C++---静态多态与动态多态

原创 Jammm 2018-08-02 18:32:37 9836 收藏 40 版权

分类专栏： C++ 文章标签： 多态 函数重载 函数模板 虚函数 纯虚函数

多态

多态按字面的意思就是多种形态，相同的方法调用，但是有不同的实现方式。多态性可以简单地概括为“一个接口，多种方法”。
C++有两种多态形式：

- 静态多态
- 动态多态

静态多态

静态多态：也称为编译期间的多态，编译器在编译期间完成的，编译器根据函数实参的类型(可能会进行隐式类型转换)，可推断出要调用那个函数，如果有对应的函数就调用该函数，否则出现编译错误。
静态多态有两种实现方式：

- 函数重载：包括普通函数的重载和成员函数的重载
- 函数模板的使用

函数重载

函数重载就像是有多种含义的动词。例如：你可以在棒球场为球队助威（root），也可以在地里种植（root）菌类植物。
根据上下文可以知道每种情况下，root的含义是什么，同样，C++中也通过上下文来确定同名函数的重载版本。
重载函数的关键是函数参数列表——也称函数特征标。包括：函数的参数数目和类型，以及参数的排列顺序。所以，重载函数与返回值，参数名无关。

```
// print()函数
void print(const char* str,int width);
void print(double i ,int width);
void print(const char* str);
// 使用print()函数时，编译器将根据所采取的用法使用有相应特征标的原型
print("abc",12);
print(2.2,55);
print("def");
```

以下这种方式的重载是错误的

```
void print(const char* str,int width);
int print(const char* str,int width);
```

重载时返回值可以不同，但特征标也必须不同。

为什么C语言中没有重载呢?

编译器在编译期间创建的一个字符串，用来指明函数的定义或原型。C和C++程序的函数在内部使用不同的名字修饰方式。
C编译器的函数名修饰规则：
对于__stdcall调用约定，编译器和链接器会在输出函数名前加上一个下划线前缀，函数名后面加上一个“@”符号和其参数的字节数，例如_functionname@number。__cdecl调用约定仅在输出函数名前加上一个下划线前缀，例如_functionname。__fastcall调用约定在输出函数名前加上一个“@”符号，后面也是一个“@”符号和其参数的字节数，例如

@functionname@number。
C++编译器的函数名修饰规则：
C++的函数名修饰规则有些复杂，但是信息更充分，通过分析修饰名不仅能够知道函数的调用方式，返回值类型，参数个数甚至参数类型。不管__cdecl，__fastcall还是__stdcall调用方式，函数修饰都是以一个“?”开始，后面紧跟函数的名字，再后面是参数表的开始标识和 按照参数类型代号拼出的参数表。对于__stdcall方式，参数表的开始标识是“@@YG”，对于__cdecl方式则是“@@YA”，对于__fastcall方式则是“@@YI”。参数表的拼写代号如下所示：
X–void
D–char
E–unsigned char
F–short
H–int
I–unsigned int
J–long
K–unsigned long（DWORD）
M–float
N–double
_N–bool
U–struct
....
所以，C++编译器能识别函数特征标的不同，从而实现重载。

函数模板

函数模板是通用的函数描述，也就是说，使用泛型来定义函数，其中泛型可用具体的类型（int 、double等）替换。通过将类型作为参数，传递给模板，可使编译器生成该类型的函数。

```
// 交换两个值，但是不清楚是int 还是 double，如果不使用模板，则要写两份代码
// 使用函数模板，将类型作为参数传递
template<class T>
class Swa(T a,T b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
};
```

动态多态

动态多态（动态绑定）：即运行时的多态，在程序执行期间(非编译期)判断所引用对象的实际类型，根据其实际类型调用相应的方法。

动态绑定

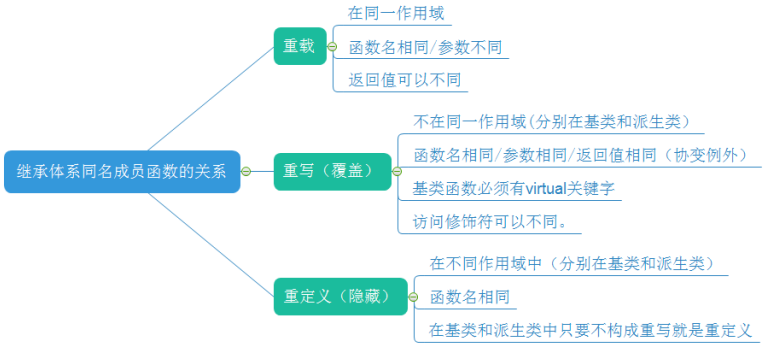
1.通过基类类型的引用或者指针调用虚函数
首先搞清楚这个对象的类型：

- 静态类型：对象声明时的类型，编译时确定
- 动态类型：目前所指对象的类型，运行时确定

```
class CDerived1:public CBase
{};

class CDerived2:public CBase
{};

int main()
{
    CDerived1* pD1 = new CDerived1;
    CBase* pBase = pD1;
    CDerived2* pD2 = new CDerived2;
    pBase = pD2;


    return 0;
}
```

2.必须是虚函数（派生类一定要重写基类中的虚函数）

```
class Base
{
public :
    virtual void FunTest1( int _iTest){cout <<"Base::FunTest1()" << endl;}
};
class Derived : public Base
{
public :
    void FunTest1( int _iTest){cout <<"Derived ::FunTest1()" << endl;}
}
```



纯虚函数

纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0”。
包含纯虚函数的类叫做**抽象类**（也叫接口类），抽象类不能实例化出对象。
纯虚函数在派生类中重新定义以后，派生类才能实例化出对象。

```
class a
{
public:
    virtual fun1();
    virtual fun2();
    .
    .
    .
    virtual …;
};

class b : public a
{
    fun1(){…;}
    fun(){…;}
    …
};
```

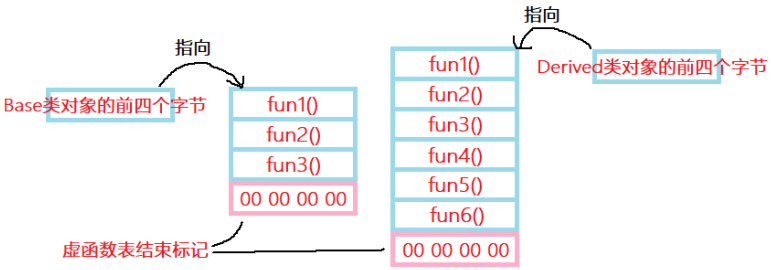
```
class c : public a
{
    …;
};
```

如果有很多类都继承了这个基类，那么每个对象中都要为创建基类消耗资源，此时出现了虚函数表。

虚函数表

对于有虚函数的类，编译器都会维护一张虚函数表(虚表)，对象的前四个字节就是指向虚表的指针(虚表指针)。
虚函数表的创建分为两种情况：
无覆盖
基类中虚函数在派生类中不是虚函数

```
class Base
{
    virtual void fun1();
    virtual void fun2();
    virtual void fun3();
}
class Derived : public Base
{
    virtual void fun4();
    virtual void fun5();
    virtual void fun6();
}
```

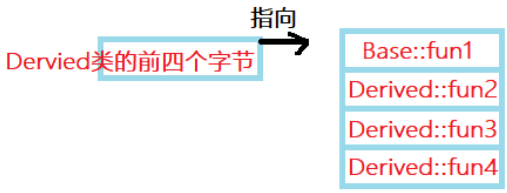


- 虚函数按声明顺序存在虚表中
- 在派生类中，前面是基类的虚函数，后面是派生类的虚函数

有覆盖

```
class Base
{
    virtual void fun1();
    virtual void fun2();
    virtual void fun3();
}
class Derived : public Base
{
    virtual void fun2();
    virtual void fun3();
    virtual void fun4();
}
```

基类的虚函数表没有变化



- 先拷贝基类的虚函数表
- 如果派生类重写了基类的某个虚函数，就用派生类的虚函数替换虚表同位置的基类虚函数
- 跟上派生类自己的虚函数

通过基类的引用或指针调用，，调用基类还是派生类的虚函数，要根据运行时根据指针或引用实际指向或引用的类型确定，调用非虚函数时，则无论基类指向的是何种类型，都调用基类的函数