31｜面向对象编程第2步：支持继承和多态

2021-10-25 宫文学

《手把手带你写一门编程语言》 课程介绍 >

讲述：宫文学
时长 23:01 大小 21.09M

你好，我是宫文学。

经过前面两节课对面向对象编程的学习，今天这节课，我们终于要来实现到面向对象中几个最核心的功能了。

面向对象编程是目前使用最广泛的编程范式之一。通常，我们说面向对象编程的核心特性有封装、继承和多态这几个方面。只要实现了这几点，就可以获得面向对象编程的各种优势，比如提高代码的可重用性、可扩展性、提高编程效率，等等。

这节课，我们就先探讨一下面向对象的这些核心特性是如何实现的，然后我会带着你动手实现一下，破解其中的技术秘密。了解了这些实现机制，能够帮助你深入理解现代计算机语言更深层次的机制。

首先，我们先来分析面向对象的几个核心特性，并梳理一下实现思路。

面向对象的核心特性及其实现机制

第一，是封装特性。

封装是指我们可以把对象内部的数据和实现细节隐藏起来，只对外提供一些公共的接口。这样做的好处，是提高了代码的复用性和安全性，因为内部实现细节只有代码的作者才能够修改，并且这种修改不会影响到类的使用者。

其实封装特性，我们在上两节课已经差不多实现完了。因为我们提供了方法的机制，让方法可以访问对象的内部数据。之后，我们只需要给属性和方法添加访问权限的修饰成分就可以了。比如我们可以声明某些属性和方法是 private 的，这样，属性和方法就只能由内部的方法去访问了。而对访问权限的检查，我们在语义分析阶段就可以轻松做到。

上一节课，我们已经分析了如何处理点符号表达式。你在程序里可以分析出点号左边的表达式的类型信息，也可以获得对象的属性和方法。再进一步，我们可以给这些属性和方法添加上访问权限的信息，那么这些私有的属性就只可以在内部访问了，比如使用 this.xxx 表达式，等等。而公有的属性仍然可以在外部访问，跟现在的实现没有区别。

第二，我们看看继承。

用直白的话来说，继承指的是一个 class，可以免费获得父类中的属性和方法，从而降低了开发工作量，提高了代码的复用度。

我写了一个示例程序，你可以看一下：

复制代码

```
function println(data:any=""){
    console.log(data);
}

class Mammal{
    weight:number = 0;
    // weight2;
    color:string;
    constructor(weight:number, color:string){
        this.weight = weight;
```

```
        this.color = color;
    }
    speak(){
        println("Hello, I'm a mammal, and my weight is " + this.weight + ".");
    }
}

class Human extends Mammal{  //新的语法要素：extends
    name:string;
    constructor(weight:number, color:string, name:string){
        super(weight,color); //新的语法要素：super
        this.name = name;
    }
    swim(){
        println("My weight is " +this.weight + ", so I swimming to exercise.")
    }
}

class Cat extends Mammal{
    constructor(weight:number, color:string){
        super(weight,color);
    }
    catchMouse(){
        println("I caught a mouse! Yammy!");
    }
}

function foo(mammal:Mammal){
    mammal.speak();
}

let mammal1 : Mammal;
let mammal2 : Mammal;

mammal1 = new Cat(1,"white");
mammal2 = new Human(20, "yellow", "Richard");

foo(mammal1);
foo(mammal2);
```

在这个示例程序中，Human 和 Cat 都继承了 Mammal 类。但人类和猫当然有很大的不同，比如人类通常会有一个姓名，具备游泳的能力，而猫则有抓老鼠的能力。

但因为它们都属于哺乳动物，所以也一定有一些共同的特征，比如都有体重和颜色，也都可以发出声音。这里，使用了继承功能以后，像 weight、color 和 speak() 这样的属性和方法，我们就不需要 Human 和 Cat 去重复实现了。

那实现继承特性的关键点是什么呢？你可以基于我们现在的技术实现来分析一下。你会发现，当我们调用 cat.color 的时候，**最关键的是要对 color 属性进行定位**。在编译期，我们需要通过引用消解，把 color 属性定位到声明它的地方，也就是在父类 Mammal 中。而在运行期，我们需要知道父类的属性的存储位置，这样才可以访问它们。

具体实现，我们在接下来的讲解中依次展开。现在我们分析一下面向对象的第三个特性，也就是多态。

第三，多态特性。

多态指的是一个类的不同子类，都实现了某个共同的方法，但具体表现是不同的。多态的好处，是我们可以在一个比较稳定的、抽象的层面上编程，而不被更加具体的、易变的实现细节干扰。

举例来说，如果我们想要在手机和电脑之间发送信息。那么在抽象的层面上，我们只需要提供 sendData 和 receiveData 这样的编程接口。而在具体实现上，这些信息可能是通过 Wi-Fi 传递的，也可能是通过 5G 网络或蓝牙传递的。系统可以根据不同的网络环境选择不同的机制，但是我们上层使用 sendData 和 recieveData 接口来编写的应用程序，不需要根据传输方式的不同而修改应用逻辑，这样就降低了整体系统的维护成本。

我们也可以修改一下示例程序来说明多态特性。我们给 Human 和 Cat 都增加了一个 speak() 方法，覆盖掉了父类的缺省实现，分别执行了不同的逻辑。

复制代码

```
class Human extends Mammal{  //新的语法要素：extends
    name:string;
    constructor(weight:number, color:string, name:string){
        super(weight,color); //新的语法要素：super
        this.name = name;
    }
    swim(){
        println("My weight is " +this.weight + ", so I swimming to exercise.")
    }
    speak(){
        println("Hello PlayScript!");
    }
}

class Cat extends Mammal{
    constructor(weight:number, color:string){
```

```
17          super(weight,color);
18      }
19      catchMouse(){
20          println("I caught a mouse! Yammy!");
21      }
22      speak(){
23          println("Miao~~");
24      }
25  }
```

你要注意 speak() 方法后面的几行代码，这里就展现了多态的强大之处。你可以声明多个 Mammal 类型的变量，给每个变量赋予 Mammal 的不同子类的对象实例。而在 foo 函数程序里，我们接受一个 Mammal 类型的参数，并让它 speak()。

你会看到，无论 Mammal 的子类将来扩展到多少种，都不会影响到 foo 函数的逻辑，foo 函数只要保持它的抽象性就好了。这种在抽象层面上编程的技术，是实现可重用的编程框架的基础，也通常是一个公司里资深的技术人员的职责。

那如果要实现多态功能，其实我们不能在编译期做什么事情，这主要是运行期的功能。因为你编译 foo 函数的时候，只知道传进来的参数是 Mammal 类型，去调用 Mammal 的 speak() 方法就好了。而在运行期，这个 speak() 方法要正确地定位到具体子类的实现上。这个技术，就做动态绑定（Dynamic Binding），或者后期绑定（Late Binding），这也是面向对象之父阿伦·凯伊（Alan Kay）所提倡的面向对象应该具备的核心特征。

接下来，我们会分别在 AST 解释器和静态编译的两个版本上，讨论运行期绑定的实现细节。

在此之前，我们还是要先修改一下编译器的前端，让它能够支持今天我们讲到的特性。

修改编译器前端

在编译器前端方面，我们仍然需要增强一下语法规则，并进行一些语义分析工作。

语法方面，主要是**增加类继承的语法**，比如：“class Humman extends Mammal”。我们把原来类声明的语法规则稍加修改就行：

复制代码

```
1 classDecl : Class Identifier ('extends' Identifier)? classTail ;
```

另外，在实现了继承以后，我们还需要用到跟 this 相对应的另一个关键字，super。通过 super 关键字，我们可以调用父类的方法。特别是，我们需要在子类的构造方法里，通过 super() 这样的格式，来调用父类的构造方法。

相对来说，我们在语义分析方面要做的工作会更多一点，主要包括：

在 calss 的符号信息里，要增加与继承关系有关的信息；

在 NamedType 类型信息里，也要建立起正确的父子关系，便于进行类型计算；

在子类的构造方法里，第一个语句必须用 super() 调用父类的构造方法。

这些工作倒是没有太复杂的实现难度，你参考一下semantic.ts中的代码。

接下来，我们仍然增强一下 AST 解释器，来支持继承和多态特性。

修改 AST 解释器

AST 解释器方面需要增强的工作包括：

为了实现继承特性，要能够在子类中访问父类的属性和方法；

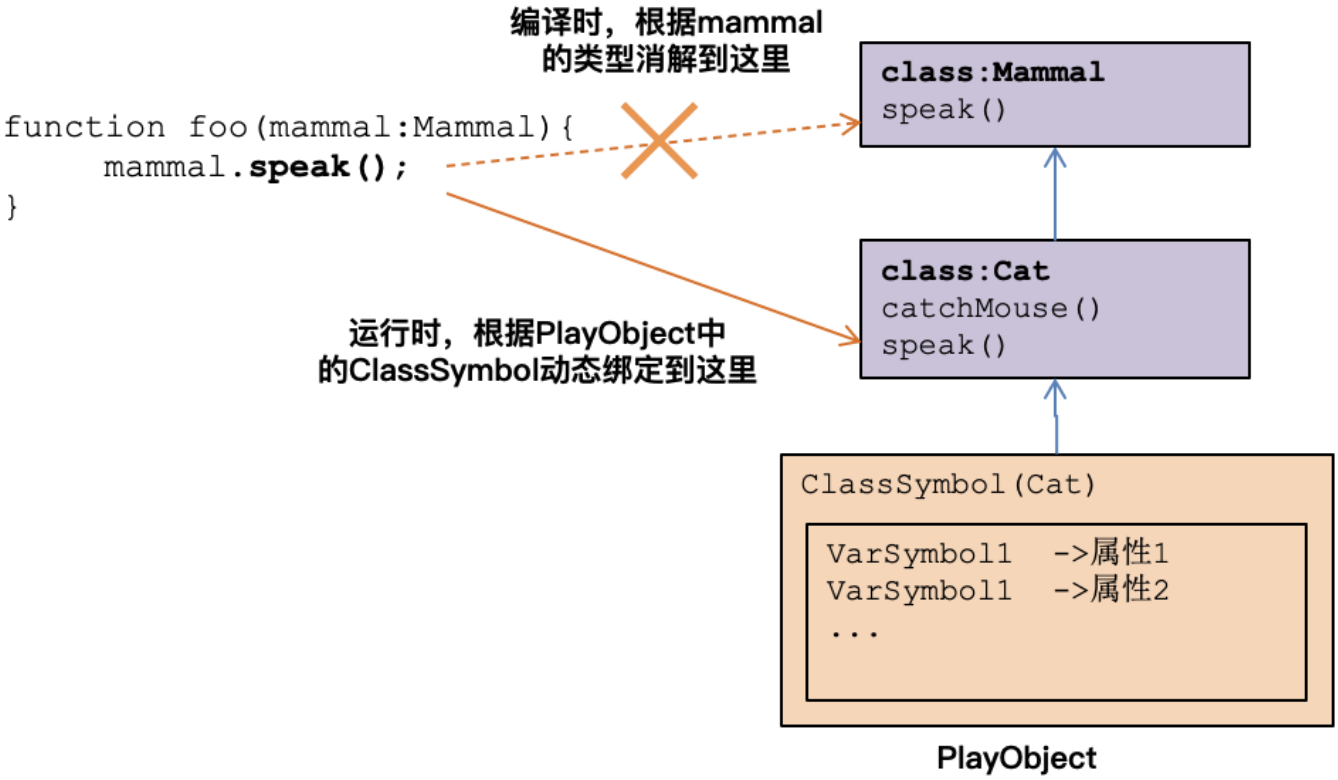
为了实现多态特性，在调用方法时，要能够正确调用具体的子类的实现。

首先我们看如何继承父类的属性。对于 AST 解释器这种运行机制来说，属性的继承实现起来比较简单。因为我们是用 PlayObject 来存放对象数据的，具体存储方式是一个 Map。无论是父类还是子类的属性，都保存在这个 Map 中就可以了。然后我们再以对象属性的 Symbol 为 Key 去访问这些数据就行了。

刚才讲的是属性的继承，那么继承父类的方法也是一样的吗？

不知道你还记不记得，在上一节课中，我们在 PlayObject 中包含了该对象所属的类的符号信息。而这个符号里，又引用了它的父类的符号。所以，我们可以借助这些父子关系的信息，逐级向上查找，直到在某一级父类里找到这个方法。

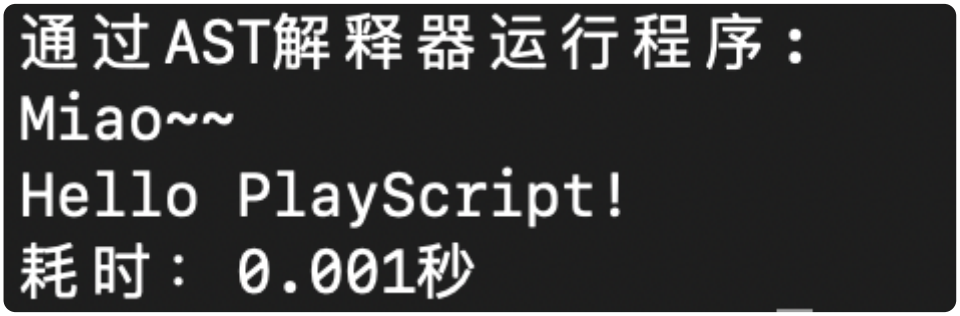
你有没有看出这里面的关键点？**我们是在运行时才知道，到底要调用那个层级的父类所实现的方法的。这也说明，我们在前端的语义分析阶段，不能把方法的调用跟具体的实现绑死，绑定的动作要留到运行时。**虽然在做引用消解的时候，我们把方法调用指向了某个方法的 Symbol，但这个 Symbol 只是用来在运行时去定位真正要调用的方法。



这个在运行时绑定的原则，其实也是实现多态特性的背后机制。在上面的示例程序中，虽然 foo 函数的签名针对的都是 Mammal 类型，但在运行时具体调用方法的时候，我们传递给方法的都是 PlayObject 对象，而 PlayObject 对象中保存的 ClassSymbol 呢，是具体的子类的符号信息，这当然就会导致调用不同子类的方法，从而也就实现了多态。

所以说，**方法的继承和多态这两件事，落实到实现上是同一件事，都是根据 PlayObject 中的符号信息，去查找到真正应该调用的方法就好了**。

具体实现上，你可以参考 play.ts的中的示例程序。



整个实现下来，你会感觉到似乎也不太难呀。对于解释执行的运行时来说，确实如此。但对于编译执行的运行时机制来说，就需要更多的技巧才能实现继承和多态的机制。

在可执行程序中实现继承和多态

在解释执行的运行机制中，我们在访问对象的属性和方法之前可以做很多工作，能让我们确定属性的地址，或者定位具体的方法。但这个过程会导致不少的额外开销。

比如，在 AST 解释器中，去访问一个属性的时候，需要查找一个 Map。这样的话，一次赋值过程可能要导致内部多次函数调用。对比我们生成的汇编语言的代码，在访问一个对象数据的时候，只需要用几个指令做内存地址的计算和数据的访问就可以了，性能很高。

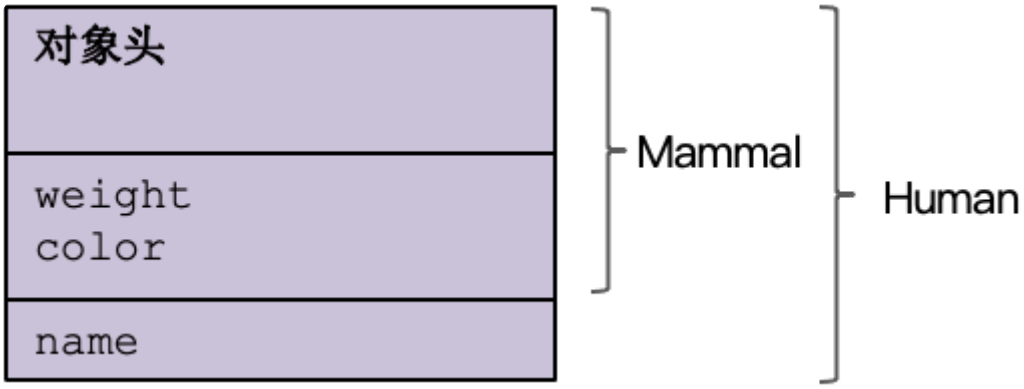
调用方法也是如此。在 C++ 这样的语言中，函数调用不可以有太多的额外开销，因为它毕竟是一门系统级的语言，是用于开发操作系统、数据库这类软件的，所以要求性能尽量高，资源占用尽量少。

所以，我们就要细致地设计一下对象的内存布局和方法的动态绑定机制，让我们的程序既具备面向对象带来的灵活性，又不会导致额外的开销。

在这节课的实现中，我们就借鉴一下 C++ 的实现机制。这个实现方式很经典，值得好好掌握。

首先说一下对象的内存布局。在对象继承的情况下，怎么保存所有的属性数据呢？包括自己这一级的属性和各级父类的属性。

你可以稍作思考。有了前面我们设计对象内存布局的经验，我相信你一定会拿出正确的技术方案。这个方案就是，不管父类和子类的数据，都集中在一块内存里连续存放。并且，我们要先放父类的数据，再放子类的数据。你可以参考一下这张图：



那为什么要先放父类的数据，再放子类的数据呢？

这是因为，这可以让我们在程序中无缝的进行类型的转换。比如，你可以把一个 Cat 对象的引用赋给一个 Mammal 变量。因为 Cat 对象的指针和 Mammal 的指针是同一个地址。反过来，你在知道一个 Mammal 变量的具体类型的情况下，也可以把一个 Mammal 引用强制转换成一个 Cat 引用，来访问 cat 特有的属性。在 TypeScript 的前端，这种类型转换是用 as 关键字实现的。在运行时，可以从基地址开始，加上一定的偏移量，就可以计算出各级子类的属性的地址。

好了，如何访问对象属性的机制就搞清楚了。那么如何在运行时找到正确的对象方法呢？这就要提一下著名 **vtable 机制**了。

vtable 的原理是什么呢？

我们先回顾一下，对于普通的函数，我们是怎么调用的。这很简单，就是用一个标签标记一下函数的入口，然后从调用者那里就可以直接跳转过来。在编译期，我们就知道对这个函数的调用肯定要跳转到这个标签。而这个标签，最后会变成内存里文本段的确定的代码地址。这就是静态绑定。

那我们如何把它改成动态绑定呢？我们可以借鉴 AST 运行时中的机制。在 AST 的运行时中，我们是在获得了 PlayObject 对象之后，根据 PlayObject 对象中的 ClassSymbol 来绑定具体的方法。

所以，vtable 也是采用了这么一个机制。vtable 是一个表格，保存了对象的每个可被覆盖的方法的代码地址。vtable 中的每个条目，都对应了一个方法。

比如一个 Human 对象的 vtable 中，条目 1 对应的是 speak() 方法，条目 2 对应的是 swim() 方法。如果 Human 没有重载父类的 speak() 方法，那条目 1 里存的就是父类的方法地址。那么如果 Human 重载了这个方法呢？那我们就用新的方法地址覆盖掉它。这样在程序执行的时候，我们通过 Mammal 对象的指针，再查找 vtable 来获得 speak() 方法的地址的时候，不同的子类的 speak() 方法的地址就是不同的。这样也就实现了方法的继承和多态。

那具体要生成什么样的汇编代码，来支持我们刚才说到的 vtable 机制和程序的跳转机制呢？我们可以看看 C++ 是怎么实现的。

我写了一个 C++ 的示例程序（ class2.cpp ），仍然实现了 Mammal、Human 和 Cat 这三个类。其中父类有两个方法 speak() 和 run() 可以被子类覆盖，也就是带有 virtual 关键字；它还有一个 breath() 方法不可以被子类覆盖。在 foo 函数中，我们分别调用了 mammal 的 speak() 方法和 breath() 方法。

复制代码

```
#include "stdio.h"

class Mammal{
    public:
    double weight;
    //这个方法不可被子类覆盖
    void breath(){
        printf("mammal breath~~\n"); //这里用c语言的库，而不是c++的cout，是为了让生成
    }
    //这个方法以virtual开头，可以被子类覆盖
    virtual void speak(){
        printf("I'm mammal.\n");
    }
    //第二个virtual方法
    virtual void run(){
        printf("I'm mammal.\n");
    }
};

class Cat : public Mammal{
    public:
    double jumpHeight;
    //覆盖了父类的speak方法
    void speak(){
        printf("I can jump %lf m.\n", jumpHeight);
    }
    //子类自己的方法
    void catchMouse(){
```

```
29          printf("I can catch mouse.\n");
30      }
31  };
32
33  class Human : public Mammal{
34      public:
35      double age;
36      //覆盖了父类的speak方法
37      void speak(){
38          printf("I'm %lf years old.\n", age);
39      }
40  };
41
42  void foo(Mammal* mammal){
43      mammal->breath();
44      mammal->speak();
45  }
46
47  int main(){
48      Cat * cat = new Cat();
49      cat->weight = 10;
50      cat->jumpHeight = 5;
51
52      Human * human = new Human();
53      human->weight = 80;
54      human->age = 18;
55
56      foo(cat);
57      foo(human);
58
59      delete cat;
60      delete human;
61
62      return 0;
63  }
64
```

然后我们用“clang++ class2.cpp -o class2”命令，把这个 C++ 程序编译成可执行文件。然后再运行这个 class2 程序，就可以得到下面的输出：

```
➜  29 git:(master) ✗ clang++ class2.cpp -o class2
➜  29 git:(master) ✗ ./class2
mammal breath~~
I can jump 5.000000 m.
mammal breath~~
I'm 18.000000 years old.
```

你能看到，对于 breath() 方法，程序调用的是父类 Mammal 的实现，这是继承。而对于 speak() 方法来说，程序调用的是两个子类各自的实现，这是重载。

接下来，你可以用“clang++ -S class2.cpp -o class2.s”命令生成汇编文件class2.s，观察 C++ 语言是怎么实现 vtable 的。

这个汇编代码有点长，如果你不熟悉它的结构，可能一下子会有点晕。不过，在你找到了规律，知道了生成汇编代码的思路以后，就会觉得容易接受了。

在这里，我带你分析一下它里面的主要代码，让你理解 C++ 程序编译后，到底是如何形成 vtable 的，又是如何实现动态绑定的。

首先看看 main 函数所生成的代码。我从里面截取了一段，对应于源代码中的前三行代码。

```
_main:                                  ## @main
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $64, %rsp
    movl    $0, -4(%rbp)
    movl    $24, %edi
    callq   __Znwm
    xorl    %esi, %esi
    movq    %rax, %rcx
    movq    %rcx, %rdi
    movl    $24, %edx
    movq    %rax, -32(%rbp)
    callq   _memset
    movq    -32(%rbp), %rdi
    callq   __ZN3CatC1Ev
    movsd   LCPI2_2(%rip), %xmm0
    movsd   LCPI2_3(%rip), %xmm1
    movq    -32(%rbp), %rax
    movq    %rax, -16(%rbp)
    movq    -16(%rbp), %rcx
    movsd   %xmm1, 8(%rcx)
    movq    -16(%rbp), %rcx
    movsd   %xmm0, 16(%rcx)
```

内置函数，从堆中申请内存
Cat对象一共占据24个字节

调用memset函数，把这24个字节的内存都设置为0。
参数1：rdi，是内存地址
参数2：esi，是0
参数3：edx，是内存的大小。

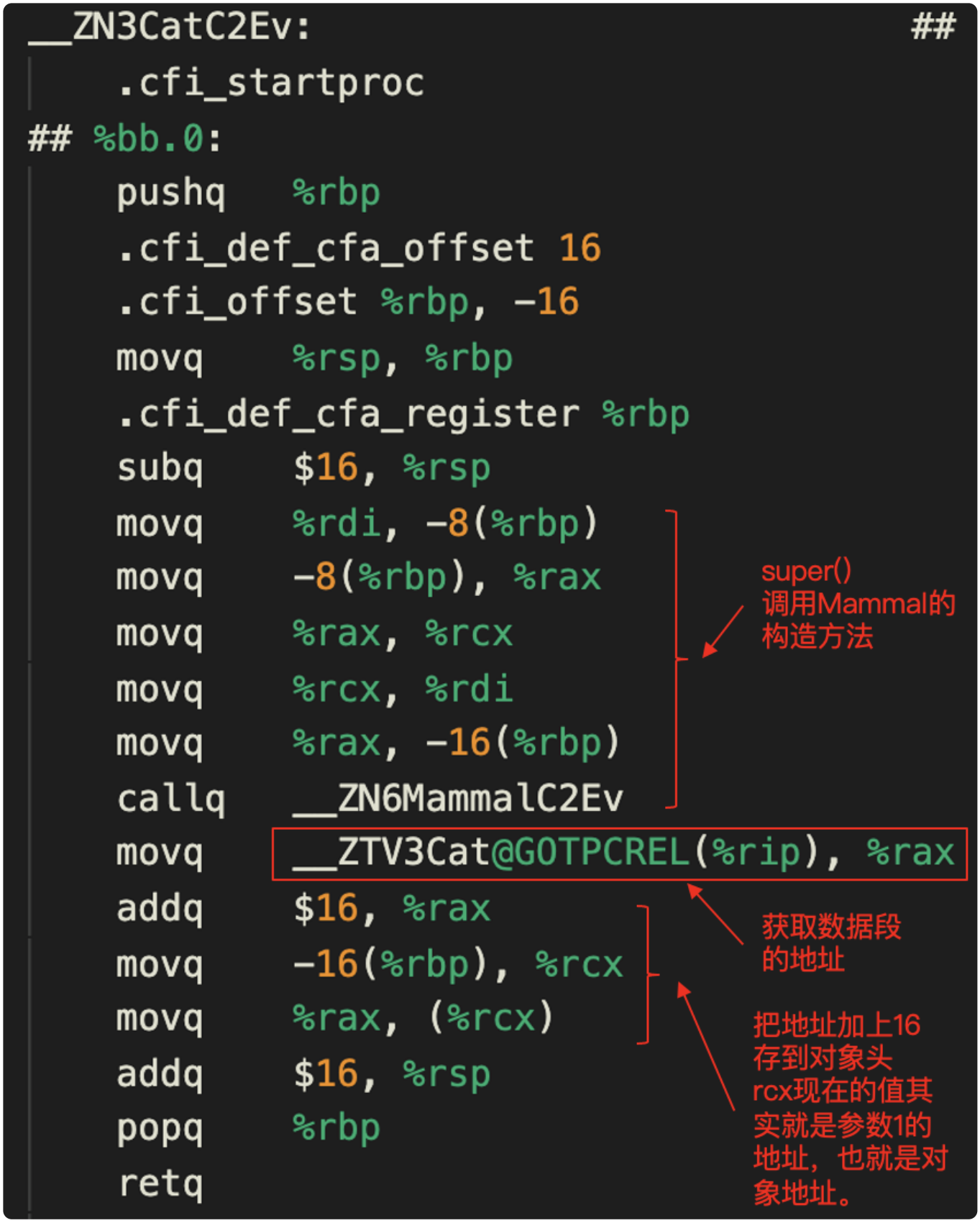
Cat对象的初始化函数

```
                                    ## 8-byte Spill
                                    ## 8-byte Reload
                                    ## xmm0 = mem[0],zero
                                    ## xmm1 = mem[0],zero
                                    ## 8-byte Reload
```

cat->weight = 10;
cat->jumpHeight = 5;
其中rcx的值，是Cat对象的起始地址
对象头占8个字节，其他两个字段分别占8个字节

从这段代码中你能看出，程序为 Cat 对象申请了 24 个字节的内存。其中前 8 个字节存的就是 vtable 的指针，后面 16 个字节分别是 weight 和 jumpHeight。其中 weight 来自父类，而 jumpHeight 则是在 Cat 类中声明的。

其中的 __ZN3CatC1Ev 是做 Cat 的对象初始化工作。你跟踪这个函数的代码，会发现它又调用了 __ZN3CatC2Ev。而在 __ZN3CatC2Ev 中做了两件重要的事情。

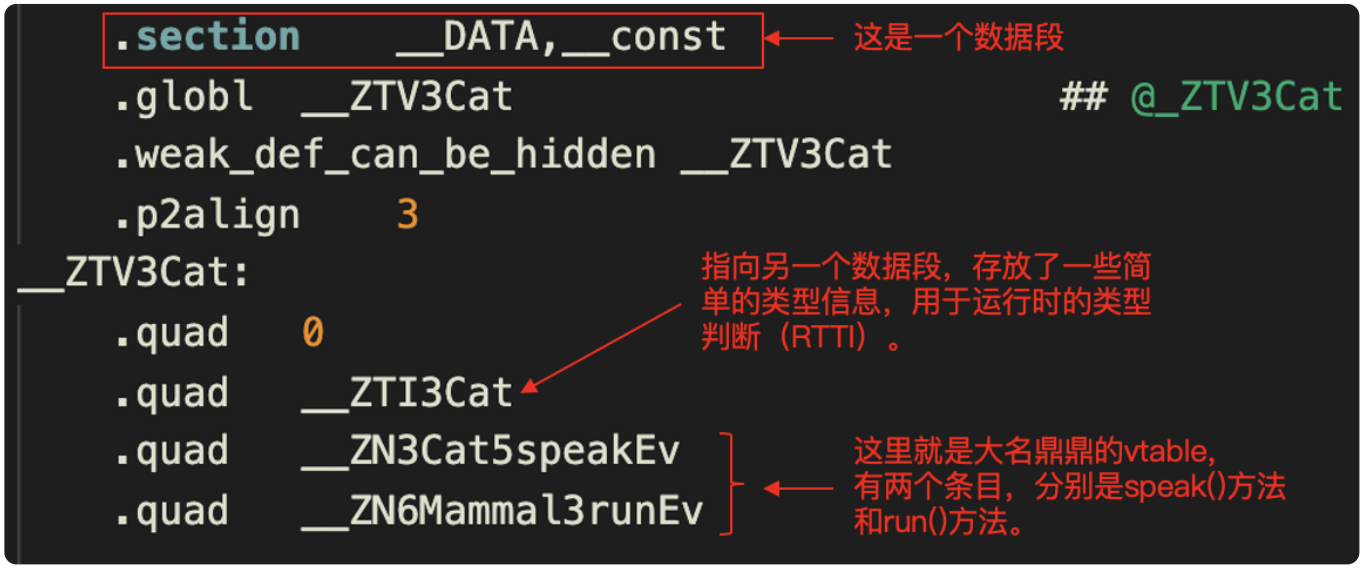


首先，是它调用了一个为父类 Mammal 做初始化的函数。然后是一条重要的语句“movq __ZTV3Cat@GOTPCREL(%rip), %rax”。它的意思是，把 __ZTV3Cat 这个标签的地址，相对于代码寄存器中的值的偏移量，存到 rax 寄存器中去。

__ZTV3Cat 这个标签指向的是一个数据段，里面保存了一些常量，就像我们之前用过的 double 常量和字符串常量一样。而这里的一些常量，是关于 Cat 类的一些描述，其中包括类型信息，以及 vtable。编译后这些信息进入可执行程序的数据区，并且可以用刚才的的指令访问。

接下来，后面几条指令，是把 __ZTV3Cat 的地址值加上 16 个字节，写到了 Cat 对象的对象头里，也就是前 8 个字节。

那为什么要加上 16 个字节呢？这就需要我们到 __ZTV3Cat 这个标签下，看看这个数据段里到底有一些什么数据。你可以看看下面的图。



你会看到，在这个段中，一共有 4 个 8 字节常量。其中第一个常量是 0，这个常量我们不用管它。第二个 8 字节常量，则是另一个标签，指向另一个数据区，里面保存了 Cat 的一些类型信息，包括类型名称等等。这些信息可被用于 RTTI 功能，也就是运行时的类型判断功能。

这里的重点在第三和第四个 8 字节区域，它们分别保存了两个虚函数的标签。第一个虚函数是 speak 函数，这个函数指向的是 Cat 所实现的 speak 函数，而不是父类的函数。而第二个虚函数，则是指向 Mammal 的实现，因为 Cat 并没有覆盖父类中的实现。这最后的 16 个字节，就是 Cat 类的 vtable。

那么，程序里具体是如何使用 vtable 来调用函数的呢？你可以再接着看看 foo 函数的实现。

函数调用了 3 个函数。在调用 breath() 方法的时候，是直接使用了 breath() 方法的标签，并没有经过 vable。这是由 C++ 的语言特性决定的。如果一个方法前面没有用 virtual 来修饰，那它就不可以被子类覆盖，因此自然也就不需要 vtable 了。

而在调用 speak() 方法的时候，代码里使用了“callq *(%rcx)*”。*这条指令的意思是，%rcx 寄存器里保存了一个内存地址。这个内存地址里保存了要执行的代码的地址，然后去执行这个代码。实际上，%rcx 寄存器就是刚才的数据区的起始地址加上 16 字节后的值，这个值正是 vtable 的地址。而*(%rcx)，实际上就是 vtable 中记录的第一个虚函数的地址。如果你要调用第二个虚函数，那么需要使用 callq *8(%rcx) 指令，基于 vtable 开头的位置偏移 8 个字节。

好了，到这里，我们就把 C++ 实现继承和多态，以及 vtable 的技术细节都分析清楚了。那么，我们在后端同样也可以做一个这样的参考实现。我们实现起来可以比 C++ 的机制更加简化。这是因为，首先，我们目前还不需要运行时类型机制，可以简化掉这部分。第二，我们把所有父类的方法都放在 vtable 中，因为目前所有父类的方法都是公共的，都是允许被覆盖的。

我提供的参考实现，仍然在asm_x86-64.ts中。

课程小结

今天的内容就是这些。关于面向对象的核心特性，我希望你记住以下几个知识点：

首先，继承和多态的核心点，都是来自动态绑定技术。也就是说，在编译期并不知道实际调用的是哪一级的方法，只有在运行期根据对象的具体类型才知道。

第二，在 AST 解释器中，我们借助 PlayObject 中存储的 ClassSymbol 信息，就能动态地找到方法的正确实现。

第三，在编译成可执行文件时，我们要借助 vtable 技术来实现继承和多态特性。vtable 是在编译时就生成了的，保存在一个数据区。在创建对象的时候，我们就把数据区中 vtable 的地址写入到对象头中。最后，我们只用一条类似于 callq *8(%rcx) 指令，就能查找出 vtable 中记录的函数地址，从而跳转过去执行。

今天这节课的内容是非常有用的。如果你使用过 C++ 技术，那么今天我带你剖析了 C++ 的汇编代码后，你会对 C++ 的实现机制理解得更加深入。如果你没有使用过 C++，那你也一定要记住 vtable 这种技术，因为它是静态编译的语言实现面向对象特性的关键。

我们用了三节课的时间，实现了面向对象最核心的一些特性。在此基础上，我们可以扩展到支持更多的特性，比如 TypeScript 是支持接口的，我们可以把接口特性添加上。再比如，我们还可以把 Norminal 的类型系统改成 Structural 的，让类型之间的兼容更灵活，并且还可以看看这个时候用 vtable 来实现多态还行不行。

思考题

我们今天讨论了用 vtable 在静态编译中实现多态。那么你能不能挑战一下，看能不能提供另外的方案来实现多态？你可以天马行空地想一想，并在留言区分享你的观点。

并且，如果我们的类型系统改成 Structural 的，那么我们需要如何修改现在的 vtable 机制，才能准确地在运行时绑定正确的方法呢？对于这个问题，你也可以谈谈想法。

欢迎你把这节课分享给更多感兴趣的朋友。我是宫文学，我们下节课见！

资源链接

这节课的代码目录在这里！

上一篇 30 | 面向对象编程第2步：剖析一些技术细节

精选留言 (1)

写留言

奋斗的蜗牛
2021-10-26

太赞了，原来这些高级特性是这么实现的

展开