第 5 章

面向接口编程

本章将领略在仓颉语言中面向对象编程的下一代进化，即面向接口编程的魅力，如图 5-1 所示。

图 5-1　USB 接口非常便利，随处可见

相信大家已经可以相当熟练地使用 USB 接口，例如 U 盘、USB 网银、通过 USB 给手机充电。

在现代的高级编程语言中，也越来越广泛地使用“接口”的概念，而且越来越魔幻，坚持看完本章，大家定会深度体验面向接口编程的无穷乐趣。

5.1　接口

从字面意思来讲，接口仅仅是一个定义，例如 USB-C 接口是一套 IEEE 标准规范。需要有具体的产品，例如 iPad、Huawei Mate 手机根据规范去实现 USB-C 接口。

在仓颉语言中，接口也只是一个虚拟的高级类型，定义了规范，而未实现。需要有一个具体的类型，例如 record、class 去实现接口。举个例子，使用 interface 关键字定义一个接口 I1，代码如下：

```
interface I1 {
    func aa() {
    }
}
```

接口 I1 内有一个空函数 aa()，里面没有任何代码。此时定义一个 record 类型实现了 I1 接口，代码如下：

```
record r1 <: I1 {
    func aa() {
        println("aa OK!")
    }
}
```

record 类型 r1 同样没有任何功能，但实现了 I1 接口，使用操作符“<:”，然后在内部只需写一个与 I1 规定的函数名一样的函数，并且有代码。再来定义两个接口，代码如下：

```
//第 5 章/two.cj
interface I2 {
    func bb() {

    }
}

interface I3 {
    func cc() {

    }
}
```

同样 I2、I3 内部各包含一个没有任何代码的空函数 bb、cc。虽然这 3 个接口看起来空空如也，但至少规定了其中 3 个函数的名字。现有一个实在的高级类型，例如上面的那个 record 类型，同时实现了 I1、I2、I3 接口，代码如下：

```
//第 5 章/combo.cj
record r1 <: I1 & I2 & I3 {
    func aa() {
        println("aa OK!")
    }

    func bb() {
        println("bb OK!")
    }

    func cc() {
        println("cc OK!")
    }
}
```

一种类型实现了 3 个接口，用“&”操作符把 3 个接口连接起来即可。以此类推，可以

实现无限多个接口，仓颉语言对比并无限制。

实现了接口以后，使用同样非常简单，代码如下：

```
//第 5 章/testcombo.cj
func main() {
    let r11 = r1()
    r11.aa()
    r11.bb()
    r11.cc()
}
```

运行结果如下：

```
aa OK!
bb OK!
cc OK!
```

从上面的代码可以看出，接口其实就是一个拥有实体的类型，实体可以认为是有具体代码的类型，并且对自身功能进行了扩展。

例如一台 HuaweiMate 手机拥有 USB-C 接口，如图 5-2 所示，从而可以很方便地与一台同样支持 USB-C 接口的 MacBook、小米计算机等进行连接，不再局限于同品牌。

图 5-2　接口的通用性

5.2　扩展既有类型

既然接口具有扩展实体类型的功能，那么可不可以对在仓颉语言中既有的读者已经熟悉的类型（例如字符串、数值等）进行扩展呢？答案是当然可以。

以上面的 3 个 I1、I2、I3 接口为例，可以对 String 进行扩展，让其实现 3 个接口，扩展时只需使用 extend 关键字，与 record 类型实现接口非常类似，代码如下：

```
//第 5 章/extend.cj
```

```
extend String <: I1 & I2 & I3{
    func aa() {
        println("aa OK!")
    }

    func bb() {
        println("bb OK!")
    }

    func cc() {
        println("cc OK!")
    }

}
```

具体使用的代码如下：

```
//第 5 章/extendtest.cj
func main() {

    let p = "sdf"
    p.aa()
    p.bb()
    p.cc()
}
```

输出的结果如下：

```
aa OK!
bb OK!
cc OK!
```

通过以上代码看出接口的便利所在了吧？在不需要更改既有类型的任何代码的基础上，接口可以对既有的类型扩展出任意想要的功能。这样在不会发生侵入性的代码变化产生副作用的前提下实现了功能扩展。

是不是颇有打破面向对象编程纯继承关系那么深度绑定的意味了？与此同时，接口还能为其他类型所有，并不为一种类型所垄断。

5.3　面向接口编程

试想学了仓颉以后，你可以在鸿蒙操作系统上开发一个类似阿凡达的大型 3A 级游戏，你还可以使玩家具备骑摩托、驾驶汽车、开飞机的能力，甚至可以驾着各种潘多拉星球的大翼龙飞行，如图 5-3 所示，非常刺激。

图 5-3　翼龙具有“可飞行”接口

这里的要点是，很多游戏中的 NPC（非玩家角色），例如车辆、飞行器、翼龙等，具备供玩家驾驶或者飞行的能力。

通常实现这些功能的方法是面向对象编程，可以把所有这些逻辑封装到一个基类，然后衍生一堆类去继承。基类可以内嵌驾驶和飞行逻辑。

然后可以创建一堆与车辆相关的 class，给翼龙添加一个飞行能力。经过编码后会发现轿车和摩托车共享很多功能，所以创造了一个自动车的基类，从而衍生出轿车和摩托车。

与此同时，还可以设计一个飞行器基类，从而衍生出固定翼飞机，如图 5-4 所示。

图 5-4　飞行器也具备可飞行接口

“完美的设计！”你可能会想。不过，这款游戏非常超前，车辆也有可能会飞，翼龙不能飞时，也可以临时充当骑行工具，在路上奔跑。让玩家具备更丰富的游戏体验，在资源紧张

时充分利用手头工具的潜能。

这时就会出现新的问题，如何让车具备飞行器的功能，或如何让翼龙具备自动车的功能呢？

你可能会想要创建另一个基类，合成这两个基类的功能。无论如何，并没有简单的方法可以实现这种合并。

此时需要一种新的思维：面向接口编程。

什么是面向接口编程？简单来讲，接口可以让你把相似的函数、属性进行组合。在仓颉语言中，所有的类型都可以实现接口，包括 class、enum、record 类型。不过其中只有 class 具备继承功能，所以理论上，可以不使用 class 就可以实现类似继承的功能。

在仓颉语言中接口的最大优点是可以多重组合。实现一种类似混血继承、外骨骼增强型效果，而不必有强绑定的父子继承关系。

如此一来，代码就可以实现模块化了。使用模块化的接口去定义一个大程序的所有功能。

当需要一个新功能的时候不用去增加一个新对象，而是加上一个新接口即可，这样非常轻量。就好比你想要听懂英文，不需要找老师学上好久，只需购买一个现在已经非常好用的同声 AI 翻译机，你不需要改变自己，时间成本极大地降低了。当需要懂法语时，把机器升级一下即可。

有了以上的认知，就可以把这些基类替代成各种接口。有了这些接口，就可以创建一个同时实现自动车和飞行接口的飞行汽车，听起来是个绝妙的思路。

5.4　定义基础接口

首先定义一个龙（Dragon），代码如下：

```
interface Dragon {
    prop let name : String
    prop let canFly: Bool
}
```

直接使用 interface 来表示一个抽象的 Dragon，赋予两个属性，即 name 和 canFly，变量前加上 prop 代表属性，属性是一个特殊的函数，对变量进行取值和设值的包装。

接下来是可飞行（Flyable），代码如下：

```
interface Flyable {
    prop let speed: Float64
}
```

很简单，只有一个速度属性。

在使用接口之前，开发者可能都要从 Flyable 这个基类开始，作后续相关依赖此基类的继承，但是在面向接口编程中，任何功能都是从接口开始。这个技术允许封装功能的雏形，

但不需要一个基类，取而代之的是基口。

你可能已经感觉到了，这样的方法，可以让自定义高级类型时整个结构显得十分灵活。

5.5 实现接口的类型

使用 record 而不是 class 类型来定义飞的龙，代码如下：

```
//第 5 章/flydragon1.cj
record FlyDragon <: Dragon & Flyable {
    var frequency: Float64
    var amp: Float64

    prop let name : String {
        get() { "Fly Dragon" }
    }

    prop let canFly : Bool {
        get()  { true }
    }

    prop let speed : Float64 {
        get()  {  5.0 * frequency * amp }
    }

}
```

以上代码定义了一个新 record 类型，同时实现了 Dragon 和 Flyable 接口，因为三者皆是常量属性，所以只有 get()取值代码的函数体。速度 speed 是一个由频率（frequency）和振动幅度（amp）变量相乘再与 5 相乘组合出来的属性。

因为飞龙必须是会飞的，所以直接取 true，名字为 FlyDragon。

有了上面的例子，现在可以用来定义具体的结构，例如恐龙（Dinosaur）和翼龙（Pterosaur）的结构，代码如下：

```
//第 5 章/dinosaur1.cj
record Dinosaur <: Dragon {
    prop let name : String {
        get() { "恐龙" }
    }

     prop let canFly : Bool {
        get()  { false }
    }
```

```
}
```

恐龙默认为不会飞的，所以取值是 false，但是如果是恐龙中的翼龙，那就得会飞了。不过注意，这里翼龙与恐龙就不是继承关系了，翼龙可以理解成“会飞的龙”，而不是“会飞的恐龙”。

所以从代码的角度来看，翼龙的组成发生了重大变化，普通的龙只要实现“飞行”的接口就可以认为是翼龙了，而不一定非要先成为恐龙。

这就很有意思，长久以来，人们认为只有鸟会飞，会飞的一定是鸟。如果看到一个人在飞，则可能是个带有翅膀的鸟人（幻想出来的生物），但是当人类发明飞机后，发现人没有翅膀也能飞，于是飞行的概念就发生了重大的变化，大大加速了人类文明的进步。

所以翼龙的代码如下：

```
//第 5 章/pterosaur.cj
record Pterosaur <: Dragon & Flyable {
    var v : Float64

    prop let name : String {
        get() { "翼龙" }
    }

    prop let canFly : Bool {
        get()  { true }
    }

    prop let speed : Float64 {
        get()  { v }
    }

}
```

现在开始来生成一只翼龙，代码如下：

```
//第 5 章/ptest.cj
func main() {
    let p1 = Pterosaur(v: 40.5)

    println(p1.name)
    println(p1.speed)

    if (p1.canFly) {
        println("${p1.name}是会飞的龙。")
    } else {
        println("${p1.name}不会飞。")
```

```
        }
    }
```

生成一个翼龙时，直接将速度 v 设置为 40.5，speed 属性并未类似上面的 FlyDragon 中那样对频率乘振幅的算式进行包装，所以直接返回 v 的值。

输出的结果如下：

```
翼龙
40.500000
翼龙是会飞的龙。
```

此时你会发现，每只翼龙的名字都相同，这是因为之前设置的是常量。那么我只需将 Dragon 接口中的 name 定义为变量，便可更改翼龙的名字，代码如下：

```
interface Dragon {
    prop var name : String
    prop let canFly: Bool
}
```

对应的 Dinosaur 和 Pterosaur 中的 name 实现，加上设置字段 set()即可。另外需要注意的是，接口中的属性都是对变量的包装，本质上是一个函数。

所以如果要让属性可设置，则需要在 record 内部有一个变量进行对应（name 对应的是 pname），代码如下：

```
//第 5 章/pterosaur2.cj
record Pterosaur <: Dragon & Flyable {
    var v = 0.0
     var pname = "翼龙"

    prop var name : String {
        get() { pname }
        set(newValue){ pname = newValue }
    }

    prop let canFly : Bool {
        get()  { true }
    }

    prop let speed : Float64 {
        get()  { v }
    }
}
```

测试代码如下：

```
//第 5 章/ptest2.cj
func main() {
    var p1 = Pterosaur(v: 40.5)
    p1.name = "末日翼龙 2047"

    println(p1.name)
    println(p1.speed)


    if (p1.canFly) {
        println("${p1.name}是会飞的龙。")
    } else {
        println("${p1.name}不会飞。")
    }
}
```

输出的结果如下：

```
末日翼龙 2047
40.500000
末日翼龙 2047 是会飞的龙。
```

细心的你可能已发现，翼龙的 pname 同样可以设置值，代码如下：

```
p1.name = "末日翼龙 2047"
p1.pname = "翼龙 222"

println(p1.name)
```

如果按这样的顺序执行代码，则 pname 的值会覆盖直接对 name 设置的值，输出的结果如下：

```
翼龙 222
```

这样就变成了名字打架，到底以哪个为准无所适从了，这时很显然要把内部的 pname 隐藏起来，一切以接口的字段为准。那么如何隐藏内部变量的设置呢？可以标记为 private，即私有，表示仅可在一种类型内部进行操作，实例化后是不可访问的，这样就避免了直接访问，通过属性访问可以增加一些合法性或逻辑性检查等，代码如下：

```
//第 5 章/pterosaur3.cj
record Pterosaur <: Dragon & Flyable {
    private var v = 0.0
    private var pname = "翼龙"

    prop var name : String {
        get() { pname }
```