初始化

*初始化*是准备要使用的类，结构或枚举的实例的过程。该过程包括为该实例上的每个存储属性设置一个初始值，并在新实例准备好使用之前执行所需的任何其他设置或初始化。

您可以通过定义实现这个初始化过程*初始化*，这就像特殊的方法，可以被调用来创建特定类型的新实例。与Objective-C初始化器不同，Swift初始化器不返回值。它们的主要作用是确保类型的新实例在第一次使用前正确初始化。

类类型的实例也可以实现一个*deinitializer*，它在该类的实例被释放之前执行任何自定义清理。有关取消初始化程序的更多信息，请参阅取消初始化。

为存储属性设置初始值

在创建该类或结构的实例时， 类和结构*必须*将其存储的所有属性设置为适当的初始值。存储的属性不能处于不确定状态。

您可以在初始化程序中为存储的属性设置初始值，或者通过将默认属性值指定为属性定义的一部分。这些操作在以下各节中进行介绍。

> 注意
>
> 当您将默认值分配给存储的属性或在初始化程序中设置其初始值时，该属性的值将直接设置，而不调用任何属性观察器。

初始化器

调用*初始化*器来创建特定类型的新实例。以最简单的形式，初始化器就像一个没有参数的实例方法，使用`init`关键字编写：

```
init() {
    // perform some initialization here
}
```

下面的例子定义了一个叫做`Fahrenheit`存储以华氏温标表示的温度的新结构。该`Fahrenheit`结构有一个存储的属性`temperature`，它的类型为`Double`：

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
var f = Fahrenheit()
print("The default temperature is \(f.temperature)° Fahrenheit")
// Prints "The default temperature is 32.0° Fahrenheit"
```

该结构定义了一个单独的初始化器，`init`没有参数，它用一个数值`32.0`（华氏度的水凝固点）初始化储存的温度。

默认属性值

您可以在初始化程序中设置存储属性的初始值，如上所示。或者，指定*默认属性值*作为属性声明的一部分。通过在定义属性时指定初始值来指定默认属性值。

> 注意
>
> 如果属性始终采用相同的初始值，请提供默认值，而不是在初始化程序中设置值。最终结果是相同的，但默认值更紧密地将属性的初始化绑定到其声明。它使得初始化器更简洁，更清晰，并使您能够从其默认值推断出属

在本页

> 性的类型。如本章后面所述，默认值还使您更容易利用默认初始化程序和初始化程序继承。

您可以Fahrenhe

```
struct Fahrenheit {
    var temperature = 32.0
}
```

自定义初始化

您可以使用输入参数和可选属性类型定制初始化过程，也可以通过在初始化过程中分配常量属性来进行定制，如以下各节所述。

初始化参数

您可以提供*初始化参数*作为初始化程序定义的一部分，以定义自定义初始化过程的值的类型和名称。初始化参数与函数和方法参数具有相同的功能和语法。

以下示例定义了一个称为的结构`Celsius`，它存储以摄氏度表示的温度。该`Celsius`结构实现了两个名为`init(fromFahrenheit:)`and的自定义初始化程序`init(fromKelvin:)`，它使用不同温度值的值初始化结构的新实例：

```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}
let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
// boilingPointOfWater.temperatureInCelsius is 100.0
let freezingPointOfWater = Celsius(fromKelvin: 273.15)
// freezingPointOfWater.temperatureInCelsius is 0.0
```

第一个初始化器具有一个初始化参数，其参数标号为`fromFahrenheit`，参数名称为`fahrenheit`。第二个初始化器具有一个初始化参数，其参数标号为`fromKelvin`，参数名称为`kelvin`。两个初始化器都将其单个参数转换为相应的摄氏度值，并将该值存储在一个名为的属性中`temperatureInCelsius`。

参数名称和参数标签

与函数和方法参数一样，初始化参数可以同时具有在初始化器主体内使用的参数名称和调用初始化器时使用的参数标签。

但是，初始化程序在函数和方法所做的括号之前没有标识函数名称。因此，初始化程序参数的名称和类型在识别应该调用哪个初始化程序中扮演着特别重要的角色。因此，如果不提供一个参数，Swift会为初始化程序中的*每个*参数提供一个自动参数标签。

下面的例子定义了一个名为结构`Color`，具有三个恒定属性叫做`red`，`green`，和`blue`。这些属性之间存储一个值`0.0`，`1.0`用于指示颜色中红色，绿色和蓝色的数量。

`Color`为`Double`它的红色，绿色和蓝色组件提供了一个具有三种适当命名的类型参数的初始化程序。`Color`还提供了一个带有单个`white`参数的第二个初始化程序，该参数用于为所有三种颜色分量提供相同的值。

```
struct Color {
    let red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
```

```
        self.blue  = blue
    }
    init(white: Double) {

        green = white
        blue  = white
    }
}
```

Color通过为每个初始化参数提供命名值， 两个初始化器都可用于创建新实例：

```
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
let halfGray = Color(white: 0.5)
```

请注意，不使用参数标签就不能调用这些初始化器。参数标签必须始终在初始化程序中使用（如果已定义），并省略它们是编译时错误：

```
let veryGreen = Color(0.0, 1.0, 0.0)
// this reports a compile-time error - argument labels are required
```

不带参数标签的初始化参数

如果您不想为初始化参数使用参数标签，请为该参数编写一个下划线（_）而不是显式参数标签来覆盖默认行为。

下面是上述初始化参数Celsius示例的扩展版本，其中有一个额外的初始化程序，用于根据已经在摄氏温标中的值创建一个新实例： CelsiusDouble

```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
    init(_ celsius: Double) {
        temperatureInCelsius = celsius
    }
}
let bodyTemperature = Celsius(37.0)
// bodyTemperature.temperatureInCelsius is 37.0
```

初始化器调用Celsius(37.0)在其意图中是清楚的，而不需要参数标签。因此，编写该初始化程序是合适的，init(_ celsius: Double)以便可以通过提供未命名的Double值来调用它。

可选属性类型

如果您的自定义类型的存储属性在逻辑上被允许具有“无值” - 也许是因为它的值不能在初始化过程中设置，或者因为它允许在稍后的某个点上具有“无值” - 请声明属性*可选*类型。可选类型的属性会自动初始化为一个值nil，表示该属性在初始化期间故意打算具有“没有值”。

以下示例定义了一个名为的类SurveyQuestion，并具有一个String名为的可选属性response：

```
class SurveyQuestion {
    var text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
```

```
        print(text)
    }
}
let che
cheeseQuestion.ask()
// Prints "Do you like cheese?"
cheeseQuestion.response = "Yes, I do like cheese."
```

对调查问题的回答只有在被问及后才能知道，因此该response属性是用一种String?或“可选的String” 类型声明的。nil当一个新实例SurveyQuestion被初始化时，会自动分配一个默认值，意思是“没有字符串” 。

在初始化期间分配常量属性

只要在初始化完成时将其设置为一个确定的值，就可以在初始化过程中的任何时间点将值赋给常量属性。一旦一个常量属性被分配一个值，它就不能被进一步修改。

> 注意
>
> 对于类实例，可以在初始化期间仅通过引入它的类来修改常量属性。它不能被子类修改。

您可以修改SurveyQuestion上面的示例，以便为问题的text属性使用常量属性而不是变量属性，以指示在SurveyQuestion创建实例后问题不会更改。即使text属性现在是一个常量，它仍然可以在类的初始化程序中设置：

```
class SurveyQuestion {
    let text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        print(text)
    }
}
let beetsQuestion = SurveyQuestion(text: "How about beets?")
beetsQuestion.ask()
// Prints "How about beets?"
beetsQuestion.response = "I also like beets. (But not with cheese.)"
```

默认初始化程序

Swift 为任何结构或类提供了一个*默认的初始*值设定项，它为其所有属性提供默认值，并且不会提供至少一个初始化程序本身。默认初始化器只是创建一个新的实例，并将其所有属性设置为默认值。

这个例子定义了一个叫做的类ShoppingListItem，它将一个项目的名称，数量和购买状态封装在一个购物清单中：

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
    var purchased = false
}
var item = ShoppingListItem()
```

因为ShoppingListItem该类的所有属性都具有默认值，并且因为它是不带超类的基类，所以会ShoppingListItem自动获得默认的初始化器实现，该实现会创建一个新实例，并将其所有属性设置为默认值。（该name属性是一个可选String属性，因此它会自动接收默认值nil，即使此值未写入代码中。）上面的示例使用ShoppingListItem该类的默认初始化程序来创建具有初始化程序的类的新实例语法，写为ShoppingListItem()，并将这个新实例分配给一个名为的变量item。

结构类型的成员初始化程序

如果结构类型未定义任何自定义初始值设定项，则会自动接收成员初始值设定项，与默认初始值设定项不同，即使结构存储了

成员初始化程序是初始化新结构实例的成员属性的简写方法。新实例的属性的初始值可以通过名称传递给成员初始值设定项。

下面的例子定义了一个Size叫做“ widthand”的两个属性的结构height。Double通过分配一个默认值来推断这两个属性都是类型的0.0。

该Size结构自动接收init(width:height:)成员初始化程序，您可以使用它初始化新的Size实例：

```
struct Size {
    var width = 0.0, height = 0.0
}
let twoByTwo = Size(width: 2.0, height: 2.0)
```

值类型的初始化程序委托

初始化器可以调用其他初始化器来执行实例初始化的一部分。此过程称为*初始化器委派*，可避免跨多个初始化器复制代码。

对于值类型和类类型，初始化程序委托如何工作的规则以及允许使用何种形式的委托的规则不同。值类型（结构和枚举）不支持继承，因此它们的初始化器委派过程相对简单，因为它们只能委托给它们自己提供的另一个初始化器。但是，类可以继承其他类，如继承中所述。这意味着类需要额外的责任来确保它们继承的所有存储属性在初始化期间被分配一个合适的值。这些职责在下面的类继承和初始化中描述。

对于值类型，self.init在编写自己的自定义初始值设定项时，用于引用来自相同值类型的其他初始值设定项。您self.init只能从初始化程序中调用。

请注意，如果您为值类型定义了自定义初始值设定项，那么您将不再有权访问该类型的默认初始值设定项（或成员初始值设定项，如果它是结构体）。这个约束防止了一个情况，即在一个更复杂的初始化器中提供的附加基本设置被使用其中一个自动初始化器的人不小心规避了。

> 注意
>
> 如果您希望自定义值类型可以使用默认初始化程序和成员初始化程序以及您自己的自定义初始化程序进行初始化，请将自定义初始化程序写入扩展中，而不是作为值类型原始实现的一部分。有关更多信息，请参阅扩展。

以下示例定义了一个自定义Rect结构来表示一个几何矩形。的例子需要两个称为支撑结构Size和Point，两者都提供的默认值0.0对于所有其属性的：

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
```

您可以Rect通过三种方式之一来初始化下面的结构 - 通过使用其默认的零初始值origin和size属性值，通过提供特定的原点和大小，或通过提供特定的中心点和大小。这些初始化选项由作为Rect结构定义一部分的三个自定义初始值设定项表示：

```
struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
```

```
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

第一个Rect初始化程序，init()在功能上与默认的初始化程序相同，如果结构没有自己的自定义初始值设定项，该结构会收到它。这个初始化器有一个空体，由一对空的花括号表示{}。调用此初始化程序将返回一个Rect实例，其实例origin和size属性都使用其属性定义的默认值Point(x: 0.0, y: 0.0)以及Size(width: 0.0, height: 0.0)它们的属性定义进行初始化：

```
let basicRect = Rect()
// basicRect's origin is (0.0, 0.0) and its size is (0.0, 0.0)
```

第二个Rect初始化程序，init(origin:size:)在功能上与结构在没有自己的自定义初始化程序时会收到的成员初始化程序相同。这个初始化器简单地将参数值origin和size参数值分配给适当的存储属性：

```
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
                      size: Size(width: 5.0, height: 5.0))
// originRect's origin is (2.0, 2.0) and its size is (5.0, 5.0)
```

第三个Rect初始化init(center:size:)器稍微复杂一些。它通过基于center点和size值计算适当的原点开始。然后它调用（或*委托*）init(origin:size:)初始化程序，它将新的原点和大小值存储在适当的属性中：

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
                      size: Size(width: 3.0, height: 3.0))
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

在init(center:size:)初始化可能分配的新值origin，并size以相应的属性本身。但是，init(center:size:)初始化程序利用已经提供了该功能的现有初始化程序更方便（并且意图更清晰）。

> 注意
> 对于另一种方式来写这个例子中没有定义init()和init(origin:size:)初始化自己，看看扩展。

类继承和初始化

在初始化期间， 所有类的存储属性（包括类从其超类继承的任何属性）都*必须*分配一个初始值。

Swift为类类型定义了两种初始值设定项，以帮助确保所有存储的属性都获得初始值。这些被称为指定初始化器和便捷初始化器。

指定的初始化程序和便捷初始化程序

*指定的初始化器*是一个类的主要初始化器。指定的初始化程序完全初始化由该类引入的所有属性，并调用适当的超类初始化程序以继续超类链的初始化过程。

类通常只有极少数指定的初始化器，而一个类只有一个是很常见的。指定的初始化器是通过初始化发生的“漏斗”点，并且初始化过程通过它继续超类链。

每个班级必须至少有一个指定的初始化程序。在某些情况下，通过从超类继承一个或多个指定的初始化程序可满足此要求，如下面的“ 自动初始化程序继承”中所述。

*便捷初始化器*是次要的，支持一个类的初始化器。您可以定义一个便利初始值设定项，以便从与初始值设定项相同的类中调用指定初始值设定项，并将某些指定初始值设定项的参数设置为默认值。您还可以定义一个便利初始值设定器，为特定用例或输入值类型创建该类的实例。

如果你的班级不需要他们，你不必提供便利的初始化工具。创建便捷初始值设定项时，只要使用通用初始化模式的快捷方式可以节省时间或在意图中对类进行初始化。

指定和便捷初始化程序的语法

类的指定初始值设定项的编写方式与简单的初始值设定项的值类型相同：

```
init (parameters) {
    声明
}
```

便捷初始值设定项以相同样式编写，但convenience修饰符位于init关键字之前，以空格分隔：

```
方便的 init (参数) {
    声明
}
```

类类型的初始化程序委托

为了简化指定者和便捷初始值设定项之间的关系，Swift为初始值设定项之间的委托调用应用以下三条规则：

规则1
指定的初始化器必须从它的直接超类中调用指定的初始化器。

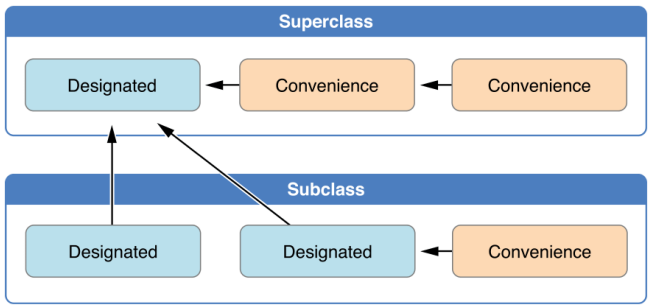
规则2
便捷初始值设定项必须调用*同一个*类中的另一个初始值设定项。

规则3
便利初始值设定程序最终必须调用指定的初始化程序。

记住这个简单的方法是：

- 指定的初始值必须始终委派了。
- 便利的初始化必须始终委派*跨越*。

这些规则如下图所示：

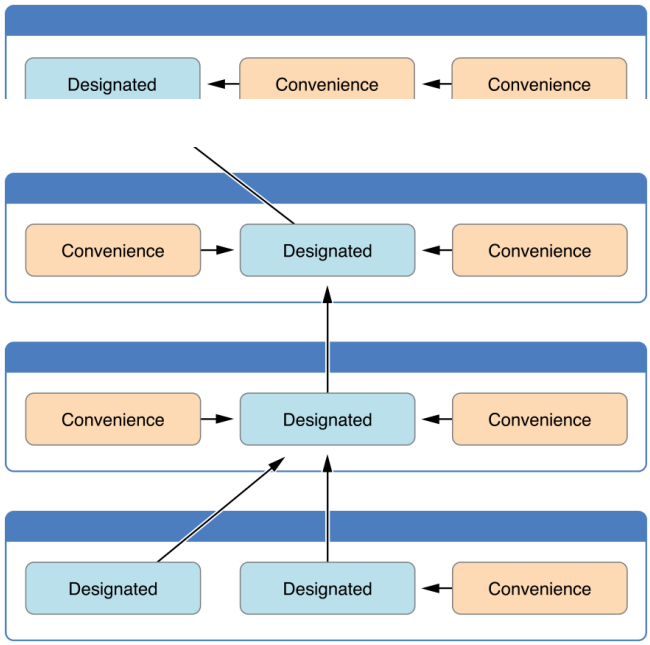


这里，超类有一个指定的初始化器和两个便利初始化器。一个便捷初始值设定项调用另一个便捷初始值设定项，它又调用单个指定的初始值设定项。这符合上面的规则2和3。超类本身没有进一步的超类，所以规则1不适用。

该图中的子类有两个指定的初始化程序和一个便利初始化程序。便利初始值设定程序必须调用两个指定的初始值设定项之一，因为它只能调用同一个类中的另一个初始值设定项。这符合上面的规则2和3。两个指定的初始化器都必须调用超类中的单个指定初始化器，以满足上面的规则1。

> 注意
>
> 这些规则不影响你的类的用户如何*创建*每个类的实例。上图中的任何初始化器均可用于创建它们所属类的完全初始化的实例。规则只影响你编写类的初始化器的实现。

下图显示了四个类的更复杂的类层次结构。它说明了这个层次结构中指定的初始化器如何作为类初始化的“漏斗”点，简化了链中类之间的相互关系：



两阶段初始化

Swift中的类初始化是一个两阶段过程。在第一阶段中，每个存储的属性由引入它的类来分配一个初始值。一旦确定了每个存储属性的初始状态，第二阶段开始，并且每个类都有机会在新实例被认为准备好使用之前进一步定制其存储的属性。

使用两阶段初始化过程使初始化安全，同时仍然为类层次结构中的每个类提供了完全的灵活性。两阶段初始化可防止在初始化之前访问属性值，并防止其他初始化程序意外地将属性值设置为不同的值。

> 注意
>
> Swift的两阶段初始化过程与Objective-C中的初始化类似。主要区别在于，在阶段1中，Objective-C为每个属性分配零或空值（如`0`或`nil`）。Swift的初始化流程更加灵活，因为它可以让你设置自定义的初始值，并且可以处理类型`0`或`nil`不是有效的默认值。

Swift的编译器执行四个有用的安全检查，以确保两阶段初始化完成没有错误：

安全检查1
指定的初始化程序必须确保在其委托给超类初始化程序之前，其类所引入的所有属性都已初始化。

如上所述，一旦所有存储的属性的初始状态已知，对象的内存仅被视为完全初始化。为了满足这个规则，指定的初始化器必须确保它的所有属性在它传递链之前被初始化。

安全检查2
指定的初始化程序必须在将值分配给继承的属性之前将其委托给超类初始化程序。如果不是，指定初始化程序分配的新值将被超类作为其自身初始化的一部分覆盖。

安全检查3
在为*任何*属性（包括由同一类定义的属性）分配值之前，便利初始值设定程序必须委托给另一个初始值设定项。如果不是，便利初始值设定项赋予的新值将被其自己的类的指定初始值设定项覆盖。

安全检查4
初始化程序不能调用任何实例方法，读取任何实例属性的值，或者`self`在完成初始化的第一阶段之后才将其引用为值。

在第一阶段结束之前，类实例不完全有效。一旦在第一阶段结束时已知该类实例有效，就只能访问属性，并且只能调用方法。

基于上面的四项安全检查，以下是两阶段初始化的演示过程：

阶段1

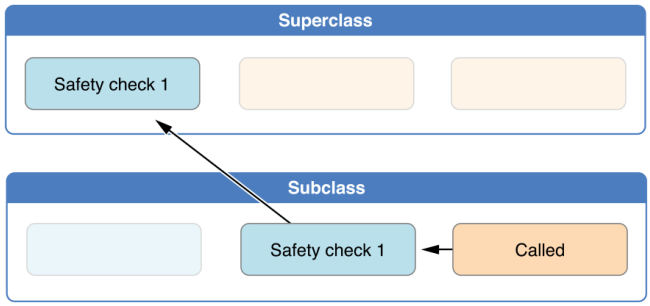
- 在一个类上调用指定的或便利的初始化程序。
- 分配了该类的新实例的内存。内存尚未初始化。
- 该类的指定初始化程序确认由该类引入的所有存储属性都具有值。这些存储的属性的内存现在已初始化。

- 指定的初始化程序将切换到超类初始化程序，以针对其自身存储的属性执行相同的任务。
- 这继续了类继承链，直到达到链的顶端。
- 一旦达到链的顶端，并且链中的最后一个类已确保其所有存储的属性都有值，则认为实例的内存已完全初始化，并」

阶段2

- 从链的顶部开始，链中的每个指定初始化程序都可以选择进一步定制实例。现在，初始化器能够访问`self`并修改其属性，调用其实例方法等等。
- 最后，链中的任何便利初始值设定项都可以选择自定义实例并使用它`self`。

以下是第1阶段如何查找假设的子类和超类的初始化调用：



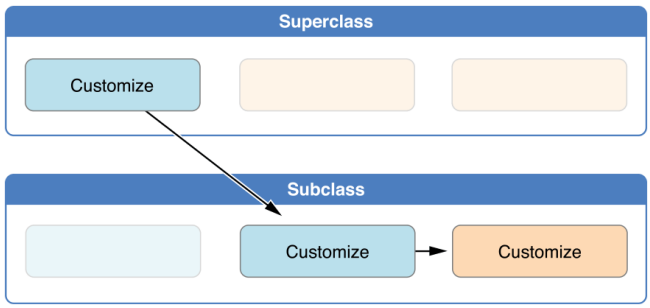
在这个例子中，初始化首先调用子类上的便利初始值设定项。此便利初始值设定程序不能修改任何属性。它将代表转移给同一个班级的指定初始化程序。

指定的初始化器确保所有子类的属性都具有一个值，如安全检查1所示。然后，它会在其超类上调用指定的初始化程序，以继续初始化链。

超类的指定初始化器确保所有的超类属性都有一个值。没有更多的超类初始化，因此不需要进一步的委派。

只要超类的所有属性都具有初始值，其内存就被认为已完全初始化，并且阶段1已完成。

以下是阶段2如何寻找相同的初始化调用：



超类的指定初始化器现在有机会进一步自定义实例（尽管它不必）。

一旦超类的指定初始化工具完成，子类的指定初始化工具就可以执行额外的自定义工作（尽管它不需要）。

最后，一旦子类的指定初始化程序完成，最初调用的便利初始化程序可以执行其他自定义。

初始化器继承和重写

与Objective-C中的子类不同，Swift子类默认不会继承它们的超类初始化器。Swift的方法避免了一个情况，即超类中的简单初始化器被更专用的子类继承，并用于创建未完全或正确初始化的子类的新实例。

> 注意
>
> 超类初始化*器*在某些情况下*是*继承的，但只有在安全和适当的情况下才能这样做。有关更多信息，请参阅下面的自动初始化程序继承。

如果你想让自定义的子类提供一个或多个相同的初始化器作为它的超类，你可以在子类中提供这些初始化器的自定义实现。

当你编写一个与超类*指定的*初始值设定项相匹配的子类初始值设定项时，你正在有效地提供对那个指定的初始值设定项的重写
的默认初始值设定项，情况也是如此，如默认初始值设定项中所述。

与重写的属性，方法或下标一样，`override`修饰符的存在会提示Swift检查超类是否有一个匹配的指定初始化程序被覆盖，并验证您的覆盖初始化程序的参数是否已按预期指定。

> 注意
>
> `override`即使您的子类的初始值设定项的实现是便捷初始值设定项，您仍然在覆盖超类指定的初始值设定项时编写修饰符。

相反，如果您编写与超类*方便*初始值设定项相匹配的子类初始值设定项，那么超类方便初始值设定项永远不会由您的子类直接调用，如上面在“类类型的初始化程序委派”中所述的规则。因此，你的子类不是（严格地说）提供超类初始化器的重写。因此，`override`在提供超类方便初始值设定项的匹配实现时，不要编写修饰符。

下面的例子定义了一个名为的基类`Vehicle`。这个基类声明了一个名为的存储属性`numberOfWheels`，默认`Int`值为`0`。这个`numberOfWheels`属性被一个被调用的属性`description`用来创建一个`String`车辆特征的描述：

```
class Vehicle {
    var numberOfWheels = 0
    var description: String {
        return "\(numberOfWheels) wheel(s)"
    }
}
```

本`Vehicle`类提供了其唯一的存储属性的默认值，并没有提供任何自定义初始化本身。其结果是，它会自动接收默认初始值，如在描述的默认初始值设定。默认初始化（如果有的话）始终是一类指定初始化，并且可以用来创建一个新的`Vehicle`带有实例`numberOfWheels`的`0`：

```
let vehicle = Vehicle()
print("Vehicle: \(vehicle.description)")
// Vehicle: 0 wheel(s)
```

下一个示例定义了一个`Vehicle`被调用的子类`Bicycle`：

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

该`Bicycle`子类定义了一个自定义的指定初始化程序，`init()`。这个指定的初始值设定项与来自超类的指定初始值设定项相匹配`Bicycle`，所以`Bicycle`此初始值设定项的版本标有`override`修饰符。

该`init()`用于初始化`Bicycle`通过调用开始`super.init()`，这就要求默认初始化`Bicycle`类的超类`Vehicle`。这可以确保`numberOfWheels`继承的属性`Vehicle`在`Bicycle`有机会修改属性之前被初始化。调用后`super.init()`，原始值将`numberOfWheels`被替换为新的值`2`。

如果您创建了一个实例`Bicycle`，您可以调用其继承的`description`计算属性来查看它的`numberOfWheels`属性如何更新：

```
let bicycle = Bicycle()
print("Bicycle: \(bicycle.description)")
// Bicycle: 2 wheel(s)
```

> 注意
>
> 子类可以在初始化期间修改继承的变量属性，但不能修改继承的常量属性。

自动初始化器继承

如上所述，默认情况下，子类不会继承其超类初始化程序。但是，如果满足某些条件，超类初始化器会自动继承。实际上，这
努力继承超类初始化程序。

假设您为在子类中引入的任何新属性提供默认值，则适用以下两个规则：

规则1
如果你的子类没有定义任何指定的初始化器，它会自动继承它所有的超类指定的初始化器。

规则2
如果你的子类提供了它*所有*超类指定的初始值设定项的实现 - 或者按照规则1继承它们，或者通过提供一个自定义实现作为它的定义的一部分 - 那么它将自动继承所有的超类方便初始值设定项。

即使您的子类添加了更多便利初始值设定项，这些规则也适用。

> 注意
> 作为满足规则2的一部分，子类可以实现超类指定初始化器作为子类便利初始化器。

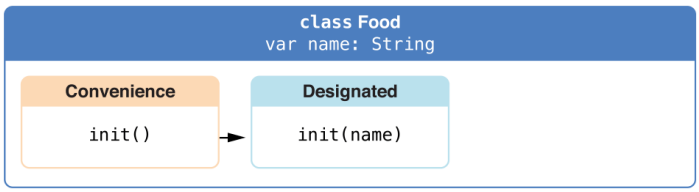
指定和便利初始化程序在行动

以下示例显示了指定的初始值设定项，便捷初始值设定项和实际上的自动初始化项继承。这个例子定义了三类所谓的层次`Food`，`RecipeIngredient`和`ShoppingListItem`，并演示了如何自己初始化互动。

调用层次结构中的基类`Food`，这是一个封装食品名称的简单类。该`Food`课程介绍一个`String`叫做物业`name`并提供两个初始化创建`Food`实例：

```
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}
```

下图显示了`Food`该类的初始化链：



类没有默认的成员初始值设定项，所以`Food`该类提供了一个指定的初始值设定项，它只接受一个名为的参数`name`。此初始值设定项可用于创建`Food`具有特定名称的新实例：

```
let namedMeat = Food(name: "Bacon")
// namedMeat's name is "Bacon"
```

`init(name: String)`来自`Food`该类 的初始化程序是作为*指定的*初始化程序提供的，因为它可确保新`Food`实例的所有存储属性都已完全初始化。本`Food`类没有超类，所以`init(name: String)`初始化并不需要调用`super.init()`来完成初始化。

该`Food`级还提供了一个*方便的*初始化，`init()`不带参数。该`init()`初始化通过委派跨越到一个新的食品提供了默认的占位符名称`Food`类的`init(name: String)`具有`name`的价值`[Unnamed]`：

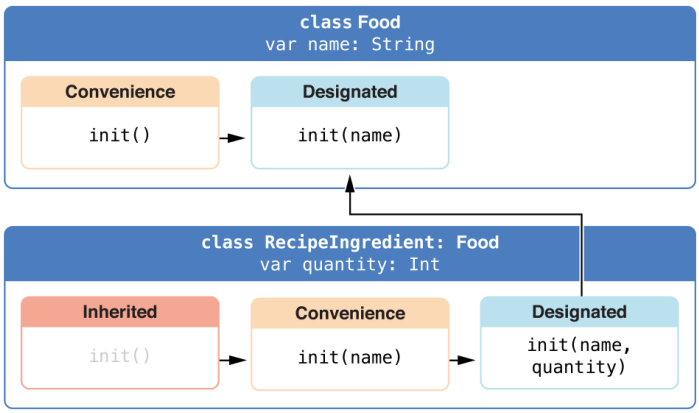
```
let mysteryMeat = Food()
```

```
// mysteryMeat's name is "[Unnamed]"
```

层次结构中的第二个类是被Food调用的子类RecipeIngredient。本RecipeIngredient类机型中的烹饪配方的成分。它引入了一
了两个初始化器

```
class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
```

下图显示了RecipeIngredient该类的初始化链：



该RecipeIngredient班有一个单一的指定初始化，init(name: String, quantity: Int)，它可以用来填充的所有新的属性的RecipeIngredient实例。该初始化器首先将传递的quantity参数分配给quantity属性，该属性是唯一引入的新属性RecipeIngredient。这样做后，初始化程序将委派给该类的init(name: String)初始化程序Food。该过程满足上述两阶段初始化中的安全检查1 。

RecipeIngredient还定义了一个便利初始值设定项，init(name: String)它用于RecipeIngredient仅通过名称创建一个实例。这个便捷初始值设定项假定数量1为RecipeIngredient没有明确数量的情况下创建的任何实例。此便捷初始值设定程序的定义使RecipeIngredient创建实例更快更方便，并且在创建多个单个数量的RecipeIngredient实例时避免代码重复。这个方便初始化器简单地委托给类的指定初始化器，传入一个quantity值1。

init(name: String)由提供 的便捷初始值RecipeIngredient设定项与来自init(name: String) *指定的*初始值设定项的参数相同Food。由于此便利初始值设定项将从其超类中覆盖指定的初始值设定项，因此必须使用override修饰符进行标记（如初始化项继承和覆盖中所述）。

尽管RecipeIngredient提供了init(name: String)初始化器作为便利初始化器，RecipeIngredient但它仍然提供了其所有超类的指定初始化器的实现。因此，RecipeIngredient它也会自动继承其所有超类的便利初始值设定项。

在这个例子中，RecipeIngredientis是超类Food，它有一个简单的初始化器调用init()。这个初始化器因此被继承RecipeIngredient。init()函数的继承版本与版本完全相同Food，只是它委托给RecipeIngredient版本init(name: String)而不是Food版本。

所有这三种初始化器都可以用来创建新的RecipeIngredient实例：

```
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
```

层次结构中的第三个也是最后一个类是RecipeIngredient被调用的子类ShoppingListItem。该ShoppingListItem级车型，因为它出现在购物清单配方成分。

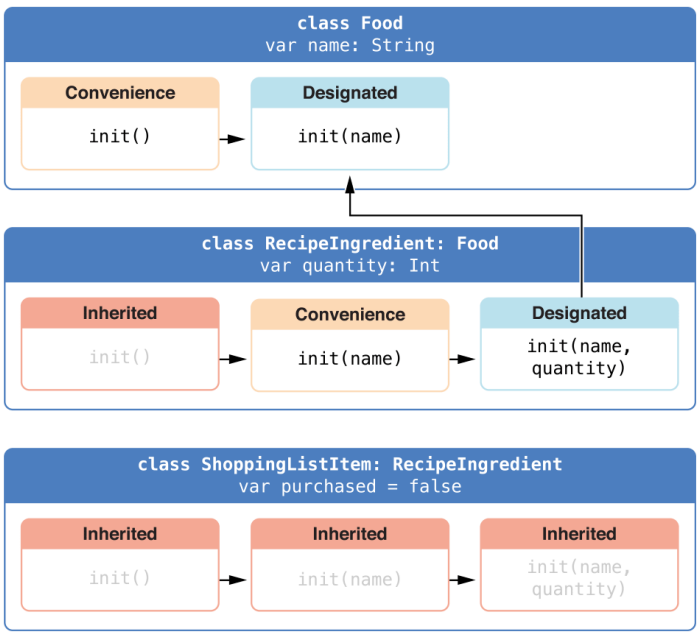
购物清单中的每个项目都以“未购买”开始。为了表示这个事实，ShoppingListItem引入一个名为的布尔属性purchased，默认值为false。ShoppingListItem还添加了一个计算description属性，它提供了一个ShoppingListItem实例的文本描述：

```
class S
    var purchased = false
    var description: String {
        var output = "\(quantity) x \(name)"
        output += purchased ? " ✔" : " ✘"
        return output
    }
}
```

> 注意
>
> ShoppingListItem没有定义一个初始值设定purchased项来提供一个初始值，因为购物清单中的项目（如这里建模的）始终始终未被购买。

因为它为所引入的所有属性提供了默认值，并且没有定义任何初始化器本身，所以会ShoppingListItem自动继承其超类中的*所有*指定和便捷初始化器。

下图显示了所有三个类的总体初始化链：



您可以使用全部三种继承的初始化器来创建一个新的ShoppingListItem实例：

```
var breakfastList = [
    ShoppingListItem(),
    ShoppingListItem(name: "Bacon"),
    ShoppingListItem(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
    print(item.description)
}
// 1 x Orange juice ✔
// 1 x Bacon ✘
// 6 x Eggs ✘
```

在这里，一个新的数组被调用breakfastList是由一个包含三个新ShoppingListItem实例的数组文字创建的。数组的类型被推断为[ShoppingListItem]。数组创建完成后，数组ShoppingListItem起始处的名称将从中更改为"[Unnamed]"，"Orange juice"并标记为已购买。打印阵列中每个项目的说明显示其默认状态已按预期设置。

Failable初始化器

定义初始化可能失败的类，结构或枚举有时很有用。此故障可能由无效的初始化参数值，缺少所需的外部资源或阻止初始化成功的其他条件触发。

为了处理可能失败的初始化条件，可以将一个或多个failable初始化器定义为类，结构或枚举定义的一部分。您通过在init关键字（init?）后面放置一个问号来编写一个failable初始化程序。

> 注意
>
> 您不能使用相同的参数类型和名称定义failable和nonfailable初始值设定项。

failable初始化器为它初始化的类型创建一个*可选*值。您return nil在可分区的初始化程序中写入以指示可以触发初始化失败的点。

> 注意
>
> 严格地说，初始化器不会返回一个值。相反，它们的作用是确保self在初始化结束时完全正确地初始化。虽然您编写return nil触发初始化失败，但您不使用return关键字来指示初始化成功。

例如，failable初始化器是为数字类型转换实现的。为了确保数字类型之间的转换能够精确保持该值，请使用init(exactly:)初始值设定项。如果类型转换无法保持该值，则初始化器失败。

```
let wholeNumber: Double = 12345.0
let pi = 3.14159

if let valueMaintained = Int(exactly: wholeNumber) {
    print("\(wholeNumber) conversion to Int maintains value of \(valueMaintained)")
}
// Prints "12345.0 conversion to Int maintains value of 12345"

let valueChanged = Int(exactly: pi)
// valueChanged is of type Int?, not Int

if valueChanged == nil {
    print("\(pi) conversion to Int does not maintain value")
}
// Prints "3.14159 conversion to Int does not maintain value"
```

下面的例子定义了一个称为的结构Animal，其中调用了一个常量String属性species。该Animal结构还定义了一个带有单个参数的可分区初始化程序species。此初始species值设定项会检查传递给初始值设定项的值是否为空字符串。如果找到空字符串，则会触发初始化失败。否则，该species属性的值将被设置，并且初始化成功：

```
struct Animal {
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}
```

你可以使用这个failable初始化器尝试初始化一个新的Animal实例并检查初始化是否成功：

```
let someCreature = Animal(species: "Giraffe")
```

```
// someCreature is of type Animal?, not Animal

if let giraffe = someCreature {
    pri
}
// Prints "An animal was initialized with a species of Giraffe"
```

如果将空字符串值传递给可分区的初始化程序的species参数，则初始化程序会触发初始化失败：

```
let anonymousCreature = Animal(species: "")
// anonymousCreature is of type Animal?, not Animal

if anonymousCreature == nil {
    print("The anonymous creature could not be initialized")
}
// Prints "The anonymous creature could not be initialized"
```

> 注意
>
> 检查空字符串值（例如""而不是"Giraffe"）与检查nil以指示缺少*可选*String值不同。在上面的例子中，一个空字符串（""）是一个有效的非自选的String。然而，动物将空串作为其species财产价值是不合适的。为了对这个限制进行建模，如果找到空字符串，则failable初始化程序会触发初始化失败。

枚举的Failable初始化器

您可以使用failable初始化程序根据一个或多个参数选择适当的枚举个案。如果提供的参数与适当的枚举大小写不匹配，则初始化程序可能会失败。

下面的例子定义称为枚举TemperatureUnit，具有三种可能的状态（kelvin，celsius，和fahrenheit）。failable初始化器用于为Character代表温度符号的值找到适当的枚举情况：

```
enum TemperatureUnit {
    case kelvin, celsius, fahrenheit
    init?(symbol: Character) {
        switch symbol {
        case "K":
            self = .kelvin
        case "C":
            self = .celsius
        case "F":
            self = .fahrenheit
        default:
            return nil
        }
    }
}
```

您可以使用这个failable初始化程序为三种可能的状态选择适当的枚举情况，并在参数不符合以下任何一种状态时导致初始化失败：

```
let fahrenheitUnit = TemperatureUnit(symbol: "F")
if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so initialization succeeded.")
}
// Prints "This is a defined temperature unit, so initialization succeeded."

let unknownUnit = TemperatureUnit(symbol: "X")
if unknownUnit == nil {
    print("This is not a defined temperature unit, so initialization failed.")
}
// Prints "This is not a defined temperature unit, so initialization failed."
```

具有原始值的枚举的Failable初始化器

具有原始值的枚举自动接收可失败的初始化程序，`init?(rawValue:)`该程序采用一个称为`rawValue`相应原始值类型的参数，并

您可以重写`TemperatureUnit`上面的示例以使用类型的原始值`Character`并利用`init?(rawValue:)`初始化程序：

```
enum TemperatureUnit: Character {
    case kelvin = "K", celsius = "C", fahrenheit = "F"
}

let fahrenheitUnit = TemperatureUnit(rawValue: "F")
if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so initialization succeeded.")
}
// Prints "This is a defined temperature unit, so initialization succeeded."

let unknownUnit = TemperatureUnit(rawValue: "X")
if unknownUnit == nil {
    print("This is not a defined temperature unit, so initialization failed.")
}
// Prints "This is not a defined temperature unit, so initialization failed."
```

初始化失败的传播

类，结构或枚举的failable初始值设定项可以从相同的类，结构或枚举中委托给另一个failable初始值设定项。同样，一个子类failable初始化器可以委托给一个超类failable初始化器。

在任何一种情况下，如果委派给其他初始化程序导致初始化失败，则整个初始化过程将立即失败，并且不会执行进一步的初始化代码。

> 注意
>
> failable初始化程序也可以委托给一个不可破解的初始化程序。如果您需要将潜在的故障状态添加到现有的初始化过程，否则不会失败，请使用此方法。

下面的例子定义了一个`Product`被调用的子类`CartItem`。该`CartItem`课程在线购物车中模拟一个项目。`CartItem`引入一个存储的常量属性，`quantity`并确保该属性的值至少为`1`：

```
class Product {
    let name: String
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}

class CartItem: Product {
    let quantity: Int
    init?(name: String, quantity: Int) {
        if quantity < 1 { return nil }
        self.quantity = quantity
        super.init(name: name)
    }
}
```

`CartItem`通过验证它已收到一个或更多的`quantity`值 来启动failable初始化程序`1`。如果`quantity`无效，则整个初始化过程立即失败，并且不会执行进一步的初始化代码。同样，用于`Product`检查`name`值的failable初始化程序，如果`name`是空字符串，则初始化程序立即失败。

如果您`CartItem`使用非空名称和数量`1`或更多的数量创建实例，则初始化会成功：

```
if let twoSocks = CartItem(name: "sock", quantity: 2) {
    print("Item: \(twoSocks.name), quantity: \(twoSocks.quantity)")
}
// Prir
```

如果您尝试创建值为的CartItem实例，则初始化程序会导致初始化失败： quantity0CartItem

```
if let zeroShirts = CartItem(name: "shirt", quantity: 0) {
    print("Item: \(zeroShirts.name), quantity: \(zeroShirts.quantity)")
} else {
    print("Unable to initialize zero shirts")
}
// Prints "Unable to initialize zero shirts"
```

同样，如果您尝试CartItem使用空name值创建实例，则超类Product初始化程序会导致初始化失败：

```
if let oneUnnamed = CartItem(name: "", quantity: 1) {
    print("Item: \(oneUnnamed.name), quantity: \(oneUnnamed.quantity)")
} else {
    print("Unable to initialize one unnamed product")
}
// Prints "Unable to initialize one unnamed product"
```

覆盖失败的初始化程序

就像其他任何初始化器一样，您可以在子类中覆盖超类failable初始化器。或者，您可以使用子类*非易失性*初始值设定项来覆盖超类failable初始化程序。这使您可以定义初始化无法失败的子类，即使允许超类的初始化失败。

请注意，如果使用非破坏子类初始值设定项覆盖failable超类初始值设定项，那么委托给超类初始值设定项的唯一方法是强制 - 解包failable超类初始值设定项的结果。

> 注意
> 你可以用一个不可破解的初始化器重写一个可以破坏的初始化器，但不能用其他方式。

下面的例子定义了一个叫做的类Document。这个类建模一个文档，该文档可以name使用一个非空字符串值的属性进行初始化nil，但不能为空字符串：

```
class Document {
    var name: String?
    // this initializer creates a document with a nil name value
    init() {}
    // this initializer creates a document with a nonempty name value
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}
```

下一个例子定义了一个Document被调用的子类AutomaticallyNamedDocument。这个AutomaticallyNamedDocument子类覆盖了两个由它引入的指定初始值设定项Document。如果实例初始化时没有名称，或者如果将空字符串传递给AutomaticallyNamedDocument初始name值设定项，这些重写确保实例具有初始值： "[Untitled]"init(name:)

```
class AutomaticallyNamedDocument: Document {
    override init() {
        super.init()
        self.name = "[Untitled]"
    }
    override init(name: String) {
```

```
        super.init()
        if name.isEmpty {
            self.name = "[Untitled]"

            self.name = name
        }
    }
}
```

将AutomaticallyNamedDocument覆盖其超类的failable init?(name:)与nonfailable初始化init(name:)初始化。由于AutomaticallyNamedDocument用空字符串大小写处理的方式不同于其超类，因此它的初始化程序不需要失败，因此它提供了初始化程序的非破坏版本。

您可以在初始化程序中使用强制解包来从超类调用failable初始化程序，作为实现子类的非破坏初始化程序的一部分。例如，UntitledDocument下面的子类总是被命名的"[Untitled]"，并且它init(name:)在初始化期间使用它超类中的failable 初始化器。

```
class UntitledDocument: Document {
    override init() {
        super.init(name: "[Untitled]")!
    }
}
```

在这种情况下，如果init(name:)超类的初始化程序曾以空字符串作为名称被调用，则强制解包操作将导致运行时错误。但是，因为它是用字符串常量调用的，所以可以看到初始化程序不会失败，因此在这种情况下不会发生运行时错误。

init！Failable初始化程序

您通常会定义一个failable初始化程序，通过在init关键字（init?）后面放置一个问号来创建适当类型的可选实例。或者，您可以定义一个可分解的初始化程序，该初始化程序可创建适当类型的隐式解包的可选实例。通过在init关键字（init!）后面放置感叹号而不是问号来做到这一点。

您可以从中init?进行委托，init!反之亦然，您可以覆盖init?，init!反之亦然。你也可以从initto 委托init!，虽然这样做会触发一个断言，如果init!初始化器导致初始化失败。

必需的初始化器

required在定义类初始化程序之前 写入修饰符，以指示类的每个子类都必须实现该初始化程序：

```
class SomeClass {
    required init() {
        // initializer implementation goes here
    }
}
```

您还必须required在所需初始化程序的每个子类实现之前编写修饰符，以指示初始化程序的要求适用于链中的其他子类。override覆盖所需的指定初始化程序时，不要编写修饰符：

```
class SomeSubclass: SomeClass {
    required init() {
        // subclass implementation of the required initializer goes here
    }
}
```

> 注意
> 如果您可以通过继承的初始化程序满足要求，则不必提供必需的初始化程序的显式实现。

使用闭包或函数设置默认属性值

如果存储属性的默认值需要一些定制或设置，则可以使用闭包或全局函数为该属性提供定制的默认值。只要属性所属类型的新

这些类型的闭包或函数通常会创建与该属性相同类型的临时值，定制该值来表示所需的初始状态，然后返回该临时值以用作属性的默认值。

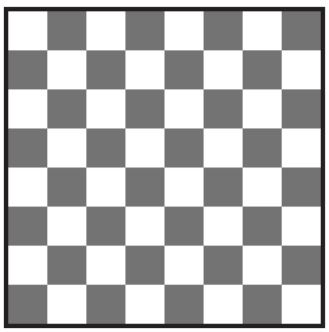
以下是关于如何使用闭包来提供默认属性值的框架大纲：

```
class SomeClass {
    let someProperty: SomeType = {
        // create a default value for someProperty inside this closure
        // someValue must be of the same type as SomeType
        return someValue
    }()
}
```

请注意，封闭的最终大括号后面是一对空括号。这告诉Swift立即执行关闭。如果省略这些圆括号，则试图将闭包本身分配给该属性，而不是闭包的返回值。

> 注意
>
> 如果您使用闭包来初始化属性，请记住，在执行闭包的位置，实例的其余部分尚未初始化。这意味着即使这些属性具有默认值，您也无法从闭包中访问任何其他属性值。您也不能使用隐式`self`属性，或者调用任何实例的方法。

下面的例子定义了一个叫做的结构`Chessboard`，它模拟棋盘的棋盘。国际象棋在8×8的棋盘上进行，黑白棋子交替出现。



为了表示这个游戏板，该`Chessboard`结构有一个叫做的属性`boardColors`，它是一个64 `Bool`值的数组。`true`数组中的值表示黑色方块，值`false`表示白色方块。数组中的第一项表示棋盘上的左上角，数组中的最后一项表示棋盘上的右下角。

该`boardColors`数组使用闭包进行初始化以设置其颜色值：

```
struct Chessboard {
    let boardColors: [Bool] = {
        var temporaryBoard = [Bool]()
        var isBlack = false
        for i in 1...8 {
            for j in 1...8 {
                temporaryBoard.append(isBlack)
                isBlack = !isBlack
            }
            isBlack = !isBlack
        }
        return temporaryBoard
    }()
    func squareIsBlackAt(row: Int, column: Int) -> Bool {
        return boardColors[(row * 8) + column]
```

```
    }
}
```

无论何时Chessb
数组中调用板上每个方块的适当颜色tmporaryBoard，并在其设置完成后返回该轴的数组的所有的返回值。
返回的数组值存储在实用程序函数中boardColors并可以查询squareIsBlackAt(row:column:)：

```
let board = Chessboard()
print(board.squareIsBlackAt(row: 0, column: 1))
// Prints "true"
print(board.squareIsBlackAt(row: 7, column: 7))
// Prints "false"
```