ethclient

代码都在其他文件中

查询区块

有两种方式查询区块信息。

调用客户端的 HeaderByNumber 来返回有关一个区块的头信息。若传入 nil，它将返回最新的区块头。

完整区块

调用客户端的 BlockByNumber 方法来获得完整区块。您可以读取该区块的所有内容和元数据，例如，区块号，区块时间戳，区块摘要，区块难度以及交易列表等等。

调用 TransactionCount 只返回一个区块的交易数目。

查询区块中的交易

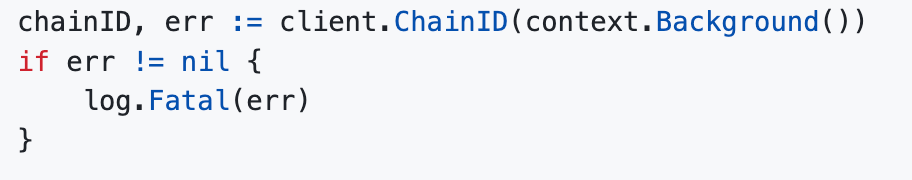
当使用 BlockByNumber 方法获取到完整的区块信息之后，可以调用区块实例的 Transactions 方法来读取块中的交易，该方法返回一个 Transaction 类型的列表。

循环遍历集合并获取交易的信息。

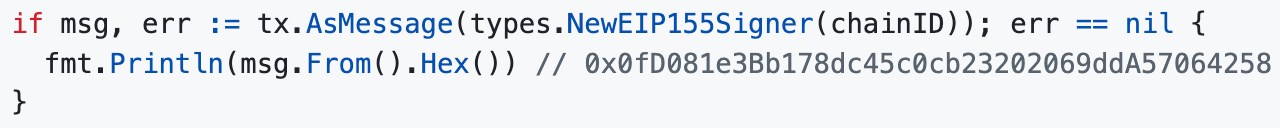


为了读取发送方的地址，我们需要在事务上调用 AsMessage，它返回一个 Message 类型，其中包含一个返回 sender（from）地址的函数。 AsMessage 方法需要 EIP155 签名者。

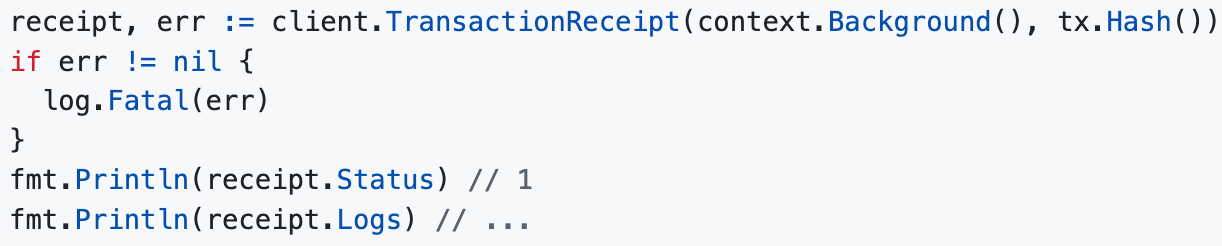
先从客户端拿到链 ID。



在使用 EIP155Signer 还原出 sender 地址：



每个交易都有一个收据，其中包含执行交易的结果，例如所有的返回值和日志，以及“1”（成功）或“0”（失败）的交易结果状态。



在不获取块的情况下遍历事务的另一种方式是调用客户端的 TransactionInBlock 方法。

此方法仅接受块哈希和块内事务的索引值。

调用 TransactionCount 来了解块中有多少个事务

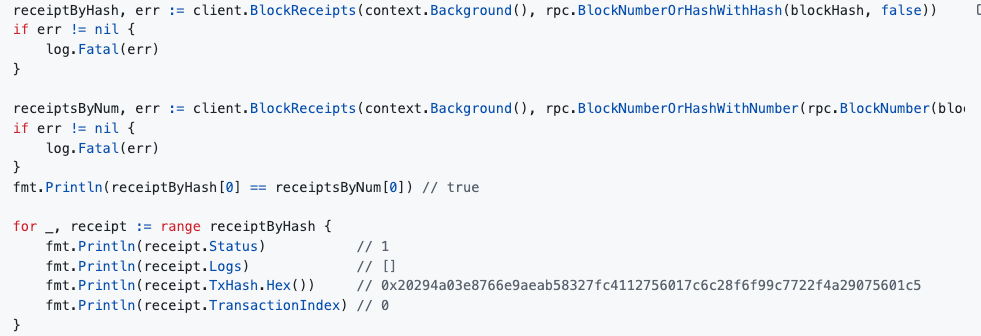


还可以使用 TransactionByHash 在给定具体事务哈希值的情况下直接查询单个事务。



查收收据

可以调用 BlockReceipts 方法就可以得到指定区块中所有的收据列表。参数可以是区块的哈希也可以是区块高度。循环遍历集合并获取收据的信息。



查询收据除了使用区块的高度或哈希以外，也可以仅使用交易哈希查询，调用 TransactionReceipt 方法：



创建新钱包

首先生成一个新的钱包，我们需要导入 go-ethereum crypto 包，该包提供用于生成随机私钥的 GenerateKey 方法。



如果已经有了私钥的 Hex 字符串，也可以使用 HexToECDSA 方法恢复私钥：



然后我们可以通过导入 golang crypto/ecdsa 包并使用 FromECDSA 方法将其转换为字节。



我们现在可以使用 go-ethereum hexutil 包将它转换为十六进制字符串，该包提供了一个带有字节切片的 Encode 方法。 然后我们在十六进制编码之后删除“0x”。



这就是用于签署交易的私钥，将被视为密码，永远不应该被共享给别人，因为谁拥有它可以访问你的所有资产。

由于公钥是从私钥派生的，因此 go-ethereum 的加密私钥具有一个返回公钥的 Public 方法。

publicKey := privateKey.Public()

将其转换为十六进制的过程与我们使用转化私钥的过程类似。 我们剥离了 0x 和前 2 个字符 04，它是 EC 前缀，不是必需的。



现在我们拥有公钥，就可以轻松生成你经常看到的公共地址。 为了做到这一点，go-ethereum 加密包有一个 PubkeyToAddress 方法，它接受一个 ECDSA 公钥，并返回公共地址。



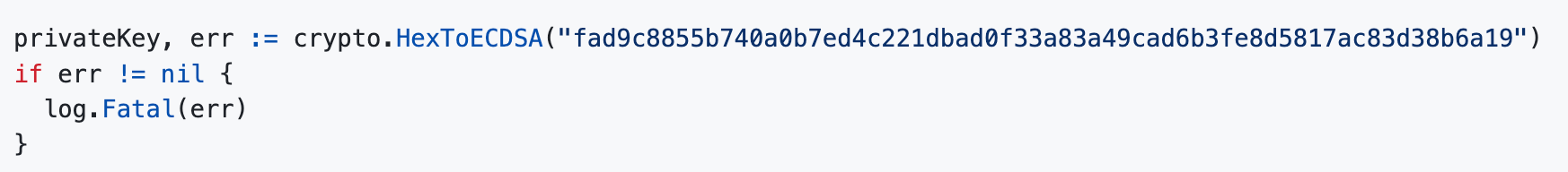
公共地址其实就是公钥的 Keccak-256 哈希，然后我们取最后 40 个字符（20 个字节）并用“0x”作为前缀。 以下是使用 golang.org/x/crypto/sha3 的 Keccak256 函数手动完成的方法。



ETH转账

转账交易包括打算转账的以太币数量，燃气限额，燃气价格，一个自增数(nonce)，接收地址以及可选择性的添加的数据。 在发送到以太坊网络之前，必须使用发送方的私钥对该交易进行签名。

假设我们已经连接了客户端，下一步就是加载私钥。

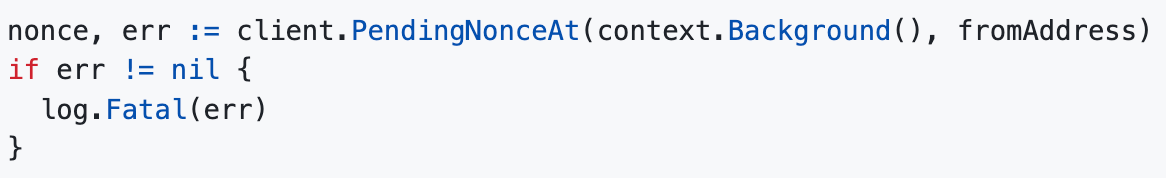


之后我们需要获得帐户的随机数(nonce)。 每笔交易都需要一个 nonce。 根据定义，nonce 是仅使用一次的数字。 如果是发送交易的新帐户，则该随机数将为“0”。 来自帐户的每个新事务都必须具有前一个 nonce 增加 1 的 nonce。很难对所有 nonce 进行手动跟踪，于是 ethereum 客户端提供一个帮助方法 PendingNonceAt，它将返回你应该使用的下一个 nonce。

该函数需要我们发送的帐户的公共地址 - 这个我们可以从私钥派生。



接下来我们可以读取我们应该用于帐户交易的随机数。



下一步是设置我们将要转移的 ETH 数量。 但是我们必须将 ETH 以太转换为 wei，因为这是以太坊区块链所使用的。 以太网支持最多 18 个小数位，因此 1 个 ETH 为 1 加 18 个零。

value := big.NewInt(1000000000000000000) // in wei (1 eth)

ETH 转账的燃气应设上限为“21000”单位。

gasLimit := uint64(21000) // in units

燃气价格必须以 wei 为单位设定。 在撰写本文时，将在一个区块中比较快的打包交易的燃气价格为 30 gwei。

gasPrice := big.NewInt(30000000000) // in wei (30 gwei)

然而，燃气价格总是根据市场需求和用户愿意支付的价格而波动的，因此对燃气价格进行硬编码有时并不理想。 go-ethereum 客户端提供 SuggestGasPrice 函数，用于根据'x'个先前块来获得平均燃气价格。

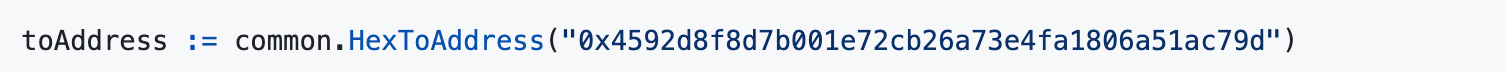
gasPrice, err := client.SuggestGasPrice(context.Background())

if err != nil {

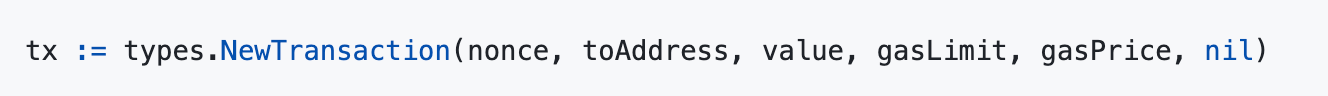
log.Fatal(err)

}

接下来我们弄清楚我们将 ETH 发送给谁。



现在我们最终可以通过导入 go-ethereum core/types 包并调用 NewTransaction 来生成我们的未签名以太坊事务，这个函数需要接收 nonce，地址，值，燃气上限值，燃气价格和可选发的数据。 发送 ETH 的数据字段为“nil”。 在与智能合约进行交互时，我们将使用数据字段，仅仅转账以太币是不需要数据字段的。



下一步是使用发件人的私钥对事务进行签名。 为此，我们调用 SignTx 方法，该方法接受一个未签名的事务和我们之前构造的私钥。 SignTx 方法需要 EIP155 签名者，这个也需要我们先从客户端拿到链 ID。



现在通过在 client 实例调用 SendTransaction 来将已签名的事务广播到整个网络。



代币转账

代币传输不需要传输 ETH，因此将交易“值”设置为“0”。

value := big.NewInt(0)

先将您要发送代币的地址存储在变量中。

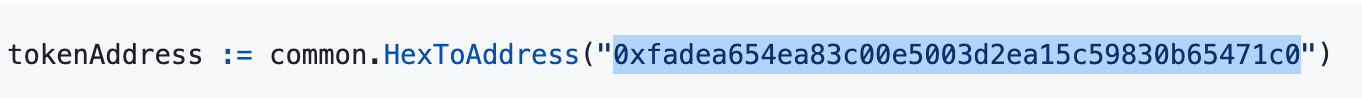


现在轮到有趣的部分。 我们需要弄清楚交易的 data 部分。 这意味着我们需要找出我们将要调用的智能合约函数名，以及函数将接收的输入。 然后我们使用函数名的 keccak-256 哈希来检索 方法 ID，它是前 8 个字符（4 个字节）。 然后，我们附加我们发送的地址，并附加我们打算转账的代币数量。 这些输入需要 256 位长（32 字节）并填充左侧。 方法 ID 不需填充。

为了演示，我创造了一个新的代币(RCCDemoToken, RDT)，可以使用 Remix 在线工具编译合约之后，部署到 Sepolia 网络：



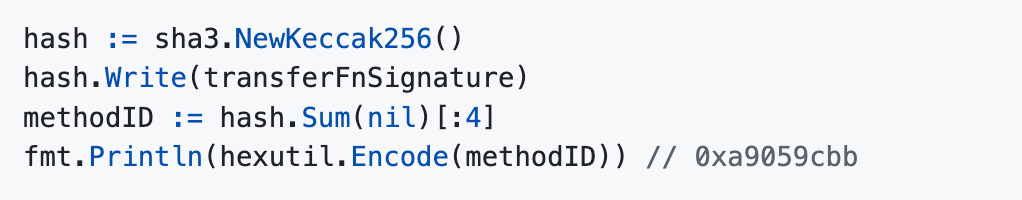
让我们将代币合约地址分配给变量。



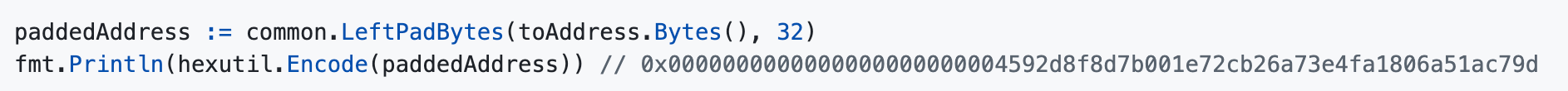
函数名将是传递函数的名称，即 ERC-20 规范中的 transfer 和参数类型。 第一个参数类型是 address（令牌的接收者），第二个类型是 uint256（要发送的代币数量）。 不需要没有空格和参数名称。 我们还需要用字节切片格式。



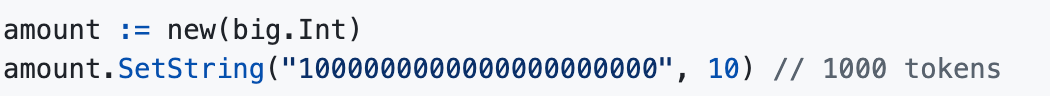
我们现在将从 go-ethereum 导入 crypto/sha3 包以生成函数签名的 Keccak256 哈希。 然后我们只使用前 4 个字节来获取方法 ID。



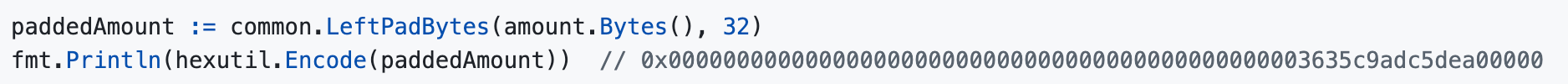
接下来，我们需要将给我们发送代币的地址左填充到 32 字节。



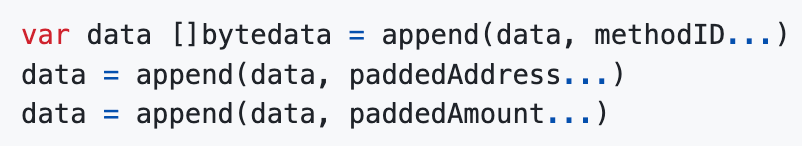
接下来我们确定要发送多少个代币，在这个例子里是 1,000 个，并且我们需要在 big.Int 中格式化为 wei。



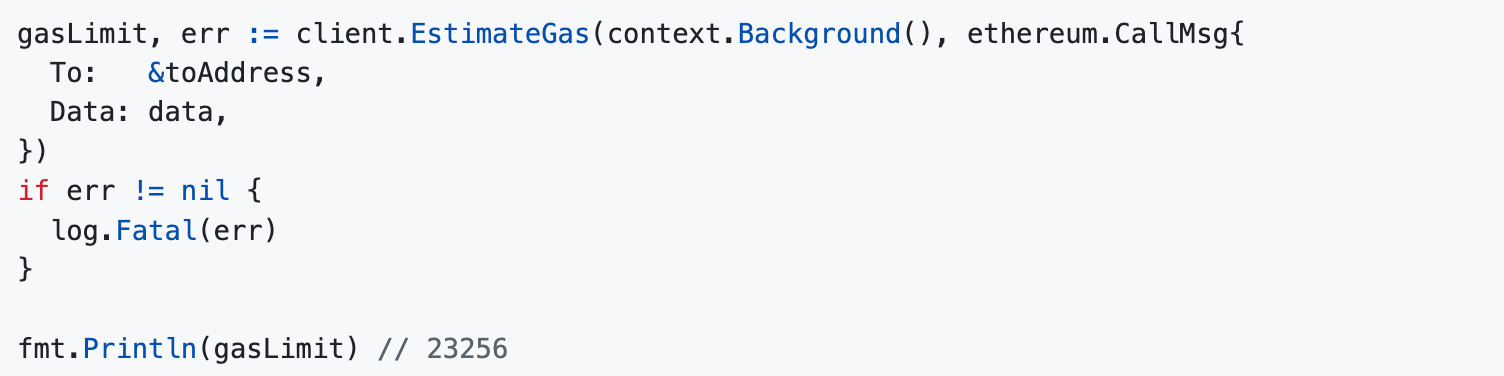
代币量也需要左填充到 32 个字节。



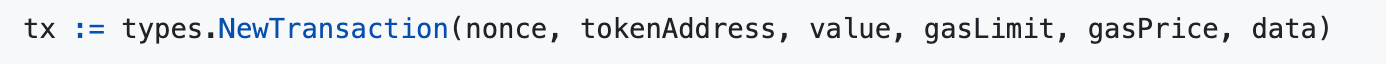
接下来我们只需将方法 ID，填充后的地址和填后的转账量，接到将成为我们数据字段的字节片。



燃气上限制将取决于交易数据的大小和智能合约必须执行的计算步骤。 幸运的是，客户端提供了 EstimateGas 方法，它可以为我们估算所需的燃气量。 这个函数从 ethereum 包中获取 CallMsg 结构，我们在其中指定数据和地址。 它将返回我们估算的完成交易所需的估计燃气上限。



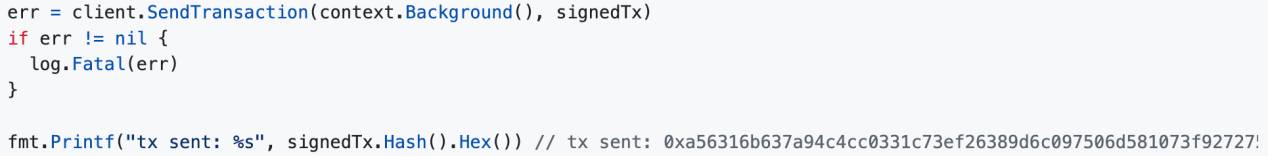
接下来我们需要做的是构建交易类型，这类似于在 ETH 转账部分中看到的，除了\_to\_字段将是代币智能合约地址。 这个常让人困惑。我们还必须在调用中包含 0 ETH 的值字段和刚刚生成的数据字节。



下一步是使用发件人的私钥对事务进行签名。 SignTx 方法需要 EIP155igner，需要我们先从客户端拿到链 ID。

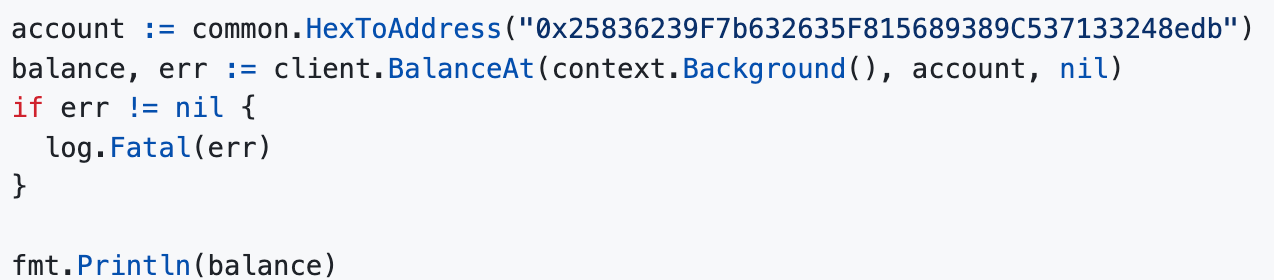


最后发送交易。

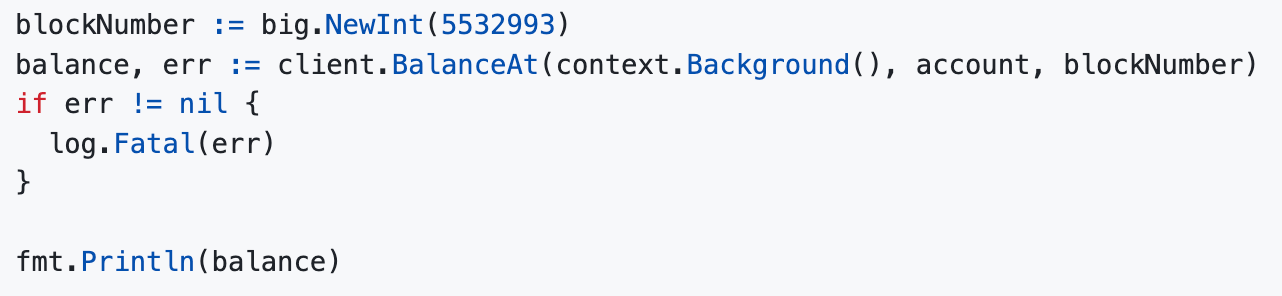


查询账户余额

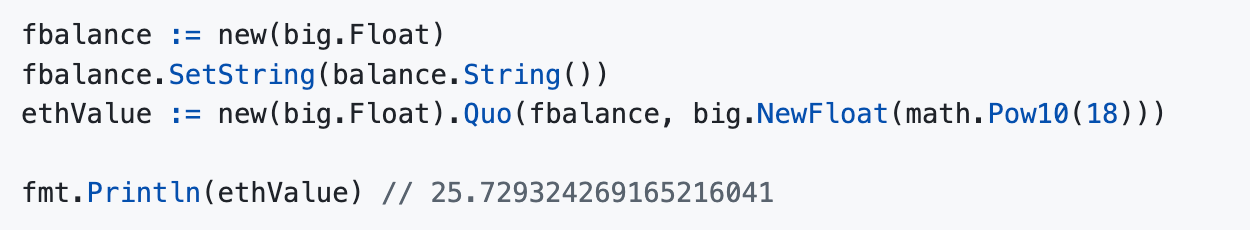
读取一个账户的余额相当简单。调用 ethclient 的 BalanceAt 方法，给它传递账户地址和可选的区块号。将区块号设置为 nil 将返回最新的余额。



传区块高度能读取指定区块时的账户余额，区块高度必须是 big.Int 类型。

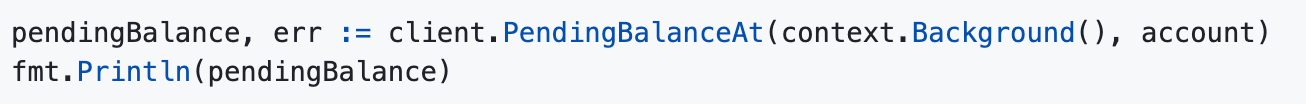


以太坊中的数字是使用尽可能小的单位来处理的，因为它们是定点精度，在 ETH 中它是\_wei\_。要读取 ETH 值，您必须做计算 wei/10^18。因为我们正在处理大数，我们得导入原生的 Go math 和 math/big 包。这是您做的转换。



待处理的余额

有时您想知道待处理的账户余额是多少，例如，在提交或等待交易确认后。客户端提供了类似 BalanceAt 的方法，名为 PendingBalanceAt，它接收账户地址作为参数。



查询代币余额 有问题

订阅区块

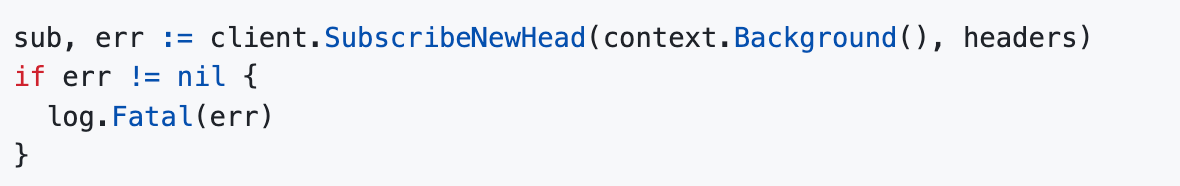
订阅区块需要 websocket RPC URL。



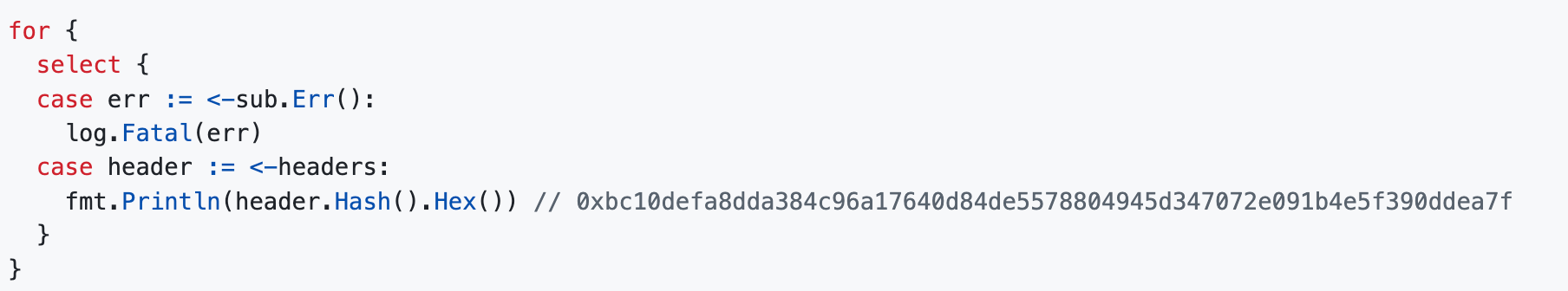
接下来，我们将创建一个新的通道，用于接收最新的区块头。



现在我们调用客户端的 SubscribeNewHead 方法，它接收我们刚创建的区块头通道，该方法将返回一个订阅对象。



订阅将推送新的区块头事件到我们的通道，因此我们可以使用一个 select 语句来监听新消息。订阅对象还包括一个 error 通道，该通道将在订阅失败时发送消息。



要获得该区块的完整内容，我们可以将区块头的摘要传递给客户端的 BlockByHash 函数。



正如您所见，您可以读取整个区块的元数据字段，交易列表等等。

部署合约

使用remix工具部署

也可以使用 Solidity 网页 IDE 工具 Remix。

以下是使用 Remix 工具部署合约到测试网的操作步骤：

首先打开 Remix IDE ，打开浏览器访问 Remix IDE 的官方网站：https://remix.ethereum.org/

导入智能合约到 Remix 的文件浏览器中，创建一个 store.sol 文件。把 2.10.0 步骤中的示例合约代码复制到这个 store.sol 文件中并保存。

在左侧菜单中选择"Solidity Compiler"，然后点击"Compile"按钮编译合约。确保编译器版本与你的合约兼容。

在左侧菜单中选择"Deploy & Run Transactions"。在"Environment"下拉菜单中，选择"Injected Web3"。允许使用 MetaMask 或其他 Web3 钱包连接到测试网。

确保 MetaMask 插件已经连接到所需的测试网（如 Goerli、Sepolia 等）。如果需要，添加测试网络到 MetaMask 并获取一些测试以太币。

部署合约 在 Remix 的"Deploy & Run Transactions"面板中：

选择你要部署的合约

如果合约构造函数需要参数，在"Deploy"按钮下方的输入框中填入参数点击"Deploy"按钮

MetaMask 会弹出一个窗口，要求你确认交易。检查 gas 费用和其他详情，然后确认交易。

等待部署完成 交易被确认后，合约将被部署到测试网。你可以在 Remix 的控制台中看到部署的详细信息，包括合约地址。

可以使用测试网的区块浏览器（https://sepolia.etherscan.io/）来验证合约是否成功部署，只需搜索合约地址即可。

部署完成后，就可以在 Remix 的"Deployed Contracts"部分与合约进行交互，调用函数或发送交易。

仅使用 ethclient 工具

以太坊中，部署合约其实也是发起了一笔交易，并不是一定需要 abigen 工具生成 go 代码。

不是只能使用生成的 go 的合约代码才能部署合约。

可以仅使用 ethclient，使用更底层的方法，直接通过发送交易的方式来部署合约。

使用 2.10.0 步骤中，生成的 store\_sol\_Store.bin 文件中的字符串作为交易数据，部署 store 合约。

import (

"context"

"crypto/ecdsa"

"encoding/hex"

"fmt"

"log"

"math/big"

"time"

"github.com/ethereum/go-ethereum"

"github.com/ethereum/go-ethereum/common"

"github.com/ethereum/go-ethereum/core/types"

"github.com/ethereum/go-ethereum/crypto"

"github.com/ethereum/go-ethereum/ethclient"

)

const (

// store合约的字节码

contractBytecode = ""

)

func main() {

// 连接到以太坊网络（这里使用 Goerli 测试网络作为示例）

client, err := ethclient.Dial("https://goerli.infura.io/v3/YOUR-PROJECT-ID")

if err != nil {

log.Fatal(err)

}

// 创建私钥（在实际应用中，您应该使用更安全的方式来管理私钥）

privateKey, err := crypto.HexToECDSA("YOUR-PRIVATE-KEY-HERE")

if err != nil {

log.Fatal(err)

}

publicKey := privateKey.Public()

publicKeyECDSA, ok := publicKey.(\*ecdsa.PublicKey)

if !ok {

log.Fatal("error casting public key to ECDSA")

}

fromAddress := crypto.PubkeyToAddress(\*publicKeyECDSA)

// 获取nonce

nonce, err := client.PendingNonceAt(context.Background(), fromAddress)

if err != nil {

log.Fatal(err)

}

// 获取建议的gas价格

gasPrice, err := client.SuggestGasPrice(context.Background())

if err != nil {

log.Fatal(err)

}

// 解码合约字节码

data, err := hex.DecodeString(contractBytecode)

if err != nil {

log.Fatal(err)

}

// 创建交易

tx := types.NewContractCreation(nonce, big.NewInt(0), 3000000, gasPrice, data)

// 签名交易

chainID, err := client.NetworkID(context.Background())

if err != nil {

log.Fatal(err)

}

signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), privateKey)

if err != nil {

log.Fatal(err)

}

// 发送交易

err = client.SendTransaction(context.Background(), signedTx)

if err != nil {

log.Fatal(err)

}

fmt.Printf("Transaction sent: %s\n", signedTx.Hash().Hex())

// 等待交易被挖矿

receipt, err := waitForReceipt(client, signedTx.Hash())

if err != nil {

log.Fatal(err)

}

fmt.Printf("Contract deployed at: %s\n", receipt.ContractAddress.Hex())

}

func waitForReceipt(client \*ethclient.Client, txHash common.Hash) (\*types.Receipt, error) {

for {

receipt, err := client.TransactionReceipt(context.Background(), txHash)

if err == nil {

return receipt, nil

}

if err != ethereum.NotFound {

return nil, err

}

// 等待一段时间后再次查询

time.Sleep(1 \* time.Second)

}

}

加载合约

加载合约是在成功部署合约后，使用合约的 abi 文件生成 go 合约代码，并且在代码中初始化这个合约的实例。

初始化合约实例时需要提供两个参数，分别是 ethclient 的实例和合约地址。

使用 2.10 中的示例合约代码生成 abi 文件：

solcjs --abi store.sol

然后使用 abigen，仅根据 abi 文件生成合约代码：

abigen --abi=Store\_sol\_Store.abi --pkg=store --out=store.go

使用生成的代码加载

package main

import (

"log"

"github.com/ethereum/go-ethereum/common"

"github.com/ethereum/go-ethereum/ethclient"

"github.com/learn/init\_order/store"

)

const (

contractAddr = "0x8D4141ec2b522dE5Cf42705C3010541B4B3EC24e"

)

func main() {

client, err := ethclient.Dial("http://127.0.0.1:7545")

if err != nil {

log.Fatal(err)

}

storeContract, err := store.NewStore(common.HexToAddress(contractAddr), client)

if err != nil {

log.Fatal(err)

}

\_ = storeContract

}

使用 Remix 在线 IDE 工具加载合约

使用 Remix 在线 IDE 工具加载合约的前提是必须有合约代码，需要先在 Remix 中把合约编译一遍之后，才可以使用合约地址加载合约。

使用 Remix 在线 IDE 工具加载合约步骤：

导入智能合约到 Remix 的文件浏览器中，创建一个 store.sol 文件。把 2.10.0 步骤中的示例合约代码复制到这个 store.sol 文件中并保存。

在左侧菜单中选择"Solidity Compiler"，然后点击"Compile"按钮编译合约。确保编译器版本与你的合约兼容。

在左侧菜单中选择"Deploy & Run Transactions"。在"Environment"下拉菜单中，选择"Injected Web3"。允许使用 MetaMask 或其他 Web3 钱包连接到测试网。

在"Contract"下来菜单中选中 store 合约，然后在"At Address"按钮边的输入框中，输入合约地址，点击"At Address"按钮。

可以看到"Deployed/Unpinned Contracts"栏新增一条记录，点击下拉按钮，即可与合约进行交互，调用函数或发送交易。

执行合约

合约事件