# 2025《区块链技术与应用》
# （微众银行）
# 大作业：在区块链上构建应用系统

## 一、背景与目标

本作业面向已掌握区块链基本概念与以太坊开发入门的学生：

- 使用 Web3j 构建一个基于 Spring Boot 的后端应用（提供 REST/CLI 接口与区块链交互）
- 使用 scaffold-eth-2 搭建一个纯前端 DApp（通过浏览器钱包与区块链交互）

## 作业背景

Web3 应用由“智能合约 + 客户端/服务端”组成。后端应用通常通过 Web3j 等 SDK 承担业务编排、签名与链上调用；前端 DApp 则直接通过钱包与合约交互（如 MetaMask）。

通过本作业，学生将贯通“合约 → 工程化 → 部署与交互”的完整链路，形成端到端的 Web3 应用开发能力。

## 作业目标

1. 能够编写与编译 Solidity 智能合约，理解 ABI、字节码及网络部署流程。
2. 熟练使用 Web3j 生成合约 Java 封装，基于 Spring Boot 实现与区块链交互的 REST 接口。
3. 掌握 React 前端项目结构与合约集成，完成一个可交互的 DApp。
4. 了解本地链（Hardhat）、测试网、主网的差异与密钥/节点管理的安全实践。
5. 具备最小可用产品（MVP）级别的需求分析、接口设计、测试与文档产出能力。

## 作业产出

1. 可运行的后端服务与前端 DApp
2. 在线源代码，github、gitee 等的 git 在线仓库，需要包含：后端服务源码、合约源码与前端部署
3. 大作业报告
4. 部署到测试网并提供交互证明（交易哈希/区块浏览器链接）
5. 可选：演示视频（3–5 分钟）

# 二、实验步骤

说明：

- 需要团队协作（≤3 人），需标注分工。
- 需要在测试链完成开发与自测

## 2.1 使用 Web3J 构建后台应用

前置要求：

- JDK 21+，Gradle
- Node.js 18+（用于合约工具链）

### 2.1.1 下载 Spring-boot-web3j 示例代码

为了同学们快速上手，提供了 Spring-boot + web3J 的示例代码，后续同学们基于该项目仓库进行修改即可。

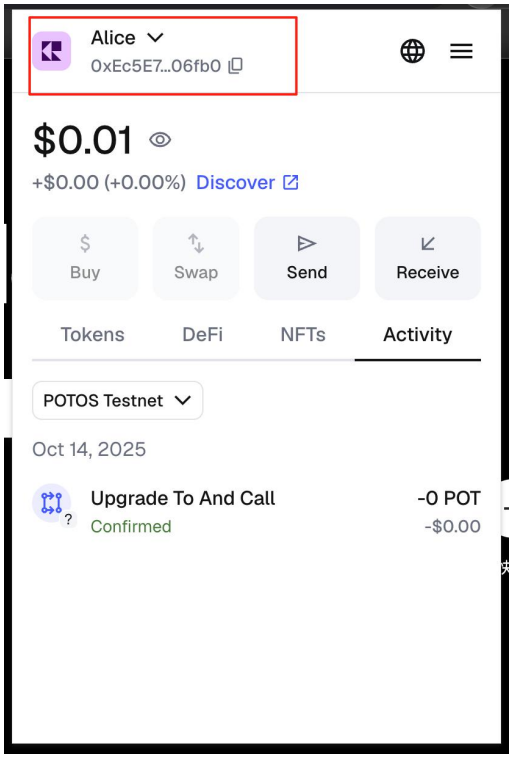
# 代码仓库：
https://github.com/kyonRay/spring-boot-web3j

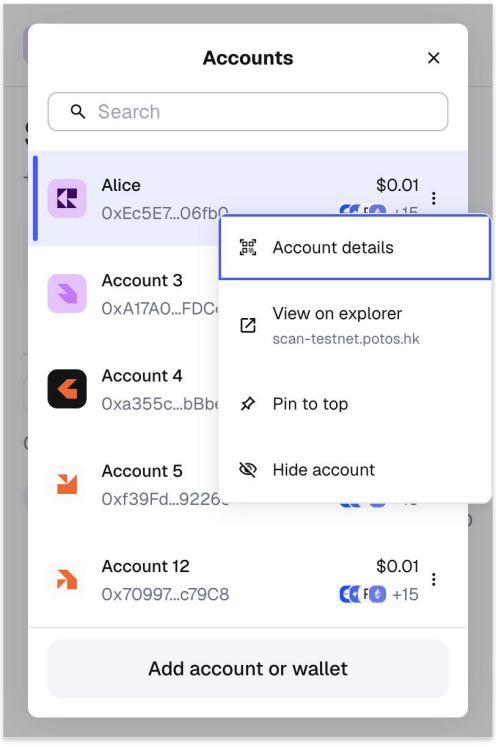
# 代码结构：
```
tree ./src/main/
./src/main/
├── java
│   └── com
│       └── wetech
│           └── demo
│               └── web3j
│                   ├── Application.java              # App 总入口
│                   ├── config
│                   │   └── Web3jConfig.java          # Web3J 配置
│                   ├── contracts                     # 合约编译 Java 接口目录
│                   │   └── simplestorage
│                   │       └── SimpleStorage.java    # 合约编译成的 Java 接口文件
│                   ├── controller
│                   │   └── SimpleStorageController.java    # Spring-boot Controller
│                   └── service
│                       └── SimpleStorageService.java       # Spring-boot Service
└── resources
```

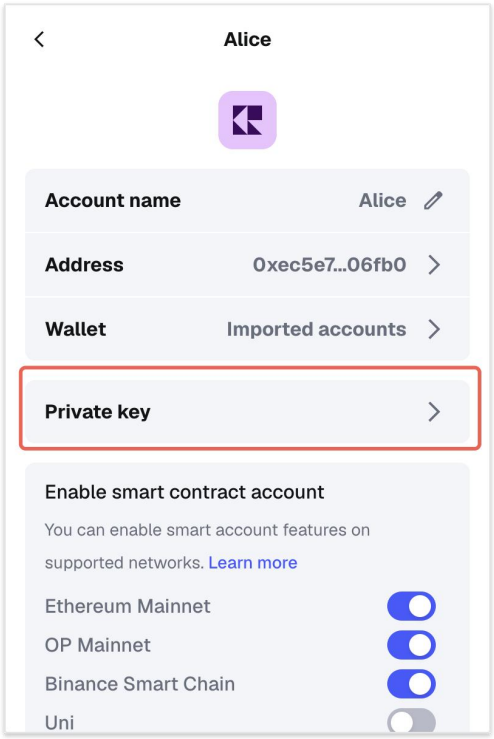
```
├── application.properties        # APP 配置文件
└── contracts
    └── SimpleStorage.sol         # 智能合约
```

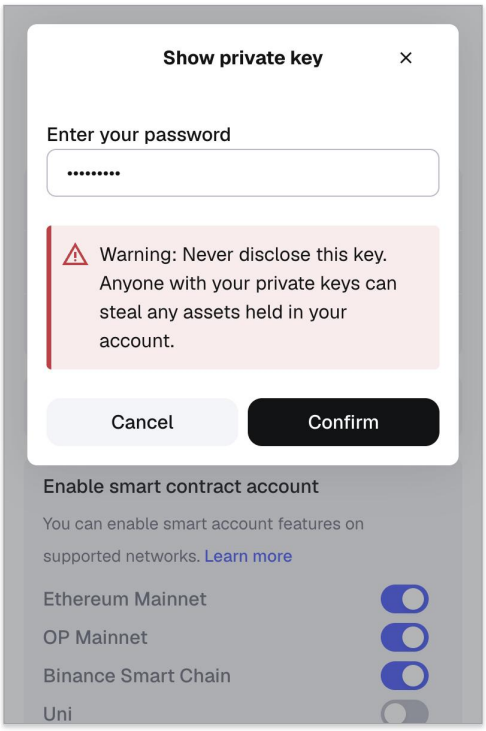
```
# 配置文件说明
# Server configuration
server.port=8080          # 服务监听端口

# Web3j configuration
web3j.client-address=https://rpc-testnet.potos.hk          # 链的 URL
# 发交易时的私钥，不要轻易泄露，可以从 MetaMask 中获取
web3j.private-key=0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80
web3j.gas-price=20000000000
web3j.gas-limit=6721975

# Logging configuration
logging.level.root=INFO
logging.level.com.wetech.demo.web3j=DEBUG
logging.level.org.web3j=INFO
```

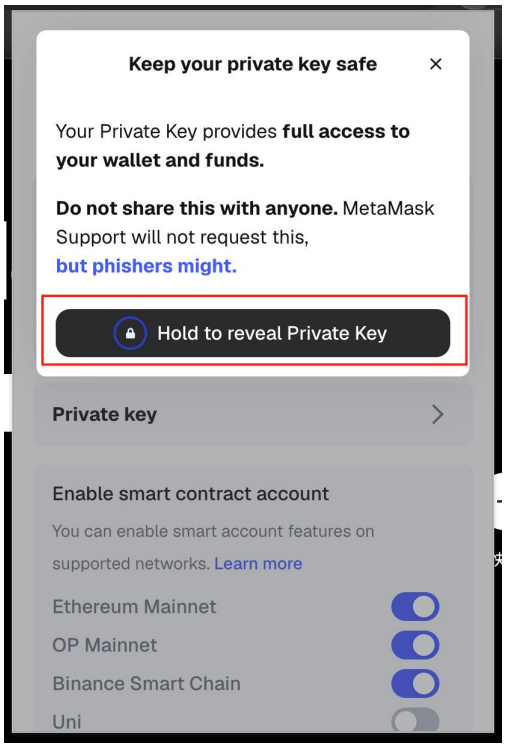
**从 MetaMask 获取私钥的步骤**

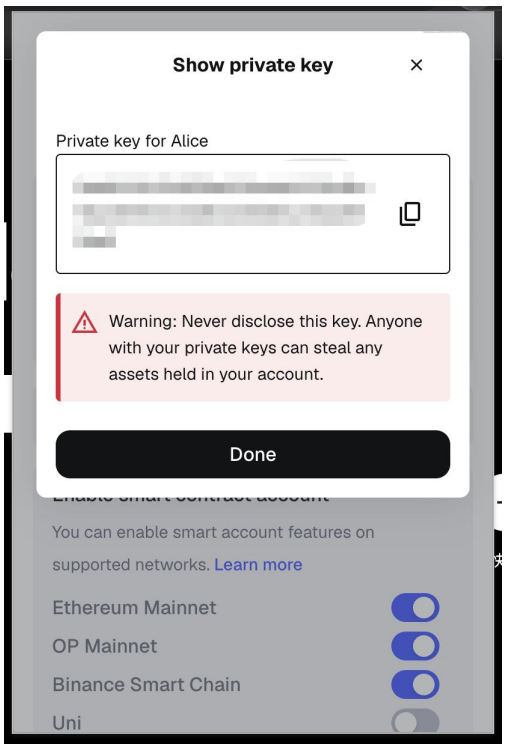




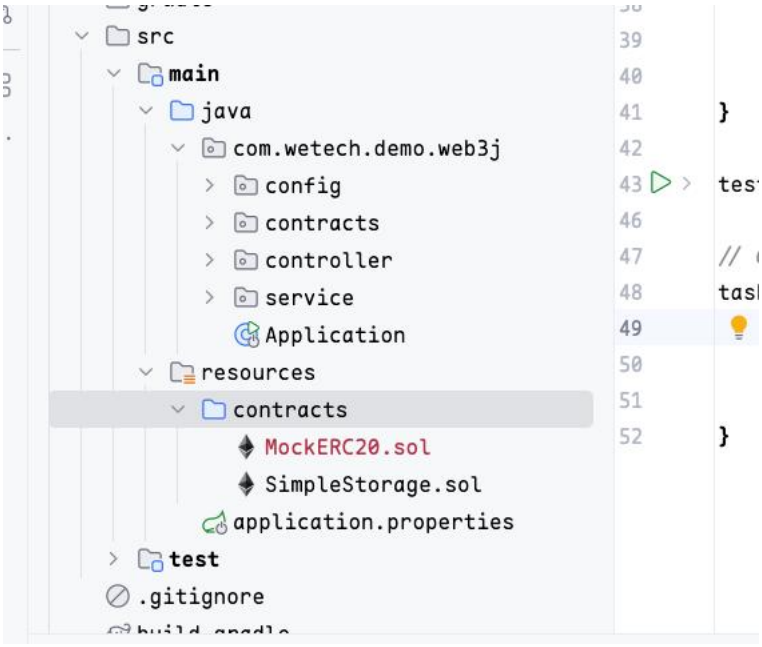



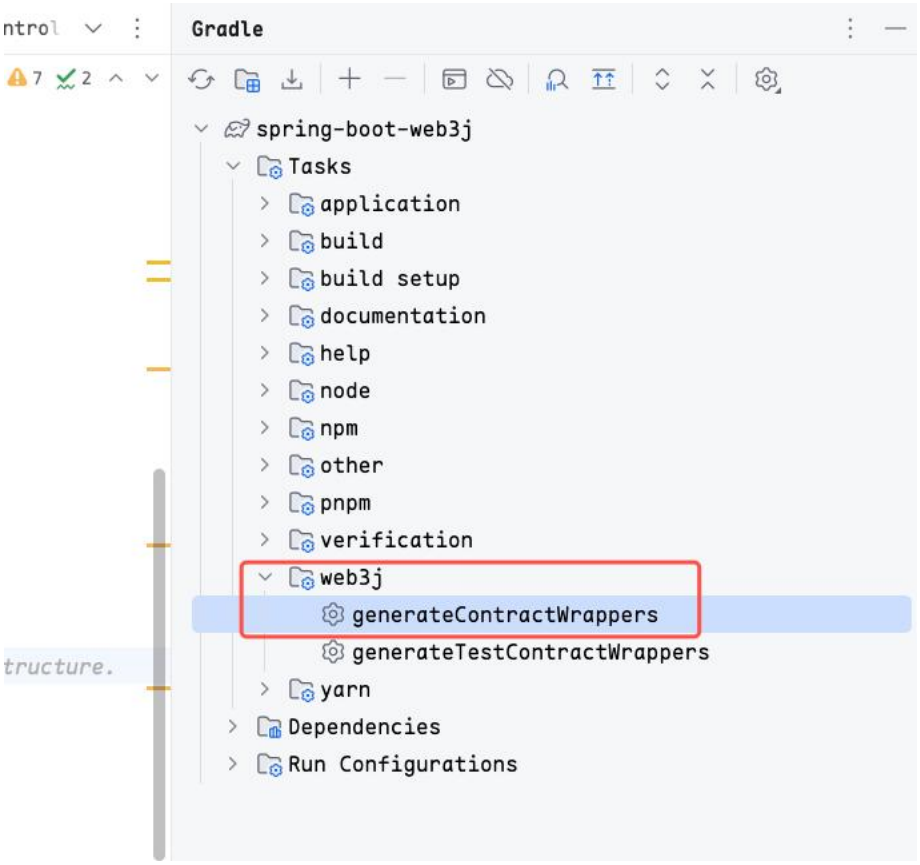

## 2.1.2 将你的智能合约编译成 Java 接口文件

在实验 2 中我们实现了一个 ERC20 合约，现在我们将该合约编译成 Java 接口文件，并集成到 Spring-boot 项目中。




将合约放到 src/resources/contracts 目录下

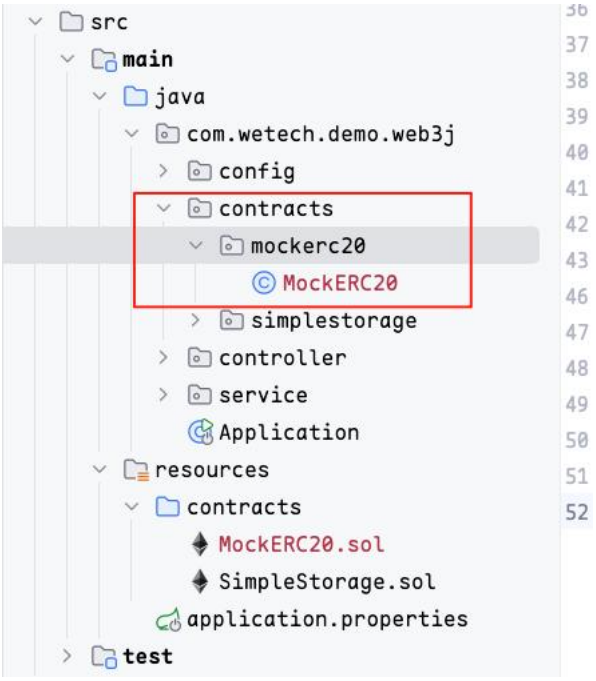
在 IDEA 中执行 gradle 命令：




如果你没有使用 IDEA，可以在命令行中执行命令：

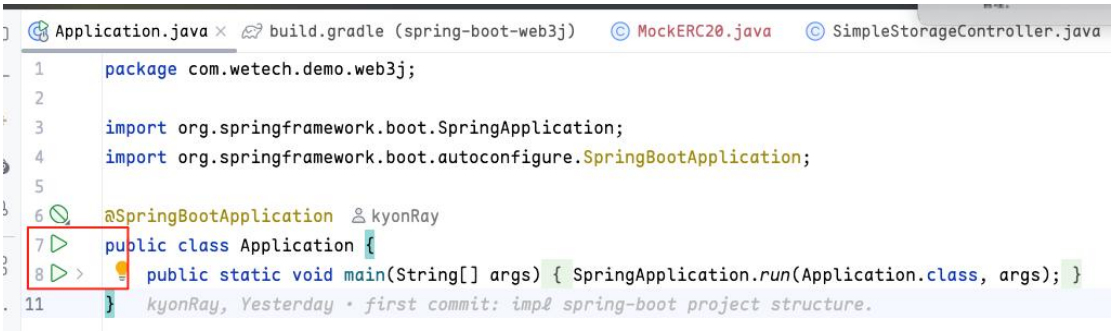
bash gradlew generateContractWrappers

执行成功后，将在 src/main/java/com/wetech/demo/web3j/contracts 目录下生成合约的 Java 接口文件。




## 2.1.3 运行项目并在访问其 HTTP 服务

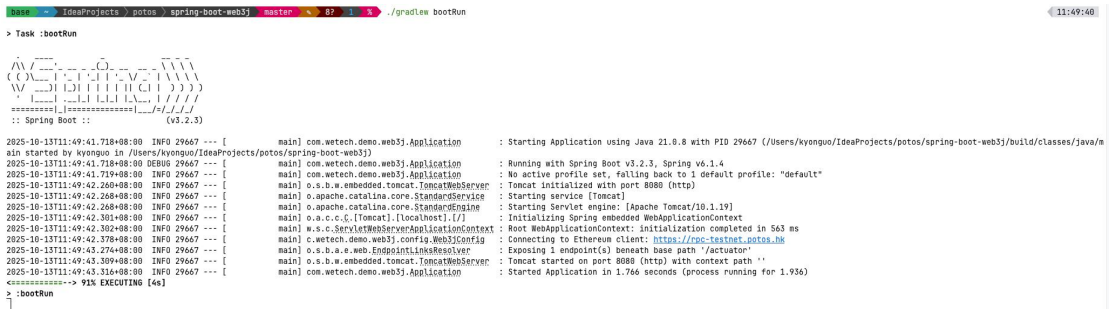
在 IDEA 点击 Application.java 中的按钮即可编译运行服务




也可以在命令行中执行命令运行服务

```
# 编译项目
bash gradlew build

# 运行项目
bash gradlew bootRun
```

运行成功将会有以下输出，服务启动，并监听 8080 端口：




部署合约：
curl -X POST http://localhost:8080/api/storage/deploy

调用合约的 setValue 接口：
curl -X POST http://localhost:8080/api/storage/value/set?value=100

调用合约的 getValue 接口：
curl -X GET http://localhost:8080/api/storage/value/get

## 2.2 构建前端 DApp

前置要求：

- Node.js 18+、pnpm 或 yarn
- 钱包（MetaMask），了解测试网与账户管理

### 2.2.1 下载 my-first-dapp 代码

为了同学们快速上手，提供了 React TS +Next.js + Hardhat + Viem 的示例代码，后续同学们基于该项目仓库修改即可。

# 代码仓库
https://github.com/kyonRay/my-first-dapp

# 代码结构

tree ./packages/ -L 2
./packages/
├── hardhat # 智能合约合约脚手架目录
│ ├── contracts ## 合约目录
│ ├── deploy ## 部署脚本
│ ├── deployments ## 部署结果
│ ├── hardhat.config.ts ## hardhat 配置

```
│   ├── package.json
│   ├── scripts                ## 脚本工具
│   ├── test
│   ├── tsconfig.json
│   └── typechain-types
└── nextjs                  # 前端项目目录
    ├── app                    ## 前端 app 主目录
    ├── components             ## 前端组件
    ├── contracts              ## 前端合约调用 ABI
    ├── hooks                  ## React hooks
    ├── next.config.ts         ## Next.js 配置
    ├── package.json
    ├── postcss.config.js
    ├── scaffold.config.ts     ## 链相关配置
    ├── services
    ├── types
    ├── utils
    └── vercel.json
```

22 directories, 13 files

# 配置文件说明 （hardhat.config.ts）

Hardhat 配置：

```
// 部署合约的私钥

const deployerPrivateKey =
  process.env.__RUNTIME_DEPLOYER_PRIVATE_KEY ??
"0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80";


const config: HardhatUserConfig = {
// solidity 编译器配置
  solidity: {
    compilers: [
      {
        version: "0.8.20",
        settings: {
          optimizer: {
            enabled: true,
            // https://docs.soliditylang.org/en/latest/using-the-compiler.html#optimizer-options
            runs: 200,
          },
        },
      },
```

```
    ],
  },
  // 默认使用网络
  defaultNetwork: "potos_testnet",
  namedAccounts: {
    deployer: {
      // By default, it will take the first Hardhat account as the deployer
      default: 0,
    },
  },
  networks: {
    hardhat: {
      forking: {
        url: `https://eth-mainnet.alchemyapi.io/v2/${providerApiKey}`,
        enabled: process.env.MAINNET_FORKING_ENABLED === "true",
      },
    },
  // bcos 测试链配置
    potos_testnet: {
      allowUnlimitedContractSize: true,
      url: `https://rpc-testnet.potos.hk`,
      chainId: 60600,
      accounts: [deployerPrivateKey],
    },
  },
};
```

# 前端链配置（scaffold.config.ts）

```
const scaffoldConfig = {
  // The networks on which your DApp is live
  // 测试链的配置
  targetNetworks: [
    {
      id: 60600,
      name: "POTOS Testnet",
      nativeCurrency: {
        decimals: 18,
        name: "POTOS Token",
        symbol: "POT",
      },
      rpcUrls: {
        default: { http: ["https://rpc-testnet.potos.hk"] },
      },
```

```
        blockExplorers: {
          default: {
            name: "POTOS Testnet Explorer",
            url: "https://scan-testnet.potos.hk",
          },
        },
      },
  ],
  // The interval at which your front-end polls the RPC servers for new data (it has no effect if you only target the local network (default is 4000))
  pollingInterval: 30000,
  rpcOverrides: {},
  // This is ours WalletConnect's default project ID.
  // You can get your own at https://cloud.walletconnect.com
  // It's recommended to store it in an env variable:
  // .env.local for local testing, and in the Vercel/system env config for live apps.
  walletConnectProjectId: process.env.NEXT_PUBLIC_WALLET_CONNECT_PROJECT_ID || "3a8170812b534d0ff9d794f19a901d64",
  onlyLocalBurnerWallet: true,
} as const satisfies ScaffoldConfig;
```

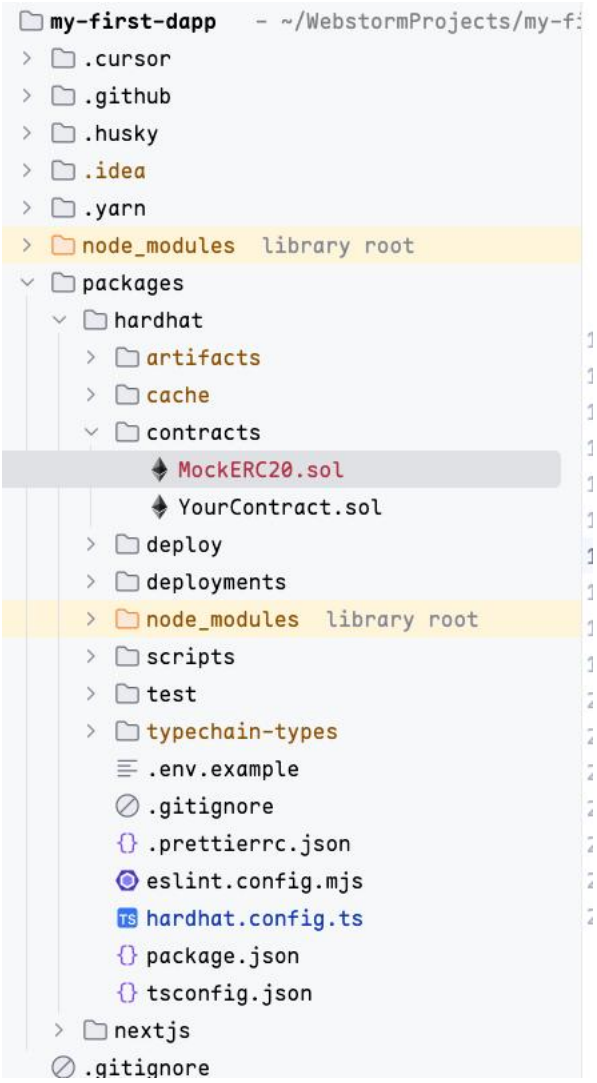
# 编译项目

```
$ yarn install
```

## 2.2.2 编译合约代码

在实验 2 中我们实现了一个 ERC20 合约，现在我们将该合约编译并部署，并集成到前端项目中。

将合约放到 packages/hardhat/contracts/目录下



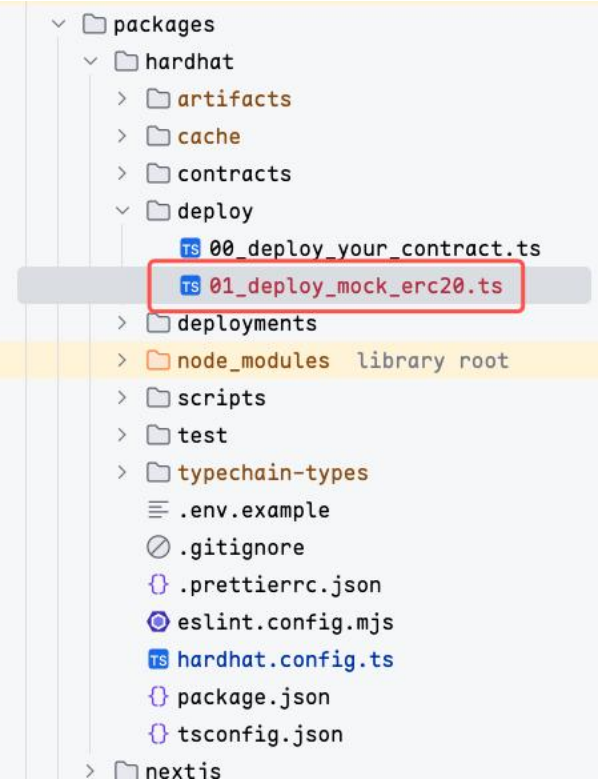

在命令行中执行命令：

```
$ yarn compile
Generating typings for: 8 artifacts in dir: typechain-types for target: ethers-v6
Successfully generated 34 typings!
Compiled 6 Solidity files successfully (evm target: paris).
```

执行成功后将在目录下生成合约编译信息：

### 2.2.3 部署合约到区块链

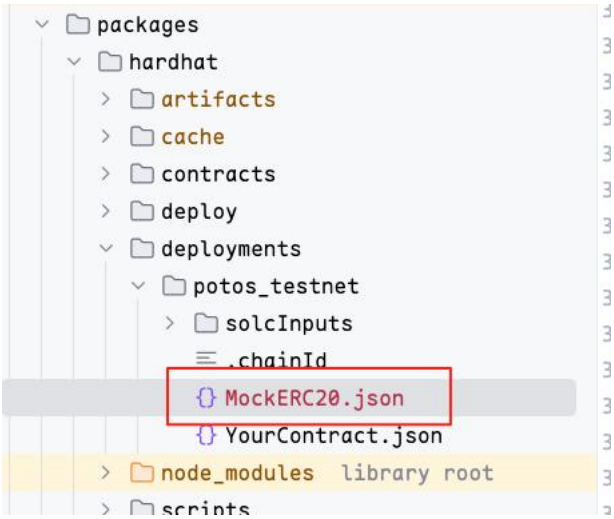
在 packages/hardhat/deploy 目录下放置部署 ts 文件



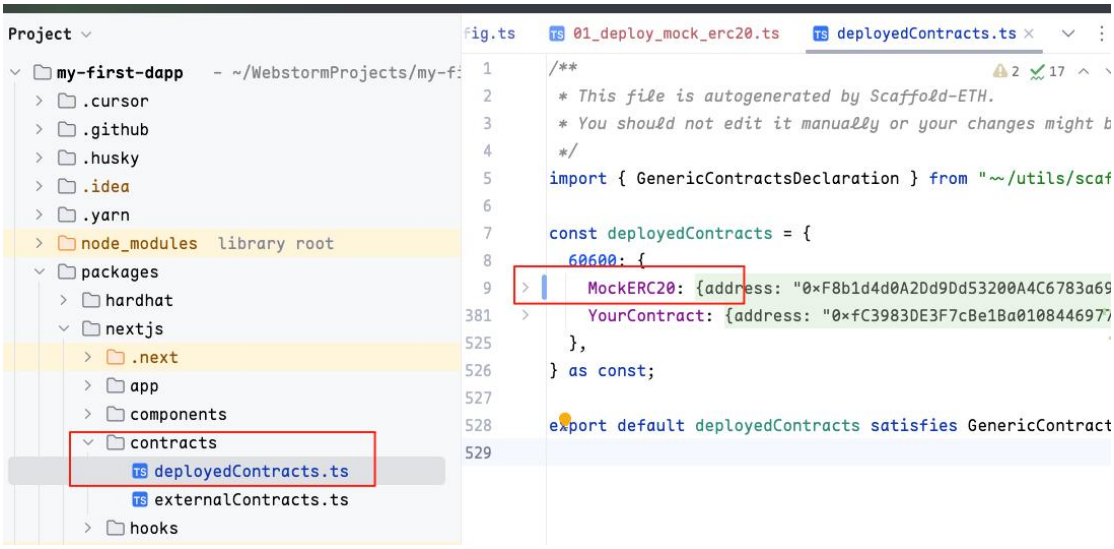

在命令行中输入以下命令即可完成部署，此处的 tags 是可以自定义的。

```
$ yarn deploy --tags MockERC20

Generating typings for: 1 artifacts in dir: typechain-types for target: ethers-v6
Successfully generated 22 typings!
Compiled 1 Solidity file successfully (evm target: paris).
deploying "MockERC20" (tx: 0xca1017cfc9584fc4c11df027008de8b9d8d2ccfe698126422c5fddc9ec436a17)...: deployed at 0xF8b1d4d0A2Dd9Dd53200A4C6783a69c15E3a25F4 with 57503 gas

  Updated TypeScript contract definition file on ../nextjs/contracts/deployedContracts.ts
```

在部署成功后，将会在 packages/hardhat/deployments 目录下生成部署信息，




并且将会自动更改前端的合约配置信息：

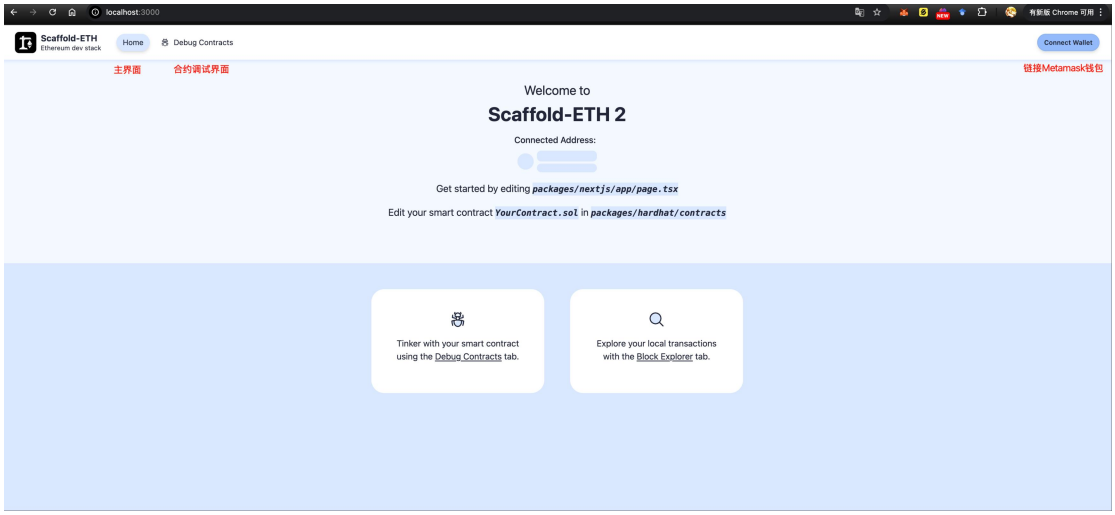



## 2.2.4 在页面上操作智能合约

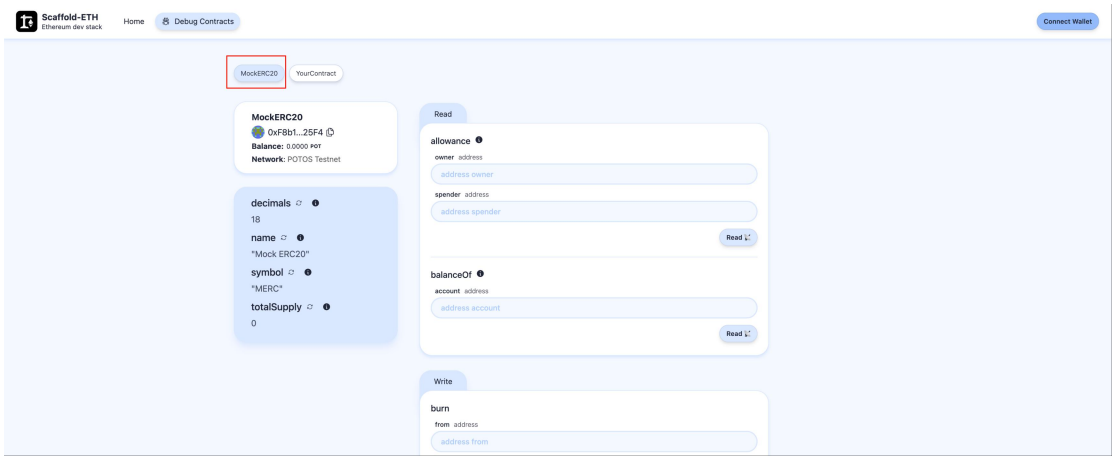
在命令行中输入命令，即可在本地启动 node 服务，可以通过浏览器访问前端。

```
$ yarn start

   ▲ Next.js 15.2.5
   - Local:        http://localhost:3000
   - Network:      http://192.168.3.1:3000

 ✓ Starting...
 ✓ Ready in 2.5s
```

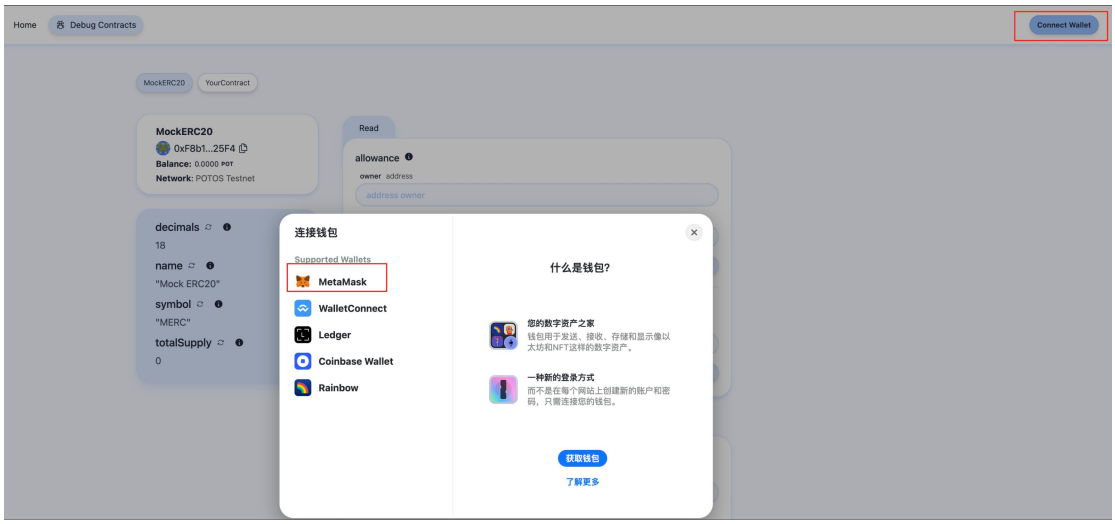
此时在浏览器中访问 http://localhost:3000 就能看到界面



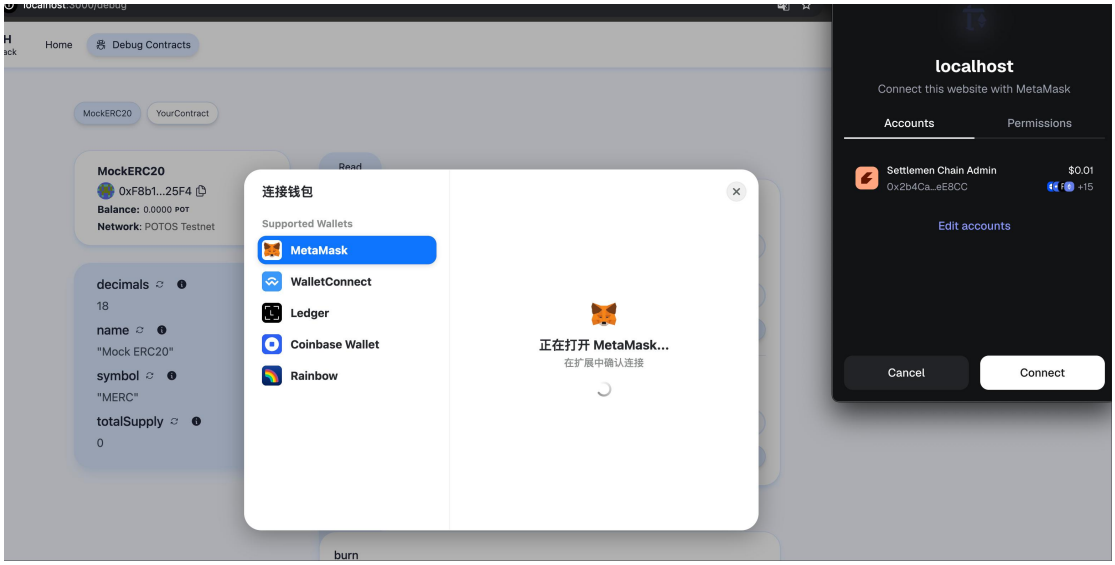

在 Debug Contracts 界面中可以看到在测试网中部署的合约




连接钱包进行调用



将会弹出 MetaMask 的浏览器插件，此时点击连接即可连上你的 DApp。




连上后，右上角将会展示所连接的账号信息。


# 三、大作业任务

根据各个组员的实际情况，分工完成以下任务：

**后台应用：**

- 仿照 service 和 controller 的写法，将你在实验 2 的 ERC20 合约集成到 Spring-boot-web3j 服务中。要求实现 ERC20 的 mint、transfer、balanceOf、approve、transferFrom 的接口
- 在命令行调用服务 HTTP 接口，部署你的 ERC20 合约，调用以上所有实现的接口。

**前端 DAPP：**

- 仿造上述 MockERC20 的做法，将你在实验 2 实现的 ERC20 合约集成到 my-first-dapp 前端项目中。
- 仿造 packages/hardhat/deploy/00_deploy_your_contract.ts 的写法，写一个属于你的 ERC20 合约的 ts 部署脚本。**此处要求部署的 tags 为 ERC20+名字缩写+组长学号，比如张三的学号是 2022001，那么张三在此处的 tags 为：ERC20ZS2022001**
- 通过前端 Debug Contracts 页面操作合约的调用，调用 ERC20 的 mint、transfer、approve、transferFrom 接口。