翻译：lifedim 校对：lifedim

构造过程（Initialization）

本页包含内容：

构造过程是为了使用某个类、结构体或枚举类型的实例而进行的准备过程。这个过程包含了为实例中的每个属性设置初始值和为其执行必要的准备和初始化任务。

构造过程是通过定义构造器（`Initializers`）来实现的，这些构造器可以看做是用来创建特定类型实例的特殊方法。与 Objective-C 中的构造器不同，Swift 的构造器无需返回值，它们的主要任务是保证新实例在第一次使用前完成正确的初始化。

类实例也可以通过定义析构器（`deinitializer`）在类实例释放之前执行特定的清除工作。想了解更多关于析构器的内容，请参考析构过程。

存储型属性的初始赋值

类和结构体在实例创建时，必须为所有存储型属性设置合适的初始值。存储型属性的值不能处于一个未知的状态。

你可以在构造器中为存储型属性赋初值，也可以在定义属性时为其设置默认值。以下章节将详细介绍这两种方法。

注意：

当你为存储型属性设置默认值或者在构造器中为其赋值时，它们的值是被直接设置的，不会触发任何属性观测器（property observers）。

构造器

构造器在创建某特定类型的新实例时调用。它的最简形式类似于一个不带任何参数的实例方法，以关键字init命名。

下面例子中定义了一个用来保存华氏温度的结构体Fahrenheit，它拥有一个Double类型的存储型属性temperature：

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}

var f = Fahrenheit()
println("The default temperature is \(f.temperature)° Fahrenheit")
// 输出 "The default temperature is 32.0° Fahrenheit”
```

这个结构体定义了一个不带参数的构造器init，并在里面将存储型属性temperature的值初始化为32.0（华摄氏度下水的冰点）。

默认属性值

如前所述，你可以在构造器中为存储型属性设置初始值；同样，你也可以在属性声明时为其设置默认值。

注意：

如果一个属性总是使用同一个初始值，可以为其设置一个默认值。无论定义默认值还是在构造器中赋值，最终它们实现的效果是一样的，只不过默认值跟属性构造过程结合的更紧密。使用默认值能让你的构造器更简洁、更清晰，且能通过默认值自动推导出属性的类型；同时，它也能让你充分

利用默认构造器、构造器继承（后续章节将讲到）等特性。
你可以使用更简单的方式在定义结构体`Fahrenheit`时为属性`temperature`设置默认值：

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

定制化构造过程

你可以通过输入参数和可选属性类型来定制构造过程，也可以在构造过程中修改常量属性。这些都将在后面章节中提到。

构造参数

你可以在定义构造器时提供构造参数，为其提供定制化构造所需值的类型和名字。构造器参数的功能和语法跟函数和方法参数相同。

下面例子中定义了一个包含摄氏度温度的结构体`Celsius`。它定义了两个不同的构造器：`init(fromFahrenheit:)`和`init(fromKelvin:)`，二者分别通过接受不同刻度表示的温度值来创建新的实例：

```
struct Celsius {
    var temperatureInCelsius: Double = 0.0
    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 是 100.0
```

```
let freezingPointOfWater = Celsius(fromKelvin:
273.15)
// freezingPointOfWater.temperatureInCelsius 是 0.0”
```

第一个构造器拥有一个构造参数，其外部名字为`fromFahrenheit`，内部名字为`fahrenheit`；第二个构造器也拥有一个构造参数，其外部名字为`fromKelvin`，内部名字为`kelvin`。这两个构造器都将唯一的参数值转换成摄氏温度值，并保存在属性`temperatureInCelsius`中。

内部和外部参数名

跟函数和方法参数相同，构造参数也存在一个在构造器内部使用的参数名字和一个在调用构造器时使用的外部参数名字。

然而，构造器并不像函数和方法那样在括号前有一个可辨别的名字。所以在调用构造器时，主要通过构造器中的参数名和类型来确定需要调用的构造器。正因为参数如此重要，如果你在定义构造器时没有提供参数的外部名字，Swift 会为每个构造器的参数自动生成一个跟内部名字相同的外部名，就相当于在每个构造参数之前加了一个哈希符号。

注意：

如果你不希望为构造器的某个参数提供外部名字，你可以使用下划线`_`来显示描述它的外部名，以此覆盖上面所说的默认行为。
以下例子中定义了一个结构体`Color`，它包含了三个常量：`red`、`green`和`blue`。这些属性可以存储0.0到1.0之间的值，用来指示颜色中红、绿、蓝成分的含量。

`Color`提供了一个构造器，其中包含三个`Double`类型的构造参数：

```
struct Color {
    let red = 0.0, green = 0.0, blue = 0.0
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
}
```

每当你创建一个新的`Color`实例，你都需要通过三种颜色的外部参数名来传值，并调用构造器。

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

注意，如果不通过外部参数名字传值，你是没法调用这个构造器的。只要构造器定义了某个外部参数名，你就必须使用它，忽略它将导致编译错误：

```
let veryGreen = Color(0.0, 1.0, 0.0)
// 报编译时错误，需要外部名称
```

可选属性类型

如果你定制的类型包含一个逻辑上允许取值为空的存储型属性--不管是因为它无法在初始化时赋值，还是因为它可以在之后某个时间点可以赋值为空--你都需要将它定义为可选类型`optional type`。可选类型的属性将自动初始化为空`nil`，表示这个属性是故意在初始化时设置为空的。

下面例子中定义了类`SurveyQuestion`，它包含一个可选字符串属性`response`：

```
class SurveyQuestion {
    var text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        println(text)
    }
}
let cheeseQuestion = SurveyQuestion(text: "Do you
like cheese?")
cheeseQuestion.ask()
// 输出 "Do you like cheese?"
cheeseQuestion.response = "Yes, I do like cheese.
```

调查问题在问题提出之后，我们才能得到回答。所以我们将属性回答`response`声明为`String?`类型，或者说是可选字符串类型`optional`

`String`。当`SurveyQuestion`实例化时，它将自动赋值为空`nil`，表明暂时还不存在此字符串。

构造过程中常量属性的修改

只要在构造过程结束前常量的值能确定，你可以在构造过程中的任意时间点修改常量属性的值。

注意：

对某个类实例来说，它的常量属性只能在定义它的类的构造过程中修改；不能在子类中修改。

你可以修改上面的`SurveyQuestion`示例，用常量属性替代变量属性`text`，指明问题内容`text`在其创建之后不会再被修改。尽管`text`属性现在是常量，我们仍然可以在其类的构造器中修改它的值：

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

默认构造器

Swift 将为所有属性已提供默认值的且自身没有定义任何构造器的结构体

或基类，提供一个默认的构造器。这个默认构造器将简单的创建一个所有属性值都设置为默认值的实例。

下面例子中创建了一个类`ShoppingListItem`，它封装了购物清单中的某一项的属性：名字（`name`）、数量（`quantity`）和购买状态`purchase state`。

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

由于`ShoppingListItem`类中的所有属性都有默认值，且它是没有父类的基类，它将自动获得一个可以为所有属性设置默认值的默认构造器（尽管代码中没有显式为`name`属性设置默认值，但由于`name`是可选字符串类型，它将默认设置为`nil`）。上面例子中使用默认构造器创造了一个`ShoppingListItem`类的实例（使用`ShoppingListItem()`形式的构造器语法），并将其赋值给变量`item`。

结构体的逐一成员构造器

除上面提到的默认构造器，如果结构体对所有存储型属性提供了默认值且自身没有提供定制的构造器，它们能自动获得一个逐一成员构造器。

逐一成员构造器是用来初始化结构体新实例里成员属性的快捷方法。我们在调用逐一成员构造器时，通过与成员属性名相同的参数名进行传值来完成对成员属性的初始赋值。

下面例子中定义了一个结构体`Size`，它包含两个属性`width`和`height`。Swift 可以根据这两个属性的初始赋值`0.0`自动推导出它们的类型`Double`。

由于这两个存储型属性都有默认值，结构体`Size`自动获得了一个逐一成员构造器 `init(width:height:)`。 你可以用它来为`Size`创建新的实例：

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

值类型的构造器代理

构造器可以通过调用其它构造器来完成实例的部分构造过程。这一过程称为构造器代理，它能减少多个构造器间的代码重复。

构造器代理的实现规则和形式在值类型和类类型中有所不同。值类型（结构体和枚举类型）不支持继承，所以构造器代理的过程相对简单，因为它们只能代理任务给本身提供的其它构造器。类则不同，它可以继承自其它类（请参考继承），这意味着类有责任保证其所有继承的存储型属性在构造时也能正确的初始化。这些责任将在后续章节类的继承和构造过程中介绍。

对于值类型，你可以使用`self.init`在自定义的构造器中引用其它的属于相同值类型的构造器。并且你只能在构造器内部调用`self.init`。

注意，如果你为某个值类型定义了一个定制的构造器，你将无法访问到默认构造器（如果是结构体，则无法访问逐一对象构造器）。这个限制可以防止你在为值类型定义了一个更复杂的，完成了重要准备构造器之后，别人还是错误的使用了那个自动生成的构造器。

注意：

假如你想通过默认构造器、逐一对象构造器以及你自己定制的构造器为值类型创建实例，我们建议你将自己定制的构造器写到扩展（`extension`）中，而不是跟值类型定义混在一起。想查看更多内容，请查看扩展章节。

下面例子将定义一个结构体`Rect`，用来展现几何矩形。这个例子需要两个辅助的结构体`Size`和`Point`，它们各自为其所有的属性提供了初始值`0.0`。

```
struct Size {
    var width = 0.0, height = 0.0
```

```
}
struct Point {
    var x = 0.0, y = 0.0
}
```

你可以通过以下三种方式为Rect创建实例--使用默认的0值来初始化origin和size属性；使用特定的origin和size实例来初始化；使用特定的center和size来初始化。在下面Rect结构体定义中，我们为着三种方式提供了三个自定义的构造器：

```
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构造器init()，在功能上跟没有自定义构造器时自动获得的默认构造器是一样的。这个构造器是一个空函数，使用一对大括号{}来描述，它没有执行任何定制的构造过程。调用这个构造器将返回一个Rect实例，它的origin和size属性都使用定义时的默认值Point(x: 0.0, y: 0.0)和Size(width: 0.0, height: 0.0)：

```
let basicRect = Rect()
// basicRect 的原点是 (0.0, 0.0)，尺寸是 (0.0, 0.0)
```

第二个Rect构造器init(origin:size:)，在功能上跟结构体在没有自定义构造器时获得的逐一成员构造器是一样的。这个构造器只是简单的将origin和size的参数值赋给对应的存储型属性：

```
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
```

```
    size: Size(width: 5.0, height: 5.0))
// originRect 的原点是 (2.0, 2.0)，尺寸是 (5.0, 5.0)
```

第三个`Rect`构造器`init(center:size:)`稍微复杂一点。它先通过`center`和`size`的值计算出`origin`的坐标。然后再调用（或代理给）`init(origin:size:)`构造器来将新的`origin`和`size`值赋值到对应的属性中：

let centerRect = Rect(center: Point(x: 4.0, y: 4.0), size: Size(width: 3.0, height: 3.0)) // centerRect 的原点是 (2.5, 2.5)，尺寸是 (3.0, 3.0)

构造器`init(center:size:)`可以自己将`origin`和`size`的新值赋值到对应的属性中。然而尽量利用现有的构造器和它所提供的功能来实现`init(center:size:)`的功能，是更方便、更清晰和更直观的方法。

注意：

如果你想用另外一种不需要自己定义`init()`和`init(origin:size:)`的方式来实现这个例子，请参考扩展。

类的继承和构造过程

类里面的所有存储型属性--包括所有继承自父类的属性--都必须在构造过程中设置初始值。

Swift 提供了两种类型的类构造器来确保所有类实例中存储型属性都能获得初始值，它们分别是指定构造器和便利构造器。

指定构造器和便利构造器

指定构造器是类中最主要的构造器。一个指定构造器将初始化类中提供的所有属性，并根据父类链往上调用父类的构造器来实现父类的初始化。

每一个类都必须拥有至少一个指定构造器。在某些情况下，许多类通过继承了父类中的指定构造器而满足了这个条件。具体内容请参考后续章节自动构造器的继承。

便利构造器是类中比较次要的、辅助型的构造器。你可以定义便利构造器来调用同一个类中的指定构造器，并为其参数提供默认值。你也可以定义便利构造器来创建一个特殊用途或特定输入的实例。

你应当只在必要的时候为类提供便利构造器，比方说某种情况下通过使用便利构造器来快捷调用某个指定构造器，能够节省更多开发时间并让类的构造过程更清、晰明。

构造器链

为了简化指定构造器和便利构造器之间的调用关系，Swift 采用以下三条规则来限制构造器之间的代理调用:

规则 1

指定构造器必须调用其直接父类的的指定构造器。

规则 2

便利构造器必须调用同一类中定义的其它构造器。

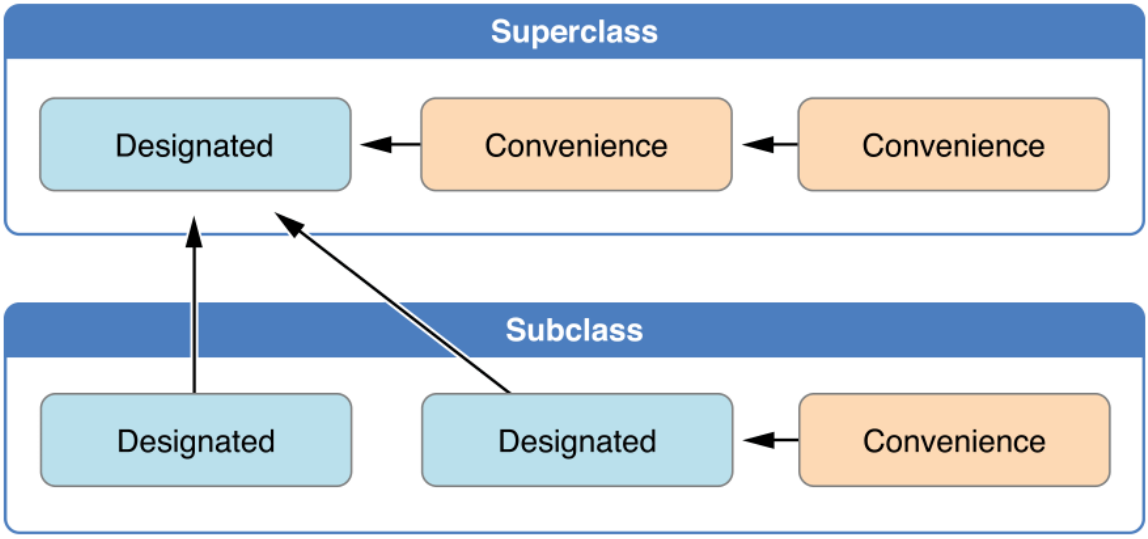
规则 3

便利构造器必须最终以调用一个指定构造器结束。

一个更方便记忆的方法是:

- 指定构造器必须总是向上代理
- 便利构造器必须总是横向代理

这些规则可以通过下面图例来说明:



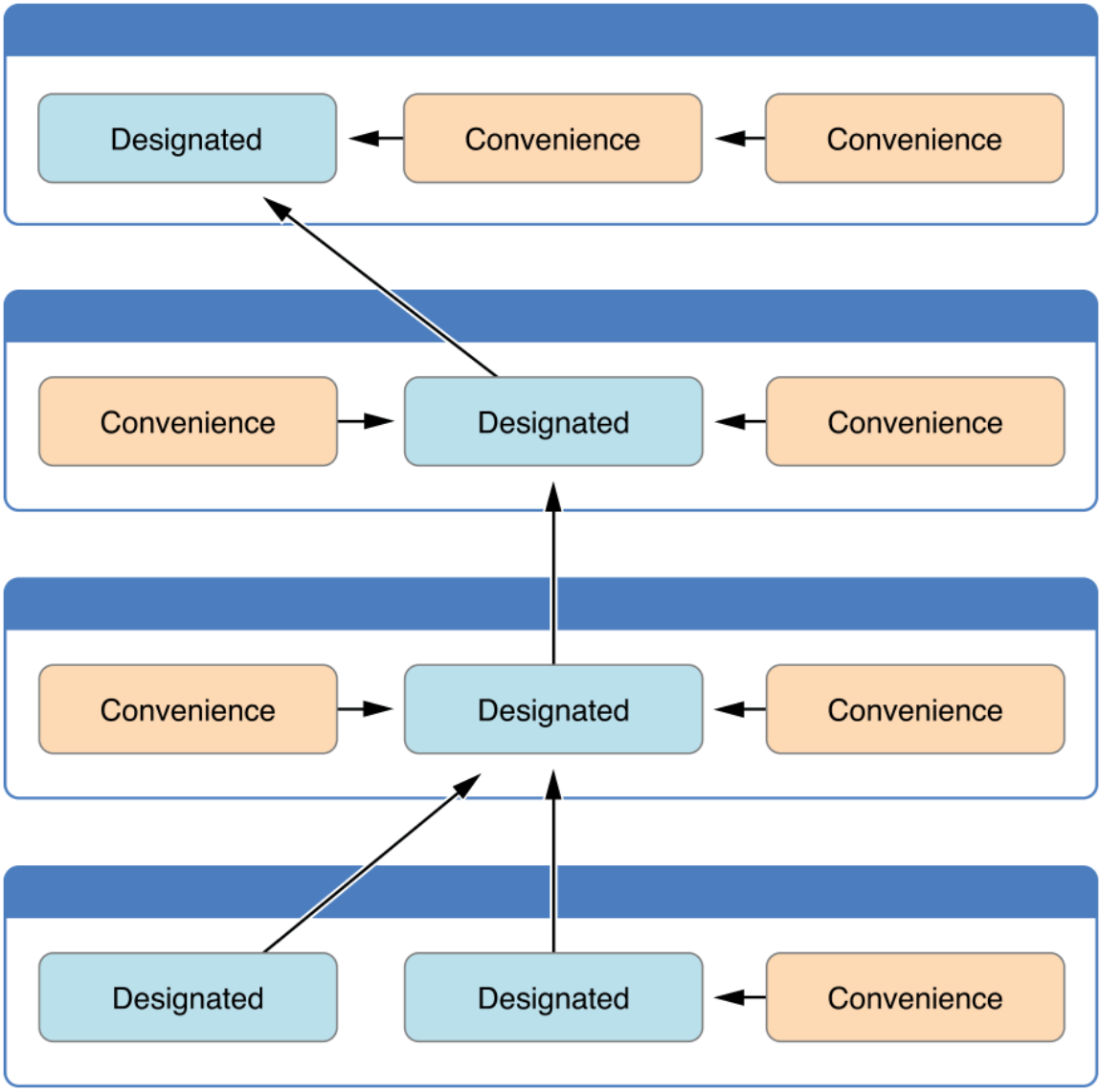
如图所示，父类中包含一个指定构造器和两个便利构造器。其中一个便利构造器调用了另外一个便利构造器，而后者又调用了唯一的指定构造器。这满足了上面提到的规则2和3。这个父类没有自己的父类，所以规则1没有用到。

子类中包含两个指定构造器和一个便利构造器。便利构造器必须调用两个指定构造器中的任意一个，因为它只能调用同一个类里的其他构造器。这满足了上面提到的规则2和3。而两个指定构造器必须调用父类中唯一的指定构造器，这满足了规则1。

注意：

这些规则不会影响使用时，如何用类去创建实例。任何上图中展示的构造器都可以用来完整创建对应类的实例。这些规则只在实现类的定义时有影响。

下面图例中展示了一种更复杂的类层级结构。它演示了指定构造器是如果在类层级中充当“管道”的作用，在类的构造器链上简化了类之间的内部关系。



两段式构造过程

Swift 中类的构造过程包含两个阶段。第一个阶段，每个存储型属性通过引入它们的类的构造器来设置初始值。当每一个存储型属性值被确定后，第二阶段开始，它给每个类一次机会在新实例准备使用之前进一步定制它们的存储型属性。

两段式构造过程的使用让构造过程更安全，同时在整个类层级结构中给予了每个类完全的灵活性。两段式构造过程可以防止属性值在初始化之前被访问；也可以防止属性被另外一个构造器意外地赋予不同的值。

注意：

Swift的两段式构造过程跟 Objective-C 中的构造过程类似。最主要的区别在于阶段 1，Objective-C 给每一个属性赋值`0`或空值（比如说`0`或`nil`）。Swift 的构造流程则更加灵活，它允许你设置定制的初始值，并自如应对某些属性不能以`0`或`nil`作为合法默认值的情况。

Swift 编译器将执行 4 种有效的安全检查，以确保两段式构造过程能顺利完成：

安全检查 1

指定构造器必须保证它所在类引入的所有属性都必须先初始化完成，之后才能将其它构造任务向上代理给父类中的构造器。

如上所述，一个对象的内存只有在其所有存储型属性确定之后才能完全初始化。为了满足这一规则，指定构造器必须保证它所在类引入的属性在它往上代理之前先完成初始化。

安全检查 2

指定构造器必须先向上代理调用父类构造器，然后再为继承的属性设置新值。如果没这么做，指定构造器赋予的新值将被父类中的构造器所覆盖。

安全检查 3

便利构造器必须先代理调用同一类中的其它构造器，然后再为任意属性赋新值。如果没这么做，便利构造器赋予的新值将被同一类中其它指定构造器所覆盖。

安全检查 4

构造器在第一阶段构造完成之前，不能调用任何实例方法、不能读取任何实例属性的值，也不能引用`self`的值。

以下是两段式构造过程中基于上述安全检查的构造流程展示：

阶段 1

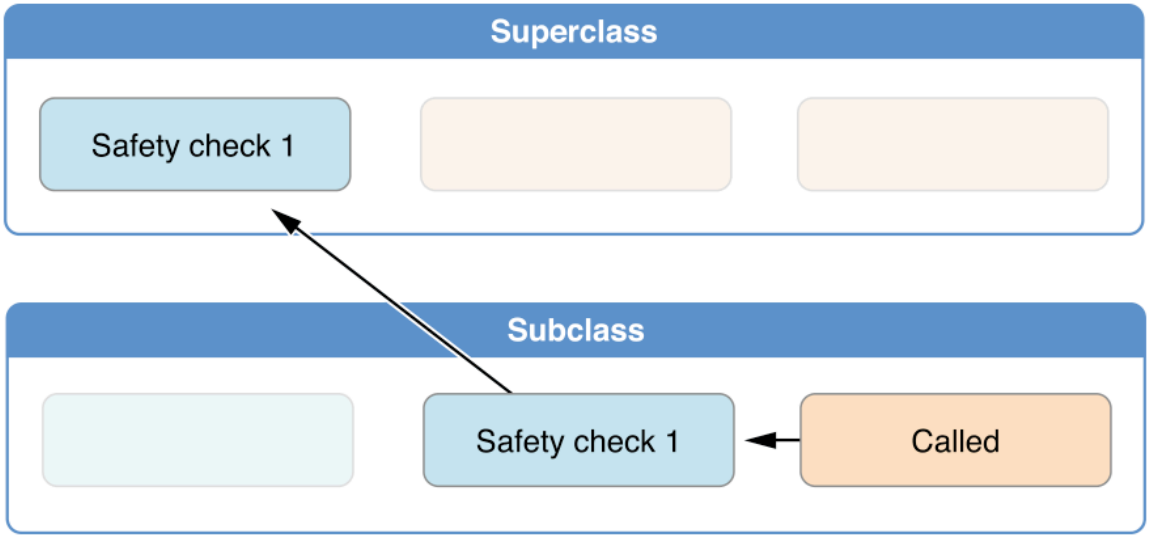
- 某个指定构造器或便利构造器被调用；

- 完成新实例内存的分配，但此时内存还没有被初始化；
- 指定构造器确保其所在类引入的所有存储型属性都已赋初值。存储型属性所属的内存完成初始化；
- 指定构造器将调用父类的构造器，完成父类属性的初始化；
- 这个调用父类构造器的过程沿着构造器链一直往上执行，直到到达构造器链的最顶部；
- 当到达了构造器链最顶部，且已确保所有实例包含的存储型属性都已经赋值，这个实例的内存被认为已经完全初始化。此时阶段1完成。

阶段 2

- 从顶部构造器链一直往下，每个构造器链中类的指定构造器都有机会进一步定制实例。构造器此时可以访问`self`、修改它的属性并调用实例方法等等。
- 最终，任意构造器链中的便利构造器可以有机会定制实例和使用`self`。

下图展示了在假定的子类和父类之间构造的阶段1：·



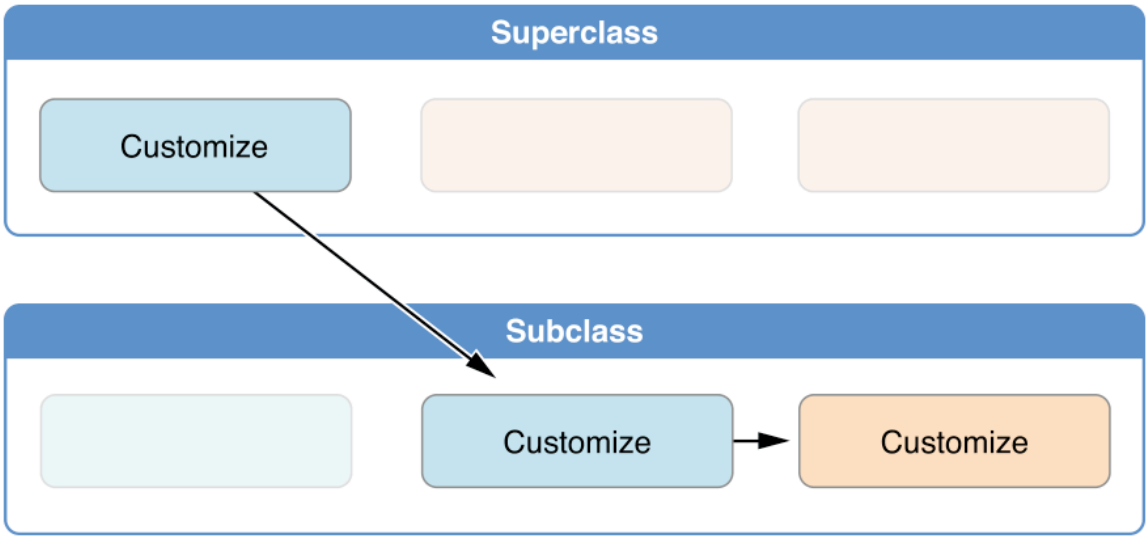
在这个例子中，构造过程从对子类中一个便利构造器的调用开始。这个便利构造器此时没法修改任何属性，它把构造任务代理给同一类中的指定构造器。

如安全检查1所示，指定构造器将确保所有子类的属性都有值。然后它将调用父类的指定构造器，并沿着造器链一直往上完成父类的构建过程。

父类中的指定构造器确保所有父类的属性都有值。由于没有更多的父类需要构建，也就无需继续向上做构建代理。

一旦父类中所有属性都有了初始值，实例的内存被认为是完全初始化，而阶段1也已完成。

以下展示了相同构造过程的阶段2：



父类中的指定构造器现在有机会进一步来定制实例（尽管它没有这种必要）。

一旦父类中的指定构造器完成调用，子类的构指定构造器可以执行更多的定制操作（同样，它也没有这种必要）。

最终，一旦子类的指定构造器完成调用，最开始被调用的便利构造器可以执行更多的定制操作。

构造器的继承和重载

跟 Objective-C 中的子类不同，Swift 中的子类不会默认继承父类的构造器。Swift 的这种机制可以防止一个父类的简单构造器被一个更专业的子类继承，并被错误的用来创建子类的实例。

假如你希望自定义的子类中能实现一个或多个跟父类相同的构造器--也许是为了完成一些定制的构造过程--你可以在你定制的子类中提供和重载与

父类相同的构造器。

如果你重载的构造器是一个指定构造器，你可以在子类里重载它的实现，并在自定义版本的构造器中调用父类版本的构造器。

如果你重载的构造器是一个便利构造器，你的重载过程必须通过调用同一类中提供的其它指定构造器来实现。这一规则的详细内容请参考构造器链。

注意：

与方法、属性和下标不同，在重载构造器时你没有必要使用关键字`override`。

自动构造器的继承

如上所述，子类不会默认继承父类的构造器。但是如果特定条件可以满足，父类构造器是可以被自动继承的。在实践中，这意味着对于许多常见场景你不必重载父类的构造器，并且在尽可能安全的情况下以最小的代价来继承父类的构造器。

假设要为子类中引入的任意新属性提供默认值，请遵守以下2个规则：

规则 1

如果子类没有定义任何指定构造器，它将自动继承所有父类的指定构造器。

规则 2

如果子类提供了所有父类指定构造器的实现--不管是通过规则1继承过来的，还是通过自定义实现的--它将自动继承所有父类的便利构造器。

即使你在子类中添加了更多的便利构造器，这两条规则仍然适用。

注意：

子类可以通过部分满足规则2的方式，使用子类便利构造器来实现父类的

指定构造器。

指定构造器和便利构造器的语法

类的指定构造器的写法跟值类型简单构造器一样：

```
init(parameters) {
    statements
}
```

便利构造器也采用相同样式的写法，但需要在`init`关键字之前放置`convenience`关键字，并使用空格将它们俩分开：

```
convenience init(parameters) {
    statements
}
```

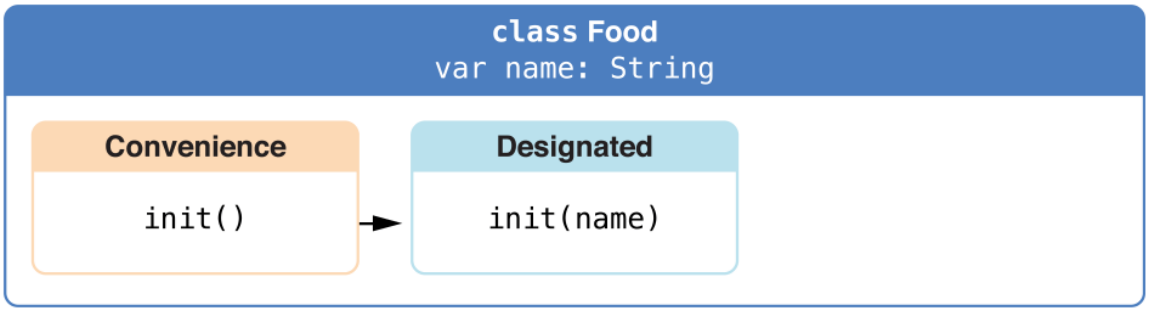
指定构造器和便利构造器实战

接下来的例子将在实战中展示指定构造器、便利构造器和自动构造器的继承。它定义了包含三个类`Food`、`RecipeIngredient`以及`ShoppingListItem`的类层次结构，并将演示它们的构造器是如何相互作用的。

类层次中的基类是`Food`，它是一个简单的用来封装食物名字的类。`Food`类引入了一个叫做`name`的`String`类型属性，并且提供了两个构造器来创建`Food`实例：

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

下图中展示了`Food`的构造器链：



类没有提供一个默认的逐一成员构造器，所以Food类提供了一个接受单一参数name的指定构造器。这个构造器可以使用一个特定的名字来创建新的Food实例：

```
let namedMeat = Food(name: "Bacon")
// namedMeat 的名字是 "Bacon”
```

Food类中的构造器init(name: String)被定义为一个指定构造器，因为它能确保所有新Food实例的中存储型属性都被初始化。Food类没有父类，所以init(name: String)构造器不需要调用super.init()来完成构造。

Food类同样提供了一个没有参数的便利构造器 init()。这个init()构造器为新食物提供了一个默认的占位名字，通过代理调用同一类中定义的指定构造器init(name: String)并给参数name传值[Unnamed]来实现：

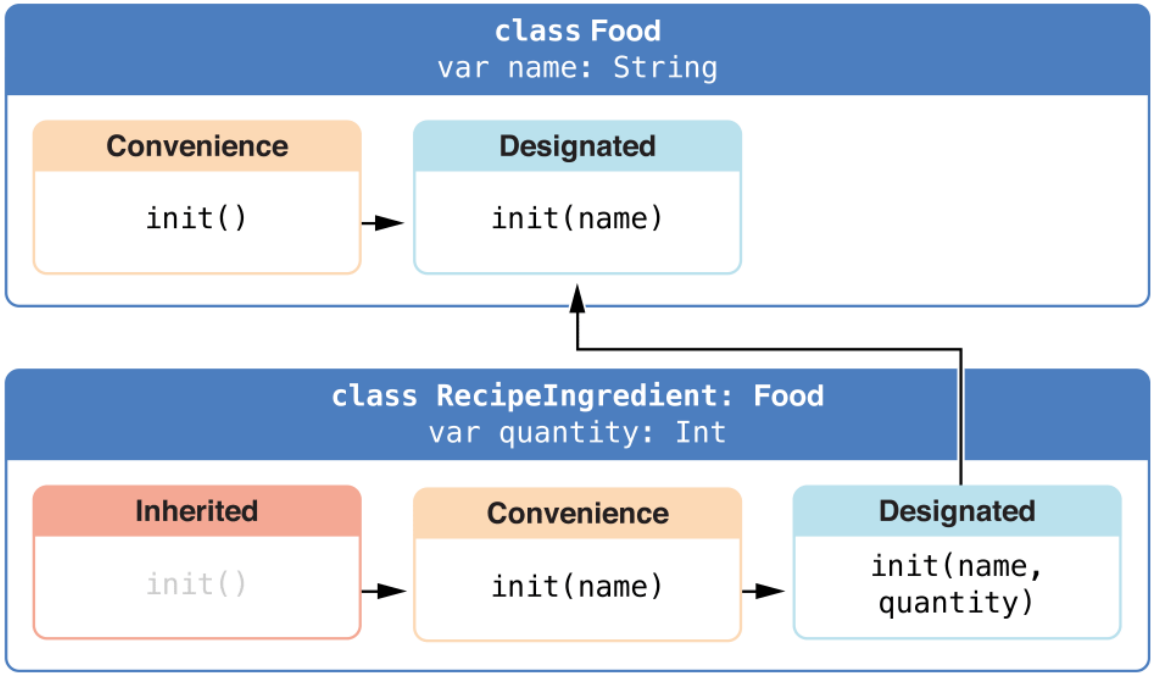
```
let mysteryMeat = Food()
// mysteryMeat 的名字是 [Unnamed]
```

类层级中的第二个类是Food的子类RecipeIngredient。RecipeIngredient类构建了食谱中的一味调味剂。它引入了Int类型的数量属性quantity（以及从Food继承过来的name属性），并且定义了两个构造器来创建RecipeIngredient实例：

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

```
    }
}
```

下图中展示了RecipeIngredient类的构造器链：



RecipeIngredient类拥有一个指定构造器init(name: String, quantity: Int)，它可以用来产生新RecipeIngredient实例的所有属性值。这个构造器一开始先将传入的quantity参数赋值给quantity属性，这个属性也是唯一在RecipeIngredient中新引入的属性。随后，构造器将任务向上代理给父类Food的init(name: String)。这个过程满足两段式构造过程中的安全检查1。

RecipeIngredient也定义了一个便利构造器init(name: String)，它只通过name来创建RecipeIngredient的实例。这个便利构造器假设任意RecipeIngredient实例的quantity为1，所以不需要显示指明数量即可创建出实例。这个便利构造器的定义可以让创建实例更加方便和快捷，并且避免了使用重复的代码来创建多个quantity为 1 的RecipeIngredient实例。这个便利构造器只是简单的将任务代理给了同一类里提供的指定构造器。

注意，RecipeIngredient的便利构造器init(name: String)使用了跟Food中指定构造器init(name: String)相同的参数。尽管RecipeIngredient这个构造器是便利构造器，RecipeIngredient依然提供了对所有父类指定构造器的实现。因此，RecipeIngredient也能自

动继承了所有父类的便利构造器。

在这个例子中，RecipeIngredient的父类是Food，它有一个便利构造器init()。这个构造器因此也被RecipeIngredient继承。这个继承的init()函数版本跟Food提供的版本是一样的，除了它是将任务代理给RecipeIngredient版本的init(name: String)而不是Food提供的版本。

所有的这三种构造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)
```

类层级中第三个也是最后一个类是RecipeIngredient的子类，叫做ShoppingListItem。这个类构建了购物单中出现的某一种调味料。

购物单中的每一项总是从unpurchased未购买状态开始的。为了展现这一事实，ShoppingListItem引入了一个布尔类型的属性purchased，它的默认值是false。ShoppingListItem还添加了一个计算型属性description，它提供了关于ShoppingListItem实例的一些文字描述：

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

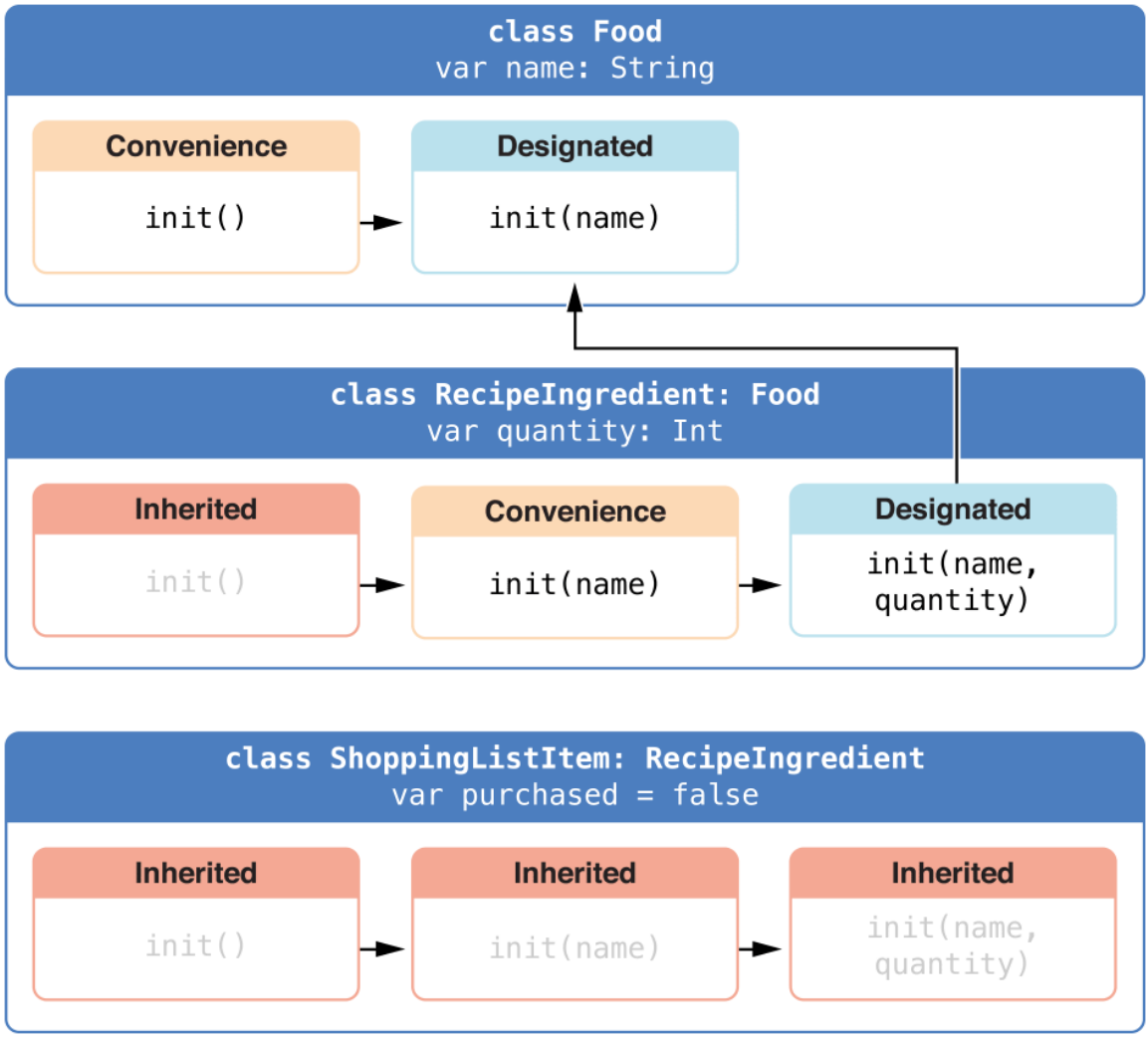
注意：

ShoppingListItem没有定义构造器来为purchased提供初始化值，这是因为任何添加到购物单的项的初始状态总是未购买。

由于它为自己引入的所有属性都提供了默认值，并且自己没有定义任何构造器，ShoppingListItem将自动继承所有父类中的指定构造器和便利构

造器。

下图种展示了所有三个类的构造器链：



你可以使用全部三个继承来的构造器来创建`ShoppingListItem`的新实例：

```
var breakfastList = [
    ShoppingListItem(),
    ShoppingListItem(name: "Bacon"),
    ShoppingListItem(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
    println(item.description)
```

```
}
// 1 x orange juice ✔
// 1 x bacon ✘
// 6 x eggs ✘
```

如上所述，例子中通过字面量方式创建了一个新数组`breakfastList`，它包含了三个新的`ShoppingListItem`实例，因此数组的类型也能自动推导为`ShoppingListItem[]`。在数组创建完之后，数组中第一个`ShoppingListItem`实例的名字从`[Unnamed]`修改为`Orange juice`，并标记为已购买。接下来通过遍历数组每个元素并打印它们的描述值，展示了所有项当前的默认状态都已按照预期完成了赋值。

通过闭包和函数来设置属性的默认值

如果某个存储型属性的默认值需要特别的定制或准备，你就可以使用闭包或全局函数来为其属性提供定制的默认值。每当某个属性所属的新类型实例创建时，对应的闭包或函数会被调用，而它们的返回值会当做默认值赋值给这个属性。

这种类型的闭包或函数一般会创建一个跟属性类型相同的临时变量，然后修改它的值以满足预期的初始状态，最后将这个临时变量的值作为属性的默认值进行返回。

下面列举了闭包如何提供默认值的代码概要：

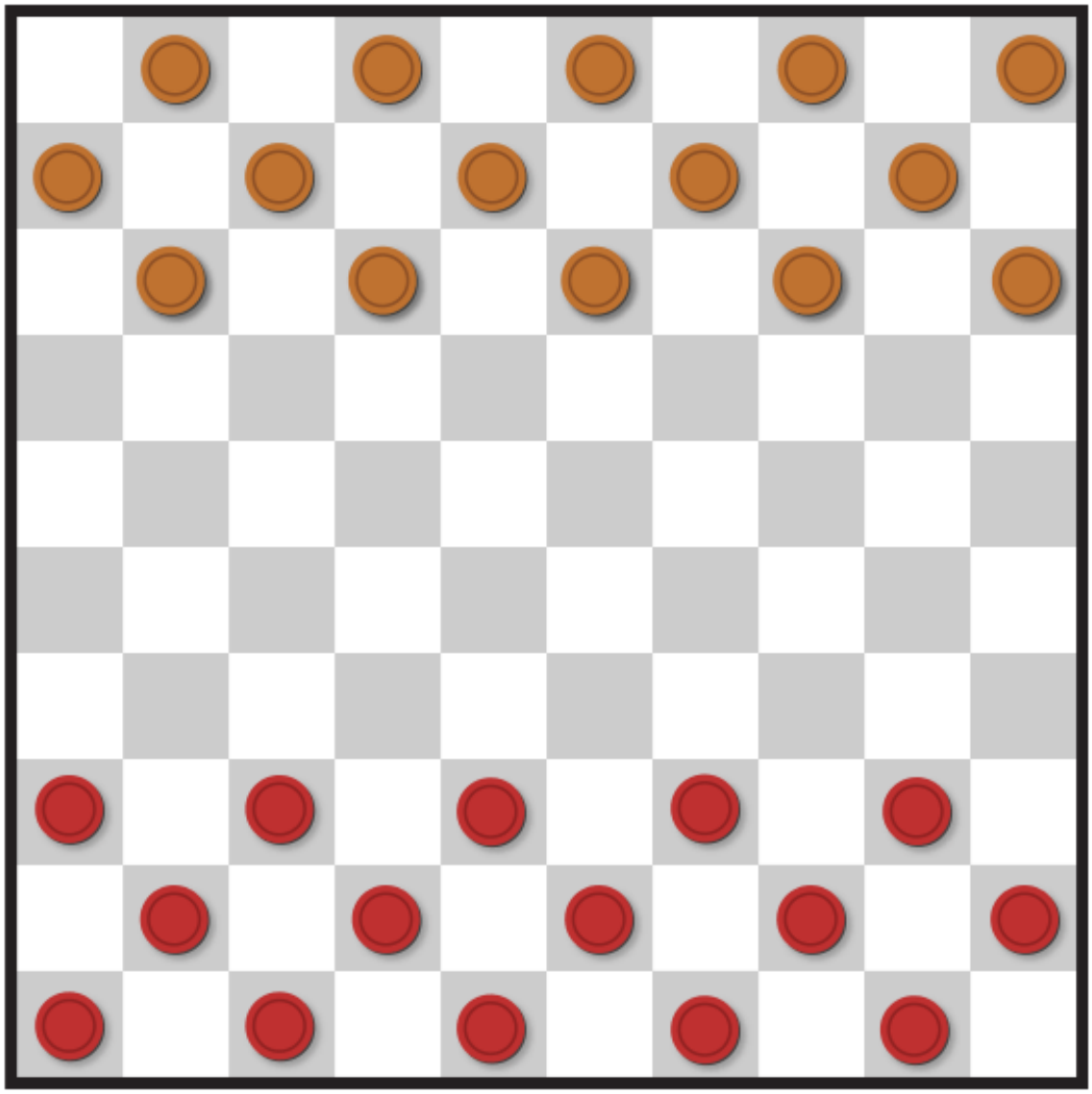
```
class SomeClass {
    let someProperty: SomeType = {
        // 在这个闭包中给 someProperty 创建一个默认值
        // someValue 必须和 SomeType 类型相同
        return someValue
        }()
}
```

注意闭包结尾的大括号后面接了一对空的小括号。这是用来告诉 Swift 需要立刻执行此闭包。如果你忽略了这对括号，相当于是将闭包本身作为值赋值给了属性，而不是将闭包的返回值赋值给属性。

注意：

如果你使用闭包来初始化属性的值，请记住在闭包执行时，实例的其它部分都还没有初始化。这意味着你不能够在闭包里访问其它的属性，就算这个属性有默认值也不允许。同样，你也不能使用隐式的`self`属性，或者调用其它的实例方法。

下面例子中定义了一个结构体`Checkerboard`，它构建了西洋跳棋游戏的棋盘：



西洋跳棋游戏在一副黑白格交替的 10x10 的棋盘中进行。为了呈现这副游戏棋盘，`Checkerboard`结构体定义了一个属性`boardColors`，它是一个包含 100 个布尔值的数组。数组中的某元素布尔值为`true`表示对应的是一个黑格，布尔值为`false`表示对应的是一个白格。数组中第一个元素代

表棋盘上左上角的格子，最后一个元素代表棋盘上右下角的格子。

boardColor数组是通过一个闭包来初始化和组装颜色值的：

```
struct Checkerboard {
    let boardColors: Bool[] = {
        var temporaryBoard = Bool[]()
        var isBlack = false
        for i in 1...10 {
            for j in 1...10 {
                temporaryBoard.append(isBlack)
                isBlack = !isBlack
            }
            isBlack = !isBlack
        }
        return temporaryBoard
        }()
    func squareIsBlackAtRow(row: Int, column: Int) ->
Bool {
        return boardColors[(row * 10) + column]
    }
}
```

每当一个新的Checkerboard实例创建时，对应的赋值闭包会执行，一系列颜色值会被计算出来作为默认值赋值给boardColors。上面例子中描述的闭包将计算出棋盘中每个格子合适的颜色，将这些颜色值保存到一个临时数组temporaryBoard中，并在构建完成时将此数组作为闭包返回值返回。这个返回的值将保存到boardColors中，并可以通squareIsBlackAtRow这个工具函数来查询。

```
let board = Checkerboard()
println(board.squareIsBlackAtRow(0, column: 1))
// 输出 "true"
println(board.squareIsBlackAtRow(9, column: 9))
// 输出 "false"
```