

DECENTRALIZED TECH



SOLIDITY PARA INICIANTES

DO ZERO AO PRIMEIRO CONTRATO INTELIGENTE

Um guia passo a passo para aqueles que estão embarcando em sua jornada para aprender sobre a linguagem de programação Solidity

Fabio Santos

DECENTRALIZED TECH

Qualquer código-fonte ou outro material complementar referenciado pelo autor neste livro está disponível para os leitores no GitHub, por meio da página do produto do livro localizada em <https://github.com/DecentralizedTech/solidityparainiciantes.git>

Copyright © Decentralized Tech, 2023

Autor: Fabio Santos

Este trabalho está sujeito a direitos autorais. Todos os direitos são reservados pelo Editor, seja a obra como um todo ou parte dela, especificamente os direitos de tradução, reimpressão, reutilização de ilustrações, recitação, transmissão, reprodução em microfilmes ou de qualquer outra forma física, assim como armazenamento e recuperação de informações, adaptação eletrônica, software de computador ou por metodologia similar ou dissimilar, conhecida atualmente ou desenvolvida no futuro.

Nomes comerciais, logotipos e imagens registrados podem aparecer neste livro. Ao invés de utilizar um símbolo de marca registrada em cada ocorrência de um nome, logotipo ou imagem registrada, nós utilizamos os nomes, logotipos e imagens apenas de forma editorial e em benefício do proprietário da marca, sem a intenção de infringir a marca registrada. O uso nesta publicação de nomes comerciais, marcas registradas, marcas de serviço e termos similares, mesmo que não sejam identificados como tais, não deve ser interpretado como uma opinião sobre se estão sujeitos a direitos de propriedade.

Embora os conselhos e informações neste livro sejam considerados verdadeiros e precisos na data de publicação, nem os autores, nem os editores, nem o editor podem aceitar qualquer responsabilidade legal por erros ou omissões que possam ter sido cometidos. O editor não oferece garantia, expressa ou implícita, com relação ao material contido neste documento.

Agradecimentos:

A jornada de criação deste livro "Solidity para Iniciantes: Do Zero ao Primeiro Contrato Inteligente" foi nada menos que uma viagem transformadora e, como tal, não teria sido possível sem o apoio e encorajamento de várias pessoas incríveis que estiveram ao meu lado.

Em primeiro lugar, gostaria de agradecer a Deus e expressar minha mais profunda gratidão a todos os colaboradores e revisores técnicos que trabalharam incansavelmente para ajudar a moldar e aprimorar o conteúdo deste livro. Sua experiência, paixão e comprometimento foram instrumentais para transformar a visão deste livro em realidade.

Agradeço também a minha família e amigos que me apoiaram incondicionalmente durante todo este processo. Sua paciência, amor e encorajamento constante foram uma fonte inestimável de força e inspiração.

Por último, mas não menos importante, gostaria de agradecer a você, o leitor. Obrigado por escolher este livro como sua guia na exploração de Solidity e contratos inteligentes. Espero sinceramente que você encontre o valor que procurava e que esta seja apenas a primeira de muitas etapas em sua jornada de aprendizado no universo blockchain.

A todos vocês, meu mais sincero obrigado.

Fabio Santos

Prefácio:

Olá e seja bem-vindo ao “Solidity para Iniciantes: Do Zero ao Primeiro Contrato Inteligente”! Este livro foi criado com o objetivo de servir como um guia passo a passo para aqueles que estão embarcando em sua jornada para aprender sobre a linguagem de programação Solidity e o fascinante mundo dos contratos inteligentes.

Quando o blockchain Ethereum foi lançado em 2015, ele trouxe uma inovação revolucionária para o campo da tecnologia blockchain - a capacidade de programar e executar contratos inteligentes de maneira descentralizada. Solidity, a linguagem de programação criada especificamente para a escrita de contratos inteligentes na Ethereum, tornou-se rapidamente uma ferramenta indispensável para desenvolvedores e entusiastas de blockchain em todo o mundo.

Aprender Solidity é como abrir uma porta para um novo universo cheio de possibilidades inexploradas. Com este livro, pretendemos oferecer a você as chaves para essa porta. Não importa se você é um programador experiente em outra linguagem de programação ou um completo iniciante no mundo da codificação, este livro foi projetado para levar você desde os conceitos básicos até o ponto onde pode escrever, testar e implantar seu primeiro contrato inteligente.

O livro é estruturado de forma a proporcionar uma progressão lógica e intuitiva através dos vários tópicos necessários para dominar Solidity. Começamos com uma introdução aos conceitos básicos de programação e blockchain antes de mergulhar nos detalhes de Solidity. A partir daí, exploramos as características únicas dos contratos inteligentes, passando por temas como manipulação de Ether, testes e publicação de seus contratos.

Fizemos o nosso melhor para apresentar os conceitos de forma clara e acessível, complementando as explicações teóricas com exemplos práticos de código e exercícios de aprendizagem ativa. Além disso, compartilhamos dicas, truques e melhores práticas que adquirimos com nossa experiência no campo.

No final deste livro, você terá uma compreensão sólida das bases de Solidity e estará equipado com o conhecimento e a confiança para começar a explorar e contribuir para o excitante mundo dos contratos inteligentes. Espero que você aproveite esta jornada tanto quanto nós apreciamos prepará-la para você. Boa leitura e feliz codificação!

Sumário

Capítulo 1: Introdução.....	3
1.1 O que é Ethereum?.....	3
1.1.1 Redes da Ethereum.....	4
1.2 O que é EVM?.....	5
1.3 O que é Blockchain?.....	5
1.4 O que são Aplicações Descentralizadas (dApps)?.....	6
1.5 O que é Solidity?.....	7
1.6 O que é Truffle?.....	8
1.7 O que é Remix IDE?.....	9
Capítulo 2: Preparando-se para Programar.....	10
2.1 Configurando o ambiente de desenvolvimento com Truffle.....	10
2.2 Configurando o ambiente de desenvolvimento com Remix IDE.....	11
2.3 Configurando uma rede de teste.....	12
2.3.1 Ganache.....	13
2.3.1.1 Como o Ganache é instalado?.....	13
2.4 Metamask.....	23
2.4.1 Como instalar o Metamask.....	25
2.4.2 Metamask e Networks.....	28
2.4.3 Como usar o Metamask no Remix IDE?.....	29
Capítulo 3: Conceitos Fundamentais da Programação Solidity.....	32
3.1 Tipos de Dados em Solidity.....	32
3.2 Variáveis e Funções.....	32
3.2.1 Públicas.....	33
3.2.2 Privadas.....	33
3.2.3 Internas.....	34
3.2.4 Externas.....	34
3.3 Variáveis Globais Solidity.....	35
3.4 Estruturas, Enumerações e Arrays.....	36
3.4.1 Structs.....	37
3.4.2 Enums.....	37
3.4.3 Mapping.....	38
3.4.4 Arrays.....	39
3.5 Função e Modificadores de Função.....	41
3.5.1 Função Construtor.....	44
3.6 Laços e Estruturas de Decisão.....	44
3.7 Eventos.....	48

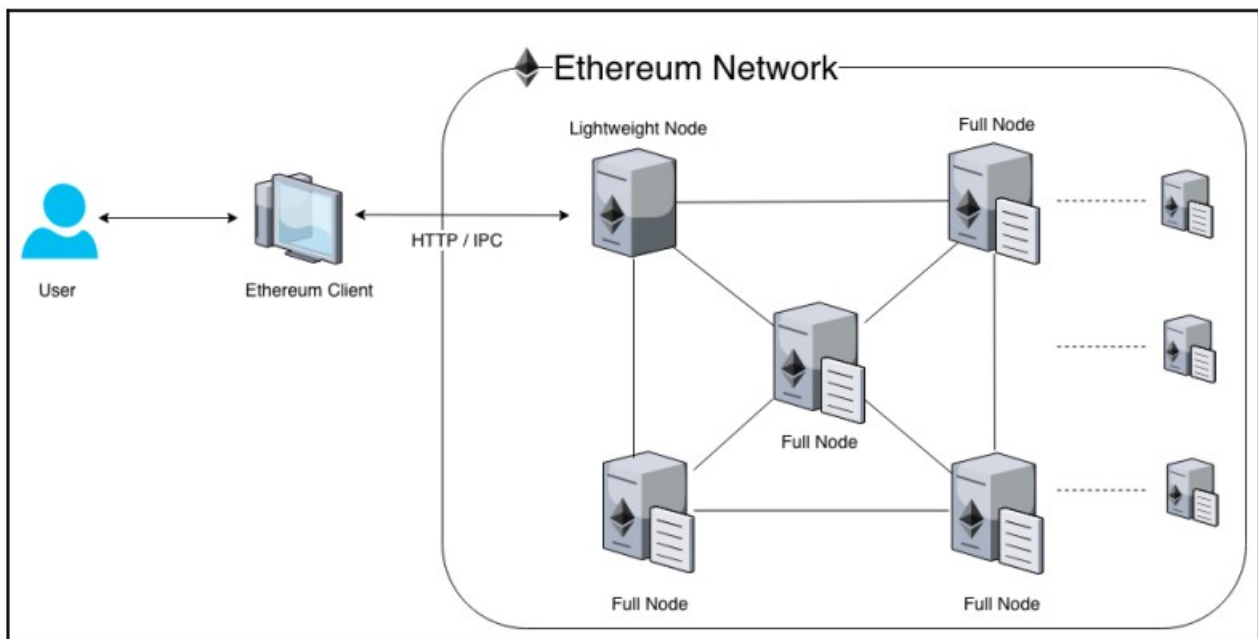
Capítulo 4: Contratos Inteligentes.....	49
4.1 O que são Contratos Inteligentes?.....	49
4.2 Propriedades dos Contratos Inteligentes.....	50
4.3 Como Contratos Inteligentes Interagem.....	51
4.4 Escrevendo um Contrato Inteligente Interativo.....	51
4.5 Compilando e Executando Contratos Interativos com Truffle e Remix IDE.....	52
Capítulo 5: Aprofundamento em Contratos Inteligentes.....	53
5.1 Herança em Solidity.....	53
5.2 Interfaces em Solidity.....	54
5.3 Bibliotecas em Solidity.....	55
5.4 Require.....	56
5.5 Exemplo de um Contrato Inteligente para Marketplace.....	57
Capítulo 6: Trabalhando com Ether em Solidity.....	60
6.1 Unidades de Ether.....	60
6.2 Transferindo Ether.....	60
6.3 Verificando Balanços.....	60
6.4 FallBack e Receive Functions.....	61
Capítulo 7: Testando seus Contratos Inteligentes.....	62
7.1 Por que Testar?.....	62
7.2 Ferramentas de Teste.....	62
7.3 Escrevendo Testes.....	62
7.4 Executando o Teste no Truffle.....	63
7.5 Testando no Remix IDE.....	64
7.6 Executando o Teste no Remix IDE.....	64
Capítulo 8: Publicando seus Contratos Inteligentes.....	66
8.1 Publicando com o Truffle.....	66
8.2 Publicando com o Remix IDE e Metamask.....	68
Capítulo 9: Próximos Passos em Solidity.....	72
9.1 Estudando Padrões de Contrato Inteligente.....	72
9.2 Aprendendo sobre Segurança de Contratos Inteligentes.....	72
9.3 Explorando Contratos DeFi e NFT.....	72
9.4 Contribuindo para Projetos de Código Aberto.....	72
9.5 Participando da Comunidade Ethereum.....	73
Capítulo 10: Conclusão.....	74

Capítulo 1: Introdução

Bem-vindo à jornada fascinante de programação Solidity! Este livro é projetado para ajudar programadores iniciantes a mergulhar no mundo da blockchain Ethereum e a dominar a linguagem de programação Solidity, que é usada para criar contratos inteligentes nesta plataforma.

1.1 O que é Ethereum?

Ethereum é uma plataforma blockchain de código aberto que permite que desenvolvedores criem e implementem contratos inteligentes descentralizados, no capítulo 4 você aprenderá a desenvolver contratos inteligente por meio da linguagem Solidity. Lançada em 2015, ela tem sido um catalisador significativo para a inovação na esfera blockchain, permitindo uma infinidade de aplicações descentralizadas (dApps).



A Ethereum Network é composta por uma rede de nós que mantêm a cópia completa da blockchain e executam a Máquina Virtual Ethereum (EVM). Esses nós trabalham em conjunto para validar transações, alcançar consenso e manter a integridade da rede. Existem diferentes tipos de nós, incluindo nós completos, que mantêm a cópia completa da blockchain, e nós leves, que sincronizam apenas informações específicas.

A Ethereum Network atualmente está em processo de transição para um mecanismo de consenso chamado Proof of Stake (PoS). No PoS, os validadores são escolhidos com base na quantidade de ether que eles "travam" como garantia. Isso substitui o mecanismo de consenso anterior, chamado Proof of Work (PoW), onde os mineradores resolvem problemas computacionais complexos para validar as transações

Para executar transações e contratos inteligentes na Ethereum, é necessário pagar uma taxa chamada "gas". O gas é uma medida de custo computacional que ajuda a evitar abusos da rede. Cada operação na Ethereum consome uma quantidade específica de gas, e a taxa de gas é determinada pelo mercado.

1.1.1 Redes da Ethereum

A Ethereum não é composta por uma única rede, mas por várias redes ou ambientes, cada um com suas próprias características e casos de uso específicos. Vamos discutir algumas das redes mais comumente usadas na Ethereum:

1. **Ethereum Mainnet:** Esta é a rede principal da Ethereum onde todos os contratos inteligentes reais são implantados e onde o Ether tem valor real. Quando as pessoas falam sobre o preço do Ether, elas estão se referindo ao valor do Ether nesta rede.

2. **Testnets:** Estas são redes de teste que simulam a rede Ethereum principal. Elas são usadas pelos desenvolvedores para testar e experimentar seus contratos inteligentes antes de implantá-los na mainnet. O Ether nessas redes é gratuito e pode ser obtido através de faucets. A testnet mais comum é a Goerli.

Goerli: Goerli é uma rede de teste multi-cliente, o que significa que é compatível com vários clientes Ethereum (como Geth e OpenEthereum). Isso a torna útil para testar a compatibilidade do cliente.

Escolher a rede certa depende do estágio do seu desenvolvimento e do que você está tentando alcançar. Para desenvolvimento inicial e testes, uma blockchain local ou uma das testnets é

uma escolha apropriada. Quando estiver pronto para lançar, a Mainnet é o lugar onde seu contrato inteligente deve acabar.

1.2 O que é EVM?

A Ethereum Virtual Machine (EVM) é um componente central do ecossistema Ethereum e desempenha um papel fundamental na execução de contratos inteligentes. Ela é responsável por executar todo o código Solidity do contrato inteligente na rede Ethereum de forma segura e isolada.

A EVM é uma máquina virtual de pilha, o que significa que opera com base em uma estrutura de dados de pilha, onde as operações ocorrem no topo da pilha. Quando um contrato inteligente é executado, ele é convertido em código de byte que a EVM pode ler e processar.

A EVM é isolada do sistema de rede, sistema de arquivos e outros processos da máquina host. Esta característica de sandbox (ou caixa de areia) a torna uma camada segura para a execução de contratos inteligentes, pois mesmo que um contrato inteligente seja malicioso ou contenha um erro, ele não pode comprometer a própria máquina host ou a rede Ethereum maior.

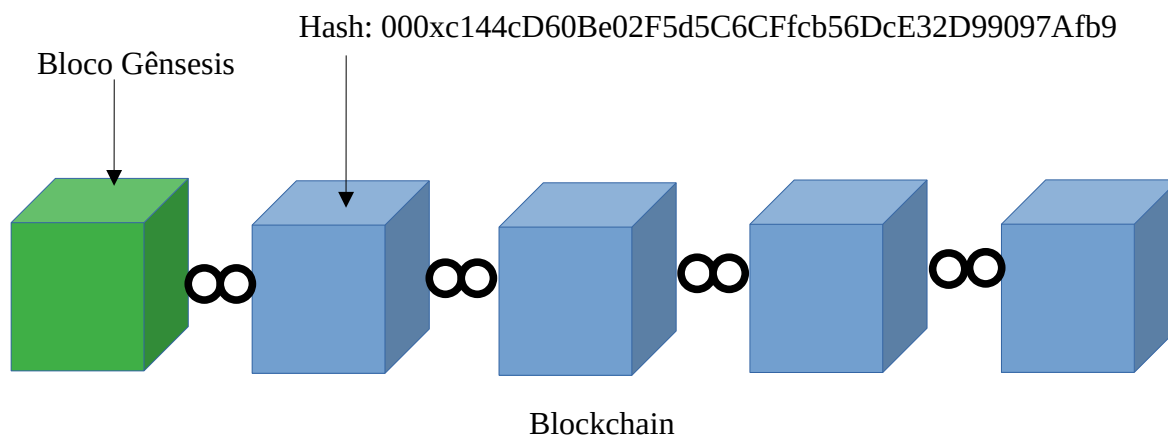
Além disso, a EVM é projetada para ser determinística. Isso significa que, dadas as mesmas entradas, a execução de um contrato inteligente sempre produzirá o mesmo resultado. Isso é fundamental para a natureza descentralizada da Ethereum, pois permite que todos os nós da rede concordem com o resultado da execução de um contrato inteligente sem necessidade de confiança.

Em suma, a Ethereum Virtual Machine (EVM) é uma parte crucial do ecossistema Ethereum, que permite a execução segura, isolada e determinística de contratos inteligentes na rede Ethereum.

1.3 O que é Blockchain?

A blockchain é uma tecnologia que permite o armazenamento de dados digitais com uma sofisticada codificação muito segura. O nome vem do inglês, onde "block" significa bloco e "chain", cadeia. Ou seja, é uma cadeia de blocos. Cada bloco contém dados e um código exclusivo chamado hash, que representa a natureza imutável e distribuída da tecnologia blockchain.

Uma vez que um bloco é completo, um novo bloco é gerado. Os blocos estão ligados uns aos outros de forma que cada bloco contém um hash criptográfico do bloco anterior, gerando uma cadeia de blocos.



Índice	0
Timestamp	2023-07-27... 391
Dados	"Bloco Gênesis"
Hash anterior	None
Nonce	0
Hash do bloco	41b103d48...1c48

Bloco Gênesis

A tecnologia blockchain e os contratos inteligentes estão intimamente ligados, pois os contratos inteligentes são um dos principais recursos que diferenciam as plataformas blockchain de segunda geração, como a Ethereum, de seu predecessor, o Bitcoin. A relação entre blockchain e contratos inteligentes é essencial para muitos dos usos mais avançados da tecnologia blockchain. A blockchain fornece o ambiente imutável, transparente e distribuído no qual os contratos inteligentes operam, enquanto os contratos inteligentes dão "vida" à blockchain, permitindo transações e operações automáticas e complexas que vão além do simples envio e recebimento de criptomoedas.

No contexto da Ethereum, por exemplo, os contratos inteligentes permitem a criação de tokens ERC20, ERC721 (NFT), a execução de aplicações descentralizadas (dApps), a criação de organizações autônomas descentralizadas (DAOs), entre muitos outros usos. Esses recursos expandem significativamente as possibilidades do que pode ser construído em uma plataforma blockchain.

1.4 O que são Aplicações Descentralizadas (dApps)?

As aplicações descentralizadas, ou dApps, são aplicações que funcionam em uma rede de computadores descentralizada. Ao contrário das aplicações tradicionais, que são executadas em servidores centralizados e são controladas por uma única entidade, as dApps operam em uma rede peer-to-peer e não são controladas por uma única entidade.

As dApps podem ser construídas sobre várias plataformas blockchain, mas a mais comum é a Ethereum, devido à sua linguagem de programação Solidity e ao seu ambiente de execução, a Ethereum Virtual Machine (EVM). Isso permitiu o desenvolvimento de uma ampla gama de dApps, desde jogos até plataformas financeiras descentralizadas (DeFi).

As dApps têm várias características distintivas:

1. **Descentralização:** As dApps são executadas em uma rede blockchain, o que significa que não há um único ponto de falha e que as dApps são resistentes à censura e ao tempo de inatividade.
2. **Código aberto:** A maioria das dApps tem seu código aberto, o que significa que qualquer pessoa pode verificar a lógica da aplicação. Isso ajuda a criar confiança, pois os usuários podem verificar se a dApp faz o que afirma fazer.
3. **Contratos Inteligentes:** As dApps utilizam contratos inteligentes para gerenciar a lógica da aplicação e armazenar dados. Isso garante que a lógica da aplicação seja executada como programado e que os dados não possam ser alterados após serem armazenados na blockchain.
4. **Tokens:** Muitas dApps têm seus próprios tokens, que são usados para incentivar os usuários e manter a segurança e a governança da aplicação.

As dApps têm o potencial de remodelar vários setores, desde finanças até mídia social, jogos e além. No entanto, elas ainda são um campo emergente com muitos desafios a serem superados, como a escalabilidade e a usabilidade. Não obstante, a promessa de um mundo descentralizado e sem confiança torna as dApps uma área emocionante para explorar e inovar.

1.5 O que é Solidity?

Solidity é uma linguagem de programação orientada a objetos de alto nível para escrever contratos inteligentes. Foi proposta em 2014 por Gavin Wood, co-fundador da Ethereum, e desde então tem sido a linguagem padrão para a criação de contratos inteligentes na Ethereum.

A linguagem foi projetada para permitir aos desenvolvedores escrever aplicações que implementam autoexecução de contratos de negócios, que podem ser usados em várias áreas, como sistemas de votação, financiamento coletivo, leilões automáticos, sistemas financeiros multiassinatura, e muitos outros.

A sintaxe de Solidity é semelhante à de JavaScript, o que a torna uma linguagem acessível para desenvolvedores com experiência em desenvolvimento web. No entanto, ao contrário do JavaScript, Solidity é fortemente tipada e suporta uma gama de recursos de programação orientada a objetos, incluindo herança, bibliotecas e definição de tipos complexos.

Uma característica distintiva de Solidity é sua integração direta com a Ethereum Virtual Machine (EVM). Cada operação em Solidity corresponde diretamente a uma operação na EVM, permitindo aos desenvolvedores manipular diretamente a memória, o armazenamento e a pilha de execução da EVM. Além disso, Solidity suporta uma gama de funções nativas para realizar operações como verificação de assinatura e interações com outros contratos.

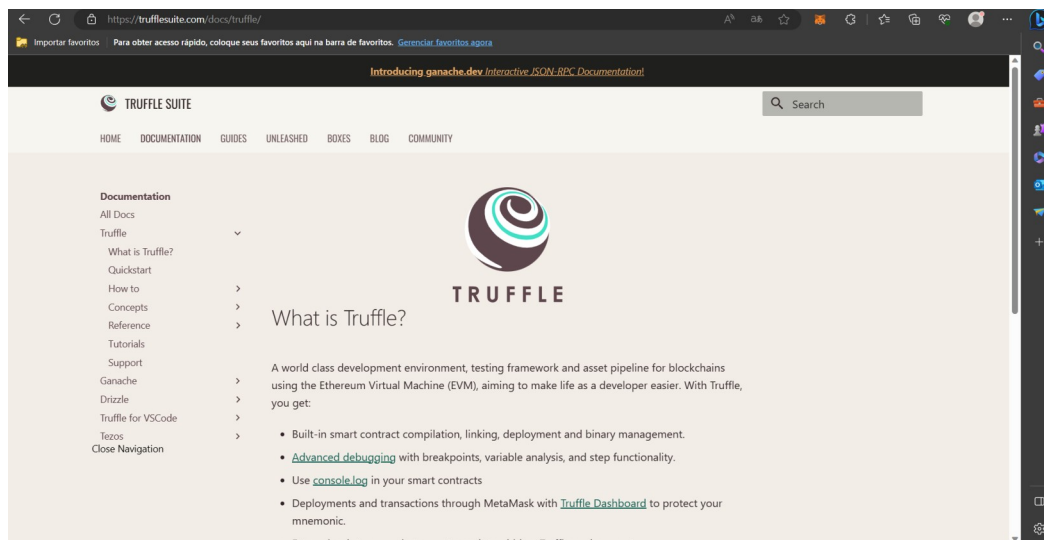
O ambiente de desenvolvimento para Solidity também amadureceu ao longo dos anos. Ferramentas como Truffle e Hardhat proporcionam um conjunto completo de recursos para desenvolvimento, teste e implantação de contratos inteligentes. O Remix IDE fornece um ambiente de desenvolvimento no navegador que é fácil de usar e acessível para novos desenvolvedores.

Apesar de sua relativa juventude como linguagem de programação, Solidity já se estabeleceu como a linguagem de escolha para a criação de contratos inteligentes e aplicações descentralizadas na Ethereum e em outras blockchains compatíveis com EVM.

1.6 O que é Truffle?

Truffle é um framework de desenvolvimento popular para Ethereum, oferecendo um conjunto de ferramentas que facilitam a escrita, compilação, teste e a implantação de contratos inteligentes. Ela

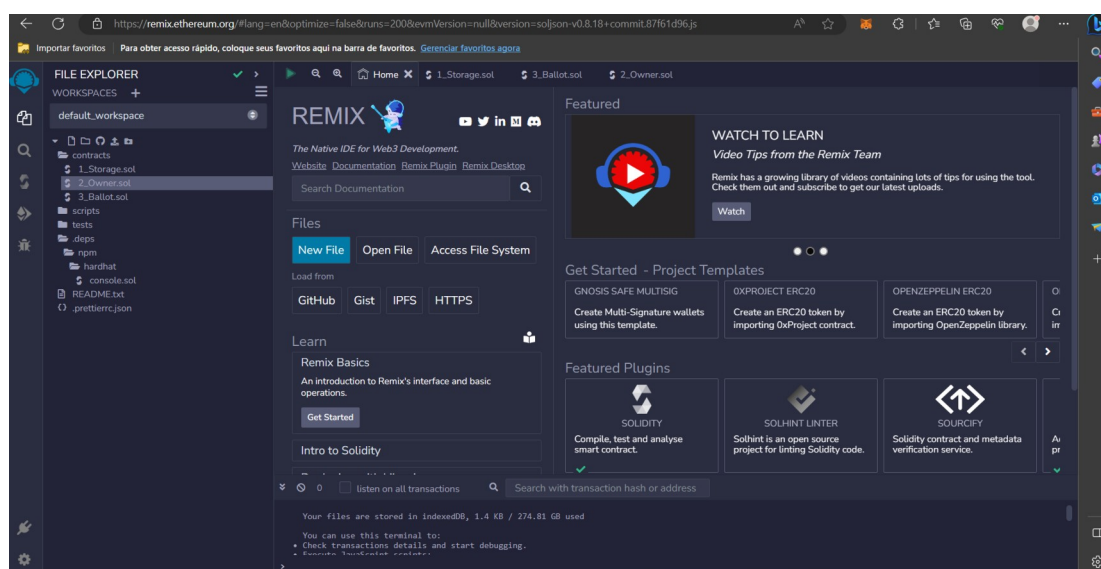
integra-se bem com o desenvolvimento moderno de front-end e pode ser usada em conjunto com outras ferramentas e bibliotecas de blockchain.



1.7 O que é Remix IDE?

Remix IDE é um ambiente de desenvolvimento baseado em navegador para Solidity que permite que você escreva, teste, depure e publique contratos inteligentes. É uma ferramenta poderosa e flexível, especialmente adequada para o desenvolvimento rápido e a experimentação.

Neste livro, exploraremos o desenvolvimento de contratos inteligentes em Solidity usando tanto o Truffle quanto o Remix IDE. Independentemente de suas preferências ou necessidades de desenvolvimento, você ganhará uma compreensão sólida de como criar, testar e implantar contratos inteligentes na blockchain Ethereum. Vamos começar!



Capítulo 2: Preparando-se para Programar

Antes de começarmos a escrever contratos inteligentes em Solidity, precisamos preparar nosso ambiente de desenvolvimento. Neste capítulo, mostraremos como configurar tanto o Truffle quanto o Remix IDE e como usar o Metamask.

2.1 Configurando o ambiente de desenvolvimento com Truffle

Para configurar o ambiente de desenvolvimento com Truffle, você precisará seguir as etapas abaixo:

1. **Instale Node.js e npm:** Node.js é um ambiente de execução JavaScript que você precisará para executar o Truffle. O npm é o gerenciador de pacotes Node.js, que você usará para instalar o Truffle. Você pode baixar Node.js e npm [aqui](https://nodejs.org/).
2. **Instale o Truffle:** Após instalar o Node.js e npm, você pode instalar o Truffle com o seguinte comando no terminal: ``npm install -g truffle``.
3. **Verifique a instalação:** Para verificar se o Truffle foi instalado corretamente, você pode usar o comando ``truffle version``. Se o Truffle estiver instalado corretamente, este comando exibirá a versão do Truffle.
4. **Crie um projeto Truffle:** Para criar um projeto Truffle, navegue até o diretório onde deseja criar o projeto e execute o comando ``truffle init``. O resultado da execução desse comando será a seguinte estrutura de diretórios e arquivos:

```
.
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
├── test
├── truffle-config.js
└── package.json
```

Aqui está o que cada diretório ou arquivo faz:

- ``contracts/``: Este diretório é onde você coloca todos os seus contratos inteligentes. O contrato ``Migrations.sol`` é um contrato especial que o Truffle usa para gerenciar e executar migrações.
- ``migrations/``: Este diretório é onde você coloca scripts de migração. Esses scripts são responsáveis por implantar seus contratos na rede Ethereum.
- ``test/``: Este diretório é onde você coloca testes para seus contratos. O Truffle suporta testes escritos tanto em JavaScript quanto em Solidity.
- ``truffle-config.js``: Este é o arquivo de configuração para o seu projeto Truffle. É aqui que você define coisas como a rede Ethereum para a qual você quer implantar, bem como outras opções de configuração.
- ``package.json``: Este é o arquivo de manifesto do seu projeto que contém metadados do projeto e dependências. Ele é usado pelo npm para gerenciar as dependências do seu projeto.

O ``truffle init`` é uma maneira rápida e fácil de começar um novo projeto Truffle, já que ele configura uma estrutura de projeto básica e alguns arquivos de exemplo para você.

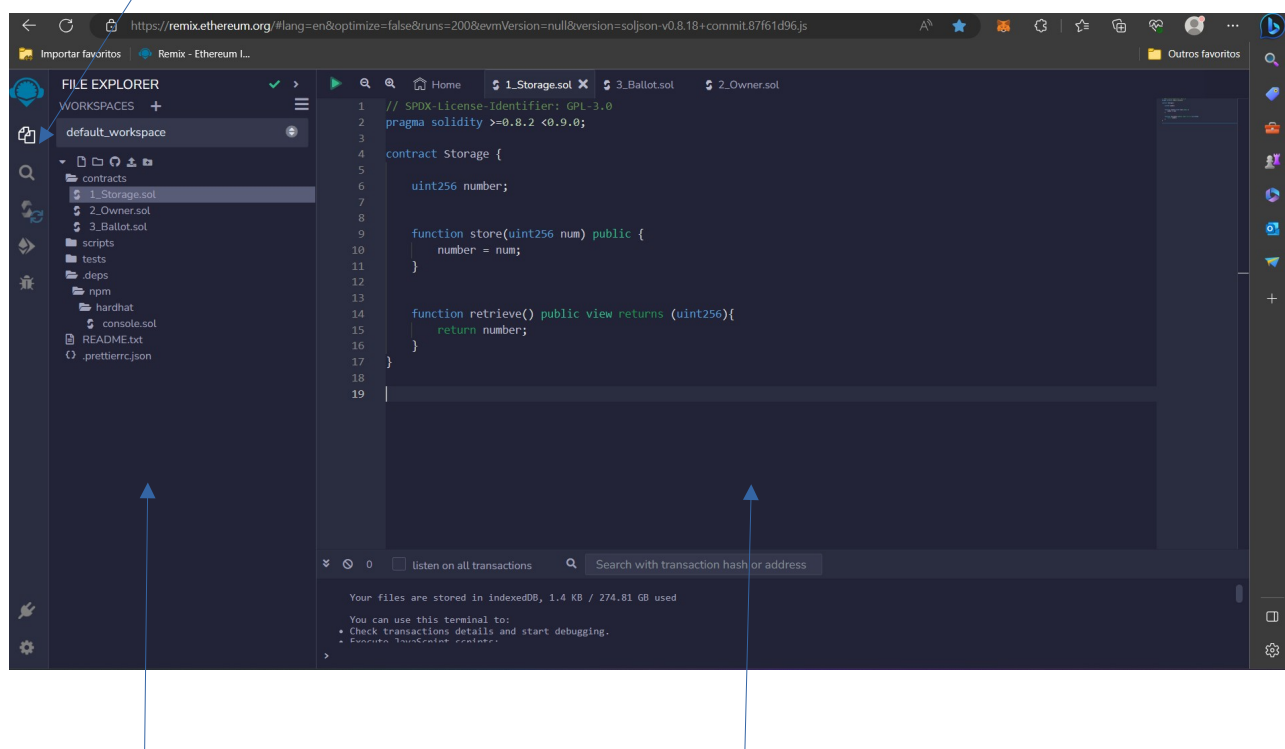
2.2 Configurando o ambiente de desenvolvimento com Remix IDE

Configurar o Remix IDE é muito mais simples, pois é uma aplicação web e não requer instalação. Para começar a usar o Remix IDE, siga estas etapas:

1. **Acesse o Remix IDE:** Abra um navegador web e acesse [<https://remix.ethereum.org>].
2. **Conheça o ambiente:** O Remix IDE tem três seções principais. À esquerda, você encontrará o explorador de arquivos. No meio, terá o editor de código. E na barra de ferramentas à esquerda do explorador de arquivos, encontrará o compilador, o implantador e o ambiente de execução.

3. Crie um novo arquivo Solidity: Para criar um novo arquivo Solidity, clique no ícone de mais (+) no explorador de arquivos e dê um nome ao seu arquivo com a extensão `.sol`.

Barra de ferramentas



Explorador de arquivos

Editor de código

Agora que temos nosso ambiente de desenvolvimento configurado, podemos começar a escrever nossos primeiros contratos inteligentes em Solidity! No próximo capítulo, discutiremos os conceitos fundamentais da programação Solidity.

2.3 Configurando uma rede de teste

A forma mais simples para testar um contrato inteligente é por meio do deploy para uma Máquina Virtual (VM) do Remix IDE que possui algumas VM (Londo, Berlin, Shanghai, Merge) que você pode usar para executar e testar o seu contrato. Uma outra forma de testar o seu contrato é por meio

do uso de uma rede de teste. Para criar uma rede de teste você poderá usar a Ferramenta Ganache ou a BuildBear.

2.3.1 Ganache

Ganache é uma parte integral da Truffle Suite e é uma blockchain Ethereum de teste pessoal e rápida que permite o desenvolvimento de contratos inteligentes. O Ganache cria uma blockchain local em sua máquina que pode ser usada para testar a implantação de contratos inteligentes, executar comandos e inspecionar o estado enquanto controla a operação da cadeia.

2.3.1.1 Como o Ganache é instalado?

A instalação do Ganache é um processo direto e pode ser realizada de duas maneiras principais: através da versão em linha de comando (Ganache CLI) ou através da versão de interface gráfica de usuário (Ganache GUI).

Instalando o Ganache CLI

O Ganache CLI pode ser instalado globalmente em seu sistema usando o npm, que é o gerenciador de pacotes padrão para Node.js. Se você já tem Node.js e npm instalados, você pode instalar o Ganache CLI executando o seguinte comando no seu terminal:

```
npm install -g ganache-cli
```

Uma vez instalado, você pode iniciar o Ganache CLI com o comando ``ganache-cli``. Isso iniciará uma instância da blockchain Ethereum em sua máquina local.

```

root@ip-172-31-45-180:~# ganache-cli -h 0.0.0.0 -p 8545
Ganache CLI v6.12.2 (ganache-core: 2.13.2)

Available Accounts
=====
(0) 0x59004CaDBC47Cfc4F4708980A0F6F6101f451cC0 (100 ETH)
(1) 0xA5d41c03af8b26f80c96b3cfCD732e8523F1ca80 (100 ETH)
(2) 0x942b91BAe45BD89171746586CD94e8A29423FE13 (100 ETH)
(3) 0x0E987C5D9E062228A3f9F5cdE18867bf7A7E5888 (100 ETH)
(4) 0x5Eb0CF3C2EA748A7ACfE0a39e64B168881e8E1C (100 ETH)
(5) 0x612b0aE379B54dd3155bD986C4d062579bb1637D (100 ETH)
(6) 0xE6CAEBE063a1241473F5A9b54e4A3C091686a126 (100 ETH)
(7) 0xdF07f1867f0a05d2f6005cFE6b4023D5840c2d5c (100 ETH)
(8) 0xFD18CA94e6D2EA92b7b38eaB47Dc04e1Bdb85fd (100 ETH)
(9) 0xD4F2f6cE6e1cacfb4C8D4C19dFa2de7240063a6F (100 ETH)

Private Keys
=====
(0) 0x648a4e5cf63875da0db0bc32b0967c98bab3c0fc5ee15dd15b715666285d779
(1) 0x8c73247c66f356ae612f2c042e0cb8396a97b7676a387c1a0de07c3769ac9605
(2) 0xacfd635b010962280e02e877b7034425df7921638123a0792e5a528b10f47a7c
(3) 0xac06d189fa775f9b6cada0ef67dd17d2a5c606b006ff0ab687dd66a4dd5d0b4d
(4) 0xdd7a26a2b8195d6a8babd697f194d1da71e1cef269ffeeef74367f600bbc06eal
(5) 0x260c3d67f500f418a3409984cc2b443fd0ff8ea71a8978a92c8f5312d72e4e16
(6) 0x3f7be7ba672332de19d9575c085b5101944402ce36da94ee6f1f97377df53e3
(7) 0x0b34904a066a5e2662bd362a04e90b61122498e08297f685780192d0de74d690
(8) 0x9a4b0fb8fbcc1106c436385e328e534c9e323ccdb16e2022bd4eca07d5a2fd0
(9) 0xe2f8f8b1682a1e3117931da6a317c48edc6b7367590aa0bdc498e2f4934a38cf

HD Wallet
=====
Mnemonic:      sweet mother weather cool agent possible route salon upon human trend firm
Base HD Path:  m/44'/60'/0'/0/{account_index}

Gas Price
=====
20000000000

Gas Limit
=====
6721975

Call Gas Limit
=====
9007199254740991

Listening on 0.0.0.0:8545

```

Instalando o Ganache GUI

Se você preferir uma interface gráfica de usuário, o Ganache também oferece uma versão GUI que pode ser baixada diretamente do site oficial da Truffle Suite.

Ganache						
ACCOUNTS BLOCKS TRANSACTIONS CONTRACTS EVENTS LOGS						
CURRENT BLOCK 0 GAS PRICE 20000000000 GAS LIMIT 6721975 HARDFORK MUIRGLACIER NETWORK ID 5777 RPC SERVER HTTP://127.0.0.1:7545 MINING STATUS AUTOMINING WORKSPACE QUICKSTART SAVE SWITCH						
MNEMONIC ?		HD PATH				
glide dog club agent urban maid oxygen illness spot engine bid poet		m/44'/60'/0'/0/account_index				
ADDRESS	BALANCE	TX COUNT	INDEX			
0xa4a1441fF9759847751f669DEF5a4222Fee84e2a	100.00 ETH	0	0			
ADDRESS	BALANCE	TX COUNT	INDEX			
0x94208d5BE7f940F24654b71c8E4139a297e75Fc7	100.00 ETH	0	1			
ADDRESS	BALANCE	TX COUNT	INDEX			
0x0FF86bC284eF4422579287bf683A473D05f9428E	100.00 ETH	0	2			
ADDRESS	BALANCE	TX COUNT	INDEX			
0xe9A8d0dc08a0A12A62B2e2a4B39494092eDc13d9	100.00 ETH	0	3			
ADDRESS	BALANCE	TX COUNT	INDEX			
0xad77eAff1Ea514c4238FD37f2140EF32B3eb64eC	100.00 ETH	0	4			
ADDRESS	BALANCE	TX COUNT	INDEX			
0x7f87ce4B66991030358fb2a53F0bcFEc115e820c	100.00 ETH	0	5			
ADDRESS	BALANCE	TX COUNT	INDEX			
0xAe454290a2f9EE191cac796497A5E5Dcf91baDC8	100.00 ETH	0	6			

Aqui estão os passos para instalar o Ganache GUI:

1. Vá para o site da Truffle Suite em <https://www.trufflesuite.com/ganache>.
2. Clique no botão "Download" na página principal. Isso o levará a uma página onde você pode escolher a versão do Ganache que corresponde ao seu sistema operacional (Windows, Mac ou Linux).
3. Baixe o arquivo e siga as instruções de instalação para o seu sistema operacional específico.
4. Depois de instalado, você pode abrir o Ganache e ele automaticamente iniciará uma blockchain local em sua máquina.

Ambas as versões do Ganache (CLI e GUI) fornecem os mesmos recursos básicos, mas a versão GUI também oferece uma interface visual intuitiva que facilita a visualização e a interação com a sua blockchain local. A escolha entre CLI e GUI depende principalmente da sua preferência pessoal.

Configurando o Ganache com a Truffle Suite

Primeiro, você inicia o Ganache e configura a blockchain local. Em seguida, você pode compilar e migrar seus contratos inteligentes para a blockchain local usando o Truffle. Uma vez que seus contratos estejam implantados, você pode interagir com eles usando o console Truffle ou um cliente Dapp. Siga os passos abaixo:

Passo 1: Inicie o Ganache

Para iniciar o Ganache, você pode usar o seguinte comando:

```
ganache-cli
```

Isso iniciará o Ganache em <http://127.0.0.1:8545>. O Ganache fornecerá 10 contas Ethereum de teste, cada uma com 100 Ether, para você usar durante o desenvolvimento.

Passo 2: Configure o Truffle

Você precisa configurar o Truffle para se conectar ao Ganache. Para fazer isso, vá para o diretório do seu projeto e crie ou edite o arquivo `truffle-config.js` para que ele se pareça com isso:

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 8545, // Porta padrão do Ganache CLI,
                  // para Ganache CLI pode ser 7545
      network_id: "*",
    },
  },
  compilers: {
    solc: {
      version: "0.8.2", // Versão do compilador Solidity
    }
  }
};
```

Essa configuração indica que o Truffle deve se conectar ao nó Ethereum executando no localhost na porta 7545 (ou 8545, dependendo da sua configuração do Ganache), e que ele deve usar a versão 0.8.2 do compilador Solidity.

Passo 3: Escrever um script de migração

Os scripts de migração são como o Truffle sabe como implantar seus contratos. Crie um novo arquivo no diretório `migrations/` chamado `1_deploy_contracts.js` e adicione o seguinte código:

```
const Storage = artifacts.require("Storage");

module.exports = function (deployer) {
  deployer.deploy(Storage);
};
```

Este é um script de migração usado na Truffle Suite, uma popular ferramenta de desenvolvimento para a Ethereum. As migrações são scripts de JavaScript que ajudam você a implantar contratos na Ethereum.

Vamos decompor o código:

```
1.`const Storage = artifacts.require("Storage");`
```

"artifacts" são uma abstração do Truffle para contratos inteligentes. Eles incluem várias informações úteis sobre o contrato, como o ABI (Application Binary Interface) e o endereço do contrato implantado. O método ``require`` é usado para carregar o artefato do contrato "Storage".

```
2.`module.exports = function (deployer) { ... }`
```

Este é o módulo exportado pelo script de migração. O objeto ``deployer`` é passado para esta função e é usado para implantar contratos na Ethereum.

```
3.`deployer.deploy(Storage);`
```

Aqui, estamos dizendo ao ``deployer`` para implantar o contrato ``Storage`` na Ethereum. O método ``deploy`` cuida de várias coisas, como a criação de uma transação para implantar o contrato, o envio da transação para a rede Ethereum e o acompanhamento da transação até que ela seja minerada.

Em resumo, este script de migração está implantando o contrato ``Storage`` na Ethereum. Para fazer isso, ele carrega o artefato do contrato ``Storage`` e, em seguida, usa o ``deployer`` para implantar o contrato.

Passo 4: Compilar e implantar o contrato

Se tudo correr bem, você deve ver uma saída indicando que o contrato foi compilado e implantado com sucesso.

Agora, seu contrato está implantado na rede de teste Ganache e você pode interagir com ele através do console Truffle, como descrito na pergunta anterior.

No diretório do seu projeto, você pode compilar e migrar seu contrato usando os seguintes comandos:

```
truffle compile
truffle migrate
```

Isso compilará seu contrato e implantá-lo na rede Ethereum emulada pelo Ganache. Se tudo ocorrer bem você deverá ver a tela abaixo:

```
D:\Livro>truffle migrate

Compiling your contracts...
=====
> Compiling .\contracts\Storage.sol
> Artifacts written to D:\Livro\build\contracts
> Compiled successfully using:
   - solc: 0.8.2+commit.661d1103.Emscripten.clang

Starting migrations...
=====
> Network name:      'development'
> Network id:       1686421991179
> Block gas limit:  6721975 (0x6691b7)

1_deploy_contracts.js
=====

Replacing 'Storage'
-----
> transaction hash:  0x004618a19b3c4749fcffa6115a14dc01dc5baad5f8250f0f40ddf5e33c697e9c
> Blocks: 0
> contract address: 0x128Ee7d55348e043a93092FFF08858Bb6c29ff52
> block number:     4
> block timestamp:  1686423236
> account:          0x5DE6a78cEC717203EC6ef2233Ef26EFCd8bD7228
> balance:          99.98366884
> gas used:         118819 (0x1d023)
> gas price:        20 gwei
> value sent:       0 ETH
> total cost:       0.00237638 ETH

> Saving artifacts
-----
> Total cost:       0.00237638 ETH

Summary
=====
> Total deployments: 1
> Final cost:       0.00237638 ETH
```

Passo 5: Interaja com seu Contrato

Após a migração, você pode interagir com seu contrato usando o console Truffle:

```
truffle console
```

A partir daqui, você pode chamar funções em seu contrato usando a sintaxe de JavaScript. Siga os passos abaixo:

1. Interaja com o contrato:

Primeiro, você precisa obter uma instância do seu contrato implantado. Você pode fazer isso usando a função `deployed()` do Truffle:

```
let instance = await Storage.deployed()
```

Agora, você pode chamar as funções do contrato usando essa instância.

Por exemplo, para chamar a função `store`, você pode usar o seguinte comando:

```
await instance.store(5)
```

Isso armazenará o número 5 no estado do contrato.

Para recuperar o número armazenado, você pode chamar a função `retrieve`:

```
let result = await instance.retrieve()
```

Depois de executar este comando, você pode ver o resultado chamando `result` no console:

```
result.toNumber()
```

Isso deve retornar o número que você armazenou anteriormente.

Por favor, note que essas instruções pressupõem que você está usando a versão mais recente do Truffle e que o seu contrato `Storage` foi compilado e implantado corretamente. As instruções também pressupõem que você está conectado a uma rede Ethereum através do Ganache, e que você tem uma conta Ethereum com algum Ether para pagar pelo gás das transações. Além disso, essas instruções podem variar dependendo da versão específica do Truffle e do Ganache que você está usando, por isso é sempre uma boa ideia verificar a documentação mais recente.

Para resumir, o Ganache é uma ferramenta valiosa para qualquer desenvolvedor de Solidity. Ele fornece uma maneira rápida e fácil de criar um ambiente de desenvolvimento de contratos inteligentes totalmente funcional, onde você pode testar e debugar seu código em um ambiente controlado antes de implantá-lo em uma rede de teste pública ou na rede principal Ethereum.

Configurando o Ganache com o Remix IDE

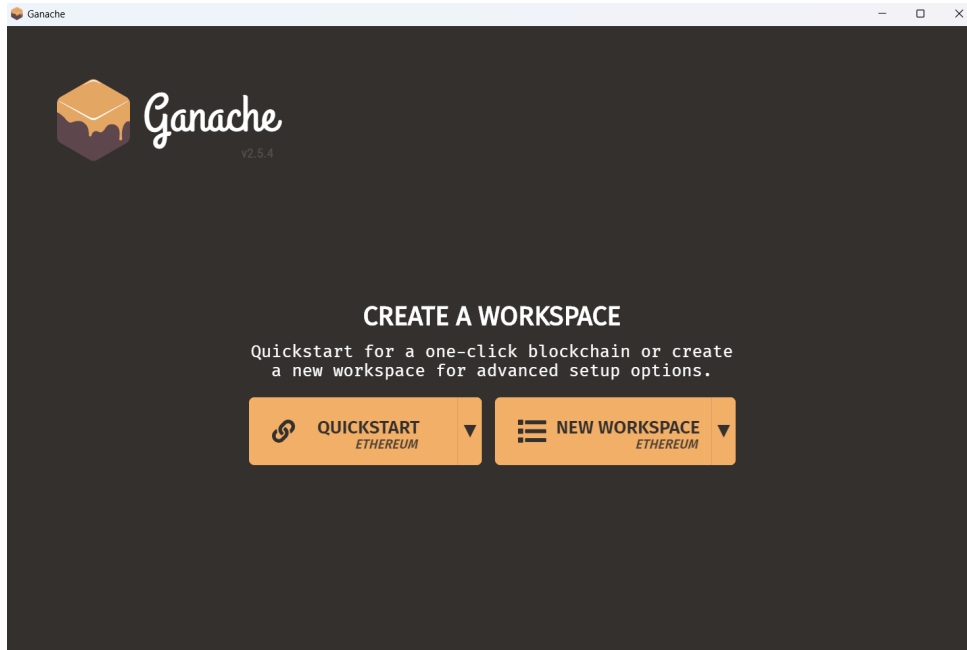
Como você já sabe o Remix IDE é um ambiente de desenvolvimento online para escrever contratos inteligentes em Solidity e para fazer deploy e interagir com eles. Embora o Remix tenha seu próprio ambiente de blockchain virtual para testes chamado JavaScript VM, também é possível conectar o Remix ao Ganache para fazer deploy e testar contratos. Isso permite que você tenha um ambiente de desenvolvimento mais realista que se aproxima de uma blockchain Ethereum real. Aqui estão os passos para conectar o Remix ao Ganache e fazer o deploy de um contrato:

Passo 1: Iniciar o Ganache

Primeiro, você precisa iniciar o Ganache. Se estiver usando o Ganache CLI, você pode simplesmente executar `ganache-cli` no terminal.

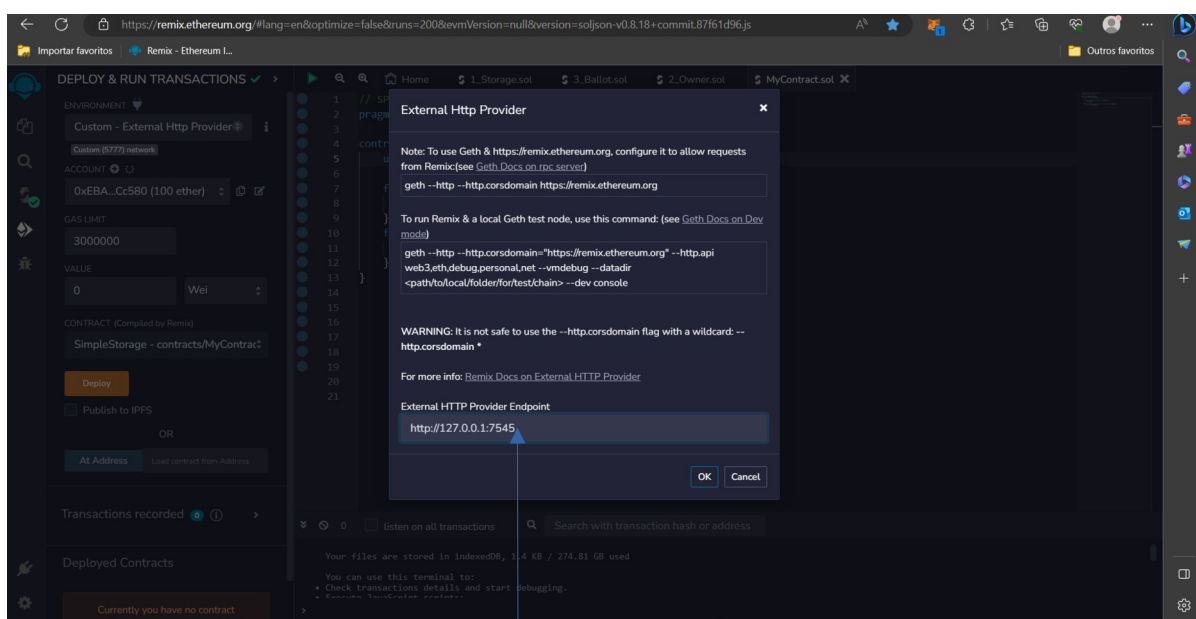
```
ganache-cli
```

Se estiver usando o Ganache GUI, abra o aplicativo e clique em "Quickstart" ou crie um novo workspace.



Passo 2: Conectar Remix ao Ganache

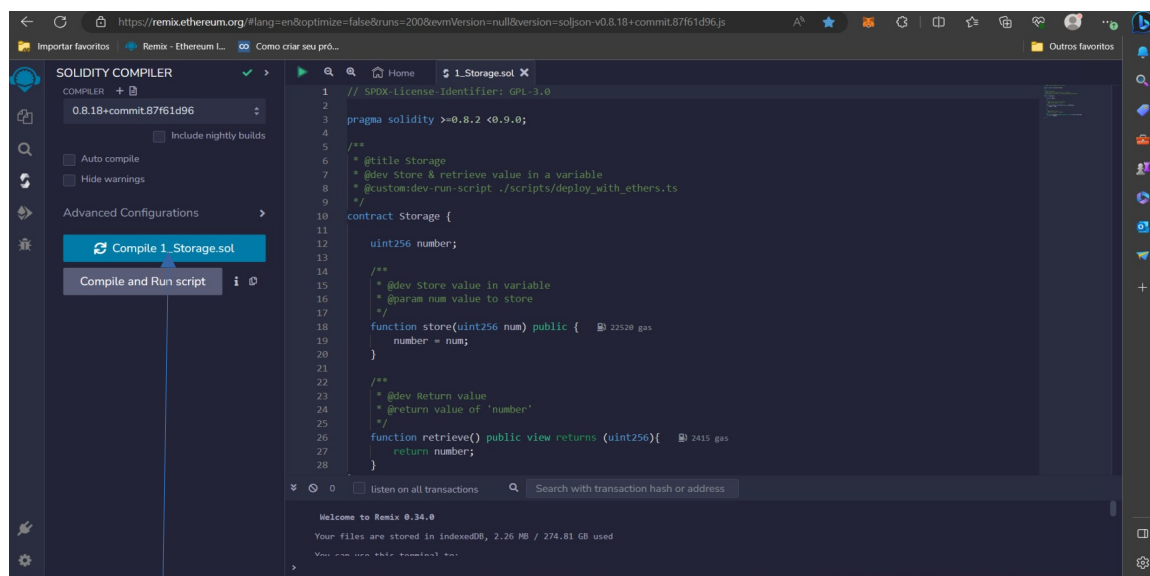
Depois que o Ganache estiver em execução, abra o Remix IDE no navegador e clique no ícone "Deploy & run transactions" na barra lateral à esquerda. Em "Environment", selecione "Custom – External http Provider". Em seguida, ele pedirá o endpoint da Web3. Se você estiver usando o Ganache localmente, provavelmente o endpoint será `http://127.0.0.1:7545`. Digite o endpoint e clique em "OK". Isso conectará o Remix ao Ganache.



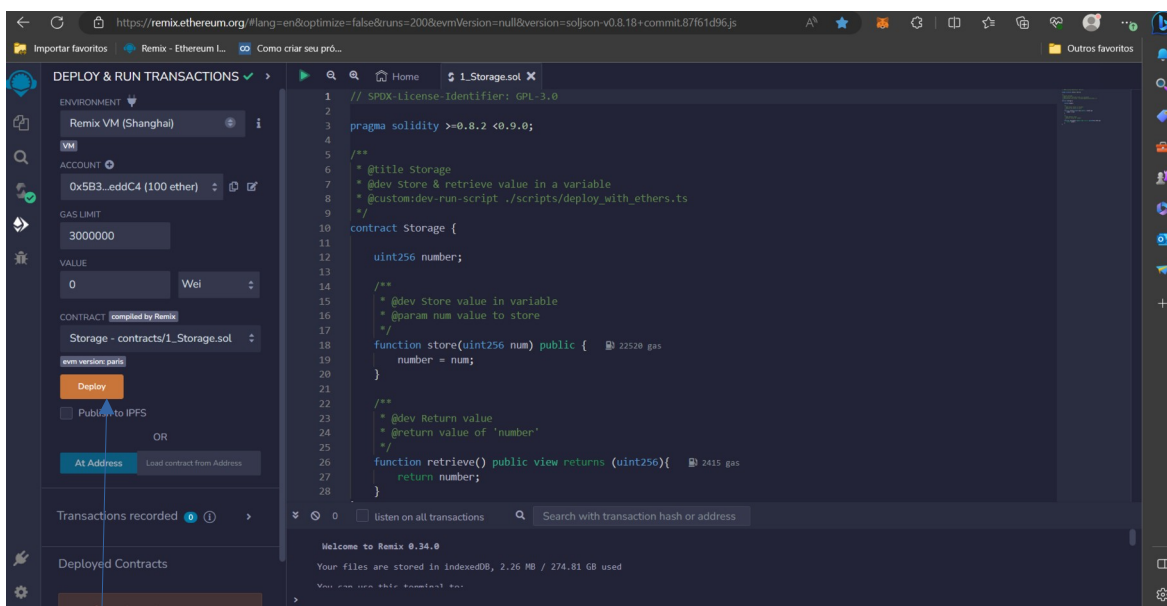
Endpoint

Passo 3: Compilar e Fazer Deploy do Contrato

Depois de conectar o Remix ao Ganache, você pode escrever ou carregar o código do seu contrato inteligente no Remix. Certifique-se de que o contrato esteja compilado sem erros. Para compilar, selecione o arquivo do contrato na lista de arquivos à esquerda, vá para a guia "Solidity compiler" e clique em "Compile". Depois que o contrato for compilado com sucesso, volte para a guia "Deploy & run transactions". Selecione o contrato que você deseja implantar na lista suspensa "Contract". Então, clique em "Deploy". Isso implantará o contrato na blockchain Ganache.



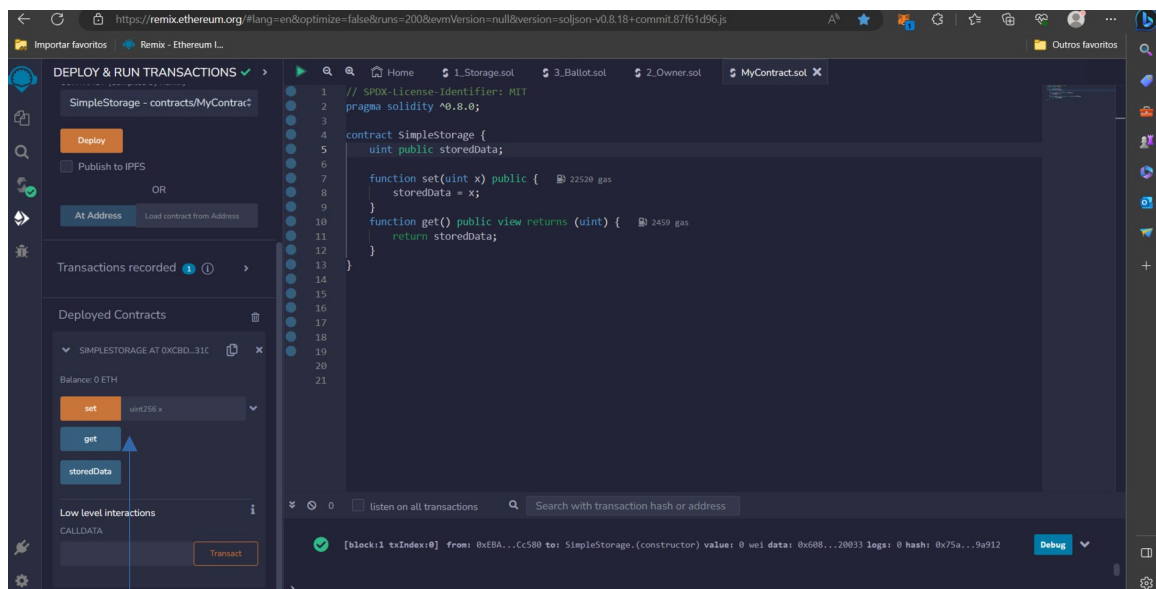
Botão para compilar o contrato



Botão para Deploy

Passo 4: Interagir com o Contrato

Após o deploy do contrato, ele aparecerá em "Deployed Contracts" na guia "Deploy & run transactions". Aqui, você pode interagir com as funções do contrato. Basta clicar nas funções e fornecer os argumentos necessários.

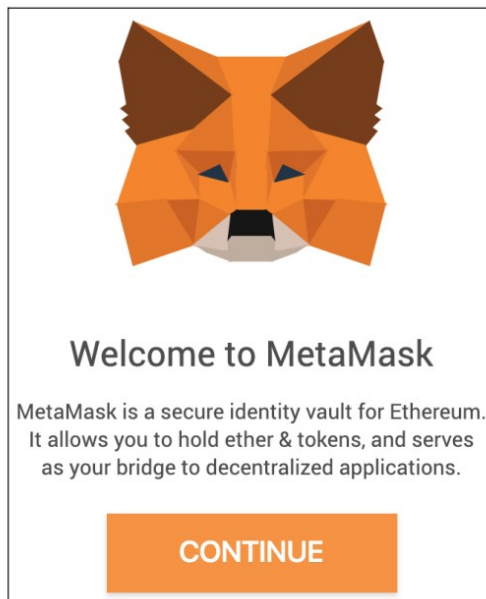


Funções do contrato

Em resumo, o Ganache e o Remix IDE podem ser usados juntos para proporcionar um ambiente de desenvolvimento completo para contratos inteligentes em Solidity, desde a escrita e compilação do código do contrato até o deploy e teste do contrato em um ambiente de blockchain realista.

2.4 Metamask

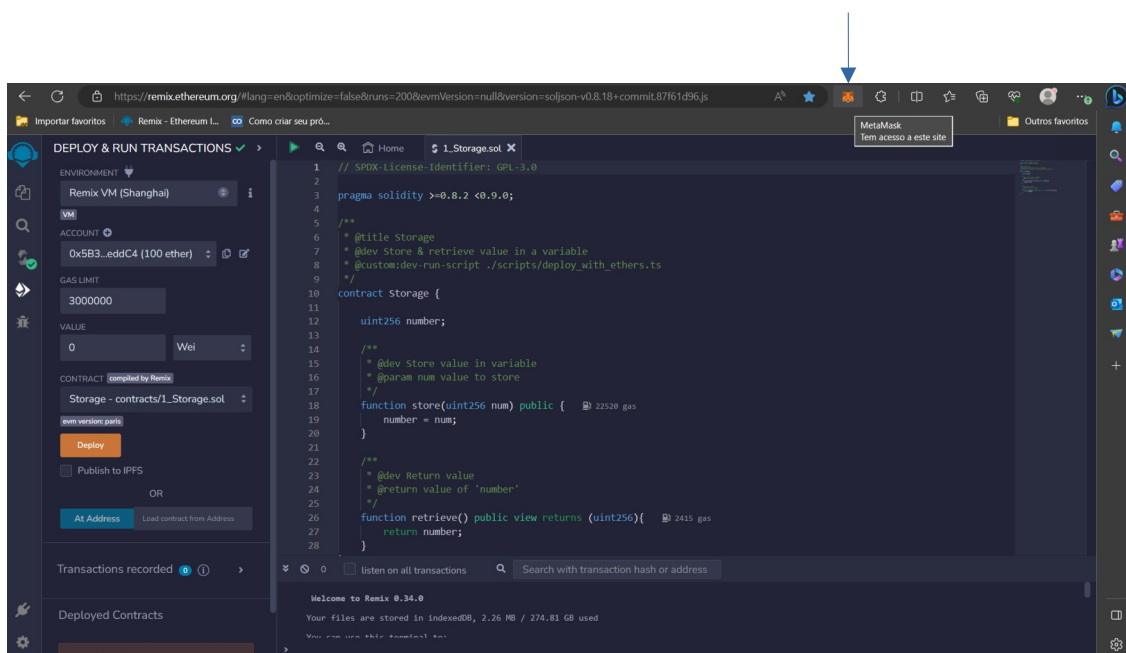
Metamask é uma extensão de navegador que permite aos usuários acessar e interagir com aplicativos descentralizados (dApps) baseados na tecnologia blockchain. Essa ferramenta revolucionária tem desempenhado um papel fundamental na adoção em massa das criptomoedas e na expansão do ecossistema da Web3.



Desenvolvido inicialmente como uma carteira Ethereum, o Metamask permite que os usuários armazenem, enviem e recebam tokens Ethereum e outros ativos digitais compatíveis com a rede Ethereum. Além disso, o Metamask também é compatível com outros blockchains e tokens, como Binance Smart Chain e Polygon, ampliando suas funcionalidades e alcance.

Uma das principais características do Metamask é a sua integração perfeita com navegadores populares, como Google Chrome, Firefox e Brave. Ao instalar o Metamask como uma extensão, os usuários podem criar e gerenciar suas carteiras digitais de forma fácil e segura diretamente no navegador. Essa acessibilidade é essencial para a adoção generalizada de criptomoedas, pois elimina a necessidade de baixar e configurar carteiras complicadas.

Extensão Metamask



Além de ser uma carteira digital, o Metamask oferece uma interface amigável para interagir com dApps. Quando os usuários acessam um dApp compatível, o Metamask detecta automaticamente a presença do aplicativo e permite que eles se conectem à sua carteira para executar transações e interagir com os recursos oferecidos pelo dApp. Isso torna a experiência de uso dos aplicativos descentralizados mais conveniente e segura, pois o Metamask solicita a aprovação do usuário antes de executar qualquer transação.

O Metamask também inclui recursos avançados de segurança, como a proteção por senha e a frase de recuperação (seed phrase). A frase de recuperação é uma sequência de palavras que permite aos usuários restaurar suas carteiras caso percam o acesso ao dispositivo. É extremamente importante manter a frase de recuperação em um local seguro, pois ela é a chave para recuperar a carteira em caso de perda ou roubo.

Com a ascensão da tecnologia blockchain e a popularização das criptomoedas, o Metamask desempenha um papel crucial na adoção dessas inovações. Ele facilita a participação dos usuários em dApps, proporcionando uma experiência amigável e segura. Além disso, o Metamask contribui para a criação de um ecossistema próspero de desenvolvedores, que podem criar e distribuir seus aplicativos descentralizados para milhões de usuários em todo o mundo.

2.4.1 Como instalar o Metamask

Claro! Aqui está um guia passo a passo sobre como instalar o Metamask em seu navegador:

Passo 1: Abra o navegador

Inicie seu navegador preferido, seja o Google Chrome, Firefox ou Brave. Certifique-se de que você esteja conectado à internet.

Passo 2: Acesse o site do Metamask

Digite "Metamask" na barra de pesquisa do seu navegador ou acesse diretamente o site oficial do Metamask em metamask.io.

Passo 3: Baixe a extensão do Metamask

No site do Metamask, você encontrará um botão de download. Clique nele para iniciar o processo de download da extensão do Metamask para o seu navegador.

Passo 4: Adicione a extensão ao navegador

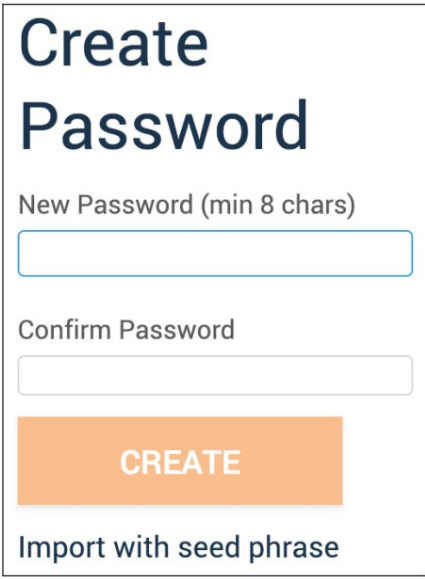
Após o download ser concluído, você verá uma notificação na parte superior do navegador. Clique nessa notificação ou procure pelo arquivo baixado em sua pasta de downloads e dê um clique duplo nele. Isso iniciará o processo de instalação da extensão do Metamask.

Passo 5: Configure sua carteira Metamask

Após a instalação, você verá o ícone do Metamask na barra de ferramentas do seu navegador. Clique nele para abrir a extensão. Uma nova guia será aberta com a tela de boas-vindas do Metamask.

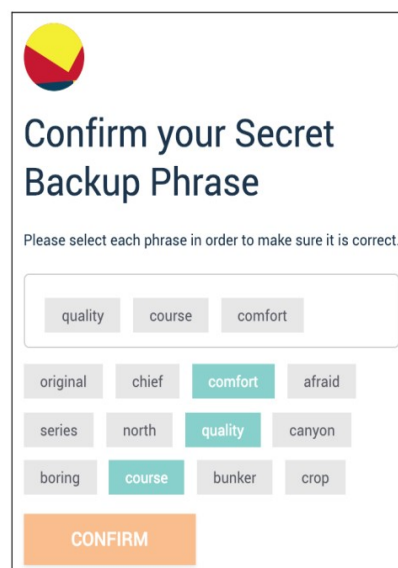
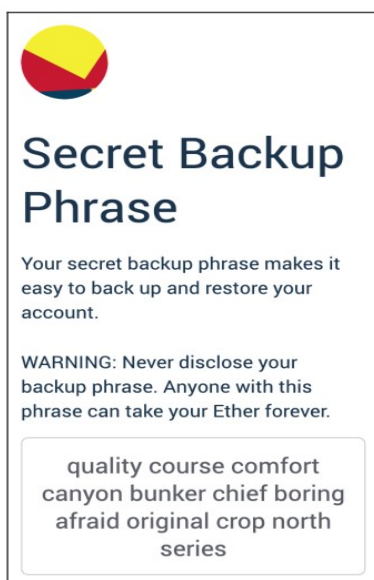
Passo 6: Crie uma nova carteira

Na tela de boas-vindas do Metamask, você terá a opção de criar uma nova carteira. Clique em "Create a Wallet" (Criar uma Carteira) e siga as instruções apresentadas. Você será solicitado a definir uma senha segura para sua carteira.

A screenshot of the Metamask 'Create Password' screen. It features a white background with a thin grey border. At the top, the words 'Create Password' are displayed in a large, bold, dark blue font. Below this, the text 'New Password (min 8 chars)' is shown in a smaller, grey font, followed by a white rectangular input field with a thin blue border. Underneath, the text 'Confirm Password' is shown in a smaller, grey font, followed by another white rectangular input field with a thin grey border. At the bottom of the form, there is a prominent orange rectangular button with the word 'CREATE' in white, uppercase letters. Below the button, the text 'Import with seed phrase' is displayed in a small, grey font.

Passo 7: Anote sua frase de recuperação

Após definir a senha, você será apresentado a uma frase de recuperação composta por 12 palavras. É extremamente importante anotar essa frase e guardá-la em um local seguro. Essa frase é a chave para recuperar sua carteira em caso de perda de acesso ao dispositivo.

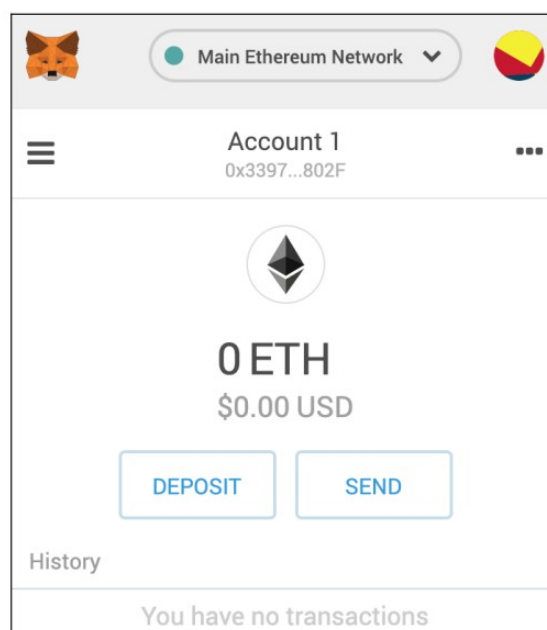


Passo 8: Aceite os termos de serviço

Após anotar sua frase de recuperação, você precisará confirmar que entendeu e concorda com os termos de serviço do Metamask.

Passo 9: Explore sua carteira Metamask

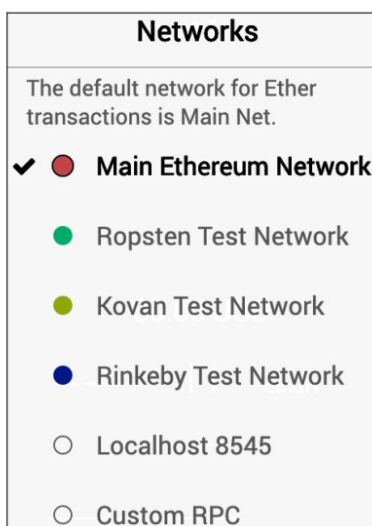
Após aceitar os termos de serviço, você terá acesso à sua carteira Metamask. Nessa interface, você pode visualizar seu saldo de criptomoedas, enviar e receber tokens, além de configurar as configurações da carteira de acordo com suas preferências.



Parabéns! Você instalou o Metamask com sucesso em seu navegador. Agora você pode explorar e interagir com aplicativos descentralizados (dApps) e aproveitar os recursos do ecossistema da Web3 diretamente no seu navegador. Lembre-se sempre de manter sua frase de recuperação em segurança e nunca compartilhá-la com ninguém.

2.4.2 Metamask e Networks

Uma das características mais notáveis do Metamask é a sua compatibilidade com diversas Networks além das Test Network da Ethereum. Essa flexibilidade permite que os usuários aproveitem as funcionalidades de outros ecossistemas blockchain, como Binance Smart Chain e Polygon, ampliando significativamente as opções disponíveis.



Com a crescente diversidade de blockchains e tokens, o Metamask desempenha um papel fundamental na integração dessas redes alternativas. Ao adicionar suporte para outras blockchains, o Metamask permite que os usuários acessem e gerenciem seus ativos digitais em diferentes redes, tudo a partir de uma única interface unificada.

Um exemplo notável é a compatibilidade com a Binance Smart Chain (BSC), uma blockchain desenvolvida pela exchange de criptomoedas Binance. Com o Metamask, os usuários podem facilmente adicionar a BSC como uma rede adicional em sua carteira, permitindo que eles interajam com tokens e aplicativos baseados na BSC sem a necessidade de uma carteira separada.

Além disso, o Metamask também é compatível com a Polygon, uma solução de escalabilidade para o Ethereum. Com a integração do Metamask, os usuários podem explorar o

ecossistema da Polygon, aproveitando transações rápidas e taxas mais baixas, enquanto ainda mantêm a segurança e a conveniência oferecidas pelo Metamask.

Essa compatibilidade com networks alternativas demonstra o compromisso do Metamask em promover a interoperabilidade e a acessibilidade no ecossistema blockchain. Ele capacita os usuários a aproveitar as vantagens de diferentes redes e ecossistemas, sem a necessidade de trocar de carteira ou aprender a usar diferentes interfaces complexas.

É importante ressaltar que, ao interagir com outras networks além do Ethereum, os usuários devem sempre estar atentos às diferenças nas propriedades e nas características dessas redes. Cada network possui suas próprias regras e padrões, portanto, é essencial entender os detalhes específicos de cada rede antes de realizar transações ou interações.

2.4.3 Como usar o Metamask no Remix IDE?

Para usar o Metamask no Remix IDE siga as seguintes etapas:

Passo 1: Instale o Metamask e conecte-se à sua carteira

Certifique-se de ter o Metamask instalado em seu navegador e que esteja conectado à sua carteira Ethereum. Se você ainda não tem o Metamask, consulte as instruções anteriores para instalá-lo.

Passo 2: Abra o Remix IDE

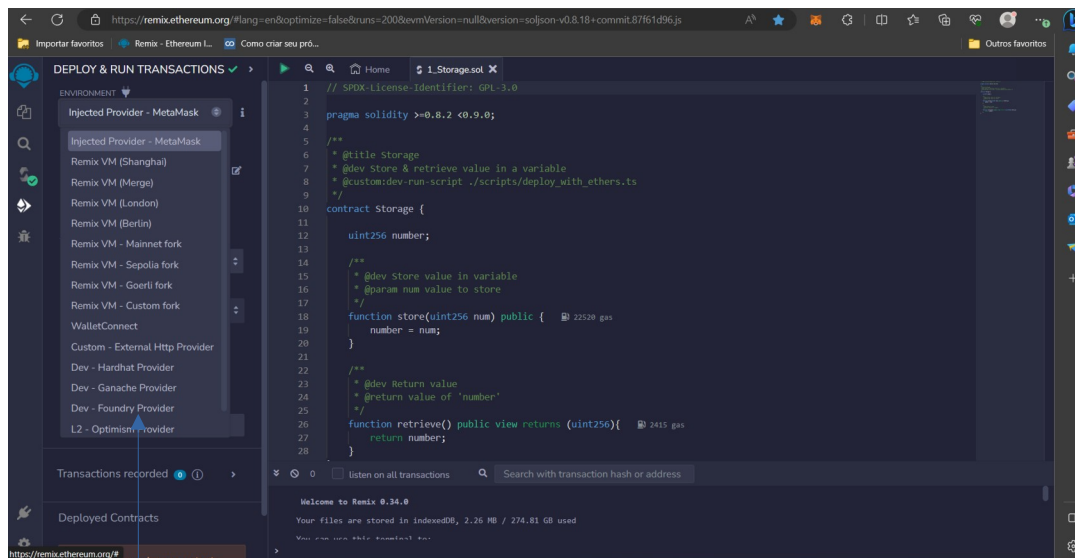
Acesse o site oficial do Remix IDE em remix.ethereum.org no seu navegador.

Passo 3: Configure o ambiente

No Remix IDE, clique na opção "Solidity" no canto superior esquerdo para escolher a versão do compilador Solidity que você deseja usar. Em seguida, clique em "Create" para criar um novo arquivo de contrato inteligente ou abra um arquivo existente.

Passo 4: Conecte o Remix IDE ao Metamask usando "injected Metamask"

No Remix IDE, você verá uma barra de ferramentas na parte superior. Nessa barra, você encontrará uma opção chamada "Environment" (Ambiente). Clique nela e selecione "injected Web3" ou "injected Metamask" no menu suspenso. Após selecionar "injected Metamask", o Remix IDE detectará automaticamente a conexão do Metamask em seu navegador.

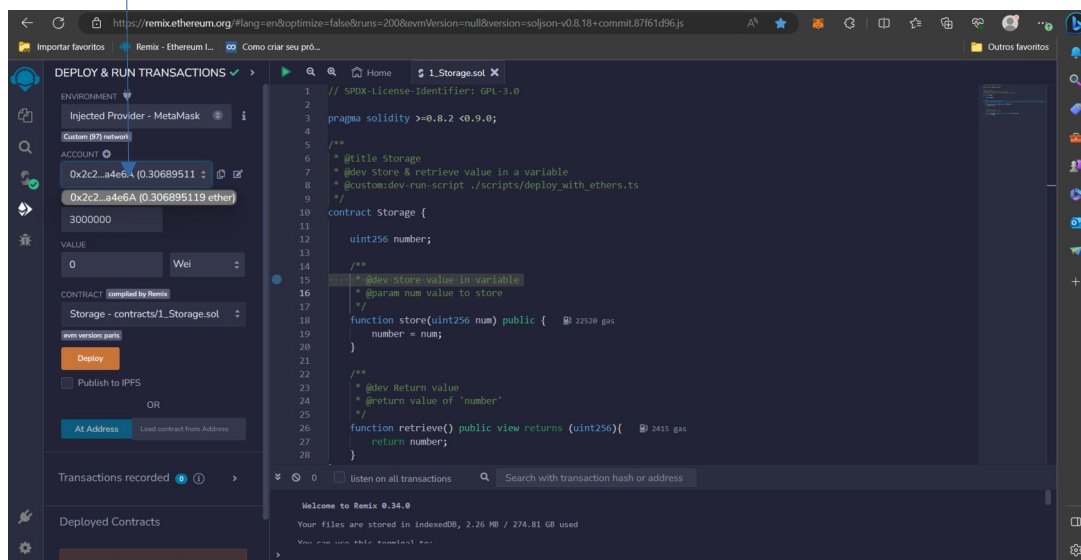


Injected Provider - Metamask

Passo 5: Escolha a conta do Metamask a ser usada

Escolha a conta que deseja usar para interagir com o Remix IDE e clique em "Next".

Conta



Passo 7: Pronto para usar o Metamask no Remix IDE

O Remix IDE agora está conectado ao Metamask usando a opção "injected Metamask". Você poderá ver o endereço da conta conectada na barra de status do Remix IDE. Isso indica que você está pronto para compilar, implantar e interagir com seus contratos inteligentes usando sua conta Metamask.

Agora você pode aproveitar todas as funcionalidades do Remix IDE e do Metamask juntos. Você pode compilar seus contratos, implantá-los em uma rede de teste ou na rede Ethereum principal e interagir com eles usando a interface amigável do Metamask. Certifique-se de ajustar as configurações do Metamask para selecionar a rede correta, dependendo do ambiente em que você está trabalhando (por exemplo, rede de teste Goerli, etc., ou a rede Ethereum principal).

Com o Metamask integrado ao Remix IDE usando a opção "injected Metamask", você terá uma poderosa combinação de ferramentas para desenvolver e testar seus contratos inteligentes de forma fácil e conveniente, facilitando o processo de criação de aplicativos descentralizados na rede Ethereum.

Capítulo 3: Conceitos Fundamentais da Programação Solidity

Agora que seu ambiente de desenvolvimento está pronto, podemos começar a explorar a programação Solidity. Neste capítulo, vamos discutir os conceitos fundamentais de Solidity.

3.1 Tipos de Dados em Solidity

Solidity é uma linguagem fortemente tipada. Alguns dos tipos de dados comuns incluem:

- **uint** (números inteiros sem sinal);
- **address** (endereços Ethereum);
- **bool** (valores booleanos);
- **bytes** (dados brutos); e
- **string** (cadeias de caracteres);

3.2 Variáveis e Funções

As variáveis são usadas para armazenar dados e as funções são usadas para manipular esses dados. As funções podem ser **públicas**, **privadas**, **internas** ou **externas**. As variáveis podem ser de **estado** (persistem entre chamadas de função) ou **locais** (existem apenas durante a execução de uma função).

3.2.1 Públicas

Funções e variáveis públicas podem ser acessadas de qualquer lugar, tanto de dentro quanto de fora do contrato. Se você declarar uma variável como pública, Solidity criará automaticamente uma função getter para essa variável.

Exemplo de função e variável públicas:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract MyContract {
    uint256 public myPublicVar;

    function myPublicFunction() public { 144 gas
        // ...
    }
}
```

3.2.2 Privadas

Funções e variáveis privadas só podem ser acessadas de dentro do próprio contrato. Elas não são visíveis em contratos herdados ou de fora do contrato.

Exemplo de função e variável privadas:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract MyContract {
    uint256 private myPrivateVar;

    function myPrivateFunction() private { infinite gas
        // ...
    }
}
```


3.2.3 Internas

Funções e variáveis internas podem ser acessadas de dentro do próprio contrato e por contratos que herdam do contrato. Eles não são acessíveis de fora do contrato.

Exemplo de função e variável internas:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract MyContract {
    uint256 internal myInternalVar;


    function myInternalFunction() internal {  infinite gas
    |   // ...
    }
}
```

3.2.4 Externas

Funções externas são semelhantes às funções públicas, mas elas só podem ser chamadas de fora do contrato e não de dentro (a menos que sejam chamadas com `this.`). Funções externas são mais eficientes em termos de gás quando recebem grandes quantidades de dados.

Exemplo de função externa:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract MyContract {
    function myExternalFunction() external {  122 gas
    |   // ...
    }
}
```

Note que as variáveis não podem ser externas, apenas as funções. Escolher a visibilidade correta para suas funções e variáveis é uma parte importante do desenvolvimento seguro de contratos inteligentes.

3.3 Variáveis Globais Solidity

As variáveis globais em Solidity fornecem informações sobre o ambiente blockchain atual. Essas variáveis podem ser usadas em qualquer lugar dentro de um contrato. Aqui estão algumas das variáveis globais mais comumente usadas:

1. **blockhash(uint blockNumber)**: Esta função retorna o hash do bloco especificado. O hash de bloco só pode ser acessado para os 256 blocos mais recentes.
2. **block.coinbase**: Este é o endereço do minerador do bloco atual.
3. **block.difficulty**: Representa a dificuldade do bloco atual.
4. **block.gaslimit**: Este é o limite de gás do bloco atual.
5. **block.number**: Este é o número do bloco atual.
6. **block.timestamp**: Esta é a marca de tempo do bloco atual. É definida quando o bloco é minado.
7. **gasleft()**: Esta função retorna a quantidade de gás ainda disponível na transação atual.
8. **msg.data**: Este é um conjunto completo de dados da chamada atual.
9. **msg.gas**: Esta é a quantidade de gás que foi enviada com a chamada atual.
10. **msg.sender**: Este é o remetente da chamada atual.
11. **msg.value**: Estes são os wei enviados com a chamada atual.
12. **now**: É um alias para `block.timestamp`.
13. **tx.gasprice**: Este é o preço do gás na transação atual.
14. **tx.origin**: É o endereço do remetente original da transação (contrato completo).

Abaixo está um exemplo de código em Solidity que demonstra o uso de algumas das variáveis globais:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

contract ExemploVariaveisGlobais {
    uint public blocoAtual;
    uint public dificuldade;
    uint public limiteGas;
    address public minerador;
    uint public valorTransacao;
    address public remetenteTransacao;
    uint public gasRestante;
    uint public timestampBloco;
    uint public precoGas;

    function atualizarVariaveis() public payable {
        blocoAtual = block.number;
        dificuldade = block.difficulty;
        limiteGas = block.gaslimit;
        minerador = block.coinbase;
        valorTransacao = msg.value;
        remetenteTransacao = msg.sender;
        gasRestante = gasleft();
        timestampBloco = block.timestamp;
        precoGas = tx.gasprice;
    }
}
```

Neste contrato, cada variável de estado corresponde a uma variável global em Solidity. A função `atualizarVariaveis()` é marcada como `public` e `payable` para que possa ser chamada por qualquer pessoa e aceite Ether. Quando chamada, ela atualiza cada variável de estado com o valor atual de sua respectiva variável global. Note que para obter o valor de `gasleft()`, `msg.value`, `msg.sender`, `block.timestamp` e `tx.gasprice` temos que realizar a transação, portanto a função é marcada como `payable`.

Estas são apenas algumas das variáveis globais em Solidity. O comportamento exato e a utilização dessas variáveis podem variar dependendo da versão específica de Solidity que você está usando, então é sempre uma boa ideia verificar a documentação mais recente.

3.4 Estruturas, Enumerações e Arrays

Solidity suporta estruturas (Structs), enumerações (Enums), mappings e arrays. Estruturas são usadas para agrupar variáveis relacionadas, enquanto as enumerações são usadas para criar tipos de dados personalizados. Um "mapping" em Solidity é uma estrutura de dados usada para armazenar pares de chave-valor, já arrays são estruturas de dados que permitem armazenar uma coleção de dados do mesmo tipo.

3.4.1 Structs

Em Solidity, um `struct` é uma maneira de definir um novo tipo de dados. É uma estrutura de dados composta que agrupa variáveis de outros tipos.

Por exemplo, imagine que você queira criar um sistema de registro de livros em um contrato inteligente. Você pode definir uma `struct` para um Livro como o seguinte:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract Library {
    struct Book {
        string title;
        string author;
        uint256 bookId;
    }

    Book public book;

    function setBook(string memory _title, string memory _author, uint256 _bookId) public {
        book = Book(_title, _author, _bookId);
    }
}
```

No exemplo acima, `Book` é uma `struct` que contém três campos: `title`, `author` e `bookId`. Agora, vamos falar sobre `enums`:

3.4.2 Enums

`Enums` são usados para criar um tipo de dados personalizado com um conjunto de constantes pré-definidas.

Aqui está um exemplo simples de um contrato que usa um `enum`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract Election {
    enum State { Created, Voting, Ended }
    State public state;

    constructor() {
        state = State.Created;
    }

    function startVoting() public {
        state = State.Voting;
    }

    function endVoting() public {
        state = State.Ended;
    }
}
```

No exemplo acima, `State` é um `enum` com três estados possíveis: `Created`, `Voting` e `Ended`. O contrato usa o `enum` para manter o controle do estado atual das eleições.

3.4.3 Mapping

Um mapping em Solidity é uma estrutura de dados que permite associar valores a chaves. Ele é declarado usando a palavra-chave `mapping`, seguida pelos tipos de dados da chave e do valor. Por exemplo, `mapping (address => uint256) balances;` declara um mapping chamado `balances` que mapeia endereços (chaves) para valores do tipo `uint256`. Os mappings são usados para criar tabelas de pesquisa eficientes, onde é possível acessar os valores rapidamente, fornecendo a chave correspondente.

Como usar um mapping:

1. Declaração: Comece declarando um mapping em seu contrato Solidity. Por exemplo:

```
mapping (address => uint256) public balances;
```

Este exemplo cria um mapping chamado `balances` que mapeia endereços para valores do tipo `uint256`. O `public` permite que outros contratos ou usuários acessem o mapping.

2. Atribuição de valores: Para atribuir um valor a uma chave específica no mapping, use a sintaxe `mapping[chave] = valor;`. Por exemplo:

```
balances[msg.sender] = 100;
```

Este exemplo atribui o valor `100` à chave `msg.sender`, que representa o endereço da conta que está fazendo a chamada.

3. Recuperação de valores: Para recuperar um valor associado a uma chave no mapping, use a sintaxe `valor = mapping[chave];`. Por exemplo:

```
uint256 myBalance = balances[msg.sender];
```

Este exemplo recupera o valor associado à chave `msg.sender` e o armazena na variável `myBalance`.

4. Atualização de valores: Para atualizar um valor existente em um mapping, simplesmente atribua um novo valor à chave correspondente. Por exemplo:

```
balances[msg.sender] = 200;
```

Este exemplo atualiza o valor associado à chave `msg.sender` para `200`.

Os mappings são úteis para armazenar informações relacionadas a um conjunto de chaves específicas. Eles são amplamente utilizados em contratos inteligentes para armazenar dados, como saldos de contas, registros de propriedade e muito mais.

É importante observar que mappings em Solidity são sempre inicializados com um valor padrão. No exemplo acima, todos os valores do mapping `balances` são inicializados como zero. Certifique-se de entender as implicações de segurança ao usar mappings. Por exemplo, você pode precisar implementar controles de acesso e lidar com colisões de chaves, dependendo dos requisitos do seu contrato. Espero que esta explicação ajude você a entender o conceito de mapping em Solidity e como usá-lo em seus contratos inteligentes!

3.4.4 Arrays

Arrays em Solidity são estruturas de dados que permitem armazenar uma coleção de elementos do mesmo tipo. Eles são usados para armazenar e acessar conjuntos de dados de forma organizada em contratos inteligentes.

Existem dois tipos principais de arrays em Solidity: arrays estáticos e arrays dinâmicos.

1. Arrays Estáticos:

Arrays estáticos têm um tamanho fixo definido na declaração e não podem ser alterados após a inicialização. Eles são declarados indicando o tipo de elemento e o tamanho do array. Por exemplo, `uint256[5] public myArray;` declara um array estático chamado `myArray` que pode conter 5 elementos do tipo `uint256`.

Os elementos de um array estático são inicializados com um valor padrão (zero para tipos numéricos, endereço vazio para tipos de endereço, etc.). Os elementos do array podem ser acessados e atualizados usando índices, começando do índice 0 até o tamanho do array menos 1. Por exemplo, `myArray[2]` acessa o terceiro elemento do array.

```
// Declaração de um array estático de números inteiros
uint256[5] public myArray;

function setElement(uint256 index, uint256 value) public {  infinite gas
    // Atribui um valor a um elemento específico do array
    myArray[index] = value;
}

function getElement(uint256 index) public view returns (uint256) {  infinite gas
    // Retorna o valor de um elemento específico do array
    return myArray[index];
}
```

2. Arrays Dinâmicos:

Arrays dinâmicos têm um tamanho variável e podem ser redimensionados após a inicialização. Eles são declarados indicando apenas o tipo de elemento. Por exemplo, `uint256[] public myDynamicArray;` declara um array dinâmico chamado `myDynamicArray` que pode conter qualquer número de elementos do tipo `uint256`. Os elementos de um array dinâmico são inicializados como vazios ou nulos.

Para adicionar elementos a um array dinâmico, você pode usar a função `push`. Por exemplo, `myDynamicArray.push(10);` adiciona o valor `10` ao final do array. Os elementos do array podem ser acessados e atualizados da mesma forma que em um array estático, usando índices.

```
// Declaração de um array dinâmico de strings
string[] public myDynamicArray;

function addElement(string memory element) public {  infinite gas
    // Adiciona um elemento ao array dinâmico
    myDynamicArray.push(element);
}

function getLength() public view returns (uint256) {  2511 gas
    // Retorna o tamanho do array dinâmico
    return myDynamicArray.length;
}

function getElement(uint256 index) public view returns (string memory) {  infinite gas
    // Retorna o valor de um elemento específico do array dinâmico
    require(index < myDynamicArray.length, "Invalid index");
    return myDynamicArray[index];
}
```

Arrays são usados em contratos inteligentes para armazenar e manipular conjuntos de dados. Eles podem ser usados para implementar listas, registros, históricos e muito mais. Além disso, Solidity fornece várias funções e propriedades embutidas para trabalhar com arrays, como `length` (para obter o tamanho do array), `push` (para adicionar um elemento) e `pop` (para remover o último elemento).

É importante observar que arrays podem consumir muita memória ou armazenamento, especialmente arrays dinâmicos com muitos elementos. Portanto, é necessário considerar o impacto nos limites de gas e nos custos de armazenamento ao usar arrays em contratos inteligentes.

3.5 Função e Modificadores de Função

As funções são um componente essencial de um contrato inteligente em Solidity. Elas contêm a lógica do contrato e definem o comportamento do contrato ao receber uma transação. Existem vários tipos de funções em Solidity, incluindo funções de visibilidade, funções de estado mutável e funções especiais.

Vamos dar uma olhada em um exemplo:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

contract MeuContrato {
    uint public contador = 0;

    function incrementarContador() public {
        contador += 1;
    }

    function obterContador() public view returns (uint) {
        return contador;
    }
}
```

Neste contrato simples, temos duas funções:

1. **incrementarContador**: Esta é uma função pública que incrementa o valor do contador. A palavra-chave ``public`` define a visibilidade da função, significando que esta função pode ser chamada de fora do contrato. Quando chamada, esta função muda o estado do contrato, então ela consome gas.

2. **obterContador**: Esta é também uma função pública, mas com a palavra-chave ``view``. As funções ``view`` não alteram o estado do contrato e, portanto, não consomem gas quando chamadas de fora de uma transação. Esta função retorna o valor atual do contador.

Note que o Solidity permite a sobrecarga de funções, o que significa que você pode ter várias funções com o mesmo nome, mas com diferentes tipos ou números de parâmetros. Existem também outras palavras-chave que você pode usar para especificar mais detalhes sobre suas funções, como ``external``, ``internal``, ``pure``, ``payable``, e assim por diante, cada uma com suas próprias características especiais. Além disso, Solidity também tem funções especiais como a função ``fallback`` e a função ``receive`` que são executadas quando o contrato recebe Ether sem que nenhuma função específica seja chamada.

Os modificadores de função são usados para alterar o comportamento de uma função. Por exemplo, o modificador ``payable`` permite que uma função receba Ether. Aqui está um exemplo de como o modificador ``payable`` pode ser usado em um contrato:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract MyContract {
    // Evento para emitir o endereço do remetente e o valor enviado
    event Received(address sender, uint amount);

    // Função para receber Ether
    receive() external payable {
        // Emitir um evento para registrar as informações do pagamento
        emit Received(msg.sender, msg.value);
    }
}
```

Neste exemplo, o contrato tem uma única função chamada ``receive``. Esta função é marcada com o modificador ``payable``, o que significa que ela pode receber Ether. Quando alguém envia Ether para o contrato (chamando a função ``receive`` e enviando Ether junto com a transação), a função emite um evento que registra o endereço do remetente e a quantidade de Ether enviada.

Esse é um exemplo muito básico, mas é uma prática comum usar o modificador `payable` em funções que precisam interagir com Ether, seja recebendo pagamentos, cobrando taxas, etc. Já uma função marcada como `pure` é uma função que não acessa nem modifica o estado do contrato.

Ela não lê nem escreve em variáveis de estado e não emite eventos. Essas funções são usadas quando você precisa executar cálculos ou transformações de dados sem interagir com a blockchain ou o contrato. As funções `pure` podem ser chamadas internamente dentro de um contrato ou externamente por outros contratos ou transações.

Exemplo de função `pure`:

```
function add(uint256 a, uint256 b) public pure returns (uint256) {  
    return a + b;  
}
```

Neste exemplo, a função `add` é marcada como `pure`. Ela recebe dois números inteiros (`a` e `b`) como parâmetros e retorna a soma desses números. Como a função não acessa o estado do contrato, ela pode ser marcada como `pure`.

Uma função marcada como `view` é uma função que não modifica o estado do contrato, mas pode ler dados do estado do contrato. Essas funções são usadas quando você precisa consultar informações do estado do contrato sem alterá-lo. As funções `view` podem ser chamadas internamente dentro de um contrato ou externamente por outros contratos ou transações.

Exemplo de função `view`:

```
function getBalance(address account) public view returns (uint256) {  
    return balances[account];  
}
```

Neste exemplo, a função `getBalance` é marcada como `view`. Ela recebe um endereço de conta como parâmetro e retorna o saldo associado a essa conta. Como a função não modifica o estado do contrato, mas apenas lê o valor do saldo, ela pode ser marcada como `view`.

É importante observar que as funções `pure` e `view` são otimizações de estado em Solidity e fornecem informações adicionais para o compilador. Ao marcar uma função corretamente, você ajuda a garantir a correta interação com o contrato e pode obter benefícios, como a redução no consumo de gás.

Ao utilizar essas funções, certifique-se de marcá-las corretamente de acordo com suas funcionalidades específicas e verifique se elas estão de acordo com as restrições e requisitos do seu contrato.

3.5.1 Função Construtor

Em Solidity, um construtor é uma função especial que é executada durante a criação do contrato e não pode ser chamada novamente após o contrato ser implantado. Ele é comumente utilizado para inicializar variáveis de estado do contrato.

Um construtor é declarado com a palavra-chave `constructor`. Ele pode ter parâmetros e ser marcado como `public` ou `internal`. Se nenhuma visibilidade é especificada, ele é considerado `public` por padrão. Vamos olhar para um exemplo de um construtor em Solidity:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

contract MeuContrato {
    uint public valor;
    address public dono;

    constructor(uint _valor) {
        valor = _valor;
        dono = msg.sender;
    }
}
```

Neste exemplo, o construtor recebe um parâmetro `_valor`, que é utilizado para inicializar a variável de estado `valor`. Também declaramos a variável `dono` e a inicializamos com `msg.sender`, que é a conta que implantou o contrato. Isso é feito dentro do construtor, portanto é executado apenas uma vez, quando o contrato é implantado.

Note que, após a implantação, o construtor não pode ser chamado novamente, portanto a `dono` e `valor` não podem ser alterados. Isso é útil para definir configurações imutáveis ou para definir a propriedade de um contrato. No entanto, você precisa ter cuidado ao inicializar seu contrato, pois erros nesse processo não podem ser corrigidos sem implantar um novo contrato.

3.6 Laços e Estruturas de Decisão

Os laços (loops) são estruturas de controle que permitem repetir um conjunto de instruções várias vezes em um contrato em Solidity. Existem três tipos principais de laços em Solidity: `for`, `while` e `do-while`.

1. Laço `for`:

O laço `for` é usado quando você sabe exatamente quantas vezes deseja repetir um conjunto de instruções. É composto por uma inicialização, uma condição de continuação e uma atualização.

Exemplo de laço `for`:

```
function sum(uint256[] memory numbers) public pure returns (uint256) {  
    uint256 total = 0;  
    for (uint256 i = 0; i < numbers.length; i++) {  
        total += numbers[i];  
    }  
    return total;  
}
```

Neste exemplo, o laço `for` itera sobre os elementos de um array `numbers` e soma cada elemento ao total. O laço começa com `i = 0`, continua enquanto `i` for menor que o comprimento do array `numbers` e incrementa `i` a cada iteração.

2. Laço `while`:

O laço `while` é usado quando você deseja repetir um conjunto de instruções enquanto uma condição for verdadeira. A condição é verificada antes da execução do bloco de código do laço.

Exemplo de laço `while`:

```
function factorial(uint256 number) public pure returns (uint256) {  
    uint256 result = 1;  
    while (number > 0) {  
        result *= number;  
        number--;  
    }  
    return result;  
}
```

Neste exemplo, o laço `while` calcula o fatorial de um número `number`. O laço continua enquanto `number` for maior que zero e multiplica o `result` pelo `number` em cada iteração.

3. Laço `do-while`:

O laço `do-while` é semelhante ao laço `while`, mas a condição é verificada após a execução do bloco de código. Isso garante que o bloco de código seja executado pelo menos uma vez.

Exemplo de laço `do-while`:

```
function countDigits(uint256 number) public pure returns (uint256) {  
    uint256 count = 0;  
    do {  
        number /= 10;  
        count++;  
    } while (number != 0);  
    return count;  
}
```

Neste exemplo, o laço `do-while` conta o número de dígitos em um número `number`. O laço divide o `number` por 10 em cada iteração até que o `number` seja igual a zero. É importante ter cuidado ao usar laços em contratos inteligentes, pois eles podem afetar o custo de execução e o limite de gas. Certifique-se de que os laços não sejam infinitos e que não haja risco de exceder os limites de gas. Os laços são ferramentas poderosas para controlar a repetição de instruções em Solidity. Ao combiná-los com estruturas de dados como arrays e mappings, você pode criar lógicas complexas e realizar cálculos iterativos em seus contratos.

4. Estruturas de Decisão:

As estruturas de decisão permitem que você tome decisões com base em condições lógicas dentro de um contrato em Solidity. As estruturas de decisão mais comuns em Solidity são `if-else` e `switch`.

5. Estrutura `if-else`:

A estrutura `if-else` permite executar diferentes blocos de código com base em uma condição.

Exemplo de estrutura `if-else`:

```
function checkEven(uint256 number) public pure returns (string memory) {  
    if (number % 2 == 0) {  
        return "Even";  
    } else {  
        return "Odd";  
    }  
}
```

Neste exemplo, a estrutura `if-else` verifica se um número é par ou ímpar. Se o número for divisível por 2 (ou seja, o resto da divisão for zero), a função retorna "Even" (par). Caso contrário, retorna "Odd" (ímpar).

6. Estrutura `switch`:

A estrutura `switch` permite executar diferentes blocos de código com base em uma expressão que assume diferentes valores.

Exemplo de estrutura `switch`:

```
function getGrade(uint256 score) public pure returns (string memory) {  
    string memory grade;  
    switch (score) {  
        case 90:  
            grade = "A";  
            break;  
        case 80:  
            grade = "B";  
            break;  
        case 70:  
            grade = "C";  
            break;  
        case 60:  
            grade = "D";  
            break;  
        default:  
            grade = "F";  
            break;  
    }  
    return grade;  
}
```

Neste exemplo, a estrutura `switch` converte um valor do tipo `Day` em uma string correspondente ao dia da semana. Cada caso representa um valor possível para `day` e executa o bloco de código correspondente. O bloco `default` é executado se nenhum dos casos corresponder ao valor de `day`.

É importante ter cuidado ao usar estruturas de decisão em contratos inteligentes, pois elas podem afetar o custo de execução e o limite de gas. Certifique-se de que as condições sejam bem definidas e abrangentes o suficiente para cobrir todos os casos possíveis. As estruturas de decisão são fundamentais para criar lógicas condicionais em contratos inteligentes e permitem que você tome diferentes ações com base em condições específicas. Espero que esses exemplos ajudem a ilustrar o uso das estruturas de decisão em Solidity!

3.7 Eventos

Os eventos em Solidity são usados para registrar a ocorrência de ações ou mudanças de estado no blockchain. Eles são especialmente úteis para interfaces de usuário, pois permitem o monitoramento de transações específicas.

Os eventos em Solidity são extremamente úteis para log e monitoramento de transações. Os eventos não podem ser lidos a partir de dentro do contrato, mas podem ser pesquisados e lidos a partir do histórico de log da blockchain. Vamos criar um exemplo simples onde um evento é emitido quando uma nova tarefa é adicionada a uma lista de tarefas.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract ToDoList {
    // Estrutura para representar uma tarefa
    struct Task {
        string description;
        bool completed;
    }

    // Array para armazenar tarefas
    Task[] public tasks;

    // Evento que será emitido quando uma nova tarefa for adicionada
    event TaskCreated(string description, uint256 taskId);

    // Função para adicionar uma tarefa
    function addTask(string memory _description) public {
        tasks.push(Task(_description, false));
        emit TaskCreated(_description, tasks.length - 1);
    }
}
```

Neste contrato, temos uma `struct` chamada `Task` que representa uma tarefa com uma descrição e um status booleano para indicar se a tarefa foi concluída ou não. Temos um array `tasks` para armazenar todas as tarefas e um evento `TaskCreated` que será emitido sempre que uma nova tarefa for adicionada à lista.

A função `addTask` é usada para adicionar uma nova tarefa à lista. Ela primeiro adiciona a tarefa ao array e, em seguida, emite o evento `TaskCreated`, passando a descrição da tarefa e o ID da tarefa (que é o índice da tarefa no array).

Capítulo 4: Contratos Inteligentes

Os contratos inteligentes são o núcleo das aplicações descentralizadas no Ethereum. Neste capítulo, vamos explorar os conceitos e a funcionalidade dos contratos inteligentes em maior profundidade.

4.1 O que são Contratos Inteligentes?

Contratos inteligentes são programas que facilitam, verificam e aplicam a negociação ou execução de um contrato. Eles são a unidade fundamental de código na blockchain Ethereum e incluem regras sobre transações e a lógica de execução automática quando as condições predefinidas são atendidas. Eles são autoexecutáveis e não necessitam de intermediários para funcionar. Isso torna as transações transparentes, irreversíveis e diretas. A estrutura básica de um contrato inteligente Solidity pode ser dividida em várias partes. Aqui está um exemplo de como isso pode parecer:

```
// SPDX-License-Identifier: MIT
// Versão do compilador
pragma solidity ^0.8.4;

// Definição do contrato
contract NomeDoContrato {
    // Variáveis de estado
    uint public variavelDeEstado;

    // Eventos
    event LogEvento(address indexed de, uint valor);

    // Modificadores
    modifier onlyOwner {
        require(msg.sender == owner, "Somente o proprietário pode executar isso");
        _;
    }

    // Funções
    function nomeFuncao(uint _valor) public onlyOwner {
        variavelDeEstado = _valor;
        emit LogEvento(msg.sender, _valor);
    }
}
```

Aqui estão os componentes principais:

1. **Versão do Compilador** (`pragma solidity ^0.8.4;`): Esta linha de código especifica a versão do compilador Solidity que o contrato deve usar. Isso ajuda a prevenir problemas de compatibilidade.
2. **Definição do Contrato** (`contract NomeDoContrato {...}`): Aqui é onde o contrato é definido. Um contrato em Solidity é semelhante a uma classe em linguagens orientadas a objetos.
3. **Variáveis de Estado**: Estas são variáveis que são armazenadas permanentemente no contrato. No exemplo acima, `variavelDeEstado` é uma variável de estado.
4. **Eventos** (`event LogEvento...`): Os eventos permitem que o contrato emita logs que o frontend da aplicação pode ouvir. Eles são usados para emitir logs que as interfaces do usuário, ou outros contratos, podem reagir.
5. **Modificadores** (`modifier onlyOwner {...}`): Modificadores são usados para mudar o comportamento de funções. Eles podem ser usados para verificar pré-condições antes de uma função ser executada.
6. **Funções** (`function nomeFuncao...`): As funções contêm a lógica do contrato. No exemplo acima, `nomeFuncao` é uma função que altera o valor da variável de estado e emite um evento.

Essa é uma visão simplificada da estrutura de um contrato inteligente. Os contratos reais podem ter muitos outros componentes, como enumerações, structs, construtores e funções de fallback, entre outros.

4.2 Propriedades dos Contratos Inteligentes

Contratos inteligentes têm várias propriedades importantes:

1. **Autonomia**: Uma vez implantados na blockchain, eles operam de forma independente de terceiros.
2. **Imutabilidade**: Depois de implantado, o código do contrato não pode ser alterado.

3. **Distribuição:** Eles existem em todas as cópias da blockchain Ethereum.
4. **Transparência:** O código-fonte dos contratos é visível para todos na rede.

4.3 Como Contratos Inteligentes Interagem


Contratos inteligentes podem interagir uns com os outros através de chamadas de função. Um contrato pode chamar uma função em outro contrato. As interações entre contratos são registradas e verificáveis, como qualquer outra transação na blockchain.

4.4 Escrevendo um Contrato Inteligente Interativo

Vamos escrever dois contratos que interagem entre si. Temos um contrato `Caller` que chama uma função no contrato `Callee`.


```
// SPDX-License-Identifier: MIT
import "./Callee.sol";

pragma solidity ^0.8.0;

contract Caller {
    function callSetX(address _callee, uint _x) public {  infinite gas
        Callee callee = Callee(_callee);
        callee.setX(_x);
    }
}
```

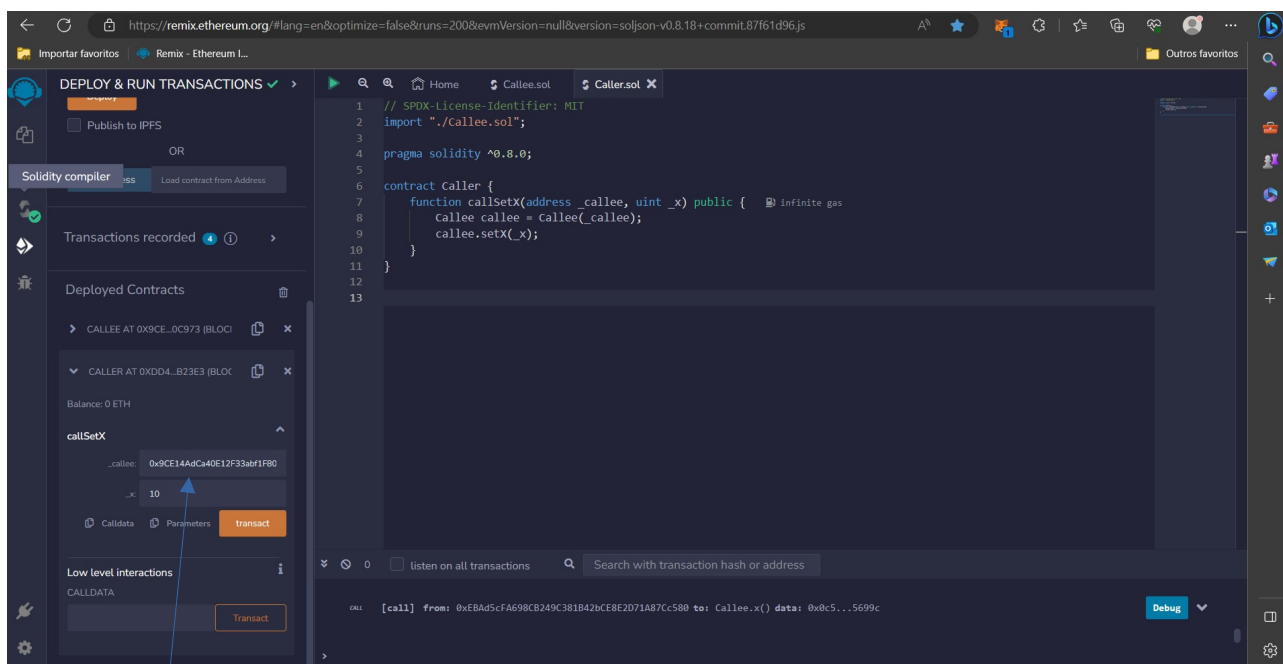
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Callee {
    uint public x;

    function setX(uint _x) public {  22520 gas
        x = _x;
    }
}
```

4.5 Compilando e Executando Contratos Interativos com Truffle e Remix IDE

Da mesma forma que no Capítulo 3, você pode compilar e executar esses contratos usando o Truffle ou o Remix IDE. Lembre-se de que, ao usar o contrato `Caller`, você precisará fornecer o endereço do contrato `Callee` implantado.



Endereço do contrato Callee

Os contratos inteligentes oferecem muitas possibilidades para a criação de aplicações descentralizadas. No próximo capítulo, aprofundaremos ainda mais os contratos inteligentes, explorando tópicos como herança, interfaces e bibliotecas.

Capítulo 5: Aprofundamento em Contratos Inteligentes

Depois de cobrir os conceitos básicos de contratos inteligentes, estamos prontos para nos aprofundar em tópicos mais avançados. Neste capítulo, exploraremos herança, interfaces e bibliotecas em Solidity.

5.1 Herança em Solidity

Solidity suporta herança, permitindo que um contrato herde de outro contrato. A herança é útil para compartilhar código comum e simplificar o design de contratos complexos.

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

contract Base {
    uint public x;

    function setX(uint _x) public { 22520 gas
        x = _x;
    }
}
```

```
// SPDX-License-Identifier: MIT
import "./Base.sol";

pragma solidity ^0.8.0;
contract Derived is Base {
    function incrementX() public { infinite gas
        x += 1;
    }
}
```

Neste exemplo, `Derived` é um subcontrato de `Base` e herda a variável `x` e a função `setX()`.

5.2 Interfaces em Solidity

As interfaces são uma maneira de definir como um contrato deve se comportar, sem implementar a lógica. As interfaces são úteis quando você quer garantir que um contrato se comportará de uma certa maneira, ou quando quer interagir com um contrato existente cujo código fonte você não possui.

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

// Interface
interface IMyInterface {
    function myFunction(uint256 value) external returns (uint256);  - gas
}

// Contrato que implementa a interface
contract MyContract {
    function doSomething(IMyInterface myContractAddress, uint256 value) external returns (uint256) {  undefined gas
        // Chama a função myFunction no contrato myContractAddress
        return myContractAddress.myFunction(value);
    }
}
```

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

// Interface
interface IMyInterface {
    function myFunction(uint256 value) external returns (uint256);  - gas
}

// Contrato que implementa a interface
contract MyOtherContract is IMyInterface {
    function myFunction(uint256 value) external pure override returns (uint256) {  infinite gas
        // Implementação da função myFunction
        return value * 2;
    }
}
```

Neste exemplo, temos uma interface chamada `IMyInterface`, que define uma função chamada `myFunction`. Em seguida, temos um contrato chamado `MyContract` que possui uma função `doSomething` que recebe um endereço de contrato `IMyInterface` e um valor. Essa função chama a função `myFunction` no contrato `myContractAddress`.


Temos também outro contrato chamado `MyOtherContract`, que implementa a interface `IMyInterface`. Ele fornece a implementação da função `myFunction`, que simplesmente retorna o dobro do valor fornecido.

Você pode implantar os contratos `MyContract` e `MyOtherContract` em uma blockchain compatível com Solidity e interagir com eles para ver como a interface é usada para chamar funções em contratos que implementam essa interface.

5.3 Bibliotecas em Solidity

As bibliotecas são um recurso poderoso que permite que você escreva código que pode ser reutilizado em vários contratos sem herança. As bibliotecas são especialmente úteis para operações complexas em tipos de dados.

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


library SafeMath {
    function add(uint a, uint b) internal pure returns (uint) {  infinite gas
        uint c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }
}
```

```
// SPDX-License-Identifier: MIT
import "./SafeMath.sol";

pragma solidity ^0.8.0;

contract MyContract {
    using SafeMath for uint;

    function add(uint a, uint b) public pure returns (uint) {  infinite gas
        return a.add(b);
    }
}
```

Neste exemplo, `MyContract` usa a biblioteca `SafeMath` para realizar uma adição segura. Agora que entendemos os conceitos avançados dos contratos inteligentes, estamos prontos para explorar como o Ether é usado e transferido em Solidity no próximo capítulo.

5.4 Require

No Solidity, `require` é uma palavra-chave que você pode usar para garantir que determinadas condições sejam atendidas antes de continuar a execução de uma função. Se a condição passada para `require` for falsa, a execução será interrompida e todas as alterações de estado serão revertidas. Isso é muito útil para validar entradas ou condições de contrato antes de prosseguir com a execução.

Por exemplo, imagine que você tenha um contrato inteligente que permite apenas ao proprietário do contrato retirar fundos. Você pode usar `require` para garantir que apenas o proprietário possa fazer isso:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

contract SimpleBank {
    address owner;

    constructor() public {
        owner = msg.sender;
    }

    function withdraw(uint amount) public {
        require(msg.sender == owner, "Only the contract owner can withdraw");
        payable(msg.sender).transfer(amount);
    }
}
```

Neste exemplo, a função `withdraw` usa `require` para verificar se `msg.sender` é o proprietário do contrato. Se `msg.sender` não for o proprietário do contrato, a transação será revertida e a mensagem de erro "Only the contract owner can withdraw" será retornada, `require` é uma ferramenta extremamente útil no kit de ferramentas do desenvolvedor Solidity. Ao usá-lo efetivamente, você pode garantir que seu contrato inteligente seja seguro e se comporte como esperado, mesmo diante de entradas ou condições inválidas.

5.5 Exemplo de um Contrato Inteligente para Marketplace

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

contract DigitalMarketplace {
    struct Product {
        uint id;
        address payable owner;
        uint price;
        bool purchased;
        string ipfsHash; // hash do IPFS do produto digital
    }

    mapping(uint => Product) public products;
    uint public productCount = 0;

    event ProductCreated(
        uint id,
        address payable owner,
        uint price,
        bool purchased,
        string ipfsHash
    );

    event ProductPurchased(
        uint id,
        address payable owner,
        address payable buyer,
        uint price,
        bool purchased,
        string ipfsHash
    );

    function createProduct(uint _price, string memory _ipfsHash) public {
        require(_price > 0, "Price must be greater than 0");
        productCount++;
        products[productCount] = Product(productCount, payable(msg.sender),
        _price, false, _ipfsHash);
```

```

        emit ProductCreated(productCount, payable(msg.sender), _price, false,
        _ipfsHash);
    }

    function purchaseProduct(uint _id) public payable {
        Product memory _product = products[_id];
        address payable _seller = _product.owner;
        require(_product.id > 0 && _product.id <= productCount, "Product does
not exist");
        require(msg.value >= _product.price, "Not enough Ether sent");
        require(!_product.purchased, "Product already purchased");
        require(_seller != msg.sender, "Cannot buy your own product");
        _product.owner = payable(msg.sender);
        _product.purchased = true;
        products[_id] = _product;
        _seller.transfer(msg.value);
        emit ProductPurchased(productCount, _seller, payable(msg.sender),
        _product.price, true, _product.ipfsHash);
    }
}

```

O contrato inteligente `DigitalMarketplace` é um mercado digital descentralizado, onde usuários podem criar e comprar produtos digitais. As principais partes do contrato são:

- **Estrutura de dados `Product`**: Esta estrutura define um produto no mercado. Cada produto tem um `id` único, um `owner` (dono), um `price` (preço), um status `purchased` (indicando se o produto foi comprado ou não), e um `ipfsHash` (o hash do arquivo do produto no IPFS).
- **Mapping `products`**: Este mapping mapeia um `uint` (que seria o ID do produto) para um `Product`. Ele é usado para armazenar todos os produtos no contrato.
- **Evento `ProductCreated` e `ProductPurchased`**: Esses eventos são emitidos quando um produto é criado ou comprado, respectivamente. Eles permitem que os clientes do contrato monitorem essas atividades.
- **Função `createProduct`**: Esta função permite a um usuário criar um produto. Ela recebe um `price` e um `ipfsHash`, cria um novo `Product`, e armazena-o no mapping `products`. Ela também emite um evento `ProductCreated`.

- **Função ``purchaseProduct``:** Esta função permite a um usuário comprar um produto existente. Ela recebe um ``_id`` do produto, verifica se o produto existe e ainda não foi comprado, se o usuário não está tentando comprar seu próprio produto, e se ele enviou Ether suficiente para cobrir o preço. Então, ela marca o produto como comprado, transfere o Ether ao vendedor, e emite um evento ``ProductPurchased``.

Este contrato fornece um exemplo completo de como um marketplace digital poderia funcionar na Ethereum, com todos os dados armazenados de forma transparente e imutável no blockchain, e todas as transações ocorrendo diretamente entre os usuários, sem intermediários.

Capítulo 6: Trabalhando com Ether em Solidity

Em contratos inteligentes Solidity, podemos manipular o Ether, a moeda nativa do Ethereum. Este capítulo se concentrará em como o Ether é usado e transferido em Solidity.

6.1 Unidades de Ether

Em Solidity, o Ether tem várias unidades, como wei, gwei e ether. 1 ether é igual a 1e18 wei ou 1e9 gwei. É importante se lembrar disso ao realizar cálculos com Ether.

6.2 Transferindo Ether

Você pode transferir Ether para um endereço com a instrução `transfer`.

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

contract SendEther {
    function send(address payable recipient) public payable {
        recipient.transfer(msg.value);
    }
}
```

Neste exemplo, a função `send` envia Ether para o `recipient`. A função é marcada como `payable`, permitindo que ela receba Ether.

6.3 Verificando Balanços

Você pode verificar o saldo de Ether de um endereço com `address.balance`.


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

contract CheckBalance {
    function getBalance(address addr) public view returns (uint) {
        return addr.balance;
    }
}
```

Neste exemplo, a função `getBalance` retorna o saldo de Ether do endereço fornecido.

6.4 FallBack e Receive Functions

A função fallback é chamada quando um contrato recebe Ether sem que nenhuma função seja chamada, ou quando nenhuma função corresponder à chamada. A função receive é chamada quando um contrato recebe Ether e nenhuma função é chamada.

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

contract FallbackAndReceive {
    fallback() external payable {}

    receive() external payable {}
}
```

Neste exemplo, as funções fallback e receive estão vazias e apenas aceitam Ether.

O uso seguro e eficaz do Ether é fundamental para a programação de contratos inteligentes no Ethereum. No próximo capítulo, aprenderemos a testar nossos contratos inteligentes para garantir que funcionem como esperado.

Capítulo 7: Testando seus Contratos Inteligentes

A testagem é uma parte crucial do desenvolvimento de contratos inteligentes. Testes adequados podem ajudar a garantir que seu contrato funcione como esperado e evitem erros caros e irreversíveis. Este capítulo abordará os testes de unidade para contratos inteligentes em Solidity.

7.1 Por que Testar?

Dada a natureza imutável dos contratos inteligentes, uma vez que eles são implantados na blockchain, não há como corrigir bugs ou falhas de segurança. Portanto, testar contratos inteligentes antes de implantá-los é crucial.

7.2 Ferramentas de Teste

Várias ferramentas podem ser usadas para testar contratos inteligentes em Solidity. No Truffle, os testes podem ser escritos em JavaScript ou Solidity. No Remix, você pode criar testes unitários para seus contratos em um ambiente de navegador.

7.3 Escrevendo Testes

Os testes geralmente envolvem a interação com seu contrato e a verificação de que o contrato se comporta conforme esperado. Por exemplo, um teste pode enviar uma transação, verificar o novo estado do contrato e compará-lo com o esperado.

Aqui está um exemplo de como um teste para um contrato de armazenamento simples pode parecer usando Truffle:

7.4 Executando o Teste no Truffle

```
const SimpleStorage = artifacts.require("SimpleStorage");

contract("SimpleStorage", accounts => {
  it("should store the value 89", async () => {
    const simpleStorageInstance = await SimpleStorage.deployed();

    // Set value of 89
    await simpleStorageInstance.set(89, { from: accounts[0] });

    // Get stored value
    const storedData = await simpleStorageInstance.get.call();

    assert.equal(storedData, 89, "The value 89 was not stored.");
  });
});
```

Para executar o código acima que visa testar um contrato inteligente, por exemplo, chamado SimpleStorage.sol você pode criar um novo arquivo chamado, por exemplo, `simpleStorage.test.js` e salvá-lo no diretório `test` do seu projeto. A estrutura básica do diretório `test` do projeto Truffle seria semelhante a isso:

```
project/
  contracts/
    SimpleStorage.sol
  migrations/
    ...
  test/
    simpleStorage.test.js
  truffle-config.js (ou truffle.js)
  ...
```

Certifique-se de substituir `"SimpleStorage"` na primeira linha do código pela correta nomenclatura do contrato com o qual você está trabalhando. Depois de armazenar o arquivo `simpleStorage.test.js` no diretório `test`, você poderá executar os testes com o comando `truffle test` no terminal, como mencionado anteriormente.

Lembre-se de que a estrutura do projeto pode variar dependendo da personalização que você fez ao criar seu projeto Truffle, mas o diretório `test` é geralmente usado para armazenar arquivos de teste de contratos inteligentes.

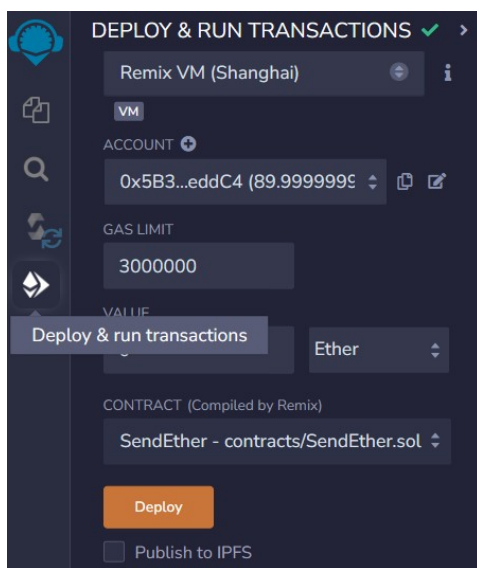
7.5 Testando no Remix IDE

Remix também fornece um ambiente para escrever testes Solidity para seus contratos. O Remix oferece uma interface gráfica para a execução de testes, tornando mais fácil a visualização e a compreensão dos resultados.

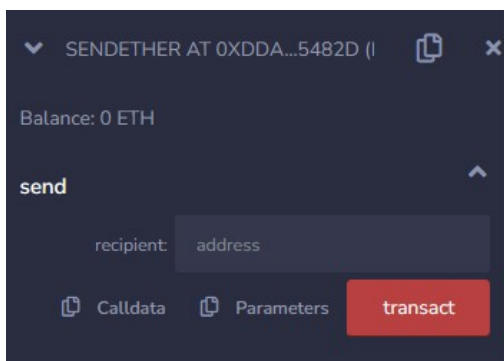
7.6 Executando o Teste no Remix IDE

O Remix IDE oferece uma maneira fácil e interativa de testar contratos inteligentes. Para executar o teste execute os seguintes passos:

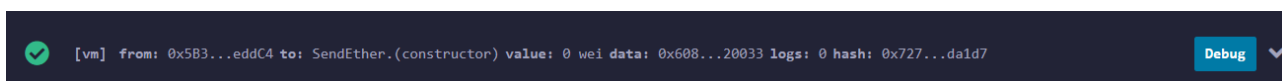
1. Na aba "Deploy & Run Transactions", localize o painel do contrato implantado.



2. Expanda a instância do contrato e localize a função que será testada, por exemplo, send.



3. Certifique-se de que o valor passado para a função que será testada seja o valor esperado para o teste.
4. Clique no botão "transact" ou "call" ao lado da função para executar o teste.
5. Aguarde até que a transação seja processada.
6. Após a execução da transação, verifique a saída no painel direito do Remix IDE. Você poderá ver o hash da transação e outros detalhes no painel direito do Remix IDE.



7. Verifique se a transação foi bem-sucedida, sem erros ou exceções.
8. Se o teste tiver assertivas, verifique se os resultados correspondem às expectativas definidas nas assertivas.

Capítulo 8: Publicando seus Contratos Inteligentes

Depois de escrever e testar seus contratos inteligentes, o próximo passo é publicá-los na rede MainNet da Ethereum. Este capítulo explicará como você pode fazer isso usando o Truffle ou o Remix IDE.

8.1 Publicando com o Truffle

Configuração da rede MainNet:

Primeiro, certifique-se de que o arquivo ``truffle-config.js`` (ou ``truffle.js``) e que esteja configurado com as informações de conta corretas para a rede MainNet. Veja abaixo um exemplo do arquivo ``truffle-config.js``.

```
const HDWalletProvider = require('@truffle/hdwallet-provider');
const infuraKey = 'YOUR_INFURA_PROJECT_ID';
const mnemonic = 'YOUR_METAMASK_MNEMONIC';

module.exports = {
  networks: {
    development: {
      host: '127.0.0.1',
      port: 8545,
      network_id: '*',
    },
    mainnet: {
      provider: () => new HDWalletProvider(mnemonic,
        `https://mainnet.infura.io/v3/${infuraKey}`),
      network_id: 1,
      gasPrice: 10000000000, // Defina o preço do gás de acordo com as condições
                              // atuais do mercado
      confirmations: 2, // Número de confirmações exigidas para considerar uma
                          // transação confirmada
      timeoutBlocks: 200, // Número máximo de blocos para aguardar a confirmação
                          // de uma transação
      skipDryRun: true, // Pule a verificação de secagem durante o deploy
    },
  },
  compilers: {
    solc: {
      version: '0.8.0',
      settings: {
        optimizer: {
          enabled: true,
          runs: 200,
        },
      },
    },
  },
};
```

Lembre-se de que você precisa substituir ``YOUR_INFURA_PROJECT_ID`` pelo seu ID do projeto Infura [<https://www.infura.io/>] e ``YOUR_METAMASK_MNEMONIC`` pela sua frase mnemônica do MetaMask. Esta configuração mostra uma rede de desenvolvimento local (``development``) e a rede MainNet da Ethereum (``mainnet``) usando o Infura como provedor.

Infura foi desenvolvido e é mantido pela ConsenSys, é uma plataforma de infraestrutura Blockchain-as-a-Service (BaaS) que oferece acesso confiável e escalável à Ethereum e à InterPlanetary File System (IPFS). Ele elimina a necessidade de um desenvolvedor ou organização executar e gerenciar nós completos ou outros recursos de infraestrutura, enquanto ainda permite a interação com a rede Ethereum.

Você pode adicionar mais redes personalizadas conforme necessário. Certifique-se de ajustar as configurações, como o preço do gás (``gasPrice``) e o número de confirmações necessárias (``confirmations``), com base nas condições atuais da rede MainNet. Certifique-se de configurar a rede MainNet corretamente e tenha uma conta com saldo de ether suficiente para cobrir as taxas de implantação do contrato.

Compilação do contrato:

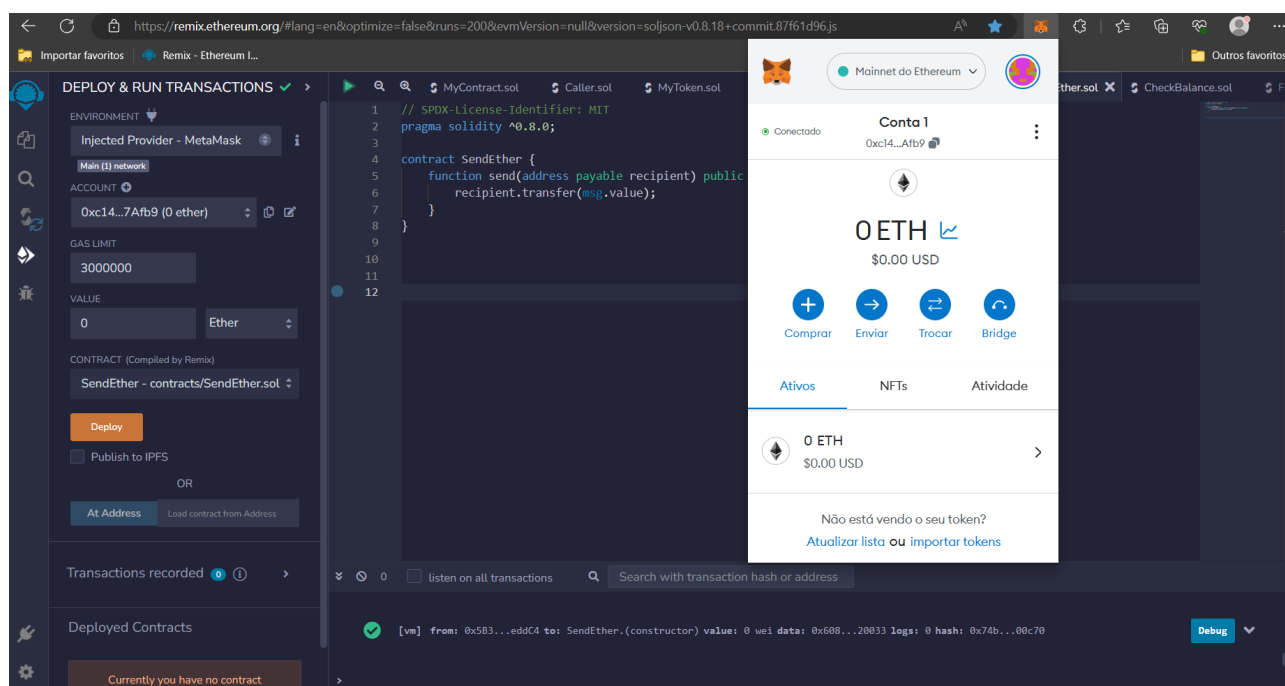
1. Execute o comando ``truffle compile`` no terminal para compilar o contrato inteligente.
2. Verifique se não há erros de compilação.

Implantação do contrato:

1. Execute o comando ``truffle migrate --network mainnet`` no terminal para implantar o contrato na rede MainNet.
2. Aguarde até que o processo de implantação seja concluído.
3. O Truffle gerenciará a interação com a rede MainNet usando o provedor de Web3 configurado anteriormente.

8.2 Publicando com o Remix IDE e Metamask

Primeiro, certifique-se de ter uma carteira Metamask esteja conectada ao Remix IDE e que esteja configurada com as informações de conta corretas para a rede MainNet. O MetaMask (<https://metamask.io/>) é uma carteira digital e uma extensão de navegador popular usada para interagir com aplicativos descentralizados (dApps) na rede Ethereum. Ele oferece uma maneira conveniente de gerenciar suas contas Ethereum, armazenar chaves privadas e assinar transações para interagir com contratos inteligentes.



Para usar o MetaMask com o Remix visando a publicação do contrato na rede Mainnet você deve seguir os passos abaixo:

1. Abra o navegador e acesse o site oficial do MetaMask em <https://metamask.io/>.
2. Siga as instruções para instalar a extensão do MetaMask no seu navegador. O MetaMask está disponível para Google Chrome, Firefox, Brave e Microsoft Edge.
3. Após a instalação, você verá o ícone do MetaMask na barra de extensões do navegador.

Configuração do MetaMask:

1. Clique no ícone do MetaMask na barra de extensões do navegador para abrir o pop-up do MetaMask.
2. Clique em "Get Started" para iniciar o processo de configuração.
3. Siga as instruções para criar uma carteira ou importar uma carteira existente usando a frase mnemônica ou o arquivo de chave privada.
4. Defina uma senha forte para proteger sua carteira.
5. Leia e aceite os termos de uso.
6. Selecione a rede Ethereum que deseja usar, como MainNet, Ropsten Testnet ou uma rede personalizada.

Conexão do MetaMask ao Remix IDE:

1. Abra o Remix IDE em <https://remix.ethereum.org/>.
2. No Remix IDE, clique no ícone "Deploy & Run Transactions" na barra lateral.
3. No painel "Environment" do Remix IDE, clique em "Injected Web3" para conectar o Remix ao MetaMask.
4. O MetaMask solicitará permissão para conectar o Remix IDE à sua conta Ethereum. Confirme a conexão.

Uso do MetaMask com o Remix IDE:

1. Após conectar o MetaMask ao Remix IDE, você poderá selecionar sua conta Ethereum no Remix IDE usando o seletor de contas.
2. Certifique-se de que você tenha saldo de ether suficiente em sua conta MetaMask para realizar as transações necessárias.
3. Quando implantar ou interagir com um contrato no Remix IDE, o MetaMask solicitará que você revise e confirme as transações, fornecendo a opção de ajustar as taxas de gás.
4. Uma vez confirmada a transação, o MetaMask solicitará que você assine a transação com sua senha do MetaMask.
5. Após a assinatura, o MetaMask enviará a transação para a rede Ethereum.

O MetaMask e o Remix IDE fornecem uma integração perfeita para o desenvolvimento e teste de contratos inteligentes. O MetaMask permite que você interaja com contratos implantados na rede Ethereum e assine transações diretamente do Remix IDE usando sua conta Ethereum.

Lembre-se de manter suas chaves privadas e frase mnemônica do MetaMask em segurança. O MetaMask é uma ferramenta poderosa para interagir com a blockchain Ethereum, mas também requer precauções adequadas para proteger seus ativos digitais.

Lembre-se de que a implantação de contratos na rede MainNet envolve transações reais e o uso de ether real. Certifique-se de ter uma estratégia adequada de gerenciamento de chaves privadas e de verificar todas as configurações antes de realizar transações na rede MainNet.

Além disso, é importante ressaltar que implantar um contrato inteligente na rede MainNet requer ether suficiente para cobrir as taxas de implantação e execução do contrato. Certifique-se de realizar testes e depuração adequados em ambientes de teste antes de implantar contratos na rede MainNet para garantir que tudo esteja funcionando corretamente.

Faucets

Antes de implantar um contrato inteligente na rede MainNet é recomendação implantar o contrato inteligente em uma rede de teste como a Goerli que envolve transações falsas e o uso de ether falso. Para obter ether falso será necessário acessar um faucet.

Faucets são serviços disponíveis na blockchain que fornecem ether falsos ou de teste para que os desenvolvedores possam realizar testes e experimentar funcionalidades sem gastar ether reais. Esses ethers falsos geralmente não têm valor monetário e são usados apenas para fins de desenvolvimento e aprendizado.

A ideia por trás dos faucets é fornecer uma maneira conveniente e rápida para os desenvolvedores obterem tokens em uma blockchain específica, sem a necessidade de comprá-los ou minerá-los. Isso permite que os desenvolvedores testem seus contratos inteligentes, dApps e outras funcionalidades sem incorrer em custos reais.

A obtenção de ethers falsos em faucets normalmente envolve alguns passos simples:

1. **Encontre um Faucet:** Existem vários faucets disponíveis para diferentes blockchains. Você pode procurar na web ou em fóruns especializados por faucets para a blockchain específica em que está interessado.

2. **Acesse o Faucet:** Acesse o site do faucet escolhido. Geralmente, você precisará fornecer o endereço da sua carteira ou um identificador específico para receber os tokens falsos.

3. **Solicite os Tokens:** Siga as instruções fornecidas pelo faucet para solicitar os tokens falsos. Isso pode envolver clicar em um botão para solicitar os tokens ou inserir informações adicionais, como um endereço de e-mail ou uma verificação de captcha.

4. **Aguarde a Distribuição:** Após fazer a solicitação, aguarde até que os tokens falsos sejam distribuídos para a sua carteira. O tempo necessário pode variar de acordo com o faucet e a blockchain.

5. **Use os Tokens para Testes:** Uma vez que os tokens falsos estejam em sua carteira, você pode usá-los para testar contratos inteligentes, realizar transações, experimentar funcionalidades de dApps e explorar outras interações na blockchain sem incorrer em custos reais.

Lembre-se de que os ethers obtidos em faucets são apenas para fins de teste e não possuem valor monetário real. Eles podem ser usados em redes de teste ou blockchains de desenvolvimento para ajudar os desenvolvedores a depurar e aprimorar seus projetos. É importante ressaltar que nem todas as blockchains possuem faucets disponíveis ou suportam tokens falsos para testes. Verifique a disponibilidade de faucets para a blockchain específica em que você está trabalhando.

Os faucets são ferramentas úteis para facilitar o desenvolvimento e os testes em ambientes blockchain sem gastar recursos reais. Eles permitem que os desenvolvedores experimentem e se familiarizem com as funcionalidades da blockchain, contribuindo para o crescimento e aprimoramento do ecossistema blockchain como um todo.

Capítulo 9: Próximos Passos em Solidity

Parabéns! Agora você tem um conhecimento básico, mas sólido, de programação Solidity. Você está equipado com as ferramentas necessárias para começar a construir seus próprios contratos inteligentes. Mas esta é apenas a ponta do iceberg, e há muitos tópicos avançados e conceitos a serem explorados. Este capítulo irá orientá-lo sobre os próximos passos em sua jornada de aprendizado Solidity.

9.1 Estudando Padrões de Contrato Inteligente

Padrões de contratos inteligentes são soluções reutilizáveis para problemas comuns em desenvolvimento de contratos inteligentes. Familiarizar-se com esses padrões pode ajudá-lo a escrever contratos mais seguros e eficientes.

9.2 Aprendendo sobre Segurança de Contratos Inteligentes

A segurança é extremamente importante ao desenvolver contratos inteligentes, pois um bug pode levar a perdas financeiras significativas. É essencial entender os tipos comuns de ataques a contratos inteligentes e como preveni-los.

9.3 Explorando Contratos DeFi e NFT

DeFi (Finanças Descentralizadas) e NFTs (Tokens Não Fungíveis) são dois dos usos mais populares de contratos inteligentes. Estudar contratos DeFi e NFT existentes pode fornecer uma visão valiosa sobre como os contratos inteligentes são usados no mundo real.

9.4 Contribuindo para Projetos de Código Aberto

Contribuir para projetos de código aberto é uma excelente maneira de melhorar suas habilidades de Solidity. Existem muitos projetos Ethereum que você pode contribuir, dependendo de seu interesse.

9.5 Participando da Comunidade Ethereum

Participar da comunidade Ethereum pode ser extremamente valioso. A comunidade é ativa e acolhedora, e você pode aprender muito com outros desenvolvedores Ethereum. Lembre-se de que se tornar proficiente em Solidity é uma jornada, não um destino. Continue aprendendo, continue construindo e continue se divertindo! No capítulo final, faremos uma revisão do que cobrimos neste livro e daremos uma breve despedida.

Capítulo 10: Conclusão

Chegamos ao fim de nossa jornada de introdução à programação em Solidity. Durante essa caminhada, abordamos desde a instalação de um ambiente de desenvolvimento adequado até a publicação de contratos inteligentes na blockchain Ethereum.

Passamos por uma série de conceitos fundamentais que servirão como base sólida para estudos mais aprofundados na linguagem Solidity. Desde tipos de variáveis, passando pelo controle de fluxo até os conceitos mais específicos da Ethereum, como endereços e transações.

Exploramos o universo dos contratos inteligentes, sua composição, funcionalidades e como eles representam a espinha dorsal de qualquer aplicação descentralizada. Discutimos as nuances de trabalhar com Ether dentro dos contratos e, ainda mais importante, destacamos a importância de testar seus contratos de maneira robusta antes da publicação.

Finalmente, enfatizamos a importância da segurança em todos os aspectos de seus contratos e oferecemos algumas orientações para os próximos passos que você pode seguir em sua jornada de aprendizado em Solidity.

Esperamos que este livro tenha ajudado a lançar uma base sólida para seus estudos em Solidity. A jornada para dominar qualquer linguagem de programação é repleta de contínuo aprendizado e prática. Portanto, incentive-se a experimentar, a construir, a errar e a aprender com esses erros.

Há uma comunidade ativa e vibrante em torno do Solidity e da Ethereum, então não hesite em procurar ajuda, colaborar e compartilhar seu conhecimento.

Obrigado por escolher este livro como seu guia inicial para a programação Solidity. Desejamos-lhe uma emocionante e recompensadora aventura no mundo do desenvolvimento de contratos inteligentes!

Boa sorte e feliz codificação!