内存安全性

快速检索 [点击收起]

1 了解内存访问冲突
    1.1 典型的内存访问
2 输入输出形式参数的访问冲突
3 在方法中对 self 的访问冲突
4 属性的访问冲突

默认情况下，Swift 会阻止你代码中发生的不安全行为。比如说，Swift 会保证在使用前就初始化，内存在变量释放后这块内存就不能再访问了，以及数组会检查越界错误。

Swift 还通过要求标记内存位置来确保代码对内存有独占访问权，以确保了同一内存多访问时不会冲突。由于 Swift 自动管理内存，大部份情况下你根本不需要考虑访问内存的事情。总之，了解一下什么情况下会潜在导致冲突是一件很重要的事情，这样你就可以避免写出对内存访问冲突的代码了。如果你的代码确实包含冲突，你就会得到编译时或运行时错误。

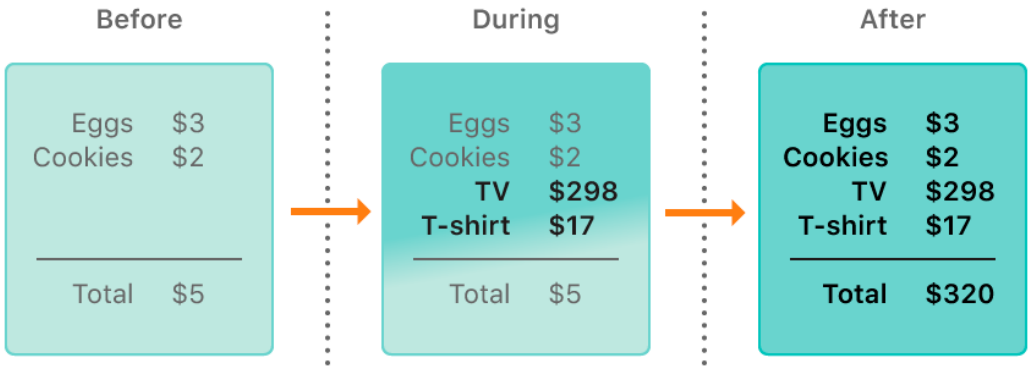
了解内存访问冲突

内存访问会在你做一些比如设置变量的值或者传递一个实际参数给函数的时候发生。比如说，下面的代码同时包含了读取访问和写入访问：

```
1 // A write access to the memory where one is stored.
2 var one = 1
3
4 // A read access from the memory where one is stored.
5 print("We're number \(one)!")
```

内存访问冲突会在你的代码不同地方同一时间尝试访问同一块内存时发生。在同一时间多处访问同一块内存会产生不可预料或者说不一致的行为。在 Swift 中，有好几种方式来修改跨越多行代码的值，从而可以在访问值的中间进行它自身的修改。

你可以想象一个类似的问题比如你是如何更新一张写在纸上的预算。更新预算是一个两步过程：首先你添加项目的名字和价格，然后你改变总价来显示当前列表中的变化。在更新的前后，你可以读取任何预算信息并且得到正确的结果，图例如下：



当你添加项目到预算时，它处在一个临时的状态，这是一个不可用的状态因为总价还没有更新来显示最新的价格。在添加新项目的过程中读取总价就会得到错误的信息。

这个例子也展示了你在修复内存访问冲突时可能会遭遇的挑战：有时多种修复冲突的方式会产生不同的结果，并且通常也不会明显哪个结果就是正确的。在这个例子中，基于你想要原本的总价还是更新后的总价，要么是 $5 要么是 $320 都可能是正确的。在你修复内存访问冲突之前，你必须决定想要哪一种。

> 注意
>
> 如果你写并发或者多线程代码，内存访问冲突可能会是一个常见问题。总之，我们这里讨论的访问冲突也可以发生在单线程并且不涉及并发和多线程代码。
>
> 如果你在单线程遇到内存访问冲突，Swift 会保证你在要么编译时要么运行时得到错误。对于多线程代码，使用Thread Sanitizer 来帮助探测线程之间的访问冲突。

典型的内存访问

在访问冲突上下文中有三种典型的内存访问需要考虑：不论访问是读取还是写入，在访问过程中，以及内存地址被访问。具体来说，冲突会在你用两个访问并满足下列条件时发生：

- 至少一个是写入访问；
- 它们访问的是同一块内存；
- 它们的访问时间重叠。

读和写的区别通常显而易见：写入访问改变了内存，但读取访问不会。内存地址则指向被访问的东西——比如说，变量、常量或者属性。访问内存的时间要么是即时的，要么是长时间的。

如果一个访问在启动后其他代码不能执行直到它结束后才能，那么这个访问就是*即时*的。基于它们的特性，两个即时访问不能同时发生。大多数内存访问都是即时的。比如，下面列出的所有读写访问都是即时的：

```
func oneMore(than number: Int) -> Int {
    return number + 1
}

var myNumber = 1
myNumber = oneMore(than: myNumber)
print(myNumber)
// Prints "2"
```

总之，还有有很多访问内存的方法，比如被称作*长时*访问的，跨越其他代码执行过程。长时访问和即时访问的不同之处在于长时访开始后在它结束之前其他代码依旧可以运行，这就是所谓的*重叠*。长时访问可以与其他长时访问以及即时访问重叠。

重叠访问主要是出现在使用了输入输出形式参数的函数以及方法或者结构体中的异变方法。特定种类的使用长时访问的 Swift 代码在下文详述。

输入输出形式参数的访问冲突

拥有长时写入访问到所有自身输入输出形式参数的函数。对输入输出形式参数的写入访问会在所有非输入输出形式参数计算之后开始，并持续到整个函数调用结束。如果有多个输入输出形式参数，那么写入访问会以形式参数出现的顺序开始。

这种长时写入访问的一个后果就是你不能访问作为输入输出传递的原本变量，就算生效范围和访问控制可能会允许你这么做——任何对原变量的访问都会造成冲突，比如说：

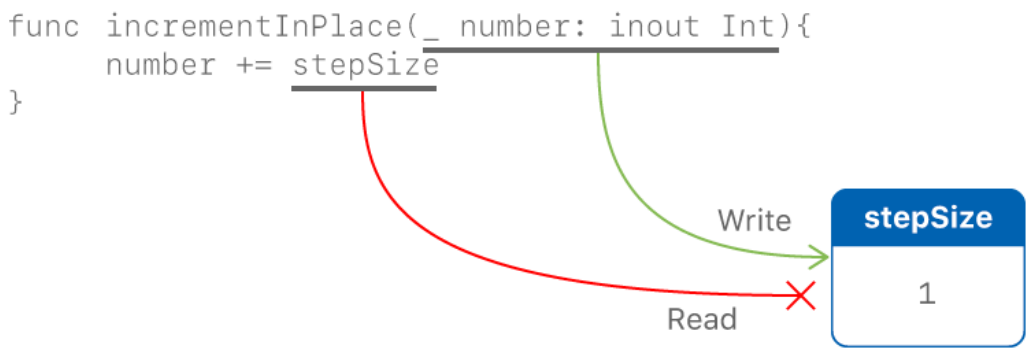
```
var stepSize = 1

func increment(_ number: inout Int) {
    number += stepSize
```

```
}

increment(&stepSize)
// Error: conflicting accesses to stepSize
```

在上面的代码中， `stepSize` 是一个全局变量，它通常可以在 `increment(_:)` 中可以访问。总之， `stepSize` 的读取访问与 `number` 的写入访问重叠了。入下图所示， `number` 和 `stepSize` 引用的是同一内存地址。读取和写入访问引用同一内存并且重叠，产生了冲突。



一种解决这个冲突的办法是显式地做一个 `stepSize` 的拷贝：

```
// Make an explicit copy.
var copyOfStepSize = stepSize
increment(&copyOfStepSize)

// Update the original.
stepSize = copyOfStepSize
// stepSize is now 2
// stepSize is now 2
```

当你在调用 `increment(_:)` 之前给 `stepSize` 了一份之后，显然 `copyOfStepSizeis` 基于当前的步长增加了。读取访问在写入访问开始前结束，所以不会再有冲突。

输入输出形式参数的长时写入访问的另一个后果是传入一个单独的变量作为实际形式参数给同一个函数的多个输入输出形式参数产生冲突。比如：

```
func balance(_ x: inout Int, _ y: inout Int) {
    let sum = x + y
    x = sum / 2
    y = sum - x
```

```
}
var playerOneScore = 42
var playerTwoScore = 30
balance(&playerOneScore, &playerTwoScore)  // OK
balance(&playerOneScore, &playerOneScore)
// Error: Conflicting accesses to playerOneScore
```

上边 `balance(_:_:)` 修改它的两个形式参数将它们的总数进行平均分配。用 `playerOneScore` 和 `playerTwoScore` 作为实际参数不会产生冲突——一共有两个写入访问在同一时间重叠，但它们访问的是不同的内存地址。相反，传入 `playerOneScore` 作为两个形式参数的值则产生冲突，因为它尝试执行两个写入访问到同一个内存地址且是在同一时间执行。

> 注意
>
> 由于操作是函数，它们同样也可以对其输入输出形式参数进行长时访问，比如，如果 `balance(_:_:)` 是一个名为 `<^>` 的操作符函数，写 `playerOneScore <^> playerOneScore` 就会造成和 `balance(&playerOneScore, &playerOneScore)` 一样的冲突。

在方法中对 self 的访问冲突

结构体中的异变方法可以在方法调用时对 `self` 进行写入访问。比如说想象一个每个玩家都有生命值的游戏，当玩家受伤时降低生命值，以及一个能量值，它在使用技能时降低。

```
struct Player {
    var name: String
    var health: Int
    var energy: Int

    static let maxHealth = 10
    mutating func restoreHealth() {
        health = Player.maxHealth
    }
}
```

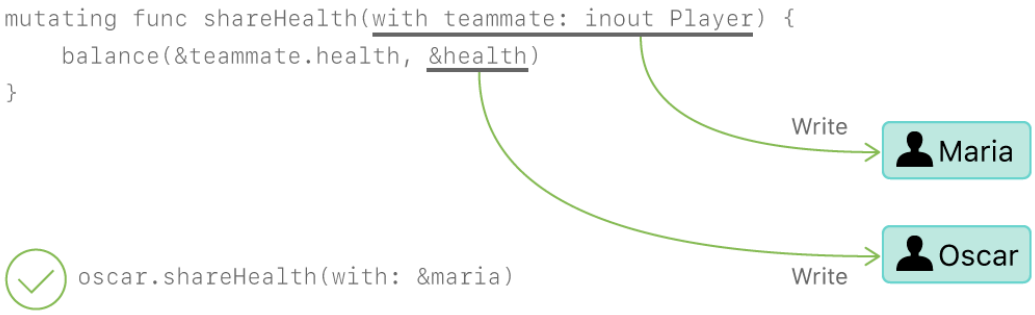
在上面 `restoreHealth()` 方法中，对 `self` 的写入访问在方法一开始就启动然后结束于方法返回。在这种情况下，在 `restoreHealth()` 中没有其他代码可能会重叠访问 `Player` 实例中的属性。下面的

`shareHealth(with:)` 方法则接收另一个 `Player` 实例作为输入输出形式参数，为重叠访问创建了可能性：

```
extension Player {
    mutating func shareHealth(with teammate: inout Player) {
        balance(&teammate.health, &health)
    }
}

var oscar = Player(name: "Oscar", health: 10, energy: 10)
var maria = Player(name: "Maria", health: 5, energy: 10)
oscar.shareHealth(with: &maria)  // OK
```

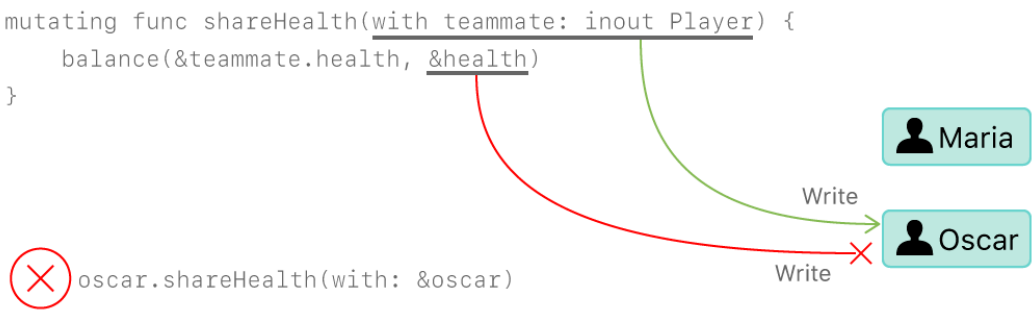
在上面的例子中，调用 Oscar 玩家的 `shareHealth(with:)` 方法分享血条给 Maria 玩家不会造成冲突。在方法调用的过程中只有一个写入访问到 `oscar` 因为 `oscar` 是异变方法中 `self` 值，并且在同一时间内只有一个写入访问到 `maria` 因为 `maria` 是以输入输出形式参数传递的。如下面的图例所示，他们访问了内存中不同的地址。就算他们的写入访问是同时发生的，但不会冲突。



总之，如果你把 `oscar` 作为实际参数传递给 `shareHealth(with:)` ，就出现了冲突：

```
oscar.shareHealth(with: &oscar)
// Error: conflicting accesses to oscar
```

异变方法在方法的自行过程中需要写入访问到 `self` ，并且输入输出形式参数同时也需要写入访问到 `teammate` 。在方法中， `self` 和 `teammate` 实际上引用自同一内存地址——如图所示。这两个写入访问引用到了同一个地址并重叠，产生冲突。



属性的访问冲突

像是结构体、元组以及枚举这些类型都是由独立值构成，比如结构体的属性或者元组的元素。由于这些都是值类型，改变任何一个值都会改变整个类型，意味着读或者写访问到这些属性就需要对整个值进行读写访问。比如说，对元组的元素进行重叠写入访问就会产生冲突：

```
var playerInformation = (health: 10, energy: 20)
balance(&playerInformation.health, &playerInformation.energy)
// Error: conflicting access to properties of playerInformation
```

在上边例子中，在同一元组的元素上调用 `balance(_:_:)` 产生冲突是因为对 `playerInformation` 产生了重叠写入访问。`playerInformation.health` 和 `playerInformation.energy` 作为输入输出形式参数传入，这就意味着 `balance(_:_:)` 在函数调用的过程中对他们产生写入访问。在这两种情况下，对元组元素的写入访问需要整个元组的写入访问。也就是说 `playerInformation` 在调用过程中有两个写入访问重叠，导致冲突。

下面的代码显示了对全局变量结构体属性的重叠写入访问，导致同样的错误。

```
var holly = Player(name: "Holly", health: 10, energy: 10)
balance(&holly.health, &holly.energy)  // Error
```

实际上，大多数对结构体属性的访问可以安全的重叠。比如果，如果上边变量 `holly` 变成局部变量而不是全局变量，那么编译器就可以保证重叠访问结构体的存储属性是安全的：

```
func someFunction() {
```

```
    var oscar = Player(name: "Oscar", health: 10, energy: 10)
    balance(&oscar.health, &oscar.energy)  // OK
}
```

在上边的例子中，Oscar 的血条和能量作为两个输入输出形式参数传递给了 `balance(_:_:)` 。编译器可以证明内存安全得以保证是因为两个存储属性不会以任何形式交互。

对重叠访问结构体的属性进行限制并不总是必要才能保证内存安全性。内存安全性是一个需要的保证，但独占访问是比内存安全更严格的要求——也就是说某些代码保证了内存安全性，尽管它违反了内存的独占访问。如果编译器可以保证非独占访问内存仍然是安全的 Swift 就允许这些内存安全的代码。具体来说，如果下面的条件可以满足就说明重叠访问结构体的属性是安全的：

- 你只访问实例的存储属性，不是计算属性或者类属性；
- 结构体是局部变量而非全局变量；
- 结构体要么没有被闭包捕获要么只被非逃逸闭包捕获。

如果编译器不能保证访问是安全的，它就不允许访问。
