Protocol和泛型的台前幕后

返回视频 (/series/protocol-and-generic)

具象函数和泛型函数的解析顺序 (https://www.boxueio.com/series/protocol-and-generic/ebook/190)

从隐式接口和编译期多态说起

Back to series (/series/protocol-and-generic)

字号
字号
默认主题
金色主题
暗色主题

自从有了泛型编程的概念，人们就开始不断尝试用这种方式去抽象一个算法或数据结构需要的核心接口集合。这和我们在面向对象编程中设计一个类接口的行为类似，却又有着根本的不同。为了理解泛型编程的思想，我们对比面向对象的方式，来看下面这个例子。

面向对象的方式

在面向对象的世界里，经常会发生下面这样的事情。假设我们有一个驾驶汽车的函数，它接受一个 `Car` 类型的参数，通过查看 `Car` 的文档我们知道，它有 `selfCheck / startEngine / shiftUp / go` 这4个方法：

```
func drive(_ car: Car) {
    if !car.selfCheck() {
        car.startEngine()
        car.shiftUp()
        car.go()
    }
}
```

但是通常，一个类支持的方法要比我们在某个具体的业务逻辑中使用的丰富：

```
class Car {
    func selfCheck() -> Bool {
        return true
    }
    func startEngine() {}
    func shiftUp() {}
    func go() {}

    func lightUp() {}
    func horn() {}
    func lock() {}
    // ...
}
```

因此，对于 `drive` 这个算法来说，虽然它只要求它的参数支持“自检”、“启动引擎”、“升挡”以及“前进”就好了。但是由于在声明里，参数的类型被定义成了 `Car` ，因此，无论需要与否，它必须是一个完整的 `Car` 类型。所谓完整，就是严格按照 `Car` 的规格包含所有的 `init` ， `deinit` 以及 `Car` 的所有方法和属性。

因此，即便一个类型实现了 `drive` 中的所有方法，但是只要它不在 `Car` 的继承体系里， `drive` 就无法正常工作。我们管这种面向对象的方式约定的接口，叫做explicit interface。

并且，由于 `Car` 是一个 `class` ，当我们传递 `Car` 的不同派生类时，各种方法的调用会在运行时被动态派发，这就是我们熟悉的运行时多态。

泛型编程的思维

但在泛型编程的世界里，故事就不同了。我们把之前的 `drive` 方法先改成这样：

```
func drive<T>(_ car: T) {
    if !car.selfCheck() {
        car.startEngine()
        car.shiftUp()
        car.go()
    }
}
```

从字面上看，意思是说， drive 接受一个 T 类型的参数，只要这个类型支持了“自检”、“启动引擎”、“升挡”以及“前进”这4个操作，就可以把车开走。当然，和C++中的泛型编程不同，在语法上，Swift要求我们明确把刚才这个要求表达出来，而不能仅仅通过 drive 的实现隐式表达这个要求。而这，就是 protocol 的作用：

```
protocol Drivable {
    func selfCheck() -> Bool
    func startEngine()
    func shiftUp()
    func go()
}
```

然后，在 drive 的实现里，我们也要明确的指定这个要求：

```
func drive<T: Drivable>(_ car: T) {
    if !car.selfCheck() {
        car.startEngine()
        car.shiftUp()
        car.go()
    }
}
```

现在，对于 drive 这个算法来说，它的要求就比之前面向对象的版本精确多了。它唯一的要求，就是类型支持算法需要的4个方法就好了，至于这个类型的对象如何初始化，如何被回收，有什么属性，统统没有约定。因此，我们管通过泛型方式约定的接口，叫做implicit interface。

然后，假设现在我们有两个独立的 class ，表示两类不同的汽车：

```
class Roadster: Drivable {
    func selfCheck() -> Bool {
        return true
    }
    func startEngine() {}
    func shiftUp() {}
    func go() {}
}

class SUV: Drivable {
    func selfCheck() -> Bool {
        return true
    }
    func startEngine() {}
    func shiftUp() {}
    func go() {}
}
```

当我们分别对它们的对象调用 drive 时：

```
drive(Roadster())
drive(SUV())
```

和面向对象中的运行时多态不同，泛型编程中调用方法的选择是在编译期完成的。编译器会根据参数的类型在正确的类中选择要调用的方法。这种行为，叫做编译期多态。

What's next?

通过这一节的内容，我们了解了泛型编程中最重要的编程思想。它和面向对象一样，都可以定义接口和实现多态。对于面向对象来说，接口是显式的，是基于类型定义和方法签名的，多态是发生在运行时的；而对于泛型编程，接口则是隐式的，是为了支持算法实现的，多态则是发生在编译期的。

当然，对于上面的两个 drive 调用，你也可以说，这不就跟重载了两个接受不同类型参数的 drive 函数一样么？它们的确有相似之处，在下一节，我们就来讨论重载函数和泛型函数之间的关系。

◀ 返回视频

(/series/protocol-and-generic)

具象函数和泛型函数的解析顺序 ▶

(https://www.boxueio.com/series/protocol-and-generic/ebook/190)

职场漂泊的你，每天多学一点。

从开发、测试到运维，让技术不再成为你成长的绊脚石。我们用打磨产品的精神去传播知识，把最新的移动开发技术，通过简单的图表， 清晰的视频，简明的文字和切实可行的例子一一向你呈现。让学习不仅是一种需求，也是一种享受。

泊学动态

一个工作十年PM终创业的故事（二） (https://www.boxueio.com/after-the-full-upgrade-to-swift3)
Mar 4, 2017

人生中第一次创业的"10有" (https://www.boxueio.com/founder-chat)
Jan 9, 2016

猎云网采访报道泊学 (http://www.lieyunwang.com/archives/144329)
Dec 31, 2015

What most schools do not teach (https://www.boxueio.com/what-most-schools-do-not-teach)
Dec 21, 2015

一个工作十年PM终创业的故事（一） (https://www.boxueio.com/founder-story)
May 8, 2015

泊学相关

关于泊学 ›

加入泊学 ›

泊学用户隐私以及服务条款 (HTTPS://WWW.BOXUEIO.COM/TERMS-OF-SERVICE)

版权声明 (HTTPS://WWW.BOXUEIO.COM/COPYRIGHT-STATEMENT)

联系泊学

Email: 10[AT]boxue.io (mailto:10@boxue.io)

QQ: 2085489246

2017 © Boxue, All Rights Reserved. 京ICP备15057653号-1 (http://www.miibeian.gov.cn/) 京公网安备 11010802020752号 (http://www.beian.gov.cn/portal/registerSystemInfo?recordcode=11010802020752)

友情链接 SwiftV (http://www.swiftv.cn) | Seay信息安全博客 (http://www.cnseay.com) | Swift.gg (http://swift.gg/) | Laravist (http://laravist.com/) | SegmentFault (https://segmentfault.com) | 靛青K的博客 (http://blog.dianqk.org/)