了解如何使用 Solidity

了解 Solidity 如何简化为 Ethereum 区块链平台对智能合约进行编程的过程。

学习目标

通过学习本模块，你将了解如何：

- 说明什么是 Solidity 以及语言功能的工作方式。
- 了解智能合同的组成部分。
- 使用 Solidity 创建基本的智能合同。

先决条件

- 了解区块链基础知识
- 了解 Ethereum 平台的相关知识
- 之前使用过 C、Python 或 JavaScript 等任何编程语言
- 基本了解编程概念
- 熟悉用于创建新指令的命令行

介绍

区块链技术的第一个主要用例是围绕加密货币，如比特币和 Ethereum。 但如果希望使用区块链来传输货币以外的数字资产，又会如何呢？ 假设使用一个供应链来处理货物的运输和交付。 或者你拥有一个在线市场，想要使用区块链技术来帮助促进产品的购买、销售和转让。

在这些示例中，你可以使用一种称为 Solidity 的编程语言来编写供应链、在线市场或其他用例的操作代码。 通过使用 Solidity，还可以指定用户操作。 通过对网络上允许的这些操作进行编程，你可以创建自己的区块链网络，这些网络对所有参与者都是安全且透明的。

在本模块中，你将了解 Solidity 语言的基础知识，并了解如何对智能合同进行编程。

什么是 Solidity

Solidity 是一种面向对象的用于编写智能合同的语言。

智能合同是存储在区块链中的程序。 它们指定有关数字资产传输的规则和行为。 可以使用 Solidity 为 Ethereum 区块链平台对智能合同进行编程。 智能合同包含状态和可编程逻辑。 智能合同通过事务执行函数。 因此，使用智能合同，可以创建业务工作流。

概述

Solidity 是 Ethereum 区块链最常用的编程语言。

Solidity 是一种基于其他编程语言（包括 C++、Python 和 JavaScript）的高级语言。 如果你熟悉这些语言中的任何一种，则应该熟悉 Solidity 代码。

Solidity 是静态类型语言，这意味着类型检查在编译时进行，而不像动态类型语言在运行时进行。 对于静态类型语言，你需要指定每个变量的类型。 例如，Python 和 JavaScript 是动态类型语言，而 C++ 是静态类型语言。

Solidity 支持继承，这意味着一个合同中存在的函数、变量和其他属性可以在另一个合同中使用。 该语言还支持复杂的用户定义类型（如 struct 和 enum），这使你可以将相关类型的数据组合在一起。

Solidity 是一种开放源代码编程语言，协作者社区越来越多。 若要了解有关 Solidity 项目以及如何参与的详细信息，请参阅 GitHub 存储库。

什么是 Ethereum?

在继续学习之前，还应熟悉 Ethereum。

Ethereum 是最受欢迎的区块链平台之一，仅次于比特币。 这是一种社区构建的技术，有自己的加密货币 Ether (ETH)，可以进行购买和销售。

Ethereum 的独特之处在于它是“全球可编程区块链”。 通过使用 Ethereum，你可以编写合同定义，也称为智能合同。 智能合同用于描述区块链参与者传输数字资产的方式。 Solidity 是用于在 Ethereum 平台上开发的主要编程语言，由 Ethereum 开发人员构建和维护。

Ethereum 虚拟机

Solidity 合同在 Ethereum 虚拟机（简称 EVM）上运行。 它是一个完全隔离的沙盒环境。 除了执行的合同外，它不会访问网络上的任何其他内容。 你现在无需对 EVM 进行详细了解，只需记住，Solidity 智能合同将部署到虚拟环境中并在虚拟环境中运行

了解语言基础知识

虽然对生产级智能合同进行编程时会涉及更多知识，但上述这些内容应能让你迈出正确的一步。

如果你了解这些概念，则可以立即开始为各种用例编写智能合同！

Pragma 指令

Pragma 是用来指示编译器检查其 Solidity 版本是否与所需版本匹配的关键字。 如果匹配，则表示源文件可以成功运行。 如果不匹配，编译器将发出错误。

请始终确保在合同定义中包含 Solidity 最新版本。 若要查找 Solidity 的当前版本，请访问 Solidity 网站。 使用源文件中的最新版本。

版本 pragma 指令如下所示：

```
pragma solidity ^0.7.0;
```

此行意味着源文件将使用高于 `0.7.0` 版本（最高版本为 `0.7.9`）的编译器进行编译。 从版本 `0.8.0` 开始，可能会引入一些中断性变更，导致源文件无法成功编译。

状态变量

状态变量是任何 Solidity 源文件的关键。 状态变量永久存储在合同存储中。

```
pragma solidity >0.7.0 <0.8.0;
contract Marketplace {
    uint price; // State variable
```

备注

合同源文件始终以定义“contract ContractName”开头。

在本例中，状态变量名为 `price`，类型为“uint”。 整数类型 uint 表示此变量是 256 位的无符号整数。 这意味着它可以存储 0 到 2(256) -1 范围内的正数。

对于所有变量定义，必须指定类型和变量名。

此外，可以将状态变量的可见性指定为：

- public：合同接口的一部分，可以从其他合同访问。
- internal：仅限从当前合同内部访问。
- 专用：仅对定义它的合同可见。

函数

在合同中，可执行代码单元成为函数。 函数描述实现一个任务的单个操作。 它们是可重用的，也可以从其他源文件（如库）进行调用。 Solidity 中函数的行为类似于其他编程语言中的函数。

下面是定义函数的一个基本示例：

```
pragma solidity >0.7.0 <0.8.0;
contract Marketplace {
    function buy() public {
        // ...
    }
}
```

这段代码显示了一个名为 `buy` 的函数，它具有公共可见性，这意味着它可以由其他合同访问。 函数可以使用以下可见性说明符之一：public、private、internal 和 external。

函数可以在内部进行调用，也可以从另一个合同外部进行调用。 函数可以接受参数并返回变量，以便在它们之间传递参数和值。

下面是一个函数示例，该函数接受一个参数（一个称为 `price` 的整数），并返回一个整数：

```
pragma solidity >0.7.0 <0.8.0;
contract Marketplace {
    function buy(uint price) public returns (uint) {
        // ...
    }
}
```

函数修饰符

函数修饰符可用于更改函数的行为。 它们的工作原理是在函数执行前检查条件。 例如，函数可以检查只有指定为卖方的用户才能列出要出售的商品。

```
pragma solidity >0.7.0 <0.8.0;
contract Marketplace {
    address public seller;
    modifier onlySeller() {
        require(
            msg.sender == seller,
            "Only seller can put an item up for sale."
        );
        _;
    }
    function listItem() public view onlySeller {
        // ...
    }
}
```

此示例介绍以下各项：

- 类型为 address 的变量，用于存储卖方用户的 20 字节 Ethereum 地址。 本模块稍后会详细介绍这些变量。
- 名为 `onlySeller` 的修饰符，用于说明只有卖方才能列出商品。
- 特殊符号 `_;`，表示函数体插入的位置。
- 使用修饰符 `onlySeller` 的函数定义。

可在函数定义中使用的其他函数修饰符包括：

- pure，用于描述不允许修改或访问状态的函数。
- view，用于描述不允许修改状态的函数。
- payable，用于描述可以接收 Ether 的函数。

事件

事件描述了合同中采取的操作。 与函数类似，事件具有在调用事件时需要指定的参数。

若要调用事件，必须将关键字“emit”与事件名称及其参数一起使用。

```
pragma solidity >0.7.0 <0.8.0;
contract Marketplace {
    event PurchasedItem(address buyer, uint price);
    function buy() public {
        // ...
        emit PurchasedItem(msg.sender, msg.value);
    }
}
```

调用事件时，事件会被捕获为事务日志中的事务，事务日志是区块链中的一种特殊数据结构。 这些日志与合同的地址相关联，已合并到区块链中，并且始终保持不变。 无法从合同中访问日志及其事件数据，并且无法修改它。

了解值类型

在本单元中，你将了解 Solidity 中的主要值类型。 值类型通过值传递，并在使用时进行复制。 编写合同时将使用的主要值类型包括“整数”、“布尔”、“string literal”、“地址”和“枚举”。

整数

每个 Solidity 源文件中都使用整数。 它们表示整数，可以有符号也可以无符号。 存储的整数大小介于 8 位到 256 位之间。

- 已签名：包括负数和正数。 可以表示为 int。
- 未签名：仅包含正数。 可以表示为 uint。

如果未指定位数，则默认值为 256 位。

以下操作可应用于整数：

- 比较：`<=`、`<`、`==`、`!=`、`=`、
- 位运算符：`& (and)`、`| (or)`、`^ (bitwise exclusive)`、`~ (bitwise`

- 算术运算符) `(addition)`、`(subtraction)`、`(multiplication)`、`/ (division)`、`% (modulo)`、`** (exponential)`

以下是整数定义的一些示例：

```
int32 price = 25; // signed 32 bit integer
uint256 balance = 1000; // unsigned 256 bit integer
balance - price; // 975
2 * price; // 50
price % 2; // 1
```

布尔型

布尔使用关键字 bool 进行定义。 它们的值始终是 `true` 或 `false`。

下面提供了定义方法：

```
bool forSale; //true if an item is for sale
bool purchased; //true if an item has been purchased
```

布尔通常用于比较语句中。 例如： 。

```
if(balance > 0 & balance > price) {
    return true;
}
if(price > balance) {
    return false;
}
```

布尔也可以用在函数参数和返回类型中。

```
function buy(int price) returns (bool success) {
    // ...
}
```

字符串文本

大多数合同文件中也使用 String literals。 它们是用双引号或单引号括起来的字符或字词。

```
    String shipped = "shipped"; // shipped
    String delivered = 'delivered'; // delivered
    String newItem = "newItem"; // newItem
```

此外，以下转义字符可以与 string literals 一起使用。

- `\<newline>` 转义为换行
- `\n` 换行
- `\r` 回车
- `\t` Tab

地址

地址是一种具有 20 字节值的类型，它表示 Ethereum 用户帐户。 此类型可以是常规“address”，也可以是“address payable”。

两者之间的区别在于，“address payable”类型是 Ether 发送到的地址，它包含额外的成员 `transfer` 和 `send`。

```
address payable public seller; // account for the seller
address payable public buyer; // account for the user
function transfer(address buyer, uint price) {
    buyer.transfer(price); // the transfer member transfers the price of the item
}
```

枚举

在 Solidity 中，可使用枚举创建用户定义类型。 之所以称之为用户定义，是因为创建合同的人员决定要包含哪些值。 枚举可用于显示许多可选择的选项，其中有一项是必需的。

例如，可以使用 enum 来表示项目的不同状态。 可以将枚举视为代表多项选择答案，其中所有值都是预定义的，你必须选择一个。 可以在合同或库定义中声明枚举。

```
enum Status {
    Pending,
    Shipped,
    Delivered
}
Status public status;
constructor() public {
    status = Status.Pending;
```

```
}
```

了解引用类型

在编写合同时，还应了解引用类型。

与总是传递值的独立副本的值类型不同，引用类型为值提供数据位置。 这三种引用类型为：结构、数组和映射。

数据位置

使用引用类型时，必须显式提供该类型的数据存储位置。 以下选项可用于指定存储类型的数据位置：

- `memory`：
  - 存储函数参数的位置
  - 生存期限制为外部函数调用的生存期
- `storage`：
  - 存储状态变量的位置
  - 生存期仅限于合同生存期
- `calldata`：
  - 存储函数参数的位置
  - 此位置对于外部函数的参数是必需的，但也可用于其他变量
  - 生存期限制为外部函数调用的生存期

引用类型总是创建数据的独立副本。

下面举例说明如何使用引用类型：

```
contract C {
  uint[] x;

  // the data location of values is memory
  function buy(uint[] memory values) public {
      x = values; // copies array to storage
      uint[] storage y = x; //data location of y is storage
      g(x); // calls g, handing over reference to x
      h(x); // calls h, and creates a temporary copy in memory
  }
  function g(uint[] storage) internal pure {}
  function h(uint[] memory) public pure {}
}
```

数组

数组是一种在集合数据结构中存储相似数据的方法。 数组可以是固定大小，也可以是动态大小。 它们的索引从 0 开始。

若要创建固定大小的 `k` 和元素类型 `T` 的数组，则需要编写 `T[k]` 。对于动态大小数组，应编写 `T[]` 。

数组元素可以是任何类型。 例如，它们可以包含“uint”、“memory”或“bytes”。 数组还可以包括“映射”或“结构”。

以下示例演示如何创建数组：

```
uint[] itemIds; // Declare a dynamically sized array called itemIds
uint[3] prices = [1, 2, 3]; // initialize a fixed size array called prices, with prices 1, 2, and 3
uint[] prices = [1, 2, 3]; // same as above
```

数组成员

以下成员既可以操作数组，又可以获取有关数组的信息：

- length：获取数组的长度。
- push()：在数组末尾追加一个元素。
- pop：从数组末尾删除元素。

下面是一些示例：

```
// Create a dynamic byte array
bytes32[] itemNames;
itemNames.push(bytes32("computer")); // adds "computer" to the array
itemNames.length; // 1
```

结构

结构是用户可以定义用来表示实际界对象的自定义类型。 结构通常用作架构或用于表示记录。

结构声明示例：

```
struct Items_Schema {
    uint256 _id;
    uint256 _price;
```

```
    string _name;
    string _description;
}
```

映射类型

映射是封装或打包在一起的键值对。 映射最接近 JavaScript 中的字典或对象。 通常使用映射来建模实际对象，并执行快速数据查找。 这些值可以包括结构等复杂类型，这使得映射类型灵活且可读性强。

下面的代码示例使用结构 `Items_Schema` ，并将 `Items_Schema` 表示的项列表保存为字典。 映射通过这种方式模拟数据库。

```
contract Items {
    uint256 item_id = 0;
    mapping(uint256 => Items_Schema) public items;
    struct Items_Schema {
      uint256 _id:
      uint256 _price:
      string _name;
    }
    function listItem(uint256 memory _price, string memory _name) public {
      items[item_id] = Items_Schema(item_id, _price, _name);
      item_id += 1;
    }
}
```

备注

映射签名 `uint256 => Items_Schema` 表示键是无符号整数类型，值是 `Items_Schema` 结构类型。

练习 - 编写第一份合同

现在，让我们将所学的内容整合成一份完整的智能合同。

在本例中，你将使用 Solidity 为简单的在线市场创建一个智能合同。 该合同将允许用户列出待售商品，并购买在售商品。 涉及两个角色：卖方和买方。

简单的市场示例

```
pragma solidity >0.7.0 <0.8.0;
contract Marketplace {
    address public seller;
```

```
    address public buyer;
    mapping (address => uint) public balances;
    event ListItem(address seller, uint price);
    event PurchasedItem(address seller, address buyer, uint price);
    enum StateType {
        ItemAvailable,
        ItemPurchased
    }
    StateType public State;
    constructor() public {
        seller = msg.sender;
        State = StateType.ItemAvailable;
    }
    function buy(address seller, address buyer, uint price) public payable {
        require(price <= balances[buyer], "Insufficient balance");
        State = StateType.ItemPurchased;
        balances[buyer] -= price;
        balances[seller] += price;
        emit PurchasedItem(seller, buyer, msg.value);
    }
}
```

让我们深入了解这份智能合同的主要组成部分：

- 分别是：
  - 三个状态变量：`buyer`、`seller` 和 `balances`
  - 两个事件：`ListItem` 和 `PurchasedItem`
  - 一个具有两个值的枚举：`ItemAvailable` 和 `ItemPurchased`
- 构造函数将卖方用户指定为 `msg.sender`，并将初始状态设置为 `ItemAvailable`。 创建合同时，将调用此构造函数。
- `buy` 函数有三个参数：`sellerbuyer` 和 `price`。 这要求买方有足够的购买能力。 然后，它将资金从买方转移到卖方，并最终发出一条消息。

后续步骤

请前往 Remix IDE 了解 Solidity 中更多的智能合同示例。 Remix 是一个浏览器内的 IDE，使你可以立即开始使用，而不必创建帐户或登录。 你可以立即编写、测试、编译和部署合同。

将此智能合同复制并粘贴到名为 `Marketplace.sol` 的新文件中的 Remix。 然后编译并部署该合同。 虽然该合同包含用于购买在售商品的函数，但你会注意到，无法提供替换货币来购买商品。 如果想挑战更高难度，可以使用所学知识编写一个函数，为买方提供余额，提供帐户地址和帐户余额。 如果要查看如何执行此操作的示例，请观看G 博士添加一个函数来初始化参与者的余额。