到底该在什么地方使用implicit optional

Back to series (/series/optional-is-not-an-option)

字号
字号
默认主题
金色主题
暗色主题

在Swift里，有一类特殊的optional，叫做implicitly unwrapped optional。如果你在Swift的官方文档中查找它，就会看到下面这样教科书一样的解释和用法。

首先，我们可以用 ！来定义一个implicitly unwrapped optional：

```
var eleven: Int! = 11
```

然后，我们就可以像访问一个普通 Int 一样访问 eleven ，而无需强行使用 ！来读取它的值：

```
if eleven == 11 {
    print(eleven)
}
```

但是，方便也是有代价的，如果implicitly unwrapped optional的值为 nil ，访问它就会立即引发运行时错误，进而使app崩溃。

每每看到这些，我们都会想，一向以安全著称的Swift，为什么要引入这么个不安分的元素呢？一个我们明知用了会有运行隐患的变量，应该在什么时候使用呢?

实际上，在Swift 3里，implicitly unwrapped optional直接出场的机会已经非常少了。简单来说，主要有两个场景：

- 用来传承Objective-C中对象指针的语义；
- 用来定义那些初始为 nil ，但一定会经过既定流程之后，就再也不会为 nil 的变量；

我们先来看第一种。

用来传承Objective-C中对象指针的语义

在Objective-C里，没有办法表达“一个对象的引用有可能为 nil ”这样的语义。OC中唯一的手段，就是返回一个对象的指针给你，然后你自己来编写处理指针指向内容的代码。

例如：

```
UIImage *image = [UIImage imageNamed: @"someImage"]
```

我们只能通过判断 image 是否为 NULL 来自行处理指针为空的情况。但当上面的API bridge到Swift之后，Swift就可以使用一个 UIImage? 来表示图片对象有可能不存在的情况。于是：

```
let image = UIImage(named: "someImage")
```

这时 image 的类型，就是一个普通的 Optional<UIImage> ，表示获得的对象有可能不存在，这看似一切都顺利成章。

实际上，所有在Objective-C中返回一个对象指针的API，bridge到Swift里，都会得到一个普通的optional。但是……

在绝大多数时候，Objective-C的API并不会返回一个空对象。例如，我们明确加载一个项目本地的图片名，对于这种一定会成功的操作，我们用optional的方式用起来就很麻烦。 if let 也好，force unwrapping也好，对于一个明知就存在的对象，这多少显得有点儿多余。

这时，implicitly unwrapped optional就派上用场了。对于明知一定会成功的调用，我们就可以用这种optional来直接获取API返回的对象：

```
let image: UIImage! = UIImage()
```

然后，我们就可以像使用普通的 UIImage 一样来访问它的属性了：

```
image.backgroundColor
```

但是，谨记，只有当你确定这绝对安全的时候，才可如此，一旦你访问的 `image` 为 `nil` ，就会立刻引发运行时错误。

接下来，我们看implicitly unwrapped optional的第二种应用场景。

用来定义那些经过既定流程之后，就再也不会为 `nil` 的变量

一个典型的例子，就是我们在 `ViewController` 中，添加UI控件的时候，XCode会把添加进来的 `@IBOutlet` 设置成implicitly unwrapped optional。例如：

```
class ViewController: UIViewController {
    @IBOutlet weak var btn: UIButton!
    // ...
}
```

这是因为，尽管在创建 `ViewController` 对象的时候， `btn` 的值会是 `nil` ，但你知道， `viewDidLoad` 方法一定会被调用， `btn` 也一定会被初始化，并且，一旦初始化完成，在 `ViewController` 对象的生命周期里，它就再也不会变成 `nil` 了。

这样，不仅解决了 `btn` 的初始化会比 `ViewController` 晚的问题，它也向 `btn` 的使用者隐藏了这是一个optional的事实，让它用起来就像是一个普通的 `UIButton` 对象，两全其美。

除了用在UI中之外，稍后我们讨论对象循环引用的问题时，还会看到更多implicitly unwrapped optional的用法。

但至少现在，你应该对implicitly unwrapped optional的用法，感到具体和实际一些了，应该说，它的存在，绝大多数时候，都不是为了写起来更简单这么个可有可无的理由。

它还是个optional，终究不是个普通对象

尽管implicitly unwrapped optional用起来，和普通的对象一样，但它终究还是一个optional，如果你查看之前我们使用的 `eleven` 就会发现，它的类型是 `ImplicitlyUnwrappedOptional<Int>` ，而不是一个简单的 `Int` 。这说明什么呢?

一方面，我们之前讨论的关于optional的所有操作，对于implicitly unwrapped optional来说，都是适用的。 `if let` ， `map` 和 `flatMap` ，optional chaining统统都没问题；

另一方面，访问implicitly unwrapped optional获得的值是一个右值，因此，它也不能作为函数的 `inout` 参数：

```
func double(_ i: inout Int) {
    i = i * 2
}

double(&eleven) // Error
```

否则，我们就会得到一个类似下面这样的错误：

以上，就是关于implicitly unwrapped optional的内容。简而言之，这更多是为了提供和历史代码的兼容，以及解决非常特定的语言环境产生的问题而存在的一种类型。而在我们平时的编程中，还是那句话，总是应该优先使用普通的optional。

两个调试optional的小技巧 (https://www.boxueio.com/series/optional-is-not-an-option/ebook/146)

返回视频 (/series/optional-is-not-an-option)

从开发、测试到运维，让技术不再成为你成长的绊脚石。我们用打磨产品的精神去传播知识，把最新的移动开发技术，通过简单的图表， 清晰的视频，简明的文字和切实可行的例子一一向你呈现。让学习不仅是一种需求，也是一种享受。

泊学动态

一个工作十年PM终创业的故事（二） (https://www.boxueio.com/after-the-full-upgrade-to-swift3)
Mar 4, 2017

人生中第一次创业的"10有" (https://www.boxueio.com/founder-chat)
Jan 9, 2016

猎云网采访报道泊学 (http://www.lieyunwang.com/archives/144329)
Dec 31, 2015

What most schools do not teach (https://www.boxueio.com/what-most-schools-do-not-teach)
Dec 21, 2015

一个工作十年PM终创业的故事（一） (https://www.boxueio.com/founder-story)
May 8, 2015

泊学相关

关于泊学 ›

加入泊学 ›

泊学用户隐私以及服务条款 (HTTPS://WWW.BOXUEIO.COM/TERMS-OF-SERVICE)

版权声明 (HTTPS://WWW.BOXUEIO.COM/COPYRIGHT-STATEMENT)

联系泊学

Email: 10[AT]boxue.io (mailto:10@boxue.io)

QQ: 2085489246

友情链接 SwiftV (http://www.swiftv.cn) | Seay信息安全博客 (http://www.cnseay.com) | Swift.gg (http://swift.gg/) | Laravist (http://laravist.com/) | SegmentFault (https://segmentfault.com) | 靛青K的博客 (http://blog.dianqk.org/)