翻译：Hawstein　校对：menlongsheng

继承（Inheritance）

本页包含内容：

- 定义一个基类（Base class）
- 子类生成（Subclassing）
- 重写（Overriding）
- 防止重写

一个类可以继承（*inherit*）另一个类的方法（methods），属性（property）和其它特性。当一个类继承其它类时，继承类叫子类（*subclass*），被继承类叫超类（或父类，*superclass*）。在 Swift 中，继承是区分「类」与其它类型的一个基本特征。

在 Swift 中，类可以调用和访问超类的方法，属性和附属脚本（subscripts），并且可以重写（override）这些方法，属性和附属脚本来优化或修改它们的行为。Swift 会检查你的重写定义在超类中是否有匹配的定义，以此确保你的重写行为是正确的。

可以为类中继承来的属性添加属性观察器（property observer），这样一来，当属性值改变时，类就会被通知到。可以为任何属性添加属性观察器，无论它原本被定义为存储型属性（stored property）还是计算型属性（computed property）。

定义一个基类（Base class）

不继承于其它类的类，称之为基类（*base calss*）。

注意：

Swift 中的类并不是从一个通用的基类继承而来。如果你不为你定义的类

指定一个超类的话，这个类就自动成为基类。
下面的例子定义了一个叫Vehicle的基类。这个基类声明了两个对所有车辆都通用的属性（numberOfWheels和maxPassengers）。这些属性在description方法中使用，这个方法返回一个String类型的，对车辆特征的描述：

```
class Vehicle {
    var numberOfWheels: Int
    var maxPassengers: Int
    func description() -> String {
        return "\(numberOfWheels) wheels; up to \
(maxPassengers) passengers"
    }
    init() {
        numberOfWheels = 0
        maxPassengers = 1
    }
}
```

Vehicle类定义了构造器（*initializer*）来设置属性的值。构造器会在构造过程一节中详细介绍，这里我们做一下简单介绍，以便于讲解子类中继承来的属性如何被修改。

构造器用于创建某个类型的一个新实例。尽管构造器并不是方法，但在语法上，两者很相似。构造器的工作是准备新实例以供使用，并确保实例中的所有属性都拥有有效的初始化值。

构造器的最简单形式就像一个没有参数的实例方法，使用init关键字：

```
init() {
    // 执行构造过程
}
```

如果要创建一个Vehicle类的新实例，使用构造器语法调用上面的初始化器，即类名后面跟一个空的小括号：

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

这个Vehicle类的构造器为任意的一辆车设置一些初始化属性值

（`numberOfWheels = 0`和`maxPassengers = 1`）。

`Vehicle`类定义了车辆的共同特性，但这个类本身并没太大用处。为了使它更为实用，你需要进一步细化它来描述更具体的车辆。

子类生成（Subclassing）

子类生成（*Subclassing*）指的是在一个已有类的基础上创建一个新的类。子类继承超类的特性，并且可以优化或改变它。你还可以为子类添加新的特性。

为了指明某个类的超类，将超类名写在子类名的后面，用冒号分隔：

```
class SomeClass: SomeSuperclass {
    // 类的定义
}
```

下一个例子，定义一个更具体的车辆类叫`Bicycle`。这个新类是在`Vehicle`类的基础上创建起来。因此你需要将`Vehicle`类放在`Bicycle`类后面，用冒号分隔。

我们可以将这读作：

“定义一个新的类叫`Bicycle`，它继承了`Vehicle`的特性”；

```
class Bicycle: Vehicle {
    init() {
        super.init()
        numberOfWheels = 2
    }
}
```

`Bicycle`是`Vehicle`的子类，`Vehicle`是`Bicycle`的超类。新的`Bicycle`类自动获得`Vehicle`类的特性，比如`maxPassengers`和`numberOfWheels`属性。你可以在子类中定制这些特性，或添加新的特性来更好地描述`Bicycle`类。

`Bicycle`类定义了一个构造器来设置它定制的特性（自行车只有2个轮子）。`Bicycle`的构造器调用了它父类`Vehicle`的构造器`super.init()`，以此确保在`Bicycle`类试图修改那些继承来的属性前`Vehicle`类已经初始化过它们了。

注意：

不像 Objective-C，在 Swift 中，初始化器默认是不继承的，见初始化器的继承与重写

`Vehicle`类中`maxPassengers`的默认值对自行车来说已经是正确的，因此在`Bicycle`的构造器中并没有改变它。而`numberOfWheels`原来的值对自行车来说是不正确的，因此在初始化器中将它更改为 2。

`Bicycle`不仅可以继承`Vehicle`的属性，还可以继承它的方法。如果你创建了一个`Bicycle`类的实例，你就可以调用它继承来的`description`方法，并且可以看到，它输出的属性值已经发生了变化：

```
let bicycle = Bicycle()
println("Bicycle: \(bicycle.description())")
// Bicycle: 2 wheels; up to 1 passengers
```

子类还可以继续被其它类继承：

```
class Tandem: Bicycle {
    init() {
        super.init()
        maxPassengers = 2
    }
}
```

上面的例子创建了`Bicycle`的一个子类：双人自行车（tandem）。`Tandem`从`Bicycle`继承了两个属性，而这两个属性是`Bicycle`从`Vehicle`继承而来的。`Tandem`并不修改轮子的数量，因为它仍是一辆自行车，有 2 个轮子。但它需要修改`maxPassengers`的值，因为双人自行车可以坐两个人。

注意：

子类只允许修改从超类继承来的变量属性，而不能修改继承来的常量属

性。
创建一个`Tandem`类的实例，打印它的描述，即可看到它的属性已被更新：

```
let tandem = Tandem()
println("Tandem: \(tandem.description())")
// Tandem: 2 wheels; up to 2 passengers
```

注意，`Tandem`类也继承了`description`方法。一个类的实例方法会被这个类的所有子类继承。

重写（Overriding）

子类可以为继承来的实例方法（instance method），类方法（class method），实例属性（instance property），或附属脚本（subscript）提供自己定制的实现（implementation）。我们把这种行为叫重写（*overriding*）。

如果要重写某个特性，你需要在重写定义的前面加上`override`关键字。这么做，你就表明了你是想提供一个重写版本，而非错误地提供了一个相同的定义。意外的重写行为可能会导致不可预知的错误，任何缺少`override`关键字的重写都会在编译时被诊断为错误。

`override`关键字会提醒 Swift 编译器去检查该类的超类（或其中一个父类）是否有匹配重写版本的声明。这个检查可以确保你的重写定义是正确的。

访问超类的方法，属性及附属脚本

当你在子类中重写超类的方法，属性或附属脚本时，有时在你的重写版本中使用已经存在的超类实现会大有裨益。比如，你可以优化已有实现的行为，或在一个继承来的变量中存储一个修改过的值。

在合适的地方，你可以通过使用`super`前缀来访问超类版本的方法，属性或附属脚本：

- 在方法`someMethod`的重写实现中，可以通过`super.someMethod()`来调用超类版本的`someMethod`方法。
- 在属性`someProperty`的 getter 或 setter 的重写实现中，可以通过`super.someProperty`来访问超类版本的`someProperty`属性。
- 在附属脚本的重写实现中，可以通过`super[someIndex]`来访问超类版本中的相同附属脚本。

重写方法

在子类中，你可以重写继承来的实例方法或类方法，提供一个定制或替代的方法实现。

下面的例子定义了`Vehicle`的一个新的子类，叫`Car`，它重写了从`Vehicle`类继承来的`description`方法：

```
class Car: Vehicle {
    var speed: Double = 0.0
    init() {
        super.init()
        maxPassengers = 5
        numberOfWheels = 4
    }
    override func description() -> String {
        return super.description() + "; "
            + "traveling at \(speed) mph"
    }
}
```

`Car`声明了一个新的存储型属性`speed`，它是`Double`类型的，默认值是`0.0`，表示“时速是0英里”。`Car`有自己的初始化器，它将乘客的最大数量设为5，轮子数量设为4。

`Car`重写了继承来的`description`方法，它的声明与`Vehicle`中的`description`方法一致，声明前面加上了`override`关键字。

`Car`中的`description`方法并非完全自定义，而是通过`super.description`使用了超类`Vehicle`中的`description`方法，然后再追加一些额外的信息，比如汽车的当前速度。

如果你创建一个`Car`的新实例，并打印`description`方法的输出，你就会发现描述信息已经发生了改变：

```
let car = Car()
println("Car: \(car.description())")
// Car: 4 wheels; up to 5 passengers; traveling at
0.0 mph
```

重写属性

你可以重写继承来的实例属性或类属性，提供自己定制的getter和setter，或添加属性观察器使重写的属性观察属性值什么时候发生改变。

重写属性的Getters和Setters

你可以提供定制的 getter（或 setter）来重写任意继承来的属性，无论继承来的属性是存储型的还是计算型的属性。子类并不知道继承来的属性是存储型的还是计算型的，它只知道继承来的属性会有一个名字和类型。你在重写一个属性时，必需将它的名字和类型都写出来。这样才能使编译器去检查你重写的属性是与超类中同名同类型的属性相匹配的。

你可以将一个继承来的只读属性重写为一个读写属性，只需要你在重写版本的属性里提供 getter 和 setter 即可。但是，你不可以将一个继承来的读写属性重写为一个只读属性。

注意：

如果你在重写属性中提供了 setter，那么你也一定要提供 getter。如果你不想在重写版本中的 getter 里修改继承来的属性值，你可以直接返回`super.someProperty`来返回继承来的值。正如下面的`SpeedLimitedCar`的例子所示。
以下的例子定义了一个新类，叫`SpeedLimitedCar`，它是`Car`的子类。类`SpeedLimitedCar`表示安装了限速装置的车，它的最高速度只能达到40mph。你可以通过重写继承来的`speed`属性来实现这个速度限制：

```
class SpeedLimitedCar: Car {
    override var speed: Double  {
    get {
```

```
            return super.speed
        }
        set {
            super.speed = min(newValue, 40.0)
        }
        }
}
```

当你设置一个`SpeedLimitedCar`实例的`speed`属性时，属性setter的实现会去检查新值与限制值40mph的大小，它会将超类的`speed`设置为`newValue`和`40.0`中较小的那个。这两个值哪个较小由`min`函数决定，它是Swift标准库中的一个全局函数。`min`函数接收两个或更多的数，返回其中最小的那个。

如果你尝试将`SpeedLimitedCar`实例的`speed`属性设置为一个大于40mph的数，然后打印`description`函数的输出，你会发现速度被限制在40mph：

```
let limitedCar = SpeedLimitedCar()
limitedCar.speed = 60.0
println("SpeedLimitedCar: \
(limitedCar.description())")
// SpeedLimitedCar: 4 wheels; up to 5 passengers;
traveling at 40.0 mph
```

重写属性观察器（Property Observer）

你可以在属性重写中为一个继承来的属性添加属性观察器。这样一来，当继承来的属性值发生改变时，你就会被通知到，无论那个属性原本是如何实现的。关于属性观察器的更多内容，请看属性观察器。

注意：

你不可以为继承来的常量存储型属性或继承来的只读计算型属性添加属性观察器。这些属性的值是不可以被设置的，所以，为它们提供`willSet`或`didSet`实现是不恰当。此外还要注意，你不可以同时提供重写的 setter 和重写的属性观察器。如果你想观察属性值的变化，并且你已经为那个属性提供了定制的 setter，那么你在 setter 中就可以观察到任何值变化了。

下面的例子定义了一个新类叫`AutomaticCar`，它是`Car`的子类。

AutomaticCar表示自动挡汽车，它可以根据当前的速度自动选择合适的挡位。AutomaticCar也提供了定制的description方法，可以输出当前挡位。

```
class AutomaticCar: Car {
    var gear = 1
    override var speed: Double {
    didSet {
        gear = Int(speed / 10.0) + 1
    }
    }
    override func description() -> String {
        return super.description() + " in gear \
(gear)"
    }
}
```

当你设置AutomaticCar的speed属性，属性的didSet观察器就会自动地设置gear属性，为新的速度选择一个合适的挡位。具体来说就是，属性观察器将新的速度值除以10，然后向下取得最接近的整数值，最后加1来得到档位gear的值。例如，速度为10.0时，挡位为1；速度为35.0时，挡位为4：

```
let automatic = AutomaticCar()
automatic.speed = 35.0
println("AutomaticCar: \(automatic.description())")
// AutomaticCar: 4 wheels; up to 5 passengers;
traveling at 35.0 mph in gear 4
```

防止重写

你可以通过把方法，属性或附属脚本标记为final来防止它们被重写，只需要在声明关键字前加上@final特性即可。（例如：@final var, @final func,@final class func,以及@final subscript）

如果你重写了final方法，属性或附属脚本，在编译时会报错。在扩展中，你添加到类里的方法，属性或附属脚本也可以在扩展的定义里标记为

final。

你可以通过在关键字`class`前添加`@final`特性（`@final class`）来将整个类标记为 final 的，这样的类是不可被继承的，否则会报编译错误。