使用 Solidity 编写 Ethereum 智能合约

> *内容出自**C2N数字游民部落***
>
> 加入社区获得更多材料：https://discord.gg/R2qW7yqg

了解如何安装和使用可用于开发智能合约的工具。

学习目标

学完本模块后，你将能够：

- 解释智能合约是什么。
- 了解智能合约的常见用例。
- 安装 Hardhat。
- 安装 Nomic Foundation 的 solidity VS Code 扩展。
- 使用 Hardhat 扩展编写智能合约。
- 使用 hardhat 测试智能合约。

先决条件

- 了解区块链的基本知识
- 了解 Ethereum 平台的相关知识
- 熟悉 Solidity 编程语言
- 已安装 Visual Studio Code
- 了解如何从 Internet 下载程序
- 了解如何使用命令行工具

介绍

随着区块链的发展，智能合约的使用愈加广泛。 在此区块链阶段，许多用例都以生成智能合约和业务应用程序为中心。 我们开始在近乎各行各业中看到真实的区块链应用程序，这一时刻令人心潮澎湃。

区块链吸引了许多开发人员，他们都想要创建自己的应用程序。 这些开发人员需要了解如何使用 Solidity 创建和测试智能合约。 他们还需要了解可用于管理区块链应用程序的优秀免费开发工具。

智能合约是什么？

智能合约是区块链中存储的一种程序。 智能合约将区块链从数据扩展到代码。 它们表示各个参与方之间的协议。 它们是编码协议，当操作发生时，代码将运行并提供响应。

此类合约的所有条款和条件都以编程方式定义。 定义将为区块链的参与者指定规则、要求和奖励。 它还会指定如何在各参与方之间传输数字资产。 每个智能合约将分配有一个 20 个字节的地址，用作合约的唯一标识。

智能合约将自行运行，发送事件来触发状态转换，并调用函数。 它们非常适用于区块链技术，因为素不相识的人们可以通过它们按照指定的安全方式开展业务，而不需要任何中间商。

智能合约通常与 Ethereum 一起使用。 Ethereum 是全球首个可编程的区块链。 通过它，可以定义智能合约，来帮助传输数字资产，例如以太币。

用于编写合约的语言是 Solidity。 Solidity 是图灵完备语言，这意味着可以通过明确的定义和编码编写复杂的合约。

由于每个状态转换都会进行记录并且都不可变，因此在将合约发布到生产环境之前，应对它进行全面测试。 Bug 修复可能会产生巨大成本，甚至会严重损害系统。

智能合约的主要属性和优点如下：

- 透明：区块链用户可以读取智能合约，并可以使用 API 来访问这些合约。
- 不可变性：智能合约的执行将创建不可更改的日志。
- 分发：该合约的输出由该网络的节点验证。 合约状态可以公开显示。 在某些情况下，甚至可以看到“私有”变量。

用例

智能合约可为许多行业和流程带来好处。 请考虑以下用例。

保险：当发生某些事件时，智能合约可自动触发索赔，从而简化索赔过程。 然后，若要确定用户将收到的补偿金额，可在区块链中记录索赔详细信息。 此功能可以减少处理时间和人为错误。

投票：智能合约有助于投票自动化和透明化。 每个合约都是一种投票，代表着投票者的身份。 由于区块链是不可变的，这意味着区块链无法更改，因此无法篡改投票。

供应链：随着物品沿着供应链移动，智能合约可记录所有权，并可确认在任何给定时间由谁负责某项产品。 在任何阶段，都可以通过智能合约准确查明产品应处于的位置。 如果供应链中的任何一个参与方未能按时送达，其他每个参与方都会知道何处出现了问题。

记录保留：许多行业都可以使用智能合约来提高记录保留的速度和安全性。 可以使用区块链技术将记录数字化，并对其进行安全的加密和存储。 此外，可以限制访问，以便只有允许的人员才能访问记录。

财产所有权：智能合约可以记录财产的所有者。 通过它们可以快速高效地记录所有权。 智能合约还有助于及时安全地转移所有权。

用于处理智能合约的工具

许多工具都可帮助你快速高效地开发智能合约。 以下各节介绍了一些可以了解的集成开发环境 (IDE)、扩展和框架。

IDE

- Visual Studio Code：此代码编辑器已经过重新定义和优化，专门用于生成和调试现代 Web 和云应用程序。 在此模块中，我们将使用 Visual Studio Code 完成所有练习。
- Remix：一种基于浏览器的编译器和 IDE，可以通过它来使用 Solidity 生成 Ethereum 合约，并使用它调试交易。 Remix 是探索合约示例的好途径。 你可以使用它来编写、测试和部署自己的合约。 在此模块中，我们不会使用 Remix，但你可以在合约练习中自行体验它。

扩展

- solidity VS Code 扩展：此扩展可以简化在 Ethereum 账本上创建、生成和部署智能合约的方式。集成了对solidity的支持和对hardhat的支持 。 在此模块中，我们将使用此扩展编写和测试智能合约。

框架

- Hardhat：Hardhat是一个编译、部署、测试和调试以太坊应用的开发环境。它可以帮助开发人员管理和自动化构建智能合约和dApps过程中固有的重复性任务，并围绕这一工作流程轻松引入更多功能。这意味着hardhat在最核心的地方是编译、运行和测试智能合约
- OpenZeppelin：使用 OpenZeppelin 工具编写、部署和操作去中心化应用程序。 OpenZeppelin 提供了两个产品：合约库和 SDK。 在此模块中，我们不会使用 OpenZeppelin，但稍后你可以在编写安全区块链应用程序时体验它。

练习 - 安装 Hardhat

Hardhat 是最常用的 Ethereum 开发环境和测试框架。 可以使用 Node 包管理器 (NPM) 安装它。

关于 Hardhat

Hardhat 具有以下好处：

- 可生成、编译、部署和测试智能合约
- 可管理网络，提供了本地网络，也可以只部署到公共和专用网络
- 可管理项目依赖项的包
- 具有交互式控制台，可直接进行合约通信和管理

安装 Hardhat

可以使用节点包管理器安装 Hardhat。 在终端中键入以下内容：

```
npm install --save-dev hardhat
```

有关 Hardhat 入门的详细信息，请参阅 Hardhat 快速入门。

网络支持

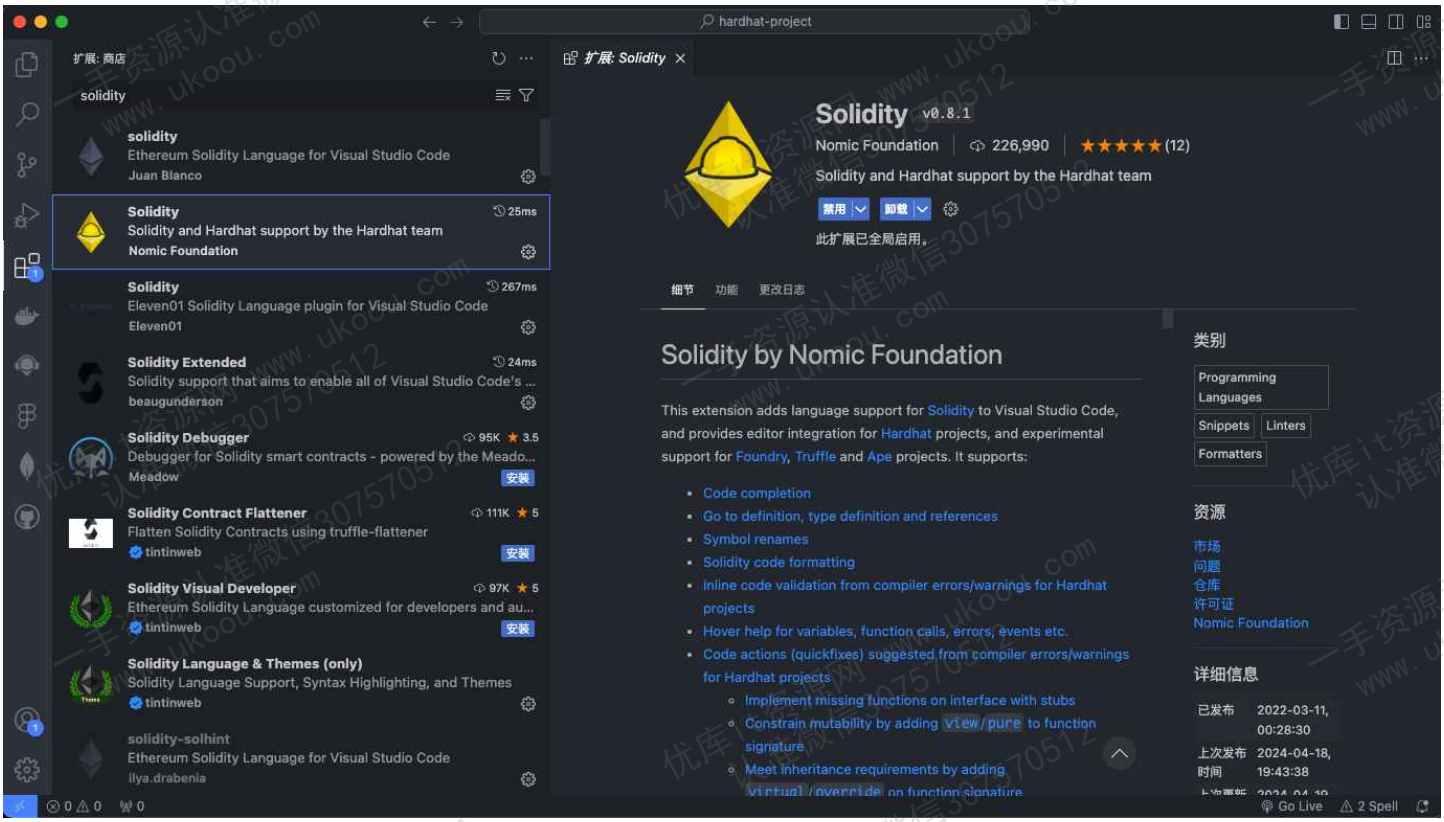
Hardhat 还内置了 Hardhat Network，这是一个专为开发而设计的本地以太坊网络。它允许您部署合约、运行测试和调试代码。同时也支持将合约部署到不同的外部的evm网络上；

练习 - 安装 Solidity VS Code 扩展

Solidity VSCode 扩展：此扩展向 Visual Studio Code 添加了对 Solidity 的语言支持，并为 Hardhat 项目提供了编辑器集成，以及对 Foundry、Truffle 和 Ape 项目的实验性支持

安装 solidity扩展

在 Visual Studio Code 的左侧边栏中，选择“扩展”。 搜索“solidity”，然后选择Nomic 基金会的 Solidity插件进行安装。



在使用 Hardhat 扩展之前，请确保已安装：

- Node.js 和 npm：若要确认已安装 Node.js，请打开终端并键入 `node` 。 如果安装了 Node.js，终端将返回计算机上的 Node.js 的版本。 也可通过在终端中键入 `npm` 来确认是否安装了 Node 包管理器 (NPM)。
- Git：若要确认已安装 Git，请打开终端并键入 `git` 。 如果安装了 Git，终端将返回可用的 Git 命令的列表。

开始使用

如果已安装所有依赖项，请使用 Hardhat 扩展创建第一个项目：

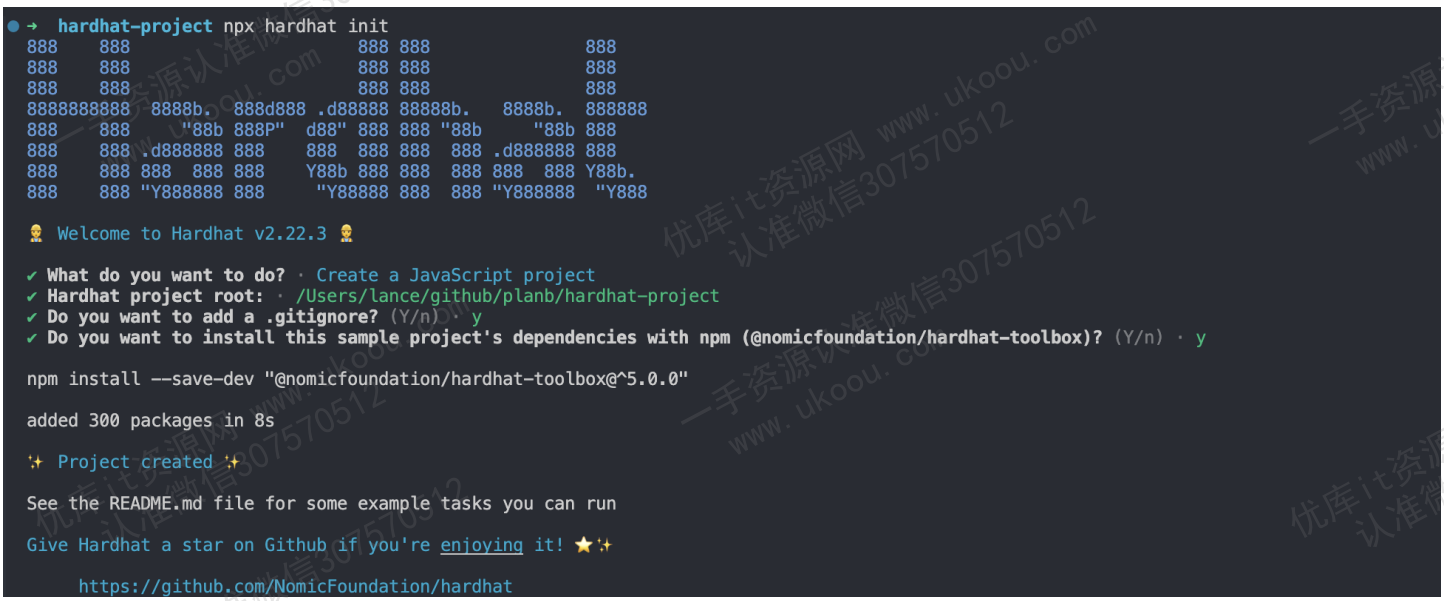
1. 在计算机上，为项目添加一个空目录。 若要从 Visual Studio Code 中创建目录，请依次转到“终端”“新终端”，然后键入 mkdir hardhat-project。 记下这个新目录的位置。 稍后需要用到此信息。
2. 进入文件夹，用npm初始化项目：`npm init` ;然后按照提示创建空项目，然后在项目中安装hardhat

```
npm install --save-dev hardhat
```

3. 用hardhat 创建示例项目：

```
npx hardhat init
```

我们就按照默认的选项快速创建项目



创建 Solidity 项目后，查看该项目的文件。

该项目包含 Solidity 代码样板。 请注意以下目录：

- 合约：contracts文件夹中包含 Lock.sol 合约
- ignition：包含对我们的示例合约的部署代码 `ignition/modules/Lock.js`
- 测试：包含以对Lock合约的测试文件 `test/Lock.js`

你还会看到下面的配置文件：

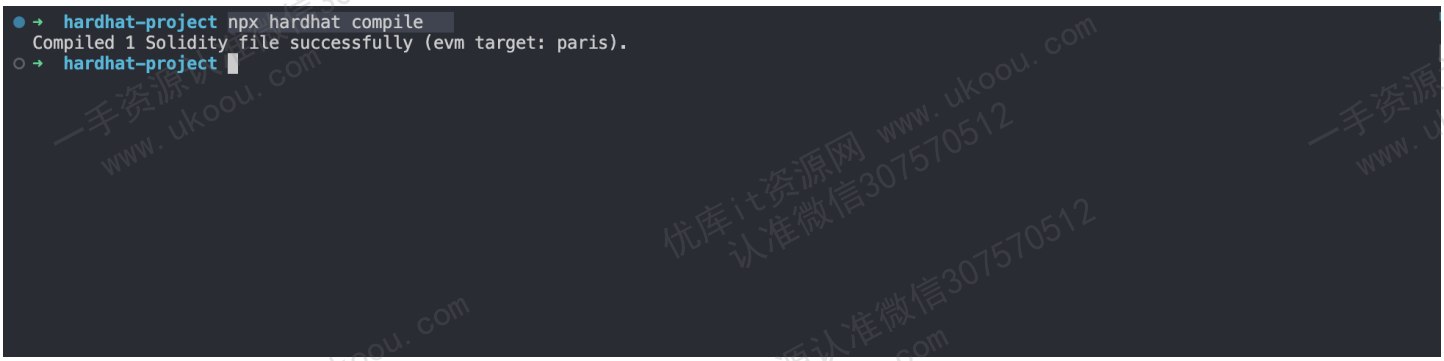
- package.json：定义项目详细信息和依赖项
- hardhat.config.js：关于hardhat的配置

现在，我们返回到项目本身。 首先，我们将重点介绍“合约”目录。

编译合约

我们将从“contracts”文件夹内的Lock.sol 智能合约开始。

在命令行运行： `npx hardhat compile`



编译成功后会生成artifacts文件夹

部署合约

成功编译合约后，可以在本地部署它

1. 这里我们先将合约部署到hardhat 内置的本地网络，启动本地网络： `npx hardhat node`

启动成功后会生成20个账号已经对应的私钥

```
→  hardhat-project npx hardhat node
Started HTTP and WebSocket JSON-RPC server at http://127.0.0.1:8545/

Accounts
========

WARNING: These accounts, and their private keys, are publicly known.
Any funds sent to them on Mainnet or any other live network WILL BE LOST.

Account #0: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 (10000 ETH)
Private Key: 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80

Account #1: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8 (10000 ETH)
Private Key: 0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d

Account #2: 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC (10000 ETH)
Private Key: 0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a

Account #3: 0x90F79bf6EB2c4f870365E785982E1f101E93b906 (10000 ETH)
Private Key: 0x7c852118294e51e653712a81e05800f419141751be58f605c371e15141b007a6

Account #4: 0x15d34AAf54267DB7D7c367839AAf71A00a2C6A65 (10000 ETH)
Private Key: 0x47e179ec197488593b187f80a00eb0da91f1b9d0b13f8733639f19c30a34926a

Account #5: 0x9965507D1a55bcC2695C58ba16FB37d819B0A4dc (10000 ETH)
Private Key: 0x8b3a350cf5c34c9194ca85829a2df0ec3153be0318b5e2d3348e872092edffba

Account #6: 0x976EA74026E726554dB657fA54763abd0C3a0aa9 (10000 ETH)
Private Key: 0x92db14e403b83dfe3df233f83dfa3a0d7096f21ca9b0d6d6b8d88b2b4ec1564e

Account #7: 0x14dC79964da2C08b23698B3D3cc7Ca32193d9955 (10000 ETH)
Private Key: 0x4bbbf85ce3377467afe5d46f804f221813b2bb87f24d81f60f1fcdbf7cbf4356

Account #8: 0x23618e81E3f5cdF7f54C3d65f7FBc0aBf5B21E8f (10000 ETH)
Private Key: 0xdbda1821b80551c9d65939329250298aa3472ba22feea921c0cf5d620ea67b97

Account #9: 0xa0Ee7A142d267C1f36714E4a8F75612F20a79720 (10000 ETH)
Private Key: 0x2a871d0798f97d79848a013d4936a73bf4cc922c825d33c1cf7073dff6d409c6

Account #10: 0xBcd4042DE499D14e55001CcbB24a551F3b954096 (10000 ETH)
Private Key: 0xf214f2b2cd398c806f84e317254e0f0b801d0643303237d97a22a48e01628897

Account #11: 0x71bE63f3384f5fb98995898A86B02Fb2426c5788 (10000 ETH)
Private Key: 0x701b615bbdfb9de65240bc28bd21bbc0d996645a3dd57e7b12bc2bdf6f192c82
```

2. 部署合约。运行：

`npx hardhat ignition deploy ./ignition/modules/Lock.ts --network localhost`

然后我们将看到部署后的信息如下：

```
→  hardhat-project npx hardhat ignition deploy ./ignition/modules/Lock.js --network localhost
Hardhat Ignition 🚀

Deploying [ LockModule ]

Batch #1
  Executed LockModule#Lock

[ LockModule ] successfully deployed 🚀

Deployed Addresses

LockModule#Lock - 0x5FbDB2315678afecb367f032d93F642f64180aa3
```

其中部署后的lock合约地址在上图最后一行；

练习 - 编写智能合约

在此单位中，我们将向前面创建的 hardhat-project 添加新的智能合约。

创建装运合约

将创建的智能合约用于跟踪从在线市场购买的商品的状态。 创建该合约时，装运状态设置为 `Pending`。 装运商品后，则将装运状态设置为 `Shipped` 并会发出事件。 交货后，则将商品的装运状态设置为 `Delivered`，并发出另一个事件。

开始此练习：

1. 在所创建项目的“contracts”目录中，右键单击该文件夹，然后选择新建名为 Shipping.sol 的文件。
2. 复制以下合约内容，并将其粘贴到新文件中。

```
pragma solidity >=0.4.25 <0.9.0;
contract Shipping {
    // Our predefined values for shipping listed as enums
    enum ShippingStatus { Pending, Shipped, Delivered }
    // Save enum ShippingStatus in variable status
    ShippingStatus private status;
    // Event to launch when package has arrived
    event LogNewAlert(string description);
    // This initializes our contract state (sets enum to Pending once the program starts)
    constructor(){
        status = ShippingStatus.Pending;
    }
    // Function to change to Shipped
    function Shipped() public {
        status = ShippingStatus.Shipped;
        emit LogNewAlert("Your package has been shipped");
    }
    // Function to change to Delivered
    function Delivered() public {
        status = ShippingStatus.Delivered;
        emit LogNewAlert("Your package has arrived");
    }
    // Function to get the status of the shipping
    function getStatus(ShippingStatus _status) internal pure returns (string memory) {
    // Check the current status and return the correct name
     if (ShippingStatus.Pending == _status) return "Pending";
     if (ShippingStatus.Shipped == _status) return "Shipped";
     if (ShippingStatus.Delivered == _status) return "Delivered";
    }
   // Get status of your shipped item
    function Status() public view returns (string memory) {
        ShippingStatus _status = status;
        return getStatus(_status);
    }
}
```

1. 查看该合约，以查看要发生的情况。 确认你可以成功生成该合约。
2. 在“资源管理器”窗格中，右键单击该合约名称，然后选择“生成合约”以编译该智能合约。

添加部署文件

现在我们将添加一个部署文件到 ignition/modules/Shipping.js

1. 将以下代码添加到该文件：

```
const { buildModule } = require("@nomicfoundation/hardhat-ignition/modules");

module.exports = buildModule("ShippingModule", (m) => {
  const shipping = m.contract("Shipping", []);
  m.call(shipping, "Status", []);
  return { shipping };
});

```

部署

1. 先启动本地网络，如果已经启动就跳过这一步

`npx hardhat node`

2. 执行部署命令：
   `npx hardhat ignition deploy ignition/modules/Shipping.js --network localhost`

执行结果如上

练习 - 测试智能合约

在此部分中，我们将为装运合约编写新的 JavaScript 测试。

开始测试

首先我们创建一个新的测试文件。

1. 在资源管理器窗格中，将鼠标悬停在“test”文件夹上方，然后右键单击。 选择“新建文件”，并创建名为 Shipping.js 的新文件。

2. 复制以下代码并将它粘贴到测试文件中：

```
const { expect } = require("chai");
const hre = require("hardhat");

describe("Shipping", function () {
  let shippingContract;
  before(async () => {
    // 生成合约实例并且复用
    shippingContract = await hre.ethers.deployContract("Shipping", []);
  });
  it("should return the status Pending", async function () {
    // assert that the value is correct
    expect(await shippingContract.Status()).to.equal("Pending");
  });
  it("should return the status Shipped", async () => {
    // Calling the Shipped() function
    await shippingContract.Shipped();
    // Checking if the status is Shipped
    expect(await shippingContract.Status()).to.equal("Shipped");
  });
});

```

事件测试

我们还要测试合约中发送的事件。添加一个测试，以确认事件会返回所需的说明。 将此测试放在文件的最后一个测试之后。 在最后一行的一组右大括号的正前方创建新行，在其中添加此测试。

```
  it("should return correct event description", async () => {
    // Calling the Delivered() function
    // Check event description is correct
    await expect(shippingContract.Delivered())
      // 验证事件是否被触发
      .to.emit(shippingContract, "LogNewAlert")
      // 验证事件的参数是否符合预期
      .withArgs("Your package has arrived");
  });
```

运行测试

在终端中键入以下内容：

```
npx hardhat test test/Shipping.js
```

应该看到所有测试都成功通过：