Aula 7 – Mapeamentos e Arrays: Armazenando Dados em Contratos

No universo dos smart contracts, a capacidade de armazenar e gerenciar dados de forma eficiente e segura é tão fundamental quanto a lógica que eles executam. Pense em qualquer aplicação que você usa diariamente: ela precisa guardar informações sobre usuários, transações, configurações e muito mais. Em um contrato inteligente, essa necessidade não é diferente, mas o ambiente da blockchain impõe desafios e peculiaridades únicas.

Imagine que seu contrato precisa lembrar quem é o proprietário de um token específico, ou quais usuários têm permissão para realizar certas ações. Como ele faria isso de maneira organizada e acessível, sem que cada busca custasse uma fortuna em taxas de rede? É aqui que entram os mapeamentos e os arrays, as ferramentas essenciais para construir as "bases de dados" dos seus contratos na blockchain.

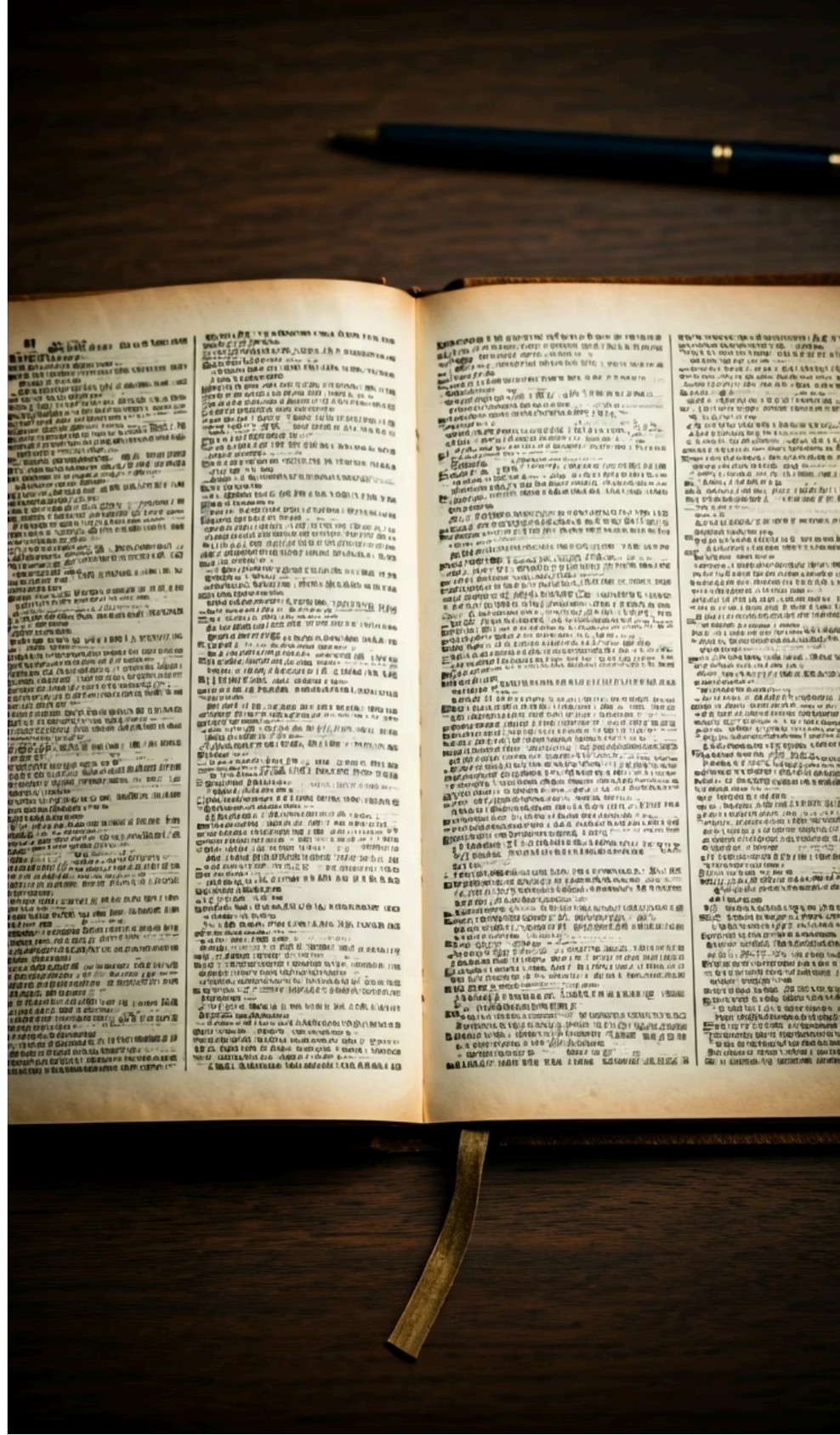
Nesta aula, vamos desvendar como essas estruturas funcionam, explorando suas características, vantagens e as melhores práticas para utilizá-las. Ao final, você será capaz de escolher a estrutura de dados mais adequada para diferentes cenários, otimizando o armazenamento e o custo de gás dos seus contratos, e construindo aplicações descentralizadas (DApps) mais robustas e eficientes. Prepare-se para dar um salto na sua jornada de desenvolvimento em Solidity!

Mapeamentos: A Chave para Associações Rápidas

Já pensou em como um contrato inteligente consegue associar um endereço de usuário ao seu saldo de tokens, ou a uma permissão específica? Se tivéssemos que percorrer uma lista gigante de todos os usuários a cada vez, seria inviável e caríssimo. Precisamos de uma forma de encontrar informações rapidamente, usando uma **"chave"** que nos leve diretamente ao **"valor"** desejado.

É exatamente para isso que serve a estrutura **mapping** em Solidity. Ela funciona como um dicionário ou uma lista telefônica, onde você tem uma chave única (como um nome ou um número de telefone) que aponta para um valor correspondente (o número de telefone ou o endereço). Em vez de procurar linha por linha, você vai direto ao que precisa.

> 💡 **Eficiência é tudo:** Essa estrutura é fundamental para a eficiência dos contratos, pois permite acessos de dados quase instantâneos, independentemente do tamanho do mapeamento. Isso é crucial em um ambiente onde cada operação tem um custo associado, o famoso "gás".



Mapeamentos em Ação e Suas Peculiaridades

Para entender melhor, vamos a um exemplo prático. Suponha que você esteja criando um contrato para um sistema de votação. Você precisa saber se um endereço já votou e qual foi o seu voto. Um mapping é perfeito para isso.

Chave

Endereço do usuário (address)

Associação

Mapeamento direto e eficiente

Valor

Dados do voto (bool, uint)

```
// Exemplo de Mapeamento para controle de votos
contract SistemaDeVotacao {
    mapping(address => bool) public jaVotou;
    mapping(address => uint) public votoDoUsuario;

    function votar(uint _opcaoVoto) public {
        require(!jaVotou[msg.sender], "Voce ja votou!");
        jaVotou[msg.sender] = true;
        votoDoUsuario[msg.sender] = _opcaoVoto;
        // ... lógica para contabilizar votos ...
    }
}
```

Características Importantes

⚠️ Não Iteráveis

Você não consegue listar todas as chaves ou todos os valores diretamente. Você só pode consultar um valor se souber a chave.

🔄 Valores Padrão

Mapeamentos sempre retornam um valor padrão para chaves não definidas (false para bool, 0 para uint, address(0) para address).

> **Segurança em Primeiro Lugar:** Ao usar mapeamentos para controle de acesso ou saldos, é vital garantir que as funções que modificam esses dados tenham as devidas permissões, talvez utilizando modificadores de função ou bibliotecas como a OpenZeppelin para AccessControl, que frequentemente usam mapeamentos internamente para gerenciar papéis.

Arrays: Coleções Ordenadas de Dados

Enquanto os mapeamentos são excelentes para associações chave-valor, há momentos em que precisamos de uma **lista ordenada** de itens, onde a posição importa ou onde queremos iterar sobre todos os elementos. Pense em uma lista de tarefas, um histórico de transações ou uma galeria de fotos. Para esses cenários, os arrays são a ferramenta ideal em Solidity.

Um array é como uma prateleira organizada, onde cada item tem um lugar específico (um índice numérico) e você pode acessá-lo por essa posição. Diferente de um mapeamento, onde a busca é por uma chave arbitrária, no array a busca é pela sua posição na sequência.

Tipos de Arrays

Arrays de Tamanho Fixo

Declarados com um número predefinido de elementos, como uma prateleira com um número exato de compartimentos.

Arrays Dinâmicos

Podem crescer ou encolher conforme a necessidade, adicionando ou removendo compartimentos.

```
// Exemplo de Array de tamanho fixo
contract ListaDeItensFixa {
 string[3] public nomesDeItens;

 constructor() {
 nomesDeItens[0] = "Maçã";
 nomesDeItens[1] = "Banana";
 nomesDeItens[2] = "Laranja";
 }

 function obterItem(uint _indice) public view returns (string memory) {
 require(_indice < nomesDeItens.length, "Indice fora dos limites!");
 return nomesDeItens[_indice];
 }
}
```

Arrays Dinâmicos: Flexibilidade na Blockchain

Flexibilidade

A vida real raramente se encaixa em caixas de tamanho fixo.

E se sua lista de tarefas crescer, ou se novos usuários se registrarem no seu DApp? É aí que os arrays dinâmicos brilham. Eles são como uma prateleira que pode se expandir ou encolher conforme a necessidade, adicionando ou removendo compartimentos.

Arrays dinâmicos são declarados sem um tamanho específico, e você pode adicionar (**push**) ou remover (**pop**) elementos, alterando seu tamanho em tempo de execução. Isso oferece uma flexibilidade enorme para gerenciar coleções de dados que mudam ao longo do tempo, como uma lista de participantes em um evento ou um histórico de transações.

```
// Exemplo de Array de tamanho dinâmico
contract ListaDeParticipantes {
    address[] public participantes;

    function adicionarParticipante(address _novoParticipante) public {
        participantes.push(_novoParticipante);
    }

    function removerUltimoParticipante() public {
        require(participantes.length > 0, "Nao ha participantes para remover.");
        participantes.pop();
    }

    function obterNumeroDeParticipantes() public view returns (uint) {
        return participantes.length;
    }
}
```

⚠️ **Atenção ao Custo:** A capacidade de adicionar e remover elementos torna os arrays dinâmicos extremamente versáteis. No entanto, é crucial lembrar que cada operação de escrita na blockchain tem um custo de gás. Adicionar muitos elementos ou remover repetidamente pode se tornar caro, especialmente se o array crescer demais.

Arrays em Detalhe: Acesso e Manipulação

01

Acesso por Índice

Os índices começam em 0. Para acessar o primeiro elemento: meuArray[0], segundo: meuArray[1]

02

Operação Push

Adiciona um elemento ao final do array dinâmico

03

Operação Pop

Remove o último elemento do array dinâmico

04

Palavra-chave Delete

Redefine o valor para padrão, mas não diminui o length

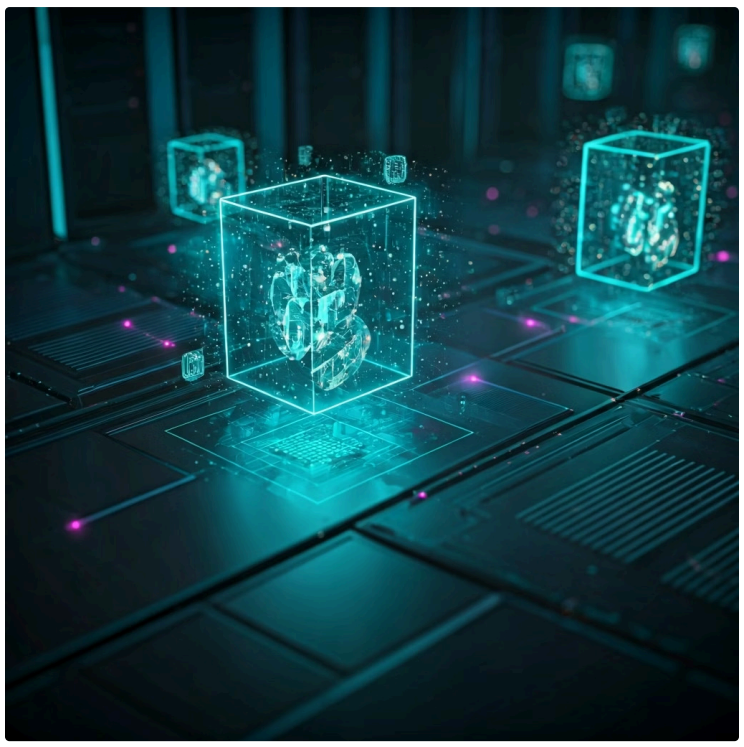
Remoção Eficiente de Elementos

Para realmente "remover" um elemento e encurtar o array, especialmente de uma posição intermediária, a estratégia comum é **mover o último elemento para a posição do elemento a ser removido** e então usar pop.

```
// Exemplo de manipulação de array
contract GerenciadorDeTarefas {
    string[] public tarefas;

    function adicionarTarefa(string memory _descricao) public {
        tarefas.push(_descricao);
    }

    function removerTarefa(uint _indice) public {
        require(_indice < tarefas.length, "Indice invalido.");
        if (_indice != tarefas.length - 1) {
            tarefas[_indice] = tarefas[tarefas.length - 1];
        }
        tarefas.pop();
    }

    function obterTodasAsTarefas() public view returns (string[] memory) {
        return tarefas;
    }
}
```

> ⚡ **Cuidado com Iterações:** É importante ter cuidado ao iterar sobre arrays muito grandes em funções view ou pure, pois mesmo que não custem gás para o chamador, podem exceder o limite de gás de bloco se o array for excessivamente grande, impedindo a execução da transação.

Armazenamento de Coleções na Blockchain: Considerações Cruciais



A blockchain não é um banco de dados tradicional. Cada byte de informação armazenado no estado de um contrato é replicado em milhares de nós ao redor do mundo e persiste para sempre. **Isso tem um custo, e esse custo é o gás.**

Tipos de Armazenamento

Storage

Variáveis de estado persistentes

Custo: Muito Alto 💰💰💰

Memory

Variáveis temporárias em funções

Custo: Baixo 💰

Calldata

Dados de entrada imutáveis

Custo: Muito Baixo 💚

A escolha entre mapping e array, e como você os utiliza, impacta diretamente o custo e a escalabilidade do seu DApp. Mapeamentos são eficientes para buscas diretas, mas não para iteração. Arrays são ótimos para listas ordenadas e iteração, mas podem se tornar caros se forem muito grandes e você precisar modificá-los ou iterar sobre eles em transações.

Conceito	Âmbito/Aplicação	Base/Origem	Exemplo
Mapping	Associações chave-valor, controle de acesso	Tabela hash/Dicionário	`balances[address] = uint`
Array Fixo	Coleções de tamanho conhecido e imutável	Lista ordenada	`string[5] nomes`
Array Dinâmico	Coleções de tamanho variável, histórico de eventos	Lista ordenada expansível	address[] participantes

Padrões de Uso e Segurança com Mapeamentos e Arrays

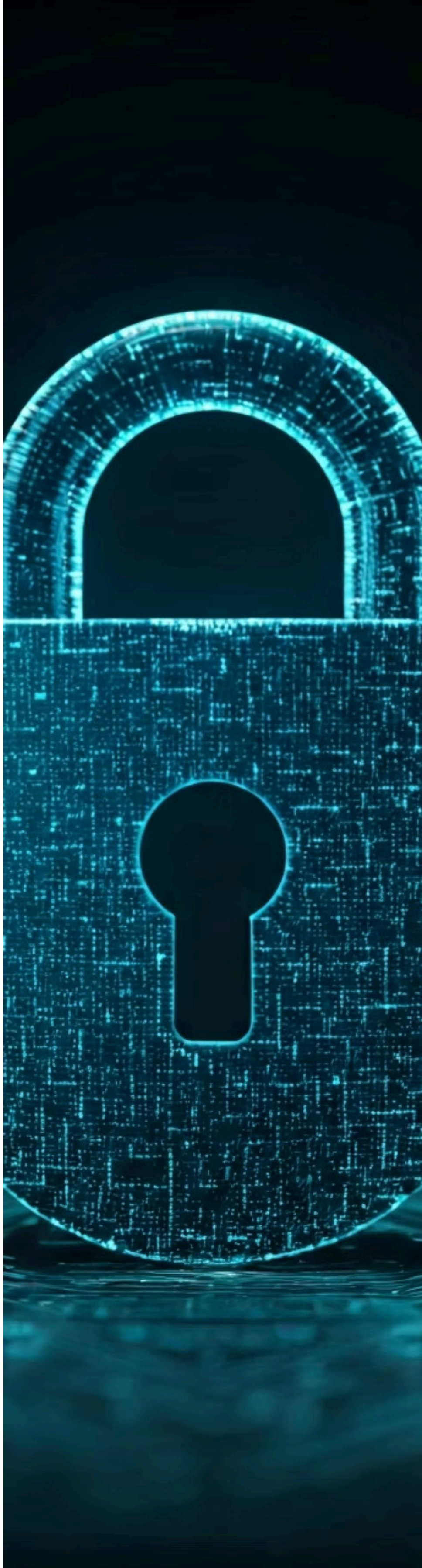
A forma como você estrutura seus dados pode ser a diferença entre um contrato seguro e um vulnerável. Mapeamentos e arrays são frequentemente usados em padrões de segurança cruciais. Por exemplo, um `mapping(address => bool)` é a base para muitos sistemas de controle de acesso, onde **true** significa que o endereço tem uma permissão específica.

Padrão Whitelist (Lista Branca)

Um padrão comum onde apenas endereços pré-aprovados podem interagir com certas funções. Isso pode ser implementado com um `mapping(address => bool) public isWhitelisted;`

```
// Exemplo de controle de acesso com mapping
contract AcessoRestrito {
    address public owner;
    mapping(address => bool) public administradores;

    constructor() {
        owner = msg.sender;
        administradores[msg.sender] = true;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Nao e o proprietario.");
        _;
    }

    modifier onlyAdmin() {
        require(administradores[msg.sender], "Nao e um
administrador.");
        _;
    }

    function adicionarAdmin(address _novoAdmin) public
onlyOwner {
        administradores[_novoAdmin] = true;
    }

    function funcaoRestrita() public onlyAdmin {
        // ... logica restrita a administradores ...
    }
}
```

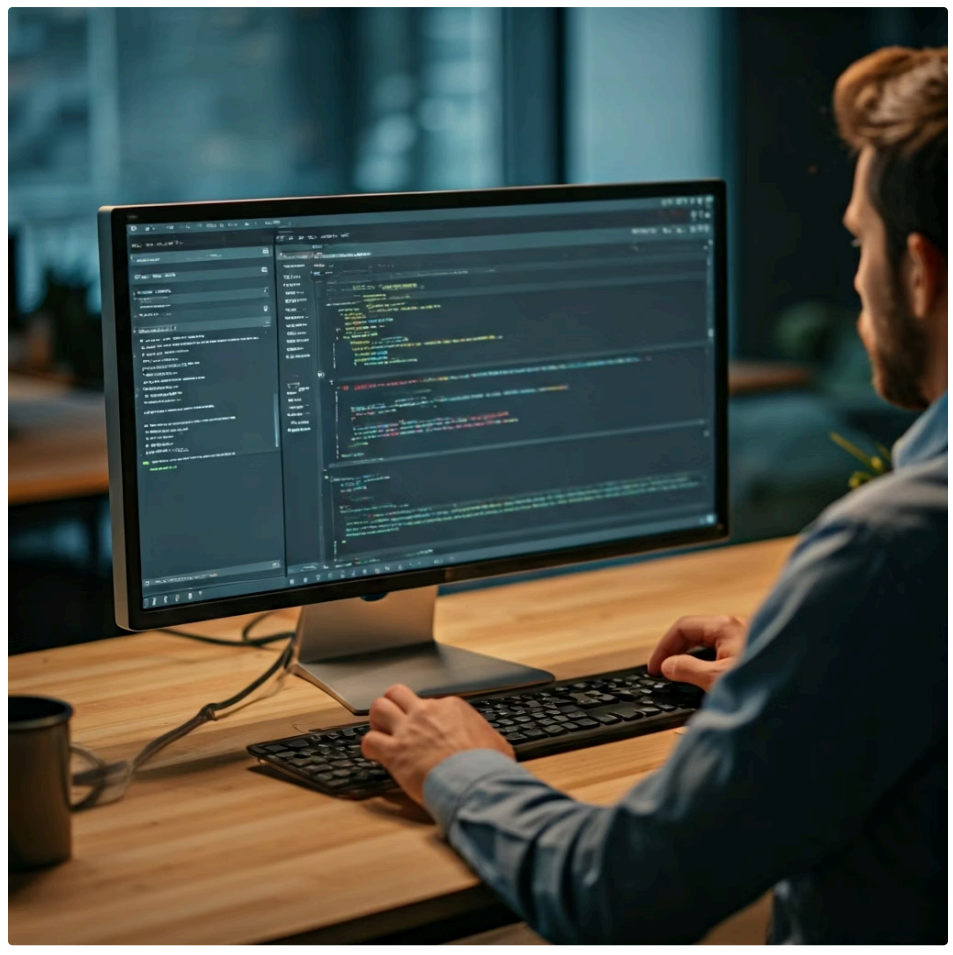
> ⚠️ **Cuidado com Loops:** Ao usar arrays, especialmente dinâmicos, tome cuidado com loops que iteram sobre eles. Se um array crescer muito, um loop pode consumir todo o gás do bloco, tornando a função inutilizável. É um erro comum que pode ser evitado pensando em limites ou em padrões de paginação para dados grandes.



Ferramentas Modernas e Otimização

🛠️ Framework Hardhat

No desenvolvimento de smart contracts, não basta apenas saber a sintaxe; é preciso usar as ferramentas certas para construir, testar e otimizar. O framework Hardhat é uma ferramenta moderna e amplamente adotada que facilita muito a vida do desenvolvedor ao lidar com mapping e arrays.



Capacidades do Hardhat

Simulação Local

Simule transações localmente antes de implantar na rede principal

Estimativa de Gás

Estime custos de gás para operações que envolvem mapping e arrays

Testes Robustos

Escreva testes para garantir que suas estruturas de dados se comportem como esperado

Estratégias de Otimização

Minimizar Escritas

Cada sstore (escrita na storage) é cara. Reduza ao máximo.

Evitar Iterações

Se precisar iterar, considere paginação ou lógica off-chain.

Usar View/Pure

Funções que apenas leem o estado são gratuitas para o chamador.

> 💡 **Dica Profissional:** Ao testar uma função que adiciona muitos itens a um array dinâmico, você pode usar Hardhat para verificar se o custo de gás permanece dentro de limites aceitáveis. Se não, você pode precisar repensar sua estratégia de armazenamento, talvez usando eventos para registrar dados que não precisam estar no estado do contrato, ou implementando soluções de armazenamento off-chain para grandes volumes de dados.

CONSOLIDAÇÃO

Recapitulando os Conceitos Fundamentais

Nesta aula, mergulhamos nas estruturas de dados fundamentais para qualquer smart contract: os mapeamentos e os arrays. Vimos que os **mapeamentos** são ideais para associações chave-valor, oferecendo acesso rápido e eficiente, como um dicionário. Já os **arrays**, sejam de tamanho fixo ou dinâmico, nos permitem gerenciar coleções ordenadas de dados, essenciais para listas e históricos.

Mapeamentos

- Associações chave-valor
- Acesso rápido e direto
- Não iteráveis
- Ideais para controle de acesso

Arrays

- Coleções ordenadas
- Fixos ou dinâmicos
- Iteráveis
- Ideais para listas e históricos

Segurança

- Controle de acesso robusto
- Validação de permissões
- Prevenção de DoS
- Uso de bibliotecas auditadas

Otimização

- Minimizar escritas na storage
- Evitar loops grandes
- Testar custos de gás
- Considerar soluções off-chain

Em Prática

Ao projetar seu próximo smart contract, comece pensando em como os dados serão armazenados. Precisa de uma busca rápida por um identificador único? Use um mapping. Precisa de uma lista ordenada que pode crescer? Um array dinâmico é a resposta. Lembre-se sempre das implicações de gás e das melhores práticas de segurança, como as oferecidas pela OpenZeppelin, e utilize ferramentas como Hardhat para testar e otimizar suas implementações.

Autoavaliação

1

Vantagem dos Mapeamentos

Qual a principal vantagem de usar um mapping em vez de um array para associar um endereço de usuário ao seu saldo de tokens?

1. Mapeamentos permitem iteração fácil sobre todas as chaves.
2. Mapeamentos são mais baratos para armazenar grandes volumes de dados.
3. **Mapeamentos oferecem acesso direto e eficiente a valores por meio de uma chave, sem a necessidade de iteração.**
4. Mapeamentos podem ter seu tamanho alterado dinamicamente com push e pop.

2

Arrays Fixos vs Dinâmicos

Em Solidity, qual a principal diferença entre um array de tamanho fixo e um array dinâmico?

1. Arrays fixos são mais caros em termos de gás.
2. Arrays dinâmicos não podem ser iterados.
3. **Arrays fixos têm um número predefinido de elementos que não pode ser alterado após a declaração, enquanto arrays dinâmicos podem crescer ou encolher.**
4. Arrays dinâmicos só podem armazenar tipos de dados primitivos.

3

Função Delete

Ao usar a função delete em um elemento de um array em Solidity, o que acontece?

1. O elemento é removido do array e o length do array diminui.
2. O elemento é removido do array, mas o length do array permanece o mesmo.
3. **O valor do elemento naquele índice é redefinido para seu valor padrão, e o length do array permanece o mesmo.**
4. Apenas arrays dinâmicos podem usar a função delete.

4

Otimização de Gás

Qual das seguintes práticas é recomendada para otimizar o custo de gás ao lidar com arrays e mapping em smart contracts?

1. Armazenar todos os dados possíveis diretamente no estado do contrato.
2. Iterar sobre arrays muito grandes em funções que modificam o estado.
3. **Minimizar as operações de escrita na storage e considerar soluções off-chain para grandes volumes de dados.**
4. Usar mapping para todas as coleções de dados, independentemente da necessidade de iteração.

5

Questão Dissertativa

Explique como a escolha entre mapping e array pode impactar a segurança de um smart contract, especialmente em relação ao controle de acesso e à prevenção de ataques de negação de serviço (DoS).

Gabarito e Próximos Passos

📋 Gabarito

1 **Resposta: c)**

2 **Resposta: c)**

3 **Resposta: c)**

4 **Resposta: c)**

🚀 Próxima Aula

Aula 8 – Structs, Enums e Modificadores de Função

Vamos expandir ainda mais suas ferramentas para modelar dados complexos e controlar o fluxo de execução dos seus contratos, combinando o que aprendemos hoje com novas estruturas e conceitos poderosos.

📚 Recursos Adicionais

Documentação Oficial do Solidity

Para aprofundar nos detalhes técnicos de mapping e arrays.

Documentação da OpenZeppelin

Para explorar padrões de segurança e contratos auditados que utilizam essas estruturas.

Documentação do Hardhat

Para aprender a testar e otimizar o uso de dados em seus contratos.

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