**Capítulo 10: Ethereum na Prática**

Este capítulo introduz o ambiente de desenvolvimento do Ethereum. Vários exemplos serão apresentados neste capítulo para complementar os conceitos teóricos fornecidos no capítulo anterior.

Começaremos considerando o processo prático de pagamento no Ethereum, e algumas inovações recentes e futuras para o blockchain. Em seguida, examinaremos redes de teste, que podem desempenhar um papel vital ao testar contratos inteligentes antes de implantá-los na rede principal. Depois, examinaremos a opção de rede privada no Ethereum, que permite a criação de uma rede privada independente. Essa rede privada pode ser usada como um livro-razão distribuído compartilhado entre entidades participantes e para o desenvolvimento e teste de contratos inteligentes.

Embora existam outros clientes disponíveis para o Ethereum, o Geth é o principal cliente para Ethereum e a ferramenta padrão preferida, e, como tal, este capítulo utiliza o Geth para os exemplos. Também usaremos o Remix IDE para escrever contratos inteligentes e o MetaMask para implantar contratos em uma rede privada que criaremos.

Ao longo do caminho, consideraremos os seguintes tópicos:

• Pagamentos em Ethereum  
• Inovações no Ethereum  
• Programação com Geth  
• Configuração de um ambiente de desenvolvimento  
• Introdução ao Remix IDE  
• Interação com o blockchain Ethereum com MetaMask

**Pagamentos em Ethereum**

Nesta seção, veremos como o Ethereum funciona do ponto de vista do usuário. Para esse propósito, apresentaremos o caso de uso mais comum de transferência de fundos — em nosso caso de uso, de um usuário (Bashir) para outro (Irshad). Usaremos dois clientes Ethereum, um para enviar fundos e outro para receber. Existem várias etapas envolvidas neste processo, conforme segue:

1. Primeiro, ou um usuário solicita dinheiro enviando o pedido ao remetente, ou o remetente decide enviar dinheiro ao receptor. Podemos usar qualquer software de carteira Ethereum. A solicitação pode ser enviada enviando o endereço Ethereum do receptor ao remetente.

Por exemplo, há dois usuários, Bashir e Irshad. Se Irshad solicitar dinheiro a Bashir, então ela pode enviar um pedido a Bashir usando um código QR. Assim que Bashir receber essa solicitação, ele irá escanear o código QR ou digitar manualmente o endereço Ethereum de Irshad e enviar o Ether para o endereço de Irshad. Esse pedido é codificado como um código QR, mostrado na seguinte captura de tela, que pode ser compartilhado por e-mail, texto ou estar disponível para escaneamento visual ou qualquer outro método de comunicação:



*Figure 10.1: QR code as shown in a blockchain wallet application*

1. Assim que Bashir receber essa solicitação, ele irá escanear esse código QR ou copiar ou digitar o endereço Ethereum no software da carteira e iniciar uma transação. Imagine que Bashir quer enviar fundos para Irshad. O remetente insere tanto o valor quanto o endereço de destino em um software de carteira para enviar o Ether ao receptor. Pouco antes de enviar o Ether, a etapa final é confirmar a transação.
2. Assim que o pedido (transação) de envio de dinheiro é construído no software da carteira, ele é então transmitido para a rede Ethereum. A transação é assinada digitalmente pelo remetente como prova de que ele é o proprietário do Ether.
3. Essa transação é então capturada por nós chamados mineradores na rede Ethereum para verificação e inclusão no bloco. Neste estágio, a transação ainda está não confirmada.

O fluxo de transação funciona fundamentalmente da mesma forma em todos os softwares de carteira, portanto, qualquer software de carteira pode ser usado. Existem muitos tipos diferentes de softwares de carteira disponíveis online para sistemas operacionais iOS, Android e desktop.

1. Uma vez verificada e incluída no bloco, o processo de Prova de Trabalho (PoW) começa.
2. Uma vez que um minerador encontra a resposta para o problema de PoW ao fazer hash repetidamente do bloco com um novo nonce, esse bloco é imediatamente transmitido para os outros nós, que então verificam o bloco e o PoW.
3. Se todas as verificações forem aprovadas, então esse bloco é adicionado ao blockchain, e os mineradores são pagos com recompensas conforme apropriado.
4. Finalmente, Irshad recebe o Ether, que será exibido no software da carteira em uso.

Com esse exemplo, concluímos nossa discussão sobre o uso mais comum da rede Ethereum: transferir Ether de um usuário para outro.

**Inovações no Ethereum**

Nesta seção, abordaremos algumas das inovações no Ethereum. O Ethereum está em constante desenvolvimento. Assim como os BIPs (Bitcoin Improvement Proposals), temos as propostas de melhoria do Ethereum (EIPs) no Ethereum para sugerir e implementar melhorias na plataforma. Você pode acompanhar os EIPs aqui: <https://eips.ethereum.org>. Agora abordaremos algumas melhorias importantes.

**Bomba de dificuldade (difficulty time bomb)**

Além do ajuste de dificuldade baseado na diferença de timestamp, há também outro elemento que aumenta a dificuldade de mineração exponencialmente a cada 100.000 blocos. Esta é a chamada bomba de dificuldade, ou era do gelo, introduzida na rede Ethereum, que tornará muito difícil minerar no blockchain Ethereum em algum momento no futuro.

Uma vez ativada, ao longo do tempo, a bomba de dificuldade torna a mineração no Ethereum 1.x tão proibitivamente lenta que se torna inviável, resultando na chamada “era do gelo”. Em outras palavras, esse mecanismo aumenta exponencialmente a dificuldade de mineração por PoW até um nível em que a geração de blocos se torna impossível, forçando assim os mineradores a migrar para o sistema de Prova de Participação (PoS) do Ethereum chamado Casper.

A atualização London (<https://ethereum.org/en/history/#london>) introduziu o EIP-1559. A atualização mais recente é a Paris (A Fusão: <https://ethereum.org/en/history/#paris>), que discutiremos em mais detalhes no Capítulo 13, *The Merge and Beyond*.

Na blockchain, esta transação é identificada pelo seguinte hash de transação:  
0xc63dce6747e1640abd63ee63027c3352aed8cdb92b6a02ae25225666e171009e.  
Isso pode ser visualizado no explorador de blocos em:  
<https://etherscan.io/tx/0xc63dce6747e1640abd63ee63027c3352aed8cdb92b6a02ae25225666e171009e>

Mais informações sobre o Casper estão disponíveis aqui:  
<https://github.com/ethereum/research/blob/master/papers/casper-basics/casper_basics.pdf>

**EIP-1559**

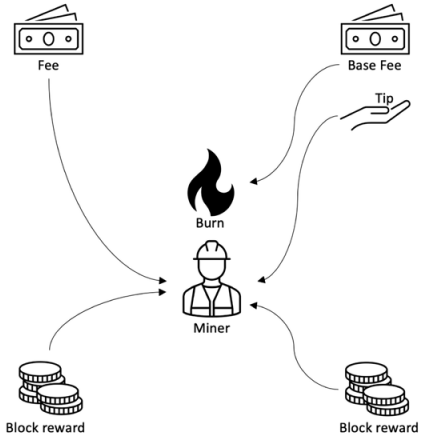
Uma maneira de mitigar, até certo ponto, as altas taxas de transação é o EIP-1559. No entanto, observe que o EIP-1559 não tem como objetivo reduzir a taxa de gas; ele apenas ajuda a prever melhor a taxa, o que pode significar que os usuários não acabarão pagando em excesso, resultando numa redução geral das taxas pagas ao longo do tempo.

O EIP-1559 foi implementado como parte da atualização London. O EIP-1559 impõe uma taxa base (por unidade de gas) que deve ser paga e tem como objetivo equilibrar oferta e demanda. Esta taxa é queimada permanentemente, o que pode diminuir a taxa de inflação do Ether. A taxa base se recalibra a cada bloco dependendo do “alvo”. Os blocos podem aumentar de tamanho alterando o MaxGasLimit, e se o bloco for maior que o alvo, então a taxa é mais alta, e mais baixa caso contrário. Os usuários também podem pagar uma gorjeta aos mineradores para priorizar suas transações, além da taxa base.

Com o aumento do tamanho do bloco, pode ser possível lançar um ataque de negação de serviço (DoS), onde a rede poderia ser sobrecarregada com blocos de tamanho grande sucessivos. O EIP-1559 torna o cálculo da taxa de gas mais previsível e melhora a experiência do usuário, o que pode resultar em os usuários não pagarem em excesso.

Observe que o EIP-1559 não é um mecanismo para reduzir a taxa de transação porque uma taxa alta é um problema devido a um problema de escalabilidade inerente, ou seja, um tamanho de bloco fixo, e, consequentemente, um mecanismo de oferta e demanda. Quando a rede está ocupada, um tamanho de bloco fixo resulta em os usuários aguardarem por períodos indefinidos para que sua transação seja incluída no bloco.

Soluções simples podem parecer possíveis para resolver esse problema, mas elas não são tão simples quanto parecem. Se pudermos, de alguma forma, impor uma taxa fixa e uma ordenação de transações universal impulsionada por consenso juntamente com tamanhos de bloco flexíveis, então esse problema pode ser resolvido. No entanto, se não houver mais incentivo para os mineradores devido às restrições de taxa fixa, então talvez os mineradores abandonem o blockchain.

****

*Figura 10.2: Mecanismo do EIP-1559, antes e depois da atualização London*

O EIP-1559 funciona com base em três variáveis:

* **baseFeePerGas**: Este é um novo campo introduzido na estrutura de bloco. Representa o preço mínimo que deve ser pago para que uma transação seja incluída em um bloco. Com o EIP-1559, os blocos podem aumentar ou diminuir de tamanho conforme a demanda da rede até o limite de 30 milhões de gas.

A taxa base é calculada com base nos blocos anteriores, o que torna as taxas de gas mais previsíveis para os usuários. O cálculo baseia-se em uma fórmula que compara o tamanho do bloco anterior com o tamanho-alvo. A taxa base aumenta ou diminui com base na diferença entre o tamanho atual do bloco e o alvo, que é 15 milhões de gas por bloco. Se um bloco estiver 100% cheio, a taxa base aumentará 12,5%; se o bloco estiver apenas 50% cheio, a taxa base permanecerá a mesma; e se o bloco estiver 0% cheio, a taxa base diminui 12,5%. Isso significa que a rede atinge o equilíbrio em 50% de capacidade ao calibrar as taxas de acordo com o nível de utilização da rede.

Essa taxa de crescimento exponencial torna economicamente inviável que o tamanho do bloco permaneça alto indefinidamente. As carteiras podem calcular a taxa base de forma determinística com base nas informações disponíveis nos blocos anteriores. Além disso, a taxa base é queimada para evitar um ataque em que mineradores poderiam conspirar para aumentar permanentemente a taxa base.

* **maxPriorityFeePerGas**: Esta é uma variável controlada pelos usuários. Os usuários a definem e a adicionam às transações. Ela representa a parte da taxa de transação que vai para o minerador. Esta é a gorjeta do minerador que permite que as transações sejam executadas mais rapidamente.
* **maxFeePerGas**: Este é outro parâmetro definido pelo usuário. Ele representa o valor máximo que um usuário está disposto a pagar por sua transação. Ele inclui baseFeePerGas e maxPriorityFeePerGas. A diferença entre maxFeePerGas e baseFeePerGas + maxPriorityFeePerGas é reembolsada ao usuário.

Em resumo, o EIP-1559 torna as taxas de transação mais previsíveis, reduz os atrasos na inclusão e confirmação das transações, melhora a experiência do usuário por meio de um mecanismo automatizado de taxa/lance e ajusta as taxas com base na atividade da rede, ou seja, oferta e demanda.

Uma maneira de pensar sobre isso é que o EIP-1559 substitui a volatilidade dos preços das taxas pela volatilidade no tamanho dos blocos, e, com o tempo, o tamanho dos blocos se estabiliza porque os usuários que não conseguem pagar uma gorjeta ao minerador para priorizar seu bloco desistirão, e eventualmente o equilíbrio de blocos com 50% do tamanho será mantido novamente.

Em resumo, antes da atualização London, o cálculo do gas era:

Unidades de gas (limite) \* Preço por unidade de gas

Após a atualização London, o cálculo do gas passou a ser:

Unidades de gas (limite) \* (Taxa base + Gorjeta)

**A Fusão e as Próximas Atualizações**

As próximas atualizações são a chamada “fusão” (*merge*) e as cadeias de fragmentos (*shard chains*). A fusão uniu a atual rede principal do Ethereum em 15 de setembro de 2022 com a cadeia Beacon. A cadeia Beacon é uma cadeia PoS (Proof of Stake) e atualmente está ativa, fornecendo a cadeia principal para os fragmentos e para a cadeia Ethereum fundida.

Observe que agora não existe mais o Ethereum 2.0, como originalmente planejado. *Eth2* ou *Ethereum 2.0* era um termo usado para uma nova versão distinta do Ethereum, mas agora foi substituído por *The Merge*. A atualização *The Merge* fundiu tanto o Ethereum1 quanto o que deveria ser o Ethereum 2.0, e uma única plataforma surgiu, chamada simplesmente de *Ethereum*, não *Ethereum 2.0*. Um modelo mental para esclarecer isso é que o Eth1 é agora a chamada camada de execução (*execution layer*), onde ocorre a execução das transações, e o Eth2 é a camada de consenso (*consensus layer*) responsável pela Prova de Participação (PoS). Veremos mais sobre isso em um capítulo separado sobre a Fusão do Ethereum e futuras atualizações no Capítulo 13, *The Merge and Beyond*.

Em seguida, discutiremos o procedimento de instalação e uso de alguns dos clientes Ethereum.

**Programando com Geth**

Primeiramente, descreveremos o Geth e exploraremos várias operações que podem ser realizadas com este cliente.

**Instalando e configurando o cliente Geth**

O procedimento de instalação detalhado em <https://geth.ethereum.org/docs/getting-started/installing-geth> descreve a instalação de clientes Ethereum no macOS e Linux. Instruções para outros sistemas operacionais também estão disponíveis.

Uma vez concluída a instalação, o Geth pode ser iniciado simplesmente emitindo o comando geth no terminal. Ele vem pré-configurado com todos os parâmetros necessários para se conectar à rede Ethereum ao vivo (mainnet):

$ geth

Quando o cliente Ethereum é iniciado, ele começa a se sincronizar com o restante da rede. Existem três tipos de mecanismos de sincronização disponíveis, a saber: *snap*, *full* e *light*:

* **Snap**: Este é o modo padrão que mantém os estados dos 128 blocos mais recentes na memória, permitindo acesso imediato às transações nesse intervalo. O nó neste modo também armazena pontos de verificação entre o bloco de sincronização inicial (um bloco recente) e os 128 blocos mais recentes, o que permite reconstruir estados. Este modo começa baixando os cabeçalhos de um conjunto de blocos. Uma vez que os cabeçalhos são verificados, o restante dos dados, como recibos e corpos dos blocos, é baixado. Ele também baixa os dados brutos do estado e constrói as árvores de estado em paralelo.
* **Full**: Neste modo de sincronização, o cliente Geth gera o estado mais recente executando e verificando todos os blocos desde o bloco gênese. Neste modo, apenas os estados dos 128 blocos mais recentes são armazenados. Os blocos mais antigos são descartados regularmente e representados com pontos de verificação que podem ser usados para regenerar o estado, se necessário. Existem também nós “full archive” que mantêm todos os dados desde o bloco gênese, e os dados antigos nunca são excluídos. Atualmente, o tamanho do blockchain Ethereum é muito grande para ser baixado e mantido sem dificuldades em hardware de entrada. Portanto, geralmente recomenda-se o uso de SSDs para um nó *full archive*, de modo que a latência de disco não cause atrasos no processamento.
* **Light**: Este é o modo mais rápido e baixa e armazena apenas a árvore de estado atual. Neste modo, o cliente não baixa nenhum bloco histórico e apenas processa blocos mais recentes.

O modo de sincronização é configurável no cliente Geth via o parâmetro:

--syncmode valor

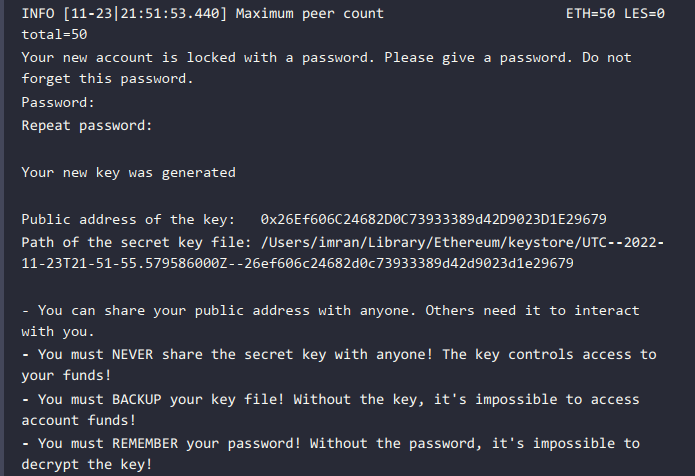
Aqui, o valor pode ser *snap*, *full* ou *light*, por exemplo, --syncmode light inicia o cliente Geth no modo de sincronização leve.

**Criando uma nova conta no Geth**

Novas contas podem ser criadas via linha de comando usando o Geth ou qualquer outra interface de linha de comando de cliente. Execute o seguinte comando para adicionar uma nova conta:

$ geth account new

Este comando produzirá uma saída semelhante à seguinte:

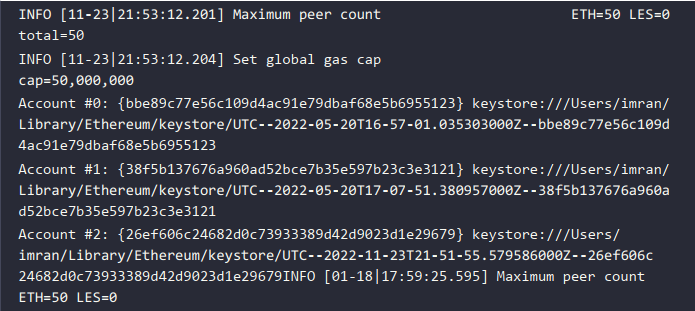


* Você pode compartilhar seu endereço público com qualquer pessoa. Outros precisam dele para interagir com você.
* Você NUNCA deve compartilhar a chave secreta com ninguém! A chave controla o acesso aos seus fundos!
* Você deve FAZER BACKUP do seu arquivo de chave! Sem a chave, é impossível acessar os fundos da conta!
* Você deve LEMBRAR sua senha! Sem a senha, é impossível descriptografar a chave!

A lista de contas pode ser exibida usando o cliente Geth emitindo o seguinte comando:

$ geth account list

Este comando produzirá uma saída semelhante à seguinte:



A seguir, veremos diferentes métodos para interagir com o blockchain.

**Consultando o blockchain usando o Geth**

Há diferentes métodos disponíveis para consulta ao blockchain. Primeiro, para conectar-se à instância em execução do cliente, pode-se usar uma API local IPC ou RPC.

Há três métodos para interagir com o blockchain usando o Geth:

* Console do Geth
* Geth attach
* Geth JSON-RPC

O console do Geth e o Geth attach são usados para interagir com o blockchain usando um ambiente JavaScript REPL. O JSON-RPC é um mecanismo de chamada de procedimento remoto que utiliza o formato de dados JSON para codificar suas chamadas. Em termos mais simples, é uma RPC codificada em JSON.

**Console do Geth**

O Geth pode ser iniciado no modo console executando o seguinte comando:

$ geth console

Isso iniciará o ambiente JavaScript interativo no qual comandos JavaScript podem ser executados para interagir com o blockchain Ethereum, por exemplo, para obter saldo, número de blocos e muitos outros comandos.

**Geth attach**

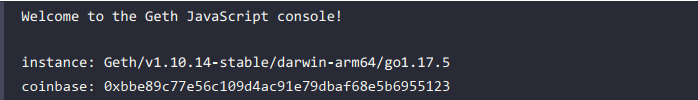
Quando um cliente Geth já estiver em execução, o console JavaScript interativo pode ser invocado conectando-se a essa instância. Isso é possível executando o comando geth attach.

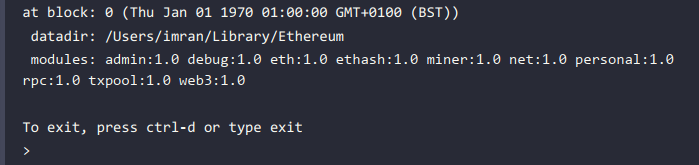
O console JavaScript do Geth pode ser usado para executar várias funções. Por exemplo, uma conta pode ser criada conectando o Geth.

O Geth pode ser conectado ao daemon em execução, como mostrado na captura de tela a seguir:

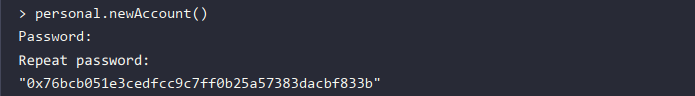
$ geth attach

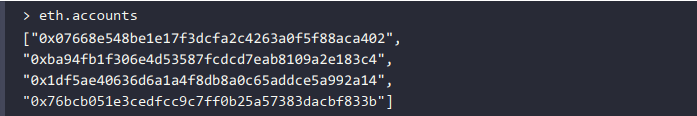
Este comando produzirá uma saída semelhante à seguinte:

Observe que você verá endereços e caminhos de diretórios diferentes ao executar isso em seu computador.

Assim que o Geth for conectado com sucesso à instância em execução do cliente Ethereum, ele exibirá o prompt de comando >, que fornece uma interface interativa de linha de comando para interagir com o cliente Ethereum usando notações JavaScript.

Por exemplo, uma nova conta pode ser adicionada usando o seguinte comando no console do Geth:

A lista de contas também pode ser exibida de forma semelhante:



**API JSON-RPC do Geth**

JSON significa *JavaScript Object Notation*. É um formato de texto leve e fácil de entender usado para transmissão e armazenamento de dados. Uma chamada de procedimento remoto (*remote procedure call* – RPC) é um conceito de sistemas distribuídos. É um mecanismo usado para invocar um procedimento em outro computador. Parece como se uma chamada local estivesse sendo feita porque é necessário escrever código para lidar com interações remotas.

Observe que os leitores verão um endereço diferente.

Mais detalhes sobre RPC e JSON podem ser encontrados aqui: <https://www.jsonrpc.org>

Para este capítulo, é suficiente saber que a API JSON-RPC é amplamente usada no Ethereum para permitir que usuários e *dApps* interajam com o blockchain.

Há vários métodos disponíveis para interagir com o blockchain. Um é usar o console do Geth, que faz uso da API Web3 para consultar o blockchain. A API Web3 faz uso da API JSON-RPC. Outro método é fazer chamadas JSON-RPC diretamente, sem usar a API Web3. Nesse método, chamadas RPC diretas podem ser feitas ao cliente Geth via HTTP. Por padrão, o RPC do Geth escuta na porta TCP 8545.

Agora, veremos alguns exemplos envolvendo o uso da API JSON-RPC. Usaremos uma ferramenta comum chamada curl (<https://curl.haxx.se>) para esse propósito.

Para que esses exemplos funcionem, primeiro o cliente Geth precisa ser iniciado com os parâmetros apropriados. Se houver uma sessão existente do Geth em execução com outros parâmetros, interrompa essa instância e execute o comando geth, como mostrado abaixo, que iniciará o Geth com o RPC disponível. O usuário também pode controlar quais APIs são expostas:

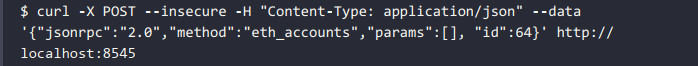
$ geth --http --http.api "eth,net,web3,personal"

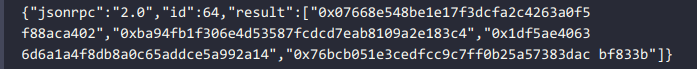
Neste comando, o Geth é iniciado com as flags --rpc e --rpcapi, junto com uma lista de APIs que são expostas:

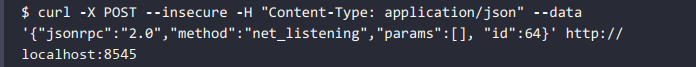
* A flag --rpc ativa o servidor HTTP-RPC.
* A flag --rpcapi é usada para definir quais APIs estarão disponíveis pela interface HTTP-RPC.  
  Isso inclui várias APIs como eth, net, web3 e personal.

Para cada um dos exemplos a seguir, execute o comando curl no terminal, conforme mostrado em cada um dos exemplos abaixo.

A lista de contas pode ser obtida emitindo o seguinte comando:

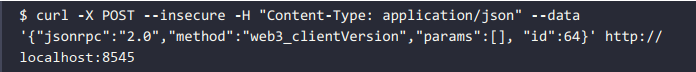
Isso exibirá a seguinte saída JSON, que lista todas as contas Ethereum possuídas pelo cliente:

Podemos verificar se a rede está ativa usando o comando mostrado aqui:

Isso exibirá a seguinte saída com o resultado true, indicando que a rede está ativa:

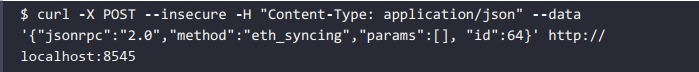
{"jsonrpc":"2.0","id":64,"result":true}

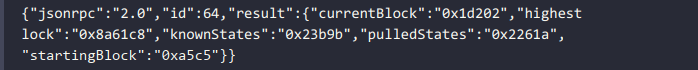
Podemos encontrar a versão do cliente Geth usando este comando:

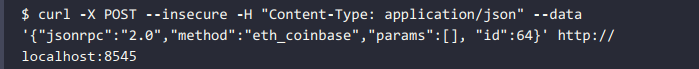
Isso exibirá a versão do cliente Geth:

{"jsonrpc":"2.0","id":64,"result":"Geth/v1.9.9-stable-01744997/linux-amd64/go1.13.4"}

Para verificar o status mais recente de sincronização, podemos usar o seguinte comando:

Isso exibirá os dados relativos ao status de sincronização ou retornará false:

O endereço coinbase pode ser consultado usando:

Isso exibirá a saída conforme mostrado abaixo, indicando o endereço coinbase do cliente:

{"jsonrpc":"2.0","id":64,"result":"0x07668e548be1e17f3dcfa2c4263a0f5f88aca402"}

Esses são apenas alguns exemplos das APIs extremamente ricas que estão disponíveis no cliente Geth do Ethereum.

Nesta seção, cobrimos as APIs JSON-RPC do Geth e vimos alguns exemplos de como consultar o blockchain via interface RPC. Com isso, completamos a introdução ao Geth.

**Configurando um ambiente de desenvolvimento**

Uma abordagem comum e sensata para desenvolver e testar contratos inteligentes do Ethereum é dentro de uma rede privada local ou um ambiente simulado como o Ganache. Depois que todos os testes relevantes forem bem-sucedidos em uma rede de teste pública, os contratos podem então ser implantados na rede principal pública. Existem, no entanto, variações nesse processo.

Mais informações e documentos oficiais sobre as APIs RPC do Geth estão disponíveis no seguinte link:  
<https://eth.wiki/json-rpc/API>

Muitos desenvolvedores optam por desenvolver e testar contratos apenas em um ambiente simulado local e, em seguida, implantá-los diretamente na rede principal pública ou em suas redes privadas/empresariais de produção. Desenvolver primeiro em um ambiente simulado e depois implantar diretamente em uma rede pública pode levar a um tempo de produção mais rápido, já que configurar redes privadas pode demorar mais do que configurar um ambiente local de desenvolvimento com um simulador de blockchain.

Existem novas ferramentas e frameworks disponíveis, como o **Truffle** e o **Ganache**, que tornam o desenvolvimento e teste para o Ethereum mais fáceis. Exploraremos essas ferramentas com mais profundidade no Capítulo 11, *Tools, Languages, and Frameworks for Ethereum Developers*, mas primeiro, usaremos uma abordagem manual na qual desenvolveremos um contrato inteligente e o implantaremos manualmente via linha de comando em uma rede privada. Frameworks e ferramentas tornam o desenvolvimento mais fácil, mas ocultam a maioria dos detalhes internos que são úteis para iniciantes construírem uma base sólida de conhecimento.

Vamos começar conectando-nos a uma rede de teste.

**Conectando-se a redes de teste**

O cliente Go do Ethereum (<https://geth.ethereum.org>), Geth, pode ser conectado a uma rede de teste usando uma das seguintes opções de linha de comando disponíveis com o Geth. O Geth pode ser executado com uma flag para conectar-se à rede desejada:

* --goerli: Rede Goerli — rede de teste pré-configurada com prova de autoridade
* --rinkeby: Rede Rinkeby — rede de teste pré-configurada com prova de autoridade
* --ropsten: Rede Ropsten — rede de teste pré-configurada com prova de trabalho
* --sepolia: Rede Sepolia — rede de teste pré-configurada com prova de trabalho

Este comando conecta-se à rede Sepolia:

$ geth –sepolia

Um explorador de blocos para a rede de teste Sepolia está localizado em:  
<https://sepolia.etherscan.io>  
e pode ser usado para rastrear transações e blocos na rede de teste Ethereum.

Agora vamos experimentar a construção de uma rede privada e, em seguida, veremos como um contrato pode ser implantado nesta rede usando ferramentas de linha de comando.

**Criando uma rede privada**

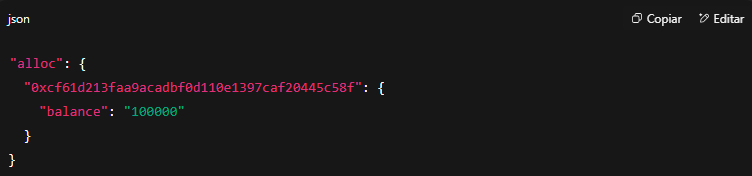
Nesta seção, iniciaremos uma rede privada e a prepararemos para uso. Utilizaremos os seguintes componentes, conforme especificado no capítulo anterior:

* **ID da Rede**: O ID da rede 786 foi escolhido para a rede privada de exemplo.
* **Arquivo Gênese**: Um arquivo gênese personalizado que será usado está mostrado aqui:



Este arquivo é salvo como um arquivo de texto com a extensão .json, por exemplo, privategenesis.json. Opcionalmente, o Ether pode ser pré-alocado especificando os endereços dos beneficiários e a quantidade de Wei, mas isso geralmente não é necessário, pois estando em uma rede privada, o Ether pode ser minerado muito rapidamente.

Para pré-alocar uma conta com Ether, uma seção pode ser adicionada ao arquivo gênese, como mostrado aqui:



* **Diretório de dados**: No exemplo a seguir, é ~/etherprivate/.

Além dos três componentes mencionados, é desejável desativar a descoberta de nós para que outros nós na internet não descubram sua rede privada e para que ela seja segura. Isso pode ser feito executando o Geth com a flag --nodiscover, que desativa o mecanismo de descoberta de pares. Se outras redes tiverem o mesmo arquivo gênese e ID de rede, elas podem se conectar à sua rede privada, o que pode resultar em problemas de segurança. A chance de haver o mesmo ID de rede e bloco gênese é muito baixa, mas mesmo assim, desativar a descoberta de nós é uma boa prática e é recomendado.

A conexão de outra rede ou nó à sua rede privada pode resultar em violações de segurança, vazamento de informações e outros incidentes indesejáveis. No entanto, observe que redes privadas geralmente são executadas dentro de ambientes empresariais e são protegidas por práticas de segurança padrão da empresa, como firewalls.

Desabilitar a descoberta de pares também nos permite definir uma lista de pares estáticos que temos em nossa rede. Isso nos dá a habilidade adicional de controlar quem pode entrar em nossa rede privada.

Para configurar essa lista, os IDs dos nós são adicionados a um arquivo de configuração chamado static-nodes.json. Este arquivo geralmente é colocado no diretório de dados do Geth (cliente Ethereum) executável. Esse diretório também é onde os arquivos chaindata (banco de dados) e keystore são salvos. Por padrão, o diretório de dados está localizado em <diretório pessoal do usuário>/Library/Ethereum, mas pode ser configurado usando a flag --datadir.

O nome do arquivo deve ser static-nodes.json, sob o diretório de dados. Isso é valioso em uma rede privada porque dessa forma, a rede é limitada apenas a nós conhecidos. Um exemplo de arquivo static-nodes.json é mostrado a seguir:

[

"enode://44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3be2787375e932fb4885885f6452f6efa77f@xxx.xxx.xxx.xxx:TCP\_PORT"

]

Aqui, xxx é o endereço IP e TCP\_PORT pode ser qualquer porta TCP válida e disponível no sistema. A longa string hexadecimal é o ID do nó.

Agora que entendemos os vários aspectos e componentes necessários para configurar uma rede privada, incluindo o arquivo gênese e outros arquivos de configuração relevantes, vamos prosseguir com a configuração da nossa própria rede privada usando Ethereum.

**Inicializando a rede privada**

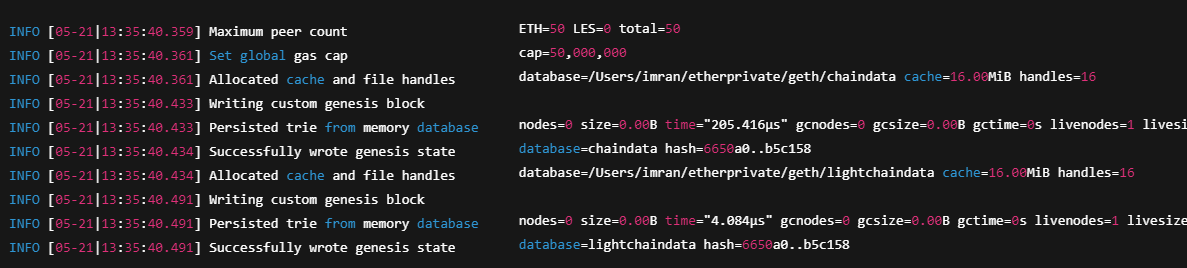
A primeira etapa é criar um diretório chamado etherprivate no diretório home do usuário:

$ mkdir ~/etherprivate

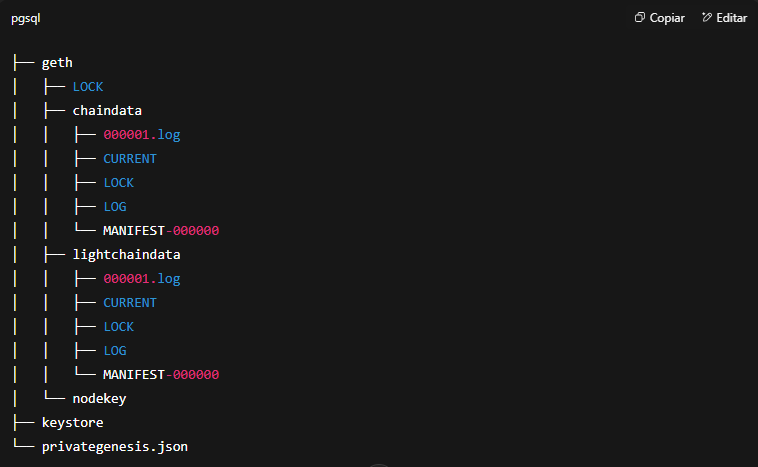
Esse comando criará o diretório. Depois que o diretório for criado, coloque o arquivo privategenesis.json mostrado anteriormente dentro dele. Neste ponto, armazenado sob o diretório pessoal do usuário, temos um diretório chamado ~/etherprivate, que contém o arquivo gênese chamado privategenesis.json. Estamos prontos para iniciar nossa rede. O comando inicial para iniciar a rede privada é o seguinte:

$ geth init ~/etherprivate/privategenesis.json --datadir ~/etherprivate

Isso produzirá uma saída semelhante à seguinte:



Se você vir a mensagem Successfully wrote genesis state, então tudo está certo e você deve ver uma estrutura de diretório como a mostrada abaixo:

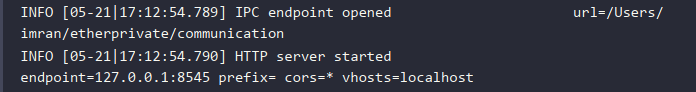


Essa estrutura de diretórios contém os dados da cadeia (*chaindata*), logs e um *keystore*. Para que o Geth inicie, o seguinte comando pode ser emitido:

**Iniciando a rede privada**

Para que o Geth seja iniciado, o seguinte comando pode ser emitido:

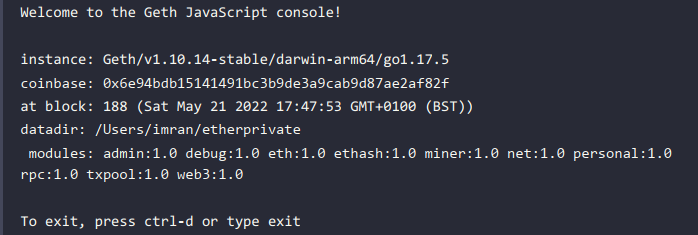
Isso produzirá várias mensagens de saída conforme o Geth é iniciado. Uma parte importante do log a ser observada são as seguintes linhas:

Essas linhas mostram informações sobre o endpoint de Comunicação entre Processos (*Inter-Process Communication – IPC*), o endpoint HTTP e as informações da conta *coinbase* (base de Ether). Essas informações são úteis para os exemplos fornecidos posteriormente nesta seção.

Agora o Geth pode ser conectado via IPC à instância do cliente Geth em execução na rede privada usando o seguinte comando:

$ geth attach ~/etherprivate/geth.ipc

Esse comando abrirá o console interativo JavaScript para executar a sessão da rede privada:

Você pode notar que uma mensagem de aviso aparece quando o Geth é iniciado:

AVISO: Nenhuma conta etherbase definida e nenhuma conta encontrada como padrão.

Essa mensagem aparece porque atualmente não há contas disponíveis na nova rede de teste e nenhuma conta está definida como *etherbase* para receber recompensas de mineração. Esse problema pode ser resolvido criando uma nova conta e definindo essa conta como *etherbase*. Isso também será necessário quando a mineração for realizada na rede de teste.

**Mecanismos IPC**

Mecanismos de Comunicação entre Processos (IPC) são usados para permitir a comunicação entre diferentes processos ou threads executando localmente em um sistema operacional. Existem diferentes mecanismos de IPC, incluindo *pipes* (tubulações) e *signals* (sinais). Um *pipe* é um buffer limitado entre processos realizado, na prática, por um descritor de arquivo compartilhado entre os processos. Um *signal* é um método usado para chamar a atenção de um processo. O Geth usa *pipes* para habilitar a comunicação entre instâncias do Geth.

Isso é mostrado nos comandos a seguir. Observe que esses comandos são inseridos no console JavaScript do Geth. O comando a seguir cria uma nova conta. Nesse contexto, a conta será criada na rede privada com ID 786, pois esta é a rede que criamos anteriormente:

> personal.newAccount("Password123")

"0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f"

Depois que a conta é criada, a próxima etapa é defini-la como a conta *etherbase*/*coinbase*, para que a recompensa de mineração vá para essa conta. Isso pode ser feito usando o seguinte comando:

> miner.setEtherbase(personal.listAccounts[0])

True

Atualmente, a conta *etherbase* não tem saldo, como pode ser visto usando o comando:

> eth.getBalance(eth.coinbase).toNumber();

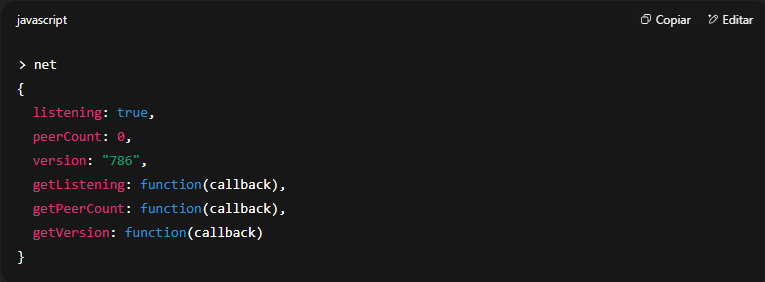
0

Nesta seção, criamos uma rede privada com um arquivo gênese personalizado. Também criamos uma nova conta em nossa rede privada. Agora, antes de explorar mais métodos, vamos entender alguns fundamentos do uso do console JavaScript.

**Experimentando o console JavaScript do Geth**

No console JavaScript, podemos realizar várias operações. Uma dica geral é que, se pressionarmos duas vezes a tecla Tab no teclado, uma lista completa dos objetos disponíveis será exibida.

Além disso, quando um comando é digitado, ele pode ser autocompletado pressionando a tecla Tab duas vezes. Se a tecla Tab for pressionada duas vezes, então a lista de métodos disponíveis também será exibida. Além do comando mencionado, para obter uma lista de métodos disponíveis de um objeto, basta pressionar Enter. Um exemplo é mostrado abaixo, que mostra uma lista de todos os métodos disponíveis para o objeto net:



Também está disponível o objeto eth, que possui vários métodos. Embora existam muitos métodos nesse objeto, o mais comum é o getBalance, que podemos usar para consultar o saldo atual de Ether. Isso é mostrado no exemplo a seguir:

> eth.getBalance(eth.coinbase)

150000000000000000000

Após a mineração, uma quantidade significativa pode ser vista aqui. A mineração é extremamente rápida, pois trata-se de uma rede privada sem competição para resolver o PoW, e também no arquivo gênese, a dificuldade da rede foi configurada como muito baixa.

O saldo acima é mostrado em Wei. Se quisermos ver a saída em Ether, podemos usar o objeto web3, como mostrado aqui:

> web3.fromWei(eth.getBalance(eth.coinbase), "ether")

150

Existem alguns outros comandos que podem ser usados para consultar a rede privada. Alguns exemplos são mostrados a seguir:

**Obter o preço atual do gas:**

> eth.gasPrice

1000000000

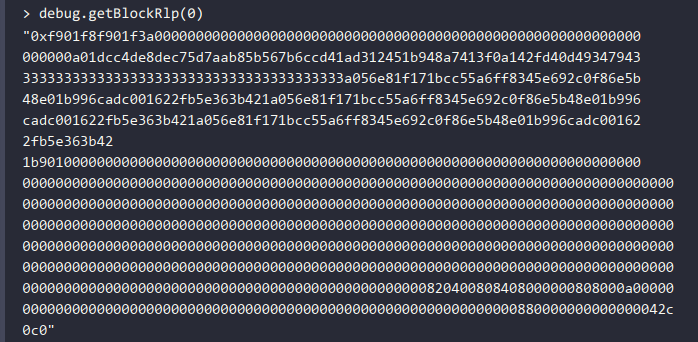
Obter o número mais recente de bloco:

> eth.blockNumber

30

**O objeto debug pode ser útil ao depurar problemas.** Um comando de exemplo é mostrado aqui; no entanto, há muitos outros métodos disponíveis. Uma lista desses métodos pode ser visualizada digitando debug.

O método a seguir retornará o RLP do bloco 0:

A saída acima mostra o bloco 0 no formato codificado RLP (Recursive Length Prefix).

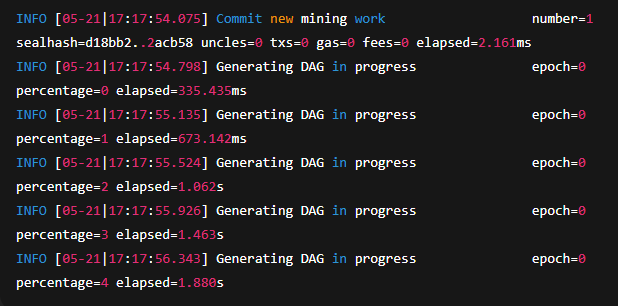
Agora estamos prontos para iniciar a mineração em nossa rede privada.

**Mineração e envio de transações**  
Agora que iniciamos nossa rede privada, a mineração pode começar simplesmente emitindo o seguinte comando. Este comando recebe um parâmetro: o número de threads. No exemplo a seguir, dois threads serão alocados para o processo de mineração especificando 2 como argumento para a função start:

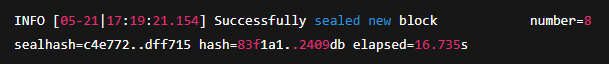
> miner.start(2)

Aqui também podemos fornecer um parâmetro inteiro. Por exemplo, se fornecermos 1, ele usará apenas um núcleo de CPU para mineração, o que ajuda com problemas de desempenho, caso usar todos os recursos da CPU esteja reduzindo o desempenho do sistema. Um comando de exemplo usando apenas uma CPU é miner.start(1). Em sistemas onde há apenas uma CPU, emitir o comando anterior inevitavelmente usará apenas uma CPU. No entanto, em um sistema multicore, fornecer o número de núcleos que podem ser usados para mineração ajuda a lidar com qualquer preocupação de desempenho, pois vários threads de mineração podem ser executados em paralelo em vários núcleos.

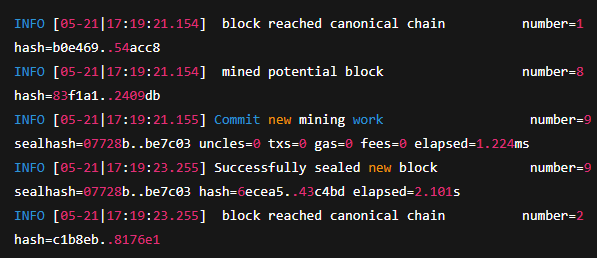
Após a emissão do comando anterior como preparação para a mineração, o processo de geração do DAG começa, o que produz uma saída semelhante à mostrada aqui:



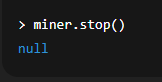
Depois que a geração do DAG é concluída, o processo de mineração começa e blocos são produzidos. O Geth produzirá uma saída semelhante à mostrada a seguir. Pode-se ver que os blocos estão sendo minerados com sucesso com a mensagem de bloco potencial minerado:



DAG significa *Directed Acyclic Graph* (grafo acíclico dirigido). No contexto do algoritmo de PoW Ethash do Ethereum, DAG refere-se ao Dagger, que é um algoritmo de PoW resistente a ASICs e com uso intensivo de memória, baseado em grafos acíclicos dirigidos moderadamente conectados. O objetivo do algoritmo Dagger é fornecer um algoritmo de PoW resistente a ASICs e com uso intensivo de memória. O algoritmo Dagger funciona gerando um DAG de alguns gigabytes a cada 30.000 blocos. Este DAG serve como recurso para o PoW, onde o mecanismo de PoW precisa escolher subconjuntos do DAG, o que depende do *nonce* e do cabeçalho do bloco.

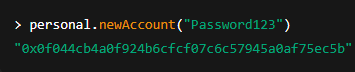


A mineração pode ser parada usando o seguinte comando:

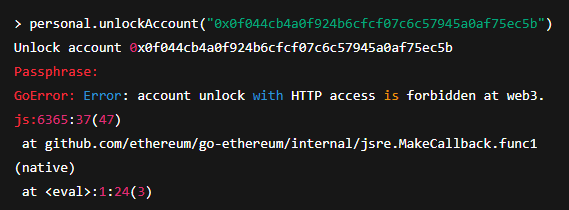


Agora vamos ver como criar uma nova conta e enviar uma transação em nossa rede privada.

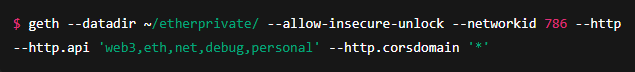
* Criar uma nova conta. Observe que Password123 é a senha escolhida como exemplo, mas você pode escolher qualquer uma:



Desbloqueie a conta antes de enviar transações. Usamos desbloqueio inseguro permitido; caso contrário, as contas não podem ser desbloqueadas com acesso HTTP. Se for esse o caso, você verá uma mensagem de erro conforme a seguir:



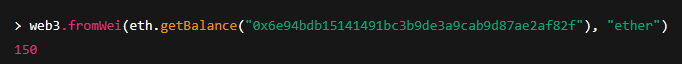
Para contornar o erro, reinicie o Geth usando a flag --allow-insecure-unlock:



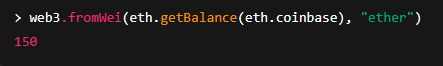
Agora desbloqueamos ambas as contas que criamos anteriormente com o comando personal.newAccount() no início da rede privada que estamos criando, e logo antes do comando unlockAccount() nesta página.

A primeira conta que criamos no início da nossa rede privada é 0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f, e outra, 0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b, está logo no início desta página:

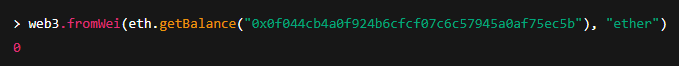
Agora, uma vez que essas contas estão desbloqueadas, podemos emitir alguns comandos adicionais para consultar os saldos que elas possuem. Primeiro, vamos verificar o saldo da nossa conta que criamos anteriormente quando iniciamos nossa rede privada pela primeira vez:



Como essa conta também é nossa conta *coinbase*, que é a conta padrão que recebe a recompensa da mineração; também podemos consultar o saldo de forma ligeiramente diferente, especificando eth.coinbase no comando, conforme mostrado aqui:

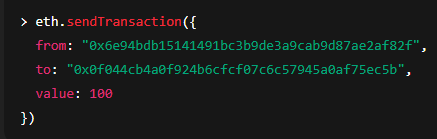


Por fim, verificamos o saldo da outra conta que criamos. Como ela não recebeu nenhuma recompensa de mineração, o saldo é 0 como esperado:



Agora vamos tentar enviar transações de uma conta para outra. Neste exemplo, enviaremos Ether da conta 0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f para 0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b usando o comando sendTransaction, conforme mostrado a seguir.

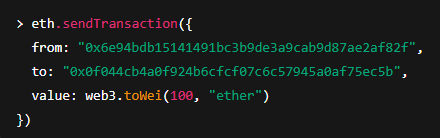
Primeiro, enviamos um valor de 100:



Este comando gera o *hash* da transação, que é um identificador único usado para identificar uma transação específica. A saída é a seguinte:

"0xba427f7c5590530d75375e3670b856200c684141f9c787bf8e1bf622305880d9"

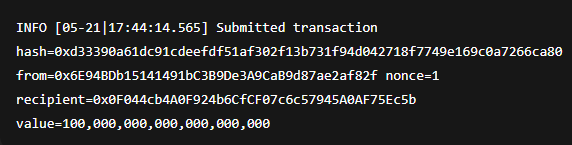
Esse comando transferirá 100 Wei (a menor unidade de Ether) para a conta de destino. Para transferir o valor em Ether, 100 ETH em nosso exemplo, podemos usar o comando de forma ligeiramente diferente e utilizar web3.toWei, que converterá o valor de Ether para Wei. Para isso, emitimos o comando como mostrado aqui:



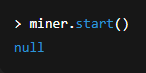
Este comando gera o hash da transação, como mostrado aqui:

"0xd33390a61dc91cdeefdf51af302f13b731f94d042718f7749e169c0a7266ca80"

O comando anterior utiliza web3.toWei, que recebe dois parâmetros: value, que é 100 em nosso exemplo, e uma *string*, "ether", que é a unidade de valor. Isso significa que 100 ETH serão convertidos para seu equivalente em Wei e usados no comando sendTransaction. O resultado desse comando será a transferência de 100 ETH da conta de origem para a conta de destino. Também observe a mensagem de log no terminal onde executamos o comando Geth anteriormente no início da rede privada:



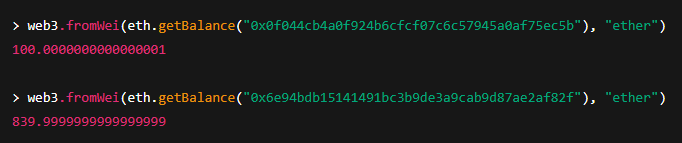
Essa mensagem mostra que a transação foi enviada, mas precisa ser minerada primeiro para que a transação tenha efeito. Como paramos a mineração anteriormente, para que essa transação seja processada, precisamos reiniciar a mineração; caso contrário, a transação não será processada:



Agora verifique nos logs que a mineração começou a funcionar:



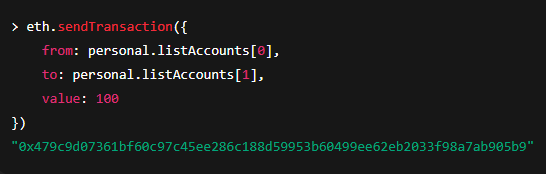
Observe que a quantidade de fundos foi transferida:



Lembre-se de que, à medida que a mineração continua, você verá números ligeiramente diferentes e mais Ether na conta de origem (*coinbase*), e a conta de destino agora terá 100 ETH, que transferimos.

Pode ser um pouco trabalhoso digitar todos esses IDs de conta. Em vez de digitar esses longos IDs de conta, também podemos usar o método listAccounts[], que recebe um parâmetro inteiro para referenciar a conta — por exemplo, 0, que representa a primeira conta que criamos.

Agora, vejamos um exemplo de uso do método listAccounts[], conforme mostrado aqui:



Agora também podemos consultar informações sobre a transação que executamos anteriormente. Lembra como ela retornou um hash de transação? Podemos usá-lo para descobrir detalhes sobre a transação: 

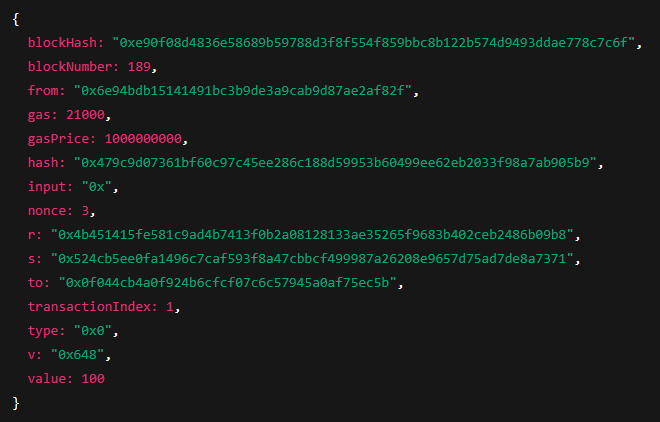
Isso produzirá a seguinte saída:



Observe o campo root na saída anterior — a raiz da transação — que estará disponível no cabeçalho do bloco como a raiz de Merkle do trie de transações. Da mesma forma, podemos consultar mais informações sobre a transação usando o método getTransaction:



Isso produzirá a seguinte saída:



Se você não vir o recibo da transação, certifique-se de que a mineração está ativa; emita miner.start() para iniciar a mineração. O bloco que contém essa transação precisa ter sido minerado para que os recibos sejam produzidos.

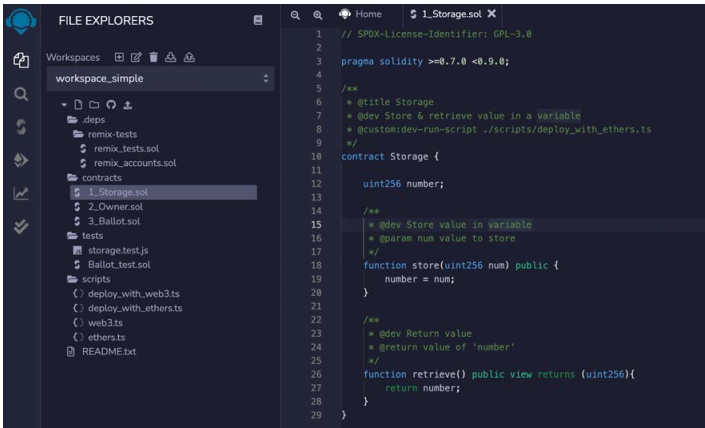
**Apresentando a Remix IDE**

Existem vários Ambientes de Desenvolvimento Integrados (*IDEs*) disponíveis para o desenvolvimento em Solidity. A maioria dos IDEs está disponível online e é apresentada por meio de interfaces web. O Remix (anteriormente Browser-Solidity) é o IDE mais comumente utilizado para construir e depurar contratos inteligentes. Ele é discutido aqui.

Remix é um ambiente baseado na web para o desenvolvimento e teste de contratos usando Solidity. É um IDE com muitos recursos que **não** roda em uma blockchain ao vivo; na verdade, é um ambiente simulado no qual contratos podem ser implantados, testados e depurados. Está disponível em:

<https://remix.ethereum.org>

Um exemplo da interface é mostrado a seguir:



**Figura 10.3: Remix IDE**

No lado esquerdo, há uma coluna com diferentes ícones. Esses ícones representam vários plugins do Remix IDE. Quando você executa o Remix pela primeira vez, nenhum plugin é exibido. Para adicionar plugins ao Remix IDE, é necessário acessar o gerenciador de plugins para ativar os plugins de que você precisa.

Há vários plugins disponíveis, incluindo, entre outros, um provedor Ganache, análise estática de Solidity e um implantador de contratos, apenas para citar alguns; há muitos outros. Eles podem ser acessados clicando no ícone de plugin no canto inferior esquerdo do IDE.

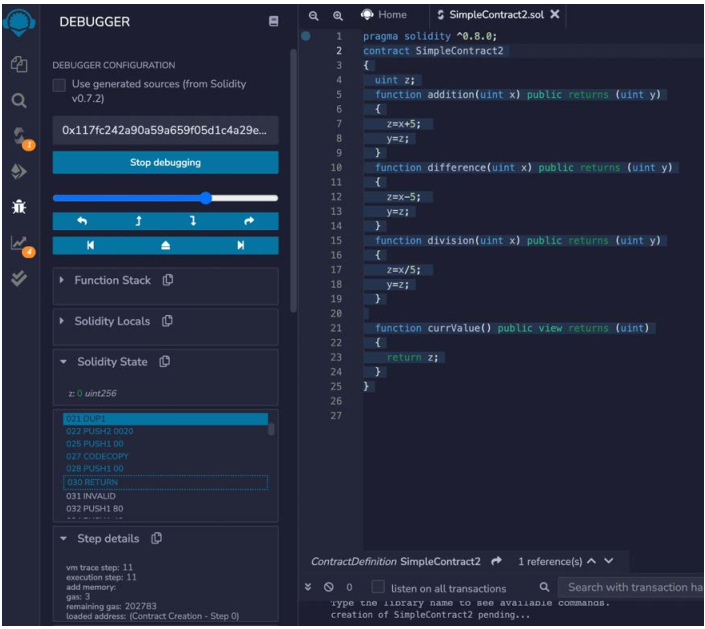
Uma vez ativados, os plugins aparecerão na coluna mais à esquerda do IDE, conforme mostrado na imagem a seguir. Novamente, observe que isso só mostrará os plugins ativados, e outros plugins ou plugins locais podem ser ativados conforme necessário.

No lado direito, há um editor de código com destaque de sintaxe e formatação de código, e no lado esquerdo há vários plugins disponíveis que podem ser usados para implantar, depurar, testar e interagir com o contrato.

Vários recursos, como interação com transações, opções para conectar-se à VM JavaScript, configuração de um ambiente de execução, um depurador, verificação formal e análise estática estão disponíveis. Eles podem ser configurados para conectar-se a ambientes de execução como:

* a **JavaScript VM**,
* **Web3 injetado** – onde Mist, MetaMask ou um ambiente similar fornece o ambiente de execução –, ou
* o **provedor Web3**, que permite conexão com o cliente Ethereum local em execução (por exemplo, Geth) via IPC ou RPC sobre HTTP (um endpoint de provedor Web3).

O Remix também possui um plugin de depuração para a EVM que é muito poderoso e pode ser usado para realizar rastreamentos e análises detalhadas do bytecode da EVM. Um exemplo é mostrado aqui:

**Figura 10.4: Remix IDE, depuração**

A captura de tela acima mostra diferentes elementos do Remix IDE quando o **depurador** está em execução.  
O depurador tem o código-fonte decodificado em instruções da EVM. O usuário pode percorrer as instruções uma a uma e examinar o que o código-fonte faz quando é executado.

No lado direito, o código-fonte é exibido. Abaixo dele está o log de saída, que mostra mensagens informativas e dados relacionados à compilação e ao status de execução, bem como informações de transação/contrato.

No depurador, os opcodes são exibidos, juntamente com informações de memória, estado e pilha, o que torna a depuração mais fácil e intuitiva. Esse é um recurso muito útil, especialmente em casos de depuração de código complexo.

Na próxima seção, usaremos o **MetaMask**, uma extensão de navegador que serve como carteira de criptomoeda e interface para blockchains e dApps.

**Interagindo com a Blockchain Ethereum usando o MetaMask**

O MetaMask permite a interação com a blockchain Ethereum por meio dos navegadores Firefox e Chrome. Ele **injeta um objeto web3** no contexto JavaScript do site em execução, o que permite capacidade imediata de interface para dApps. Essa injeção permite que dApps interajam diretamente com a blockchain.

O MetaMask também permite o gerenciamento de contas e registra todas as transações para essas contas. Isso atua como um método de verificação antes de qualquer transação ser executada na blockchain. O usuário vê uma interface segura para revisar a transação para **aprovação ou rejeição** antes que ela possa alcançar a blockchain de destino.

Ele permite conectividade com várias redes Ethereum. Um recurso interessante a notar é que o MetaMask pode conectar-se a qualquer blockchain Ethereum via uma conexão RPC personalizada. Ele pode se conectar **não apenas a blockchains remotas**, mas também a blockchains locais em execução. Tudo o que precisa é de uma conexão RPC exposta por um nó executando a blockchain. Se estiver disponível, o MetaMask pode se conectar e permitirá que um navegador da web se conecte a ele via o objeto web3. O MetaMask também pode ser usado para se conectar a uma blockchain de teste local (ou simulada), como Ganache e TestRPC.

**Instalando o MetaMask**

O MetaMask funciona como um plugin ou extensão no navegador da web. Está disponível para os navegadores Chrome, Firefox, Opera e Brave.  
A ideia principal por trás do desenvolvimento do MetaMask é fornecer uma interface com a blockchain Ethereum. Ele permite o gerenciamento eficiente de contas e conectividade com a blockchain Ethereum sem a necessidade de executar localmente o software do nó Ethereum.

O MetaMask permite a conectividade com a blockchain Ethereum por meio da infraestrutura disponível no [Infura](https://infura.io). Isso permite que os usuários interajam com a blockchain **sem precisar hospedar nenhum nó localmente**.

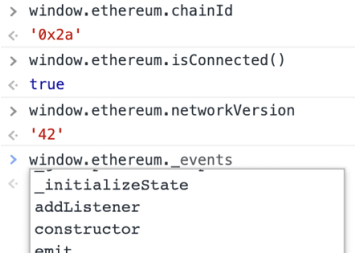
Aqui, abordaremos o processo de instalação. Estamos usando o Chrome como exemplo; no entanto, as instruções são as mesmas para Firefox e Brave:

1. Acesse [https://metamask.io/](https://metamask.io), onde estão disponíveis os links para download da extensão para seu navegador.

Mais informações estão disponíveis em: <https://github.com/MetaMask/metamask-plugin>

1. Instale a extensão para o navegador. A instalação será rápida e, se tudo ocorrer bem, você verá uma mensagem dizendo **Bem-vindo ao MetaMask**.
2. Clique em **Começar (Get Started)**, e duas opções serão exibidas: importar uma carteira existente usando a frase-semente, ou **criar uma nova carteira**. Selecionaremos **Criar uma Carteira** aqui.
3. Em seguida, crie uma senha.
4. Opcionalmente, configure uma Frase de Backup Secreta.
5. Após a conclusão de tudo, você verá uma mensagem de congratulação.
6. Após clicar no botão **Tudo Pronto (All Done)**, será exibida a visualização principal do MetaMask.
7. Após a configuração, você verá a visualização principal de contas.

Uma vez instalado, a API de provedor Ethereum estará disponível no contexto do navegador, que pode ser usada para interagir com a blockchain.  
O MetaMask **injeta uma API global** nos sites em window.ethereum.  
Um exemplo da API Ethereum é mostrado aqui usando o console JavaScript no Chrome:



\*\*Figura 10.5: Objeto Ethereum\*\*

O MetaMask pode conectar-se a várias redes. Por padrão, ele conecta-se à **mainnet** do Ethereum. Outras redes incluem, entre outras, a rede de testes **Sepolia** e a rede de testes **Goerli**.  
Além disso, ele pode se conectar a qualquer nó local via **Localhost 8545**, e a uma **RPC personalizada**, o que significa que ele pode se conectar a qualquer rede desde que as informações da conexão RPC estejam disponíveis.

Agora, vejamos como funciona o gerenciamento de contas no MetaMask.

**Criando e financiando uma conta com o MetaMask**

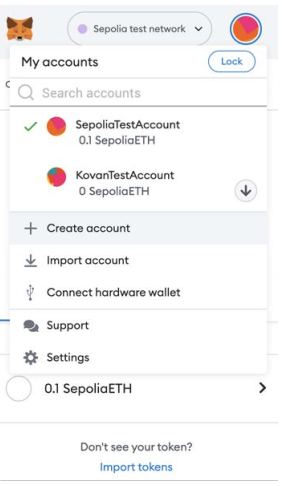
Vamos agora criar uma conta e financiá-la com algum ETH, tudo usando o MetaMask. Observe que estamos conectados à rede de teste Sepolia.  
Abra o MetaMask e certifique-se de que ele está conectado à rede de teste Sepolia:

1. Abra a extensão do MetaMask e clique no círculo no canto superior direito. O menu será exibido; clique em **Criar Conta (Create Account)**.
2. Digite o novo nome da conta, por exemplo **SepoliaTestAccount**, e clique em **Criar**, o que imediatamente criará uma nova conta.

Uma vez criada, podemos financiá-la com os seguintes passos.

1. Copie o endereço da conta para a área de transferência.
2. Vá até <https://sepolia-faucet.pk910.de> e insira o endereço da conta **SepoliaTestAccount**.  
   Você receberá algum ETH em sua conta.

Agora, observe no MetaMask que algum ETH está disponível.



\*\*Figura 10.6: Menu Minhas Contas no MetaMask\*\*

1. Agora crie outra conta e transfira algum ETH da **SepoliaTestAccount** para a nova conta. Você pode fazer isso com os seguintes passos:
2. Clique no ícone de envio (**Send**).
3. Selecione a conta de destino na opção **Transferência entre minhas contas (Transfer between my accounts)**.
4. Pressione **Próximo (Next)**.
5. Clique na opção **EDITAR (EDIT)** no canto superior direito para editar a prioridade da transação: baixa, média ou alta.
6. Clique ainda em **Opções avançadas (Advanced options)** para alterar o limite de gás (o limite de gás é o número máximo de unidades de gás que você está disposto a usar).

Um faucet alternativo pode ser encontrado em:  
<https://sepolia-faucet.pk910.de>  
Se você não encontrar nenhum faucet funcionando, simplesmente pesquise por "sepolia testnet faucet" em seu buscador favorito.

1. Confirme a transação.
2. Agora, observe que o Ether foi enviado para a outra conta.

Observe também os logs detalhados e as informações relativas à transação clicando sobre ela.  
Informações detalhadas também podem ser visualizadas no **explorador de blocos** relevante, clicando na opção **Ver no explorador de blocos (View on block explorer)**, que levará você a este link com o resumo da transação no Etherscan:

<https://sepolia.etherscan.io/tx/0x96e3688cba95c68c2922453872a3fd1d98060b5884f74866f69bda1db8f610cd>

**Usando MetaMask e Remix IDE para implantar um contrato inteligente**

Como o MetaMask injeta um objeto web3 no navegador, podemos usá-lo com o Remix IDE para implantar contratos na blockchain.  
É muito fácil implantar novos contratos usando o MetaMask e o Remix. O Remix IDE fornece uma interface onde contratos podem ser escritos em Solidity e, em seguida, implantados na blockchain.

Usaremos o Remix IDE e o MetaMask para implantar um **contrato inteligente de exemplo** na blockchain privada em execução localmente, que criamos anteriormente neste capítulo.  
Neste exercício, será usado um contrato simples que pode realizar **vários cálculos aritméticos simples** sobre um parâmetro de entrada.  
Como ainda não introduzimos o Solidity, o objetivo aqui é demonstrar **apenas o processo de implantação e interação com o contrato**.

Mais informações sobre codificação e o Solidity serão fornecidas no **Capítulo 11**, *Ferramentas, Linguagens e Frameworks para Desenvolvedores Ethereum*, após o qual o código a seguir se tornará fácil de entender.  
Aqueles que já estão familiarizados com JavaScript ou qualquer outra linguagem semelhante, como C, acharão o código quase autoexplicativo.

Agora, vejamos alguns exemplos de como o MetaMask pode ser usado na prática com a rede privada que criamos anteriormente neste capítulo.

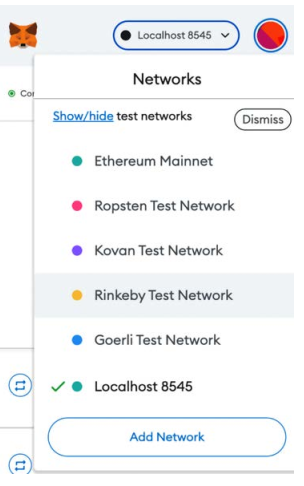
As unidades de gás podem ser especificadas para “**Taxa máxima prioritária**” e “**Taxa máxima**”.  
A taxa prioritária máxima (também conhecida como “gorjeta para mineradores”) vai diretamente para os mineradores e os incentiva a priorizar sua transação.  
Você normalmente pagará o valor máximo definido.  
A taxa máxima é o valor mais alto que você pagará (taxa base + taxa prioritária).  
Geralmente não é necessário alterar nenhuma dessas configurações, mas para controlar a taxa da transação e a velocidade de execução, essas opções podem ser ajustadas de acordo.  
Quanto maior a gorjeta para o minerador, maior será a velocidade da transação, pois os mineradores irão priorizá-la devido às taxas mais altas.

<https://sepolia.etherscan.io/tx/0x96e3688cba95c68c2922453872a3fd1d98060b5884f74866f69bda1db8f610cd>

**Adicionando uma rede personalizada ao MetaMask e conectando com o Remix IDE**

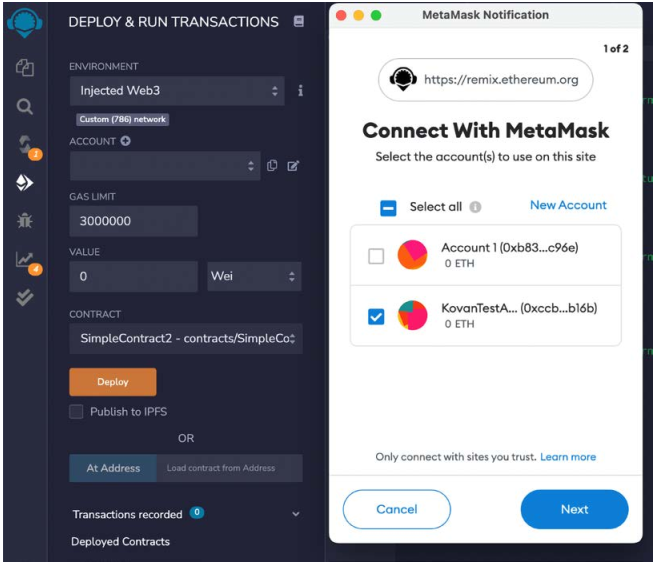
Primeiro, certifique-se de que você tem o MetaMask disponível, conforme configurado anteriormente.  
Nesta seção, adicionaremos nossa rede privada local ao MetaMask e, em seguida, interagiremos com ela usando o Remix IDE.

1. Abra o navegador Google Chrome, onde o MetaMask está instalado.  
   Selecione **Localhost 8545**, onde a instância do Geth da nossa rede privada está escutando:



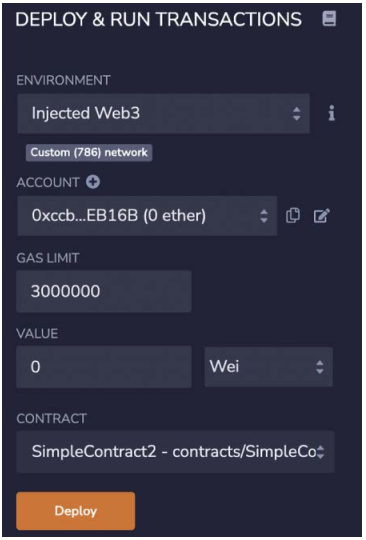
\*\*Figura 10.7: Seleção de rede no MetaMask\*\*

1. Navegue até o **Remix IDE** para desenvolvimento de contratos inteligentes Ethereum em seu navegador, no endereço:  
   <https://remix.ethereum.org>
2. Uma vez no site, observe a opção **DEPLOY & RUN TRANSACTIONS** na coluna à esquerda.  
   Escolha **Injected Web3** como o **ENVIRONMENT**, e uma janela do MetaMask será aberta:



\*\*Figura 10.8: Remix IDE e interação com MetaMask\*\*

Observe que, quando o Remix IDE está conectado ao MetaMask, ele mostrará informações da rede, como **Custom (786)** e informações da conta, conforme mostrado na captura de tela a seguir:



\*\*Figura 10.9: Informações de rede e de conta no Remix\*\*

Nesta seção, vimos como o Remix pode se conectar ao MetaMask usando o ambiente de **Web3 injetado**.  
O MetaMask pode ser conectado à **mainnet** ou a qualquer outra rede (local ou de teste), mas do ponto de vista do Remix, contanto que o objeto web3 injetado esteja disponível, ele se conectará a ele.

No nosso exemplo, como estamos usando uma rede privada, conectamo-nos à nossa rede privada por meio do MetaMask, e o Remix está se conectando ao MetaMask usando o ambiente Web3 injetado.  
Agora, mesmo que essa conectividade esteja disponível, o MetaMask **não sabe sobre as contas que criamos** na nossa rede privada.  
Para que o MetaMask opere sobre contas existentes, precisamos **importá-las a partir do keystore existente**, no nosso caso, o keystore da rede privada 786.  
Veremos como isso é feito na próxima seção.

Deseja que eu continue com a próxima seção: **Importando contas no MetaMask usando arquivos de keystore**?

**Você disse:**

continue

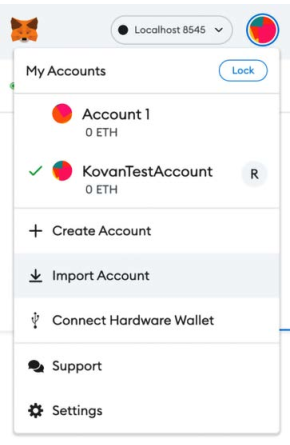
**O ChatGPT disse:**

Dando continuidade à tradução literal do capítulo 10:

**Importando contas no MetaMask usando arquivos de keystore**

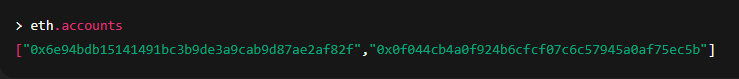
Podemos importar contas Ethereum existentes no MetaMask importando os arquivos de keystore gerados pelo Geth como resultado da criação de contas.  
Para demonstrar como isso funciona, agora importaremos as contas da nossa blockchain de rede privada, chamada **786**, para o MetaMask.

No MetaMask, a opção **Import Account (Importar Conta)** está disponível no menu **Minhas Contas**, conforme mostrado na captura de tela a seguir:

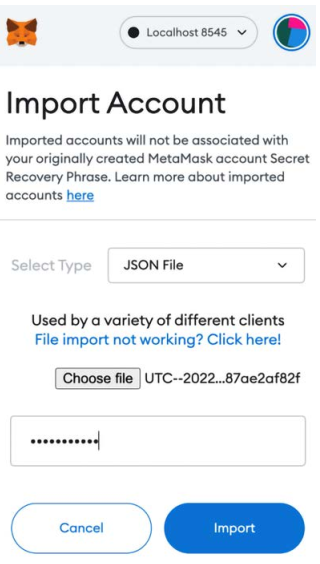


\*\*Figura 10.10: Opção de importação de conta no MetaMask\*\*

Agora vamos importar as contas.  
Lembre-se de que criamos duas contas anteriormente em nossa rede privada 786:



Podemos importar essas contas para o MetaMask usando os respectivos arquivos JSON de keystore.



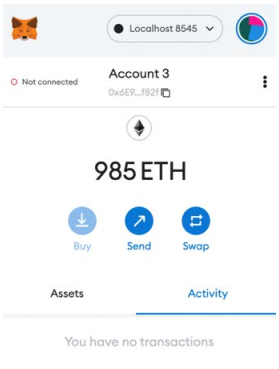
\*\*Figura 10.11: Arquivo de importação JSON no MetaMask\*\*

Os passos são os seguintes:

1. Primeiro, escolha o arquivo JSON a partir do diretório keystore da rede privada, ~/etherprivate/keystore.  
   Os arquivos de keystore da nossa rede privada estão listados aqui:

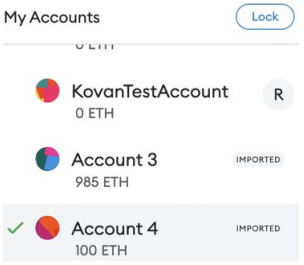


1. Simplesmente navegue até o keystore e selecione o arquivo de keystore, depois insira a senha (anteriormente, quando criamos as contas pela primeira vez, definimos a senha como Password123), e clique em **Importar**. Pode levar alguns segundos para importar. Quando for importada, a conta será visível na janela do MetaMask, conforme mostrado aqui:



\*\*Figura 10.12: Conta importada no MetaMask\*\*

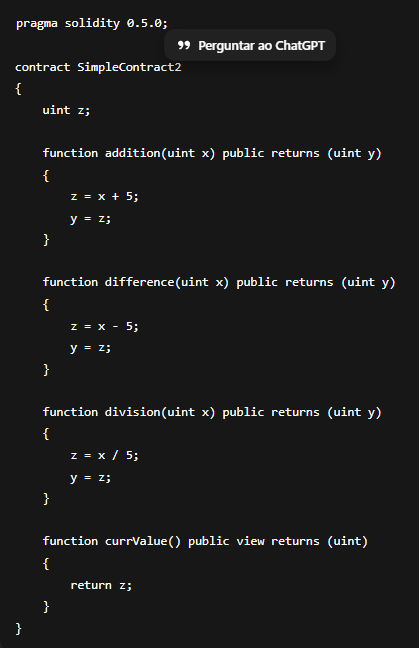
1. Da mesma forma, podemos importar a outra conta, selecionando o arquivo de keystore e importando-o.  
   Por fim, teremos duas contas listadas no MetaMask, conforme mostrado na imagem a seguir:

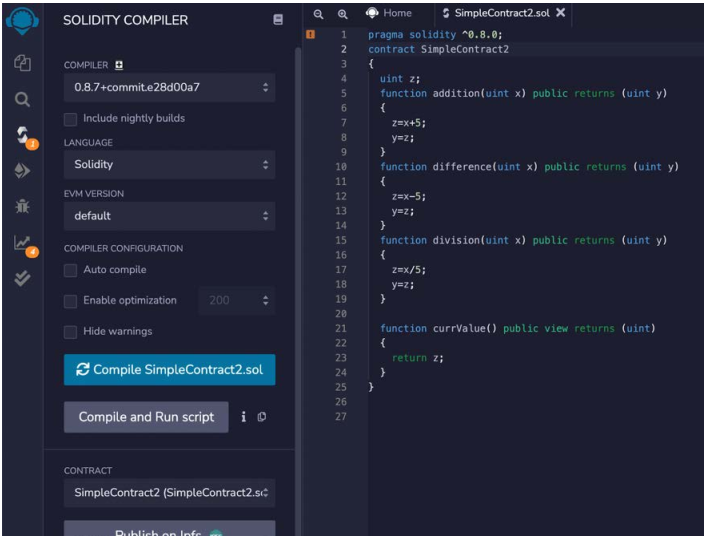


\*\*Figura 10.13: MetaMask – duas contas importadas\*\*

**Implantando um contrato com o MetaMask**

Nesta seção, escreveremos um contrato inteligente simples e o implantaremos em nossa rede privada usando o MetaMask.  
Primeiro, no Remix, criamos um novo arquivo. Digite o código mostrado para SimpleContract2 conforme segue:

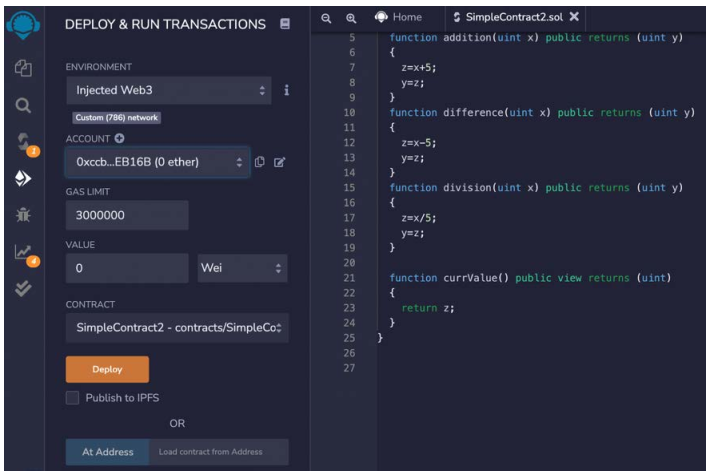
]



\*\*Figura 10.14: SimpleContract2 no Remix\*\*

O código aparecerá assim no Remix.

Agora compilamos o contrato inteligente SimpleContract2 clicando no botão Compile SimpleContract2.sol, conforme mostrado acima.

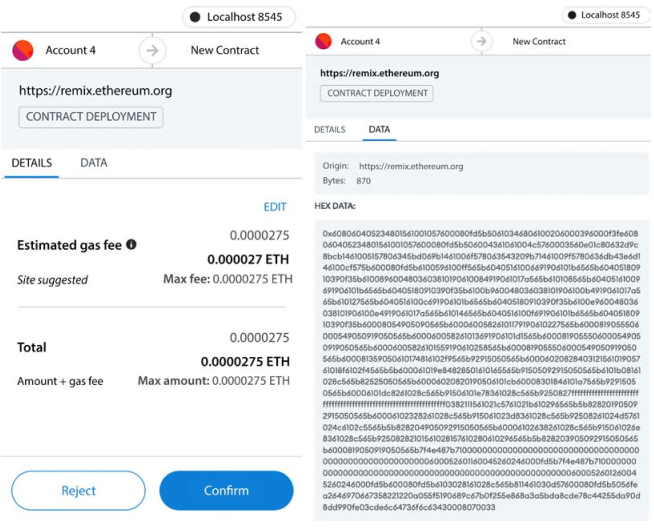


\*\*Figura 10.15: Implantação de contrato inteligente\*\*

Uma vez compilado com sucesso, implantamos o contrato inteligente na rede privada:

Clique em Deploy, e revise os detalhes do contrato. Na aba DATA, você pode ver o código do contrato.

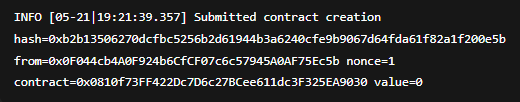
Depois clique no botão Confirmar.



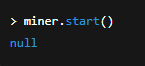
\*\*Figura 10.16: Implantação de contrato pelo MetaMask\*\*

Observe como nos logs do Geth vemos Submitted contract creation, o que significa que uma transação de criação de contrato foi enviada e reconhecida como resultado da ação de implantação a partir do Remix IDE e da ação Confirmar no MetaMask.

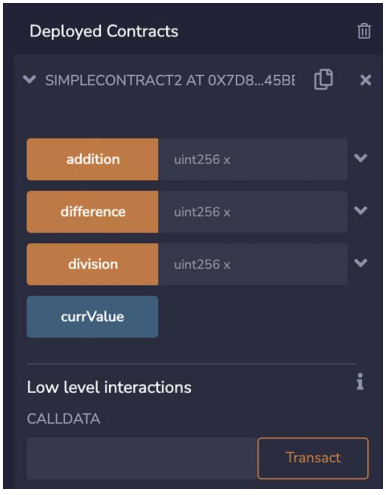
Note que também é mostrado o hash completo e o endereço do contrato:



Agora iniciamos novamente a mineração, emitindo o seguinte comando no console JavaScript do Geth.  
Se a mineração já estiver em execução, este passo não é necessário:



Uma vez que o contrato é minerado e implantado, você pode vê-lo no Remix, na visualização **Deployed Contracts**:

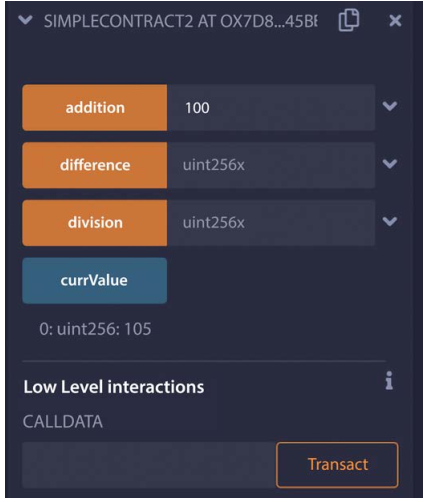


\*\*Figura 10.17: Contrato implantado com MetaMask\*\*

Com isso, implantamos com sucesso nosso contrato inteligente de exemplo na rede privada.  
A seguir, veremos como podemos interagir com este contrato usando o Remix IDE e o MetaMask.

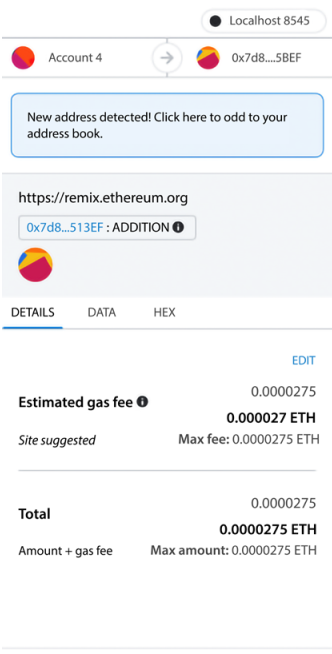
**Interagindo com um contrato através do MetaMask usando o Remix IDE**

Agora podemos interagir com o contrato usando o MetaMask.  
Executamos a função **addition**, inserimos o valor **100** no campo de entrada e clicamos no botão **addition**, como mostrado na captura de tela abaixo, no lado esquerdo:



\*\*Figura 10.18: Funções do contrato inteligente\*\*

Isso invocará a janela do MetaMask, como mostrado na captura de tela à direita:



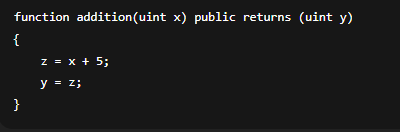
\*\*Figura 10.19: Interação com contrato no MetaMask\*\*

Clique em **Confirmar**, e a transação será minerada como de costume, porque temos um minerador rodando em segundo plano no Geth.

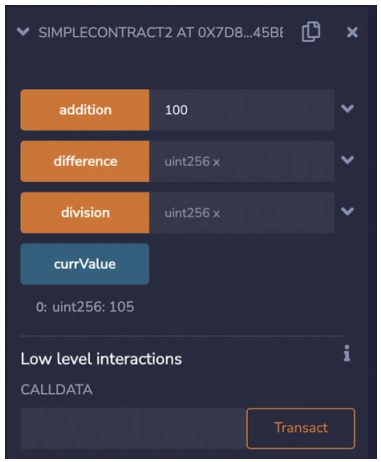
Agora clique no botão **currValue**, que irá ler o contrato para obter o valor atual, que é, como esperado, **105**.  
Esse valor é o resultado da operação de adição realizada.

**<Recapitulando>**

Lembre-se de que fornecemos o valor 100, e no contrato inteligente já codificamos o número 5 na função addition, como mostrado aqui:



Isso significa que os 100 fornecidos por nós como entrada são somados a 5, que está codificado na função addition do contrato inteligente, e o resultado desse cálculo é **105**, como esperado.



\*\*Figura 10.20: Recuperando o valor atual do contrato inteligente\*\*

Esse é um programa muito simples, mas o objetivo deste exercício é demonstrar o uso do MetaMask e como ele pode se conectar com a **cadeia privada local**, usando o provedor RPC em execução no Geth na porta **8545**.

Preste atenção especial às configurações do compilador Solidity.  
Escolha a versão apropriada do compilador e a versão da EVM de acordo com o seu código-fonte e com a versão da EVM do seu cliente Geth.  
Se isso não for selecionado corretamente, você poderá enfrentar problemas.  
Por exemplo, se for escolhida uma versão incorreta da EVM, ao interagir com o contrato, pode aparecer um erro como:  
**Gas estimation failed**.

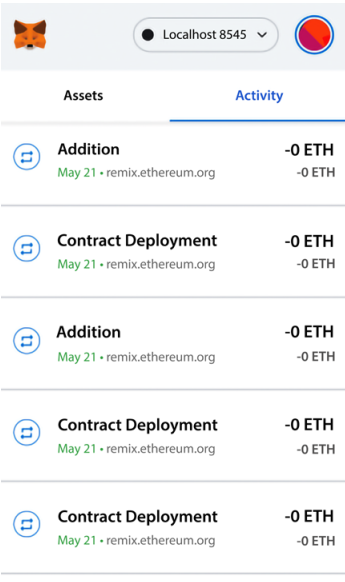
Da mesma forma, se for escolhida uma versão incorreta da EVM, então você também verá uma mensagem de erro na janela do Remix.  
É melhor configurar corretamente as configurações, que podem simplesmente ser deixadas como padrão na maioria dos casos, **a menos que haja recursos específicos de uma versão da EVM que você esteja testando**.  
Compilar para a versão errada da EVM pode resultar em comportamento incorreto e falhas.  
Por exemplo, se você tiver escolhido uma versão desatualizada da EVM, então mesmo que a compilação seja bem-sucedida, o bytecode em tempo de execução pode não funcionar corretamente e relatar um erro dizendo que o *opcode* é inválido.  
Em outros casos, a execução pode até falhar silenciosamente ou com pouca indicação nos logs do Geth sobre o que aconteceu, levando a horas de tempo perdido em depuração.  
Isso é especialmente verdadeiro em redes privadas; portanto, certifique-se de que a versão correta da EVM que corresponde à versão do cliente Geth esteja sendo usada.

Observe também que a versão do compilador no Remix IDE está definida para a versão da EVM especificada no código-fonte do Solidity na linha:

pragma solidity 0.8.0;

Além disso, a versão da EVM deve ser deixada como **default**, que é a última versão da EVM.

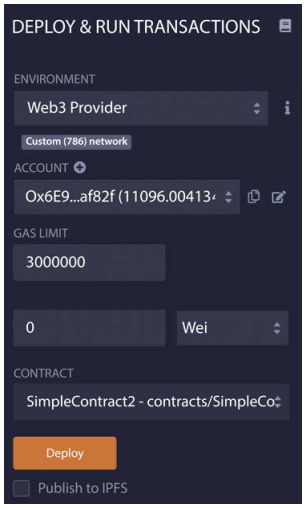
Na visualização do MetaMask, você também pode ver o **histórico de transações** do contrato clicando na aba **Atividades (Activity)**:



\*\*Figura 10.21: Histórico de transações no MetaMask\*\*

**Outra forma de se conectar à rede privada local (nó Geth)**

Outra forma de se conectar à rede privada local (nó Geth) é conectar-se diretamente usando o **Web3 provider**, disponível em **DEPLOY & RUN TRANSACTIONS**.  
Isso permitirá a comunicação direta com a blockchain por meio do Remix via RPC, **sem necessidade do ambiente Web3 injetado**:

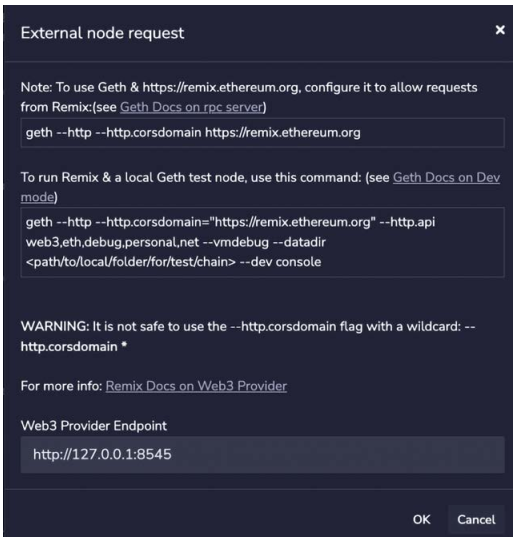


\*\*Figura 10.22: Implantar e executar transações no Remix IDE\*\*

Observe na captura de tela acima que o **ENVIRONMENT** está definido como **Web3 Provider**, que simplesmente se conecta a:

<http://localhost:8545>

Uma vantagem desse método é que ele é um método rápido e fácil para conectar o Remix à blockchain subjacente (o nó Geth).  
Não há necessidade de ferramentas adicionais; tudo o que você precisa é do Remix IDE (que também é baseado em navegador) e do seu nó Geth em execução localmente.  
A desvantagem, porém, é que esse método **não é muito seguro**.  
Para conectar o Remix IDE a um nó local, você pode executar o cliente Geth usando a flag --http.corsdomain, conforme mostrado na captura de tela abaixo:



\*\*Figura 10.23: Provedor Web3 no Remix\*\*

Observe que nosso processo Geth está sendo executado na rede privada com os parâmetros necessários já configurados, o que permite a conexão do **Web3 provider**, como mostrado aqui.  
Observe a flag --http.corsdomain na Figura 10.22 sendo usada aqui.  
Se o processo geth ainda **não** estiver em execução, emita o seguinte comando:



Observe que, no comando acima, temos a flag --http.corsdomain definida.  
Se não tivéssemos essa flag definida, então a única opção teria sido conectar via **Web3 injetado**, que, na verdade, é uma forma **mais segura** de conectar-se à blockchain usando o MetaMask.  
Dessa forma, usamos isso como padrão, mas, se você tiver uma rede de teste local simples, **você pode sempre usar a opção Web3 Provider** para interagir rapidamente com a blockchain usando o Remix IDE.

Como estamos usando uma rede privada, nosso comando é um pouco diferente do mostrado anteriormente, mas faz uso da flag --http.corsdomain, que é essencial para permitir a conectividade do nó Geth local com o Remix.

**Usar a flag --http.corsdomain com o caractere curinga ('\*')**

Usar a flag --http.corsdomain com o caractere curinga (--http.corsdomain '\*') pode ser apropriado com uma cadeia de testes local e contas de teste.  
No entanto, **não é recomendável em redes públicas**, pois o caractere curinga \* **permitirá que qualquer um se conecte ao nó**.  
Portanto, é recomendado que o acesso seja restrito especificando **apenas URLs confiáveis**.

Agora, podemos parar o minerador; lembre-se de que este é o minerador que iniciamos anteriormente durante a configuração da rede privada:



Com este exemplo, agora cobrimos como o MetaMask pode ser usado para se conectar a uma rede local, e como **Remix e MetaMask**, ou apenas o **Remix**, podem ser usados para se conectar à rede local e implantar contratos inteligentes.  
Também vimos como interagir com um contrato implantado.

O Remix é um IDE muito rico em recursos e **não cobrimos tudo**; no entanto, este capítulo introduziu formalmente você ao Remix, descrevendo a maioria dos principais recursos em detalhes, e continuaremos a usá-lo nos próximos capítulos.

Também é útil ter um mecanismo para visualizar **uma lista consolidada e detalhes de todas as transações** na blockchain.  
Para este propósito, são usados **exploradores de blocos**.  
Há muitos serviços disponíveis para blockchains públicas na internet.  
Existem também muitos projetos de código aberto disponíveis.

Como há muitas opções, **não descreveremos um específico aqui**; todos funcionam de maneira mais ou menos semelhante.  
Faça uma pesquisa no GitHub em:

<https://github.com/search?q=blockchain+explorer>

...para encontrar um apropriado e explorar mais.

**Resumo**

Neste capítulo, exploramos as redes de teste do Ethereum e como configurar redes privadas do Ethereum.  
Após a introdução inicial à configuração de uma rede privada, também vimos como a ferramenta de linha de comando do Geth pode ser usada para executar várias funções e como podemos interagir com a blockchain Ethereum.  
Também vimos como **MetaMask** e **Remix** podem ser usados para implantar contratos inteligentes.