可选链

可选链是一个调用和查询可选属性、方法和下标的过程，它可能为 nil 。如果可选项包含值，属性、方法或者下标的调用成功；如果可选项是 nil，属性、方法或者下标的调用会返回 nil 。多个查询可以链接在一起，如果链中任何一个节点是 nil，那么整个链就会得体地失败。

> 注意
>
> Swift 中的可选链与 Objective-C 中的 nil 信息类似，但是它却工作在任意类型上，而且它能检测成功还是失败。

可选链代替强制展开

你可以通过在你希望如果可选项为非 nil 就调用属性、方法或者脚本的可选值后边使用问号（?）来明确可选链。这和在可选值后放叹号（!）来强制展开它的值非常类似。主要的区别在于可选链会在可选项为 nil 时得体地失败，而强制展开则在可选项为 nil 时触发运行时错误。

为了显示出可选链可以在 nil 值上调用，可选链调用的结果一定是一个可选值，就算你查询的属性、方法或者下标返回的是非可选值。你可以使用这个可选项返回值来检查可选链调用是成功（返回的可选项包含值），还是由于链中出现了 nil 而导致没有成功（返回的可选值是 nil ）。

另外，可选链调用的结果与期望的返回值类型相同，只是包装成了可选项。通常返回 Int 的属性通过可选链后会返回一个 Int? 。

接下来的一些代码片段演示了可选链与强制展开的不同并允许你检查是否成功。

首先，定义两个类，Person 和 Residence ：

```
1  class Person {
2      var residence: Residence?
3  }
4  class Residence {
5      var numberOfRooms = 1
6  }
7
```

Residence 实例有一个 Int 属性叫做 numberOfRooms ，它带有默认值 1 . Person 实例有一个 Residence? 类型的可选 residence 属性。
如果你创建一个新的 Person 实例，得益于可选项的特性，它的 residence 属性会默认初始化为 nil 。下面的代码中， john 拥有值为 nil 的 residence 属性：

```
1  let john = Person()
```

如果你尝试访问这个人的 residence 里的 numberOfRooms 属性，通过在 residence 后放一个叹号来强制展开它的值，你会触发一个运行时错误，因为 residence 根本没有值可以展开：

```
let roomCount = john.residence!.numberOfRooms
// this triggers a runtime error
```

上边的代码会在 john.residence 有一个非 nil 值时成功并且给 roomCount 赋值一个包含合适房间号的 Int 值。总之，这段代码一定会在 residence 为 nil 时触发运行时错误，如同上边展示的那样。
可选链提供另一种访问 numberOfRooms 的方法。要使用可选链，使用问号而不是叹号：

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

这将会告诉 Swift 把可选 residence 属性“链接”起来并且取回 numberOfRooms 的值，如果 residence 存在的话。

由于尝试访问 numberOfRooms 有失败的潜在可能，可选链尝试返回一个 Int? 类型的值，或者说“可选 Int ”。当 residence 为 nil，就如同上边的栗子，这个可选 Int 将也会是 nil，来反映出不能访问 numberOfRooms 这个事实。可选 Int 通过可选绑定来展开整数并赋值非可选值给 roomCount 变量。

注意就算 numberOfRooms 是非可选的 Int 也是适用的。事实上通过可选链查询就意味着对 numberOfRooms 的调用一定会返回 Int? 而不是 Int 。

你可以赋值一个 Residence 实例给 john.residence ，这样它就不会再有 nil 值：

```
john.residence = Residence()
```

john.residence 现在包含了实际的 Residence 实例，而不是 nil 。如果你尝试用与之前相同的可选链访问 numberOfRooms ，它就会返回一个 Int? 包含默认 numberOfRooms 值 1 ：

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "John's residence has 1 room(s)."
```

为可选链定义模型类

你可以使用可选链来调用属性、方法和下标不止一个层级。这允许你在相关类型的复杂模型中深入到子属性，并检查是否可以在这些自属性里访问属性、方法和下标。

下边的代码段定义了四个模型类用于随后的栗子，包括多层可选链的栗子。这些栗子通过添加 Room 和 Address 类扩展了上边的 Person 和 Residence 模型，以及相关的属性、方法和下标。

Person 类与之前的定义方式相同：

```
class Person {
    var residence: Residence?
}
```

Residence 类比以前要复杂一些。这次， Residence 类定义了一个叫做 rooms 的变量属性，它使用一个空的 [Room] 类型空数组初始化：

```
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        get {
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    func printNumberOfRooms() {
        print("The number of rooms is \(numberOfRooms)")
    }
    var address: Address?
}
```

由于这个版本的 Residence 储存了 Room 实例的数组，它的 numberOfRooms 属性使用计算属性来实现，而不是储存属性。计算属性 numberOfRooms 只是返回rooms数组的 count 属性值。
作为给它的 rooms 数组赋值的快捷方式，这个版本的 Residence 提供了一个可读写的下标来访问 rooms 数组的索引位置。

这个版本的 Residence 同样提供了一个叫做 printNumberOfRooms 的方法，它打印住所中的房间号。

最终，Residence 定义了一个可选属性叫做 address ，它是一个 Address? 类型，这个属性的 Address 类类型在下面定义。

rooms 数组使用的 Room 类型仅有一个属性叫做 name ，还有一个初始化器来给这个属性设置合适的房间名：

```
class Room {
    let name: String
    init(name: String) { self.name = name }
}
```

这个模型的最后一个类型叫做 Address 。这个类型有三个 String? 类型可选属性。前两个属性， buildingName 和 buildingNumber ，是定义地址中特定建筑部分的代替方式。第三个属性， street ，是给地址里街道命名的：

```
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if buildingName != nil {
            return buildingName
        } else if buildingNumber != nil && street != nil {
            return "\(buildingNumber) \(street)"
        } else {
            return nil
        }
    }
}
```

Address 类同样提供了一个方法叫做 buildingIdentifier() ，它有一个 String? 类型的返回值。这个方法检查地址的属性并返回 buildingName 如果它有值的话，或者把 buildingNumber 与 street 串联起来，如果它们都有值的话，或者就是 nil 。

通过可选链访问属性

如同可选链代替强制展开中展示的那样，你可以使用可选链来访问可选值里的属性，并且检查这个属性的访问是否成功。

使用上边定义的类来创建一个新得 Person 实例，并且尝试如之前一样访问它的 numberOfRooms 属性：

```
let john = Person()
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

由于 john.residence 是 nil ，这个可选链调用与之前一样失败了。
你同样可以尝试通过可选链来给属性赋值：

```
let someAddress = Address()
someAddress.buildingNumber = "29"
someAddress.street = "Acacia Road"
john.residence?.address = someAddress
```

在这个栗子中，给 john.residence 的 address 属性赋值会失败，因为 john.residence 目前是 nil 。
这个赋值是可选链的一部分，也就是说 = 运算符右手侧的代码都不会被评判。在先前的栗子中，不容易看出 someAddress 没有被评判，因为赋值一个常量不会有任何副作用。下边的栗子做同样的赋值，但它使用一个函数来创建地址。函数会在返回值之前打印“函数被调用了”，这可以让你看到 = 运算符右手侧是否被评判。

```
func createAddress() -> Address {
    print("Function was called.")

    let someAddress = Address()
    someAddress.buildingNumber = "29"
    someAddress.street = "Acacia Road"

    return someAddress
}
john.residence?.address = createAddress()
```

你可以看到 createAddress() 函数没有被调用，因为没有任何东西打印出来。

通过可选链调用方法

你可以使用可选链来调用可选项里的方法，并且检查调用是否成功。你甚至可以在没有定义返回值的方法上这么做。

Residence 类中的 printNumberOfRooms() 方法打印了当前 numberOfRooms 的值。方法看起来长这样：

```
func printNumberOfRooms() {
    print("The number of rooms is \(numberOfRooms)")
}
```

这个方法没有指定返回类型。总之，如没有返回值的函数中描述的那样，函数和方法没有返回类型就隐式地指明为 Void 类型。意思是说它们返回一个 () 的值或者是一个空的元组。

如果你用可选链在可选项里调用这个方法，方法的返回类型将会是 Void?，而不是 Void，因为当你通过可选链调用的时候返回值一定会是一个可选类型。这允许你使用 if 语句来检查是否能调用 printNumberOfRooms() 方法，就算是方法自身没有定义返回值也可以。通过对比调用 printNumberOfRooms 返回的值是否为 nil 来确定方法的调用是否成功：

```
if john.residence?.printNumberOfRooms() != nil {
    print("It was possible to print the number of rooms.")
} else {
    print("It was not possible to print the number of rooms.")
}
// Prints "It was not possible to print the number of rooms."
```

如果你尝试通过可选链来设置属性也是一样的。上边通过可选链访问属性中的例子尝试设置 address 值给 john.residence，就算是 residence 属性是 nil 也行。任何通过可选链设置属性的尝试都会返回一个 Void? 类型值，它允许你与 nil 比较来检查属性是否设置成功：

```
if (john.residence?.address = someAddress) != nil {
    print("It was possible to set the address.")
} else {
    print("It was not possible to set the address.")
}
// Prints "It was not possible to set the address."
```

通过可选链访问下标

你可以使用可选链来给可选项下标取回或设置值，并且检查下标的调用是否成功。

> 注意
>
> 当你通过可选链访问一个可选项的下标时，你需要把问号放在下标括号的*前边*，而不是后边。可选链的问号一定是紧跟在可选项表达式的后边的。

下边的栗子尝试使用下标取回 Residence 类里 john.residence 属性的数组 rooms 里第一个房间的名字。由于 john.residence 目前是 nil，下标的调用失败了：

```
if let firstRoomName = john.residence?[0].name {
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// Prints "Unable to retrieve the first room name."
```

可选链问号在下标的调用中紧跟 john.residence，在下标的方括号之前，因为 john.residence 在可选链被访问时是可选值。
同样的，你可以尝试通过在可选链里用下标来设置一个新值：

```
john.residence?[0] = Room(name: "Bathroom")
```

这个下标设置的尝试同样失败了，因为 residence 目前还是 nil 。
如果你创建并且赋值一个实际的 Residence 实例给 john.residence ，在 rooms 数组里添加一个或者多个 Room 实例，你就可以通过可选链使用 Residence 下标来访问 rooms 数组里的实际元素了：

```
let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "Living Room"))
johnsHouse.rooms.append(Room(name: "Kitchen"))
john.residence = johnsHouse
if let firstRoomName = john.residence?[0].name {
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// Prints "The first room name is Living Room."

```

访问可选类型的下标

如果下标返回一个可选类型的值——比如说 Swift 的 Dictionary 类型的键下标——放一个问号在下标的方括号*后边*来链接它的可选返回值：

```
var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]
testScores["Dave"]?[0] = 91
testScores["Bev"]?[0] += 1
testScores["Brian"]?[0] = 72
// the "Dave" array is now [91, 82, 84] and the "Bev" array is now [80, 94, 81]
```

上面的栗子中定义了一个叫做 testScores 的字典，它包含两个键值对把 String 类型的键映射到一个整型值的数组。这个栗子用可选链把 "Dave" 数组中第一个元素设为 91 ；把 "Bev" 数组的第一个元素增加 1 ；然后尝试设置 "Brian" 数组中的第一个元素。前两个调用是成功了，因为 testScores 字典包含了 "Dave" 和 "Bev" 这两个键。第三个调用失败了，因为字典 testScores 并没有包含 "Brian" 键。

链的多层连接

你可以通过连接多个可选链来在模型中深入访问属性、方法以及下标。总之，多层可选链不会给返回的值添加多层的可选性。

也就是说：

- 如果你访问的值不是可选项，它会因为可选链而变成可选项；
- 如果你访问的值*已经*是可选的，它不会因为可选链而变得*更加*可选。

因此：

- 如果你尝试通过可选链取回一个 Int 值，就一定会返回 Int?，不论通过了多少层的可选链；
- 类似地，如果你尝试通过可选链访问 Int? 值，Int? 一定就是返回的类型，无论通过了多少层的可选链。

下边的栗子尝试访问 john 的 residence 属性里的 address 属性里的 street 属性。这里一共使用了两层可选链，以链接 residence 和 address 属性，它们都是可选类型：

```
if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
// Prints "Unable to retrieve the address."
```

john.residence 的值当前包含合法的 Residence 实例。总之，john.residence.address 的值目前为 nil。因此，john.residence?.address?.street 的调用失败了。
需要注意的是上面的栗子中，你尝试取回的 street 属性。它的类型为 String?。
john.residence?.address?.street 的返回值自然也是 String?，即使对属性的可选项来说已经通过了两层可选链。

如果你设置一个 Address 实例作为 john.residence.address 的值，并且为地址的 street 属性设置一个实际的值，你就可以通过多层可选链访问 street 属性的值了：

```
let johnsAddress = Address()
johnsAddress.buildingName = "The Larches"
johnsAddress.street = "Laurel Street"
john.residence?.address = johnsAddress
if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
// Prints "John's street name is Laurel Street."
```

在上面的栗子中，对 john.residence 的 address 属性赋值能够成功，是因为 john.residence 的值目前包含了一个可用的 Residence 实例。

用可选返回值链接方法

先前的例子说明了如何通过可选链来获取可选类型属性的值。你还可以通过可选链来调用返回可选类型的方法，并且如果需要的话可以继续对方法的返回值进行链接。

在下面的栗子通过可选链来调用 Address 的 buildingIdentifier() 方法。这个方法返回 String? 类型值。正如上面所说，通过可选链调用的方法的最终返回的类型还是 String?：

```
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
    print("John's building identifier is \(buildingIdentifier).")
}
// Prints "John's building identifier is The Larches."
```

如果你要进一步对方法的返回值进行可选链，在方法 buildingIdentifier() 的圆括号*后面*加上可选链问号：

```
if let beginsWithThe =
    john.residence?.address?.buildingIdentifier()?.hasPrefix("The") {
        if beginsWithThe {
            print("John's building identifier begins with \"The\".")
        } else {
            print("John's building identifier does not begin with \"The\".")
        }
}
// Prints "John's building identifier begins with "The"."
```

> 注意
>
> 在上面的例子中，在方法的圆括号*后面*加上可选链问号，是因为链中的可选项是 buildingIdentifier() 的返回值，而不是 buildingIdentifier() 方法本身。