24        mov     %eax,(%esp)
81 8048653:      e8 78 ff ff ff  call    80485d0 <
_ZN4BaseD1Ev>
82 8048658:      b8 00 00 00 00  mov     $0x0,%eax
83 804865d:      83 e0 01        and     $0x1,%eax
84 8048660:      85 c0           test    %eax,%eax
85 8048662:      74 0b           je      804866f <
_ZN2D1D1Ev+0x31>
86 8048664:      8b 45 08        mov     0x8(%ebp),%eax
87 8048667:      89 04 24        mov     %eax,(%esp)
88 804866a:      e8 f1 fd ff ff  call    8048460 <
_ZdlPv@plt>
89 804866f:      c9              leave
90 8048670:      c3              ret
91 8048671:      90              nop
92
93 ;; D1::~~D1()
94 08048672 <_ZN2D1D0Ev>:
95 8048672:      55              push    %ebp
96 8048673:      89 e5           mov     %esp,%ebp
97 8048675:      83 ec 18        sub     $0x18,%esp
98 8048678:      8b 45 08        mov     0x8(%ebp),%eax
99 804867b:      89 04 24        mov     %eax,(%esp)
100 804867e:      e8 bb ff ff ff  call    804863e <
_ZN2D1D1Ev>
101 8048683:      8b 45 08        mov     0x8(%ebp),%eax
102 8048686:      89 04 24        mov     %eax,(%esp)
103 8048689:      e8 d2 fd ff ff  call    8048460 <
_ZdlPv@plt>
104 804868e:      c9              leave
105 804868f:      c3              ret
106
107 ;; D1::fun1()
108 08048690 <_ZN2D14fun1Ev>:
109 8048690:      55              push    %ebp
110 8048691:      89 e5           mov     %esp,%ebp

```

```

111 8048693:      5d                pop     %ebp
112 8048694:      c3                ret
113 8048695:      90                nop

```

代码 2: Base 的汇编代码

汇编代码中的 `_ZN4BaseC1Ev` (4-9 行) 即为 `Base` 的构造函数。第 4 和 5 行是每个函数开始执行的例行公事: 保存上一个函数²调用栈的栈基址, 然后将本函数的栈基址保存到 `%ebp` 寄存器。第 6 行将构造函数的第一个入参³ (即 `*__vptr`) 存入寄存器 `%eax`, 紧接着把地址 `0x80487d8` (基类 `Base` 虚函数表的首地址) 赋给 `%eax` 所指的内存, 从而将 `*__vptr` 指向了类的虚函数表。同理可知, 派生类 `D1` 的虚函数表的首地址为 `0x80487c0`。

到这里我们清楚了类实例化后, `*__vptr` 是如何初始化为虚函数表的首地址的。但这样还不足以说明 `*__vptr` 确实是指向了类的虚函数表, 因为从上面的代码中根本看不出地址 `0x80487d8` 和 `0x80487c0` 就是虚函数表的地址。下面我们接着做进一步的验证。

通过 `objdump` 解析二进制文件中的只读数据段的内容如图 3 中的上半部分所示。解析后的只读数据段的首列是程序加载后的内存地址⁴, 紧接着后面 4 列是内存地址对应的值, 每行 16 个字节。从图中可以看出, 地址 `0x80487d8` 的值为 `d0850408`, 因为笔者的系统是小端字节序的, 转换过来就是 `080485d0`。从代码 2 中可以找出 `080485d0` 即为函数 `Base::~Base()` 的地址, 即图 3 中以红色椭圆标识部分。依次类推, 地址 `0x80487dc` 的值为 `f8850408`, 对应函数 `Base::~Base()`⁵; 地址 `0x80487e0` 的值为 `16860408`, 对应函数 `Base::fun1()`; 地址 `0x80487e4` 的值为 `1c860408`, 对应函数 `Base::fun2()`; 这样就构成了类 `Base` 的整张虚函数表。相信你也可以用样的方法推出类 `D1` 的虚函数表。

²调用该函数的函数, 即调用者 (caller)。

³类的成员函数的第一个入参为 `this` 指针。

⁴程序运行过程中的逻辑地址。

⁵汇编代码中有两个析构函数。

```
vampire@vampire:~/Ubuntu One$ objdump -s -j .rodata vtable
vtable:          file format elf32-i386
80487d8 : 类Base虚函数表的地址    d0850408 : 虚函数表中第一个函数的地址
Contents of section .rodata:
8048798 03000000 01000200 00000000 ec870408 .....
80487a8 b2860408 e6860408 16860408 04870408 .....
80487b8 00000000 fc870408 3e860408 72860408 .....>...r...
80487c8 90860408 1c860408 00000000 10880408 .....
80487d8 d0850408 f8850408 16860408 1c860408 .....
80487e8 32443200 68a00408 e8870408 10880408 2D2.h.....
80487f8 32443100 68a00408 f8870408 10880408 2D1.h.....
8048808 34426173 65000000 28a00408 08880408 4Base...(.....
vampire@vampire:~/Ubuntu One$
080485d0 : 函数Base::~Base()的地址

80485cf:      c3                ret
080485d0 <ZN4BaseD1Ev>:
80485d0:      55                push    %ebp
80485d1:      89 e5             mov     %esp,%ebp
80485d3:      83 ec 18          sub     $0x18,%esp
80485d6:      8b 45 08          mov     0x8(%ebp),%eax
80485d9:      c7 00 d8 87 04 08 movl    $0x80487d8,(%eax)
80485df:      b8 00 00 00 00    mov     $0x0,%eax
80485e4:      83 e0 01          and     $0x1,%eax
80485e7:      85 c0             test    %eax,%eax
80485e9:      74 0b             je      80485f6 <ZN4BaseD1Ev+0x26>
80485eb:      8b 45 08          mov     0x8(%ebp),%eax
80485ee:      89 04 24          mov     %eax,(%esp)
80485f1:      e8 6a fe ff ff    call    8048460 <_ZdlPv@plt>
80485f6:      c9                leave
80485f7:      c3                ret
```

图 3: 虚函数表及虚函数

3 结论

类的虚函数表在代码编译链接后即已确定，位于编译后二进制文件中的只读数据段；并且编译器会自动在类的构造函数中插入将 `*__vptr` 指向虚函数表的代码，从而实现类在实例化时自动将 `*__vptr` 赋值为类的虚函数表的首地址。