C++ 虚函数表及多态内部原理详解

高效程序员 2021-06-18 08:46

收录于合集
#C++
103个

亲~赶快动动手指 点击关注哦~~

置顶/星标公众号👇，硬核文章第一时间送达！

高效程序员
聚焦程序人生，践行终身成长。专注分享 IT 技术「C++/Python/Linux/Qt 等」、学习资料、职场经验、热点资讯，有趣、好玩、靠谱！（关注回复 …
181篇原创内容

公众号

链接 | https://blog.csdn.net/songguangfan/article/details/87898915

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。

虚函数表

每个含有虚函数的类都有一个虚函数表（Virtual Table）来实现的。简称为V-Table。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。

1、 每一个类都有虚函数列表。

2、 虚表可以继承，如果子类没有重写虚函数，那么子类虚表中仍然会有该函数的地址，只不过这个地址指向的是基类的虚函数实现。如果基类3个虚函数，那么基类的虚表中就有三项（虚函数地址），派生类也会有虚表，至少有三项，如果重写了相应的虚函数，那么虚表中的地址就会改变，指向自身的虚函数实现。如果派生类有自己的虚函数，那么虚表中就会添加该项。

3、 派生类的虚表中虚函数地址的排列顺序和基类的虚表中虚函数地址排列顺序相同，子类独有的虚函数放在后面。

当定义一个有虚函数类的对象时，对象的第一块的内存空间就是一个指向虚函数列表的指针。

在这举个例子

假设我们有这样的一个类：

```
class Base
{
public:
    virtual void f(){std::cout<<"Base::f"<<std::endl;}
    virtual void g(){std::cout<<"Base::g"<<std::endl;}
    virtual void h(){std::cout<<"Base::h"<<std::endl;}
};
typedef  void(*Fun)(void);


int main()
{
    Base b;
    Fun pFun = NULL;
    std::cout << (long*)(&b) << std::endl;
    pFun = (Fun)*((long*)*(long*)(&b));
    pFun();
    pFun = (Fun)*((long*)*(long*)(&b)+1);
    pFun();
    pFun = (Fun)*((long*)*(long*)(&b)+2);
    pFun();
    return 0;
}
```

https://blog.csdn.net/songguangfan

由于例程的操作环境是64位系统，所以用long*强转。其中(long*)(&b)就是虚函数表地址，(long*)*(long*)(&b)就是第一个函数地址，代码运行结果如下：

```
(base) root@van-ThinkPad-T490:/home/van/practise# ./a.out
0x7ffd107f2788
Base::f
Base::g
Base::h
```

在程序中取出对象b的地址，根据对象的布局可以得出就是虚表的地址，根据这个地址可以把虚表的第一个内存单元的内容取出，然后强制转换成一个函数指针，利用这个函数指针来访问虚函数。又因为虚表是连续的，利用每次+1可以来访问下一个内存单元。

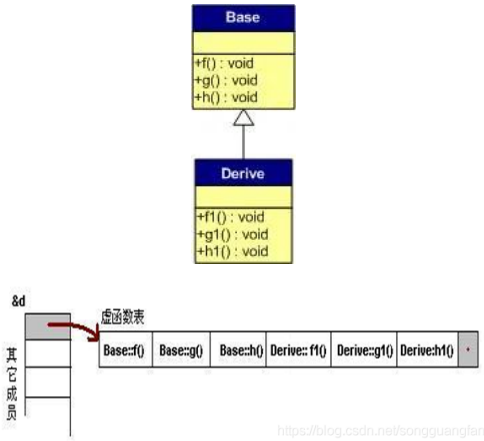
如果对代码不理解的话，可以看这幅图就会懂了

注意：虚函数表在最后会有一个结束标志，为1说明还有虚表，为0表示没有虚表了 。（编译器不同，结束标志可能存在差异）

下面，将分别具体说明“无虚函数覆盖”和“有虚函数覆盖”时的虚函数表的情况。

（一）无虚函数覆盖

没有任何的继承，虚函数表如下图



根据示意图，编写的代码如下图所示：

```cpp
#include <iostream>

class Base
{
public:
    virtual void f(){std::cout<<"Base::f"<<std::endl;}
    virtual void g(){std::cout<<"Base::g"<<std::endl;}
    virtual void h(){std::cout<<"Base::h"<<std::endl;}
};


class Derived:public Base
{
public:
    virtual void f1(){std::cout<<"Base::f1"<<std::endl;}
    virtual void g1(){std::cout<<"Base::g1"<<std::endl;}
    virtual void h1(){std::cout<<"Base::h1"<<std::endl;}
};



typedef  void(*Fun)(void);


int main()
{
    Derived d;
    Fun pFun = NULL;
    std::cout << (long*)(&d) << std::endl;
    pFun = (Fun)*((long*)*(long*)(&d));
    pFun();
    pFun = (Fun)*((long*)*(long*)(&d)+1);
    pFun();
    pFun = (Fun)*((long*)*(long*)(&d)+2);
    pFun();
    pFun = (Fun)*((long*)*(long*)(&d)+3);
    pFun();
    pFun = (Fun)*((long*)*(long*)(&d)+4);
    pFun();
    pFun = (Fun)*((long*)*(long*)(&d)+5);
    pFun();
    return 0;
}
```

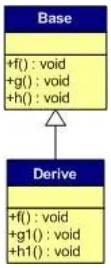
https://blog.csdn.net/songguangfan

和上一个程序一样，根据取出虚表里面的地址强制转换成函数指针，同样，利用虚表的连续性，每次指针+1调用对应的虚函数。可以得出虚函数按照其声明顺序存放于虚函数表中的，子类自己的虚函数是排在父类虚函数之后的。运行结果如下图

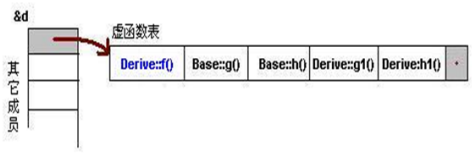
```
0x7ffd3b9e8438
Base::f
Base::g
Base::h
Base::f1
Base::g1
Base::h1
```

（二）一般继承（有虚函数覆盖）

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。如图所示：



在这个类的设计中，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：



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。

这样就会出现虚调用

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。下面我们用一个示例代码来看一下

```
class Base
{
public:
    virtual void f(){std::cout<<"Base::f"<<std::endl;}
    virtual void g(){std::cout<<"Base::g"<<std::endl;}
    virtual void h(){std::cout<<"Base::h"<<std::endl;}
};


class Derived:public Base
{
public:
    virtual void f(){std::cout<<"Derived::f"<<std::endl;}
    virtual void g1(){std::cout<<"Derived::g1"<<std::endl;}
    virtual void h1(){std::cout<<"Derived::h1"<<std::endl;}
};



typedef  void(*Fun)(void);


int main()
{
    Base *b = new Derived;
    Fun pFun = NULL;
    std::cout << (long*)(b) << std::endl;
    pFun = (Fun)*((long*)*(long*)(b));
    pFun();
    pFun = (Fun)*((long*)*(long*)(b)+1);
    pFun();
    pFun = (Fun)*((long*)*(long*)(b)+2);
    pFun();
    pFun = (Fun)*((long*)*(long*)(b)+3);
    pFun();
    pFun = (Fun)*((long*)*(long*)(b)+4);
    pFun();
    return 0;
}
```

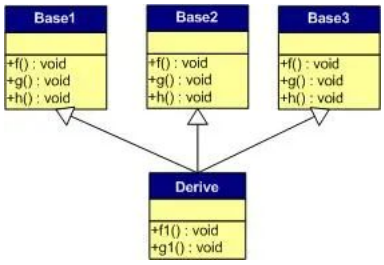
https://blog.csdn.net/songguangfan

运行结果如下,确实如我们以上分析的那样，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：

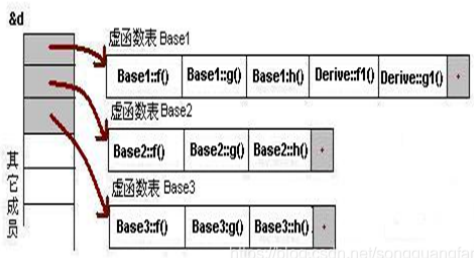
```
0x556aeb00feb0
Derived::f
Base::g
Base::h
Derived::g1
Derived::h1
```

（三）多重继承（无虚函数覆盖）

下面我们再看看多重继承的情况



对于子类实例中的虚函数表，是下面这个样子：



从图上我们可以看到

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。

下面我们根据上图来实现一下

```cpp
#include <iostream>

class Base1
{
public:
    virtual void f(){std::cout<<"Base1::f"<<std::endl;}
    virtual void g(){std::cout<<"Base1::g"<<std::endl;}
    virtual void h(){std::cout<<"Base1::h"<<std::endl;}
};



class Base2
{
public:
    virtual void f(){std::cout<<"Base2::f"<<std::endl;}
    virtual void g(){std::cout<<"Base2::g"<<std::endl;}
    virtual void h(){std::cout<<"Base2::h"<<std::endl;}
};

class Base3
{
public:
    virtual void f(){std::cout<<"Base3::f"<<std::endl;}
    virtual void g(){std::cout<<"Base3::g"<<std::endl;}
    virtual void h(){std::cout<<"Base3::h"<<std::endl;}
};



class Derived:public Base1, Base2, Base3
{
public:
    virtual void f1(){std::cout<<"Derived::f1"<<std::endl;}
    virtual void g1(){std::cout<<"Derived::g1"<<std::endl;}
};
```

https://blog.csdn.net/songguangfan

```cpp
41
42 typedef  void(*Fun)(void);
43
44
45 int main()
46 {
47     Derived d;
48     Fun pFun = NULL;
49     long** pvtab=(long**)&d;
50     pFun = (Fun)pvtab[0][1];
51     pFun();
52     pFun = (Fun)pvtab[1][1];
53     pFun();
54     pFun = (Fun)pvtab[2][1];
55     pFun();
56     pFun = (Fun)pvtab[0][3];
57     pFun();
58     pFun = (Fun)pvtab[0][4];
59     pFun();
60     return 0;
61 }
```

https://blog.csdn.net/songguangfan

运行结果如下：

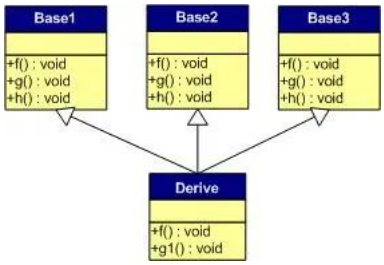
```
Base1::g
Base2::g
Base3::g
Derived::f1
Derived::g1
```

在这个程序中，子类有多个父类，因此从每个父类都继承了一个虚表，因此会有3个虚表，根据代码和运行结果会发现，排列的顺序和继承的顺序一样，子类自己的虚函数排在第一个虚表的后面。程序中没有改写虚函数 ，因此没有覆盖。同时主函数中应用的是一个二重指针，利用二维数组取每个虚函数地址。

（四）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。



子类虚函数列表如图所示

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。

子类虚函数列表访问代码如下：

```
#include <iostream>

class Base1
{
public:
    virtual void f(){std::cout<<"Base1::f"<<std::endl;}
    virtual void g(){std::cout<<"Base1::g"<<std::endl;}
    virtual void h(){std::cout<<"Base1::h"<<std::endl;}
};



class Base2
{
public:
    virtual void f(){std::cout<<"Base2::f"<<std::endl;}
    virtual void g(){std::cout<<"Base2::g"<<std::endl;}
    virtual void h(){std::cout<<"Base2::h"<<std::endl;}
};

class Base3
{
public:
    virtual void f(){std::cout<<"Base3::f"<<std::endl;}
    virtual void g(){std::cout<<"Base3::g"<<std::endl;}
    virtual void h(){std::cout<<"Base3::h"<<std::endl;}
};



class Derived:public Base1, Base2, Base3
{
public:
    virtual void f(){std::cout<<"Derived::f"<<std::endl;}
    virtual void g1(){std::cout<<"Derived::g1"<<std::endl;}
};
```

https://blog.csdn.net/songguangfan

```
typedef  void(*Fun)(void);


int main()
{
    Derived d;
    Fun pFun = NULL;
    long** pvtab=(long**)&d;
    pFun = (Fun)pvtab[0][0];
    pFun();
    pFun = (Fun)pvtab[1][0];
    pFun();
    pFun = (Fun)pvtab[2][0];
    pFun();
    pFun = (Fun)pvtab[0][3];
    pFun();
    return 0;
}
```

https://blog.csdn.net/songguangfan

程序运行结果如下所示：

```
Derived::f
Derived::f
Derived::f
Derived::g1
```