Solidity智能合约开发知识

第11.1课：Hardhat 3 环境搭建

学习目标：理解Hardhat 3的核心特性和优势、掌握Hardhat 3开发环境的搭建方法、学会项目初始化和配置、理解Ignition部署系统、能够完成从编译到部署的完整流程

预计学习时间：2-3小时

难度等级：初级到中级

重要提示：Hardhat 3是当前最先进的Solidity开发框架，掌握它对于进行专业级智能合约开发至关重要。从Remix到Hardhat 3，是从学习工具到专业开发工具的重要转变。

目录

---

1. Hardhat 3概述

1.1 为什么需要Hardhat

在学习Hardhat之前，很多开发者可能已经接触过Remix这个在线IDE。Remix确实是一个很好的学习工具，它零配置、开箱即用，界面友好，非常适合初学者快速上手和进行简单的测试。

Remix的优势：

- 零配置，打开浏览器就能用
- 界面直观，操作简单
- 适合快速测试和验证想法
- 不需要安装任何本地工具

Remix的局限性：

但是，当我们进入实际项目开发时，Remix的局限性就显现出来了：

1. **缺乏版本控制能力**：

    - 无法使用Git进行版本管理
    - 难以追踪代码变更历史
    - 团队协作困难
2. **无法进行自动化测试**：
    - 没有测试框架支持
    - 无法编写自动化测试用例
    - 难以保证代码质量
3. **不支持复杂的项目结构**：
    - 无法组织大型项目
    - 难以管理多个合约文件
    - 依赖管理困难
4. **无法集成CI/CD流程**：
    - 无法自动化构建和部署
    - 难以实现持续集成
    - 缺乏专业的开发工作流

Hardhat的优势：

这时候，我们就需要Hardhat这样的专业开发框架：

1. **完整的本地开发环境**：
    - 提供本地区块链模拟
    - 支持快速迭代开发
    - 零配置启动测试网络
2. **强大的测试框架**：
    - 集成Mocha和Chai
    - 支持TypeScript测试
    - 可以生成测试覆盖率报告
3. **自动化部署脚本**：
    - 支持脚本部署
    - 提供Ignition声明式部署
    - 支持多网络部署
4. **丰富的插件生态**：
    - 大量社区插件
    - 易于扩展功能
    - 与主流工具集成
5. **完美的Git支持**：
    - 支持版本控制
    - 适合团队协作
    - 可以集成CI/CD
6. **多网络支持**：
    - 本地网络、测试网、主网无缝切换
    - 统一的配置管理
    - 灵活的网络配置

学习路径：

从学习路径来看，Remix适合入门学习，Hardhat适合专业开发，最终目标是能够完成企业级项目的开发。掌握Hardhat是成为专业智能合约开发者的必经之路。

1.2 Hardhat 2 vs Hardhat 3

在开始学习之前，我们先来了解一下Hardhat 2和Hardhat 3的主要区别。这些区别非常重要，因为它们决定了我们如何配置和使用Hardhat。

版本要求对比：

特性	Hardhat 2	Hardhat 3
Node.js版本	>= 14	>= 22（硬性要求）
默认Solidity版本	0.8.19	0.8.24
TypeScript支持	需要手动配置	开箱即用
部署系统	脚本或hardhat-deploy	Ignition（新增）
插件安装	单独安装	Toolbox统一安装包
网络引擎	传统引擎	EDR引擎（更快）

为什么选择Hardhat 3：

1. **现代化架构**：
    - 全新的EDR运行时引擎
    - 更好的性能和稳定性
    - 面向未来的设计
2. **Ignition部署系统**：
    - 声明式部署定义
    - 自动状态管理
    - 错误恢复支持
    - 让部署更加可靠
3. **更好的开发体验**：
    - TypeScript开箱即用
    - 类型支持更强大
    - 插件系统改进
4. **官方主推版本**：
    - 持续更新和维护
    - 社区支持活跃
    - 长期支持保障

1.3 Hardhat 3核心功能

Hardhat 3提供了七个核心功能模块，每个模块都针对特定的开发需求：

1. 内置测试网络：

- Hardhat Network使用EDR引擎
- 本地区块链模拟环境
- 零配置启动
- 交易确认非常快速
- 提供完整的调试信息

2. 智能合约测试：

- 集成Mocha和Chai测试框架
- 支持Solidity测试
- 可以进行自动化测试
- 生成测试覆盖率报告
- 分析Gas消耗

3. Ignition部署系统：

- 声明式的部署定义方案
- 支持自动状态管理
- 错误恢复支持
- 依赖自动处理
- 支持跨链部署

4. 脚本部署：

- 传统的脚本部署方式
- 适合简单的部署场景
- 支持多网络
- 控制灵活

5. 合约验证：

- 自动在Etherscan上验证合约源码
- 实现源码透明化
- 支持一键验证命令
- 支持多网络

6. 调试工具：

- 支持console.log调试
- 提供堆栈跟踪和错误定位
- 进行Gas分析
- 详细的错误信息

7. 丰富的插件生态：

- Gas Reporter：Gas消耗报告
- Etherscan验证：合约验证
- Typechain：TypeScript类型生成
- OpenZeppelin升级：代理合约升级
- 大量社区插件

这些功能模块共同构成了一个完整的智能合约开发环境，能够满足从开发到部署的全流程需求。

---

2. 开发环境准备

2.1 Node.js和npm介绍

在开始使用Hardhat之前，我们需要先了解Node.js和npm，因为Hardhat是基于Node.js开发的。

Node.js是什么：

Node.js是一个JavaScript运行时环境，它基于Chrome的V8引擎，可以在服务器端运行JavaScript。Node.js采用异步非阻塞I/O模型，跨平台支持，广泛应用于：

- 开发工具（如Hardhat）
- 后端服务
- 命令行工具
- 构建脚本

npm是什么：

npm是Node Package Manager的缩写，是Node.js的包管理工具。它负责：

- 包的安装和管理
- 版本控制
- 脚本运行

在Hardhat项目中，我们通过npm来：

- 安装Hardhat本身
- 管理项目依赖
- 安装OpenZeppelin等库
- 管理各种插件

版本要求：

需要特别注意的是，Hardhat 3对Node.js版本有严格要求：

- **必须使用Node.js 22.0.0或更高版本**
- npm会随Node.js自动安装
- 推荐使用LTS长期支持版本

这是硬性要求，如果版本不符合，Hardhat 3将无法正常运行。

2.2 Node.js安装（Windows）

对于Windows用户，安装Node.js的步骤如下：

步骤1：访问官网下载：

1. 访问nodejs.org官网
2. 官网提供两个版本选择：
    - **LTS版本**：长期支持版本，推荐使用
    - **Current版本**：包含最新特性
3. 无论选择哪个版本，都要确保版本号大于等于22.0.0

步骤2：安装：

1. 双击下载的.msi安装文件
2. 接受许可协议
3. 选择安装路径（通常使用默认路径即可）
4. **重要**：一定要勾选"Add to PATH"选项，这样Node.js才能在任何目录下使用

步骤3：验证安装：

安装完成后，我们需要验证安装是否成功：

1. 打开命令提示符（CMD）或PowerShell
2. 输入以下命令验证Node.js版本：

   ```
   node --version
   ```

   应该输出 `v22.x.x` 或更高的版本号
3. 输入以下命令验证npm版本：

   ```
   npm --version
   ```

   应该输出 `10.x.x` 或更高的版本号

如果版本号正确，说明安装成功。

2.3 Node.js安装（macOS/Linux）

对于macOS和Linux用户，安装方式略有不同。

macOS用户：

有两种安装方式：

方式1：官网下载安装包：

1. 从官网下载.pkg安装包
2. 双击安装

方式2：使用Homebrew（推荐）：

1. 如果还没有安装Homebrew，运行安装脚本：

   ```
   /bin/bash -c "$(curl -fsSL
   https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
   ```
2. 安装好Homebrew后，使用以下命令安装Node.js 22：

   ```
   brew install node@22
   ```

Linux用户（Ubuntu/Debian）：

推荐使用NodeSource仓库：

1. 运行以下命令添加仓库：

```
curl -fsSL https://deb.nodesource.com/setup_22.x | sudo -E bash -
```

2. 使用apt-get安装：

```
sudo apt-get install -y nodejs
```

验证安装：

无论使用哪种方式，都要确保Node.js版本大于等于v22.0.0：

```
node --version
npm --version
```

推荐：使用nvm：

另外，推荐使用nvm（Node Version Manager），这个工具可以让你轻松切换不同的Node.js版本：

1. 安装nvm（macOS/Linux）：

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh | bash
```

2. 安装Node.js 22：

```
nvm install 22
```

3. 切换到Node.js 22：

```
nvm use 22
```

使用nvm的好处是可以轻松切换不同版本的Node.js，这对于需要维护多个项目的情况非常有用。

---

3. 项目初始化

3.1 创建Hardhat 3项目

现在，我们开始创建第一个Hardhat 3项目。这个过程非常简单，只需要几个命令。

步骤1：创建项目文件夹：

在终端中，创建一个新的项目目录：

```
mkdir my-hardhat3-project
cd my-hardhat3-project
```

步骤2：初始化npm项目：

运行以下命令初始化npm项目：

```
npm init -y
```

这会自动创建一个`package.json`文件，包含了项目的基本信息和依赖管理配置。`-y`参数表示使用默认配置，不需要交互式输入。

步骤3：安装Hardhat 3：

运行以下命令安装Hardhat 3：

```
npm install --save-dev hardhat@latest
```

这个命令会：

- 下载Hardhat 3及其所有依赖
- 将Hardhat添加到开发依赖中
- 安装过程通常需要1到2分钟
- 完成后会在项目目录下创建`node_modules`文件夹

步骤4：初始化Hardhat项目：

安装完成后，我们初始化Hardhat项目：

```
npx hardhat --init
```

这会启动一个交互式配置向导。

3.2 初始化配置选择

当运行 `npx hardhat --init` 时，向导会引导你完成以下配置步骤：

步骤1：选择 Hardhat 版本
向导会询问你想要使用的版本：

- `Which version of Hardhat would you like to use?` -> 选择 **`hardhat-3`**。

步骤2：确定项目路径

- `Where would you like to initialize the project?` -> 通常输入 `.` 表示当前目录。

步骤3：选择项目类型
Hardhat 3 提供了三个主要模板：

1. **A TypeScript Hardhat project using Node Test Runner and Viem**：
   - 使用现代的 Viem 库和 Node.js 原生测试运行器。
   - 适合追求极致速度和现代 API 的开发者。
2. **A TypeScript Hardhat project using Mocha and Ethers.js（推荐）**：
   - 使用传统的 Ethers.js 和 Mocha/Chai 测试框架。
   - 社区资源最丰富，也是本课程主要使用的配置。
3. **A minimal Hardhat project**：
   - 仅创建最基本的配置，适合有经验的开发者手动搭建。

步骤4：转换为 ESM 项目（重要）
Hardhat 3 强制要求使用 ESM 模式：

- `Hardhat only supports ESM projects. Would you like to change ".\package.json" to turn your project into ESM?` -> 选择 **`true`**。
- 这会在你的 `package.json` 中添加 `"type": "module"`。

其他配置：
随后向导还会询问：

1. **是否要添加 .gitignore 文件**：建议选择 **Yes**。
2. **是否要安装项目依赖**：建议选择 **Yes**（会自动安装 `@nomicfoundation/hardhat-toolbox` 等）。

3.3 初始化完成

初始化完成后，Hardhat会创建完整的项目结构：

创建的文件和目录：

1. **contracts目录**：
   - 存放Solidity合约文件
   - 包含一个Counter.sol示例合约
   - 可能包含Counter.t.sol Solidity测试文件
2. **scripts目录**：
   - 存放部署和交互脚本
   - 包含一个send-op-tx.ts示例脚本
   - 展示如何使用TypeScript编写部署脚本
3. **test目录**：
   - 存放单元测试文件
   - 包含一个Counter.ts文件
   - 这是TypeScript格式的测试文件
4. **ignition目录**（Hardhat 3新增）：
   - 存放Ignition部署模块
   - 在ignition/modules目录下有Counter.ts文件
   - 展示如何使用Ignition进行部署
5. **配置文件**：
   - `hardhat.config.ts`：TypeScript格式的配置文件
   - `package.json`：npm配置文件
   - `tsconfig.json`：TypeScript配置文件
   - `.gitignore`：Git忽略文件

现在，你已经有了一个完整的Hardhat 3开发环境。

---

4. 项目结构详解

4.1 目录结构概览

让我们详细了解一下Hardhat 3的项目结构。理解项目结构对于高效使用Hardhat非常重要。

完整的项目结构：

```
my-hardhat3-project/
├── contracts/           # Solidity合约目录
│   └── Counter.sol     # 示例合约
├── scripts/             # 部署和交互脚本
│   └── send-op-tx.ts   # 示例脚本
├── test/                # 测试文件目录
│   └── Counter.ts       # 测试文件
├── ignition/            # Ignition部署模块（Hardhat 3新增）
│   ├── modules/         # 部署模块定义
│   │   └── Counter.ts   # 示例部署模块
│   ├── parameters/      # 参数配置文件（可选）
│   └── deployments/     # 部署记录（自动生成）
├── node_modules/         # 依赖包目录
├── hardhat.config.ts    # Hardhat配置文件
├── package.json         # npm配置文件
├── tsconfig.json        # TypeScript配置文件
└── .gitignore           # Git忽略文件
```

4.2 contracts目录详解

`contracts`目录存放所有的Solidity智能合约文件，以`.sol`为扩展名。

示例合约：Counter.sol：

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.24;

contract Counter {
    uint256 public number;
    event Increment(uint256 by);

    function setNumber(uint256 newNumber) public {
        number = newNumber;
    }

    function increment() public {
        number++;
        emit Increment(1);
    }

    function incBy(uint256 amount) public {
        number += amount;
        emit Increment(amount);
    }
}
```

合约说明：

- 使用SPDX许可证标识符

- 指定Solidity版本为0.8.24（Hardhat 3的默认版本）
- 定义了一个公开的状态变量`number`
- 提供了设置数字、递增和按量递增的功能
- 会在递增时触发事件

Hardhat 3的特性：

- 默认使用Solidity 0.8.24版本
- 支持`.t.sol`测试文件（Solidity测试）
- 编译优化更好
- 编译速度更快

4.3 scripts目录详解

`scripts`目录存放部署和交互脚本，这些脚本使用ethers.js或viem编写，是传统的脚本部署方式，适合简单的部署场景。

示例脚本：send-op-tx.ts：

```
import hre from "hardhat";

async function main() {
  // 获取签名者
  const [deployer] = await hre.ethers.getSigners();

  // 获取账户余额
  const balance = await hre.ethers.provider.getBalance(deployer.address);
  console.log("Deploying contracts with account:", deployer.address);
  console.log("Account balance:", hre.ethers.formatEther(balance), "ETH");

  // 部署合约（Hardhat 3新API）
  const counter = await hre.ethers.deployContract("Counter");
  await counter.waitForDeployment();

  // 获取合约地址（Hardhat 3新API）
  const address = await counter.getAddress();
  console.log("Counter deployed to:", address);
}

main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

脚本说明：

1. **导入hardhat运行环境**：
    - 使用`import hre from "hardhat"`
    - `hre`是Hardhat Runtime Environment的缩写

2. **获取签名者**：
    - 使用`await hre.ethers.getSigners()`
    - 返回一个签名者数组，取第一个作为部署账户
3. **获取账户余额**：
    - 使用`hre.ethers.provider.getBalance()`
    - 使用`formatEther`方法格式化余额
4. **部署合约（Hardhat 3新API）**：
    - 使用`hre.ethers.deployContract("Counter")`
    - 这个方法比之前的`getContractFactory`方式更简洁
5. **等待部署确认**：
    - 使用`waitForDeployment()`等待部署确认
6. **获取合约地址（Hardhat 3新API）**：
    - 使用`getAddress()`方法
    - 替代了之前的`.address`属性

运行脚本：

```
npx hardhat run scripts/send-op-tx.ts
```

如果要部署到特定网络，可以添加`--network`参数：

```
npx hardhat run scripts/send-op-tx.ts --network sepolia
```

4.4 test目录详解

`test`目录存放单元测试文件，可以是TypeScript或JavaScript格式，也可以使用Solidity测试。测试使用Mocha和Chai框架。

示例测试：Counter.ts：

```
import { expect } from "chai";
import { network } from "hardhat";

const { ethers } = await network.connect();

describe("Counter", function () {
  it("Should emit the Increment event", async function () {
    const counter = await ethers.deployContract("Counter");

    await expect(counter.increment())
      .to.emit(counter, "Increment")
      .withArgs(1n);
  });

  it("Should increment correctly", async function () {
    const counter = await ethers.deployContract("Counter");
```

```
    await counter.incBy(5);
    expect(await counter.number()).to.equal(5n);
  });
});
```

测试说明：

1. **导入依赖**：
   - 导入`chai`的`expect`函数
   - 导入`hardhat`的`network`对象
2. **获取ethers对象（Hardhat 3新方式）**：
   - 使用`await network.connect()`
   - 这是Hardhat 3的新方式
3. **测试结构**：
   - 使用Mocha的`describe`和`it`结构
   - `describe`定义测试套件
   - `it`定义具体的测试用例
4. **测试用例**：
   - 第一个测试检查事件触发
   - 第二个测试检查递增功能

运行测试：

```
npx hardhat test
```

如果要运行特定的测试文件：

```
npx hardhat test test/Counter.ts
```

Hardhat 3的测试特点：

- TypeScript开箱即用
- 更好的类型支持
- 支持Solidity测试
- 执行速度更快

4.5 ignition目录详解（Hardhat 3新特性）

`ignition`目录是Hardhat 3的新特性，用于Ignition部署系统。这是一个声明式部署系统，支持自动状态管理，适合复杂的系统部署。

目录结构：

```
ignition/
├── modules/            # 部署模块定义
│   └── Counter.ts      # 示例部署模块
├── parameters/          # 参数配置文件（可选）
│   ├── sepolia.json    # Sepolia网络参数
│   └── mainnet.json    # 主网参数
└── deployments/         # 部署记录（自动生成）
    └── chain-11155111/ # 按链ID组织
```

示例模块：Counter.ts：

```
import { buildModule } from "@nomicfoundation/hardhat-ignition/modules";

export default buildModule("CounterModule", (m) => {
  const counter = m.contract("Counter");

  // 部署后调用incBy方法，初始化值为10
  m.call(counter, "incBy", [10n]);

  return { counter };
});
```

模块说明：

1. **导入buildModule**：
   - 从`@nomicfoundation/hardhat-ignition/modules`导入
2. **创建模块**：
   - 使用`buildModule`函数创建
   - 第一个参数是模块名称
   - 第二个参数是一个函数，接收模块构建器`m`
3. **定义部署**：
   - 使用`m.contract("Counter")`定义要部署的合约
4. **部署后调用**：
   - 使用`m.call()`在部署后调用合约的方法
   - 这里调用了`incBy`函数，传入参数10
5. **返回对象**：
   - 返回包含部署合约引用的对象

Ignition的优势：

1. **声明式部署**：
   - 定义部署逻辑，而不是执行步骤
   - 更清晰、更易维护
2. **自动状态管理**：
   - 跟踪已部署的合约
   - 不会重复部署
3. **错误恢复**：

- 如果部署失败，可以继续执行
- 自动跳过已完成的步骤

4. **依赖自动处理**：
   - 自动处理合约之间的依赖关系
   - 按正确顺序部署
5. **跨链部署**：
   - 支持多链部署
   - 统一的部署定义

部署命令：

```
npx hardhat ignition deploy ignition/modules/Counter.ts
```

如果要部署到特定网络：

```
npx hardhat ignition deploy ignition/modules/Counter.ts --network sepolia
```

查看部署状态：

```
npx hardhat ignition status chain-11155111
```

`chain-11155111`对应Sepolia测试网，不同的网络有不同的链ID。

---

5. 配置文件详解

5.1 hardhat.config.ts核心配置

`hardhat.config.ts`是Hardhat的核心配置文件，使用TypeScript格式，包含了网络配置、编译器版本设置和插件配置。

完整的配置文件示例：

```
import { HardhatUserConfig } from "hardhat/config";
import "@nomicfoundation/hardhat-toolbox";
import "dotenv/config";

const config: HardhatUserConfig = {
  solidity: {
    version: "0.8.24",
    settings: {
      optimizer: {
        enabled: true,
        runs: 200,
      },
    },
  },
  networks: {
```

```
    hardhat: {
      type: "edr-simulated",
      chainId: 31337,
    },
    sepolia: {
      type: "http",
      url: process.env.SEPOLIA_URL || "",
      accounts: process.env.PRIVATE_KEY ? [process.env.PRIVATE_KEY] : [],
      chainId: 11155111,
    },
    mainnet: {
      type: "http",
      url: process.env.MAINNET_URL || "",
      accounts: process.env.PRIVATE_KEY ? [process.env.PRIVATE_KEY] : [],
      chainId: 1,
    },
  },
  gasReporter: {
    enabled: process.env.REPORT_GAS === "true",
    currency: "USD",
  },
  etherscan: {
    apiKey: process.env.ETHERSCAN_API_KEY || "",
  },
};

export default config;
```

配置说明：

1. **导入模块**：
   - `HardhatUserConfig`：TypeScript类型定义，提供类型检查
   - `@nomicfoundation/hardhat-toolbox`：Hardhat 3的核心工具包，包含了常用的插件
   - `dotenv/config`：自动加载.env文件中的环境变量
2. **配置对象**：
   - 使用`HardhatUserConfig`类型，确保类型安全
3. **Solidity配置**：
   - 指定编译器版本为0.8.24
   - 在settings中配置优化器
4. **网络配置**：
   - 每个网络都必须指定`type`字段（Hardhat 3要求）
   - 本地网络使用`edr-simulated`
   - 测试网和主网使用`http`
5. **Gas报告配置**：
   - 通过环境变量控制是否启用
6. **Etherscan配置**：
   - 用于合约验证

5.2 Solidity编译器配置

Solidity编译器配置可以很简单，也可以很复杂，取决于项目需求。

简单配置：

```
solidity: "0.8.24"
```

这种方式适合大多数项目，直接指定版本字符串。

对象配置：

如果需要更精细的控制，可以使用对象配置：

```
solidity: {
  version: "0.8.24",
  settings: {
    optimizer: {
      enabled: true,
      runs: 200,
    },
  },
}
```

多编译器配置：

如果项目中有多个合约使用不同的Solidity版本，可以配置多个编译器：

```
solidity: {
  compilers: [
    {
      version: "0.8.24",
      settings: {
        optimizer: {
          enabled: true,
          runs: 200,
        },
      },
    },
    {
      version: "0.7.6",
      settings: {
        optimizer: {
          enabled: true,
          runs: 200,
        },
      },
    },
  ],
}
```

优化器配置：

`optimizer`配置很重要：

- `enabled: true`：启用优化器，可以降低Gas消耗，推荐在生产环境使用
- `runs: 200`：优化运行次数
  - `200`：适中的值，也是默认值
  - `1`：优化部署成本，适合一次性部署的合约
  - `10000`或更高：优化执行成本，适合会被频繁调用的合约

Hardhat 3的编译改进：

- 编译速度更快
- 优化支持更好
- 支持增量编译，只编译修改过的文件

5.3 网络配置详解

网络配置是Hardhat 3的重要变化，每个网络都必须指定`type`字段。

本地Hardhat网络：

```
hardhat: {
  type: "edr-simulated",
  chainId: 31337,
}
```

- `type: "edr-simulated"`：EDR引擎模拟网络
- `chainId: 31337`：本地网络的链ID
- 默认提供10个测试账户，每个账户有10000 ETH
- 适合本地开发和测试

Sepolia测试网：

```
sepolia: {
  type: "http",
  url: process.env.SEPOLIA_URL || "",
  accounts: process.env.PRIVATE_KEY ? [process.env.PRIVATE_KEY] : [],
  chainId: 11155111,
}
```

- `type: "http"`：HTTP RPC网络
- `url`：从环境变量`SEPOLIA_URL`读取RPC节点URL
- `accounts`：从环境变量`PRIVATE_KEY`读取私钥
- `chainId: 11155111`：Sepolia的链ID

主网配置：

```
mainnet: {
  type: "http",
  url: process.env.MAINNET_URL || "",
  accounts: process.env.PRIVATE_KEY ? [process.env.PRIVATE_KEY] : [],
  chainId: 1,
}
```

- 配置方式与测试网类似
- `chainId: 1`：以太坊主网的链ID
- **注意**：主网是真实资金，部署前务必充分测试并备份私钥

为什么需要type字段：

Hardhat 3对网络强制要求`type`字段，这是为了：

- 明确网络类型
- 支持不同的网络引擎（EDR、HTTP等）
- 提供更好的类型检查
- 老项目迁移时最容易踩坑的地方

5.4 其他配置选项

Gas报告配置：

```
gasReporter: {
  enabled: process.env.REPORT_GAS === "true",
  currency: "USD",
}
```

- `enabled`：通过环境变量`REPORT_GAS`控制是否启用
- `currency`：报告货币单位
- 需要Coinmarketcap API来获取价格

Etherscan配置：

```
etherscan: {
  apiKey: process.env.ETHERSCAN_API_KEY || "",
}
```

- 用于合约验证
- 从环境变量读取API密钥
- 验证命令：`npx hardhat verify --network sepolia ADDRESS`

---

6. 网络配置

6.1 环境变量配置

为了安全地管理敏感信息，我们需要使用环境变量。

创建.env文件：

在项目根目录创建`.env`文件，存储敏感信息：

```
# RPC节点URL
SEPOLIA_URL=https://sepolia.infura.io/v3/your-project-id
MAINNET_URL=https://mainnet.infura.io/v3/your-project-id

# 私钥（敏感信息，切勿泄露）
PRIVATE_KEY=your-private-key-here

# API密钥
ETHERSCAN_API_KEY=your-etherscan-api-key
COINMARKETCAP_API_KEY=your-coinmarketcap-api-key
```

环境变量说明：

1. **RPC节点URL**：
   - 可以从Infura、Alchemy等服务获取
   - 格式通常是`https://sepolia.infura.io/v3/your-project-id`
2. **私钥**：
   - 是敏感信息，必须妥善保管
   - 切勿泄露
   - 在.env文件中直接写入私钥字符串
3. **API密钥**：
   - Etherscan API密钥：用于合约验证
   - Coinmarketcap API密钥：用于Gas报告的价格转换

安装dotenv：

```
npm install dotenv --save-dev
```

在配置文件中，导入`dotenv/config`，这会自动加载.env文件中的环境变量：

```
import "dotenv/config";
```

然后就可以使用`process.env.VARIABLE_NAME`访问这些变量。

安全最佳实践：

1. **必须创建.env文件**存储敏感信息
2. **必须添加.env到.gitignore**，不要提交到Git
3. **不要提交私钥到Git**
4. **使用环境变量而不是硬编码**

禁止的做法：

1. 硬编码私钥在代码中
2. 提交.env文件到仓库

3. 分享私钥给他人
4. 使用主网私钥进行测试

6.2 获取RPC节点URL

要连接到测试网或主网，我们需要RPC节点URL。有多个服务提供商可以选择：

Infura：

1. 访问infura.io
2. 注册账号并创建项目
3. 获取项目ID
4. RPC URL格式：`https://sepolia.infura.io/v3/YOUR-PROJECT-ID`

Alchemy：

1. 访问alchemy.com
2. 注册账号并创建应用
3. 获取API密钥
4. RPC URL格式：`https://eth-sepolia.g.alchemy.com/v2/YOUR-API-KEY`

公共RPC节点：

也可以使用公共RPC节点，但可能有速率限制：

- Sepolia: `https://rpc.sepolia.org`
- Mainnet: `https://eth.llamarpc.com`

6.3 获取测试ETH

在测试网上部署合约需要测试ETH。获取方式：

Sepolia水龙头：

1. **Alchemy Sepolia Faucet**：
    - 访问：https://sepoliafaucet.com
    - 连接钱包
    - 每天可以领取0.5 ETH
2. **Infura Sepolia Faucet**：
    - 访问：https://www.infura.io/faucet/sepolia
    - 需要注册账号
    - 每天可以领取0.5 ETH
3. **PoW Faucet**：
    - 访问：https://sepolia-faucet.pk910.de
    - 需要挖矿证明
    - 可以获取更多测试ETH

---

7. Ignition部署系统

7.1 Ignition概述

Ignition是Hardhat 3全新的部署系统，采用声明式部署定义，提供了比传统脚本部署更强大和可靠的功能。

Ignition的核心特性：

1. **声明式部署定义**：
   - 定义部署逻辑，而不是执行步骤
   - 更清晰、更易维护
2. **自动状态管理**：
   - 跟踪已部署的合约
   - 不会重复部署
   - 支持增量部署
3. **错误恢复**：
   - 如果部署失败，可以继续执行
   - 自动跳过已完成的步骤
   - 支持部分部署恢复
4. **依赖自动处理**：
   - 自动处理合约之间的依赖关系
   - 按正确顺序部署
   - 支持复杂的依赖图
5. **跨链部署**：
   - 支持多链部署
   - 统一的部署定义
   - 可以部署到不同的网络

Ignition的适用场景：

- 多合约系统部署
- 生产环境部署
- 需要状态管理的场景
- 有复杂依赖关系的项目

7.2 基础Ignition模块

让我们看一个基础的Ignition模块示例：

```
import { buildModule } from "@nomicfoundation/hardhat-ignition/modules";

export default buildModule("CounterModule", (m) => {
  const counter = m.contract("Counter");

  // 部署后调用incBy方法，初始化值为10
  m.call(counter, "incBy", [10n]);

  return { counter };
});
```

模块说明：

1. **导入buildModule**：

    - 从`@nomicfoundation/hardhat-ignition/modules`导入
2. **创建模块**：
    - 使用`buildModule`函数创建
    - 第一个参数是模块名称
    - 第二个参数是一个函数，接收模块构建器`m`
3. **定义部署**：
    - 使用`m.contract("Counter")`定义要部署的合约
    - 合约名称必须与contracts目录下的文件名匹配
4. **部署后调用**：
    - 使用`m.call()`在部署后调用合约的方法
    - 第一个参数是合约引用
    - 第二个参数是方法名
    - 第三个参数是参数数组
5. **返回对象**：
    - 返回包含部署合约引用的对象
    - 这些引用可以在其他模块中使用

7.3 多合约部署

Ignition特别适合部署多合约系统，因为它能自动处理依赖关系。

示例：Token和Vault系统：

```
import { buildModule } from "@nomicfoundation/hardhat-ignition/modules";

export default buildModule("VaultSystem", (m) => {
  // 先部署Token合约
  const token = m.contract("Token");

  // 然后部署Vault合约，传入Token地址
  const vault = m.contract("Vault", {
    args: [token],
  });

  // 可选：给某个地址转账一些Token
  const deployer = m.getAccount(0);
  m.call(token, "transfer", [deployer, 1000n]);

  return { token, vault };
});
```

依赖处理：

- Vault合约依赖于Token合约
- Ignition会自动先部署Token，然后部署Vault
- 正确传递依赖关系，无需手动管理顺序

7.4 参数化部署

对于不同网络，可能需要不同的参数。Ignition支持参数化部署。

使用参数文件：

在`ignition/parameters`目录下创建参数文件：

sepolia.json：

```
{
  "SimpleStorageModule": {
    "SimpleStorage": {
      "args": [200]
    }
  }
}
```

mainnet.json：

```
{
  "SimpleStorageModule": {
    "SimpleStorage": {
      "args": [100]
    }
  }
}
```

部署时，Ignition会自动读取对应网络的参数文件，使用指定的参数进行部署。

7.5 Ignition部署命令

部署到本地网络：

```
npx hardhat ignition deploy ignition/modules/Counter.ts
```

部署到测试网：

```
npx hardhat ignition deploy ignition/modules/Counter.ts --network sepolia
```

查看部署状态：

```
npx hardhat ignition status chain-11155111
```

`chain-11155111`对应Sepolia测试网，不同的网络有不同的链ID。

验证已部署的合约：

```
npx hardhat ignition verify chain-11155111
```

这会自动验证所有已部署的合约。

8. 常用命令与工具

8.1 编译相关命令

编译所有合约：

```
npx hardhat compile
```

这会编译contracts目录下的所有Solidity文件，生成artifacts和cache目录。

清理缓存：

如果遇到编译问题，可以清理缓存：

```
npx hardhat clean
```

然后重新编译。

8.2 测试相关命令

运行所有测试：

```
npx hardhat test
```

运行特定测试文件：

```
npx hardhat test test/Counter.ts
```

显示Gas报告：

设置环境变量`REPORT_GAS=true`，然后运行测试：

```
REPORT_GAS=true npx hardhat test
```

8.3 部署相关命令

传统脚本部署：

```
# 部署到本地网络
npx hardhat run scripts/send-op-tx.ts

# 部署到Sepolia测试网
npx hardhat run scripts/send-op-tx.ts --network sepolia

# 部署到主网
npx hardhat run scripts/send-op-tx.ts --network mainnet
```

Ignition部署：

```
# 部署到本地网络
npx hardhat ignition deploy ignition/modules/Counter.ts

# 部署到Sepolia测试网
npx hardhat ignition deploy ignition/modules/Counter.ts --network sepolia

# 查看部署状态
npx hardhat ignition status chain-11155111

# 验证已部署的合约
npx hardhat ignition verify chain-11155111
```

8.4 网络相关命令

启动本地节点：

```
npx hardhat node
```

这会启动一个HTTP和WebSocket JSON-RPC服务器，默认地址是`http://127.0.0.1:8545/`。

在本地节点上运行脚本：

```
npx hardhat run scripts/send-op-tx.ts --network localhost
```

8.5 其他命令

打开Hardhat控制台：

```
npx hardhat console
```

可以交互式地测试合约。

查看帮助信息：

```
npx hardhat help
```

命令格式：

```
npx hardhat [task] [options]
```

- `npx`：用于运行npm包中的命令
- `hardhat`：CLI工具
- `task`：要执行的任务
- `options`：可选参数

常用选项：

- `--network`：指定网络
- `--show-stack-traces`：显示详细错误信息
- `--verbose`：显示详细输出

---

9. 实战演示

9.1 完整开发流程

让我们进行一个完整的实战演示，从创建项目到部署合约。

步骤1：创建项目：

```
mkdir my-first-hardhat3-project
cd my-first-hardhat3-project
npm init -y
npm install --save-dev hardhat@latest
npx hardhat --init
```

选择创建TypeScript项目，添加.gitignore，安装依赖。

步骤2：编译合约：

```
npx hardhat compile
```

Hardhat会使用0.8.24版本编译合约文件。编译成功后，会在项目目录下生成artifacts目录（存放编译产物）和cache目录（存放编译缓存）。

步骤3：运行测试：

```
npx hardhat test
```

测试框架会运行所有测试用例。对于Counter合约，测试会检查事件触发和递增功能。如果所有测试通过，会显示"2 passing"以及执行时间。

步骤4：启动本地节点：

```
npx hardhat node
```

这会启动一个本地的Hardhat网络，提供HTTP和WebSocket JSON-RPC服务，默认地址是`http://127.0.0.1:8545/`。

终端会显示所有预置的测试账户及其地址和私钥，每个账户都有10000 ETH。

步骤5：使用Ignition部署：

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

部署完成后，会显示合约地址。

步骤6：验证部署：

```
npx hardhat ignition status chain-31337
```

查看部署状态，确认合约已成功部署。

步骤7：测试合约交互：

可以编写脚本或使用Hardhat console与部署的合约进行交互，确认功能正常。

9.2 多合约系统部署示例

让我们看一个更复杂的例子：部署一个包含多个合约的系统。

合约1：Token.sol：

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.24;

contract Token {
    mapping(address => uint256) public balanceOf;

    function transfer(address to, uint256 amount) public {
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
    }
}
```

合约2：Vault.sol：

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.24;

import "./Token.sol";

contract Vault {
    Token public token;
    mapping(address => uint256) public deposits;

    constructor(address _token) {
        token = Token(_token);
    }

    function deposit(uint256 amount) public {
        token.transfer(address(this), amount);
        deposits[msg.sender] += amount;
    }
}
```

Ignition模块：VaultSystem.ts：

```
import { buildModule } from "@nomicfoundation/hardhat-ignition/modules";

export default buildModule("VaultSystem", (m) => {
  // 先部署Token合约
  const token = m.contract("Token");

  // 然后部署Vault合约，传入Token地址
  const vault = m.contract("Vault", {
    args: [token],
  });

  // 可选：给部署者转账一些Token
  const deployer = m.getAccount(0);
  m.call(token, "transfer", [deployer, 1000n]);

  return { token, vault };
});
```

部署：

```
npx hardhat ignition deploy ignition/modules/VaultSystem.ts
```

Ignition会按照依赖关系自动处理部署顺序，确保Token先部署，然后Vault使用Token的地址进行部署。

---

10. 最佳实践

10.1 项目组织

目录结构：

- 保持清晰的目录结构
- 按功能组织合约文件
- 使用有意义的文件名

代码组织：

- 使用导入语句组织代码
- 避免循环依赖
- 保持合约职责单一

10.2 版本管理

Git使用：

- 使用Git进行版本控制
- 提交有意义的commit信息
- 使用分支管理不同功能

.gitignore配置：

确保.gitignore包含：

```
node_modules/
.env
artifacts/
cache/
coverage/
```

10.3 安全实践

私钥管理：

- 永远不要提交私钥到Git
- 使用环境变量管理敏感信息
- 为不同环境使用不同的私钥

测试：

- 编写完整的测试用例
- 测试边界情况
- 测试异常情况

10.4 部署实践

使用Ignition：

- 对于复杂系统，使用Ignition部署
- 利用状态管理功能
- 使用参数文件管理不同环境的配置

验证合约：

- 部署后立即验证合约
- 使用Etherscan验证
- 确保源码透明

---

11. 常见问题与解决方案

11.1 Node.js版本问题

问题：Hardhat 3要求Node.js >= 22，但当前版本不满足。

解决方案：

1. 检查当前版本：

   ```
   node --version
   ```

2. 升级Node.js到22或更高版本
3. 使用nvm管理版本：

```
nvm install 22
nvm use 22
```

11.2 编译错误

问题：合约编译失败。

解决方案：

1. 检查Solidity版本是否匹配
2. 清理缓存：`npx hardhat clean`
3. 重新编译：`npx hardhat compile`
4. 检查导入路径是否正确

11.3 网络连接问题

问题：无法连接到测试网或主网。

解决方案：

1. 检查RPC URL是否正确
2. 检查网络配置中的type字段
3. 确认环境变量已正确设置
4. 检查网络连接

11.4 部署失败

问题：部署合约失败。

解决方案：

1. 检查账户余额是否足够
2. 检查Gas限制设置
3. 查看详细错误信息：`--show-stack-traces`
4. 使用Ignition的错误恢复功能

11.5 测试失败

问题：测试用例失败。

解决方案：

1. 检查测试网络配置
2. 确认合约已正确编译
3. 检查测试代码逻辑
4. 查看详细错误信息

---

12. 学习资源与总结

12.1 官方资源

Hardhat官方文档：

- 官网：https://hardhat.org
- 文档：https://hardhat.org/docs
- Hardhat 3新特性文档
- Ignition部署指南

GitHub仓库：

- Hardhat主仓库：https://github.com/NomicFoundation/hardhat
- 示例项目：https://github.com/NomicFoundation/hardhat-boilerplate

12.2 社区资源

Discord社区：

- Hardhat Discord服务器
- 活跃的开发者社区
- 及时的技术支持

相关工具文档：

- Ethers.js v6文档
- Viem文档
- OpenZeppelin合约库

12.3 核心知识点总结

通过本课程的学习，你应该已经掌握了：

1. **Hardhat 3的核心特性**：
    - EDR引擎带来的性能提升
    - Ignition部署系统
    - TypeScript开箱即用
2. **开发环境搭建**：
    - Node.js 22+安装
    - 项目初始化
    - 依赖管理
3. **项目结构理解**：
    - contracts目录
    - scripts目录
    - test目录
    - ignition目录
4. **配置文件**：
    - hardhat.config.ts配置
    - 网络配置
    - 编译器配置
5. **Ignition部署系统**：
    - 声明式部署定义

    - 状态管理
    - 错误恢复
6. **常用命令**：
    - 编译、测试、部署
    - 网络管理
    - 合约验证

12.4 下一步学习

深入学习：

1. **Hardhat 3编译和部署**：
    - 编译选项配置
    - 复杂模块编写
    - 多合约依赖处理
2. **测试框架**：
    - 编写完整的测试用例
    - 测试覆盖率
    - Gas分析
3. **插件开发**：
    - 自定义插件
    - 集成第三方工具
4. **CI/CD集成**：
    - GitHub Actions
    - 自动化部署
    - 持续集成

12.5 总结

Hardhat 3是当前最先进的Solidity开发框架，掌握它对于进行专业级智能合约开发至关重要。

关键收获：

1. **从Remix到Hardhat 3**：实现了专业开发工作流的转型
2. **掌握Ignition部署系统**：现代化的声明式部署方式
3. **完整的开发环境**：编译、测试、部署一体化
4. **多网络支持**：轻松切换开发、测试和生产环境
5. **包管理能力**：使用npm管理项目依赖
6. **性能提升**：EDR引擎带来了更快的速度

实践建议：

- 在本地充分测试
- 使用测试网验证
- 编写完整的测试
- 遵循最佳实践
- 持续学习新特性

恭喜！你已经完成了Hardhat 3环境的搭建、项目初始化、合约编译、测试执行和Ignition部署。现在你可以开发复杂的DApp，编写完整的测试，使用Ignition进行部署，部署到各种网络，使用专业的开发工作流。

祝你学习愉快，开发顺利！
