Properties

Stored Properties

如果一结构（Structure）的实例（Instance）被声明给一个常量，不能修改实例中的属性（property），即使实例中的属性为变量。

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}

let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// this range represents integer values 0, 1, 2, and 3
rangeOfFourItems.firstValue = 6
// this will report an error, even though firstValue is a variable property
```

此行为是由于结构是值类型。当值类型的实例被标记为常量时，其所有属性也被标记为常量。

对于类（引用类型）而言，情况并非如此。如果您将引用类型的实例分配给常量，则仍然可以更改该实例的变量属性。

Lazy Stored Properties

当属性的初始值取决于某外部因素，并且在实例初始化完成之前并不知道该外部因素的值时，惰性属性很有用。

当属性的初始值需要复杂或计算量大的设置，直到真正需要时才应执行初始化，这时惰性属性也很有用。

使用Lazy Stored Properties应注意线程安全。

Computed Properties

计算属性实际上并没有存储值。相反，它们提供了一个getter和一个可选的setter来间接检索和设置其他属性和值。

```
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point { // 计算属性
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }
}
```

如果计算属性的setter没有为要设置的新值定义名称，则使用默认名称`newValue`。 这是`center`计算属性setter可简写为：

```
set {
    origin.x = newValue.x - (size.width / 2)
    origin.y = newValue.y - (size.height / 2)
}
```

如果getter的整个主体是单个表达式，则getter隐式返回该表达式。因此getter可简写为：

```
get {
    Point(x: origin.x + (size.width / 2), y: origin.y + (size.height / 2))
}
```

Read-Only Computed Properties

A computed property with a getter but no setter is known as a *read-only computed property*. A read-only computed property always returns a value, and can be accessed through dot syntax, but cannot be set to a different value.

> You must declare computed properties—including read-only computed properties—as variable properties with the `var` keyword, because their value is not fixed. The `let` keyword is only used for constant properties, to indicate that their values cannot be changed once they are set as part of instance initialization.

You can simplify the declaration of a read-only computed property by removing the `get` keyword and its braces:

```
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
        return width * height * depth
    }
}
let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
// Prints "the volume of fourByFiveByTwo is 40.0"
```

Property Observers

Property observers observe and respond to changes in a property's value. Property observers are called every time a property's value is set, even if the new value is the same as the property's current value.

对于继承的属性，可以通过在子类中重写该属性来添加属性观察器。对于自己定义的计算属性，请使用属性的设置器来观察和响应值更改，而不是尝试创建观察者。

可以选择在属性上定义这些观察者之一或全部：

- `willSet` is called just before the value is stored.
- `didSet` is called immediately after the new value is stored.

如果您实现了`willSet`观察者，则会将新的属性值作为常量参数传递。您可以为此参数指定一个名称，作为`willSet`实现的一部分。如果您未在实现中写入参数名称和括号，则该参数将使用默认参数名称`newValue`。

同样，如果您实现了`didSet`观察器，则会传递一个包含旧属性值的常量参数。您可以命名参数或使用默认参数名称`oldValue`。如果您在其自己的`didSet`观察器中为属性分配值，则分配的新值将替换刚刚设置的值。

> The `willSet` and `didSet` observers of superclass properties are called when a property is set in a subclass initializer, after the superclass initializer has been called. They are not called while a class is setting its own properties, before the superclass initializer has been called.

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
```

```
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// Added 536 steps
```

The `StepCounter` class declares a `totalSteps` property of type `Int`. This is **a stored property** with `willSet` and `didSet` observers.

The `didSet` observer is called after the value of `totalSteps` is updated.

> If you pass a property that has observers to a function as an in-out parameter, the `willSet` and `didSet` observers are always called. This is because of the copy-in copy-out memory model for in-out parameters: The value is always written back to the property at the end of the function. For a detailed discussion of the behavior of in-out parameters, see In-Out Parameters.

Property Wrappers

属性包装器在管理属性存储方式的代码与定义属性的代码之间增加了一层隔离。例如，如果您具有提供线程安全检查或将其基础数据存储在数据库中的属性，则必须在每个属性上编写该代码。使用属性包装器时，只需在定义包装器时编写一次管理代码，然后通过将其应用于多个属性来重用该管理代码。

To define a property wrapper, you make a structure, enumeration, or class that defines a `wrappedValue` property. In the code below, the `TwelveOrLess` structure ensures that the value it wraps always contains a number less than or equal to 12. If you ask it to store a larger number, it stores 12 instead.

```
@propertyWrapper
struct TwelveOrLess {
    private var number: Int
    init() { self.number = 0 }
    var wrappedValue: Int {
        get { return number }
        set { number = min(newValue, 12) }
    }
}
```

> The declaration for `number` in the example above marks the variable as `private`, which ensures `number` is used only in the implementation of `TwelveOrLess`. Code that's written anywhere else accesses the value using the getter and setter for `wrappedValue`, and can't use `number` directly. For information about `private`, see Access Control.

通过在属性定义前写wrapper's name将一个wrapper应用于属性：

```
struct SmallRectangle {
    @TwelveOrLess var height: Int
    @TwelveOrLess var width: Int
}

var rectangle = SmallRectangle()
print(rectangle.height)
// Prints "0"

rectangle.height = 10
print(rectangle.height)
// Prints "10"

rectangle.height = 24
print(rectangle.height)
// Prints "12"
```

当您将包装器应用于属性时，编译器将合成为包装器提供存储的代码和提供通过包装器访问属性的代码。您也可以编写使用属性包装器行为的代码，而无需利用特殊的属性语法。例如，这是先前代码清单中SmallRectangle的另一版本，该版本将其属性明确地包装在TwelveOrLess结构中，而不是在定义属性时使用@TwelveOrLess：

```
struct SmallRectangle {
    private var _height = TwelveOrLess()
    private var _width = TwelveOrLess()
    var height: Int {
        get { return _height.wrappedValue }
        set { _height.wrappedValue = newValue }
    }
    var width: Int {
        get { return _width.wrappedValue }
        set { _width.wrappedValue = newValue }
    }
}
```

The `_height` and `_width` properties store **an instance of the property wrapper**, `TwelveOrLess`. The getter and setter for `height` and `width` wrap access to the `wrappedValue` property.

Setting Initial Values for Wrapped Properties

The code in the examples above sets the initial value for the wrapped property by giving `number` an initial value in the definition of `TwelveOrLess`. Code that uses this property wrapper, can't specify a different initial value for a property that's wrapped by `TwelveOrLess`—for example, the definition of `SmallRectangle` can't give `height` or `width` initial values. To support setting an initial value or other customization, the property wrapper needs to add an initializer. Here's an expanded version of `TwelveOrLess` called `SmallNumber` that defines initializers that set the wrapped and maximum value:

```
@propertyWrapper
struct SmallNumber {
    private var maximum: Int
    private var number: Int

    var wrappedValue: Int {
        get { return number }
        set { number = min(newValue, maximum) }
    }

    init() {
        maximum = 12
        number = 0
    }
    init(wrappedValue: Int) {
        maximum = 12
        number = min(wrappedValue, maximum)
    }
    init(wrappedValue: Int, maximum: Int) {
        self.maximum = maximum
        number = min(wrappedValue, maximum)
    }
}
```

The definition of `SmallNumber` includes three initializers—`init()`, `init(wrappedValue:)`, and `init(wrappedValue:maximum:)`—which the examples below use to set the wrapped value and the maximum value.

When you apply a wrapper to a property and you don't specify an initial value, Swift uses the `init()` initializer to set up the wrapper. For example:

```
struct ZeroRectangle {
    @SmallNumber var height: Int
    @SmallNumber var width: Int
}

var zeroRectangle = ZeroRectangle()
print(zeroRectangle.height, zeroRectangle.width)
// Prints "0 0"
```

The instances of `SmallNumber` that wrap `height` and `width` are created by calling `SmallNumber()`. The code inside that initializer sets the initial wrapped value and the initial maximum value, using the default values of zero and 12. The property wrapper still provides all of the initial values, like the earlier example that used `TwelveOrLess` in `SmallRectangle`. Unlike that example, `SmallNumber` also supports writing those initial values as part of declaring the property.

When you specify an initial value for the property, Swift uses the `init(wrappedValue:)` initializer to set up the wrapper. For example:

```
struct UnitRectangle {
    @SmallNumber var height: Int = 1
    @SmallNumber var width: Int = 1
}

var unitRectangle = UnitRectangle()
print(unitRectangle.height, unitRectangle.width)
// Prints "1 1"
```

When you write `= 1` on a property with a wrapper, that's translated into a call to the `init(wrappedValue:)` initializer. The instances of `SmallNumber` that wrap `height` and `width` are created by calling `SmallNumber(wrappedValue: 1)`. The initializer uses the wrapped value that's specified here, and it uses the default maximum value of 12.

当您在自定义属性后的括号中写入参数时，Swift将使用接受这些参数的初始化程序来设置包装器。

For example, if you provide an initial value and a maximum value, Swift uses the `init(wrappedValue:maximum:)` initializer:

```
struct NarrowRectangle {
    @SmallNumber(wrappedValue: 2, maximum: 5) var height: Int
    @SmallNumber(wrappedValue: 3, maximum: 4) var width: Int
}

var narrowRectangle = NarrowRectangle()
print(narrowRectangle.height, narrowRectangle.width)
// Prints "2 3"

narrowRectangle.height = 100
narrowRectangle.width = 100
print(narrowRectangle.height, narrowRectangle.width)
// Prints "5 4"
```

By including arguments to the property wrapper, you can set up the initial state in the wrapper or pass other options to the wrapper when it's created. This syntax is the most general way to use a property wrapper. You can provide whatever arguments you need to the attribute, and they're passed to the initializer.

When you include property wrapper arguments, you can also specify an initial value using assignment. Swift treats the assignment like a `wrappedValue` argument and uses the initializer that accepts the arguments you include. For example:

```
struct MixedRectangle {
    @SmallNumber var height: Int = 1
    @SmallNumber(maximum: 9) var width: Int = 2
}

var mixedRectangle = MixedRectangle()
print(mixedRectangle.height)
// Prints "1"

mixedRectangle.height = 20
print(mixedRectangle.height)
// Prints "12"
```

The instance of `SmallNumber` that wraps `height` is created by calling `SmallNumber(wrappedValue: 1)`, which uses the default maximum value of 12. The instance that wraps `width` is created by calling `SmallNumber(wrappedValue: 2, maximum: 9)`.

Projecting a Value From a Property Wrapper

除了*wrapped value*，属性包装器还可以通过定义 *projected value*来公开其他功能，例如，管理对数据库的访问的属性包装器可以在其 *projected value*上公开*flushDatabaseConnection()*方法。

The name of the projected value is the same as the wrapped value, except it begins with a dollar sign (`$`).

In the `SmallNumber` example above, if you try to set the property to a number that's too large, the property wrapper adjusts the number before storing it. The code below adds a `projectedValue` property to the `SmallNumber` structure to keep track of whether the property wrapper adjusted the new value for the property before storing that new value.

```
@propertyWrapper
struct SmallNumber {
    private var number: Int
    var projectedValue: Bool
    init() {
        self.number = 0
        self.projectedValue = false
    }
    var wrappedValue: Int {
        get { return number }
        set {
            if newValue > 12 {
                number = 12
                projectedValue = true
            } else {
                number = newValue
                projectedValue = false
            }
        }
    }
}
struct SomeStructure {
    @SmallNumber var someNumber: Int
}
var someStructure = SomeStructure()

someStructure.someNumber = 4
print(someStructure.$someNumber) // 通过.$someNumber访问其projected value
// Prints "false"

someStructure.someNumber = 55
print(someStructure.$someNumber)
// Prints "true"
```

Writing `someStructure.$someNumber` accesses the wrapper's projected value.

A property wrapper can return **a value of any type** as its projected value.

A wrapper that needs to expose（公开） more information can return an instance of some other data type, or it can return `self` to expose the instance of the wrapper as its projected value.

When you access a projected value from code that's part of the type, like a property getter or an instance method, you can omit `self.` before the property name, just like accessing other properties. The code in the following example refers to the projected value of the wrapper around `height` and `width` as `$height` and `$width`:

```
enum Size {
    case small, large
}

struct SizedRectangle {
    @SmallNumber var height: Int
    @SmallNumber var width: Int

    mutating func resize(to size: Size) -> Bool {
        switch size {
        case .small:
            height = 10
            width = 20
        case .large:
            height = 100
            width = 100
        }
        return $height || $width
    }
}
```

The wrapper prevents the value of those properties from being larger than 12, and it sets the projected value to `true`, to record the fact that it adjusted their values. At the end of `resize(to:)`, the return statement checks `$height` and `$width` to determine whether the property wrapper adjusted either `height` or `width`.

Global and Local Variables

Stored variables, like stored properties, provide storage for a value of a certain type and allow that value to be set and retrieved.

However, you can also define *computed variables* and define observers for stored variables, in either a global or local scope. Computed variables calculate their value, rather than storing it, and they are written in the same way as computed properties.

> **Global constants and variables are always computed lazily**, in a similar manner to Lazy Stored Properties. Unlike lazy stored properties, global constants and variables do not need to be marked with the `lazy` modifier.
>
> **Local constants and variables are never computed lazily.**

Type Properties （类型属性）

Instance properties are properties that belong to an instance of a particular type. Every time you create a new instance of that type, it has its own set of property values, separate from any other instance.

You can also define properties that belong to the type itself, not to any one instance of that type. There will only ever be one copy of these properties, no matter how many instances of that type you create. These kinds of properties are called *type properties*.

> **Unlike stored instance properties, you must always give stored type properties a default value. This is because the type itself does not have an initializer that can assign a value to a stored type property at initialization time.**
>
> **Stored type properties are lazily initialized on their first access. They are guaranteed to be initialized only once, even when accessed by multiple threads simultaneously**, and they do not need to be marked with the `lazy` modifier.

In Swift, however, type properties are written as part of the type's definition, within the type's outer curly braces, and each type property is explicitly scoped to the type it supports.

You define type properties with the `static` keyword. **For computed type properties for class types, you can use the `class` keyword instead to allow subclasses to override the superclass's implementation.** The example below shows the syntax for stored and computed type properties:

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 1
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 6
    }
}
class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int { //子类可重写
        return 107
    }
}
```

> The computed type property examples above are for read-only computed type properties, but you can also define read-write computed type properties with the same syntax as for computed instance properties.

Querying and Setting Type Properties

```
print(SomeStructure.storedTypeProperty)
// Prints "Some value."
SomeStructure.storedTypeProperty = "Another value."
print(SomeStructure.storedTypeProperty)
// Prints "Another value."
print(SomeEnumeration.computedTypeProperty)
// Prints "6"
print(SomeClass.computedTypeProperty)
// Prints "27"
```