继承

4.2

这是一篇社区协同翻译的文章，你可以点击右边区块信息里的『改进』按钮向译者提交改进建议。

一个类可以 *继承* 另一个类的方法，属性以及其他特性。当一个类继承另一个类时，继承类称为 *子类*，被继承类称为 *父类*。在 Swift 中，继承是类类型区别于其他类型的基本特性。

Swift 中的子类可以调用和访问父类的方法，属性和下标，还可以重写这些方法，属性和下标来优化或修改它们的行为。Swift 会通过检查重写的定义和父类的定义是否匹配来帮助确保重写是正确的。

子类还可以给继承的属性添加属性观察器，以便在属性值发生变化时得到通知。属性观察器可以被添加到任何属性，不管它原始定义是储存属性还是计算属性。

定义一个基类

不继承任何类的类被称为 *基类* 。

> 注意
>
> Swift 中的类并不继承自一个统一的类。定义类时如果不指定父类那么该类自动成为基类。

下面的代码样例定义了一个基类 `Vehicle` 。这个基类定义了一个存储属性 `currentSpeed` ，其默认值为 `0.0` （推断该属性类型为 `Double` ）。和一个只读的 `String` 类型的计算属性 `description` ， `description` 使用 `currentSpeed` 的值来创建车辆的描述。

基类 `Vehicle` 还定义了一个空方法 `makeNoise` 。这个方法之后会被 `Vehicle` 的子类重写：

```
class Vehicle {
```

```
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }
    func makeNoise() {
        // 空方法，不是所有车辆都发出噪音
    }
}
```

你可以使用 *初始化语法* 来创建一个 `Vehicle` 的实例，用类型名并紧跟小括号来表示：

```
let someVehicle = Vehicle()
```

你可以访问已经创建的 `Vehicle` 实例的 `description` 属性来打印人类可读的车辆当前速度的描述：

```
print("Vehicle: \(someVehicle.description)")
// Vehicle: traveling at 0.0 miles per hour
```

`Vehicle` 类定义了车辆共有的特性，但其本身并没有太大作用。为了使其更加有用，你需要优化它来描述更具体的车辆类型。

子类化

子类化 是在已有类的基础上创建新类的行为。 子类可以继承父类的特性，然后对其进行更改。你还可以为子类添加新的特性。

为了表示一个类拥有父类，声明子类时要在后面写上父类的类名，使用冒号分割：

```
class SomeSubclass: SomeSuperclass {
    // 子类在这里定义
}
```

下面的例子中定义了一个叫 `Bicycle` 的类，它继承于父类 `Vehicle` ：

```
class Bicycle: Vehicle {
    var hasBasket = false
}
```

这个新的类 `Bicycle` 会自动获取 `Vehicle` 类的所有特性，如 `currentSpeed`

Virle 翻译于 5个月前

0 重译

由 RecherJ 审阅

、 `description` 属性和它的方法 `makeNoise()` 。

除了继承的特性外， `Bicycle` 类中还定义了一个新的存储属性 `hasBasket` ，它的初始值是 `false` （推断出该属性为 `Bool` 类型）.

默认情况下，你新创建的任何 `Bicycle` 实例（自行车）都没有basket（篮子）。 `Bicycle` 实例创建之后，你可以将 `hasBasket` 属性修改为 `true` 来表示是否有 `basket` ：

```
let bicycle = Bicycle()
bicycle.hasBasket = true
```

你也可以修改 `Bicycle` 实例继承的 `currentSpeed` 属性，还可以查询继承的 `description` 属性：

```
bicycle.currentSpeed = 15.0
print("Bicycle: \(bicycle.description)")
// Bicycle: traveling at 15.0 miles per hour
```

子类本身也是可以被继承的，下面的例子为 `Bicycle` 创建了一个名为「tandem」的双座自行车子类：

```
class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}
```

`Tandem` 类继承了 `Bicycle` 类中所有的属性和方法，同时 `Bicycle` 类继承了 `Vehicle` 类中所有的属性和方法。 `Tandem` 类还添加了一个新的存储属性 `currentNumberOfPassengers` ，初始值为 `0` 。

如果你创建了一个 `Tandem` 类的实例， 则可以使用任何新创建的属性和继承的属性， 还可以查询从 `Vehicle` 类继承的只读属性 `description` ：

```
let tandem = Tandem()
tandem.hasBasket = true
tandem.currentNumberOfPassengers = 2
tandem.currentSpeed = 22.0
print("Tandem: \(tandem.description)")
// Tandem: traveling at 22.0 miles per hour
```

Virle 翻译于 5个月前

0 重译

由 Aufree 审阅

重写

一个子类可以对实例方法、类方法、实例属性、类属性或下标进行自定义实现，如果没有自定义则会从超类继承，这就是所谓的 *重写* 。

重写从父类继承的特性，你需要在定义重写时添加 `override` 前缀。这表明你打算重写一个特性并且没有使用错误的匹配定义。意外的重写会导致不可预料的行为，且任何没有使用 `override` 关键词修饰的重写声明在编译代码时会被标记为错误。

`override` 关键词也提醒 Swift 编译器去检查你重写的这个类的超类（或它的某个父类）是否有匹配重写版本的声明。这个检查确保你的重写定义是正确无误的。

访问父类的方法、属性和下标

当你在子类中重写方法、属性或下标时，有时把父类的实现作为重写的一部分是很有用的。例如，你可以优化现有的实现，或将修改后的值存储在现有的继承变量中。

在适当的情况下，应该用 `super` 前缀访问父类的方法、属性或下标：

- 一个名为 `someMethod()` 的重写方法可以在实现中通过 `super.someMethod()` 来调用父类的 `someMethod()` 方法。
- 一个名为 `someProperty` 的属性可以在重写 getter 或 setter 的实现中通过 `super.someProperty` 访问父类的 `someProperty` 属性。
- 一个 `someIndex` 的重写下标可以在重写下标的实现中通过 `super[someIndex]` 访问父类的相同下标。

重写方法

你可以通过在子类中重写继承的实例或类型方法来对方法的实现进行修改。

下面的例子定义了一个 `Vehicle` 的新子类 `Train`，在 `Train` 中重写了继承自 `Vehicle` 的 `makeNoise()` 方法：

```
class Train: Vehicle {
    override func makeNoise() {
        print("Choo Choo")
    }
}
```

如果你创建一个 `Train` 的新实例并调用它的 `makeNoise()` 方法，你可以看到调用的是子类 `Train` 中的该方法：

Virle 翻译于 5个月前

0 重译

由 Aufree 审阅

wzshare 翻译于 5个月

0 重译

由 Aufree 审阅

wzshare 翻译于 5个月

0 重译

由 Aufree 审阅

```
let train = Train()
train.makeNoise()
// 输出 "Choo Choo"
```

重写属性

可以重写继承的实例或类型属性，以便为该属性提供自己定义的 getter 和 setter，或给重写的属性添加属性观察器观测先前属性值的改变。

重写属性的 Getters 和 Setters

无论继承的属性原来是存储属性还是计算属性，都可以提供自定义的 getter（如果 setter 适用，也包括 setter）来重写*任何*继承属性。子类不知道继承的属性是存储属性还是计算属性，子类只知道继承的属性具有特定的名称和类型。你必须始终声明要重写的属性的名称和类型，以使编译器能够检查你重写的属性是否与具有相同名称和类型的父类属性匹配。

通过在子类属性中重写 getter 和 setter，可以将继承的只读属性重写为读写属性，但是，你不能将继承的读写属性重写为只读属性。

> 注意
>
> 如果你重写属性的 setter 就必须同时重写属性的 getter。如果你不想在重写 getter 中修改继承属性的值，你可以简单地在 getter 中返回 `super.someProperty`，其中 `someProperty` 是你想要重写的属性名称。

下面的例子中定义了一个新的类 `Car`，它是 `Vehicle` 的子类。`Car` 类声明了一个存储属性 `gear`，默认值为 `1`。`Car` 还重写了从 `Vehicle` 中继承的 `description` 属性，用来提供包括当前档位的自定义描述：

```
class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \(gear)"
    }
}
```

重写的 `description` 属性首先调用了 `super.description`，它返回的是 `Vehicle` 类中的 `description` 属性。然后，`Car` 类中重写的 `description` 版本在末尾添加了一些额外的文本，用来描述当前的档位：

如果你创建了一个 `Car` 类的实例并且设置了 `gear` 和 `currentSpeed` 属性，你

wzshare 翻译于 5个月

0 重译

由 Aufree 审阅

Virle 翻译于 5个月前

0 重译

由 Aufree 审阅

就能看到它的 `description` 属性返回了 `Car` 类中重写过的描述：

```
let car = Car()
car.currentSpeed = 25.0
car.gear = 3
print("Car: \(car.description)")
// Car: traveling at 25.0 miles per hour in gear 3
```

重写属性观察者

你可以用属性重写特性给一个继承属性增加属性观察者。 无论这个属性的初始值是多少，当它的值改变时你将会收到通知。更多有关属性观察者的信息，参阅 属性观察者.

> 注意
>
> 你不能给常量存储属性或只读属性增加属性观察者。 因为这些属性值不能被修改，所以它是不能提供 `willSet` 或 `didSet` 的重写实现。
>
> 当然，你不能为同一个属性同时提供 `setter` 重写和 `didSet` 观察者。 如果你想观察这个属性值的改变，并且你已经为这个属性提供了一个重写的 `setter` 方法，那么你能在这个自定义 `setter` 方法里观察到它任何值的改变。

下面定义了一个叫 `AutomaticCar` 的新类， 它是 `Car` 的子类。 `AutomaticCar` 表示一个能自动变速的汽车类，它能基于当前速度自动选择一个合适的档位：

```
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}
```

当你更改 `AutomaticCar` 实例的 `currentSpeed` 属性值时，这个属性的 `didSet` 观察者将会观察到值的改变并根据新速度给 `gear` 属性设置一个合适的档位。按照换算规则，档位是新 `currentSpeed` 值除以 `10` 后四舍五入取整再 + `1` 的值。 例如，速度 `35.0` 将代表 `4` 档位：

```
let automatic = AutomaticCar()
```

```
automatic.currentSpeed = 35.0
print("AutomaticCar: \(automatic.description)")
// 自动汽车:  当前行驶的速度是 35.0 mph,  档位为 4
```

防止重写

你可以通过标记方法、属性或下标为 *final* 来防止它被重写。通过在方法、属性或下标前添加关键字 `final` （比如 `final var` 、 `final func` 、 `final class func` 和 `final subscript` ）来完成此操作。

任何重写子类中的 final 方法、属性或下标的尝试都会在编译时报错。添加到类扩展中的方法、属性或下标页可以在扩展中标记为 final。

你可以通过在类的定义中 `class` 关键字前添加 `final` 修饰符将整个类标记为 final，比如 `final class` 。任何对标记为 final 的类进行继承的子类都会在编译时报错。

上一篇 下一篇

点赞

参与译者：5

更多职位

讨论数量: 0

只看当前版本讨论

发起讨论

暂无话题~