**C++ 虚函数表解析**

**陈皓**

<http://blog.csdn.net/haoel>

**前言**

C++中的虚函数的作用主要是实现了多态的机制。关于多态，简而言之就是用父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。这种技术可以让父类的指针有“多种形态”，这是一种泛型技术。所谓泛型技术，说白了就是试图使用不变的代码来实现可变的算法。比如：模板技术，RTTI技术，虚函数技术，要么是试图做到在编译时决议，要么试图做到运行时决议。

关于虚函数的使用方法，我在这里不做过多的阐述。大家可以看看相关的C++的书籍。在这篇文章中，我只想从虚函数的实现机制上面为大家 一个清晰的剖析。

当然，相同的文章在网上也出现过一些了，但我总感觉这些文章不是很容易阅读，大段大段的代码，没有图片，没有详细的说明，没有比较，没有举一反三。不利于学习和阅读，所以这是我想写下这篇文章的原因。也希望大家多给我提意见。

言归正传，让我们一起进入虚函数的世界。

**虚函数表**

对C++ 了解的人都应该知道虚函数（Virtual Function）是通过一张虚函数表（Virtual Table）来实现的。简称为V-Table。在这个表中，主是要一个类的虚函数的地址表，这张表解决了继承、覆盖的问题，保证其容真实反应实际的函数。这样，在有虚函数的类的实例中这个表被分配在了这个实例的内存中，所以，当我们用父类的指针来操作一个子类的时候，这张虚函数表就显得由为重要了，它就像一个地图一样，指明了实际所应该调用的函数。

这里我们着重看一下这张虚函数表。C++的编译器应该是保证虚函数表的指针存在于对象实例中最前面的位置（这是为了保证取到虚函数表的有最高的性能——如果有多层继承或是多重继承的情况下）。 这意味着我们通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。

听我扯了那么多，我可以感觉出来你现在可能比以前更加晕头转向了。 没关系，下面就是实际的例子，相信聪明的你一看就明白了。

假设我们有这样的一个类：

class Base {

public:

virtual void f() { cout << "Base::f" << endl; }

virtual void g() { cout << "Base::g" << endl; }

virtual void h() { cout << "Base::h" << endl; }

};

按照上面的说法，我们可以通过Base的实例来得到虚函数表。 下面是实际例程：

typedef void(\*Fun)(void);

Base b;

Fun pFun = NULL;

cout << "虚函数表地址：" << (int\*)(&b) << endl;

cout << "虚函数表 — 第一个函数地址：" << (int\*)\*(int\*)(&b) << endl;

// Invoke the first virtual function

pFun = (Fun)\*((int\*)\*(int\*)(&b));

pFun();

实际运行经果如下：(Windows XP+VS2003, Linux 2.6.22 + GCC 4.1.3)

虚函数表地址：0012FED4

虚函数表 — 第一个函数地址：0044F148

Base::f

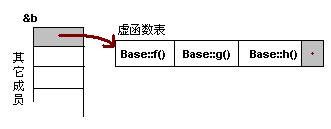
通过这个示例，我们可以看到，我们可以通过强行把&b转成int \*，取得虚函数表的地址，然后，再次取址就可以得到第一个虚函数的地址了，也就是Base::f()，这在上面的程序中得到了验证（把int\* 强制转成了函数指针）。通过这个示例，我们就可以知道如果要调用Base::g()和Base::h()，其代码如下：

(Fun)\*((int\*)\*(int\*)(&b)+0); // Base::f()

(Fun)\*((int\*)\*(int\*)(&b)+1); // Base::g()

(Fun)\*((int\*)\*(int\*)(&b)+2); // Base::h()

这个时候你应该懂了吧。什么？还是有点晕。也是，这样的代码看着太乱了。没问题，让我画个图解释一下。如下所示：

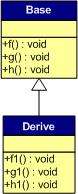


注意：在上面这个图中，我在虚函数表的最后多加了一个结点，这是虚函数表的结束结点，就像字符串的结束符“/0”一样，其标志了虚函数表的结束。这个结束标志的值在不同的编译器下是不同的。在WinXP+VS2003下，这个值是NULL。而在Ubuntu 7.10 + Linux 2.6.22 + GCC 4.1.3下，这个值是如果1，表示还有下一个虚函数表，如果值是0，表示是最后一个虚函数表。

下面，我将分别说明“无覆盖”和“有覆盖”时的虚函数表的样子。没有覆盖父类的虚函数是毫无意义的。我之所以要讲述没有覆盖的情况，主要目的是为了给一个对比。在比较之下，我们可以更加清楚地知道其内部的具体实现。

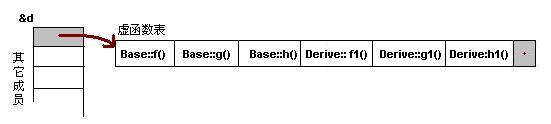
**一般继承（无虚函数覆盖）**

下面，再让我们来看看继承时的虚函数表是什么样的。假设有如下所示的一个继承关系：



请注意，在这个继承关系中，子类没有重载任何父类的函数。那么，在派生类的实例中，其虚函数表如下所示：

对于实例：Derive d; 的虚函数表如下：



我们可以看到下面几点：

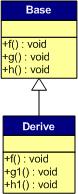
1）虚函数按照其声明顺序放于表中。

2）父类的虚函数在子类的虚函数前面。

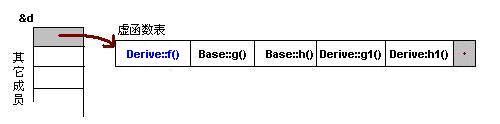
我相信聪明的你一定可以参考前面的那个程序，来编写一段程序来验证。

**一般继承（有虚函数覆盖）**

覆盖父类的虚函数是很显然的事情，不然，虚函数就变得毫无意义。下面，我们来看一下，如果子类中有虚函数重载了父类的虚函数，会是一个什么样子？假设，我们有下面这样的一个继承关系。



为了让大家看到被继承过后的效果，在这个类的设计中，我只覆盖了父类的一个函数：f()。那么，对于派生类的实例，其虚函数表会是下面的一个样子：



我们从表中可以看到下面几点，

1）覆盖的f()函数被放到了虚表中原来父类虚函数的位置。

2）没有被覆盖的函数依旧。

这样，我们就可以看到对于下面这样的程序，

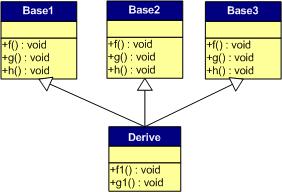
Base \*b = new Derive();

b->f();

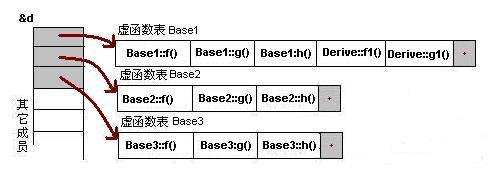
由b所指的内存中的虚函数表的f()的位置已经被Derive::f()函数地址所取代，于是在实际调用发生时，是Derive::f()被调用了。这就实现了多态。

**多重继承（无虚函数覆盖）**

下面，再让我们来看看多重继承中的情况，假设有下面这样一个类的继承关系。注意：子类并没有覆盖父类的函数。



对于子类实例中的虚函数表，是下面这个样子：



我们可以看到：

1） 每个父类都有自己的虚表。

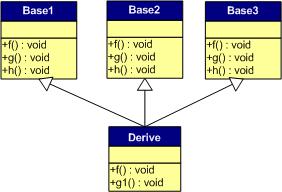
2） 子类的成员函数被放到了第一个父类的表中。（所谓的第一个父类是按照声明顺序来判断的）

这样做就是为了解决不同的父类类型的指针指向同一个子类实例，而能够调用到实际的函数。

**多重继承（有虚函数覆盖）**

下面我们再来看看，如果发生虚函数覆盖的情况。

下图中，我们在子类中覆盖了父类的f()函数。



下面是对于子类实例中的虚函数表的图：



我们可以看见，三个父类虚函数表中的f()的位置被替换成了子类的函数指针。这样，我们就可以任一静态类型的父类来指向子类，并调用子类的f()了。如：

Derive d;

Base1 \*b1 = &d;

Base2 \*b2 = &d;

Base3 \*b3 = &d;

b1->f(); //Derive::f()

b2->f(); //Derive::f()

b3->f(); //Derive::f()

b1->g(); //Base1::g()

b2->g(); //Base2::g()

b3->g(); //Base3::g()