



下载APP



38 | 综合实现（一）：如何实现面向对象编程？

2020-09-11 宫文学

编译原理实战课

[进入课程 >](#)**讲述：宫文学**

时长 25:22 大小 23.24M



你好，我是宫文学。

从 20 世纪 90 年代起，面向对象编程的范式逐渐成为了主流。目前流行度比较高的几种语言，比如 Java、JavaScript、Go、C++ 和 Python 等，都支持面向对象编程。

那么，为了支持面向对象编程，我们需要在语言的设计上，以及编译器和运行时的实现上，考虑到哪些问题呢？

这一讲，我就带你来探讨一下如何在一门语言里支持面向对象特性。这是一个很综合的问题，会涉及很多的知识点，所以很有助于帮你梳理和贯通与编译原理有关的知识。



那么，我们就先来分析一下，面向对象特性都包括哪些内容。

面向对象语言的特性

日常中，虽然我们经常会使用面向对象的语言，但如果要问，到底什么才是面向对象？我们通常会说得含含糊糊。最常见的情况，就是会拿自己所熟悉的某种语言的面向对象特性，想当然地认为这就是面向对象语言的全部特性。

不过，在我们的课程里，我想从计算机语言设计的角度，带你重新梳理和认识一下面向对象的编程语言，把面向对象按照清晰的逻辑解构，这样也便于讨论它的实现策略。在这个过程中，你可能会对面向对象产生新的认识。

特征 1：对象

面向对象编程语言的核心，是把世界看成了一个一个的对象，比如汽车、动物等。这些对象包含了数据和代码。数据被叫做字段或属性，而代码通常又被叫做是方法。

此外，**这些对象之间还会有一定的关系**。比如，汽车是由轮子、发动机等构成的，这叫做**聚合关系**。而某个班级会有一个班主任，那么班级和作为班主任的老师之间，会有一种**引用关系**。

对象之间还可以互相发送消息。比如，司机会“通知”汽车，让它加速或者减速。在面向对象的语言中，这通常是通过方法调用来实现的。但也并不局限于这种方式，比如对象之间还可以通过异步的消息进行互相通讯，不过一般的编程语言都没有原生支持这种通讯方式。我们在讨论 **Actor 模式** 的时候，曾经提到过 Actor 之间互相通讯的方式，就有点像对象之间互发消息。

特征 2：类和类型体系

很多面向对象的语言都是基于类（class）的，并且类也是一种自定义的类型。这个类型是对象的模板。而对象呢，则是类的实例。我们还可以再印证一下，前面在探究 **元编程** 的实现机制时，学过的 Meta 层次的概念。对象属于 M0 层，而类属于 M1 层，它为对象制定了一个标准，也就是对象中都包含了什么数据和方法。

其实，**面向对象的语言并不一定需要类这个概念**，这个概念更多是来自于类型理论，而非面向对象的语言一样可以支持类型和子类型。类型的好处主要是针对静态编译的语言的，因为这样就可以通过类型，来限制可以访问的对象属性和方法，从而减少程序的错误。

而有些面向对象的语言，比如 JavaScript 并没有类的概念。也有的像 Python，虽然有类的概念，但你可以随时修改对象的属性和方法。

特征 3：重用—继承 (Inheritance) 和组合 (Composition)

在软件工程里，我们总是希望能重用已有的功能。像 Java、C++ 这样的语言，能够让子类型重用父类型的一些数据和逻辑，这叫做**继承**。比如 Animal 有 speak() 方法，Cat 是 Animal 的子类，那么 Cat 就可以继承这个 speak() 方法。Cat 也可以重新写一个方法，把父类的方法覆盖掉，让叫声更像猫叫。

不过，并不是所有的面向对象编程语言都喜欢通过继承的方式来实现重用。你在网上可以找到很多文章，都在分析继承模式的缺陷。像 Go 语言，采用的是**组合方式**来实现重用。在这里，我引用了[一篇文章](#)中的例子。在这个例子中，作者首先定义了一个 author 的结构体，并给这个结构体定义了一些方法：

[复制代码](#)

```
1  type author struct {      //结构体: author(作者)
2      firstName string      //作者的名称
3      lastName  string
4      bio       string      //作者简介
5  }
6
7  func (a author) fullName() string {    //author的方法: 获取全名
8      return fmt.Sprintf("%s %s", a.firstName, a.lastName)
9  }
10
11 type post struct {         //结构体: 文章
12     title    string        //文章标题
13     content  string        //文章内容
14     author   string        //文章作者
15 }
16
17 func (p post) details() {    //文章的方法: 获取文章的详细内容。
18     fmt.Println("Title: ", p.title)
19     fmt.Println("Content: ", p.content)
20     fmt.Println("Author: ", p.author.fullName())
21     fmt.Println("Bio: ", p.author.bio)
22 }
```

关于 struct，这里我想再给你强调几个知识点。熟悉 C 语言的同学应该都了解结构体 (struct)。有一些比较新的语言，比如 Go、Julia 和 Rust，也喜欢用结构体来作为复合

数据类型，而不愿意用 `class` 这个关键字，而且它们也普遍摒弃了用继承来实现重用的思路。Go 提倡的是组合；而我们上一讲提到的泛型，也开始在重用方面承担了越来越重要的角色，就像在 Julia 语言里那样；Rust 和另外一些语言（如 Scala），则使用了一种叫做 **Trait（特征）** 的技术。

Trait 有点像 Java 和 Go 语言中的接口，它也是一种类型。不过它比接口还多了一些功能，那就是 Trait 里面的方法可以具体地实现。

我们用 Trait 可以替代传统的继承结构，比如，一个 Cat 可以实现一个 Speakable 的 Trait，而不需要从 Animal 那里继承。如果 Animal 还有其他的特征，比如说 `reproduce()`，繁殖，那也可以用一个 Trait 来代替。这样，Cat 就可以实现多个 Trait。这会让类型体系更加灵活，比如实现 Speakable 的不仅仅有动物，还可以是机器人。

在像 Scala 这样的语言中，Trait 里不仅仅可以有方法，还可以有成员变量；而在 Ruby 语言中，我们把这种带有变量和方法的可重用的单元，叫做 Mixin（混入）。

无论 Trait 还是 Mixin，都是基于组合的原理来实现重用的。而且由于继承、接口、Trait 和 Mixin 都可以看做是实现子类型的方式，因此也都可以支持多态。因为继承、接口、Trait 和 Mixin 一般都有多个具体的实现类，所以在调用相同的方法时，会有不同的功能。

特征 4：封装（Encapsulation）

我们知道，软件工程中的一个原则是信息隐藏，我们通常会称为**封装（encapsulation）**，意思是软件只把外界需要知道的信息和功能暴露出来，而内部具体的实现机制，只有作者才可以修改，并且不会影响到它的使用者。

同样的，实现信息封装其实也不是面向对象才有的概念。有的语言的模块（Module）和包（Package）等，都可以作为封装的单元。

在面向对象的语言里，通常对象的一些方法和属性可以被公共访问的，而另一些方法和属性是内部使用的，其访问是受限的。比如，Java 语言会对可以公共访问的成员加 `public` 关键字，对只有内部可以访问的成员加 `private` 关键字。

好了，以上就是我们总结的面向对象语言的特征了。这里你要注意，面向对象编程其实是一个比较宽泛的概念。对象的概念是它的基础，然后语言的设计者再把类型体系、软件重

用机制和信息封装机制给体现出来。在这个过程中，不同的设计者会有不同的取舍。所以，希望你不要僵化地理解面向对象的概念。比如，以为面向对象就必须有类，就必须有继承；以为面向对象才导致了多态，等等。这些都是错误的理解。

接下来，我们再来看看各门语言在实现这些面向对象的特征时，都要解决哪些关键技术问题。

如何实现面向对象的特性？

要实现一门面向对象的语言，我们重点要了解三个方面的关键工作：一是编译器在语法和语义处理方面要做哪些工作；二是运行期对象的内存布局的设计；三是在存在多态的情况下，如何实现方法的绑定。

我们首先从编译器在语法和语义处理上所做的工作开始学起。

编译器前端的工作

我们都知道，编译器的前端必须完成与类和对象有关的语法解析、符号表处理、引用消解、类型分析等工作。那么要实现一门面向对象的语言，编译器也需要完成这些工作。

第一，从语法角度来看，语言的设计者要设计与类的声明和使用有关的语法。

比如：

如何声明一个类？毕竟每种语言的风格都不同。

如何声明类的构造方法？

如何声明类与父类、接口、Trait 等的关系？

如何实例化一个对象？像 Java 语言就需要 new 关键字，而 Python 就不需要。

.....

也就是说，编译器在语法分析阶段，至少要能够完成上述的语法分析工作。

第二，是要维护符号表，并进行引用消解。

在语义分析阶段，每个类会作为自定义类型被加入到符号表里。这样，在其他引用到该类型的地方，比如用该类型声明了一个变量，或者一个方法的参数或返回值里用到了该类型，编译器就能够做正确的引用消解。

另外，面向对象程序的引用消解还有一个特殊之处。因为父类中的成员变量、方法甚至类型的作用域会延伸到子类，所以编译器要能够在正确的作用域内做引用消解。比如，在一个方法体内，如果发现某个变量既不是本地变量，又不是参数，那么程序就要去找类的成员变量。在当前的类里找不到，那就要到父类中逐级去找。

还有一点，编译器在做引用消解的时候，还可以完成访问权限的检查。我们知道，对象要能够实现信息封装。对于编译器来说，这个功能实现起来很简单。在做引用消解的时候，检查类成员的访问权限就可以了。举个例子，假设你用代码访问了某个私有的成员变量，或者私有的方法，此时程序就可以报错；而在这个类内部的代码中，就可以访问这些私有成员。这样就实现了封装的机制。

第三，要做类型检查和推断。

使用类型系统的信息，在变量赋值、函数调用等地方，会进行类型检查和推断。我们之前学过的关于子类型、泛型等知识，在这里都可以用上。

OK，以上就是编译器前端关于实现面向对象特性的重点工作了，我们接下来看看编译器在运行时的一个设计重点，就是对象的内存布局。

对象的内存布局

在第二个模块，研究几个不同的编译器的时候，我们已经考察过各种编译器在保存对象时所采用的内存布局。像 Java、Python 和 Julia 的对象，一般都有一个固定的对象头。对象头里可以保存各种信息，比如到类定义的指针、与锁有关的标志位、与垃圾收集有关的标志位，等等。

对象头之后，通常就是该类的数据成员。如果存在父类，那么就既要保存父类中的成员变量，也要保存子类中的成员变量。像 Python 这样的语言，对内存的使用比较浪费，通常是用一个内部字典来保存成员变量；但对于 Java 这样的语言，则要尽量节约着用内存。

我举个例子。假设某个 Java 类里有两个成员变量，那么这两个成员变量会根据声明的顺序，排在对象头的后面。如果成员变量是像 `Int` 这样的基础数据，那么程序就要保存它的值；而如果是 `String` 等对象类型，那么就要保存一个指向该对象的指针。

在 Java 语言中，当某个类存在父类的情况下，那么父类的数据成员一定要排在前面。

这是为什么呢？我给你举一个例子。在这个例子中，有一个父类 `Base`，有两个子类分别是 `DerivedA` 和 `DerivedB`。

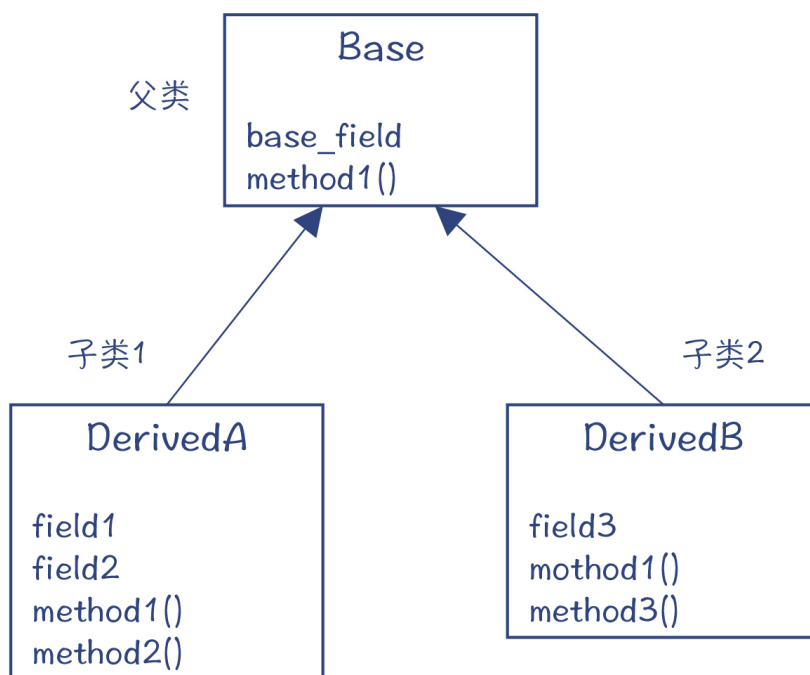


图 1：Base 类有两个子类

如果两个子类分别有一个实例 `a` 和 `b`，那么它们的内存布局就是下面的样子：

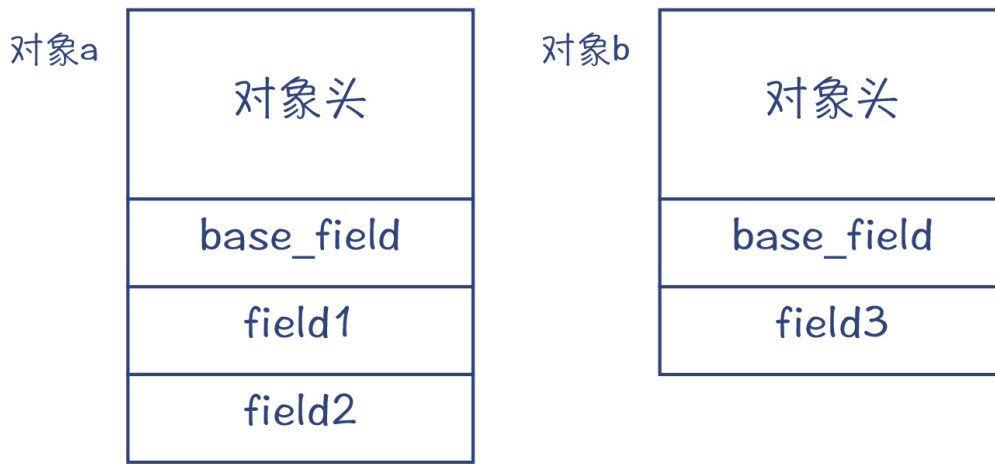


图 2：两个子类的实例的内存布局

那么你可能要问了，**为什么父类的数据成员要放在前面，子类的要放在后面？**这也要从编译的角度说起。

我们知道，在生成的汇编代码里，如果要访问一个类的成员变量，其实都是从对象地址加上一个偏移量，来得到成员变量的地址。而这样的代码，针对父类和各种不同的子类的对象，要都能正常运行才行。所以，该成员变量在不同子类的对象中的位置，最好是固定的，这样才便于生成代码。

不过像 C++ 这样的语言，由于它经常被用来编写系统级的程序，所以它不愿意浪费任意一点内存，因此就不存在对象头这样的开销。但是由于 C++ 支持多重继承，所以当某个类存在多个父类的情况下，在内存里安排不同父类的成员变量，以及生成访问它们的正确代码，就要比 Java 复杂一些。

比如下面的示例代码中，c 同时继承了 a 和 b。你可以把对象 obj 的地址分别转换为 a、b 和 c 的指针，并把这个地址打印出来。

复制代码

```
1 class a { int a_; };
2 class b { int b_; };
3 class c : public a, public b { };
4 int main(){
5     c obj;
6     printf("a=0x%08x, b=0x%08x, c=0x%08x\n", (a*)&obj, (b*)&obj, (c*)&obj);
7 }
```


看到这段代码，你发现什么了呢？

你会发现，a 和 c 的指针地址是一样的，而 b 的地址则要大几个字节。这是因为，在内存里程序会先排 a 的字段，再排 b 的字段，最后再排 c 的字段。编译器做指针的类型转换（cast）的时候，要能够计算出指针的正确地址。

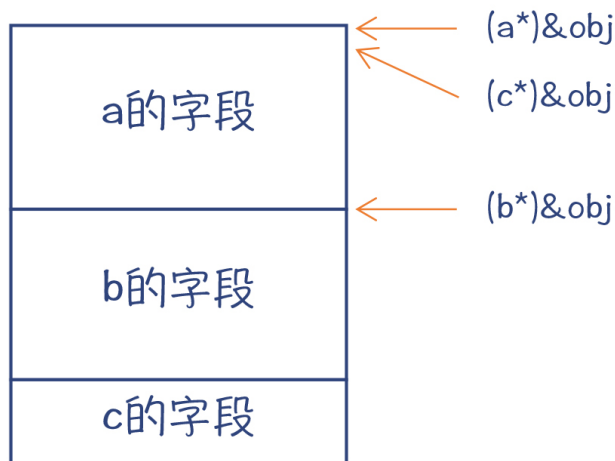


图 3：多重继承情况下的内存布局

好，现在你就已经见识到了编译器在运行期针对对象的内存布局的设计了。接下来，我们再来看看针对多态的情况，编译器对实现方法的绑定是怎么做的。

方法的静态绑定和动态绑定

当你的程序调用一个对象的方法的时候，这个方法到底对应着哪个实现，有的时候在编译期就能确定了，比如说当这个方法带有 `private`、`static` 或 `final` 关键字的时候。这样，你在编译期就知道去执行哪段字节码，这被叫做**静态绑定 (Static Binding)**，也可以叫做早期绑定 (Early Binding) 或者静态分派 (Static Dispatching)。

另外，对于重载 (Overload) 的情况，也就是方法名称一样、参数个数或类型等不一样的情况，也是可以在编译期就识别出来的，所以也可以通过静态绑定。

而在存在子类型的情况下，我们到底执行哪段字节码，只有在运行时，根据对象的实际类型才能确定下来。这个时候就叫做**动态绑定 (Dynamic binding)**，也可以叫做后期绑定 (Late binding) 或者动态分派 (Dynamic Dispatching)。

动态绑定也是面向对象之父阿伦·凯伊 (Alan Kay) 所提倡的面向对象的特征：**绑定时机要尽量地晚**。绑定时机晚，意味着你在编程的时候，可以编写尽量通用的代码，也就是代码里使用的是类型树中更靠近树根的类型，这种类型更通用，也就可以让使用这种类型编写的代码能适用于更多的子类型。

那么动态绑定在运行时是怎么实现的呢？对于 Java 语言来说，其实现机制对于每个 JVM 可以是不同的。不过，我们可以参考 C++ 的实现机制，就可以猜测出 JVM 的实现机制。

在 C++ 语言中，动态绑定是通过一个 **vtable 的数据结构**来实现的。vtable 是 Virtual Method Table 或 Virtual Table 的简称。在 C++ 里，如果你想让基类中的某个方法可以被子类覆盖 (Override)，那么你在声明该方法的时候就要带上 **virtual 关键字**。带有虚方法的类及其子类的对象实例，都带有一个指针，指向一个表格，这也就是 vtable。

vtable 里保存的是什么呢？是每个虚方法的入口地址。我们来看一个例子，这个例子中有 Base、DerivedA 和 DerivedB 三个类：

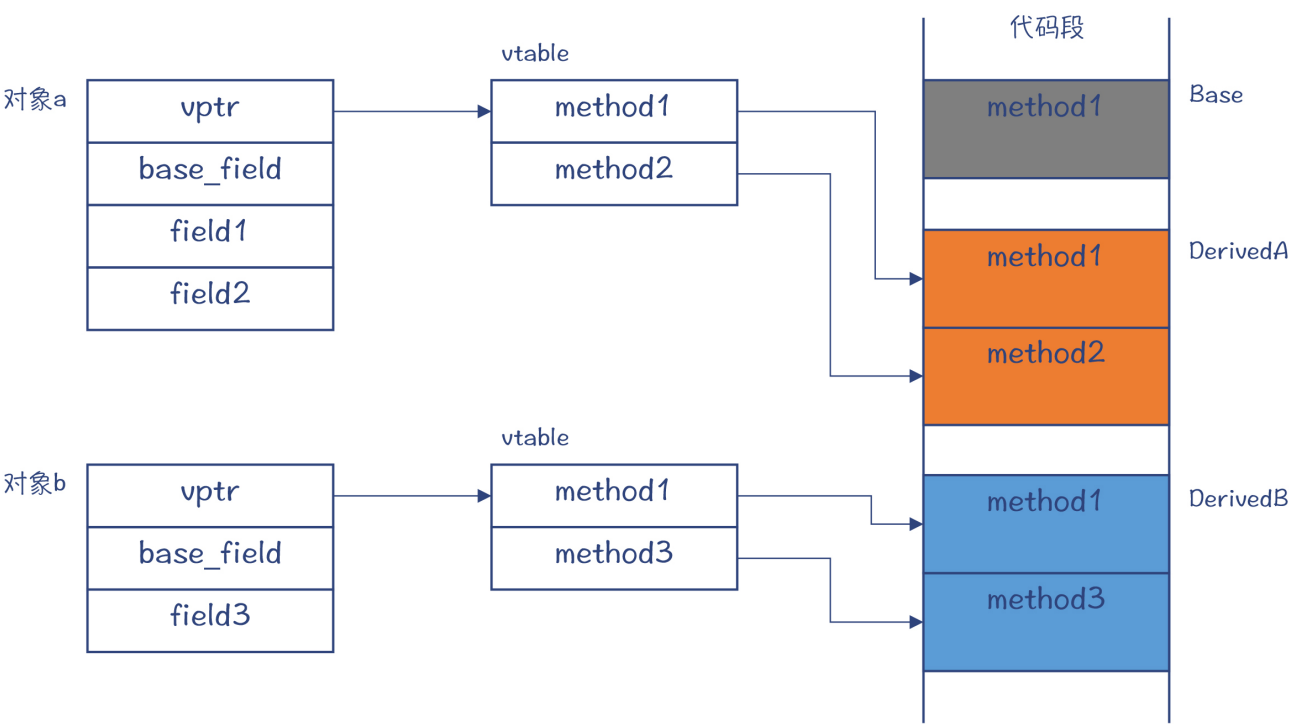


图 4：vtable 和动态绑定的原理

在对象 a 和 b 的 vtable 中，各个方法的指针都会指向这个对象实际调用的代码的入口地址。

那么编译器的工作就简单了。因为它可以基于对象的地址，得到 vtable 的地址，不用去管这个对象是哪个子类的实例。

然后，编译器只需要知道，当前调用的方法在 vtable 中的索引值就行了，这个索引值对于不同的子类的实例，也都是是一样的，但是具体指向的代码地址可能会不同。这种调用当然需要比静态绑定多用几条指令，因为对于静态绑定而言，只需要跳转到某个特定的方法的代码地址就行了，不需要通过 vtable 这个中间数据结构。不过，这种代价是值得的，因为它支持了面向对象中基于子类型的多态，从而可以让你编写更具通用性的代码。

图 4 中，我展示的只是一个示意结构。实际上，vtable 中还包含了一些其他信息，比如能够在运行时确定对象类型的信息，这些信息也可以用于在运行时进行指针的强制转换。

上面的例子是单一继承的情况。**对于多重继承，还会有多个 vptr 指针，指向多个 vtable。**你参考一下多重继承下的内存布局和指针转换的情况，应该就可以自行脑补出多重继承下的方法绑定技术。

我们接着回到 Java。Java 语言中调用这种可被重载的方法，生成的是 **invokevirtual 指令**，你在之前阅读 Java 字节码的时候一定遇到过这个指令。那么我现在可以告诉你，这个 virtual 就是沿用了 C++ 中虚方法的概念。


OK，理解了实现面向对象特性所需要做的一系列重点工作以后，我们来挑战一个难度更高的目标，这也是一个有技术洁癖的人会非常想实现的目标，就是让一切数据都表达为对象。

如何实现一切数据都是对象？


在 Java、C++ 和 Go 这些语言中，基础数据类型和对象类型是分开的。基础数据类型包括整型、浮点型等，它们不像对象类型那样有自己的方法，也没有类之间的继承关系。

这就给我们的编程工作造成了很多不便。比如，针对以上这两类不同数据类型的编程方式是不一致的。在 Java 里，你不能声明一个保存 int 数据的 ArrayList。在这种情况下，你只能使用 Integer 类型。

不过，Java 语言的编译器还是尽量提供了一些便利。举个例子，下面示例的两个语句都是合法的。在需要用到一个 Integer 对象的时候，你可以使用一个基础的 int 型数据；反过来亦然。

 复制代码

```
1 Integer b = 2;  
2 int c = b + 1;
```

在研究  Java 编译器的时候，你已经发现它在语义分析阶段提供了自动装箱（boxing）和拆箱（unboxing）的功能。比如说，如果发现代码里需要的是一个 Integer 对象，而代码里提供的是一个 int 数据，那么程序就自动添加相应的 AST 节点，基于该 int 数据创建一个 Integer 对象，这就叫做**装箱功能**。反之呢，把 Integer 对象转换成一个 int 数据，就叫做**拆箱功能**。装箱和拆箱功能属于一种语法糖，它能让编程更方便一些。

说到这里，你可能会想到，**既然编译器可以实现自动装箱和拆箱，那么在 Java 语言里，是不是根本就不用提供基础数据类型了，全部数据都用对象表达行不行？**这样的话，语言的表达性会更好，我们写起程序来也更简单。

不过，现在要想从头修改 Java 的语法是不可能了。但也有其他基于 JVM 的语言做到了这一点，比如 Scala。在 Scala 里，所有数据都是对象，各个类型构成了一棵有着相同根节点的类型树。对于对象化的整型和浮点型数据，编译器可以把它们直接编译成 JVM 的基础数据类型。

可仅仅这样还不够。**在 Java 里面，需要自动装箱和拆箱机制，很大一部分原因是 Java 的泛型机制。**那些使用泛型的 List、Map 等集合类，只能保存对象类型，不能保存基础数据类型。但这对于非常大的一个集合来说，用对象保存整型数据要多消耗几倍的内存。那么，我们能否优化集合类的实现，让它们直接保存基础数据，而不是保存一个个整型对象的引用呢？

通过上一讲的学习，我们也知道了，Java 的泛型机制是通过**类型擦除**来实现的，所以集合类里面只能保存对象引用，无法保存基础数据。既然 JVM 平台缺省的类型擦除技术行不通，那么是否可以对类型参数是值类型的情况下做特殊处理呢？

这是可以做到的。你还记得，C++ 实现泛型采用的是元编程技术。那么在 JVM 平台上，你其实也可以**通过元编程技术，针对值类型生成不同的代码**，从而避免创建大量的小对象，降低内存占用，同时减少 GC 的开销。Scala 就是这么做的，它会通过注解技术来完成这项任务。如果你对 Scala 的具体实现机制感兴趣，可以参考 [这篇文章](#)。

课程小结

这一讲我通过面向对象这个话题，带你一起综合性地探讨了语言设计、编译器和运行时的多个知识点。你可以带走这几个关键知识点：

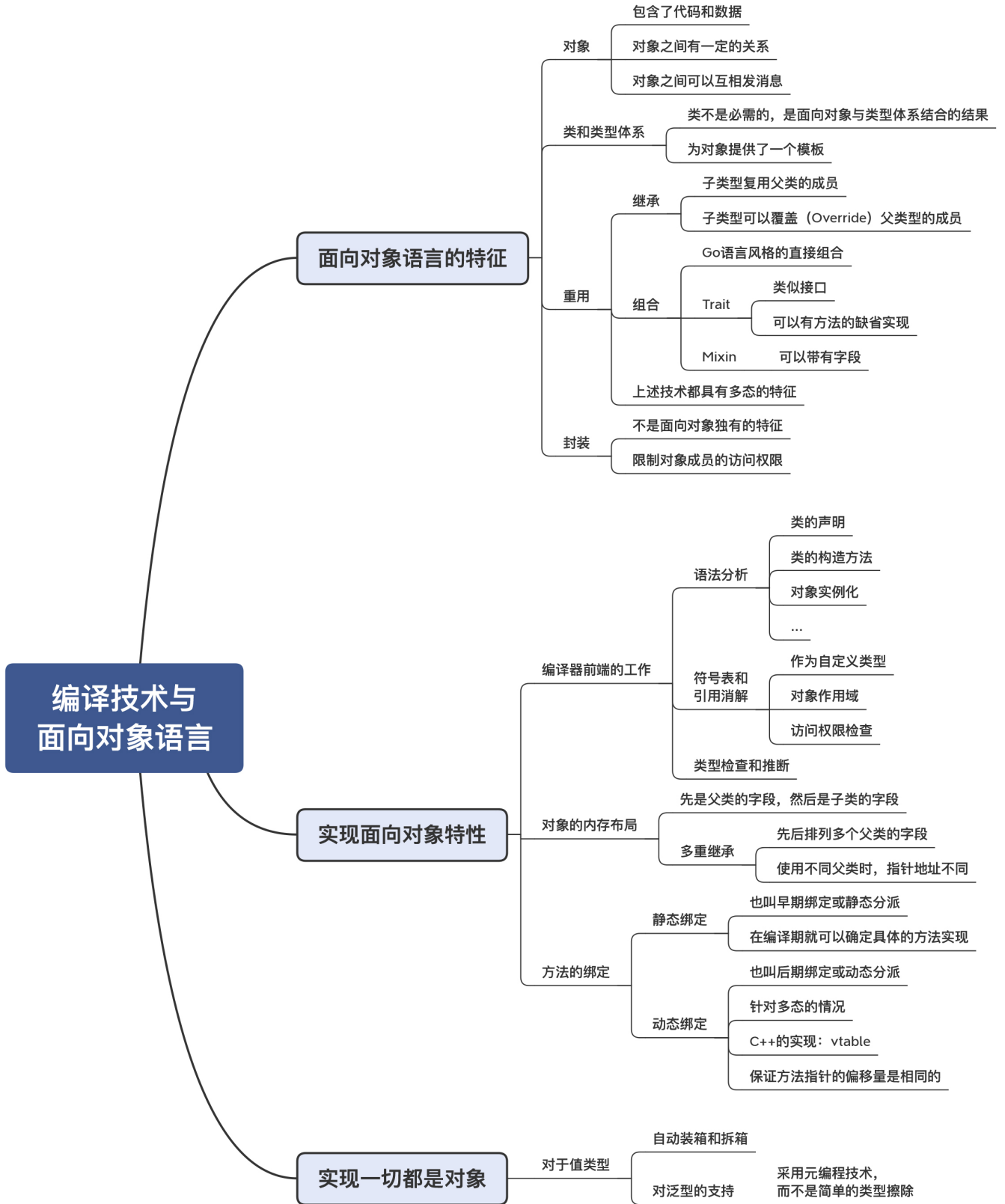
第一，要正确地理解面向对象编程的内涵，知道其实面向对象的语言可以有多种不同的设计选择，体现在类型体系、重用机制和信息封装等多个方面。对于不同的设计选择，你要都能够把它们解构，并对应到编译期和运行时的设计上。

第二，面向对象的各种特性，大多都是要在语义分析阶段进行检查和验证。

第三，对于静态编译的面向对象语言来说，理解其内存布局是关键。编译期要保证能够正确地访问对象的属性，并且巧妙地实现方法的动态绑定。

第四，如有可能，尽量让一切数据都表达为对象。让编译器完成自动装箱和拆箱的工作。

按照惯例，我把这节课的核心内容整理成了思维导图，供你参考和回顾知识点。



一课一思

有人曾经在技术活动上问 Java 语言之父詹姆斯·高斯林（James Gosling），如果重新设计 Java 语言，他会怎么做？他回答说，他会去掉 class，也就是会取消类的继承机制。那

么，对于你熟悉的面向对象语言，如果有机会重新设计的话，你会怎么建议？为什么？欢迎分享你的观点。

参考资料

1. 介绍 Trait 机制的 [论文](#)。
2. 在类型参数是值类型的情况下，Scala 以特殊的方式做实例化的 [文章](#)。

提建议

更多课程推荐

程序员的数学基础课

在实战中重新理解数学

黄申

LinkedIn 资深数据科学家



涨价倒计时 

今日秒杀 **¥79**, 9月11日涨价至 **¥129**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 37 | 高级特性（二）：揭秘泛型编程的实现机制

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。