遗产

在本页

一个类可以*继承*另一个类的方法，属性和其他特征。当一个类从另一个类继承时，继承类被称为*子类*，并且它继承的类被称为它的*超类*。继承是区分Swift中的其他类型的基本行为。

Swift中的类可以调用和访问属于它们超类的方法，属性和下标，并且可以提供它们自己的重写版本的这些方法，属性和下标以改进或修改它们的行为。Swift通过检查覆盖定义是否具有匹配的超类定义来帮助确保覆盖是正确的。

类还可以将属性观察器添加到继承的属性中，以便在属性值更改时得到通知。属性观察者可以被添加到任何属性，而不管它最初是否定义为存储属性或计算属性。

定义一个基类

任何不从另一个类继承的类都称为*基类*。

> 注意
>
> Swift类不会从通用基类继承。您在未指定超类的情况下定义的类自动成为基础类，供您进行构建。

下面的例子定义了一个名为的基类`Vehicle`。这个基类定义了一个名为的存储属性`currentSpeed`，默认值为`0.0`（推断属性类型`Double`）。该`currentSpeed`属性的值由被`String`称为`description`创建车辆描述的只读计算属性使用。

所述`Vehicle`基类还定义了一个称为方法`makeNoise`。这个方法实际上并没有为一个基本`Vehicle`实例做任何事情，但是会被`Vehicle`后面的子类自定义：

```
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 necessarily make a noise
    }
}
```

您将创建一个`Vehicle`带有*初始化程序语法*的新实例，该实例将写入类型名称后跟空括号：

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

创建新`Vehicle`实例后，您可以访问其`description`属性以打印车辆当前速度的可读描述：

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

该`Vehicle`课程定义了任意车辆的共同特征，但本身并没有多大用处。为了使其更有用，您需要对其进行改进以描述更具体的车辆类型。

子类

*子类化*是基于现有类的新类的行为。子类继承现有类的特征，然后可以对其进行优化。您还可以向子类添加新的特征。

为了表明子类具有超类，请在超类名称前面写入子类名称，并用冒号分隔：

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

```
}
```

以下示例定义了一个名为的子类Bicycle，其超类为Vehicle：

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

新的Bicycle类自动获得所有的特性Vehicle，例如它的currentSpeed和description特性及其makeNoise()方法。

除了它继承的特性之外，Bicycle该类还定义了一个新的存储属性，hasBasket默认值为false（推断Bool该属性的类型）。

默认情况下，Bicycle您创建的任何新实例都不会有篮子。在创建该实例后，您可以将该hasBasket属性设置true为特定Bicycle实例：

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

您还可以修改继承的currentSpeed一个财产Bicycle实例，查询实例的继承description财产：

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

子类本身可以被分类。下一个示例创建了Bicycle一个称为“串联”的双座自行车的子类：

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

Tandem继承所有的属性和方法Bicycle，继而从中继承所有的属性和方法Vehicle。该Tandem子类还添加了一个名为的新存储属性currentNumberOfPassengers，默认值为0。

如果您创建了一个实例Tandem，则可以使用它的任何新的和继承的属性，并查询description它所继承的只读属性Vehicle：

```
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

一个子类可以提供它自己的自定义实现，它实际上是从超类继承的实例方法，类型方法，实例属性，类型属性或下标。这被称为*覆盖*。

要覆盖否则会被继承的特性，可以使用override关键字为您的覆盖定义加前缀。这样做明确说明您打算提供覆盖并且没有提供错误的匹配定义。意外覆盖会导致意外的行为，并且override在编译代码时将没有关键字的任何覆盖都诊断为错误。

该override关键字还会提示Swift编译器检查您的重写类的父类（或其父类）是否具有与您为重写提供的声明相匹配的声明。此检查可确保您的覆盖定义是正确的。

访问超类方法，属性和下标

当您为子类提供方法，属性或下标覆盖时，使用现有的超类实现作为覆盖的一部分有时很有用。例如，您可以改进现有实现的行为，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，您可以使用super前缀访问方法，属性或下标的超类版本：

- 被重写的方法`someMethod()`可以`someMethod()`通过`super.someMethod()`在重写的方法实现中调用超类的方法来调用。
- 被称为覆盖的属性`someProperty`可以访问的超类版本`someProperty`作为`super.someProperty`压倒一切的getter或se
- 重写的下标`someIndex`可以`super[someIndex]`从重写的下标实现中访问相同下标的超类版本。

重写方法

您可以重写继承的实例或类型方法，以在您的子类中提供该方法的定制或替代实现。

下面的例子定义了一个新的`Vehicle`被调用的子类`Train`，它覆盖了继承的`makeNoise()`方法： `TrainVehicle`

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

如果你创建一个新的实例`Train`并调用它的`makeNoise()`方法，你可以看到该方法的`Train`子类版本被调用：

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

重写属性

您可以覆盖继承的实例或类型属性以为该属性提供您自己的自定义getter和setter，或者添加属性观察器以使重写属性能够观察基础属性值何时更改。

重写属性获取者和设置者

无论继承的属性是作为源存储还是计算属性实现的， 您都可以提供自定义getter（和setter，如果适用）覆盖*任何*继承的属性。继承属性的存储或计算性质不为子类所知 - 它只知道继承属性具有特定的名称和类型。您必须始终声明您正在覆盖的属性的名称和类型，以使编译器能够检查您的覆盖是否与具有相同名称和类型的超类属性相匹配。

您可以通过在子类属性覆盖中提供getter和setter来呈现继承的只读属性作为读写属性。但是，您不能将继承的读写属性显示为只读属性。

> 注意
>
> 如果您提供一个setter作为属性重写的一部分，则还必须为该重写提供一个getter。如果您不想在重写的getter中修改继承的属性值，则可以简单地通过`super.someProperty`从getter 返回继承的值，其中`someProperty`是您正在覆盖的属性的名称。

以下示例定义了一个名为的新类`Car`，该类是其子类`Vehicle`。该`Car`课程介绍了新的存储属性调用`gear`，用默认整数值`1`。的`Car`类也覆盖了`description`它从继承属性`Vehicle`，以提供包括当前齿轮定制描述：

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

该`description`属性的重写通过调用开始`super.description`，它将返回`Vehicle`该类的`description`属性。然后该`Car`课程的版本`description`会在本说明的最后添加一些额外的文字，以提供有关当前装备的信息。

如果你创建的实例`Car`类，并设置它`gear`和`currentSpeed`属性，你可以看到它的`description`属性返回中定义的定制描述`Car`类：

```
let car = Car()
```

```
car.currentSpeed = 25.0
car.gear = 3
print("Car: \(car.description)")
// Car:
```

压倒性的财产观察员

您可以使用属性重写来将属性观察器添加到继承的属性。这使您可以在继承属性的值发生更改时得到通知，而不管该属性最初是如何实现的。有关财产观察员的更多信息，请参阅财产观察员。

> 注意
>
> 您不能将属性观察器添加到继承的常量存储属性或继承的只读计算属性。这些属性的值不能被设置，因此提供一个`willSet`或一个`didSet`实现作为覆盖的一部分是不合适的。
>
> 还要注意，你不能同时为同一个属性提供重写的setter和重写属性观察者。如果您想观察对属性值的更改，并且您已经为该属性提供了自定义设置器，则可以简单地观察自定义设置器中的任何值更改。

以下示例定义了一个名为的新类`AutomaticCar`，该类是其子类`Car`。该`AutomaticCar`班表示带有自动变速箱的汽车，该变速箱根据当前速度自动选择适合的档位：

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

当你设置`currentSpeed`一个的属性`AutomaticCar`实例属性的`didSet`观察者设置实例的`gear`属性齿轮的新速度一个合适的选择。具体来说，属性观察者会选择一个新的`currentSpeed`值除以该值`10`，向下舍入到最接近的整数加上`1`。`35.0`产生一个齿轮的速度`4`：

```
let automatic = AutomaticCar()
automatic.currentSpeed = 35.0
print("AutomaticCar: \(automatic.description)")
// AutomaticCar: traveling at 35.0 miles per hour in gear 4
```

防止覆盖

您可以通过将方法，属性或脚标标记为*final*来防止方法，属性或脚标被覆盖。通过写做到这一点`final`的方法，属性，或标的介绍人关键字之前修饰符（如`final var`，`final func`，`final class func`，和`final subscript`）。

任何试图覆盖子类中的最终方法，属性或下标的尝试都会报告为编译时错误。您在扩展中添加到类中的方法，属性或下标也可以在扩展的定义中标记为final。

您可以通过在其类定义（）中`final`的`class`关键字前写入修饰符来将整个类标记为final `final class`。任何对最终类进行子类化的尝试都会报告为编译时错误。