

内存安全

4.2 ▾

Swift 编程语言 / 内存安全

这是一篇社区协同翻译的文章，你可以点击右边区块信息里的『改进』按钮向译者提交改进建议。

通常情况下，Swift 可以防止代码中发生不安全的行为。例如，在你变量被访问的时候 Swift 会确保它已经被初始化了，内存被释放之后就不在允许访问了，而且通过索引访问数组元素也会检查是不是有越界的错误。

Swift 也会通过要求访问一块内存空间的代码保持互斥，以此来确保在多线程访问同一内存空间时没有冲突。因为 Swift 是自动管理内存的，所以通常你不必关心访问内存的事情。但是，了解在哪个地方可能会出现内存访问冲突也是很重要的，为此你可以避免编写会引起内存访问冲突的代码。如果你的代码中有冲突，你会收到编译期或者运行时的警告。

理解产生内存访问冲突的原因

当你做一些像给一个变量赋值或者向一个方法传参数这样的操作时，你就是在访问内存。例如，下面的实例代码中包含了一个读内存的操作还有一个写入操作：

```
// 写内存的操作
var one = 1

// 读内存的操作
print("We're number \(one)!")
```

当你有多处代码在同一时间访问同一块内存地址时，内存访问冲突就可能会出现。在同一时间多次访问同一快内存地址会造成无法预料的后果。在 Swift 中，修改一个变量有多种途径分几行代码完成，在修改它的过程中也可以访问该变量。

设想一下怎样更新一张写在纸上的预算清单，你便会发现类似的问题。

更新预算清单需要分两步：首先，添加条目的名称还有价格，然后更新包含目前清单上所有条目的总价。如下图所示，在你更新清单之前或者之后，你可以读取清单中的任意信息，并

Infinity

翻译进度

10

分块数量

5

参与人数



bbx1209 翻译于 5个月

0

重译

由 Summer 审阅



bbx1209 翻译于 5个月

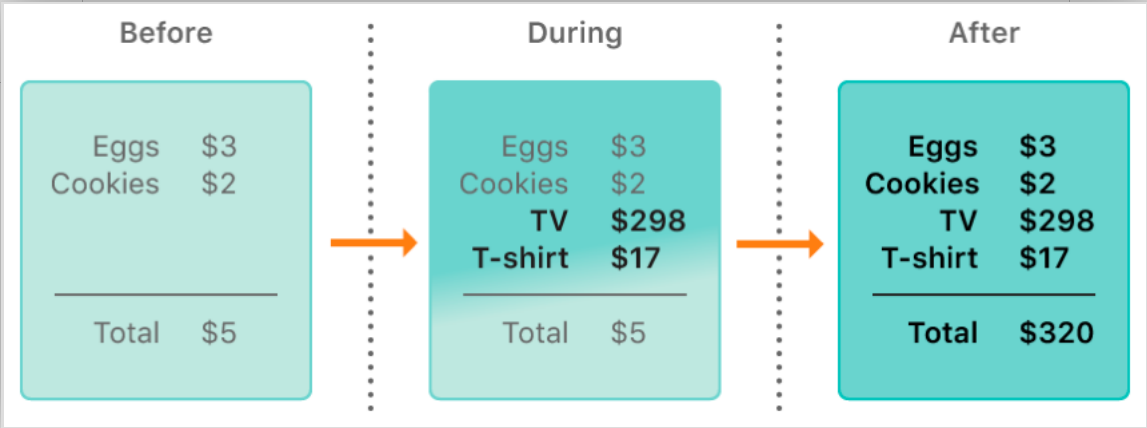
0

重译

由 Aufree 审阅



且读出的数据是正确的。



当你正在往预算清单中添加条目的时候，它处于临时的，无效的状态，因为还没来得及把新添加的条目更新到总价中。在添加条目的过程中读取的总价是不正确的。

这个例子同时也说明了修正内存访问冲突所面临的一种挑战：有多种方法来解决这种有不同结果的冲突，而且这些方法也没有明显的孰优孰劣。在这个例子中，正确结果取决于你到底想要的是原始总价还是更新之后的总价，所以 \$5 或 \$320 都可以是正确的结果。在修正访问冲突前，你需要确定你倾向于哪种结果。

注意

如果你在写并发或是多线程的代码，内存访问冲突是很常见的问题。然而，访问冲突也可能发生在单线程中，并且 不涉及 并发或是多线程。

如果你在单线程中遇到内存访问冲突，Swift 会保证你在编译或是运行时得到错误。对于多线程的代码，可以参考「[线程检查](#)」检查多线程访问冲突。

## 内存访问的要素

参照内存访问的上下文，内存访问有三个要素：是不是读或者写操作，该操作的时长，被访问的内存地址。具体的来说，当你有两个访问操作放生如下情形时，就会发生访问冲突：

- 至少有一个是写操作。
- 它们访问的是同一块内存地址。
- 它们的访问时间发生了重叠。

内存读写操作的区别通常是明显的：写操作会改变内存中存储的地址，但是读操作不会。内存中存储的地址是正在访问的对象--- 例如，一个变量，常量，或者属性。内存访问的时长可以使短暂的瞬间也可以很长。

如果在一个访问完成之前，其他的代码一定不会执行，那么这个访问就是 *瞬时的*。本质上，两个瞬时访问操作是不可能在同一时间执行的。大部分的内存访问都是瞬时的。例如，下面



chdzq 翻译于 5个月前

👍 0

重译

由 Aufree 审阅



bbx1209 翻译于 5个月前

👍 0

重译

由 Summer 审阅

代码清单所示的读写访问都是瞬时的：

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

var myNumber = 1
myNumber = oneMore(than: myNumber)
print(myNumber)
// 打印 「 2 」
```

不过，还有一些被称为 长期 访问的内存存取方法，它们嵌套调用其他代码。瞬时访问与长期访问的区别就是在后者调用结束之前，其他的代码可能会并发执行，这种现象被称为 相交。一个长期访问可以与另一个长期访问或者瞬时访问相交。

相交访问主要发生在使用 in-out 参数的函数和方法或者结构体的方法中。这些特殊类型的长期的 Swift 内存访问代码，将在接下来的章节中讨论。

## In-Out 参数引起的访问冲突

函数对它的所有 in-out 参数拥有长期写访问权。in-out 参数的写访问是在所有的非 in-out 参数处理完之后开始，持续到函数调用完毕为止。如果有多个 in-out 参数，写访问开始的顺序和参数的顺序一致。

拥有长期的写访问权会造成一个问题，就是你不能一边把值作 in-out 传递一边又去访问这个原始值，即使作用域和访问权限是允许的--任何访问原始值都会造成冲突。比如：

```
var stepSize = 1

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

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

上面的代码中， `stepSize` 是全局变量，并且它可以在 `increment(_:)` 函数内部正常访问。然而， `stepSize` 的读访问权和 `number` 的写访问权重叠了。像下图展示的那样， `number` 和 `stepSize` 指向同一块内存地址。同一块内存的读和写访问权重叠了，因而产生了冲突。



bbx1209 翻译于 5个月

👍 0

重译

由 Summer 审阅

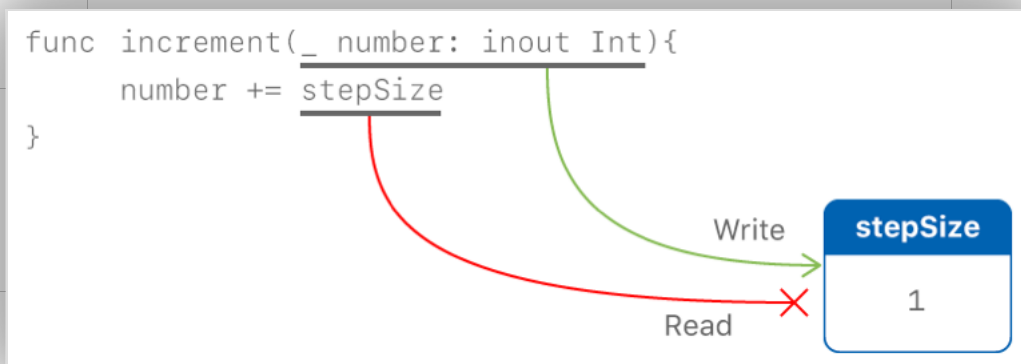


chdzq 翻译于 5个月前

👍 0

重译

由 Aufree 审阅



一种解决这个冲突的方法是显性拷贝一个 `stepSize` 的副本：

```
// 显性拷贝一个副本。
var copyOfStepSize = stepSize
increment(&copyOfStepSize)

// 更新初始值
stepSize = copyOfStepSize
// stepSize 现在是 2
```

当你在调用 `increment(_:)` 函数前，对 `stepSize` 进行了拷贝，那么很明显 `copyOfStepSize` 的值已经被当前的 `step size` 增加了。在写入权限开始之前读取权限就已经结束了，所以不存在访问冲突。

另一个由 in-out 参数的长期写入权限引起的访问冲突是将单一变量作为参数传进有多元 in-out 参数的函数。例如：

```
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)
// 错误：访问 playerOneScore 冲突
```

上面的 `balance(_:_:)` 函数修改了它的两个参数，将二者的和平均分给了它们。将 `playerOneScore` 和 `playerTwoScore` 作为参数传入函数时，并没有引起冲突 --- 虽然有两个写入权限恰好重叠了，但是它们访问的是不同的内存地址。相反，仅透传 `playerOneScore` 作为两个参数的值引起了访问冲突，因为它试图在同一时间对同一内存地址执行两个写入权限。



bbx1209 翻译于 5个月

👍 0

重译

由 RecherJ 审阅

注意

因为算子也是函数，它们同样拥有长时间访问其 in-out 参数的能力。例如，假如

`balance(_:_:)` 是一个名为 `<^>` 算子函数，那么这么写

`playerOneScore <^> playerOneScore` 会导致与

`balance(&playerOneScore, &playerOneScore)` 一样的冲突。

## 方法内部 self 的访问冲突

一个结构体的 mutating 方法在其被调用期间对 `self` 有写访问权。例如，构思这样一个游戏，游戏中每一个玩家都有一定血量，每次受到伤害就会减少，并且玩家还拥有一定法力值，每当使用特殊技能也会减少。

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

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

在上面的 `restoreHealth()` 方法中，对 `self` 的写访问从方法开始执行一直到方法返回。在这种情况下，方法 `restoreHealth()` 里的其他代码不可以对 `Player` 实例的属性有重叠访问。下面的方法 `shareHealth(with:)` 持有了另一个 `Player` 实体做为 in-out 参数，使得重叠访问成为了可能。

```
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，玩家 Oscar 调用方法 `shareHealth(with:)` 跟玩家 Maria 平分血量也



bbx1209 翻译于 5个月

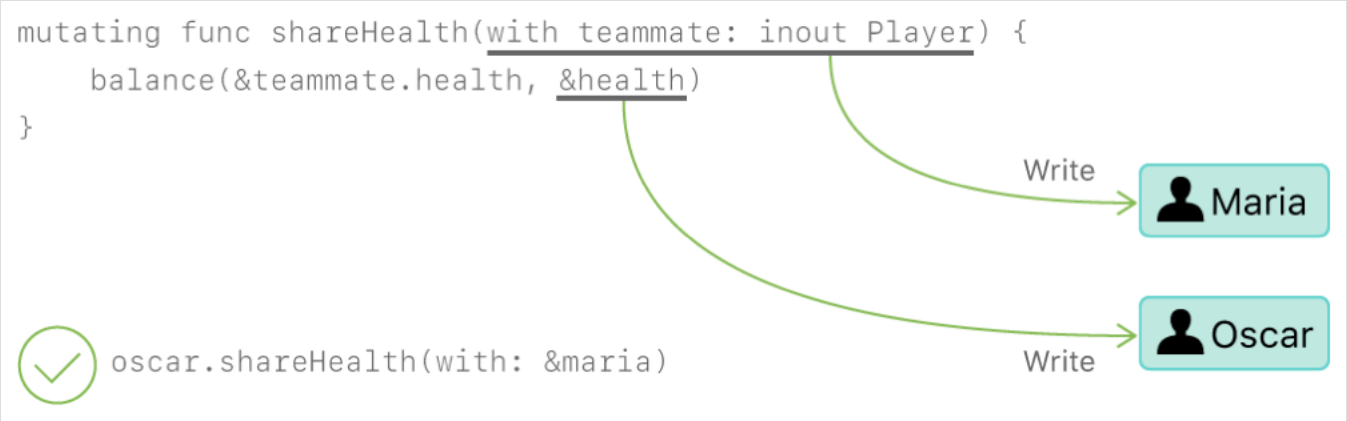
👍 0

重译

由 chdzq 审阅



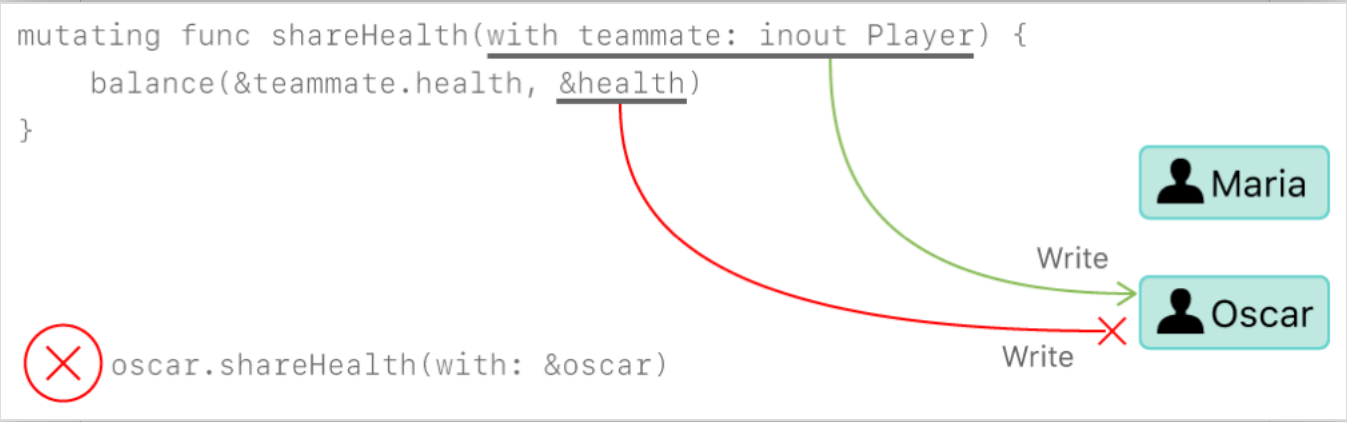
没有引起访问冲突。在这个方法调用期间，会对 `oscar` 发起写访问，由于在 mutating 方法里 `self` 就是 `oscar`，所以当 `maria` 被当做 in-out 参数传递时，同时对 `maria` 也发起了写访问。如下图所示，他们访问内存中的不同的地址。尽管他们在同一时间写访问重叠了，也没有造成冲突。



不过，如果你将 `oscar` 作为参数传进 `shareHealth(with:)` 方法中，就会产生冲突：

```
oscar.shareHealth(with: &oscar)  
// 错误: oscar 访问冲突
```

mutating 方法执行的过程中，都需要对 `self` 有写访问权，而同时 in-out 参数也需要对 `teammate` 有写访问权。在该方法中，`self` 和 `teammate` 所指向的是同一个内存地址 ---就如下图展示的那样。这对同一块内存进行了两个写访问，并且重叠了，因此产生了冲突。



## 属性访问冲突

诸如结构体、元组、还有枚举这样的数据类型，都是由独立的值构成的，比如结构体的属性或者元组的元素。由于这些数据类型都是值类型，改变其中任意一个值就相当于改变了其整体，也就是说对一个属性的读写访问需要对其整体进行读写访问。例如，元组元素的写访问重叠就会产生冲突：

```
var playerInformation = (health: 10, energy: 20)
```



bbx1209 翻译于 5个月

👍 0 重译

由 Aufree 审阅

```
balance(&playerInformation.health, &playerInformation.energy)
```

```
// 错误: playerInformation 的属性访问冲突
```

上面的例子中，对一个元组的元素调用 `balance(_:_:)` 方法产生了访问冲突，因为存在对 `playerInformation` 的写访问重

叠。`playerInformation.health` 和 `playerInformation.energy` 作为 in-out 参数传入方法，这意味着在 `balance(_:_:)` 方法执行期间需要对它们进行写访问。在这两种情况下，一个写访问针对元组的元素，一个写访问针对整个元组。意思是说在该方法执行期间 `playerInformation` 有两个重叠的写访问，因此引发了冲突。

下面的代码显示了，对一个全局结构体的属性进行重叠写访问时出现了相同的错误。

```
var holly = Player(name: "Holly", health: 10, energy: 10)
```

```
balance(&holly.health, &holly.energy) // 错误
```

实际操作中，大部分对结构体属性的重叠访问是安全的。例如，上述示例中的变量

`holly` 被修改成了局部变量而不是全局变量，编译器可以证明对结构体的存储属性进行重叠访问是安全的：

```
func someFunction() {
```

```
    var oscar = Player(name: "Oscar", health: 10, energy: 10)
```

```
    balance(&oscar.health, &oscar.energy) // OK
```

```
}
```

上面的例子中，Oscar 的血量和法力值都作为 in-out 参数传入 `balance(_:_:)` 方法。

编译器可以保证内存安全，因为这两个存储属性在任何情况下都不会相互影响。

用来确保内存安全，对结构体属性重叠访问的限制不总是必须的。内存安全是期望得到的保证，但是相比于内存安全，独占访问是更加严格的要求 --- 也就是说，一些代码可以保证内存安全，尽管它违反了对内存的独占访问。如果编译器可以确保这种非独占的内存访问是安全的，那么 Swift 也允许编写这种内存安全的代码。特别是，如果符合下列条件，编译器就可以确保对结构体属性的重叠访问是安全的：

- 你访问的仅仅是实例变量的存储属性，而不是计算属性或者类属性。
- 结构体是局部变量，不是全局变量。
- 结构体没有被任意闭包捕获，或者仅被非逃逸闭包捕获。

如果编译器不能够确保这个访问是安全的，则不允许该访问。



bbx1209 翻译于 5个月

👍 0

重译

由 Aufree 审阅

本文中的所有译文仅用于学习和交流目的，转载请务必注明文章译者、出处、和本文链接

我们的翻译工作遵照 [CC 协议](#)，如果我们的工作有侵犯到您的权益，请及时联系我们。

-----

← 上一篇

下一篇 →

👍 点赞

👤 参与译者：5



更多职位


💬 讨论数量：0


✎ 发起讨论

☐ 只看当前版本讨论

暂无话题~

兄弟社区

 [Laravel China](#)

 [PythonCaff.com](#)

 [GolangCaff.com](#)

 [VuejsCaff.com](#)

资源推荐

资源推荐

其他信息

👤 [软件外包](#)

👜 [商务合作](#)

💬 [联系站长](#)



