Solidity智能合约开发知识

第13.1课：Ethers.js基础与MetaMask交互

学习目标：理解Ethers.js与Web3.js的核心差异、掌握Provider连接区块链的各种方法、熟悉读取区块链数据和合约状态、学会监听合约事件、理解MetaMask Provider的工作原理、掌握请求账户连接和网络切换、学会发送交易和签名消息、能够构建完整的DApp前端应用

预计学习时间：4-5小时

难度等级：中级

重要提示：Ethers.js是一个现代化的以太坊JavaScript库，它提供了简洁的API和强大的功能，让我们能够轻松地构建去中心化应用的前端部分。掌握Ethers.js与MetaMask的集成，是构建用户友好的DApp的关键技能。通过本课程的学习，你将能够从前端应用连接区块链、读取数据、发送交易，实现完整的用户交互流程。

目录

---

1. 为什么选择Ethers.js

1.1 Ethers.js的诞生背景

在以太坊生态系统中，前端应用需要与区块链进行交互，这需要一个可靠的JavaScript库来处理各种操作。早期的Web3.js虽然功能完善，但在现代化开发中遇到了一些挑战。随着JavaScript生态系统的发展，开发者对更好的开发体验、更小的包体积、更清晰的API设计有了更高的要求。

Ethers.js应运而生，它由Richard Moore开发，专注于提供现代化、简洁、高效的以太坊交互体验。Ethers.js的设计理念是：**简单、强大、类型安全**。它不仅仅是一个工具库，更是一种全新的开发体验，让开发者能够专注于业务逻辑，而不是处理复杂的API调用。

在实际开发中，开发者经常遇到这样的问题：Web3.js的包体积太大，影响页面加载速度；API设计不够直观，需要处理回调函数；TypeScript支持不够完善，类型推断不够准确；错误信息不够清晰，调试困难。这些问题在大型项目中尤其明显，不仅降低了开发效率，也影响了用户体验。

Ethers.js通过现代化的设计解决了这些问题。它采用纯Promise API，支持async/await语法，代码更加简洁和易读。原生TypeScript支持提供了完整的类型推断，能够在编译时发现更多错误。模块化设计使得包体积更小，支持按需导入，减少打包体积。清晰的错误信息帮助开发者快速定位问题。

1.2 Ethers.js的核心优势

1. 现代化设计：

Ethers.js采用纯Promise API，完全摒弃了回调函数，这意味着你可以直接使用async和await语法，代码更加简洁和现代化。这种设计不仅提高了代码的可读性，也降低了出错的可能性。

原生TypeScript支持是Ethers.js的另一个重要优势。它提供了完整的类型定义，支持类型推断，能够在编译时发现类型错误。这对于大型项目来说非常重要，能够提前发现潜在问题，减少运行时错误。

API设计非常直观，方法名称直接反映了功能。例如，`provider.getBalance()`用于获取余额，`contract.getNumber()`用于调用合约函数。这种设计降低了学习成本，使得开发者能够快速上手。

2. 包体积优势：

Ethers.js采用模块化设计，支持按需导入。这意味着你只需要导入你使用的功能，而不是整个库。这大大减少了打包体积，提升了页面加载速度。

相比Web3.js的约1MB包体积，Ethers.js只有约200KB。这对于前端应用来说是一个明显的优势，特别是在移动端或网络条件较差的环境中，这种体积差异会直接影响用户体验。

模块化设计还带来了更好的Tree-shaking支持。现代打包工具可以自动移除未使用的代码，进一步减少最终打包体积。

3. 开发体验：

Ethers.js提供了清晰的错误信息，每个错误都有详细的说明和建议的修复方法。这对于调试和开发效率提升很大。错误信息不仅包括错误类型，还包括错误位置、可能的原因等。

完善的文档是Ethers.js的另一个优势。官方文档非常详细，包含了大量的示例和说明。社区也非常活跃，有大量的教程和问题解答。

活跃的社区支持确保了Ethers.js能够持续更新和改进。GitHub上有丰富的讨论和问题解答，Stack Overflow上有大量的问答。这种社区支持对于学习和解决问题非常重要。

4. 功能完善：

Ethers.js提供了完善的Provider抽象，支持多种连接方式，包括HTTP、WebSocket、Infura、Alchemy等。这种抽象使得切换不同的连接方式变得非常简单。

丰富的工具函数是Ethers.js的另一个优势。它提供了大量的工具函数，包括地址格式化、单位转换、签名验证等。这些工具函数覆盖了DApp开发中的大部分需求。

强大的签名支持使得Ethers.js能够处理各种签名场景，包括标准消息签名、EIP-191签名、EIP-712结构化签名等。这种全面的签名支持对于构建复杂的DApp非常重要。

1.3 适用场景

Ethers.js特别适合以下场景：

前端DApp开发：如果你正在构建去中心化应用的前端部分，Ethers.js是理想的选择。它提供了完整的区块链交互功能，包括读取数据、发送交易、监听事件等。

现代化JavaScript项目：如果你使用TypeScript、React、Vue等现代化框架，Ethers.js的原生TypeScript支持和现代化API设计能够很好地集成到你的项目中。

性能敏感应用：如果你的应用对性能有严格要求，Ethers.js的小包体积和模块化设计能够帮助你优化应用性能。

需要良好开发体验的项目：如果你重视开发体验，Ethers.js的清晰API、完善文档、活跃社区能够帮助你提高开发效率。

当然，Ethers.js并不是万能的。如果你的项目需要特定的Web3.js功能，或者团队已经熟悉Web3.js，那么继续使用Web3.js也是合理的选择。但总的来说，对于大多数DApp开发场景，Ethers.js都是一个优秀的选择。

---

2. Ethers.js vs Web3.js 对比

2.1 技术架构对比

包大小对比：

特性	Ethers.js	Web3.js
包体积	约200KB	约1MB
模块化	支持按需导入	需要导入整个库
Tree-shaking	完全支持	部分支持

TypeScript支持：

特性	Ethers.js	Web3.js
原生支持	是	否
类型定义	完整	需要额外安装
类型推断	强	中等

2.2 API设计对比

异步处理方式：

Web3.js使用回调与Promise混合的方式，代码可能如下：

```
web3.eth.getBalance(address, (error, result) => {
  if (error) {
    console.error(error);
  } else {
    console.log(result);
  }
});
```

Ethers.js采用纯Promise API，代码更加简洁：

```
const balance = await provider.getBalance(address);
console.log(balance);
```

错误处理：

Web3.js的错误处理需要手动检查，而Ethers.js使用标准的异常机制，可以使用try-catch处理：

```
try {
  const balance = await provider.getBalance(address);
  console.log(balance);
} catch (error) {
  console.error('获取余额失败:', error.message);
}
```

2.3 功能特性对比

Provider抽象：

特性	Ethers.js	Web3.js
Provider抽象	完善	基础
连接方式	多种	多种
切换Provider	简单	需要重新初始化

签名支持：

特性	Ethers.js	Web3.js
标准消息签名	支持	支持
EIP-191签名	支持	需要额外处理
EIP-712签名	原生支持	需要额外库

2.4 选择建议

选择Ethers.js的场景：

- 新项目开发
- 需要现代化API设计
- 重视包体积和性能
- 使用TypeScript
- 需要良好的开发体验

选择Web3.js的场景：

- 已有项目使用Web3.js
- 团队熟悉Web3.js
- 需要特定的Web3.js功能
- 与特定工具集成

两者并存：

实际上，这两个库可以并存，根据项目需求灵活选择。但总的来说，对于新项目，推荐使用Ethers.js，因为它提供了更好的开发体验和性能。

---

3. Provider连接详解

3.1 Provider的概念

Provider是Ethers.js中连接区块链的抽象层。它封装了与区块链节点通信的细节，提供了统一的接口来访问区块链数据。理解Provider的概念对于有效使用Ethers.js非常重要。

Provider的主要作用是提供只读访问区块链的能力。它不能发送交易或签名消息，这些操作需要Signer。这种设计分离了只读操作和可签名操作，使得代码更加清晰和安全。

Ethers.js支持多种类型的Provider，每种都有其适用场景。选择合适的Provider类型对于应用的性能和稳定性非常重要。

3.2 Provider类型

1. JsonRpcProvider：

JsonRpcProvider是最常用的Provider类型，它通过HTTP或HTTPS连接RPC节点。这种连接方式简单可靠，适合大多数场景。

创建JsonRpcProvider非常简单，只需要传入RPC URL：

```
import { ethers } from 'ethers';

const provider = new ethers.JsonRpcProvider('https://mainnet.infura.io/v3/YOUR_API_KEY');
```

JsonRpcProvider支持所有标准的JSON-RPC方法，包括查询余额、获取区块信息、调用合约等。它是最通用的Provider类型，可以与任何兼容的RPC节点连接。

2. WebSocketProvider：

WebSocketProvider通过WebSocket连接RPC节点，支持实时订阅功能。这种连接方式适合需要实时监听区块链事件的场景。

创建WebSocketProvider需要传入WebSocket URL：

```
const provider = new
ethers.WebSocketProvider('wss://mainnet.infura.io/ws/v3/YOUR_API_KEY');
```

WebSocketProvider的优势在于它支持实时事件订阅，不需要轮询。这对于监听新区块、待处理交易等场景非常有用。但需要注意的是，WebSocket连接可能不稳定，需要处理重连逻辑。

3. InfuraProvider和AlchemyProvider：

InfuraProvider和AlchemyProvider是专门为Infura和Alchemy服务设计的Provider。它们提供了更稳定的连接和更好的性能，但需要相应的API密钥。

创建这些Provider非常简单：

```
// InfuraProvider
const provider = new ethers.InfuraProvider('mainnet', 'YOUR_INFURA_API_KEY');

// AlchemyProvider
const provider = new ethers.AlchemyProvider('mainnet', 'YOUR_ALCHEMY_API_KEY');
```

这些Provider的优势在于它们提供了更稳定的连接、更好的性能、更多的功能。它们还支持直接使用网络名称，Ethers.js会自动处理URL的构建。

4. BrowserProvider（MetaMask Provider）：

BrowserProvider是浏览器扩展Provider，用于与用户钱包进行交互。这是我们这节课的重点，它允许DApp与MetaMask等钱包扩展进行交互。

创建BrowserProvider需要传入window.ethereum对象：

```
if (typeof window.ethereum !== 'undefined') {
  const provider = new ethers.BrowserProvider(window.ethereum);
}
```

BrowserProvider的特殊之处在于它需要用户授权才能访问账户信息。它提供了与用户钱包交互的能力，包括请求账户连接、发送交易、签名消息等。

3.3 连接测试网和主网

连接测试网：

连接测试网时，我们可以使用测试网的RPC URL，或者直接使用网络名称。例如，连接Sepolia测试网：

```
// 使用RPC URL
const provider = new ethers.JsonRpcProvider('https://sepolia.infura.io/v3/YOUR_API_KEY');

// 使用InfuraProvider
const provider = new ethers.InfuraProvider('sepolia', 'YOUR_INFURA_API_KEY');
```

测试网的优势在于可以免费获取测试代币，进行开发和测试，而不需要消耗真实的ETH。

连接主网：

连接主网时，我们需要使用主网的RPC URL或Provider：

```
// 使用RPC URL
const provider = new ethers.JsonRpcProvider('https://mainnet.infura.io/v3/YOUR_API_KEY');

// 使用InfuraProvider
const provider = new ethers.InfuraProvider('mainnet', 'YOUR_INFURA_API_KEY');
```

主网连接需要可靠的RPC节点，建议使用Infura、Alchemy等专业服务，它们提供了稳定的连接和良好的性能。

3.4 验证连接

连接Provider后，我们需要验证连接是否正常。Ethers.js提供了几个方法来检查连接状态：

获取区块号：

```
const blockNumber = await provider.getBlockNumber();
console.log('当前区块号:', blockNumber);
```

如果能够成功获取区块号，说明连接正常。如果连接失败，会抛出异常。

获取网络信息：

```
const network = await provider.getNetwork();
console.log('网络名称:', network.name);
console.log('链ID:', network.chainId.toString());
```

获取网络信息可以帮助我们确认Provider是否正确配置，以及当前连接的是哪个网络。

检查连接状态：

```
try {
  const blockNumber = await provider.getBlockNumber();
  console.log('连接成功，当前区块号:', blockNumber);
} catch (error) {
  console.error('连接失败:', error.message);
}
```

通过try-catch可以捕获连接错误，并给出相应的提示。这对于构建用户友好的应用非常重要。

---

4. 读取区块链数据

4.1 区块信息查询

Provider提供了丰富的读取方法，让我们能够获取区块链上的各种信息。区块信息是最基础的数据，了解如何查询区块信息对于理解区块链状态非常重要。

获取最新区块号：

```
const blockNumber = await provider.getBlockNumber();
console.log('最新区块号:', blockNumber);
```

获取最新区块号是最常用的操作之一，它可以用来显示当前区块链的高度，或者作为查询其他数据的参考点。

获取区块详情：

```
// 获取最新区块
const block = await provider.getBlock('latest');
console.log('区块哈希:', block.hash);
console.log('区块号:', block.number);
console.log('时间戳:', new Date(block.timestamp * 1000));
console.log('交易数量:', block.transactions.length);

// 获取特定区块
const specificBlock = await provider.getBlock(1000000);
```

获取区块详情可以让我们了解区块的完整信息，包括区块哈希、时间戳、交易数量等。这对于构建区块浏览器或分析工具非常有用。

获取包含交易的区块：

```
const blockWithTxs = await provider.getBlockWithTransactions('latest');
console.log('区块中的交易:', blockWithTxs.transactions);
```

获取包含交易的区块可以让我们查看区块中的所有交易详情，这对于分析区块内容或构建交易历史功能很有帮助。

4.2 账户信息查询

查询账户余额：

```
const address = '0x742d35Cc6634C0532925a3b844Bc9e7595f0bEb';
const balance = await provider.getBalance(address);
console.log('余额(Wei):', balance.toString());
console.log('余额(ETH):', ethers.formatEther(balance));
```

查询账户余额是最常用的操作之一。Ethers.js返回的是BigNumber类型，我们需要使用formatEther方法将其转换为ETH单位。这对于显示用户余额或检查账户状态非常重要。

查询账户nonce：

```
const nonce = await provider.getTransactionCount(address);
console.log('账户nonce:', nonce);
```

查询账户nonce可以帮助我们了解账户已经发送了多少笔交易。这对于构建交易历史或检查交易状态很有用。

查询合约代码：

```
const code = await provider.getCode(address);
if (code === '0x') {
  console.log('这是一个普通账户');
} else {
  console.log('这是一个合约账户');
  console.log('合约代码长度:', code.length);
}
```

查询合约代码可以帮助我们判断一个地址是普通账户还是合约账户。这对于构建地址分析工具或验证合约部署很有用。

4.3 交易信息查询

获取交易详情:

```
const txHash = '0x...';
const tx = await provider.getTransaction(txHash);
console.log('发送者:', tx.from);
console.log('接收者:', tx.to);
console.log('金额:', ethers.formatEther(tx.value));
console.log('Gas限制:', tx.gasLimit.toString());
console.log('Gas价格:', tx.gasPrice?.toString());
```

获取交易详情可以让我们了解交易的完整信息，包括发送者、接收者、金额、Gas信息等。这对于构建交易浏览器或分析工具非常有用。

获取交易回执:

```
const receipt = await provider.getTransactionReceipt(txHash);
console.log('交易状态:', receipt.status === 1 ? '成功' : '失败');
console.log('区块号:', receipt.blockNumber);
console.log('Gas使用:', receipt.gasUsed.toString());
console.log('日志数量:', receipt.logs.length);
```

获取交易回执可以让我们了解交易的执行结果，包括交易状态、Gas使用情况、事件日志等。这对于验证交易是否成功或分析交易成本很有用。

获取交易结果:

```
const result = await provider.getTransactionResult(txHash);
console.log('交易结果:', result);
```

获取交易结果可以让我们了解交易的执行结果，这对于调用合约函数后获取返回值很有用。

4.4 存储信息查询

读取合约存储:

```
const slot = 0; // 存储槽索引
const value = await provider.getStorage(address, slot);
console.log('存储值:', value);
```

读取合约存储可以帮助我们了解合约的内部状态。这对于调试合约或分析合约状态很有用。但需要注意的是，存储槽的布局取决于合约的实现，需要了解合约的结构才能正确解读。

4.5 工具函数

Ethers.js提供了很多实用的工具函数，帮助我们处理各种数据格式：

单位转换：

```
// ETH转Wei
const wei = ethers.parseEther('1.0');
console.log('1 ETH =', wei.toString(), 'Wei');

// Wei转ETH
const eth = ethers.formatEther(wei);
console.log(wei.toString(), 'Wei =', eth, 'ETH');

// 格式化其他单位
const gwei = ethers.formatUnits(wei, 'gwei');
console.log(wei.toString(), 'Wei =', gwei, 'Gwei');
```

单位转换是DApp开发中的常见需求。Ethers.js提供了parseEther、formatEther、formatUnits等函数，可以方便地进行各种单位转换。

地址格式化：

```
const address = '0x742d35cc6634c0532925a3b844bc9e7595f0beb';
const checksumAddress = ethers.getAddress(address);
console.log('校验和地址:', checksumAddress);
```

地址格式化可以确保地址使用正确的校验和格式，这对于避免地址错误很重要。Ethers.js的getAddress函数会自动处理地址格式转换。

其他工具函数：

```
// 计算地址
const address = ethers.computeAddress(privateKey);

// 验证地址格式
const isValid = ethers.isAddress(address);

// 格式化单位
const formatted = ethers.formatUnits(value, decimals);
```

这些工具函数覆盖了DApp开发中的大部分需求，能够帮助我们处理各种数据格式和计算。

---

5. 读取合约状态

5.1 创建合约实例

与智能合约交互是DApp开发的核心功能。在Ethers.js中，我们需要创建Contract实例来与合约进行交互。理解如何创建和使用合约实例对于构建DApp非常重要。

创建合约实例：

创建合约实例需要三个参数：合约地址、ABI（应用二进制接口），以及Provider或Signer。

```
import { ethers } from 'ethers';

const contractAddress = '0x...';
const contractABI = [
  "function getNumber() view returns (uint256)",
  "function increment()",
  "function setNumber(uint256)",
  "event Incremented(address indexed user, uint256 newValue)"
];

// 使用Provider创建只读合约实例
const contract = new ethers.Contract(contractAddress, contractABI, provider);

// 使用Signer创建可签名合约实例
const contractWithSigner = new ethers.Contract(contractAddress, contractABI, signer);
```

ABI定义了合约的接口，告诉Ethers.js如何调用合约函数。ABI可以从合约编译后的JSON文件中获取，也可以手动定义。

ABI的作用：

ABI是应用二进制接口的缩写，它定义了合约的函数签名、事件定义、错误定义等。Ethers.js使用ABI来编码函数调用和解码返回值，这使得与合约交互变得非常简单。

ABI通常包含以下信息：

- 函数名称和参数类型
- 返回值类型
- 事件名称和参数
- 错误定义

5.2 只读调用

只读调用用于调用view或pure函数。这些函数不修改合约状态，不消耗Gas，不需要签名，可以立即返回结果。

调用view函数：

```
// 调用无参数函数
const number = await contract.getNumber();
console.log('当前值:', number.toString());

// 调用带参数函数
const balance = await contract.balanceOf(address);
console.log('余额:', balance.toString());
```

只读调用非常简单，直接调用函数即可。Ethers.js会自动处理编码和解码，返回的结果可以直接使用。

批量读取数据：

为了提高效率，我们可以使用Promise.all批量读取多个数据：

```
const [name, symbol, decimals, totalSupply] = await Promise.all([
  contract.name(),
  contract.symbol(),
  contract.decimals(),
  contract.totalSupply()
]);

console.log('代币名称:', name);
console.log('代币符号:', symbol);
console.log('小数位数:', decimals);
console.log('总供应量:', totalSupply.toString());
```

批量读取可以并行执行多个调用，减少等待时间，提高应用性能。这对于需要同时显示多个数据的场景非常有用。

5.3 处理复杂返回值

处理结构体返回值：

如果函数返回结构体，我们可以直接访问结构体的各个字段：

```
const userInfo = await contract.getUserInfo(address);
console.log('用户名:', userInfo.name);
console.log('余额:', userInfo.balance.toString());
console.log('注册时间:', new Date(userInfo.registeredAt * 1000));
```

Ethers.js会自动将结构体转换为对象，我们可以直接访问各个字段。

处理数组返回值：

如果函数返回数组，我们可以遍历数组处理每个元素：

```
const users = await contract.getAllUsers();
users.forEach((user, index) => {
  console.log(`用户 ${index + 1}:`, user);
});
```

数组返回值可以直接遍历，Ethers.js会自动处理数组的解码。

5.4 错误处理

当合约调用失败时，Ethers.js会抛出异常。我们需要正确处理这些异常，以提供良好的用户体验。

捕获调用异常：

```
try {
  const number = await contract.getNumber();
  console.log('当前值:', number.toString());
} catch (error) {
  if (error.code === 'CALL_EXCEPTION') {
    console.error('合约调用失败:', error.message);
  } else {
    console.error('其他错误:', error.message);
  }
}
```

CALL_EXCEPTION表示合约调用失败，可能是函数不存在、参数错误或合约状态不允许该操作。我们需要根据不同的错误类型给出相应的提示。

常见错误类型：

- CALL_EXCEPTION：合约调用失败
- INVALID_ARGUMENT：参数错误
- NETWORK_ERROR：网络错误
- UNPREDICTABLE_GAS_LIMIT：Gas估算失败

理解这些错误类型可以帮助我们更好地处理异常情况，提供更好的用户体验。

---

6. 监听事件

6.1 事件监听基础

智能合约通过事件来通知外部应用状态变化，Ethers.js提供了强大的事件监听功能。理解如何监听和处理事件对于构建响应式的DApp非常重要。

事件监听有多种方式：

- 合约事件监听：使用on方法持续监听，once方法监听一次，queryFilter方法查询历史事件
- 区块事件监听：监听新区块和待处理交易
- Provider事件：监听网络变化和错误

6.2 监听合约事件

持续监听事件：

```
contract.on('Incremented', (user, newValue, event) => {
  console.log('Incremented事件触发:');
  console.log('  用户:', user);
  console.log('  新值:', newValue.toString());
  console.log('  区块号:', event.log.blockNumber);
  console.log('  交易哈希:', event.log.transactionHash);
});
```

使用on方法可以持续监听事件，每当事件触发时，回调函数会被调用。这对于构建实时更新的应用非常有用。

添加事件过滤器：

```
// 只监听发送到特定地址的Transfer事件
contract.on('Transfer', (from, to, value, event) => {
  if (to === targetAddress) {
    console.log('收到转账:', ethers.formatEther(value), 'ETH');
  }
});
```

添加过滤器可以只监听特定条件的事件，这对于构建通知系统或过滤特定数据非常有用。

监听一次事件：

```
contract.once('Incremented', (user, newValue, event) => {
  console.log('Incremented事件触发一次');
});
```

使用once方法可以只监听一次事件，这在某些场景下可以避免重复处理。

6.3 查询历史事件

查询最近N个区块的事件：

```
const filter = contract.filters.Incremented();
const events = await contract.queryFilter(filter, -5); // 最近5个区块
console.log('找到', events.length, '个Incremented事件');
events.forEach((event, index) => {
  console.log(`事件 ${index + 1}:`, {
    user: event.args.user,
    newValue: event.args.newValue.toString(),
    blockNumber: event.blockNumber
  });
});
```

查询历史事件可以让我们获取过去发生的事件，这对于构建交易历史或分析功能很有帮助。

查询指定区块范围内的事件：

```
const events = await contract.queryFilter(filter, 1000000, 1000100);
```

查询指定区块范围内的事件可以让我们获取特定时间段的事件，这对于分析或审计很有用。

使用过滤器查询：

```
// 查询特定地址相关的事件
const filter = contract.filters.Transfer(null, targetAddress);
const events = await contract.queryFilter(filter);
```

使用过滤器可以查询特定条件的事件，这对于构建地址相关的功能很有帮助。

6.4 监听区块事件

监听新区块：

```
provider.on('block', (blockNumber) => {
  console.log('新区块:', blockNumber);
});
```

监听新区块可以让我们实时了解区块链的状态，这对于更新区块高度显示或触发某些操作很有用。

监听待处理交易：

```
provider.on('pending', (txHash) => {
  console.log('待处理交易:', txHash);
});
```

监听待处理交易可以让我们了解交易池的状态，这对于显示交易状态或分析交易很有用。

6.5 监听网络变化

监听网络切换：

```
provider.on('network', (newNetwork, oldNetwork) => {
  console.log('网络切换:', oldNetwork?.name, '->', newNetwork.name);
  // 重新初始化应用状态
});
```

监听网络变化很重要。当用户在MetaMask中切换网络时，我们需要更新应用状态，可能需要重新加载页面或重新初始化Provider。

6.6 事件监听最佳实践

添加错误处理：

```
contract.on('error', (error) => {
  console.error('事件监听错误:', error);
});
```

添加错误处理监听器可以捕获事件监听过程中的错误，确保应用的稳定性。

清理监听器：

```
// 在组件卸载时清理监听器
contract.removeAllListeners('Incremented');
// 或清理所有监听器
contract.removeAllListeners();
```

在组件卸载时清理监听器可以避免内存泄漏，这对于长期运行的应用非常重要。

使用once方法避免重复监听：

```
// 避免重复监听
if (!listenerAdded) {
  contract.once('Incremented', handleEvent);
  listenerAdded = true;
}
```

使用once方法可以避免重复监听，这对于某些只需要处理一次的场景很有用。

---

7. MetaMask Provider

7.1 MetaMask简介

MetaMask是浏览器钱包扩展，它为用户提供了安全的钱包管理功能。理解MetaMask的工作原理对于构建与用户钱包交互的DApp非常重要。

MetaMask的主要特点包括：

用户钱包管理：

- 管理多个账户
- 支持多个网络
- 私钥安全存储在本地

交易签名功能：

- 用户需要确认每笔交易
- 采用安全的签名机制
- 无需暴露私钥

网络切换功能：

- 支持多链
- 用户可以手动切换
- 应用也可以请求切换

7.2 检测MetaMask

使用MetaMask之前，我们需要先检测MetaMask是否安装。检测很简单，我们只需要检查window.ethereum对象是否存在。

```
if (typeof window.ethereum !== 'undefined') {
  console.log('MetaMask已安装');
} else {
  console.log('请安装MetaMask');
  // 引导用户安装MetaMask
  window.open('https://metamask.io/download/', '_blank');
}
```

如果MetaMask未安装，我们应该引导用户安装。可以提供安装链接或显示提示信息。

7.3 创建MetaMask Provider

创建MetaMask Provider时，我们使用BrowserProvider类，传入window.ethereum对象。

```
if (typeof window.ethereum !== 'undefined') {
  const provider = new ethers.BrowserProvider(window.ethereum);
}
```

BrowserProvider是Ethers.js v6中的新名称，在v5中称为Web3Provider。创建Provider后，我们可以使用它来获取Signer和执行各种操作。

7.4 获取Signer

获取Signer需要用户先连接账户，然后我们可以通过getSigner方法获取Signer实例。

```
// 请求用户连接账户
await window.ethereum.request({ method: 'eth_requestAccounts' });

// 创建Provider
const provider = new ethers.BrowserProvider(window.ethereum);

// 获取Signer
const signer = await provider.getSigner();

// 获取当前账户地址
const address = await signer.getAddress();
console.log('当前账户:', address);
```

Signer用于可签名操作，比如发送交易、签名消息等。只有通过Signer才能执行需要用户授权的操作。

7.5 Provider和Signer的区别

理解Provider和Signer的区别非常重要：

Provider用于只读操作：

- 查询余额
- 获取区块号
- 读取合约状态
- 查询交易信息

Signer用于可签名操作：

- 发送交易
- 签名消息
- 调用状态修改函数
- 部署合约

这种设计分离了只读操作和可签名操作，使得代码更加清晰和安全。只有通过Signer才能执行需要用户授权的操作。

7.6 监听账户变化

监听账户变化很重要。当用户在MetaMask中切换账户时，accountsChanged事件会被触发，我们需要更新应用状态。

```
window.ethereum.on('accountsChanged', (accounts) => {
  if (accounts.length === 0) {
    console.log('用户断开连接');
    // 清空应用状态
  } else {
    console.log('账户切换:', accounts[0]);
    // 更新应用状态
    updateAccount(accounts[0]);
  }
});
```

当用户断开连接时，accounts数组为空，我们也需要相应处理。这通常意味着用户关闭了MetaMask或断开了连接。

7.7 监听网络切换

监听网络切换也很重要。当用户在MetaMask中切换网络时，chainChanged事件会被触发，我们通常需要重新加载页面或重新创建Provider。

```
window.ethereum.on('chainChanged', (chainId) => {
  console.log('网络切换，链ID:', chainId);
  // 重新加载页面或重新创建Provider
  window.location.reload();
});
```

重新加载页面可以确保所有状态都与新网络一致，这是最简单可靠的方式。也可以选择重新创建Provider和更新应用状态，但这需要更仔细的状态管理。

---

8. 请求账户连接

8.1 账户连接流程

请求用户连接账户是DApp的第一步，这个过程需要用户的明确授权。理解账户连接的流程对于构建用户友好的DApp非常重要。

账户连接的流程包括几个步骤：

1. 检测MetaMask是否安装：

- 检查window.ethereum是否存在
- 检查是否已经连接

2. 请求连接：

- 调用eth_requestAccounts方法
- 用户需要在MetaMask中确认授权

3. 获取账户：

- 获取当前账户地址
- 创建Signer实例

4. 错误处理：

- 处理用户拒绝
- 处理MetaMask未安装
- 处理网络错误

8.2 基础连接代码

基础连接代码很简单：

```
async function connectWallet() {
  // 检查MetaMask是否安装
  if (typeof window.ethereum === 'undefined') {
    alert('请安装MetaMask');
    return;
  }

  try {
    // 请求连接
    await window.ethereum.request({ method: 'eth_requestAccounts' });

    // 创建Provider
    const provider = new ethers.BrowserProvider(window.ethereum);

    // 获取Signer
    const signer = await provider.getSigner();

    // 获取账户地址
    const address = await signer.getAddress();
    console.log('已连接账户:', address);

    return { provider, signer, address };
  } catch (error) {
    if (error.code === 4001) {
      console.log('用户拒绝连接');
    } else {
```

```
      console.error('连接失败:', error);
    }
  }
}
```

调用eth_requestAccounts会弹出MetaMask的授权窗口。用户确认后，我们可以创建Provider和Signer，获取账户地址。

8.3 检查已连接账户

除了主动请求连接，我们还可以检查已连接账户。我们可以使用eth_accounts方法，这个方法不会弹出窗口，只会返回已经授权的账户。

```
async function checkConnection() {
  if (typeof window.ethereum === 'undefined') {
    return false;
  }

  try {
    const accounts = await window.ethereum.request({ method: 'eth_accounts' });
    if (accounts.length > 0) {
      console.log('已连接账户:', accounts[0]);
      return true;
    } else {
      console.log('未连接账户');
      return false;
    }
  } catch (error) {
    console.error('检查连接失败:', error);
    return false;
  }
}
```

如果返回的数组不为空，说明用户已经连接过。我们可以直接使用这些账户，而不需要再次请求授权。

8.4 React组件示例

在实际应用中，我们通常需要创建一个完整的连接组件。在React中，我们可以使用useState管理连接状态，使用useEffect在组件挂载时检查连接状态，并设置事件监听器。

```
import { useState, useEffect } from 'react';
import { ethers } from 'ethers';

function WalletConnection() {
  const [account, setAccount] = useState(null);
  const [provider, setProvider] = useState(null);
  const [signer, setSigner] = useState(null);

  useEffect(() => {
    // 检查连接状态
```

```
    checkConnection();

    // 设置事件监听器
    if (window.ethereum) {
      window.ethereum.on('accountsChanged', handleAccountsChanged);
      window.ethereum.on('chainChanged', handleChainChanged);
    }

    return () => {
      // 清理事件监听器
      if (window.ethereum) {
        window.ethereum.removeListener('accountsChanged', handleAccountsChanged);
        window.ethereum.removeListener('chainChanged', handleChainChanged);
      }
    };
  }, []);

  const checkConnection = async () => {
    if (typeof window.ethereum === 'undefined') {
      return;
    }

    try {
      const accounts = await window.ethereum.request({ method: 'eth_accounts' });
      if (accounts.length > 0) {
        await connectWallet();
      }
    } catch (error) {
      console.error('检查连接失败:', error);
    }
  };

  const connectWallet = async () => {
    try {
      await window.ethereum.request({ method: 'eth_requestAccounts' });
      const provider = new ethers.BrowserProvider(window.ethereum);
      const signer = await provider.getSigner();
      const address = await signer.getAddress();

      setProvider(provider);
      setSigner(signer);
      setAccount(address);
    } catch (error) {
      if (error.code === 4001) {
        alert('用户拒绝连接');
      } else {
        console.error('连接失败:', error);
      }
    }
  };

  const handleAccountsChanged = (accounts) => {
```

```
    if (accounts.length === 0) {
      setAccount(null);
      setProvider(null);
      setSigner(null);
    } else {
      connectWallet();
    }
  };

  const handleChainChanged = () => {
    window.location.reload();
  };

  return (
    <div>
      {account ? (
        <div>已连接: {account}</div>
      ) : (
        <button onClick={connectWallet}>连接钱包</button>
      )}
    </div>
  );
}
```

这个组件展示了完整的连接逻辑，包括检查连接状态、请求连接、处理账户变化等。

8.5 错误处理

错误处理很重要。用户可能拒绝连接，错误代码是4001。连接请求可能正在进行中，错误代码是-32002。我们需要根据不同的错误代码给出相应的提示。

```
try {
  await window.ethereum.request({ method: 'eth_requestAccounts' });
} catch (error) {
  if (error.code === 4001) {
    // 用户拒绝连接
    alert('请连接钱包以继续使用');
  } else if (error.code === -32002) {
    // 连接请求正在进行中
    alert('连接请求正在进行中，请稍候');
  } else {
    // 其他错误
    console.error('连接失败:', error);
    alert('连接失败，请重试');
  }
}
```

根据不同的错误代码给出相应的提示，可以提供更好的用户体验。

---

9. 切换网络

9.1 网络切换方式

网络切换是DApp开发中经常遇到的需求，Ethers.js提供了灵活的网络管理功能。理解如何切换网络对于构建多网络应用非常重要。

网络切换有两种方式：

用户手动切换：

- 用户在MetaMask中切换网络
- 应用监听chainChanged事件
- 自动更新Provider

程序请求切换：

- 应用调用wallet_switchEthereumChain方法
- 如果网络不存在则添加
- 用户确认后切换

9.2 获取当前网络

在切换网络之前，我们需要先获取当前网络信息。获取很简单，我们可以通过Provider的getNetwork方法获取网络信息。

```
const network = await provider.getNetwork();
console.log('网络名称:', network.name);
console.log('链ID:', network.chainId.toString());
```

我们也可以通过window.ethereum的eth_chainId方法获取链ID，注意返回的是十六进制格式。

```
const chainId = await window.ethereum.request({ method: 'eth_chainId' });
console.log('链ID(十六进制):', chainId);
console.log('链ID(十进制):', parseInt(chainId, 16));
```

9.3 监听网络变化

监听网络变化时，我们需要监听chainChanged事件。当网络切换时，我们需要重新创建Provider，更新应用状态。

```
window.ethereum.on('chainChanged', (chainId) => {
  console.log('网络切换，链ID:', chainId);
  // 重新创建Provider
  const provider = new ethers.BrowserProvider(window.ethereum);
  // 更新应用状态
  updateProvider(provider);
  // 或者重新加载页面
  window.location.reload();
});
```

重新加载页面可以确保所有状态都与新网络一致，这是最简单可靠的方式。

9.4 请求切换网络

请求切换网络时，我们调用wallet_switchEthereumChain方法，传入目标网络的链ID。

```
async function switchNetwork(chainId) {
  try {
    await window.ethereum.request({
      method: 'wallet_switchEthereumChain',
      params: [{ chainId: chainId }]
    });
    console.log('网络切换成功');
  } catch (error) {
    if (error.code === 4902) {
      // 网络不存在，需要添加
      console.log('网络不存在，需要添加');
      await addNetwork(chainId);
    } else {
      console.error('切换网络失败:', error);
    }
  }
}
```

如果网络不存在，错误代码是4902，我们需要先添加网络。

9.5 添加网络

添加网络时，我们需要提供完整的网络配置，包括链ID、网络名称、原生货币信息、RPC URL和区块浏览器URL。

```
async function addNetwork(chainId) {
  const networkConfig = {
    chainId: chainId,
    chainName: 'Sepolia Test Network',
    nativeCurrency: {
      name: 'SepoliaETH',
      symbol: 'SEP',
      decimals: 18
    },
    rpcUrls: ['https://sepolia.infura.io/v3/YOUR_API_KEY'],
    blockExplorerUrls: ['https://sepolia.etherscan.io']
  };

  try {
    await window.ethereum.request({
      method: 'wallet_addEthereumChain',
      params: [networkConfig]
    });
    console.log('网络添加成功');
  } catch (error) {
    console.error('添加网络失败:', error);
```

```
  }
}
```

不同的网络有不同的配置，我们需要为每个网络准备相应的配置。常见的网络配置包括主网、Sepolia测试网、Goerli测试网等。

9.6 网络管理封装

为了更方便地管理网络，我们可以将完整的网络管理封装成一个类。NetworkManager类可以管理支持的网络列表，提供获取当前网络、切换网络、检查网络是否支持等方法。

```
class NetworkManager {
  constructor() {
    this.supportedNetworks = {
      1: { name: 'Ethereum Mainnet', symbol: 'ETH' },
      11155111: { name: 'Sepolia Testnet', symbol: 'SEP' },
      31337: { name: 'Hardhat Network', symbol: 'ETH' }
    };
  }

  async getCurrentNetwork() {
    if (typeof window.ethereum === 'undefined') {
      throw new Error('MetaMask未安装');
    }

    const chainId = await window.ethereum.request({ method: 'eth_chainId' });
    return parseInt(chainId, 16);
  }

  isNetworkSupported(chainId) {
    return chainId in this.supportedNetworks;
  }

  async switchNetwork(chainId) {
    try {
      await window.ethereum.request({
        method: 'wallet_switchEthereumChain',
        params: [{ chainId: `0x${chainId.toString(16)}` }]
      });
    } catch (error) {
      if (error.code === 4902) {
        await this.addNetwork(chainId);
      } else {
        throw error;
      }
    }
  }

  async addNetwork(chainId) {
    const config = this.getNetworkConfig(chainId);
    await window.ethereum.request({
```

```
      method: 'wallet_addEthereumChain',
      params: [config]
    });
  }

  getNetworkConfig(chainId) {
    // 返回网络配置
    // ...
  }
}
```

使用NetworkManager时，我们可以先检查当前网络，然后切换到目标网络。如果网络不存在，NetworkManager会自动尝试添加网络。这种封装让网络管理变得更加简单和可靠。

---

10. 发送交易

10.1 交易类型

发送交易是DApp的核心功能，Ethers.js提供了完整的交易发送和跟踪功能。理解如何发送交易对于构建交互式DApp非常重要。

Ethers.js支持多种类型的交易：

ETH转账：

- 发送原生代币
- 指定接收地址和金额
- 最简单的交易类型

合约调用：

- 调用合约函数
- 修改合约状态
- 需要合约地址和ABI

合约部署：

- 部署新合约
- 传递构造函数参数
- 返回合约地址

10.2 发送ETH转账

发送ETH转账很简单。我们通过Signer的sendTransaction方法发送交易，指定接收地址和金额。

```
async function sendETH(to, amount) {
  try {
    const tx = await signer.sendTransaction({
      to: to,
      value: ethers.parseEther(amount.toString())
    });
```

```
    console.log('交易已发送，哈希:', tx.hash);

    // 等待交易确认
    const receipt = await tx.wait();
    console.log('交易确认，区块号:', receipt.blockNumber);
    console.log('Gas使用:', receipt.gasUsed.toString());

    return receipt;
  } catch (error) {
    console.error('发送交易失败:', error);
    throw error;
  }
}
```

交易发送后会返回交易对象，包含交易哈希。我们可以通过wait方法等待交易确认，确认后会返回交易回执，包含区块号和Gas使用情况。

10.3 调用合约函数

调用合约函数时，我们需要创建带Signer的合约实例。然后直接调用合约函数，Ethers.js会自动处理编码和发送。

```
// 创建带Signer的合约实例
const contract = new ethers.Contract(contractAddress, contractABI, signer);

// 调用状态修改函数
async function increment() {
  try {
    const tx = await contract.increment();
    console.log('交易哈希:', tx.hash);

    const receipt = await tx.wait();
    console.log('交易确认，区块号:', receipt.blockNumber);

    return receipt;
  } catch (error) {
    console.error('调用失败:', error);
    throw error;
  }
}

// 调用带参数的函数
async function setNumber(value) {
  try {
    const tx = await contract.setNumber(value);
    const receipt = await tx.wait();
    return receipt;
  } catch (error) {
    console.error('调用失败:', error);
    throw error;
  }
```

```
}
```

对于状态修改函数，调用会返回交易对象，我们需要等待确认。对于带多个参数的函数，我们按顺序传入参数即可。

10.4 交易选项配置

在发送交易之前，交易选项配置很重要。我们可以配置Gas限制和Gas价格。

Legacy交易：

```
const tx = await signer.sendTransaction({
  to: address,
  value: ethers.parseEther('1.0'),
  gasLimit: 21000,
  gasPrice: ethers.parseUnits('20', 'gwei')
});
```

对于Legacy交易，我们使用gasPrice指定Gas价格。

EIP-1559交易：

```
const tx = await signer.sendTransaction({
  to: address,
  value: ethers.parseEther('1.0'),
  gasLimit: 21000,
  maxFeePerGas: ethers.parseUnits('30', 'gwei'),
  maxPriorityFeePerGas: ethers.parseUnits('2', 'gwei')
});
```

对于EIP-1559交易，我们使用maxFeePerGas和maxPriorityFeePerGas指定Gas费用。

10.5 Gas估算

我们可以通过estimateGas方法估算Gas消耗，这对于优化交易成本很有帮助。

```
async function estimateGas() {
  try {
    const gasEstimate = await contract.increment.estimateGas();
    console.log('估算Gas:', gasEstimate.toString());

    // 增加10%的缓冲
    const gasLimit = gasEstimate * 110n / 100n;

    const tx = await contract.increment({ gasLimit });
    return tx;
  } catch (error) {
    console.error('Gas估算失败:', error);
    throw error;
  }
```

```
}
```

Gas估算可以帮助我们了解交易的成本，并设置合适的Gas限制。建议在估算的基础上增加一定的缓冲，以确保交易能够成功执行。

10.6 交易状态跟踪

交易状态跟踪可以帮助我们了解交易的执行情况。我们可以跟踪交易从发送到确认的整个过程。

```
async function trackTransaction(txHash) {
  const provider = signer.provider;

  // 获取交易详情
  const tx = await provider.getTransaction(txHash);
  console.log('交易状态：待确认');

  // 等待确认
  const receipt = await provider.waitForTransaction(txHash);

  if (receipt.status === 1) {
    console.log('交易成功');
    console.log('区块号:', receipt.blockNumber);
    console.log('Gas使用:', receipt.gasUsed.toString());
  } else {
    console.log('交易失败');
  }

  return receipt;
}
```

跟踪交易可以让我们了解交易的执行情况，这对于构建交易历史功能或给用户反馈很有用。

10.7 错误处理

错误处理很重要。用户可能拒绝交易，错误代码是4001。余额可能不足，错误代码是INSUFFICIENT_FUNDS。Gas估算可能失败，错误代码是UNPREDICTABLE_GAS_LIMIT，这通常意味着交易可能会失败。

```
try {
  const tx = await contract.increment();
  const receipt = await tx.wait();
  return receipt;
} catch (error) {
  if (error.code === 4001) {
    alert('用户拒绝交易');
  } else if (error.code === 'INSUFFICIENT_FUNDS') {
    alert('余额不足，请充值');
  } else if (error.code === 'UNPREDICTABLE_GAS_LIMIT') {
    alert('交易可能失败，请检查参数');
  } else {
    console.error('交易失败:', error);
```

```
    alert('交易失败，请重试');
  }
  throw error;
}
```

根据不同的错误给出相应的提示，可以提供更好的用户体验。

10.8 批量交易

批量交易时，我们可以依次发送多个交易，每个交易都需要等待确认。

```
async function batchTransactions(operations) {
  const results = [];

  for (const operation of operations) {
    try {
      const tx = await operation();
      const receipt = await tx.wait();
      results.push({ success: true, receipt });
    } catch (error) {
      results.push({ success: false, error });
    }
  }

  return results;
}
```

我们可以收集所有交易的结果，包括成功和失败的情况，这样可以给用户完整的反馈。

---

11. 签名消息

11.1 消息签名基础

消息签名是DApp开发中的重要功能，它允许我们验证用户身份而不需要发送交易。理解消息签名的原理和应用场景对于构建安全的DApp非常重要。

消息签名的常见用途包括：

身份验证：

- 证明用户拥有私钥
- 无需发送交易
- 节省Gas费用

授权操作：

- 离线签名
- 委托他人执行
- 支持时间锁定

数据完整性：

- 验证数据来源
- 防止篡改
- 提供不可否认性

11.2 标准消息签名

签名标准消息很简单。我们使用Signer的signMessage方法签名消息，传入要签名的字符串。

```
async function signMessage(message) {
  try {
    const signature = await signer.signMessage(message);
    console.log('签名:', signature);
    return signature;
  } catch (error) {
    if (error.code === 4001) {
      console.log('用户拒绝签名');
    } else {
      console.error('签名失败:', error);
    }
    throw error;
  }
}
```

签名后会返回签名结果，这是一个十六进制字符串。

11.3 验证签名

我们可以使用verifyMessage方法验证签名，恢复签名者地址，与原始地址对比即可验证签名的有效性。

```
function verifyMessage(message, signature) {
  const recoveredAddress = ethers.verifyMessage(message, signature);
  console.log('恢复的地址:', recoveredAddress);

  // 与原始地址对比
  const isValid = recoveredAddress.toLowerCase() === signer.address.toLowerCase();
  console.log('签名有效:', isValid);

  return isValid;
}
```

验证签名可以让我们确认消息确实是由指定地址签名的，这对于身份验证非常重要。

11.4 EIP-191标准消息签名

除了标准消息签名，EIP-191标准消息签名是更规范的方式。我们可以使用hashMessage方法计算消息哈希，然后签名。

```
async function signEIP191Message(message) {
  // 计算消息哈希
  const messageHash = ethers.hashMessage(message);
  console.log('消息哈希:', messageHash);

  // 签名（Ethers.js会自动处理EIP-191格式）
  const signature = await signer.signMessage(message);
  return signature;
}
```

验证时同样使用verifyMessage方法。这种方式更安全，因为消息会经过标准化处理，包括添加前缀和长度信息。

11.5 EIP-712结构化消息签名

结构化消息签名使用EIP-712标准，这是最推荐的方式。我们需要定义domain、types和value。

```
const domain = {
  name: 'MyDApp',
  version: '1',
  chainId: 1,
  verifyingContract: '0x...'
};

const types = {
  Person: [
    { name: 'name', type: 'string' },
    { name: 'wallet', type: 'address' }
  ],
  Mail: [
    { name: 'from', type: 'Person' },
    { name: 'to', type: 'Person' },
    { name: 'contents', type: 'string' }
  ]
};

const value = {
  from: {
    name: 'Alice',
    wallet: '0x...'
  },
  to: {
    name: 'Bob',
    wallet: '0x...'
  },
  contents: 'Hello, Bob!'
};

async function signTypedData() {
  const signature = await signer.signTypedData(domain, types, value);
  return signature;
}
```

```
function verifyTypedData(signature) {
  const recoveredAddress = ethers.verifyTypedData(domain, types, value, signature);
  return recoveredAddress;
}
```

domain包含应用名称、版本、链ID和验证合约地址。types定义了数据结构。value是实际要签名的数据。

签名使用signTypedData方法，验证使用verifyTypedData方法。这种方式提供了最好的用户体验和安全性，因为MetaMask会显示结构化的签名内容，用户可以清楚地看到要签名的内容。

11.6 登录认证应用

登录认证是消息签名的典型应用。我们可以生成随机nonce，构建登录消息，请求用户签名，然后验证签名。

```
async function login() {
  // 生成随机nonce
  const nonce = Math.random().toString(36).substring(7);

  // 构建登录消息
  const message = `请签名以登录\n\nNonce: ${nonce}`;

  try {
    // 请求用户签名
    const signature = await signer.signMessage(message);

    // 验证签名
    const recoveredAddress = ethers.verifyMessage(message, signature);
    const isValid = recoveredAddress.toLowerCase() === signer.address.toLowerCase();

    if (isValid) {
      console.log('登录成功');
      // 保存登录状态
      localStorage.setItem('loggedIn', 'true');
      localStorage.setItem('address', signer.address);
      localStorage.setItem('signature', signature);
    } else {
      console.log('签名验证失败');
    }

    return isValid;
  } catch (error) {
    if (error.code === 4001) {
      console.log('用户拒绝签名');
    } else {
      console.error('登录失败:', error);
    }
    return false;
  }
}
```

如果验证通过，我们可以认为用户已经登录。这种方式比传统的用户名密码更安全，因为私钥永远不会离开用户的钱包。

11.7 授权委托应用

授权委托是另一个应用场景。我们可以让用户签名一个授权消息，包含委托地址、操作类型、过期时间等信息。

```
async function createAuthorization(delegateAddress, operation, expiry) {
  const message = JSON.stringify({
    delegate: delegateAddress,
    operation: operation,
    expiry: expiry,
    timestamp: Date.now()
  });

  try {
    const signature = await signer.signMessage(message);

    // 保存授权信息
    const authorization = {
      signer: signer.address,
      delegate: delegateAddress,
      operation: operation,
      expiry: expiry,
      signature: signature,
      message: message
    };

    return authorization;
  } catch (error) {
    console.error('创建授权失败:', error);
    throw error;
  }
}

function verifyAuthorization(authorization) {
  const recoveredAddress = ethers.verifyMessage(authorization.message,
authorization.signature);
  const isValid = recoveredAddress.toLowerCase() === authorization.signer.toLowerCase();

  if (!isValid) {
    return false;
  }

  // 检查是否过期
  if (Date.now() > authorization.expiry) {
    return false;
  }

  return true;
}
```

其他人可以使用这个签名来执行操作，我们可以在链上或链下验证签名的有效性。这种方式可以实现离线授权，非常灵活。

11.8 错误处理

错误处理时，用户可能拒绝签名，错误代码是4001。消息格式可能无效，错误代码是-32602。

```
try {
  const signature = await signer.signMessage(message);
  return signature;
} catch (error) {
  if (error.code === 4001) {
    alert('用户拒绝签名');
  } else if (error.code === -32602) {
    alert('消息格式无效');
  } else {
    console.error('签名失败:', error);
    alert('签名失败，请重试');
  }
  throw error;
}
```

根据不同的错误给出相应的提示，可以提供更好的用户体验。

---

12. 综合示例

12.1 DApp类封装

现在让我们来看一个完整的DApp示例，它整合了前面学到的所有知识点。这个DApp类封装了完整的交互功能。

```
import { ethers } from 'ethers';

class DApp {
  constructor() {
    this.provider = null;
    this.signer = null;
    this.account = null;
    this.contracts = {};
  }

  // 初始化
  async init() {
    // 检查MetaMask是否安装
    if (typeof window.ethereum === 'undefined') {
      throw new Error('请安装MetaMask');
    }

    // 创建Provider
    this.provider = new ethers.BrowserProvider(window.ethereum);
```

```
    // 设置事件监听器
    this.setupEventListeners();

    // 检查连接状态
    await this.checkConnection();
  }

  // 设置事件监听器
  setupEventListeners() {
    window.ethereum.on('accountsChanged', (accounts) => {
      if (accounts.length === 0) {
        this.disconnect();
      } else {
        this.connectWallet();
      }
    });

    window.ethereum.on('chainChanged', () => {
      window.location.reload();
    });
  }

  // 检查连接状态
  async checkConnection() {
    try {
      const accounts = await window.ethereum.request({ method: 'eth_accounts' });
      if (accounts.length > 0) {
        await this.connectWallet();
      }
    } catch (error) {
      console.error('检查连接失败:', error);
    }
  }

  // 连接钱包
  async connectWallet() {
    try {
      await window.ethereum.request({ method: 'eth_requestAccounts' });
      this.provider = new ethers.BrowserProvider(window.ethereum);
      this.signer = await this.provider.getSigner();
      this.account = await this.signer.getAddress();
      console.log('已连接账户:', this.account);
      return this.account;
    } catch (error) {
      if (error.code === 4001) {
        throw new Error('用户拒绝连接');
      } else {
        throw error;
      }
    }
  }
```

```
// 断开连接
disconnect() {
  this.provider = null;
  this.signer = null;
  this.account = null;
  this.contracts = {};
}

// 加载合约
loadContract(address, abi) {
  if (!this.signer) {
    throw new Error('请先连接钱包');
  }

  const contract = new ethers.Contract(address, abi, this.signer);
  this.contracts[address] = contract;
  return contract;
}

// 读取余额
async getBalance(address = null) {
  const targetAddress = address || this.account;
  if (!targetAddress) {
    throw new Error('请先连接钱包或指定地址');
  }

  const balance = await this.provider.getBalance(targetAddress);
  return ethers.formatEther(balance);
}

// 发送ETH
async sendETH(to, amount) {
  if (!this.signer) {
    throw new Error('请先连接钱包');
  }

  const tx = await this.signer.sendTransaction({
    to: to,
    value: ethers.parseEther(amount.toString())
  });

  const receipt = await tx.wait();
  return receipt;
}

// 调用合约函数
async callContractFunction(contractAddress, functionName, ...args) {
  const contract = this.contracts[contractAddress];
  if (!contract) {
    throw new Error('合约未加载');
  }
```

```
    const tx = await contract[functionName](...args);
    const receipt = await tx.wait();
    return receipt;
  }

  // 读取合约状态
  async readContractState(contractAddress, functionName, ...args) {
    const contract = this.contracts[contractAddress];
    if (!contract) {
      throw new Error('合约未加载');
    }

    const result = await contract[functionName](...args);
    return result;
  }

  // 监听合约事件
  onContractEvent(contractAddress, eventName, callback) {
    const contract = this.contracts[contractAddress];
    if (!contract) {
      throw new Error('合约未加载');
    }

    contract.on(eventName, callback);
  }

  // 签名消息
  async signMessage(message) {
    if (!this.signer) {
      throw new Error('请先连接钱包');
    }

    const signature = await this.signer.signMessage(message);
    return signature;
  }
}
```

这个DApp类封装了完整的交互功能，包括连接钱包、加载合约、读取数据、发送交易、监听事件、签名消息等。

12.2 使用示例

使用这个DApp类很简单：

```
// 创建实例
const dapp = new DApp();

// 初始化
await dapp.init();

// 连接钱包
```

```
await dapp.connectWallet();

// 加载合约
const contract = dapp.loadContract(contractAddress, contractABI);

// 读取合约状态
const number = await dapp.readContractState(contractAddress, 'getNumber');
console.log('当前值:', number.toString());

// 监听事件
dapp.onContractEvent(contractAddress, 'Incremented', (user, newValue) => {
  console.log('Incremented事件:', user, newValue.toString());
});

// 调用合约函数
await dapp.callContractFunction(contractAddress, 'increment');

// 读取余额
const balance = await dapp.getBalance();
console.log('余额:', balance, 'ETH');
```

这个封装让DApp开发变得更加简单和高效，我们可以专注于业务逻辑，而不需要处理底层的交互细节。

---

13. 最佳实践

13.1 错误处理

在开发DApp时，遵循最佳实践可以让我们的应用更加稳定、安全和用户友好。

推荐做法：

- 始终使用try-catch处理异步操作
- 处理用户拒绝的情况（错误代码4001）
- 提供友好的错误提示
- 记录错误日志以便调试

避免做法：

- 不要忽略错误
- 不要静默失败
- 不要给用户显示技术性的错误信息

13.2 用户体验

推荐做法：

- 显示加载状态
- 提供交易进度反馈
- 优化等待时间
- 提供清晰的操作指引

避免做法：

- 不要让用户长时间等待而不给反馈
- 不要使用技术术语
- 不要隐藏重要的操作步骤

13.3 安全性

推荐做法：

- 验证用户输入
- 检查余额和Gas
- 使用类型安全的ABI
- 验证合约地址格式

避免做法：

- 永远不要在前端存储私钥
- 不要信任用户输入
- 不要硬编码敏感信息
- 不要忽略安全检查

13.4 性能优化

推荐做法：

- 缓存Provider实例
- 批量读取数据
- 使用事件而非轮询
- 合理使用缓存

避免做法：

- 不要频繁请求连接
- 不要重复创建Provider
- 不要过度轮询
- 不要忽略性能优化

13.5 代码组织

推荐做法：

- 使用配置文件管理RPC URL和合约地址
- 封装通用的交互逻辑
- 使用单例模式管理Provider
- 合理组织代码结构

避免做法：

- 不要硬编码RPC URL和合约地址
- 不要重复代码
- 不要忽略代码组织
- 不要使用全局变量

14. 常见错误和解决方案

14.1 连接问题

问题：MetaMask未安装

错误信息：

```
Cannot read property 'request' of undefined
```

解决方案：

```
if (typeof window.ethereum === 'undefined') {
  alert('请安装MetaMask');
  window.open('https://metamask.io/download/', '_blank');
}
```

问题：用户拒绝连接

错误信息：

```
Error: User rejected the request
```

解决方案：

```
try {
  await window.ethereum.request({ method: 'eth_requestAccounts' });
} catch (error) {
  if (error.code === 4001) {
    alert('请连接钱包以继续使用');
  }
}
```

14.2 交易问题

问题：余额不足

错误信息：

```
Error: insufficient funds
```

解决方案：

```
// 发送交易前检查余额
const balance = await provider.getBalance(address);
const required = ethers.parseEther(amount);
if (balance < required) {
  alert('余额不足，请充值');
  return;
}
```

问题：Gas估算失败

错误信息：

```
Error: cannot estimate gas
```

解决方案：

```
try {
  const gasEstimate = await contract.functionName.estimateGas(...args);
  const tx = await contract.functionName(...args, { gasLimit: gasEstimate });
} catch (error) {
  console.error('Gas估算失败，交易可能失败:', error);
}
```

14.3 合约调用问题

问题：合约调用失败

错误信息：

```
Error: call revert exception
```

解决方案：

```
try {
  const result = await contract.functionName(...args);
} catch (error) {
  if (error.code === 'CALL_EXCEPTION') {
    console.error('合约调用失败:', error.message);
    // 检查函数是否存在、参数是否正确、合约状态是否允许
  }
}
```

问题：函数不存在

错误信息：

```
Error: function "functionName" not found
```

解决方案：

- 检查ABI是否包含该函数
- 检查函数名称和参数类型是否正确
- 确保合约地址正确

14.4 网络问题

问题：网络不匹配

错误信息：

```
Error: network mismatch
```

解决方案：

```
// 检查当前网络
const network = await provider.getNetwork();
if (network.chainId !== expectedChainId) {
  await switchNetwork(expectedChainId);
}
```

问题：RPC连接失败

错误信息：

```
Error: Failed to connect to RPC
```

解决方案：

- 检查RPC URL是否正确
- 检查网络连接
- 尝试使用其他RPC端点
- 检查API密钥是否有效

---

15. 实战演示

15.1 演示环境准备

接下来，我将通过实际的代码演示，带大家完整地走一遍Ethers.js与MetaMask交互的流程。

创建Hardhat项目：

```
# 创建项目目录
mkdir ethers-demo
cd ethers-demo

# 初始化Hardhat项目
npx hardhat@latest init

# 选择TypeScript项目
# 安装ethers.js
npm install ethers@^6.0.0
```

15.2 创建测试合约

在`contracts`目录下创建`Counter.sol`：

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

contract Counter {
    uint256 public number;

    event Incremented(address indexed user, uint256 newValue);
    event SetNumber(address indexed user, uint256 newValue);

    function increment() public {
        number++;
        emit Incremented(msg.sender, number);
    }

    function setNumber(uint256 newNumber) public {
        number = newNumber;
        emit SetNumber(msg.sender, newNumber);
    }

    function getNumber() public view returns (uint256) {
        return number;
    }
}
```

15.3 部署脚本

在`scripts`目录下创建`deploy.ts`：

```
import { ethers } from "hardhat";

async function main() {
  const [deployer] = await ethers.getSigners();
  console.log("部署账户:", deployer.address);
  console.log("账户余额:", ethers.formatEther(await
ethers.provider.getBalance(deployer.address)));
```

```
  const Counter = await ethers.getContractFactory("Counter");
  const counter = await Counter.deploy();

  await counter.waitForDeployment();
  const address = await counter.getAddress();

  console.log("Counter合约已部署到:", address);
  console.log("初始number值:", (await counter.getNumber()).toString());

  return address;
}

main()
  .then((address) => {
    console.log("部署成功，合约地址:", address);
    process.exit(0);
  })
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

15.4 启动本地节点和部署

启动Hardhat节点：

```
npx hardhat node
```

部署合约：

```
npx hardhat run scripts/deploy.ts --network localhost
```

15.5 浏览器环境演示

创建`demo.html`文件，演示与MetaMask的交互：

```
<!DOCTYPE html>
<html>
<head>
  <title>Ethers.js + MetaMask 演示</title>
  <script src="https://cdn.ethers.io/lib/ethers-6.0.0.umd.min.js"></script>
</head>
<body>
  <h1>Ethers.js + MetaMask 演示</h1>

  <div>
    <button id="connectBtn">连接MetaMask</button>
    <div id="accountInfo"></div>
  </div>
```

```
  <div>
    <h2>合约交互</h2>
    <p>合约地址：<span
id="contractAddress">0x5FbDB2315678afecb367f032d93F642f64180aa3</span></p>
    <p>当前值：<span id="currentValue">-</span></p>
    <button id="incrementBtn">Increment</button>
    <button id="setNumberBtn">Set Number</button>
    <input type="number" id="numberInput" placeholder="输入数字" value="100">
  </div>

  <div>
    <h2>事件日志</h2>
    <div id="eventLog"></div>
  </div>

  <script>
    const CONTRACT_ADDRESS = '0x5FbDB2315678afecb367f032d93F642f64180aa3';
    const counterAbi = [
      "function getNumber() view returns (uint256)",
      "function increment()",
      "function setNumber(uint256)",
      "event Incremented(address indexed user, uint256 newValue)",
      "event SetNumber(address indexed user, uint256 newValue)"
    ];

    let provider, signer, contract, account;

    // 连接MetaMask
    document.getElementById('connectBtn').addEventListener('click', async () => {
      if (typeof window.ethereum === 'undefined') {
        alert('请安装MetaMask');
        return;
      }

      try {
        await window.ethereum.request({ method: 'eth_requestAccounts' });
        provider = new ethers.BrowserProvider(window.ethereum);
        signer = await provider.getSigner();
        account = await signer.getAddress();

        document.getElementById('accountInfo').innerHTML = `已连接：${account}`;
        document.getElementById('connectBtn').disabled = true;

        // 创建合约实例
        contract = new ethers.Contract(CONTRACT_ADDRESS, counterAbi, signer);

        // 读取初始值
        await updateValue();

        // 设置事件监听
        setupEventListeners();
```

```
  } catch (error) {
    if (error.code === 4001) {
      alert('用户拒绝连接');
    } else {
      console.error('连接失败:', error);
    }
  }
});

// 更新当前值
async function updateValue() {
  if (!contract) return;
  try {
    const value = await contract.getNumber();
    document.getElementById('currentValue').textContent = value.toString();
  } catch (error) {
    console.error('读取值失败:', error);
  }
}

// Increment按钮
document.getElementById('incrementBtn').addEventListener('click', async () => {
  if (!contract) {
    alert('请先连接MetaMask');
    return;
  }

  try {
    const tx = await contract.increment();
    addLog('交易已发送，哈希: ' + tx.hash);
    const receipt = await tx.wait();
    addLog('交易确认，区块号: ' + receipt.blockNumber);
    await updateValue();
  } catch (error) {
    if (error.code === 4001) {
      addLog('用户拒绝交易');
    } else {
      addLog('交易失败: ' + error.message);
    }
  }
});

// Set Number按钮
document.getElementById('setNumberBtn').addEventListener('click', async () => {
  if (!contract) {
    alert('请先连接MetaMask');
    return;
  }

  const number = document.getElementById('numberInput').value;
  if (!number) {
```

```
      alert('请输入数字');
      return;
    }

    try {
      const tx = await contract.setNumber(number);
      addLog('交易已发送，哈希：' + tx.hash);
      const receipt = await tx.wait();
      addLog('交易确认，区块号：' + receipt.blockNumber);
      await updateValue();
    } catch (error) {
      if (error.code === 4001) {
        addLog('用户拒绝交易');
      } else {
        addLog('交易失败：' + error.message);
      }
    }
  });

  // 设置事件监听
  function setupEventListeners() {
    if (!contract) return;

    contract.on('Incremented', (user, newValue, event) => {
      addLog(`Incremented事件：用户 ${user} 新值 ${newValue.toString()}`);
    });

    contract.on('SetNumber', (user, newValue, event) => {
      addLog(`SetNumber事件：用户 ${user} 新值 ${newValue.toString()}`);
    });
  }

  // 添加日志
  function addLog(message) {
    const log = document.getElementById('eventLog');
    const time = new Date().toLocaleTimeString();
    log.innerHTML += `<div>[${time}] ${message}</div>`;
    log.scrollTop = log.scrollHeight;
  }

  // 监听账户变化
  if (window.ethereum) {
    window.ethereum.on('accountsChanged', (accounts) => {
      if (accounts.length === 0) {
        location.reload();
      } else {
        location.reload();
      }
    });

    window.ethereum.on('chainChanged', () => {
      location.reload();
```

```
      });
    }
  </script>
</body>
</html>
```

15.6 演示总结

通过这个完整的演示，我们完成了以下操作：

1. 创建Hardhat项目并安装依赖
2. 编写和部署Counter合约到本地节点
3. 创建HTML页面演示Ethers.js与MetaMask的交互
4. 实现连接MetaMask钱包功能
5. 实现读取合约状态功能
6. 实现调用合约函数功能
7. 实现监听合约事件功能
8. 实现处理账户和网络变化功能

整个流程展示了Ethers.js的核心功能，从基础的Provider连接到复杂的合约交互，再到与MetaMask的集成。在实际开发中，你可以使用这些技术构建完整的DApp应用。

---

16. 学习资源与总结

16.1 官方资源

Ethers.js官方文档：

- 官方文档：https://docs.ethers.org/
- GitHub仓库：https://github.com/ethers-io/ethers.js
- 官方博客和更新

MetaMask文档：

- MetaMask文档：https://docs.metamask.io/
- MetaMask API参考

16.2 社区资源

社区支持：

- GitHub Discussions
- Stack Overflow
- Discord社区

学习资源：

- 社区教程和指南
- YouTube视频教程
- 博客文章

16.3 核心知识点总结

通过本课程的学习，你应该已经掌握了：

1. **Ethers.js基础**：
   - Ethers.js与Web3.js的对比
   - Ethers.js的核心优势
   - Provider和Signer的概念
2. **Provider连接**：
   - 各种Provider类型
   - 连接测试网和主网
   - 验证连接状态
3. **区块链数据读取**：
   - 查询区块信息
   - 查询账户信息
   - 查询交易信息
4. **合约交互**：
   - 创建合约实例
   - 读取合约状态
   - 调用合约函数
   - 监听合约事件
5. **MetaMask集成**：
   - 检测MetaMask
   - 请求账户连接
   - 实现网络切换
6. **交易和签名**：
   - 发送交易
   - 签名消息
   - 错误处理

16.4 下一步学习

深入学习：

1. **高级Provider使用**：学习WebSocket Provider、自定义Provider等
2. **多链支持**：学习如何支持多个区块链网络
3. **批量操作优化**：学习如何优化批量读取和交易
4. **前端框架集成**：学习如何在React、Vue、Next.js中集成Ethers.js

实践建议：

- 构建完整的DApp项目
- 实现复杂的交互功能
- 优化用户体验
- 参与开源项目

16.5 总结

Ethers.js是一个强大的库，掌握它能让你的DApp开发效率大幅提升。

关键收获：

1. **现代化设计**：Ethers.js提供了现代化的API设计，代码更简洁，开发体验更好
2. **类型安全**：原生TypeScript支持提供了完整的类型推断，能够在编译时发现错误
3. **功能完善**：提供了完整的区块链交互功能，包括读取数据、发送交易、监听事件等
4. **MetaMask集成**：与MetaMask的集成非常简单，可以轻松构建用户友好的DApp

实践建议：

- 在实际项目中尝试使用Ethers.js
- 遵循最佳实践和安全规范
- 持续学习和改进
- 参与社区讨论和贡献

恭喜！你已经掌握了Ethers.js基础与MetaMask交互的核心使用方法。现在你可以使用Ethers.js构建完整的DApp前端应用，实现用户与区块链的交互。

祝你学习愉快，开发顺利！
