内存安全

默认情况下，Swift可以防止代码中出现不安全的行为。例如，Swift确保变量在使用之前被初始化，在解除分配内存后不访问内存，并检查数组索引是否出现越界错误。

Swift还可以确保对同一区域内存的多次访问不会发生冲突，因为需要修改内存中某个位置的代码才能独占访问该内存。由于Swift自动管理内存，因此大部分时间您根本不必考虑访问内存。但是，了解潜在冲突发生的位置很重要，这样可以避免编写与内存冲突的代码。如果你的代码确实包含冲突，你会得到一个编译时或运行时错误。

了解对内存的冲突访问

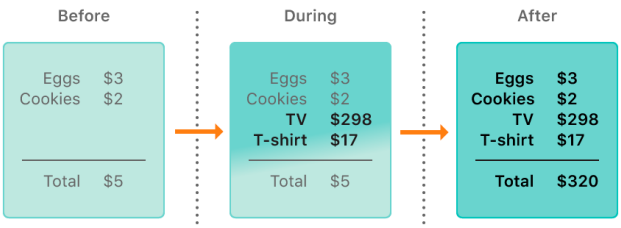
当你设置变量的值或将参数传递给函数时，访问内存发生在你的代码中。例如，下面的代码包含读取访问权限和写入访问权限：

```
// A write access to the memory where one is stored.
var one = 1

// A read access from the memory where one is stored.
print("We're number \(one)!")
```

当代码的不同部分试图同时访问内存中的相同位置时，可能会发生冲突的内存访问。同时多次访问内存中的某个位置可能会产生不可预知或不一致的行为。在Swift中，有多种方法可以修改跨越多行代码的值，从而可以尝试在自己的修改过程中访问某个值。

您可以通过考虑如何更新写在纸上的预算来看到类似的问题。更新预算分为两个步骤：首先添加项目的名称和价格，然后更改总金额以反映列表中的当前项目。更新之前和之后，您可以阅读预算中的任何信息并获得正确答案，如下图所示。



在将项目添加到预算时，它处于临时无效状态，因为总金额尚未更新以反映新添加的项目。在添加项目过程中读取总金额会给您提供不正确的信息。

这个例子还说明了在解决对内存的冲突访问时可能遇到的挑战：有时有多种方法可以解决产生不同答案的冲突，而且哪种答案是正确的并不总是显而易见的。在这个例子中，根据您是想要原始总金额还是更新的总金额，5美元或320美元可能是正确的答案。在修复冲突的访问之前，您必须确定要执行的操作。

> 注意
>
> 如果您编写了并发或多线程代码，则对内存的访问冲突可能是一个熟悉的问题。但是，这里讨论的冲突访问可能发生在单个线程上，*并且不*涉及并发或多线程代码。
>
> 如果您在单个线程内存在对内存的访问冲突，Swift保证您在编译时或运行时都会出现错误。对于多线程代码，使用Thread Sanitizer来帮助检测跨线程的冲突访问。

内存访问特性

在访问冲突的情况下，需要考虑内存访问的三个特性：访问是读取还是写入，访问的持续时间以及正在访问的内存中的位置。具体而言，如果您有两次访问满足以下所有条件，则会发生冲突：

- 至少有一个是写入权限。
- 他们访问内存中的相同位置。
- 他们的持续时间重叠。

读取和写入访问之间的区别通常很明显：写入访问会更改内存中的位置，但读取访问不会。内存中的位置指正在访问的内容，例如变量，常量或属性。内存访问的持续时间可以是瞬时的，也可以是长期的。

如果其他代码在
生。大多数内存

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

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

但是，有几种访问内存的方法，称为*长期*访问，跨越其他代码的执行。即时访问和长期访问之间的区别在于，其他代码可能在长期访问开始后但结束之前运行，这称为*重叠*。长期访问可以与其他长期访问和即时访问重叠。

重叠访问主要出现在使用函数和方法中的进出参数或结构的变异方法的代码中。下面将讨论使用长期访问的Swift代码的具体类型。

冲突访问输入输出参数

一个函数可以长期写入其所有输入参数。对所有非入出参数进行求值并在该函数调用的整个持续时间内持续进入入出参数的写访问开始。如果有多个输入输出参数，则写入访问将按照参数出现的顺序开始。

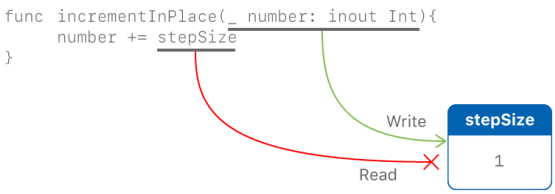
这种长期写入访问的一个结果是，即使作用域规则和访问控制允许它访问原始变量（即使作用域规则和访问控制会允许），对原始访问的任何访问都会产生冲突。例如：

```
var stepSize = 1

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

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

在上面的代码中，`stepSize`是一个全局变量，它通常可以从内部访问`incrementInPlace(_:)`。但是，读访问`stepSize`与写访问重叠`number`。如在下文中的图所示，两者`number`并`stepSize`指代相同的位置在存储器中。读取和写入访问引用相同的内存并且它们重叠，产生冲突。



解决这一冲突的一种方法是制作一份明确的副本`stepSize`：

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

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

`stepSize`在调用之前 制作副本时`incrementInPlace(_:)`，很明显该值`copyOfStepSize`会增加当前步长。在写访问开始之前读访问结束，所以没有冲突。

对输入输出参数进行长期写入访问的另一个后果是将单个变量作为参数传递给同一个函数的多个输入输出参数会产生冲突。例

```
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
```

`balance(_:_:)`上面 的函数修改了它的两个参数，以便在它们之间均匀地划分总值。使用`playerOneScore`和`playerTwoScore`作为参数调用它并不会产生冲突 - 有两个写入访问时间重叠，但它们访问内存中的不同位置。相反，`playerOneScore`作为两个参数的值传递会产生冲突，因为它会尝试同时对内存中的同一位置执行两次写入访问。

> 注意
>
> 由于操作员是功能，他们也可以长期访问其输入参数。例如，如果`balance(_:_:)`是一个名为运算符的函数`<^>`，写入`playerOneScore <^> playerOneScore`会导致与之相同的冲突`balance(&playerOneScore, &playerOneScore)`。

在方法中冲突访问自我

结构上的变异方法在`self`方法调用期间具有写访问权限。例如，考虑一种游戏，其中每个玩家具有在受到伤害时减少的健康量和在使用特殊能力时减少的能量量。

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

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

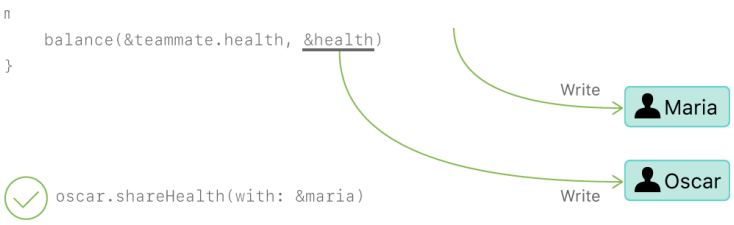
在`restoreHealth()`上面的方法中，写入权限`self`从方法的开始处开始并持续到方法返回。在这种情况下，没有其他代码`restoreHealth()`可以对`Player`实例的属性进行重叠访问。`shareHealth(with:)`下面的方法将另一个`Player`实例作为输入输出参数，从而创建重叠访问的可能性。

```
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`shareHealth(with:)`奥斯卡玩家与玛丽亚玩家分享健康的方法不会导致冲突。`oscar`在方法调用期间有写权限，因为它`oscar`是`self`变异方法中的值，并且`maria`在相同的持续时间内有写入权限，因

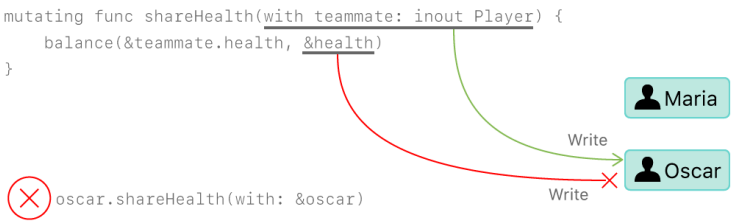
为它maria是作为输入参数传递的。如下图所示，他们访问内存中的不同位置。即使这两个写访问时间重叠，它们也不会发生冲突。



但是，如果你oscar作为参数传递给它shareHealth(with:)，会有冲突：

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

在方法self的持续时间内，变异方法需要写入权限，输入输出参数需要写入权限teammate的时间相同。在该方法中，无论是self和teammate指的是相同的位置在内存如示于下图中。这两个写访问指的是相同的内存并且它们重叠，产生冲突。



对属性的访问冲突

结构，元组和枚举类型由各个组成元素组成，例如结构的属性或元组的元素。因为这些是值类型，所以对任何一个值进行变异都会改变整个值，这意味着对其中一个属性的读取或写入访问需要读取或写入整个值。例如，对元组的元素进行重叠写访问会产生冲突：

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

在上面的例子中，调用balance(_:_:)元组的元素会产生冲突，因为存在重叠的写入访问playerInformation。两者playerInformation.health和playerInformation.energy都作为输入参数传递，这意味着balance(_:_:)需要在函数调用期间对它们进行写入访问。在这两种情况下，对元组元素的写访问都需要对整个元组进行写访问。这意味着有两个写入访问playerInformation时间重叠，导致冲突。

下面的代码显示，对于存储在全局变量中的结构的属性进行重叠写访问时会出现相同的错误。

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

实际上，大多数对结构属性的访问都可以安全地重叠。例如，如果上述示例中的变量holly更改为局部变量而不是全局变量，则编译器可以证明重叠访问结构的存储属性是安全的：

在本页

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

在上面的例子中，奥斯卡的健康和能量是作为两个输入参数传递给的balance(_:_:)。编译器可以证明内存安全性被保留，因为两个存储的属性不以任何方式进行交互。

对重叠访问结构属性的限制并不总是需要保持内存安全。内存安全是所需的保证，但独占访问是比内存安全更严格的要求 - 这意味着某些代码保留内存安全，即使它违反了对内存的独占访问。如果编译器可以证明对存储器的非独占访问仍然安全，那么Swift允许这种内存安全的代码。具体而言，如果满足以下条件，它可以证明重叠访问结构的属性是安全的：

- 您只访问实例的存储属性，而不是计算属性或类属性。
- 该结构是局部变量的值，而不是全局变量。
- 该结构要么不被任何闭包捕获，要么仅通过非闭包闭包捕获。

如果编译器无法证明访问是安全的，则不允许访问。