Aula 19 – Lendo Dados do Contrato a partir do Frontend

Imagine que você acabou de construir um edifício incrível, com todas as funcionalidades modernas e seguras. Esse edifício é o seu Smart Contract, e ele está lá, firme e forte, na blockchain. Agora, como as pessoas que visitam esse edifício (seus usuários) conseguem ver as informações que ele guarda, como o número de andares, o nome do proprietário ou o status de uma reserva, sem ter que pagar uma taxa para entrar?

Essa é a essência do que vamos explorar nesta aula. No mundo dos Smart Contracts e DApps, interagir com a blockchain pode ter um custo, o famoso "gás". No entanto, nem toda interação exige esse pagamento. Muitas vezes, precisamos apenas "olhar" para o contrato e extrair informações que já estão lá, sem modificar seu estado.

Nosso objetivo aqui é desvendar como o frontend de uma aplicação descentralizada (DApp) pode se comunicar com um Smart Contract para ler dados de forma eficiente e gratuita. Ao final desta aula, você será capaz de identificar e implementar chamadas a funções view e pure em seus contratos, garantindo que suas DApps exibam informações importantes sem onerar seus usuários com custos de transação. Prepare-se para conectar o mundo visível da interface do usuário com a lógica imutável da blockchain.

A Necessidade de "Olhar" sem "Tocar": Funções view e pure

No universo dos Smart Contracts, cada ação que altera o estado da blockchain – como enviar tokens, registrar um novo usuário ou atualizar um saldo – custa gás. É como pagar um pedágio para cada mudança que você faz na estrada principal. No entanto, muitas vezes, o que precisamos é apenas verificar o mapa, saber a velocidade máxima permitida ou ver o próximo ponto de interesse, sem realmente mudar nada na estrada.

É exatamente para isso que existem as funções view e pure em Solidity. Elas são as "janelas" para o seu Smart Contract, permitindo que qualquer pessoa visualize as informações armazenadas ou execute cálculos sem custo. Isso é crucial para a experiência do usuário em DApps, pois ninguém quer pagar para simplesmente ver seu saldo ou o nome de um produto.

> **Analogia do Caixa Eletrônico:** Você pode sacar dinheiro (o que altera seu saldo e custa uma taxa de serviço – uma transação com gás), mas também pode apenas consultar seu extrato ou saldo (o que não altera nada e geralmente não tem custo direto – uma chamada view ou pure).

Essa distinção é fundamental para construir DApps eficientes e amigáveis.

Desvendando view e pure: As Funções Leitoras

Para entender como ler dados, precisamos primeiro diferenciar as duas categorias de funções que não modificam o estado do contrato: view e pure. Ambas são gratuitas para o usuário quando chamadas de fora da blockchain (por um frontend, por exemplo), mas possuem nuances importantes.

Função view

Uma função view é como um observador que pode olhar para o estado atual do contrato. Ela pode ler as variáveis de estado declaradas no contrato, mas não pode modificá-las. Por exemplo, se você tem uma variável que armazena o saldo de um usuário, uma função view pode retornar esse saldo. Ela "vê" o que está lá.

Função pure

Já uma função pure é ainda mais restritiva. Ela é "pura" porque não interage de forma alguma com o estado do contrato. Ela não pode ler variáveis de estado, nem modificá-las. Funções pure são usadas para realizar cálculos que dependem apenas dos argumentos de entrada da função. Imagine uma calculadora dentro do contrato: você insere dois números e ela retorna a soma, sem precisar saber de nada que está armazenado no contrato.

Quadro Comparativo: view vs. pure

Característica	Função view	Função pure
Acesso ao Estado	Pode ler variáveis de estado do contrato.	Não pode ler variáveis de estado do contrato.
Modificação	Não pode modificar variáveis de estado.	Não pode modificar variáveis de estado.
Custo de Gás	Gratuita quando chamada externamente.	Gratuita quando chamada externamente.
Exemplo	`function getBalance() view returns (uint)`	function add(uint a, uint b) pure returns (uint)

Implementando Funções view e pure em Solidity

Agora que entendemos a teoria, vamos ver como isso se traduz em código Solidity. A implementação é bastante direta, mas a escolha correta entre view e pure é crucial para a segurança e eficiência do seu contrato.

Considere um contrato simples que armazena um nome e um número, e que também pode realizar uma operação matemática básica.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MyDataContract {
    string public contractName = "Meu Contrato de Dados"; // Variável de estado
    uint public contractVersion = 1; // Variável de estado

    // Função view: lê uma variável de estado
    function getContractName() public view returns (string memory) {
        return contractName;
    }

    // Função view: lê outra variável de estado
    function getContractVersion() public view returns (uint) {
        return contractVersion;
    }

    // Função pure: realiza um cálculo sem acessar o estado do contrato
    function multiply(uint a, uint b) public pure returns (uint) {
        return a * b;
    }
}
```

getContractName() e getContractVersion()

São funções **view** porque acessam as variáveis de estado contractName e contractVersion.

multiply()

É uma função **pure** porque seu resultado depende apenas dos parâmetros a e b, sem tocar em nenhuma variável de estado do contrato.

Essas funções podem ser chamadas por qualquer frontend sem custo de gás para o usuário final.

Conectando o Frontend: A Ponte entre o Usuário e o Contrato

Com nossas funções view e pure prontas no Smart Contract, o próximo passo é fazer com que o frontend da nossa DApp consiga chamá-las e exibir os dados. Para isso, utilizaremos bibliotecas JavaScript como ethers.js ou web3.js, que são as ferramentas padrão para interagir com a blockchain Ethereum a partir de aplicações web.

> **A lógica é a seguinte:** o frontend precisa saber o endereço do contrato na blockchain (onde ele está "morando") e a Interface Binária de Aplicação (ABI) do contrato. A ABI é como um manual de instruções que descreve todas as funções do contrato, seus parâmetros e tipos de retorno.

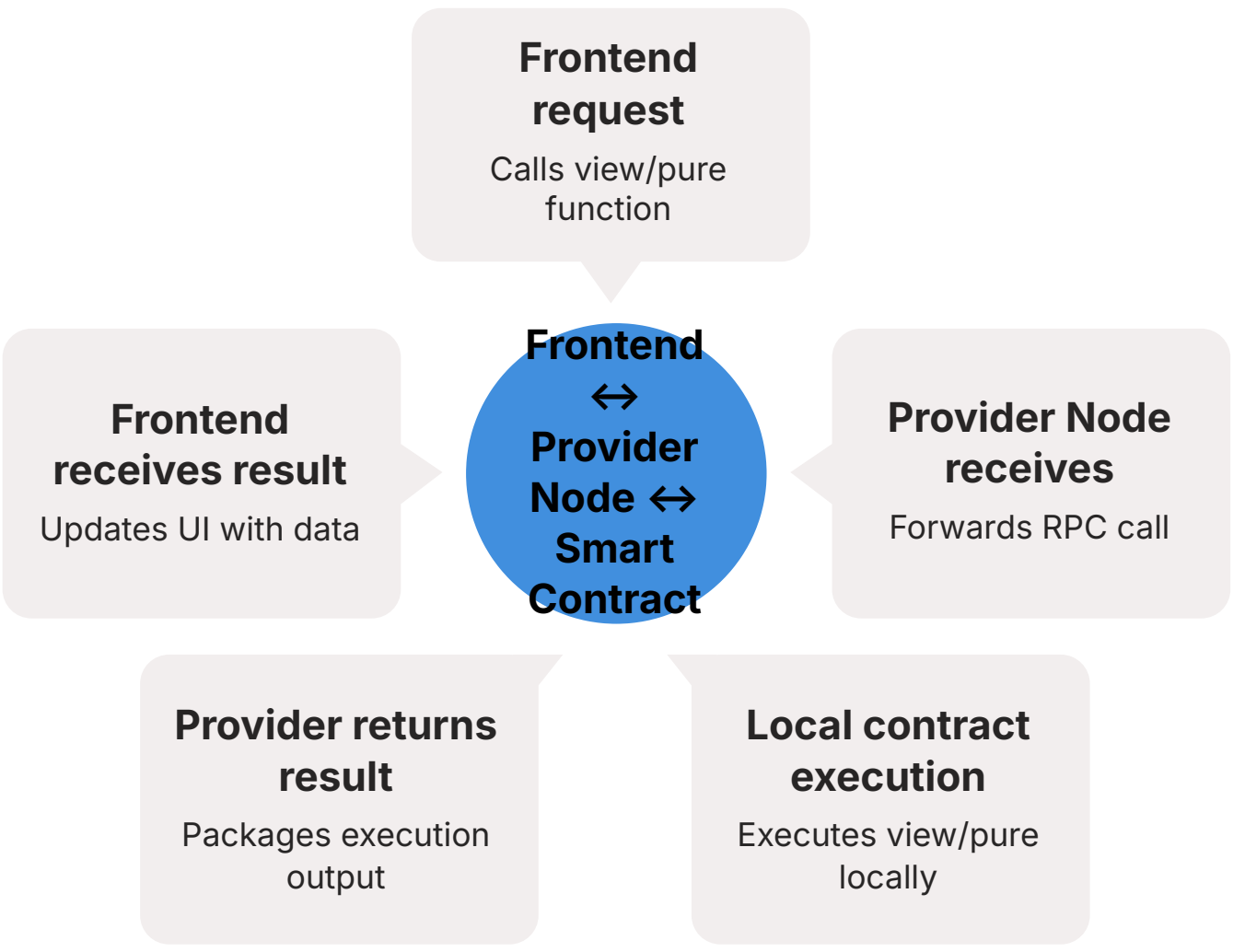
Imagine que você tem um controle remoto (o frontend) e um aparelho de TV (o Smart Contract). Para o controle funcionar, você precisa saber onde a TV está (endereço do contrato) e quais botões ela tem e o que cada um faz (a ABI). Sem isso, o controle remoto é inútil.

```
// Exemplo simplificado de como chamar uma função view com ethers.js
import { ethers } from "ethers";

// 1. Endereço do seu contrato na blockchain
const contractAddress = "0x..."; // Substitua pelo endereço real do seu contrato

// 2. ABI do seu contrato (gerada durante a compilação)
const contractABI = [
    // Apenas as funções que você quer chamar
    "function getContractName() view returns (string)",
    "function getContractVersion() view returns (uint)",
    "function multiply(uint a, uint b) pure returns (uint)"
];

// 3. Conectar a um provedor Ethereum (ex: Metamask, Infura, Alchemy)
// Usamos um provedor de leitura, pois não faremos transações
const provider = new ethers.providers.Web3Provider(window.ethereum); // Para Metamask
// Ou new ethers.providers.JsonRpcProvider("https://mainnet.infura.io/v3/YOUR_INFURA_PROJECT_ID");

// 4. Criar uma instância do contrato
const contract = new ethers.Contract(contractAddress, contractABI, provider);

// 5. Chamar as funções view/pure
async function fetchData() {
    try {
        const name = await contract.getContractName();
        const version = await contract.getContractVersion();
        const result = await contract.multiply(5, 10);

        console.log("Nome do Contrato:", name); // Ex: "Meu Contrato de Dados"
        console.log("Versão do Contrato:", version.toString()); // Ex: "1"
        console.log("Resultado da Multiplicação:", result.toString()); // Ex: "50"
    } catch (error) {
        console.error("Erro ao ler dados do contrato:", error);
    }
}

fetchData();
```

Este trecho de código ilustra o processo básico. O provider é a conexão com a rede Ethereum, o contract é a representação do seu Smart Contract no JavaScript, e as chamadas await contract.getContractName() são as invocações diretas das funções do contrato.

O Fluxo de Dados: Da Blockchain ao Navegador

Entender o fluxo de dados é crucial para depurar e otimizar suas DApps. Quando o frontend chama uma função view ou pure, a requisição não é enviada para ser minerada em um bloco. Em vez disso, ela é enviada diretamente para um nó da rede Ethereum (o provider que configuramos).



Esse nó executa a função localmente, no seu próprio ambiente, sem broadcast para a rede. O resultado é então retornado diretamente para o frontend. É um processo rápido e eficiente, pois não envolve a complexidade e o tempo de confirmação de uma transação na blockchain.

> **Analogia da Biblioteca:** Imagine que você está em uma biblioteca (a blockchain) e quer saber o título de um livro (dado do contrato). Em vez de pedir para o bibliotecário (minerador) registrar seu pedido e esperar que ele seja processado por todos os outros bibliotecários, você simplesmente vai até a estante e lê o título diretamente. É uma consulta instantânea.

Essa arquitetura garante que a exibição de informações em sua DApp seja tão responsiva quanto em qualquer aplicação web tradicional, enquanto a segurança e a imutabilidade dos dados permanecem garantidas pela blockchain.

Lidando com Tipos de Dados e Erros

Tratamento de BigNumber

Ao ler dados de um Smart Contract, é comum encontrar tipos de dados que precisam de tratamento especial no JavaScript. Por exemplo, uint (inteiros sem sinal) em Solidity podem ser muito grandes para serem representados com precisão como números JavaScript padrão. Nesses casos, bibliotecas como ethers.js retornam esses valores como objetos BigNumber, que precisam ser convertidos para string ou uint para exibição ou manipulação.

```
// Exemplo de tratamento de BigNumber
const bigNumberValue = await
contract.getLargeNumber(); // Retorna um
BigNumber
console.log("Valor como BigNumber:",
bigNumberValue);
console.log("Valor como string:",
bigNumberValue.toString());
console.log("Valor como número JS (cuidado com
precisão):", bigNumberValue.toNumber());
```

Tratamento de Erros

Além disso, é fundamental implementar um bom tratamento de erros. Chamadas a funções view e pure podem falhar por diversos motivos: o contrato não existe no endereço fornecido, a rede está inacessível, ou a função foi chamada com parâmetros incorretos. Usar blocos try-catch é uma prática recomendada para fornecer feedback útil ao usuário e evitar que a aplicação quebre.

Segurança em 2025: Embora funções view e pure não alterem o estado do contrato, a forma como os dados são exibidos e interpretados no frontend pode ter implicações. Sempre valide e saneie os dados recebidos do contrato antes de exibi-los, especialmente se eles puderem ser inseridos por usuários.

Cenários de Aplicação e Melhores Práticas

A capacidade de ler dados de contratos sem custo de gás abre um leque vasto de possibilidades para DApps. Pense em um marketplace NFT: o frontend pode exibir a lista de NFTs disponíveis, seus preços, proprietários e metadados chamando funções view. Um jogo blockchain pode mostrar o placar dos jogadores, o status de um item ou o nível de um personagem. Uma plataforma de votação pode exibir os resultados parciais ou finais.

Melhores Práticas:

01

Granularidade

Crie funções view e pure específicas para cada pedaço de informação que você precisa. Evite funções genéricas que retornem muitos dados desnecessários.

02

Otimização de Gás (no contrato)

Embora view e pure sejam gratuitas para chamadas externas, elas ainda consomem gás *interno* se chamadas por outras funções dentro do contrato (que aí sim, custam gás). Otimize-as para serem o mais eficientes possível.

03

Documentação

Documente claramente o que cada função view e pure retorna e quais são seus parâmetros. Isso facilita a integração com o frontend.

04

Cache (com cautela)

Para dados que não mudam com frequência, o frontend pode implementar um cache local para evitar chamadas repetitivas à blockchain. No entanto, sempre considere a "frescura" dos dados e a tolerância a dados desatualizados.

05

Testes

Teste exaustivamente suas funções view e pure e sua integração com o frontend para garantir que os dados sejam lidos e exibidos corretamente em diferentes cenários.

Ferramentas Modernas e o Ecossistema

O desenvolvimento de DApps está em constante evolução, e as ferramentas modernas, como o framework Hardhat, simplificam muito a vida do desenvolvedor. O Hardhat não só facilita a compilação e o deploy de contratos, mas também oferece um ambiente de desenvolvimento local para testes, onde você pode simular chamadas a funções view e pure rapidamente.

Hardhat

Framework completo para desenvolvimento, compilação e deploy de contratos. Oferece ambiente de testes local e integração nativa com ethers.js.

ethers.js

Biblioteca JavaScript moderna para interação com a blockchain Ethereum. Geração automática de ABI e suporte completo a funções view e pure.

OpenZeppelin

Bibliotecas auditadas que garantem segurança e seguem padrões robustos, minimizando vulnerabilidades mesmo em funções de leitura.

A integração com bibliotecas como ethers.js é nativa, e a geração da ABI é automática, o que agiliza o processo de conexão do frontend. Além disso, a ênfase em segurança, com o uso de bibliotecas auditadas como OpenZeppelin, garante que as funções que você escreve, mesmo as de leitura, sigam padrões robustos e minimizem vulnerabilidades.

Fundamento essencial: Aprender a ler dados de contratos é um pilar fundamental para qualquer desenvolvedor Web3. É a base para construir interfaces de usuário ricas e interativas que realmente trazem a blockchain para perto do usuário final, sem a barreira do custo de gás para cada clique. Dominar essa técnica é essencial para criar DApps que sejam não apenas funcionais, mas também agradáveis de usar.

Reflexões sobre a Experiência do Usuário

A experiência do usuário (UX) em DApps é um campo que tem recebido muita atenção, e a forma como lidamos com a leitura de dados é um componente chave. Um usuário espera que, ao abrir uma aplicação, as informações sejam carregadas rapidamente e sem custos ocultos. Funções view e pure são os heróis silenciosos que tornam isso possível.

Se cada consulta de saldo ou visualização de um item em um marketplace exigisse uma transação e o pagamento de gás, a usabilidade seria drasticamente comprometida. A lentidão e o custo seriam barreiras intransponíveis para a adoção em massa. Ao otimizar a leitura de dados, estamos não apenas seguindo as melhores práticas técnicas, mas também construindo pontes para que mais pessoas possam interagir com a Web3 de forma fluida e intuitiva.

> É importante lembrar que, embora a chamada a essas funções seja gratuita para o usuário, o nó que processa a requisição ainda gasta recursos. Por isso, mesmo em funções view e pure, a eficiência do código Solidity é sempre uma boa prática. Um contrato bem escrito é um contrato que respeita os recursos da rede e, por extensão, a experiência de seus usuários.

Em Prática: Construindo um DApp Responsivo

Para solidificar o aprendizado, pense em um cenário prático. Você está desenvolvendo uma DApp de gerenciamento de tarefas descentralizada. O usuário precisa ver a lista de suas tarefas pendentes, as tarefas concluídas e os detalhes de cada tarefa. Todas essas informações podem ser lidas do Smart Contract usando funções view.

Carregar Página

Frontend chama `getPendingTasks()` e `getCompletedTasks()`

Clicar em Tarefa

Chama `getTaskDetails(taskId)` para exibir detalhes

Ação do Usuário

Apenas "adicionar tarefa" ou "marcar como concluída" exigem transação e gás

Ao carregar a página, o frontend faria chamadas assíncronas para getPendingTasks() e getCompletedTasks(). Ao clicar em uma tarefa específica, getTaskDetails(taskId) seria chamada. Nenhuma dessas ações custaria gás ao usuário, proporcionando uma experiência fluida e instantânea. Apenas quando o usuário decide "adicionar uma nova tarefa" ou "marcar como concluída" é que uma transação seria enviada, exigindo confirmação e gás.

O equilíbrio perfeito: Essa distinção clara entre leitura e escrita é o que permite que DApps se comportem de maneira semelhante às aplicações web tradicionais em termos de responsividade, ao mesmo tempo em que aproveitam a segurança e a descentralização da blockchain. É a arte de equilibrar a tecnologia subjacente com a usabilidade final.

Consolidação e Próximos Passos

O que aprendemos

Mergulhamos no mundo da leitura de dados de Smart Contracts a partir do frontend, focando nas funções view e pure. Aprendemos que elas são essenciais para construir DApps eficientes e amigáveis, permitindo que os usuários acessem informações sem incorrer em custos de gás.

Diferenças fundamentais

Vimos a diferença entre view (que lê o estado) e pure (que apenas calcula com base em entradas), como implementá-las em Solidity e como chamá-las usando ethers.js no JavaScript.

Aplicação prática

A capacidade de ler dados de forma gratuita e eficiente é a espinha dorsal de qualquer DApp interativa. Sem ela, a experiência do usuário seria inviável devido aos custos e à latência.

Em prática

Ao dominar as funções view e pure, você garante que suas aplicações descentralizadas sejam tão responsivas quanto as centralizadas, mas com a segurança e transparência da blockchain.

Próxima aula

Aula 20 – Escrevendo Transações a partir do Frontend: Você aprenderá a dar o próximo passo crucial: como o frontend pode iniciar transações que alteram o estado do contrato.

Autoavaliação

1

Qual a principal vantagem de utilizar funções view e pure em um Smart Contract quando chamadas por um frontend?

- a) Elas permitem que o contrato altere seu estado de forma mais rápida.
- b) Elas não consomem gás, tornando as interações gratuitas para o usuário.
- c) Elas garantem que apenas o proprietário do contrato possa acessar os dados.
- d) Elas são as únicas funções que podem ser chamadas de fora da blockchain.

2

Uma função pure em Solidity:

- a) Pode ler e modificar variáveis de estado do contrato.
- b) Pode ler variáveis de estado, mas não modificá-las.
- c) Não pode ler nem modificar variáveis de estado do contrato.
- d) É utilizada para enviar Ether para outros endereços.

3

No contexto de um frontend interagindo com um Smart Contract, o que é a ABI?

- a) O endereço único do contrato na blockchain.
- b) A linguagem de programação utilizada para escrever o contrato.
- c) Um arquivo que descreve as funções do contrato, seus parâmetros e retornos.
- d) A biblioteca JavaScript usada para conectar o frontend à blockchain.

4

Se uma função view retorna um valor uint256 em Solidity, como ele é geralmente representado em JavaScript ao usar ethers.js?

- a) Como um número JavaScript padrão (number).
- b) Como uma string formatada.
- c) Como um objeto BigNumber.
- d) Como um array de bytes.

5

Questão dissertativa

Explique a diferença fundamental entre uma função view e uma função pure em termos de acesso ao estado do Smart Contract e forneça um exemplo de uso para cada uma.

Gabarito e Recursos Adicionais

Gabarito

1 **Resposta:** b) Elas não consomem gás, tornando as interações gratuitas para o usuário.

2 **Resposta:** c) Não pode ler nem modificar variáveis de estado do contrato.

3 **Resposta:** c) Um arquivo que descreve as funções do contrato, seus parâmetros e retornos.

4 **Resposta:** c) Como um objeto BigNumber.

Recursos Adicionais

📚 Documentação oficial do Solidity

Para aprofundar nos modificadores de estado view e pure.

💻 Documentação da ethers.js

Para detalhes sobre a interação com contratos via JavaScript.

🔍 Exemplos de DApps open-source

Para ver como a leitura de dados é implementada em projetos reais.

NOTA IMPORTANTE: As informações regulatórias/legais/técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais para verificar alterações.