C++中的面向对象多态实现

王祯

背景：虚函数是C++多态的核心机制

虚函数是实现多态的主要方式。

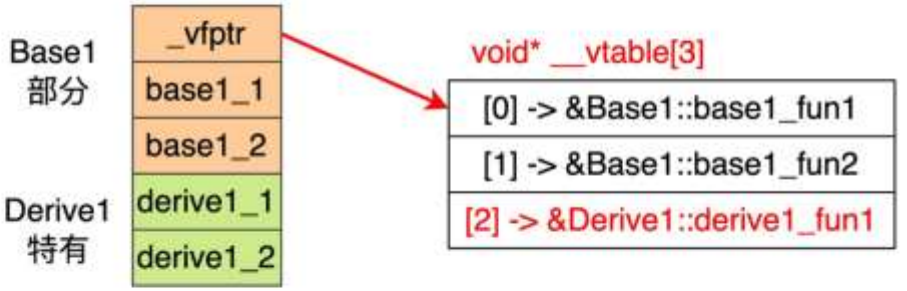
程序员需要深入理解虚函数的实现机制

```cpp
struct printer {
    virtual ~printer() = default;
    virtual void print() const = 0;
};

struct impl_A : printer {
    void print() const override {
        std::cout << "Hello world A!" << std::endl;
    }
};

struct impl_B : printer {
    void print() const override {
        std::cout << "Hello world B!" << std::endl;
    }
};

int main() {
    printer* ptrA = new impl_A();
    printer* ptrB = new impl_B();
    ptrA->print();
    ptrB->print();
    delete ptrA;
    delete ptrB;
    return 0;
}
```

虚函数实现过于复杂，程序员难以理解

为什么要这么做？

2. 避免静态绑定，在使用父类指针或引用调用子类对象的成员函数时，如果没有使用虚函数，则会进行静态绑定（Static Binding），即只能调用父类的成员函数，无法调用子类特有的成员函数。
3. 虚函数的调用是动态绑定*（Dynamic Binding）的，即在运行时根据指针或引用所指向的对象类型来选择调用哪个函数，从而实现动态多态性。
4. 抽象类是一种不能直接实例化的类，只能被其他类继承并实现其虚函数。通过定义纯虚函数*（Pure Virtual Function），可以使得一个类成为抽象类，强制其子类必须实现该函数。



二、实现原理

在 C++ 中，虚函数的实现原理基于两个关键概念：虚函数表*和虚函数指针*。

- 虚函数表：每个包含虚函数的类都会生成一个虚函数表（Virtual Table），其中存储着该类中所有虚函数的地址。虚函数表是一个由指针构成的数组，每个指针指向一个虚函数的

本报告论证：
C++ 的类多态本质是把对象和虚函数表相关联

发生了什么？

一个朴素的猜测：虚函数的实现被存在了对象之中。
或者其函数指针被存在了对象中。

```cpp
struct printer {
    virtual ~printer() = default;
    virtual void print() const = 0;
};

struct impl_A : printer {
    void print() const override {
        std::cout << "Hello world A!" << std::endl;
    }
};

struct impl_B : printer {
    void print() const override {
        std::cout << "Hello world B!" << std::endl;
    }
};

int main() {
    printer* ptrA = new impl_A();
    printer* ptrB = new impl_B();
    ptrA->print();
    ptrB->print();
    delete ptrA;
    delete ptrB;
    return 0;
}
```

```
wsDebugLauncher.exe
crosoft-MIEngine-En
interpreter=mi'
Hello world A!
Hello world B!
```

进行验证：

改变虚函数的个数，对象的大小不变！

排除之前的可能性。

```cpp
struct printer {
    virtual ~printer() = default;
    virtual void print() const = 0;
    virtual void print_1() const = 0;
};

struct impl_A : printer {
    void print() const override { std::cout << "Hello world A!" << std::endl; }
    void print_1() const override { std::cout << "Hello world A1!" << std::endl; }
};

struct simple_printer {
    virtual ~simple_printer() = default;
    virtual void print() const = 0;
};

struct impl_B : simple_printer {
    void print() const override { std::cout << "Hello world A!" << std::endl; }
};

int main() {
    printer* ptrA = new impl_A();
    simple_printer* ptrB = new impl_B();
    cout << sizeof(*ptrA) << endl;
    cout << sizeof(*ptrB) << endl;
    delete ptrA;
    delete ptrB;
    return 0;
}
```

```
PS C:\MyFiles\Rep
wsDebugLauncher.e
crosoft-MIEngine-
interpreter=mi'
8
8
```

虚函数到底在哪里?

下面，从编译器的角度，用汇编来探究虚函数的位置，以及调用虚函数时，到底发生了什么。

查看汇编

Clang x86-64编译

```
impl_B::impl_B() [base object constructor]:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     qword ptr [rbp - 8], rdi
        mov     rdi, qword ptr [rbp - 8]
        mov     qword ptr [rbp - 16], rdi
        call    simple_printer::simple_printer() [base object constructo
        mov     rax, qword ptr [rbp - 16]
        lea     rcx, [rip + vtable for impl_B]
        add     rcx, 16
        mov     qword ptr [rax], rcx
        add     rsp, 16
        pop     rbp
        ret
```

对象impl_B构造过程：取得vtable地址，放入对象中

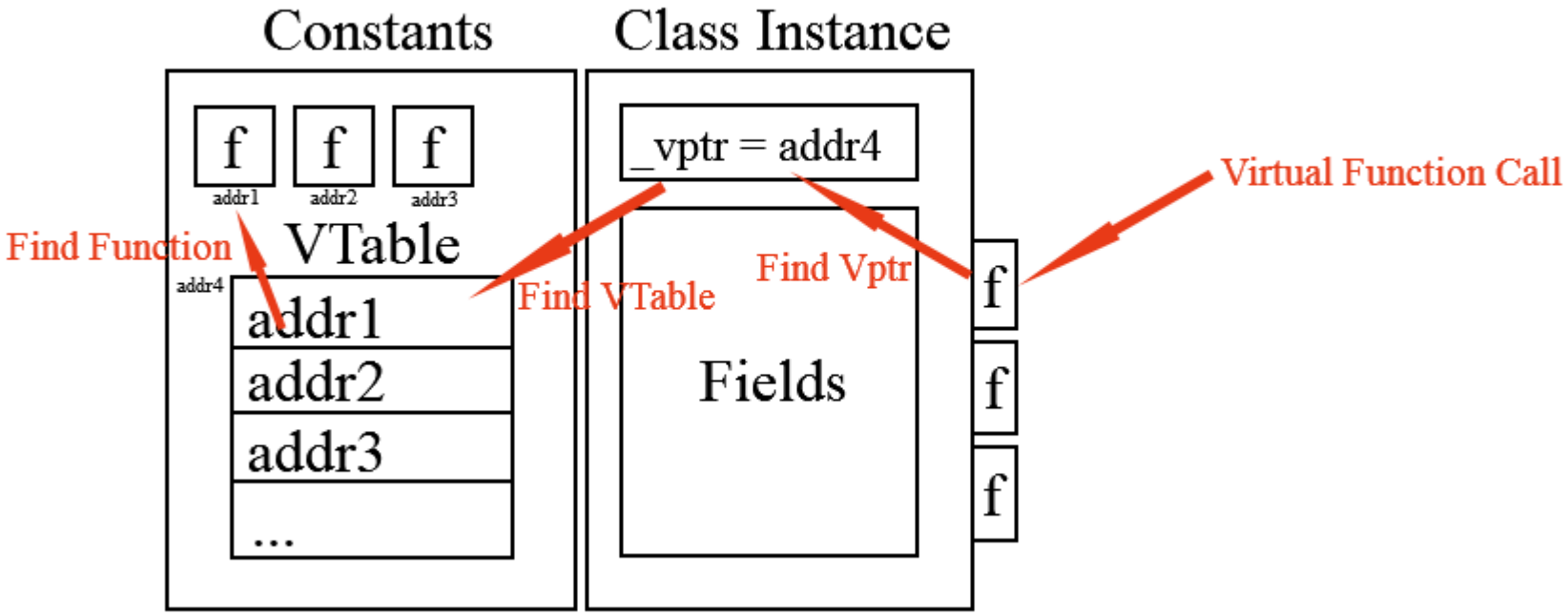
调用过程：获取虚函数地址，调用虚函数

```
mov     rax, qword ptr [rbp - 32]
mov     qword ptr [rbp - 16], rax
mov     rdi, qword ptr [rbp - 16]
mov     rax, qword ptr [rdi]
call    qword ptr [rax + 16]
```

```
vtable for impl_B:
        .quad   0
        .quad   typeinfo for impl_B
        .quad   impl_B::~impl_B() [base object destructor
        .quad   impl_B::~impl_B() [deleting destructor]
        .quad   impl_B::print() const
```

可视化：虚函数的调用过程如图

为什么这么做?
减少内存消耗 √
提高对象构造效率 √
减少cache miss √



结论

C++的类多态本质是把对象和虚函数表相关联。虚函数表中储存了一系列函数指针，当调用虚函数时，先寻找虚函数表，再寻找虚函数指针，再调用真正的函数实现。