第3章 智能合约的开发、测试与部署

本章节将从智能合约的起源开始。前面的区块链基础知识讨论了加密、散列和点对点网络等这些成熟算法和技术是如何被创造性地应用到区块链这个去中心化的、可信的、分布式的、不可更改的账本的创新中的。

智能合约的概念早在比特币问世之前就已经存在。计算机科学家尼克·萨博详细介绍了他的加密货币比特黄金(bitcoin gold)的概念。他在1994年发表的论文*Smart Contracts*是智能合约的开山之作。实际上,萨博在20多年前就创造了“智能合约”一词。智能合约是以太坊区块链的核心和主要推动力。智能合约的设计和编码不当会导致重大故障,例如DAO Hack和Parity钱包锁定事件。通过本章的学习,操作者可以设计、编码、部署和执行智能合约。

智能合约的许多变体在区块链环境中十分普遍。Linux Foundation的Hyperledger区块链具有称为Chaincode的智能合约功能。由于以太坊是通用的主流区块链,因此本书选择讨论智能合约的以太坊实现。

3.1 什么是智能合约

智能合约是按照用户的需求编写代码,并部署和运行在以太坊虚拟机上。智能合约是数字化的,它在代码中固化了账户之间交易的规则。智能合约有利于通过原子化交易实现数字资产的转移,也可以用于存储重要数据,这些数据可以用来记录信息、事件、关系、余额,以及现实世界中的合同中需要约定的信息。智能合约类似于面向对象的class类,因此,一个合约可以调用另外一个合约,就像操作者可以在类对象之间进行互相调用和实例化一样。也可以这样认为,智能合约就是由函数构成的小程序。操作者可以新建一个合约,借助合约中的函数去查看区块链上的数据,以及按照一些规则去更新数据。

以太坊的一个重要贡献是智能合约层，该合约层支持在区块链上执行任意代码。智能合约允许用户定义复杂的操作。智能合约增强了以太坊区块链成为强大的去中心化计算系统的能力。

3.2 Remix

编写智能合约的工具有很多种，如 Visual Studio。其中，最简单、快速的开发方法是使用基于浏览器的开发工具，如 Remix。打开网页 http://remix.ethereum.org 就可以直接使用。Remix IDE(Integrated Development Environment)是一个开放源代码的 Web 和桌面应用程序。它缩短了开发周期，并具有丰富的带有直观图形用户界面(Graphical User Interface，GUI)的插件集。Remix 可以在浏览器上进行智能合约的创建、开发、部署和调试。合约维护有关的操作(如创建、发布、调试)都可以在同一个环境下完成，而不需要切换到其他的窗口或页面。Remix 除了在线版本，也可以在 github 下载软件包，经过编译，在本地使用。

本章使用 Remix 开发环境进行构建、测试智能合约，并使用 Remix 部署智能合约，通过简单的 Web 界面调用合约。在学习中，操作者必须在测试环境中尝试与智能合约相关的各种概念，以便理解和应用这些概念。

3.2.1 基础模块

通过浏览器访问地址 http://remix.zhiguxingtu.com/，可以打开 Remix 的主页面，如图 3-1 所示。其中，单击插件面板中相应的图标，则其对应的插件便显示在侧面板中；大多数(但不是全部)插件在侧面板显示其 GUI。主面板用于编辑文件，在选项卡中是可以用于 IDE 编译的插件或文件；可以在终端查看与 GUI 交互的结果，也可以在此处运行脚本。

1. 主页面入口

主页位于主面板的选项卡中。也可以通过单击插件面板顶部的徽标来访问主页面，如图 3-2 所示。

2. 插件管理器

为了使 Remix 灵活地集成其他功能，可以在插件面板中通过单击启用和关闭插件。常用的插件有 SOLIDITY COMPILER、DEPLOY & RUN TRANSACTIONS，

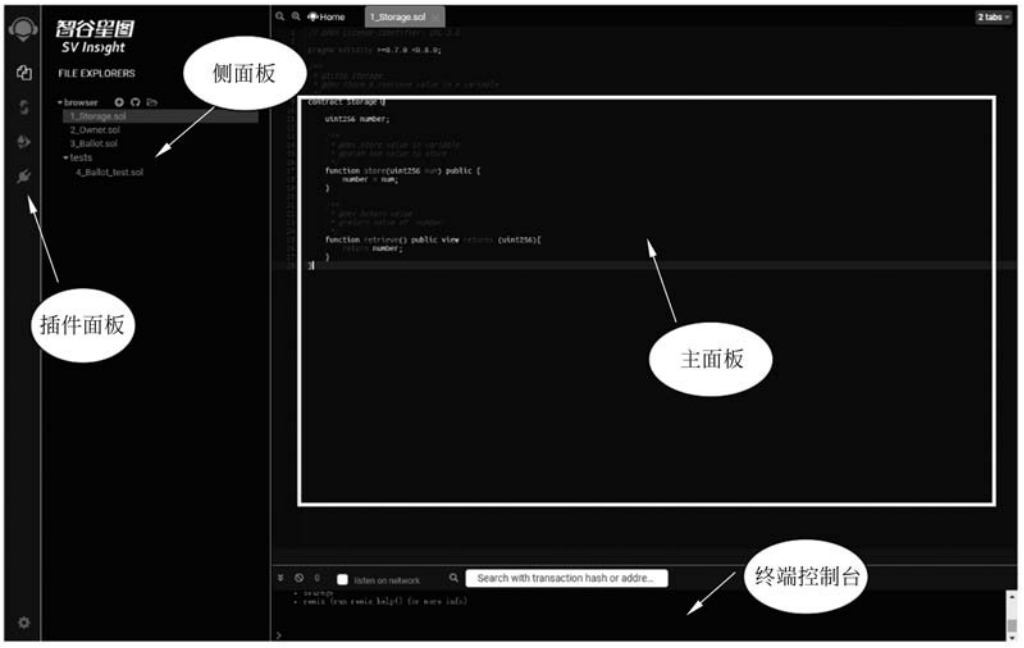


图 3-1　Remix 主页面布局

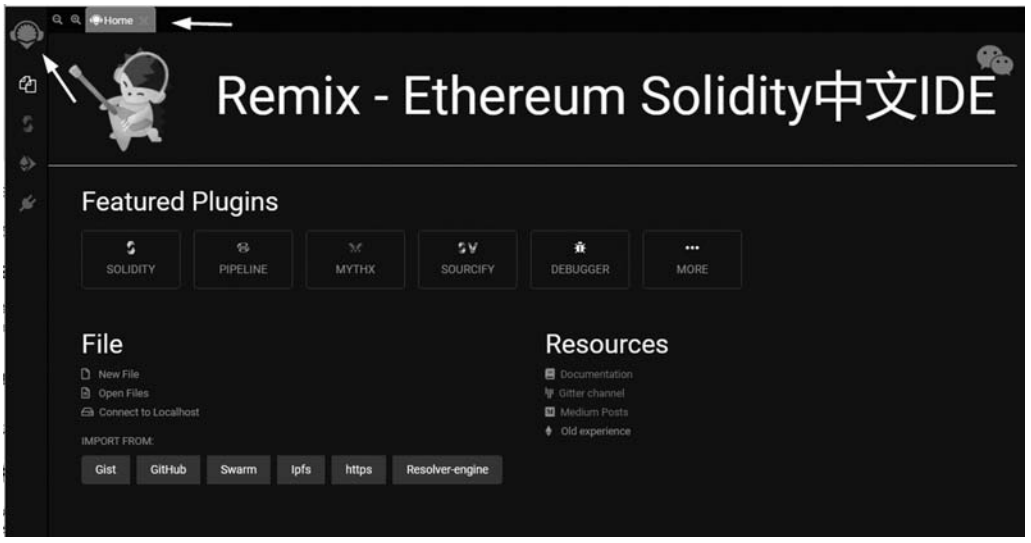


图 3-2　Remix 主页面入口

如图 3-3 所示。

3. 主题

Remix 提供了多种主题选择，可以通过插件面板下方的 选择深色主题或灰色主题，如图 3-4 所示。

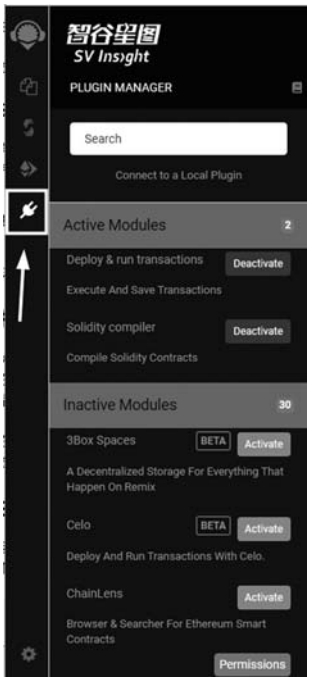
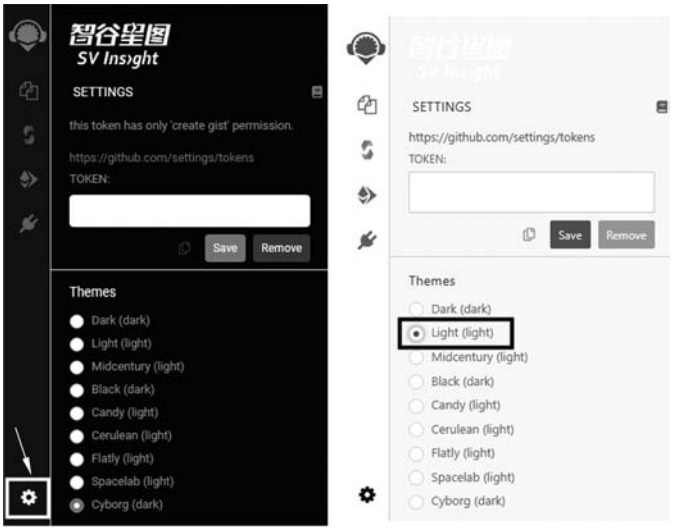


图 3-3　插件面板



(1) 深色主题　　(2) 灰色主题

图 3-4　设置 Remix 主题

4. 文件浏览器

要进入 FILE EXPLORERS 模块，单击 图标，如图 3-5 所示。

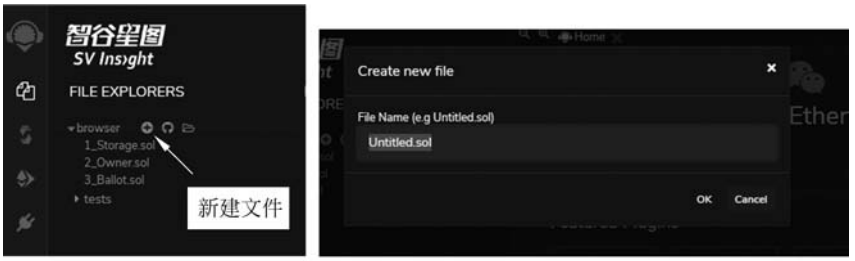
默认情况下，Remix 仅将文件存储在浏览器的本地存储(local storage)中。在文件浏览器的 browser 文件夹中包含了一个示例项目。如果打开 Remix IDE 没有看到项目示例，则可以尝试清除浏览器缓存的操作，它们就会出现。

图 3-5　激活“文资源管理器”模块

5. 建立新文件

单击新建文件图标 ，在弹出的 Create new file 对话框中输入文件名，新的文件将在编辑器中打开，如图 3-6 所示。

在创建文件时，新文件将被放置在当前选定的文件夹中。如果未选择任何内容，则这个文件将放置在文件夹的根目录中。所以要注意在创建新文件时，文件放置在了哪个文件夹中。如图 3-7 所示，右击“新文件夹”，在弹出的快捷菜单中选择 Create File，建立的新文件就放置在“新文件夹”目录中。



(1) 单击图标　　(2) 输入文件名

图 3-6　建立新文件

6. 加载本地的文件

单击图标，可以将本地计算机的文件上传到浏览器的本地存储，同时会显示在文件浏览器中，如图 3-8 所示。

图 3-7　在指定文件夹下创建新文件

图 3-8　加载本地文件至文件浏览器中

7. 右击文件

右击文件，将弹出一个上下文菜单，可以删除或者重新命名文件，如图 3-9 所示。

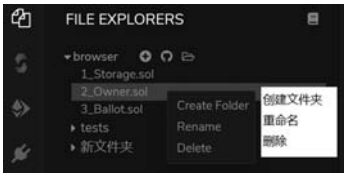


图 3-9　文件重命名和删除操作

8. Solidity 编译器

每一次修改当前文件或选择另外一个文件时，Remix 编辑器都会重新编译代码，并提供 Solidity 关键字语法的突出显示，如图 3-10 所示。

9. 终端

在终端窗口中，显示了与 Remix 交互时进行的重要操作信息。它集成了 JavaScript 和 Web3 对象，允许执行与当前上下文交互的 JavaScript 脚本。操作者可以搜索或清除终端中的日志，如图 3-11 所示。

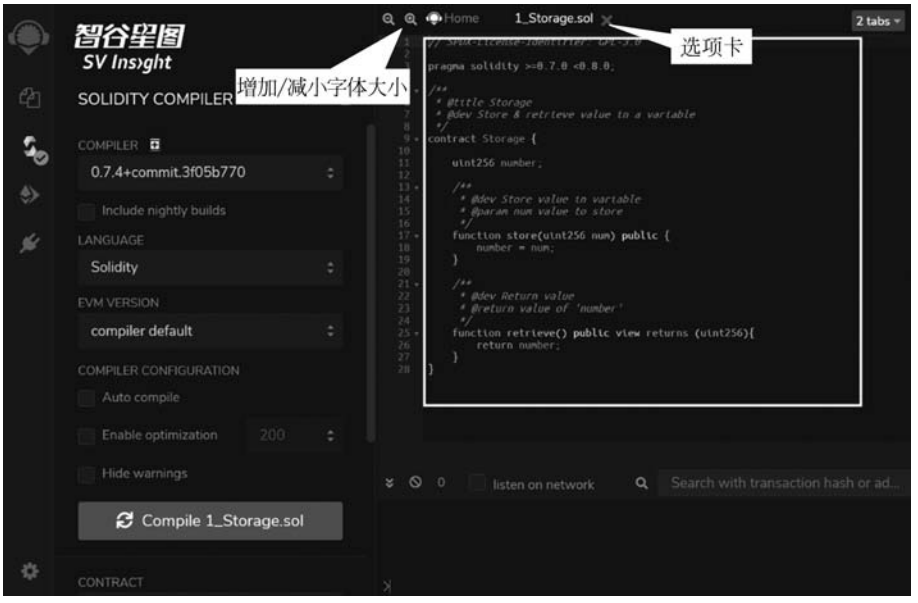


图 3-10　Solidity 关键字语法显示

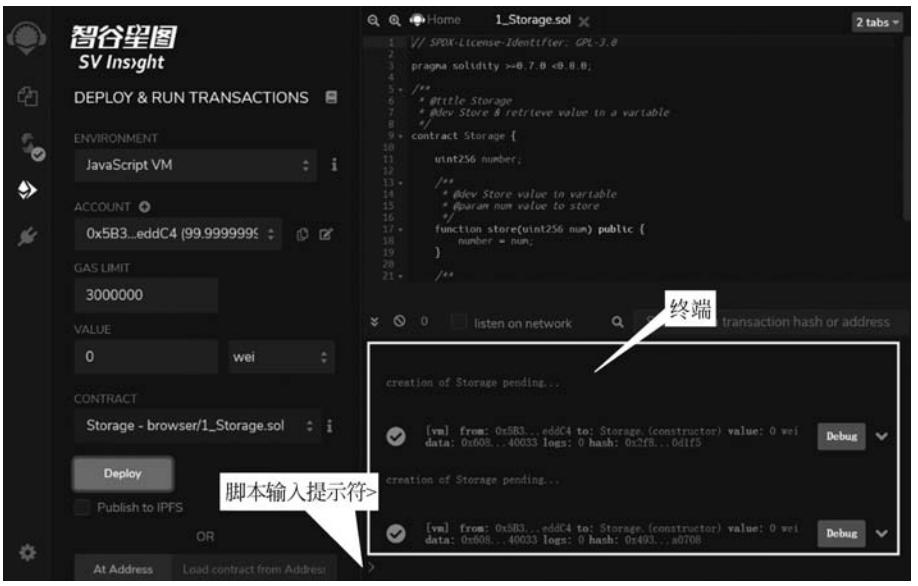


图 3-11　终端窗口

3.2.2　典型模块

1. 编译器

单击图标面板的 ，会切换到 SOLIDITY COMPILER，如图 3-12 所示。单击编

译按钮(图 3-12 中 D)会触发编译。如果希望每次修改文件保存后都对文件进行编译,请选中 Auto compile(图 3-12 中 E)。

从 Solidity 0.5.7 版本开始,Remix 可以编译 Yul 文件。操作者可以使用 LANGUAGE 下拉菜单(图 3-12 中 B)来切换语言。

Remix 允许选择不同的以太坊分叉进行编译。可以在 EVM VERSION 下拉菜单(图 3-12 中 C)选择一个特定的以太坊硬分叉,默认的版本是 compiler default。

由于一个合约文件代码中,可以包含多个合约,并且合约文件还可以导入其他的合约文件,因此通常需要编译多个合约。但是,一次只能对一个合约的编译详细信息进行检索(图 3-12 中 F)。单击 Compilation Details 按钮(图 3-12 中 G)时,将在弹出的窗口中显示当前合约的详细信息,如 BYTECODE、应用程序二进制接口(Application Binary Interface,ABI)以及 WEB3DEPLOY 等信息,如图 3-13 所示。

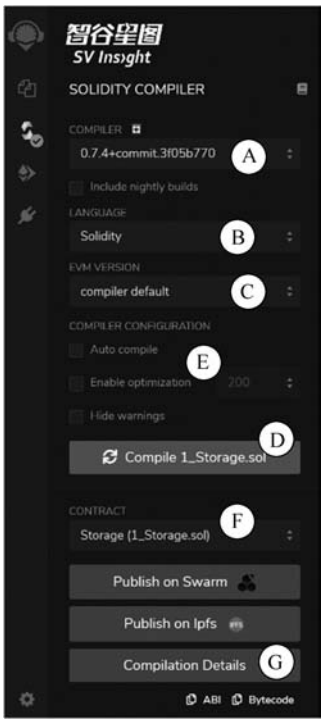


图 3-12　编译面板选项设置

图 3-13　编译后生成的 ABI 和 BYTECODE

编译后主要有两种产物,分别为 ABI 规范和合约字节码。ABI 是一个接口,由带有参数的外部函数和公共函数组成。其他使用者如果准备调用合约里面的函数,就可以使用 ABI 来实现。字节码是合约的体现形式,它运行在以太坊上面。在发布时,字节码是必需的,ABI 只有在调用合约里面的函数时才会用到。操作者可以使用 ABI

创建一个新的合约示例。

合约的发布本身就是一个交易。因此，为了发布合约，操作者需要新建一个交易。在发布时，需要提供字节码和 ABI。由于交易在运行时需要消耗 gas，这些 gas 就需要由合约提供。一旦交易被打包写到区块链上后，操作者就可以通过合约地址来使用合约了，调用方也可以通过新地址调用合约里面的函数。

图 3-14　编译错误和警告

在边栏的最下方会显示编译错误或警告等信息，如图 3-14 所示。即使编译器没有显示错误信息，解决显示的警告问题也是很重要的。

编译成功后，Remix 会为每个编译好的合同创建两个 JSON 文件。其中一个文件包含了 Solidity 编译的输出，这个文件将被命名为 contractName_metadata.json。

另一个 JSON 文件名为 contractName.json，包含了编译的工件。它包含了字节码（bytecode）、部署的字节码（deployedBytecode）、gas 预估（gasEstimates）、方法标识符（methodIdentifiers）和 ABI，如图 3-15 所示。

图 3-15　合约编译生成的工件文件

为了生成这些工件（artifacts）文件，单击 ⚙ 图标，在弹出菜单中的 General settings 部分，勾选第一个复选框，如图 3-16 所示。然后，这些元数据文件将在编译文件时生成，并被放置在 artifacts 文件夹中，在 Files Explorers 插件中可以看到。

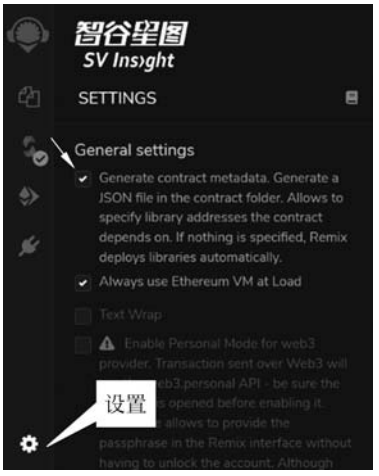


图 3-16　生成合同元数据选项

2. 部署和运行

单击图标，会切换到部署和运行交易（DEPLOY & RUN TRANSACTIONS）模块，如图 3-17 所示。该模块允许把交易发送到当前的环境（ENVIRONMENT）。

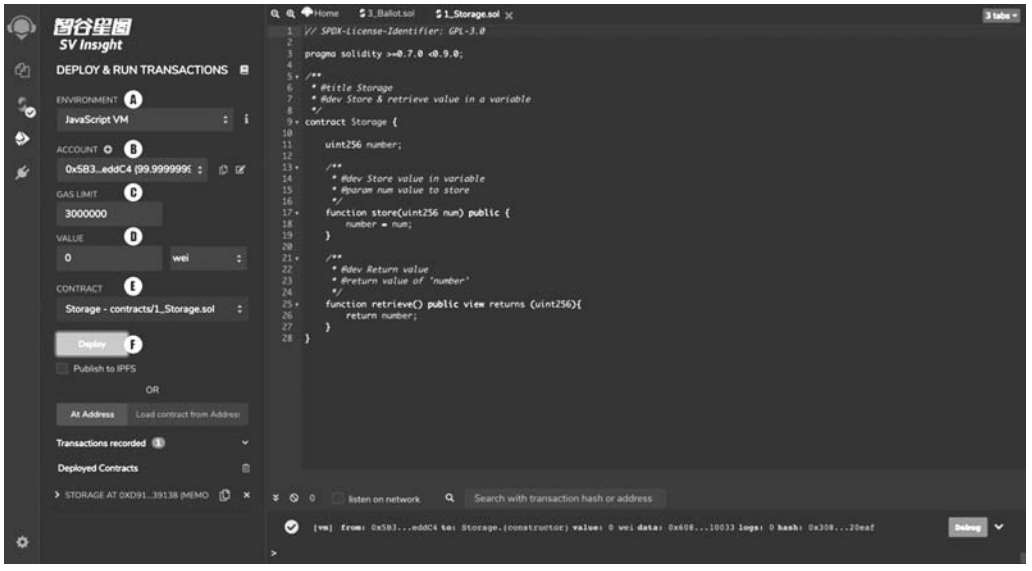


图 3-17　合约部署和执行交易面板

要使用此模块，需要先编译合约。如果在 CONTRACT 选择框中有合约名称，则可以使用；如果选择框中没有内容，则需要在中先选择一个合约文件，使其处于激活状态，转到 SOLIDITY COMPILER 进行编译，再切换到 DEPLOY & RUN TRANSACTIONS 。

(1) ENVIRONMENT 选择框(图 3-17 中 A 部分)包括三个选项。JavaScript VM 选项中所有交易将在浏览器的沙盒区块链中执行,即重新加载页面时,将启动一个新的区块链,旧的区块链将不被保存;选择 Injected Provider 选项,Remix 将连接到注入的 Web3 提供程序,MetaMask 是注入 Web3 的提供程序的示例;选择 Web3 Provider 选项,Remix 将连接到远程节点,需要将 URL 提供给选定的提供程序 geth、parity 或任何以太坊客户端。

(2) ACCOUNT 选择框(图 3-17 中 B 部分)列出与当前环境关联的账户列表(及其关联的余额)。在 JsVM 上,可以选择 5 个账户,每个账户的初始余额是 100 ether。如果将注入的 Web3 与 MetaMask 一起使用,则需要在 MetaMask 中更改账户。

(3) GAS LIMIT 选择框(图 3-17 中 C 部分)设置了在 Remix 中提交的所有交易所允许的最大 gas 量。

(4) VALUE 选择框(图 3-17 中 D 部分)设置发送到合约或 payable 功能的 eth、wei、Gwei 等的数量(注:payable 功能的按钮将显示为红色)。每次执行交易后,VALUE 值始终重置为 0。

(5) CONTRACT 选择框(图 3-17 中 E 部分)可以部署合约示例,在选择框指定了合约文件后,单击 Deploy 按钮,将部署所选的合约(这可能需要几秒钟)。需要注意的是,如果合约的构造函数(constructor)具有参数,则需要在部署时指定它们。

(6) At Address 用于访问已经部署的合约,它假定操作者给定的地址是当前合约的一个示例。Remix 不会对提供的地址是否是该合约的示例进行检查,因此使用此功能要小心,并确保操作者信任该地址的合同。

(7) Deployed Contracts 显示已经部署的合约列表,展开列表,可以看到自动生成的 UI(也称为 udapp),通过 UI 可以进行交互操作,如图 3-18 所示。单击列表左侧按钮 ,会显示合约的函数(function)按钮,如图 3-19 所示。这些按钮会根据函数功能的不同显示不同的颜色,Solidity 中的函数 view()或 pure()会显示为蓝色的按钮,此类型的交易不会改变区块的状态,只会返回合约中存储的值,且单击此类按钮,不会花费任何 gas。显示为橙色按钮的函数,会改变合约的状态,因此会产生交易成本,消耗 gas,此类函数发起的交易不接受以太币,即 VALUE 不能有值。具有 payable 功能的函数将显示为红色,此类型交易允许接受 VALUE 值,可以在 GAS LIMIT 字段下方的 VALUE 字段设置发送的 ether 数量,如图 3-20 所示。

如果函数需要参数,那么必须在输入框中输入所有的参数。输入框中的提示信息会告诉操作者每个参数的数据类型,当参数的数据类型是数字和地址时,不需要用双引号,但是字符串类型的参数需要使用双引号。多个参数之间用逗号进行分割,如图 3-21 所示。在图 3-21 的示例中,函数 store()具有 2 个参数,数据类型是 uint256 和 string。

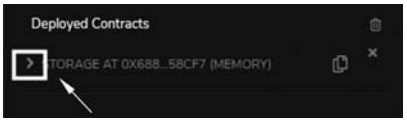


图 3-18　已经部署的合约列表

图 3-19　合约中的函数

图 3-20　设置 value 的数量及单位

图 3-21　函数参数设置方式

除了在折叠视图中输入参数，单击符号可以展开参数，这样可以一次输入一个参数，以减少折叠视图中输入参数时的混乱。

要将数组或结构（struct）作为参数传递，需要将其放入“[]”中，并且需要在该 Solidity 文件的顶部，添加语句“pragma experimental ABIEncoderV2”，合约的代码示例如下：

```
pragma solidity >= 0.5.0 < 0.7.4;
pragma experimental ABIEncoderV2;

contract Sunshine {
    struct Garden {
      uint slugCount;
      uint wormCount;
      Flower[] theFlowers;
    }
    struct Flower {
        uint flowerNum;
        string color;
    }
    Flower public _flower;

    function picker(Garden memory gardenPlot) public {
        _flower = gardenPlot.theFlowers[0];
        uint a = gardenPlot.slugCount;
        uint b = gardenPlot.wormCount;
        Flower[] memory cFlowers = gardenPlot.theFlowers;
        uint d = gardenPlot.theFlowers[0].flowerNum;
```

```
        string memory e = gardenPlot.theFlowers[0].color;
    }
    function getFlower() public view returns ( Flower memory){
        return _flower;
    }
}
```

部署合约并打开部署示例后，可以将 [1,2,[[3,"Petunia"]]] 作为参数进行传递。函数 picker()接收一个 Garden 类型的结构体。该结构用方括号包裹，参数内又嵌套了一个数据类型为 Flower 结构的数组[3,"Petunia"]，如图 3-22 所示。

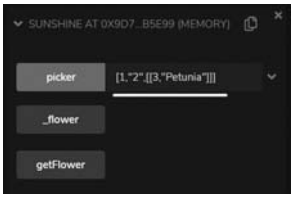


图 3-22　函数参数为数组或结构时的填写方式

3. 调试器

Remix 调试器通过帮助操作者观察合约执行时的运行时行为来定位问题。它工作在 Solidity 及其生成的合约字节码中。可以暂停合约执行以检查合约代码、状态变量、局部变量和堆栈变量，并查看从合约代码生成的 EVM 指令。

调试器在单步执行交易时会显示合约的状态。在 Remix 中提交交易以后，或者通过制定之前的交易地址来使用调试功能。要启动调试会话，需要执行以下操作之一：无论成功与否，当提交的交易出现在终端窗口时，可以单击 Debug 按钮，调试器将在面板中被激活，如图 3-23 所示；在插件管理器中单击 ，在交易哈希输入框中输入已经部署的交易地址，然后单击 Stop debugging 按钮，如图 3-24 所示。

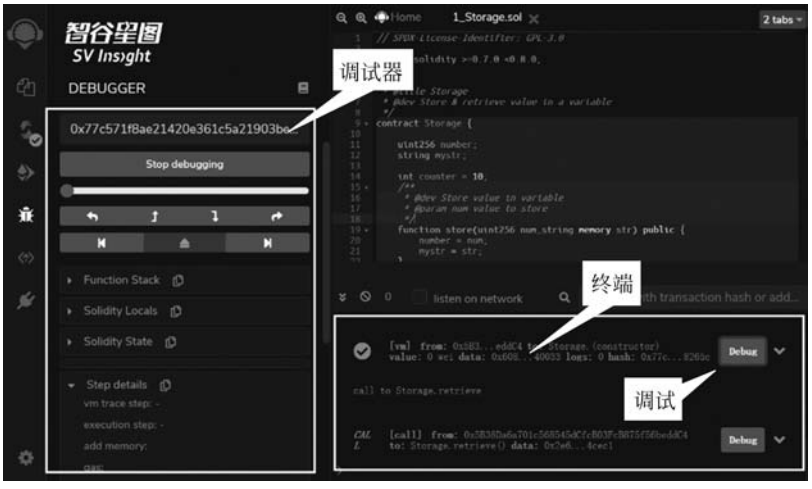


图 3-23　调试面板

图 3-24　通过交易地址调试的方式

调试器将在编辑器中突出显示相关的合约代码。如果要停止调试，请单击按钮 Stop debugging。

1）调试器导航

调试面板顶部是调试器的导航功能，如图 3-25 所示。

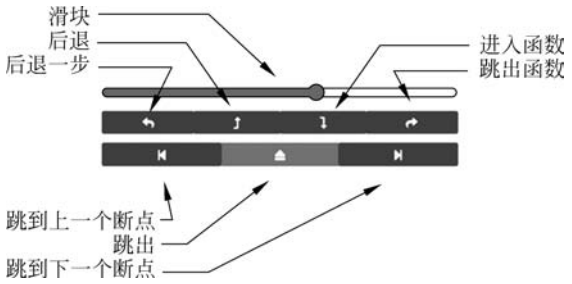


图 3-25　调试器的导航功能

（1）拖动滑块(Slider)时，会同步在代码编辑器中突出显示相关的合约代码。同时交易的操作码也会同步滚动。每个操作码的交易状态也会同步发生变化，这些变化会反映在调试器的面板中，如图 3-26 所示。

（2）后退一步(Step over back)。单击 按钮将转到上一个操作码。如果上一步是调用其他函数，则不会进入被调用的函数内。

（3）后退(Step back)。单击 按钮，返回上一个操作码。

（4）进入函数(Step into)。单击 按钮，将定位到下一个操作码。如果该操作是调用一个函数，则会进入该函数。

（5）跳出函数(Step over forward)。单击 按钮，也将定位到下一个操作码。如果该操作是调用一个函数，则不会进入该函数。但是被调用函数会被执行。

（6）跳到上一个断点(Jump to prev breakpoint)。单击 按钮，滑块会移动至当

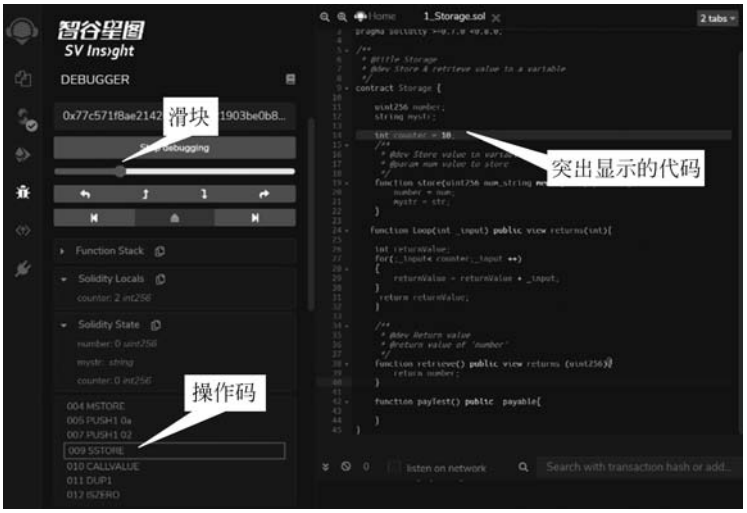


图 3-26　滑块

前位置最近的上一个断点设置处。如图 3-26 所示，假如当前调试的操作码在 23 行，则单击 后，将定位到 20 行。

（7）跳出(Jump out)。在函数调用过程中，单击 按钮，将结束此次调用。

（8）跳到下一个断点(Jump to next breakpoint)。单击 按钮，滑块会移动至当前位置最近的下一个断点设置处。

调试的一个重要方面是在感兴趣的代码行停止执行，断点有助于做到这点，在编辑器中单击行号，可以设置断点，如图 3-27 所示。再次单击将删除断点。这样在执行函数期间，当到达此行时，执行会被暂停。

图 3-27　断点的设置

如果将断点设置在声明变量的行中，则可能会触发两次：第一次将变量初始化为零；第二次为变量分配实际值。

2）调试器面板

调试器面板包括以下几类。

（1）函数堆栈(Function Stack)面板列出正与交易交互的函数，如图 3-28 所示。

图 3-28　函数堆栈示例

(2) 本地变量(Solidity State)面板列出函数的局部变量,如图 3-29 所示。

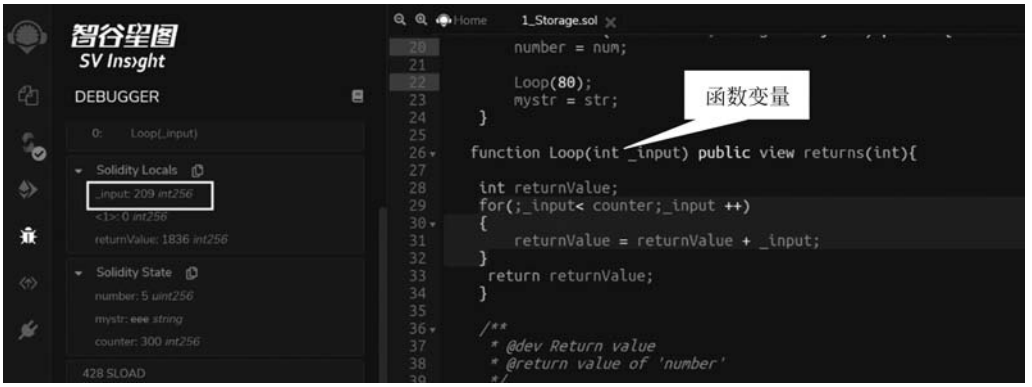


图 3-29　函数局部变量示例

(3) 状态变量(Solidity State)面板显示合约的状态变量。以太坊拥有一个保存代码和数据的存储器,使用区块链跟踪这个存储器随着时间的变化。就像通用目的存储程序计算机一样,以太坊可以把代码加载进状态机,然后运行这些代码,并把状态转换的结果保存在区块链上,如图 3-30 所示。

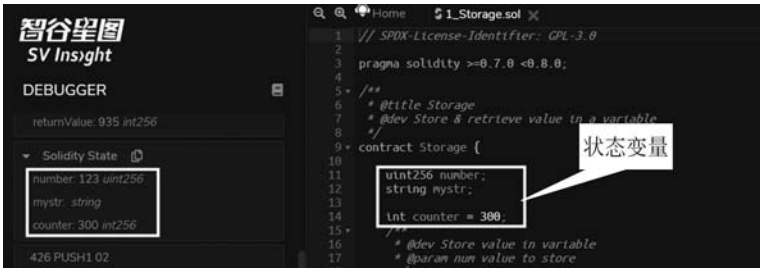


图 3-30　函数状态变量示例

(4) 操作码面板显示步骤序号和调试器当前的操作码，如图 3-31 所示。操作码可以分为算术操作、栈操作、处理流程操作、系统操作、逻辑操作、环境操作和区块操作。

图 3-31　操作码面板示例

常见的算术操作包括 ADD(对栈顶的两个条目进行加法)、MUL(对栈顶的两个条目进行乘法)、SUB(对栈顶的两个条目进行减法)、DIV(整数除法)、SDIV(带符号的整数除法)、MOD(模运算)、SMOD(带符号的模运算)、ADDMOD(先做加法然后进行模运算)、MULMOD(先做乘法然后进行模运算)、EXP(乘方运算)、SIGNEXTEND(符号扩展操作)、SHA3(对内存中的一段数据进行 Keccak-256 哈希运算)。

常见的栈操作包括 POP(移除栈顶的一个条目)、MLOAD(从内存中加载一个字)、MSTORE(向内存中保存一个字)、MSTORE8(向内存中保存一个字节)、SLOAD(从存储中加载一个字)、SSTORE(向存储中保存一个字)、MSIZE(获得当前已分配内存的字节数大小)、PUSHx(将 x 字节的一个条目放到栈顶，x 可以是 1～32 的整数)、DUPx(复制栈顶的第 x 个条目到栈顶，x 可以是 1～16 的整数)、SWAPx(交换栈顶条目和第 x+1 个栈内条目，x 可以是 1～16 的整数)。

常见的处理流程操作包括 STOP(停止执行)、JUMP(将程序计数器设置为任意数值)、JUMPI(基于条件修改程序计数器的值)、PC(取得程序计数器的数值)、JUMPDEST(标记一个有效的跳转地址)。

常见的系统操作包括 LOGx(增加一条带有 x 个主题的日志数据，x 值可以是 0～4 的整数)、CREATE(用关联代码创建一个新账户)、CALL(向另一个账户发起消息调用，也就是运行另一个账户的代码)、CALLCODE(用另一个账户的代码向当前账户发起消息调用)、RETURN(停止执行并返回输出数据)、DELEGATECALL(用其他账户的代码向当前账户发起消息调用，但 sender 和 value 的数值保持不变)、STATICCALL(向一个账户发起静态消息调用)、REVERT(停止执行并撤销状态修改，但保持返回数据和剩余 gas)、INVALID(预设的无效指令)、SELFDESTRUCT(停止执行，并将当前账户标记为自毁账户)。

常见的逻辑操作包括 LT(小于比较操作)、GT(大于比较操作)、SLT(有符号小于比较操作)、SGT(有符号大于比较操作)、EQ(等于比较操作)、ISZERO(简单的非操作)、AND(按位与操作)、OR(按位或操作)、XOR(按位异或操作)、NOT(按位非操作)、BYTE(从一个字中取得一个字节数据)。

常见的环境操作包括 GAS(取得可用 gas 的数量，减去这个指令的消耗)、ADDRESS

(取得当前账户的地址)、BALANCE(取得指定账户的余额)、ORIGIN(取得触发这次EVM执行的EOA地址)、CALLER(取得当前执行的调用者地址)、CALLVALUE(取得当前执行的调用者所发送的以太币数量)、CALLDATALOAD(取得当前执行的输入数据)、CALLDATASIZE(取得当前输入数据的字节大小)、CALLDATACOPY(将当前输入数据复制到内存中)、CODESIZE(当前环境运行的代码的字节大小)、CODECOPY(将当前环境运行的代码复制到内存中)、GASPRICE(取得由初始交易所制定的gas价格)、EXTCODESIZE(取得任意账户代码的字节大小)、EXTCODECOPY(将任意账户的代码复制到内存中)、RETURNDATASIZE(取得在当前环境中的前一次调用的输出数据字节大小)、RETURNDATACOPY(将前一次调用的输出数据复制到内存中)。

常见的区块操作包括BLOCKHASH(取得最新的256个完整区块中某个区块的哈希)、COINBASE(取得当前区块的区块奖励受益人地址)、TIMESTAMP(取得当前区块的时间戳)、NUMBER(取得当前区块的区块号)、DIFFICULTY(取得当前区块的难度)、GASLIMIT(取得当前区块的gas上限)。

(5) 堆栈(Stack)面板显示EVM堆栈,如图3-32所示。EVM有一个基于堆栈的架构,在一个栈中保存了所有内存数值。EVM的数据处理单位被定义为256位的字,并且它还具有以下数据组件:一个不可变的程序代码存储区ROM(Read-Only Memory),其加载了要执行的智能合约字节码;一个内容可变的内存,被严格地初始化为全0;一个永久的存储,其作为以太坊状态的一部分存在,也会被初始化为全0。

图3-32 EVM堆栈面板示例

(6) 内存(Memory)是一个与栈共同存在的、独立的临时存储空间。每个新的消息调用都会清除内存。内存是线性的,可以在字节级别进行寻址。读取限制为256位,而写入则可以为8位或256位。内存面板由3列组成,第一列是内存中的位置;第二列是十六进制编码值;第三列是解码值。如果什么都没有,则显示"?",如图3-33所示。为了更好地显示数据,可以向右拖动主面板和侧面板之间的边框,使Remix的侧面板更宽一些。

(7) 存储(Storage[Completely Loaded])面板显示持久性存储,如图3-34所示。状态变量按照它们在合约中定义的顺序保存在一系列的存储槽中,每一个存储槽都有32字节。

图 3-33　Memory 示例

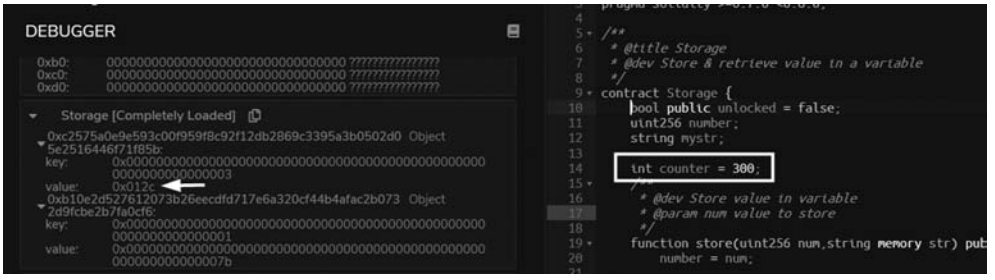


图 3-34　Storage[Completely Loaded]示例

(8) 调用堆栈(Call Stack),所有的计算都是在一个叫作调用堆栈的数据数组上进行的。它的最大大小为 1024 个元素,包含 256 位的字,如图 3-35 所示。

(9) 调用数据(Call Data)包含函数参数,如图 3-36 所示。

图 3-35　Call Stack 示例

图 3-36　Call Data 示例

(10) 返回值(Return Value)显示函数的返回值,只有当运行到 RETURN 操作时才显示,如图 3-37 所示。

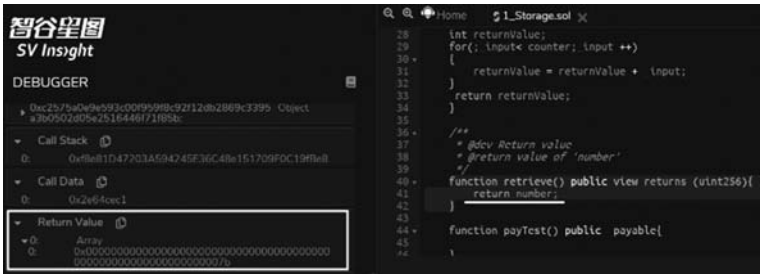


图 3-37　Return Value 示例

(11) 完整的存储变化(Full Storage Changes),函数结束时才显示所有修改后的合约存储值,如图 3-38 所示。

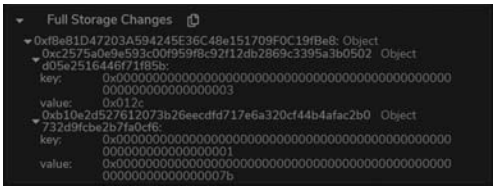


图 3-38　Full Storage Changes 示例

3.2.3　单元测试

单击图标栏的图标，将打开 SOLIDITY UNIT TESTING 面板。如果以前从未使用过此插件，没有看到此图标，则必须从 Remix 插件管理器中将其激活，如图 3-39 所示。

成功加载后，插件如图 3-40 所示。

图 3-39　激活单元测试插件

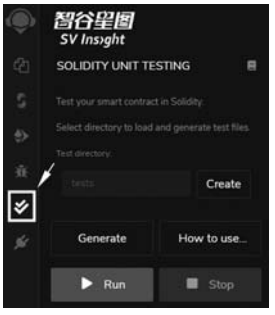


图 3-40　单元测试面板

1. 测试目录

插件需要提供一个目录，可以在输入框中输入目录名，然后单击 Create 按钮创建该目录。也可以单击 选择目录，如图 3-41 所示。选择后，此目录将用于加载测试文件和存储新生成的测试文件。

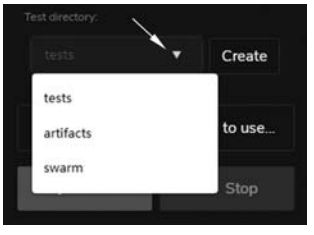


图 3-41　选择测试目录

2. 生成测试文件

选择要进行测试的合约文件，然后单击 Generate 按钮。它将在测试目录中生成一个专门用于该合约的测试文件。如果未选择任何合约文件，单击 Generate 按钮后将创建一个名为 newFile_test.sol 的测试文件，如图 3-42 所示。该文件包含足够的信息，可以更好地了解合约单元测试的方法。

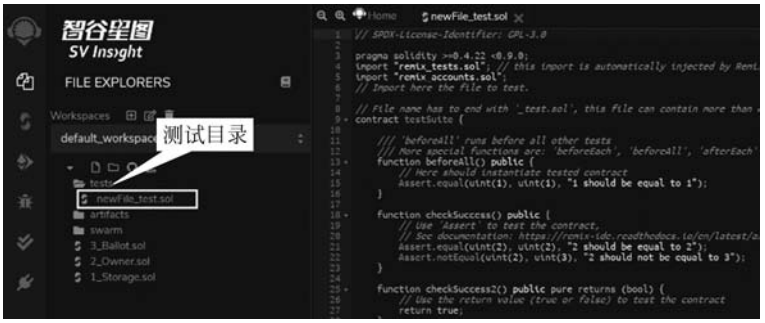


图 3-42　生成测试文件

newFile_test.sol 文件如下所示：

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.22 <0.9.0;
import "remix_tests.sol"; // This import is automatically injected by Remix.
import "remix_accounts.sol";
// Import here the file to test.

// File name has to end with '_test.sol', this file can contain more than one testSuite contracts.
contract testSuite {

    /// 'beforeAll' runs before all other tests.
    /// More special functions are: 'beforeEach', 'beforeAll', 'afterEach' & 'afterAll'.
    function beforeAll() public {
        // Here should instantiate tested contract.
        Assert.equal(uint(1), uint(1), "1 should be equal to 1");
    }

    function checkSuccess() public {
        // Use 'Assert' to test the contract.
    /* See documentation:
    https://remix-ide.readthedocs.io/en/latest/assert_library.html
     */
        Assert.equal(uint(2), uint(2), "2 should be equal to 2");
        Assert.notEqual(uint(2), uint(3), "2 should not be equal to 3");
    }

    function checkSuccess2() public pure returns (bool) {
        // Use the return value (true or false) to test the contract
        return true;
    }
        function checkFailure() public {
        Assert.equal(uint(1), uint(2), "1 is not equal to 2");
    }
```

```
    // Custom Transaction Context.
/* See more:
https://remix-ide.readthedocs.io/en/latest/unittesting.html#customization
*/
    /// #sender: account-1
    /// #value: 100
    function checkSenderAndValue() public payable {
        // Account index varies 0-9, value is in wei
        Assert.equal(msg.sender, TestsAccounts.getAccount(1), "Invalid sender");
        Assert.equal(msg.value, 100, "Invalid value");
    }
}
```

3. 编写测试

编写足够的单元测试文件，以确保合约在不同情况下能够按预期工作。Remix 注入了一个可用于测试的内置库（参见 https://remix-ide.readthedocs.io/en/latest/assert_library.html）。为使测试更具结构性，测试合约文件中定义了 4 种特殊功能：beforeEach()——每次测试前运行；beforeAll()——在所有测试之前运行；afterEach()——每次测试后运行；afterAll()——在所有测试后运行。

4. 运行测试

完成测试文件的编写后，选择文件并单击 Run 按钮执行测试。测试将在独立的环境中执行，在完成测试后，将显示测试结果的摘要信息，如图 3-43 所示。

图 3-43　测试结果的摘要信息

对于失败的测试，将提供更详细的信息来分析问题。单击失败的测试摘要信息将在编辑器中突出显示相关的代码行，如图 3-44 所示。

图 3-44　查看测试失败的相关代码

5. 停止测试

如果想要停止测试的执行，单击 Stop 按钮。

6. 自定义设置

Remix 可以设置各种定制条件，以正确测试合约。

（1）自定义编译器上下文：在运行测试之前，可以在编译器插件面板 COMPILE 下拉框选择不同的 EVM Solidity 版本，也可以同时启用优化进行配置，如图 3-45 所示。

图 3-45　合约编译的自定义设置

（2）自定义交易上下文：为了与合约的方法进行交互，交易的主要参数来自账户地址（address）、ether 值和 gas。可以通过对这些参数设置不同值，自定义测试方法的行为。可以使用 NatSpec 注释为 msg.sender 和 msg.value 设置交易自定义的值，例如：

```
// #sender: account-0
// #value: 10
function checkSenderIs0AndValueis10 () public payable {
    Assert.equal(msg.sender, TestsAccounts.getAccount(0), "wrong sender in
checkSenderIs0AndValueis10");
    Assert.equal(msg.value, 10, "wrong value in checkSenderIs0AndValueis10");
}
```

使用说明：

- 必须在 function 的 NatSpec 中定义参数。
- 每个参数使用前缀“#”和冒号“:”结束。如#sender:和#value:。

- 目前，自定义仅适用于参数 sender 和 value。
- msg. sender 是合约方法内部访问的交易的地址。应该以固定格式 account-<account_index>定义，例如 account-0。
- remix_accounts. sol 必须导入到测试文件中才能使用自定义 # sender。
- value 与交易一起发送，在交易中 wei 使用 msg. value 合约方法进行访问。它是一个数字类型。

7. 断言库

(1) Assert. ok(value[,message])。其中，value：<bool>；message：<string>。测试 value 是否为真，如果失败则返回消息(message)。示例代码如下：

```
Assert.ok(true);
// OK
Assert.ok(false, "it\'s false");
// Error: it's false
```

(2) Assert. equal(actual,expected[,message])。其中，actual：<uint|int|bool|address|bytes32|string>；expected：<uint|int|bool|address|bytes32|string>；message：<string>。

测试实际值(actual)和预期值(expected)是否相同，失败时返回信息(message)。示例代码如下：

```
Assert.equal(string("a"), "a");
// OK
Assert.equal(uint(100), 100);
// OK
foo.set(200)
Assert.equal(foo.get(), 200);
// OK
Assert.equal(foo.get(), 100, "value should be 200");
// Error: value should be 200
```

(3) Assert. notEqual(actual,expected[,message])。其中，actual：<uint|int|bool|address|bytes32|string>；expected：<uint|int|bool|address|bytes32|string>；message：<string>。

测试实际值(actual)和预期值(expected)是否不一致，失败时返回信息(message)。示例代码如下：

```
Assert.notEqual(string("a"), "b");
// OK
foo.set(200)
Assert.notEqual(foo.get(), 200, "value should not be 200");
// Error: value should not be 200
```

(4) Assert. greaterThan(value1, value2[, message])。其中,value1:<uint|int>;value2:<uint|int>; message:<string>。

测试 value1 是否大于 value2,失败时返回消息(message)。示例代码如下:

```
Assert.greaterThan(uint(2), uint(1));
// OK
Assert.greaterThan(uint(-2), uint(1));
// OK
Assert.greaterThan(int(2), int(1));
// OK
Assert.greaterThan(int(-2), int(-1), "-2 is not greater than -1");
// Error: -2 is not greater than -1
```

(5) Assert.lesserThan(value1,value2[,message])。其中,value1:<uint|int>;value2:<uint|int>; message:<string>。

测试 value1 是否小于 value2,失败时返回消息(message)。示例代码如下:

```
Assert.lesserThan(int(-2), int(-1));
// OK
Assert.lesserThan(int(2), int(1), "2 is not lesser than 1");
// Error: 2 is not lesser than 1
```