本课程文字讲义中涉及到知识点和概念的使用参见视频教程，您可通过以下方式联系并加入开发者引擎社区获取：

第二课 Cadence 语言深入学习（上）

本节课与**第三课**为深入学习 Cadence 语言，主要分为三个部分：

1、深入函数，结构体，资源

2、Cadence 语言中的高级概念

3、引用时间及其他

正式的内容开始之前，先补充一个上节课关于常量的一个知识点，如下图。

```
access(all) contract HelloWorld {

    access(all) let greeting: {UInt: String}

    init() {
        self.greeting = {1: "abc", 2: "cdf"}

        self.greeting[2] = "changed"
    }

    access(all) fun hello(inx: UInt): String {
        return self.greeting[inx]?? "nonexist"
    }
}
```

合约中使用 let 定义了一个字典常量并在 init()当中初始化，然后接下来更新了这个字典中的一个 key-value 对，显然它没有报错。这时候有人就会有疑问了，常量不是只能被赋值一次吗？为什么这里可以再次被更改呢？我们可以用指针的概念来理解，字典本身的 key-value 对存储在一个地方，而我定义的 Greeting 常量它指向了字典存储的位置。所以这里常量含义指代的是这种指向关系，而真正的字典存储区域是不受影响的，是可以随意更改的。如下图，当试图再次更改指向关系时，肯定会报错。既然字典是这样，那么数组或者说结构体是不是也是这样？大家可以试一下。答案是的。

```
access(all) contract HelloWorld {

    access(all) let greeting: {UInt: String}

    init() {
        self.greeting = {1: "abc", 2: "cdf"}

        self.greeting[2] = "changed"

        self.greeting = {1: "lmn", 2: "opqrs"}
    }

    access(all) fun hello(inx: UInt): String {
        return self.greeting[inx]?? "nonexist"
    }
}
```

一、深入函数，结构体，资源

1.1、深入函数学习

上节课我们学习了函数的一些基本用法，包括函数的定义，入参，返回值等在使用时的一些注意事项，其中特别的引入了一个新的概念叫做函数标签。接下来我们继续学习函数的使用。

1、函数类型

与其他语言类似，Cadence 中函数的类型由入参的个数、类型和返回值的类型共同唯一确定，而与定义的函数名称和入参的标签无关。如下左图中 add1()和 add2()属于同一类型，同时也可以看到函数类型的表示形式：用一对小括号里包含用冒号隔开的两部分内容，第一部分是用一对小括号括起来的函数的入参，展示出了所需入参的个数和类型，第二部分声明返回值的类型。如果既没有入参也没有返回值的函数类型，声明定义如下右图。

```
let add1: ((Int, Int):Int) =
    fun(a:Int, b: Int): Int{
        return a + b
    }
let sum1 = add1(a: 100, b: 200)

let add2: ((Int, Int):Int) =
    fun(c:Int, d: Int): Int{
        return c + d
    }
let sum2 = add2(c: 100, d: 200)
```

```
let printf: (():Void) =
    fun ()  {
        log("func without param and return")
    }
printf()
```

从理论上讲，如果一个语言能支持函数类型，那么它可以做到支持函数重载。但目前 Cadence 不支持函数的重载，因此在写平时开发过程中，同一个作用域内函数名字要唯一。

2、前置和后置条件

在定义一个函数时，Cadence 语言允许用 pre 关键字来声明一块前置条件。在前置条件当中，通常对函数的入参，或者函数中用到的一些全局变量等进行验证，判断其是否符合某些条件或预期。这部分逻辑在真正执行函数时最先执行，之后才执行函数体的主逻辑。

在主逻辑执行完成得到了返回值之后，Cadence 语言允许用 post 关键字声明一块后置条件。在后置条

件当中，通常对返回值进行验证，是否结果达到预期。

此外，在 pre{}前置条件块和 post{}的后置条件块里面书写的语句必须是布尔表达式，Cadence 不允许在其中进行任何运算等操作，如下图所示：

```
access(all) contract HelloWorld {
    access(all) var balance: UInt64
    init() {
        self.balance = 0
    }
    access(all) fun transfer(to: Address, amount: UInt64): UInt64 {
        pre {
            to != nil: "Invalid Address"
            amount > 0: "amount must greater than 0"
        }

        post {
            result == before(self.balance) + amount: "transfer error"
        }

        self.balance = self.balance + amount

        return self.balance
    }
}
```

如上图，pre{}前置条件块中，逻辑表达式 to != nil 判断地址是否非空：如果为 true 则地址非空有效，程序继续执行；如果为 false 则终止函数的执行，并返回“Invalid Address”的错误提示。post{}后置条件块中与 pre{}中类似，但涉及到两个关键字，第一个是 result，代表函数的返回值，如上图定义中的 self balance 的值。第二个是 before，严格来说是一个内置函数，因为它可以接受一个参数，其功能就是获取在调用本函数之前，这个指定参数变量的值。比如在调用 transfer(0x123456, 100)之前，self.balance=100；在执行 post{}之前 self.balance 变成了 200；那么 post{}中，before(self.balance)获取的是调用函数之前的值，即 100。详情演示参见视频教程。

3、闭包函数

基本上其他语言中也都有闭包函数，特别是 js 语言中更是无处不在。在 Cadence 语言中，本文从形式和功能两个方面来看一下。

从形式上来看，定义一个函数，然后在这个函数内部又定义了一个函数，并且把这个内部定义的函数返回。那么实际上我真正使用的是内部定义的函数，这就是从形式上的看到一个闭包的样子，与其他语言差别不大。

从功能上来说，使得在调用完某个函数之后，其内部的局部变量还能被继续访问，如下示例：

```
// Declare a function named `makeCounter` which returns a function that
// each time when called, returns the next integer, starting at 1.
//
fun makeCounter(): (): Int) {
    var count = 0
    return fun (): Int {
        // NOTE: read from and assign to the non-local variable
        // `count`, which is declared in the outer function.
        //
        count = count + 1
        return count
    }
}

let test = makeCounter()
test()  // is `1`
test()  // is `2`
```

以上为官方示例，makeCounter()中定义了一个局部变量 count。如果没有闭包， 则在每次调用完 makeCounter()以函数之后， count 会被系统清理。如上通过闭包形式的处理，count 还可以继续被访问，相当于吧 count 的有效期延长了。

1.2、深入结构体学习

上节课学习了一下结构体的基础使用，如定义，初始化等。相对来说，结构体没有什么难点，这里再总结需要注意的三点内容：

1、在结构体内部，支持自定义方法，这一点有些像类的使用；

2、结构可以像简单类型那样，加上问号？构成 Optional 类型；

3、不支持集成，也不支持抽象类型等等这些复杂的面向对象的机制。唯一跟面向对象有点关系的就是接口，我们在第三课会详细介绍。

1.3、深入资源学习

上节课学习了一下资源的基础使用，定义，初始化，移动以及销毁。资源实际上就是一类特殊的结构体加上对其存在性及使用行为等方面的严格限制。每个资源有两个内置属性：uuid 可以理解为资源的唯一识别号；owner 代表这个资源属于谁。此外，涉及资源的使用，除了上节课提到的移动符号<-，还有个用于两个资源变量交换的符号<->，在本节的视频教程演示中会用到。

1、再次理解资源的移动

如下图是个资源作为入参的简单的示例：

```
access(all) fun hello(token: @Token) {
    //destroy token
}
```

当我调用此函数时，将一个资源移动到入参的位置，如果到了这个函数里面，你对这个资源不做任何处理，既不把它销毁，也不把它存起来，那么一定是会报资源丢失错误的。

再看如下与闭包结合的一个示例：

```
pub fun makeToken(token: @Token): ((): @Token) {
    return fun (): @Token {
        return <-token
    }
}

access(all) fun hello() {
    let token: @Token <- create  Token()
    let mt = self.makeToken(token: <- token)
    destroy mt()
}
```

调用 helllo()执行流程如下：

1）创建了 Token 资源移动给 makeToken()的入参；

2）makeToken()返回一个闭包赋值给 mt；

3）由于闭包的功能，调用 mt()将 makeToken()的局部常量 token 移动出来；

4）进一步处理该资源，如销毁。

看着似乎没啥问题，最多也就是 token 资源绕了一圈，但总归资源被妥善处理了，可还是报错，为什么呢？这里最关键的一点在于返回的 mt()闭包函数可以被多次调用。如果 Cadence 允许这段代码，那就有点儿相当于传进去了一个资源，却可以取出无限个资源。这是一个比较有意思的示例。

2、嵌套资源

简单来说就是资源里面定义资源，此时在外层资源一定要声明 destroy()函数，相当于析构函数，在这个析构函数里要手动把内嵌的资源销毁掉。比如一个书架资源，然后包含若干书籍资源，如下图：

```
pub resource Book {
    //...
}

pub resource BookShelf {
    pub let books: @[Book]

    init() {
        self.books <- []
    }

    destroy() {
        destroy self.books
    }
}
```

以下关于资源数组和资源字典的内容，详细使用参见视频演示教程。在演示视频中详细分析和推理了使用时的注意事项，本讲义中只列举理论知识点。

3 资源数组

1）定义声明形式：pub let books: @[Book]

2）向资源数组中添加一个资源：self.books.append(<- book)

3）从资源数组中移除一个资源：let bk <- self.books.remove(at: index); destroy bk

视频演示中还提到了一种方式：let bk <- self.books[0] <- create Book()(或者直接使用交换符号); destroy bk，理解即可实际中不建议使用，原因为此种方式严格意义上并未真正的删除。

此外，还有两点使用注意事项：不能直接遍历资源数组；不能直接的使用索引下标获取资源。

4、资源字典

1）定义声明形式：pub let books: @{String: Book}

2）向资源字典中添加一个资源：let old <- self.books.insert(key: "b3", <-book); destroy old

3）从资源字典中删除一个资源：let bk <- self.books.remove(key: key); destroy bk

视频演示中也提到了类似于资源数组的删除方式，同样理解即可，不建议使用。

本节课结束~