Aula 9 – Frameworks de Teste Avançado: Foundry (Parte 1)

Bem-vindos à nona aula do nosso curso, onde mergulharemos em uma ferramenta essencial para qualquer desenvolvedor blockchain sério: o Foundry. No universo dos contratos inteligentes, onde cada linha de código pode gerenciar milhões em valor e qualquer erro pode ser catastrófico e irreversível, a qualidade dos testes não é apenas uma boa prática, é uma necessidade absoluta. Pense na construção de um arranha-céu: você confiaria em uma estrutura sem testes rigorosos de cada pilar e viga? No blockchain, a analogia é ainda mais crítica, pois não há "desfazer" após a implantação.

Até agora, você provavelmente já se familiarizou com a lógica de programação e talvez até tenha explorado algumas linguagens como JavaScript ou Python. Mas, quando se trata de testar contratos escritos em Solidity, muitas ferramentas tradicionais podem parecer um desvio, exigindo que você mude de contexto e linguagem. Isso pode tornar o processo mais lento e propenso a erros. É aqui que o Foundry entra em cena, oferecendo uma abordagem revolucionária que alinha a linguagem de teste diretamente com a linguagem do contrato.

Nesta aula, nosso objetivo é desvendar o Foundry, capacitando você a configurar seu ambiente de desenvolvimento, escrever testes unitários eficientes e, o mais importante, manipular o estado da blockchain de forma controlada para simular cenários complexos. Ao final, você estará apto a utilizar o forge init para iniciar projetos, o forge test para executar seus testes e os poderosos *cheatcodes* para criar ambientes de teste realistas. Prepare-se para elevar a segurança e a robustez dos seus contratos inteligentes a um novo patamar, um passo fundamental para quem busca excelência no desenvolvimento blockchain.

O Cenário dos Testes em Blockchain: Por Que Foundry?

Imutabilidade

Contratos implantados não podem ser alterados, tornando bugs permanentes e potencialmente devastadores.

Alto Valor

Contratos gerenciam ativos digitais reais, onde vulnerabilidades podem resultar em perdas financeiras massivas.

Rede Global

Interações com uma rede descentralizada exigem testes que simulem cenários complexos e realistas.

No dinâmico e implacável mundo da blockchain, um contrato inteligente é mais do que apenas código; ele é um acordo autoexecutável que gerencia ativos digitais, define regras e interage com uma rede global. A imutabilidade desses contratos após a implantação significa que qualquer vulnerabilidade ou bug pode ter consequências financeiras devastadoras e irreparáveis. É por isso que a fase de testes não é um luxo, mas uma etapa crítica que exige as ferramentas mais eficientes e confiáveis disponíveis.

> **Ferramentas Tradicionais vs. Foundry:** Historicamente, ferramentas populares como Truffle e Hardhat dominaram o cenário de desenvolvimento e teste, oferecendo ecossistemas robustos e flexíveis. No entanto, muitas delas são construídas sobre bases JavaScript ou TypeScript, o que significa que, para testar um contrato Solidity, você precisa escrever seu código de teste em uma linguagem diferente.

Isso introduz uma camada de tradução mental e, por vezes, de complexidade, que pode desacelerar o desenvolvimento e dificultar a depuração. Imagine tentar conversar com alguém usando um tradutor em tempo real para cada frase – eficiente, mas não tão direto quanto falar a mesma língua.

É nesse contexto que o Foundry surge como uma alternativa poderosa e cada vez mais adotada, oferecendo uma filosofia de teste radicalmente diferente: **escrever testes diretamente em Solidity**. Essa abordagem "Solidity-native" elimina a barreira da linguagem, permitindo que os desenvolvedores pensem e escrevam seus testes na mesma sintaxe e lógica de seus contratos. O resultado é um ciclo de feedback mais rápido, maior clareza e uma integração mais profunda entre o código do contrato e seu respectivo teste, preparando o terreno para lidar com a crescente complexidade de dApps modernos, como aqueles que incorporam Abstração de Contas ou soluções de Escalabilidade de Camada 2.

Introdução ao Foundry: Testes Rápidos Escritos em Solidity

Imagine que você é um chef de cozinha e está desenvolvendo uma nova receita. Em vez de preparar o prato inteiro para provar um único ingrediente, seria muito mais eficiente provar cada ingrediente individualmente e fazer pequenos ajustes antes de combiná-los. No desenvolvimento de contratos inteligentes, o Foundry oferece essa mesma eficiência, permitindo que você "prove" cada parte do seu código de forma isolada e rápida, diretamente na linguagem que você já domina: Solidity.

01

Mesma Linguagem

Testes escritos em Solidity, eliminando a necessidade de alternar entre linguagens.

02

Ciclo Rápido

Feedback instantâneo durante o desenvolvimento, acelerando a iteração.

03

Integração Profunda

Lógica de teste espelha diretamente a lógica do contrato.

O Foundry é um conjunto de ferramentas para desenvolvimento de contratos inteligentes que se destaca por sua capacidade de permitir que os testes sejam escritos diretamente em Solidity. Isso significa que, em vez de alternar entre Solidity para o contrato e JavaScript para o teste, você permanece no mesmo ambiente de linguagem. Essa coesão não só acelera o processo de escrita e depuração, mas também torna os testes mais intuitivos e fáceis de manter, pois a lógica do teste espelha a lógica do contrato de forma mais direta.

> **Estrutura Básica de um Teste:** Para começar a escrever testes com Foundry, você cria um novo contrato Solidity que herda de `forge-std/Test.sol`. Dentro deste contrato de teste, qualquer função que comece com o prefixo `test` será automaticamente detectada e executada pelo `forge test`.

Além disso, você pode definir uma função `setUp()` que será executada antes de cada teste, permitindo que você configure o estado inicial do seu contrato ou implante dependências, garantindo que cada teste comece de um ponto de partida limpo e previsível. Essa estrutura simples, mas poderosa, é a base para a construção de uma suíte de testes robusta e eficiente.

Configurando o Ambiente: O Primeiro Passo com forge init

Antes de podermos construir qualquer coisa, precisamos de um local de trabalho bem organizado e com todas as ferramentas à mão. No mundo do Foundry, esse local de trabalho é o seu projeto, e a maneira mais rápida e eficiente de configurá-lo é usando o comando `forge init`. Pense no `forge init` como um kit de ferramentas inicial que já vem com tudo o que você precisa para começar a desenvolver e testar seus contratos inteligentes, sem a necessidade de configurar manualmente cada arquivo e pasta.

Componentes do Foundry

- **forge:** Compilação, teste e implantação de contratos
- **anvil:** Nó EVM local para desenvolvimento e teste
- **cast:** Ferramenta de linha de comando para interagir com contratos

Instalação Simples

O processo de instalação é surpreendentemente simples e direto, geralmente envolvendo um único comando no terminal que baixa e configura o ambiente para você.

Uma vez instalado, o comando `forge init` cria uma estrutura de projeto padrão, que inclui diretórios para seus contratos (`src`), seus testes (`test`), scripts de implantação (`script`) e bibliotecas externas (`lib`), além de um arquivo de configuração `foundry.toml`.

```
# 1. Instalar o Foundry (se ainda não tiver)
curl -L https://foundry.paradigm.xyz | bash
foundryup

# 2. Criar um novo projeto Foundry
mkdir meu-projeto-foundry
cd meu-projeto-foundry
forge init
```

> **Vantagem da Estrutura Pré-definida:** Essa estrutura pré-definida é uma das grandes vantagens do Foundry, pois segue as melhores práticas da comunidade e permite que você se concentre imediatamente na escrita do código, em vez de gastar tempo organizando o projeto. É como receber um carro novo com o motor já ligado e o GPS configurado para o seu destino: você só precisa sentar e dirigir.

Essa agilidade é crucial em um ambiente de desenvolvimento rápido como o blockchain, onde a capacidade de iterar e testar rapidamente pode ser a diferença entre o sucesso e o fracasso de um projeto.

Anatomia de um Projeto Foundry e Primeiros Testes

Com o `forge init` executado, seu novo projeto Foundry agora tem uma estrutura de diretórios bem definida. Entender essa anatomia é fundamental para navegar e organizar seu código de forma eficaz. Pense nisso como aprender o layout de uma nova casa: saber onde fica a cozinha, o quarto e a sala de estar torna a vida muito mais fácil. No Foundry, cada diretório tem um propósito específico, otimizado para o fluxo de trabalho de desenvolvimento de contratos inteligentes.

src/

O coração do seu projeto, onde seus contratos inteligentes em Solidity residem. Cada arquivo .sol aqui representa um contrato ou uma biblioteca que você está desenvolvendo.

test/

Aqui é onde você escreverá todos os seus testes para os contratos em src/. Os arquivos de teste também são contratos Solidity, mas com o propósito exclusivo de verificar a funcionalidade do seu código principal.

script/

Este diretório é para scripts de implantação ou outras operações on-chain que você pode querer automatizar. Eles também são escritos em Solidity e podem ser executados com forge script.

lib/

Contém as bibliotecas externas que seu projeto utiliza, como forge-std (a biblioteca padrão do Foundry para testes) ou outras dependências que você pode adicionar via forge install.

foundry.toml

O arquivo de configuração principal do seu projeto, onde você pode definir opções de compilação, redes, e outras configurações.

Exemplo: Contrato Counter e Seu Teste

Para ilustrar, vamos criar um contrato simples e seu teste correspondente. Imagine um contrato `Counter.sol` em `src/` que apenas incrementa um número. Em `test/`, você criaria um `Counter.t.sol` para testar se o incremento funciona corretamente.

Counter.sol (src/)

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

contract Counter {
    uint256 public number;

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

    function increment() public {
        number++;
    }
}
```

Counter.t.sol (test/)

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/Counter.sol";

contract CounterTest is Test {
    Counter public counter;

    function setUp() public {
        counter = new Counter();
        counter.setNumber(0);
    }

    function testIncrement() public {
        counter.increment();
        assertEq(counter.number(), 1);
    }

    function testSetNumber() public {
        counter.setNumber(123);
        assertEq(counter.number(), 123);
    }
}
```

Essa separação clara de responsabilidades entre código de produção e código de teste é uma prática recomendada que facilita a manutenção e a escalabilidade do projeto. Ao manter seus testes próximos ao código que eles verificam, você garante que a suíte de testes evolua junto com o contrato, mantendo a integridade e a segurança.

Escrevendo Testes Unitários com forge test – Fundamentos

Os testes unitários são a espinha dorsal de qualquer software confiável, e no desenvolvimento de contratos inteligentes, eles são absolutamente cruciais. Eles se concentram em verificar a menor unidade de funcionalidade possível – uma função ou método específico – em isolamento, garantindo que cada peça do seu contrato se comporte exatamente como esperado. Pense em um relojoeiro que testa cada engrenagem individualmente antes de montá-las no mecanismo complexo de um relógio; cada engrenagem precisa funcionar perfeitamente para que o relógio marque a hora certa.

Escrever Funções de Teste

Adicione funções que começam com `test` no seu contrato de teste.

Usar Asserções

Utilize asserções como `assertEq()`, `assertTrue()`, `assertFalse()` para verificar condições.

Configurar Estado Inicial

Use a função `setUp()` para preparar o ambiente antes de cada teste.

Executar Testes

Digite `forge test` no terminal para compilar e executar todos os testes.

Com o Foundry, escrever testes unitários é uma experiência fluida e intuitiva, graças à sua abordagem Solidity-native. Uma vez que você tem seu contrato de teste configurado (como vimos na página anterior), você pode começar a adicionar funções de teste que começam com `test`. Dentro dessas funções, você usará as asserções fornecidas pela biblioteca `forge-std` para verificar as condições. As asserções são como perguntas que você faz ao seu código: "Este valor é igual a X?", "Esta condição é verdadeira?". Se a resposta for a esperada, o teste passa; caso contrário, ele falha, indicando um problema no seu contrato.

Asserções Comuns:

- `assertEq(a, b)` - Verifica se dois valores são iguais
- `assertTrue(condition)` - Verifica se uma condição é verdadeira
- `assertFalse(condition)` - Verifica se uma condição é falsa

Exemplo: Testando uma Calculadora

```
// test/Calculator.t.sol (assumindo um contrato Calculator.sol em src/)
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/Calculator.sol";

contract CalculatorTest is Test {
    Calculator public calculator;

    function setUp() public {
        calculator = new Calculator();
    }

    function testAdd() public {
        calculator.add(5, 3);
        assertEq(calculator.getResult(), 8, "A soma de 5 e 3 deveria ser 8.");
    }

    function testSubtract() public {
        calculator.subtract(10, 4);
        assertEq(calculator.getResult(), 6, "A subtração de 10 por 4 deveria ser 6.");
    }
}
```

A função `setUp()` é sua aliada aqui, pois permite que você configure o estado inicial do seu contrato antes de cada teste, garantindo que cada teste seja independente e reproduzível. Para executar seus testes, basta navegar até a raiz do seu projeto no terminal e digitar `forge test`. O Foundry compilará e executará todos os testes, fornecendo um feedback claro sobre quais passaram e quais falharam.

Aprofundando em forge test: Convenções e Boas Práticas

Dominar os fundamentos do `forge test` é um excelente começo, mas para construir suítes de testes robustas e fáceis de manter, é essencial ir além do básico e adotar convenções e boas práticas. Assim como em qualquer linguagem de programação, a clareza e a consistência no código de teste são tão importantes quanto no código de produção. Elas garantem que outros desenvolvedores (e seu eu futuro!) possam entender rapidamente o propósito de cada teste e depurar problemas de forma eficiente.

test_

Para testes unitários regulares que verificam o comportamento esperado.

testFail_

Para testes que esperam que uma transação falhe (reverta). O Foundry automaticamente passa o teste se a transação reverter.

testFuzz_

Para testes de fuzzing, onde a função aceita parâmetros que o Foundry irá gerar aleatoriamente para testar uma ampla gama de entradas.

Convenções de Nomenclatura

Uma das convenções mais importantes no Foundry diz respeito à nomenclatura das funções de teste. Além do prefixo `test` para testes unitários padrão, existem outros prefixos que comunicam a intenção do teste.

Simulando Diferentes Chamadores

Outra prática crucial é a simulação de diferentes chamadores de contrato. Contratos inteligentes frequentemente têm lógica que depende de quem está chamando uma função (por exemplo, apenas o proprietário pode pausar o contrato). O Foundry oferece os *cheatcodes* `vm.startPrank(address who)` e `vm.stopPrank()` (ou o mais simples `vm.prank(address who)`) para simular que uma transação está sendo enviada por um endereço específico.

> **Analogia:** Isso é como ter um ator que pode mudar de personagem instantaneamente para testar diferentes cenários de permissão. Essa capacidade é vital para garantir que as regras de acesso e autorização do seu contrato funcionem conforme o esperado, especialmente em dApps complexos que interagem com múltiplos usuários e outros contratos.

Exemplo Prático

```
// test/AccessControl.t.sol
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/AccessControl.sol";

contract AccessControlTest is Test {
 AccessControl public ac;
 address public OWNER = makeAddr("owner");
 address public NON_OWNER = makeAddr("non_owner");

 function setUp() public {
 vm.prank(OWNER);
 ac = new AccessControl();
 }

 function testOnlyOwnerCanPause() public {
 vm.prank(OWNER);
 ac.pause();
 assertTrue(ac.paused(), "Contrato deveria estar pausado pelo owner.");
 }

 function testFailNonOwnerCannotPause() public {
 vm.prank(NON_OWNER);
 ac.pause(); // Espera-se que esta chamada reverta
 }
}
```

Introdução aos Cheatcodes: Manipulando o Estado da Blockchain

Testar contratos inteligentes de forma abrangente muitas vezes exige mais do que apenas chamar funções e verificar resultados. É preciso simular cenários complexos que envolvem tempo, saldos de contas, interações com outros contratos e até mesmo a manipulação direta do estado da blockchain. No entanto, replicar essas condições em uma rede real seria impraticável, demorado e caro. É aqui que os ***cheatcodes*** do Foundry se tornam ferramentas indispensáveis, agindo como um "modo Deus" para seus testes.

Controle de Tempo

Avance ou retroceda o timestamp da blockchain para testar lógica sensível ao tempo sem esperar.

Gerenciamento de Saldos

Defina o saldo de ETH de qualquer endereço instantaneamente para simular diferentes cenários financeiros.

Simulação de Usuários

Faça com que transações sejam enviadas por endereços específicos para testar permissões e autorização.

Manipulação de Código

Implante bytecode arbitrário em qualquer endereço para criar contratos mock ou simular dependências.

Os cheatcodes são funções especiais fornecidas pelo ambiente de execução EVM (Ethereum Virtual Machine) do Foundry, o Anvil. Eles permitem que você altere o estado da blockchain de maneiras que seriam impossíveis ou muito difíceis em uma rede pública. Pense neles como superpoderes que você pode usar durante seus testes: você pode viajar no tempo, dar dinheiro a qualquer endereço, ou até mesmo mudar o código de um contrato já implantado. Essa capacidade de manipular o ambiente de teste com precisão cirúrgica é o que torna o Foundry tão potente para testes avançados.

> **"A principal vantagem dos cheatcodes é a capacidade de criar cenários de teste realistas e extremos sem a necessidade de esperar por blocos, minerar transações ou gastar ETH real."**

Isso significa que você pode testar condições de corrida, vulnerabilidades de re-entrância, lógica sensível ao tempo, e interações complexas entre múltiplos contratos e usuários de forma rápida e determinística. Eles são a ponte entre a teoria do seu contrato e a prática de como ele se comportaria em um ambiente real, permitindo que você explore cada canto e fenda da sua lógica com confiança.

Cheatcodes Essenciais: vm.warp() e vm.roll()

No desenvolvimento de contratos inteligentes, o tempo e o número do bloco são variáveis cruciais que afetam a lógica de muitos protocolos. Contratos de vesting, timelocks, leilões, e até mesmo alguns mecanismos de governança dependem de timestamps ou números de bloco para determinar quando certas ações podem ser executadas. Testar essas funcionalidades de forma eficaz em um ambiente real seria extremamente lento, exigindo que você esperasse por minutos, horas ou até dias para que as condições de tempo fossem atendidas.

vm.warp(uint256 newTimestamp)

Esta função permite que você defina o timestamp do próximo bloco a ser minerado. Se você precisa testar uma função que só pode ser chamada após uma semana, basta "avançar" o tempo em uma semana com `vm.warp()`.

Exemplo: `vm.warp(block.timestamp + 7 days);`

vm.roll(uint256 newBlockNumber)

Similarmente, `vm.roll()` permite que você defina o número do próximo bloco. Isso é útil para contratos que dependem de um número de bloco específico para ativar certas funcionalidades.

Exemplo: `vm.roll(block.number + 100);`

É aqui que os cheatcodes `vm.warp()` e `vm.roll()` se tornam indispensáveis. Eles permitem que você manipule o tempo e o número do bloco da EVM local do Foundry (Anvil) instantaneamente, sem ter que esperar.

> Pense nesses cheatcodes como um controle remoto de "viagem no tempo" para sua blockchain de teste. Você pode pular para o futuro para testar um evento que só ocorreria daqui a um mês, ou voltar no tempo (com cuidado, pois isso pode afetar o estado) para re-executar um cenário.

Exemplo Prático: Testando um Timelock

```
// test/Timelock.t.sol
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/Timelock.sol";

contract TimelockTest is Test {
    Timelock public timelock;
    uint256 public constant DELAY = 1 weeks;

    function setUp() public {
        timelock = new Timelock(DELAY);
    }

    function testFailExecuteBeforeDelay() public {
        // Tenta executar imediatamente, o que deve falhar
        timelock.execute();
    }

    function testExecuteAfterDelay() public {
        // Avança o tempo para depois do delay
        vm.warp(block.timestamp + DELAY + 1);
        timelock.execute();
        assertTrue(timelock.executed(), "A função deveria ter sido executada após o delay.");
    }

    function testExecuteAtSpecificBlock() public {
        // Avança o número do bloco
        vm.roll(block.number + 100);
        // ... (chamada de função relevante)
    }
}
```

Essa capacidade de controlar o tempo e o progresso da blockchain é fundamental para testar a robustez de protocolos DeFi, cronogramas de lançamento de NFTs, ou qualquer lógica que dependa de marcos temporais, garantindo que seu contrato se comporte corretamente sob diferentes condições de tempo.

Cheatcodes para Endereços e Balanços: vm.deal() e vm.prank()

Em contratos inteligentes, as interações frequentemente envolvem diferentes usuários e seus respectivos saldos de Ether (ETH). Testar cenários onde um usuário tem fundos insuficientes, ou onde apenas um endereço específico (como o proprietário) pode executar uma ação, é crucial para a segurança e a funcionalidade do seu dApp. No entanto, em um ambiente de teste padrão, gerenciar múltiplos endereços e seus saldos pode ser tedioso e propenso a erros.

vm.deal()

Define o saldo de ETH de qualquer endereço para um valor específico.

vm.prank()

Simula que a próxima transação está sendo enviada por um endereço específico.

Teste de Segurança

Combine ambos para testar permissões e lógica de pagamento com precisão.

Detalhamento dos Cheatcodes

vm.deal(address who, uint256 newBalance)

Este cheatcode permite que você defina o saldo de ETH de qualquer endereço para um valor específico. Precisa testar uma função payable com um usuário que tem 10 ETH? Basta usar `vm.deal()` para dar a ele essa quantia.

É como ter um caixa eletrônico ilimitado para seus testes, permitindo que você configure qualquer cenário de saldo sem a necessidade de enviar transações reais.

vm.prank() / vm.startPrank() / vm.stopPrank()

Estes cheatcodes são usados para simular que uma transação está sendo enviada por um endereço específico.

- `vm.prank(who)` - Faz com que a *próxima* chamada seja feita por `who`
- `vm.startPrank(who)` - Inicia uma sessão onde *todas* as chamadas são feitas por `who`
- `vm.stopPrank()` - Encerra a sessão de prank

> **Analogia:** Isso é como um ator que pode vestir diferentes máscaras para interpretar múltiplos personagens em uma peça, testando como o contrato reage a cada um.

Exemplo Prático: Gateway de Pagamento

```
// test/PaymentGateway.t.sol
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/PaymentGateway.sol";

contract PaymentGatewayTest is Test {
    PaymentGateway public gateway;
    address public BUYER = makeAddr("buyer");
    address public SELLER = makeAddr("seller");
    uint256 public constant ITEM_PRICE = 1 ether;

    function setUp() public {
        vm.prank(SELLER);
        gateway = new PaymentGateway();
    }

    function testSuccessfulPurchase() public {
        vm.deal(BUYER, ITEM_PRICE); // Dá ao comprador o ETH necessário
        vm.prank(BUYER); // Comprador faz a chamada
        gateway.purchase{value: ITEM_PRICE}();

        assertEq(address(gateway).balance, ITEM_PRICE, "Gateway deveria ter recebido o pagamento.");
        assertEq(gateway.itemCount(BUYER), 1, "Comprador deveria ter 1 item.");
    }

    function testFailInsufficientFunds() public {
        vm.deal(BUYER, ITEM_PRICE - 1 wei); // Comprador com fundos insuficientes
        vm.prank(BUYER);
        gateway.purchase{value: ITEM_PRICE}(); // Espera-se que esta chamada reverta
    }
}
```

A combinação de `vm.deal()` e `vm.prank()` é extremamente poderosa. Você pode, por exemplo, simular que um usuário sem fundos tenta comprar um item, ou que um usuário não autorizado tenta chamar uma função restrita. Essa capacidade de controlar quem está interagindo com o contrato e com quais recursos é vital para garantir que as permissões e a lógica de pagamento do seu contrato funcionem conforme o esperado, protegendo seus usuários e seus ativos.

Cheatcodes Avançados: vm.etch() e vm.load()

À medida que os contratos inteligentes se tornam mais complexos, eles frequentemente interagem com outros contratos, oráculos, ou até mesmo precisam de uma maneira de simular contratos que ainda não foram implantados ou que possuem um comportamento específico. Além disso, em cenários de depuração avançada, pode ser necessário inspecionar ou até mesmo modificar diretamente o armazenamento de um contrato. Para essas situações, o Foundry oferece cheatcodes mais avançados que fornecem um controle ainda maior sobre o ambiente EVM.

vm.etch(address addr, bytes code)

Permite que você implante *qualquer* bytecode em *qualquer* endereço. Incrivelmente útil para criar contratos "mock" (simulados) para dependências externas.

vm.load(address account, bytes32 slot)

Permite que você leia o valor de um slot de armazenamento específico de um contrato. Útil para verificar o estado interno sem precisar de uma função pública.

vm.store(address account, bytes32 slot, bytes32 value)

Permite que você escreva um valor diretamente em um slot de armazenamento de um contrato. Ferramenta poderosa para forçar estados específicos.

Casos de Uso

Contratos Mock com vm.etch()

Em vez de implantar um contrato complexo de oráculo real para cada teste, você pode usar `vm.etch()` para implantar um contrato simples que apenas retorna valores predefinidos, isolando o teste do seu contrato principal.

> É como ter um "faz-tudo" que pode construir qualquer tipo de edifício em qualquer local para seus testes.

Depuração com vm.load() e vm.store()

Esses cheatcodes são ferramentas de nível cirúrgico que permitem aos desenvolvedores simular ambientes complexos, testar interações com contratos externos de forma controlada e depurar problemas de armazenamento que seriam quase impossíveis de identificar de outra forma.

> Esses cheatcodes avançados são ferramentas de nível cirúrgico que permitem aos desenvolvedores simular ambientes complexos, testar interações com contratos externos de forma controlada e depurar problemas de armazenamento que seriam quase impossíveis de identificar de outra forma.

Eles são particularmente úteis ao testar cenários de upgrade de contratos, onde você precisa interagir com diferentes versões de um contrato ou simular a migração de dados. A capacidade de manipular diretamente o bytecode e o armazenamento oferece um nível de controle sem precedentes, permitindo que você teste até mesmo os cenários mais extremos e complexos com confiança.

Integrando Cheatcodes em Testes Complexos

A verdadeira força dos cheatcodes do Foundry não reside em seu uso isolado, mas na capacidade de combiná-los para orquestrar cenários de teste que espelham a complexidade do mundo real. Contratos inteligentes raramente operam em vácuo; eles interagem com múltiplos usuários, dependem de condições de tempo, e podem ter lógicas que mudam com base em saldos ou eventos externos. Testar essas interações multifacetadas exige uma abordagem coordenada, onde cada cheatcode atua como uma peça em um quebra-cabeça maior.

01

Configurar Usuários e Saldos

Use `vm.deal()` e `vm.prank()` para criar usuários com fundos específicos.

02

Simular Interações

Execute chamadas de contrato simulando diferentes usuários e cenários.

03

Manipular Tempo

Avance o tempo com `vm.warp()` para testar lógica temporal.

04

Mockar Dependências

Use `vm.etch()` para simular contratos externos ou oráculos.

05

Verificar Estado Final

Use asserções para confirmar que o contrato está no estado esperado.

Exemplo: Protocolo de Empréstimo

Imagine que você está testando um protocolo de empréstimo descentralizado. Você precisa simular um usuário depositando fundos, outro usuário pegando um empréstimo, o tempo avançando para que os juros se acumulem, e talvez um terceiro usuário tentando resgatar fundos após um evento de liquidação.

```
// test/LendingProtocol.t.sol
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/LendingProtocol.sol";

contract LendingProtocolTest is Test {
    LendingProtocol public protocol;
    address public LENDER = makeAddr("lender");
    address public BORROWER = makeAddr("borrower");
    uint256 public constant DEPOSIT_AMOUNT = 10 ether;
    uint256 public constant LOAN_AMOUNT = 5 ether;
    uint256 public constant INTEREST_PERIOD = 7 days;

    function setUp() public {
        vm.prank(LENDER);
        protocol = new LendingProtocol();
    }

    function testLoanLifecycle() public {
        // 1. LENDER deposita fundos
        vm.deal(LENDER, DEPOSIT_AMOUNT);
        vm.prank(LENDER);
        protocol.deposit{value: DEPOSIT_AMOUNT}();
        assertEq(protocol.getAvailableLiquidity(), DEPOSIT_AMOUNT, "Liquidez incorreta.");

        // 2. BORROWER pega um empréstimo
        vm.prank(BORROWER);
        protocol.borrow(LOAN_AMOUNT);
        assertEq(protocol.getLoanBalance(BORROWER), LOAN_AMOUNT, "Empréstimo incorreto.");

        // 3. Avança o tempo para acumular juros
        vm.warp(block.timestamp + INTEREST_PERIOD);

        // 4. BORROWER tenta pagar o empréstimo (com juros)
        uint256 expectedRepayment = LOAN_AMOUNT + protocol.calculateInterest(LOAN_AMOUNT,
INTEREST_PERIOD);
        vm.deal(BORROWER, expectedRepayment);
        vm.prank(BORROWER);
        protocol.repay{value: expectedRepayment}();
        assertEq(protocol.getLoanBalance(BORROWER), 0, "Empréstimo não foi pago.");
    }
}
```

Orquestração de Testes: Essa orquestração de cheatcodes permite que você crie uma "história" de teste, onde cada etapa simula uma ação ou um evento na blockchain. É como dirigir uma peça de teatro, onde você controla cada ator (endereço), o cenário (estado do contrato), e o tempo (timestamp do bloco).

Essa capacidade de simular sequências complexas de eventos é inestimável para identificar bugs que só se manifestariam sob condições específicas, garantindo que seu contrato seja resiliente e seguro em uma ampla gama de cenários operacionais.

Foundry e as Tendências Atuais: Preparando-se para o Futuro

O ecossistema blockchain está em constante evolução, com novas tecnologias e padrões emergindo rapidamente. Para um framework de teste ser relevante, ele precisa ser capaz de acompanhar essas tendências, permitindo que os desenvolvedores testem as inovações mais recentes. O Foundry, com sua flexibilidade e poder de manipulação do EVM, está excepcionalmente posicionado para lidar com os desafios de teste impostos pelas tendências de 2025 e além.

Abstração de Contas (ERC-4337)

Carteiras de smart contracts que melhoram a UX, permitindo funcionalidades como pagamentos de taxas por terceiros e autenticação multifator.

Soluções de Escalabilidade (Layer 2)

Optimistic Rollups e ZK-Rollups que aumentam a capacidade da blockchain mantendo a segurança.

Interoperabilidade Cross-Chain

Protocolos que permitem comunicação e transferência de valor entre diferentes blockchains.

Testando Abstração de Contas

Uma das tendências mais significativas é a **Abstração de Contas (ERC-4337)**. Este padrão visa melhorar a experiência do usuário (UX) em dApps, permitindo carteiras de smart contracts que não exigem gerenciamento de *seed phrases* e possibilitam funcionalidades como pagamentos de taxas por terceiros ou autenticação multifator.

Desafios de Teste

- Simular diferentes *paymasters*
- Testar *bundlers* e suas interações
- Validar lógicas de assinatura complexas
- Verificar fluxos de autorização

Soluções com Foundry

O Foundry, com seus cheatcodes como `vm.prank()` para simular diferentes chamadores e `vm.deal()` para gerenciar saldos de *paymasters*, oferece o controle granular necessário para testar esses fluxos intrincados com precisão.

Testando Soluções Layer 2

Outra área de inovação são as **Soluções de Escalabilidade (Layer 2)**, como Optimistic Rollups (Arbitrum, Optimism) e ZK-Rollups (zkSync, StarkNet). Embora o Foundry teste em um EVM local (Anvil), a capacidade de simular interações entre a Layer 1 e a Layer 2 é crucial.

> **Casos de Uso:** Testar a lógica de depósitos e saques de pontes entre L1 e L2, ou a validação de provas de L2 na L1. A velocidade do Foundry e a capacidade de `vm.etch()` para mockar contratos de ponte ou validadores na L1 permitem que os desenvolvedores testem os componentes críticos que interagem com essas soluções de escalabilidade.

Isso garante que a lógica de segurança e transferência de ativos funcione conforme o esperado, mesmo em um ambiente multi-camadas.

Interoperabilidade e Cross-Chain: O Papel dos Testes Avançados

O futuro da blockchain é inegavelmente multi-chain. À medida que o ecossistema amadurece, a necessidade de comunicação e transferência de valor entre diferentes redes blockchain se torna cada vez mais premente. Protocolos de **Interoperabilidade e Cross-Chain**, como Chainlink CCIP e LayerZero, são a vanguarda dessa tendência, permitindo que contratos inteligentes em uma blockchain interajam com contratos ou dados em outra. No entanto, essa interconexão introduz uma camada exponencial de complexidade e, consequentemente, de riscos de segurança.

Desafio Multi-Chain

Testar contratos que dependem de mensagens cross-chain ou transferências de ativos entre cadeias é um desafio monumental.

Complexidade Distribuída

Você precisa considerar latência, segurança das mensagens, validação de provas e consistência do estado em ambientes distribuídos.

Solução com Foundry

Mockar e simular interações cross-chain usando cheatcodes avançados para isolar e controlar as entradas e saídas.

Estratégias de Teste Cross-Chain

Mock de Validadores

Use `vm.etch()` para implantar um contrato "mock" que simula o comportamento de um validador de mensagens cross-chain.

Simulação de Estado

Use `vm.store()` para forçar um estado que represente uma mensagem recebida de outra cadeia.

Teste de Pontes

`vm.prank()` pode simular a chamada de um contrato de ponte, testando a lógica de recepção.

> Testar contratos que dependem de mensagens cross-chain ou de transferências de ativos entre cadeias é um desafio monumental. Você não está mais apenas testando a lógica de um contrato em uma única EVM; você precisa considerar a latência, a segurança das mensagens, a validação de provas e a consistência do estado em ambientes distribuídos.

É aqui que o Foundry e seus cheatcodes avançados demonstram seu valor inestimável. Embora o Foundry teste em uma única instância EVM local, ele oferece as ferramentas para *mockar* e *simular* as interações cross-chain. Essa capacidade de isolar e controlar as entradas e saídas de interações cross-chain permite que os desenvolvedores testem a lógica interna de seus contratos de forma robusta, garantindo que eles reajam corretamente a mensagens e eventos externos, mesmo que a complexidade total do ambiente multi-chain não seja replicada.

Vantagem Estratégica: Ao dominar essas técnicas de teste avançado, você estará preparado para desenvolver e manter contratos que operam em um ecossistema blockchain cada vez mais interconectado e complexo.

Consolidação e Próximos Passos

Chegamos ao fim da primeira parte da nossa jornada com o Foundry, e você já deu um passo gigantesco para se tornar um desenvolvedor blockchain mais seguro e eficiente. Vimos que o Foundry é uma ferramenta revolucionária que permite escrever testes diretamente em Solidity, eliminando a barreira da linguagem e acelerando o ciclo de desenvolvimento. Exploramos como configurar um projeto com `forge init`, como escrever testes unitários eficazes com `forge test` e, crucialmente, como os poderosos *cheatcodes* nos permitem manipular o estado da blockchain para simular cenários complexos e realistas.

5

Cheatcodes Principais

Aprendidos para manipular tempo, saldos, usuários e código

3

Tendências Cobertas

Account Abstraction, Layer 2, e Cross-Chain

100%

Solidity-Native

Testes escritos na mesma linguagem dos contratos

Recapitulação dos Conceitos-Chave

A capacidade de controlar o tempo (`vm.warp`, `vm.roll`), os saldos e os chamadores (`vm.deal`, `vm.prank`), e até mesmo o bytecode e o armazenamento de contratos (`vm.etch`, `vm.load`, `vm.store`) é um superpoder que eleva a qualidade dos seus testes a um novo patamar. Essa precisão é fundamental para lidar com as complexidades das tendências atuais, como a Abstração de Contas, as Soluções de Escalabilidade de Camada 2 e a Interoperabilidade Cross-Chain, garantindo que seus contratos sejam robustos e seguros em um ecossistema em constante evolução.

Em Prática

- Sempre comece um novo projeto de contrato inteligente com `forge init` para uma estrutura organizada.
- Escreva testes unitários para cada função crítica do seu contrato, usando `assertEq`, `assertTrue`, etc.
- Utilize `setUp()` para garantir um estado inicial limpo para cada teste.
- Não hesite em usar `vm.prank()` para simular diferentes usuários e `vm.deal()` para gerenciar saldos em seus testes.
- Para lógica sensível ao tempo, `vm.warp()` e `vm.roll()` são seus melhores amigos.

Autoavaliação

Questões

1

Vantagem do Foundry

Qual é a principal vantagem do Foundry em comparação com frameworks de teste baseados em JavaScript/TypeScript para contratos Solidity?

1. Maior velocidade de compilação de JavaScript.
2. Capacidade de escrever testes diretamente em Solidity.
3. Integração nativa com IDEs como VS Code.
4. Suporte exclusivo para redes de teste como Ropsten.

2

Configuração de Projeto

Para configurar um novo projeto Foundry com a estrutura de diretórios padrão (src, test, script, lib), qual comando deve ser utilizado?

1. foundry install
2. forge build
3. forge init
4. anvil start

3

Manipulação de Tempo

Você precisa testar uma função de contrato que só pode ser chamada após um período de 7 dias. Qual cheatcode do Foundry seria mais adequado para simular o avanço do tempo em seu teste?

1. vm.deal()
2. vm.prank()
3. vm.roll()
4. vm.warp()

4

Teste de Permissões

Um contrato inteligente possui uma função onlyOwner que restringe o acesso ao proprietário. Para testar se um endereço não-proprietário não consegue chamar essa função, qual combinação de cheatcodes seria mais eficaz?

1. vm.warp() e vm.deal()
2. vm.prank() e testFail_
3. vm.etch() e vm.store()
4. vm.roll() e vm.load()

5

Questão Dissertativa

Explique como os cheatcodes do Foundry contribuem para a robustez e segurança de contratos inteligentes que interagem com as tendências atuais, como Abstração de Contas (ERC-4337) ou soluções de Escalabilidade (Layer 2).

Gabarito

1. b)

2. c)

3. d)

4. b)

Próxima Aula

Na **Aula 10 – Frameworks de Teste Avançado: Foundry (Parte 2)**, aprofundaremos ainda mais no Foundry, explorando técnicas avançadas como Fuzzing, Invariant Testing e outros cheatcodes poderosos para garantir a máxima segurança e resiliência dos seus contratos.

Recursos Adicionais

Foundry Book

Documentação oficial e abrangente do Foundry.

Paradigm's Foundry Examples

Repositório com exemplos práticos de uso do Foundry.

Awesome Foundry

Uma lista curada de recursos, ferramentas e projetos relacionados ao Foundry.

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