继承

快速检索 [点击展开]

一个类可以从另一个类继承方法、属性和其他的特性。当一个类从另一个类继承的时候，继承的类就是所谓的*子类*，*而*这个类继承的类被称为*父类*。在 Swift 中，继承与其他类型不同的基础分类行为。

在 Swift 中类可以调用和访问属于它们父类的方法、属性和下标脚本，并且可以提供它们自己重写的方法，属性和下标脚本来定义或修改它们的行为。Swift 会通过检查重写定义都有一个与之匹配的父类定义来确保你的重写是正确的。

类也可以向继承的属性添加属性观察器，以便在属性的值改变时得到通知。可以添加任何属性监视到属性中，不管它是被定义为存储还是计算属性。

定义一个基类

任何不从另一个类继承的类都是所谓的*基类*。

> 注意
>
> Swift 类不会从一个通用基类继承。你没有指定特定父类的类都会以基类的形式创建。

下面的栗子定义了一个叫做 `Vehicle` 的基类。这个基类定义了一个称为 `currentSpeed` 的存储属性，使用默认值 `0.0`（推断为一个 `Double` 类型的属性）。 `currentSpeed` 属性的值被用在一个称为 `description` 的 `String` 只读计算属性来创建一个 `vehicle` 的描述。

`Vehicle` 基类也定义了一个称为 `makeNoise` 的方法。这个方法实际上

不会为这个 `Vehicle` 基类的实例做任何事，但是稍后它可以被 `Vehicle` 的子类自定义：

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour
"
    }
    func makeNoise() {
        // do nothing - an arbitrary vehicle doesn't necess
arily make a noise
    }
}
```

你使用初始化语法创建了一个新的 `Vehicle` 实例，写为 **类型名** 后面跟着一个空括号：

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

在创建了一个新的 `Vehicle` 实例之后，你可以访问它的 `description` 属性来输出一个人类可读的汽车当前速度的描述：

```
print("Vehicle: \(someVehicle.description)")

// Vehicle: traveling at 0.0 miles per hour
```

`Vehicle` 类为任意的车辆定义了共同的特征，但是对它本身没有太大用处。为了让它更有用，你需要重定义它来描述更具体的车辆种类。

子类

*子类*是基于现有类创建新类的行为。子类从现有的类继承了一些特征，你可以重新定义它们。你也可以为子类添加新的特征。

为了表明子类有父类，要把子类写在父类的前面，用冒号分隔：

```
class SomeSubclass: SomeSuperclass {
    // subclass definition goes here
}
```

下面的例子定义了一个称为 `Bicycle` 的子类，继承自 `Vehicle` ：

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

新的 `Bicycle` 类自动获得了 `Vehicle` 的所有特征，例如它的 `currentSpeed` 和 `description` 属性以及 `makeNoise()` 方法。

除了继承的特征， `Bicycle` 类定义了一个新的存储属性 `hasBasket` ，并且默认值为 `false`（属性的类型被推断为 `Bool` ）。

默认情况下，任何你新建的 `Bicycle` 实例都都不会有篮子。在 `Bicycle` 类的实例创建之后，你可以将它的 `hasBasket` 属性设置为 `true` ：

```
let bicycle = Bicycle()

bicycle.hasBasket = true
```

你也可以在 `Bicycle` 类实例中修改继承而来的 `currentSpeed` 属性，或是查询实例中继承的 `description` 属性：

```
bicycle.currentSpeed = 15.0

print("Bicycle: \(bicycle.description)")

// Bicycle: traveling at 15.0 miles per hour
```

子类本身也可以被继承。下个栗子创建了一个 `Bicycle` 的子类，称为”tandem”的两座自行车：

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

`Tandem` 继承了 `Bicycle` 中所有的属性和方法，也继承了 `Vehicle` 的所有属性和方法。 `Tandem` 子类也添加了一个新的称为 `currentNumberOfPassengers` 的存储属性，并且有一个默认值 `0` ：

```
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
```

重写

子类可以提供它自己的实例方法、类型方法、实例属性，类型属性或下标脚本的自定义实现，否则它将会从父类继承。这就所谓的*重写*。

要重写而不是继承一个特征，你需要在你的重写定义前面加上 `override` 关键字。这样做说明你打算提供一个重写而不是意外提供了一个相同定义。意外的重写可能导致意想不到的行为，并且任何没有使用 `override` 关键字的重写都会在编译时被诊断为错误。

`override` 关键字会执行 Swift 编译器检查你重写的类的父类(或者父类

的父类)是否有与之匹配的声明来供你重写。这个检查确保你重写的定义是正确的。

访问父类的方法、属性和下标脚本

当你为子类提供了一个方法、属性或者下标脚本时，有时使用现有的父类实现作为你重写的一部分是很有用的。比如说，你可以重新定义现有实现的行为，或者在现有继承的变量中存储一个修改过的值。

你可以通过使用 `super` 前缀访问父类的方法、属性或下标脚本，这是合适的：

- 一个命名为 `someMethod()` 的重写方法可以通过 `super.someMethod()` 在重写方法的实现中调用父类版本的 `someMethod()` 方法；
- 一个命名为 `someProperty` 的重写属性可以通过 `super.someProperty` 在重写的 `getter` 或 `setter` 实现中访问父类版本的 `someProperty` 属性；
- 一个命名为 `someIndex` 的重写下标脚本可以使用 `super[someIndex]` 在重写的下标脚本实现中访问父类版本中相同的下标脚本。

重写方法

你可以在你的子类中重写一个继承的实例或类型方法来提供定制的或替代的方法实现。

下面的栗子定义了一个新的 `Vehicle` 子类，称为 `Train` ，它重写了 `Train` 继承自 `Vehicle` 的 `makeNoise()` 方法：

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

```
}
```

如果你创建了一个新的 `Train` 实例并且调用它的 `makeNoise()` 方法，你可以看到 `Train` 子类版本的方法被调用了：

```
let train = Train()

train.makeNoise()

// prints "Choo Choo"
```

重写属性

你可以重写一个继承的实例或类型属性来为你自己的属性提供你自己自定义的 getter 和 setter ，或者添加属性观察器确保当底层属性值改变时来监听重写的属性。

重写属性的GETTER和SETTER

你可以提供一个自定义的Getter(和Setter，如果合适的话)来重写*任意*继承的属性，无论在最开始继承的属性实现为储属性还是计算属性。继承的属性是存储还是计算属性不对子类透明——它仅仅知道继承的属性有个特定名字和类型。你必须声明你重写的属性名字和类型，以确保编译器可以检查你的重写是否匹配了父类中有相同名字和类型的属性。

你可以通过在你的子类重写里为继承而来的只读属性添加Getter和Setter来把它用作可读写属性。总之，你不能把一个继承而来的可读写属性表示为只读属性。

> 注意
>
> 如果你提供了一个setter作为属性重写的一部分，你也就必须为重写提供一个getter。如果你不想在重写getter时修改继承属性的值，那么你可以简单通过从getter返回 `super.someProperty` 来传递继承的值， `someProperty` 就是你重写的那个属性的名字。

下面的栗子定义了一个叫做 `Car` 的新类，它是 `Vehicle` 的子类。`Car` 类引入了一个新的存储属性gear，并且有一个默认的整数值 `1` 。`Car` 类也重写了继承自 `Vehicle` 的 `description` 属性，来提供自定义的描述，介绍当前的档位：

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

`description` 属性的重写以调用 `super.description` 开始，它返回了 `Vehicle` 类的 `description` 属性。 `Car` 类的 `description` 随后就添加了一些额外的文本到描述的末尾以提供关于当前档位的信息。

如果你创建一个 `Car` 类的实例并且设置它的 `gear` 和 `currentSpeed` 属性，你就可以看到它的 `description` 属性在 `Car` 类的定义里返回了定制的描述：

```
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
```

重写属性观察器

你可以使用属性重写来为继承的属性添加属性观察器。这就可以让你在继承属性的值改变时得到通知，无论这个属性最初如何实现。关于属性观察器的更多信息，移步属性观察器（此处应有链接）。

> 注意
>
> 你不能给继承而来的常量存储属性或者只读的计算属性添加属性观察器。这些属性的值不能被设置，所以提供 `willSet` 或 `didSet` 实现作为重写的一部分也是不合适的。
>
> 也要注意你不能为同一个属性同时提供重写的setter和重写的属性观察器。如果你想要监听属性值的改变，并且你已经为那个属性提供了一个自定义的setter，那么你从自定义的setter里就可以监听任意值的改变。

下面的例子定义了一个叫做 `AutomaticCar` 的新类，它是 `Car` 的子类。`AutomaticCar` 类代表一辆车有一个自动的变速箱，可以根据当前的速度自动地选择一个合适的档位：

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

无论你在什么时候设置了 `AutomaticCar` 实例的 `currentSpeed` 属性，属性的 `didSet` 观察器都会设置实例的 `gear` 属性为新速度设置一个合适的档位。具体地说，属性观察器选择的档位就是新的 `currentSpeed` 值除以 `10` ，四舍五入到最近整数，加 `1` 。速度是 `35.0` 就对应 `4` ：

```
let automatic = AutomaticCar()

automatic.currentSpeed = 35.0

print("AutomaticCar: \(automatic.description)")

// AutomaticCar: traveling at 35.0 miles per hour in gear 4
```

阻止重写

你可以通过标记为*终点*来阻止一个方法、属性或者下标脚本被重写。通过在方法、属性或者下标脚本的关键字前写 `final` 修饰符(比如 `final var` ， `final func` ， `final class func` ， `fianl subscript`)。

任何在子类里重写终点方法、属性或下标脚本的尝试都会被报告为编译时错误。你在扩展中添加到类的方法、属性或下标脚本也可以在扩展的定义里被标记为终点。

你可以通过在类定义中在 `class` 关键字前面写 `final` 修饰符(`final class`)标记一整个类为终点。任何想要从终点类创建子类的行为都会被报告一个编译时错误。
