a little book on

ETHEREUM DEVELOPMENT WITH GO

Miguel Mota

Table of Contents

用Go来做以太坊开发

这本迷你书的本意是给任何想用Go进行以太坊开发的同学一个概括的介绍。本意是如果你已经对以太坊和Go有一些熟悉，但是对于怎么把两者结合起来还有些无从下手，那这本书就是一个好的起点。你会学习如何用Go与智能合约交互，还有如何完成一些日常的查询和任务。

这本书里有很多我希望我当初学习用Go以太坊开发的时候能有的代码范例。你上手Go语言以太坊开发的大部分所需知识，这本书里面都会手把手介绍到。

当然了，以太坊还是一直在飞速的发展的进化的。所以难免会有些过期的内容，或者你认为有可以值得提升的地方，那就是你提 issue 或者 pull request 的好机会了：）这本书是完全开源并且免费的，你可以在 github 上看到源码.

在线电子书

https://goethereumbook.org

电子书

电子书有三种格式。

- PDF
- EPUB
- MOBI

介绍

> 以太坊是一个开源，公开，基于区块链的分布式计算平台和具备智能合约（脚本）功能的操作系统。它通过基于交易的状态转移支持中本聪共识的一个改进算法。

-维基百科

以太坊是一个区块链，允许开发者创建完全去中心化运行的应用程序，这意味着没有单个实体可以将其删除或修改它。部署到以太坊上的每个应用都由以太坊网络上每个完整客户端执行。

Solidity

Solidity是一种用于编写智能合约的图灵完备编程语言。Solidity被编译成以太坊虚拟机可执行的字节码。

go-ethereum

本书中，我们将使用Go的官方以太坊实现go-ethereum来和以太坊区块链进行交互。Go-ethereum，也被简称为Geth，是最流行的以太坊客户端。因为它是用Go开发的，当使用Golang开发应用程序时，Geth提供了读写区块链的一切功能。

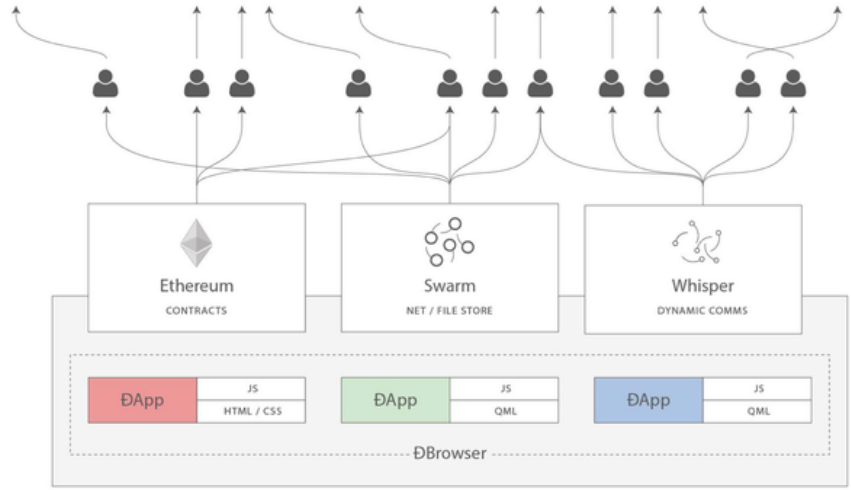
本书的例子在go-ethereum版本`1.8.10-stable`和Go版本`go1.10.2`下完成测试。

Block Explorers

Etherscan是一个用于查看和深入研究区块链上数据的网站。这些类型的网站被称为*区块浏览器*，因为它们允许您查看区块（包含交易）的内容。区块是区块链的基础构成要素。区块包含在已分配的出块时间内开采出的所有交易数据。区块浏览器也允许您查看智能合约执行期间释放的事件以及诸如支付的gas和交易的以太币数量等。

Swarm and Whisper

我们还将深入研究蜂群(Swarm)和耳语(Whisper)，分别是一个文件存储协议和一个点对点的消息传递协议，它们是实现完全去中心化和分布式应用程序需要的另外两个核心。



图片来源

寻求帮助

寻求Go(Golang)帮助可以加入gophers slack)上的#ethereum)频道。

介绍部分足够了，让我们开始吧。

客户端

客户端是以太坊网络的入口。客户端需要广播交易和读取区块链数据。在下一节中将学习如何在Go应用程序中初始化客户端。

初始化客户端

用Go初始化以太坊客户端是和区块链交互所需的基本步骤。首先，导入go-etherem的`ethclient`包并通过调用接收区块链服务提供者URL的`Dial`来初始化它。

若您没有现有以太坊客户端，您可以连接到infura网关。Infura管理着一批安全，可靠，可扩展的以太坊[geth和parity]节点，并且在接入以太坊网络时降低了新人的入门门槛。

```
client, err := ethclient.Dial("https://mainnet.infura.io")
```

若您运行了本地geth实例，您还可以将路径传递给IPC端点文件。

```
client, err := ethclient.Dial("/home/user/.ethereum/geth.ipc")
```

对每个Go以太坊项目，使用ethclient是您开始的必要事项，您将在本书中非常多的看到这一步骤。

使用Ganache

Ganache(正式名称为testrpc)是一个用Node.js编写的以太坊实现，用于在本地开发去中心化应用程序时进行测试。现在我们将带着您完成安装并连接到它。

首先通过NPM安装ganache。

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

然后运行ganache cli客户端。

```
ganache-cli
```

现在连到`http://localhost:8584`上的ganache RPC主机。

```
client, err := ethclient.Dial("http://localhost:8545")
if err != nil {
  log.Fatal(err)
}
```

在启动ganache时，您还可以使用相同的助记词来生成相同序列的公开地址。

```
ganache-cli -m "much repair shock carbon improve miss forget sock include
```

我强烈推荐您通过阅读其文档熟悉ganache。

完整代码

client.go

```
package main

import (
    "fmt"
    "log"

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

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("we have a connection")
    _ = client // we'll use this in the upcoming sections
}
```

账户

以太坊上的账户要么是钱包地址要么是智能合约地址。它们看起来像是 `0x71c7656ec7ab88b098defb751b7401b5f6d8976f`，它们用于将ETH发送到另一个用户，并且还用于在需要和区块链交互时指一个智能合约。它们是唯一的，且是从私钥导出的。我们将在后面的章节更深入地介绍公私钥对。

要使用go-ethereum的账户地址，您必须先将它们转化为go-ethereum中的 `common.Address` 类型。

```
address := common.HexToAddress("0x71c7656ec7ab88b098defb751b7401

fmt.Println(address.Hex()) // 0x71C7656EC7ab88b098defB751B7401B5f6d8
```

您可以在几乎任何地方使用这种类型，您可以将以太坊地址传递给go-ethereum的方法。既然您已经了解账户和地址的基础知识，那么让我们在下一节中学习如何检索ETH账户余额。

完整代码

address.go

```
package main

import (
    "fmt"

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

func main() {
    address := common.HexToAddress("0x71c7656ec7ab88b098defb751b740

    fmt.Println(address.Hex())       // 0x71C7656EC7ab88b098defB751B7401B
    fmt.Println(address.Hash().Hex()) // 0x000000000000000000000000071c76
    fmt.Println(address.Bytes())     // [113 199 101 110 199 171 136 176 152 2
}
```

账户余额

读取一个账户的余额相当简单。调用客户端的 `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。

```
account := common.HexToAddress("0x71c7656ec7ab88b098defb751b7401
balance, err := client.BalanceAt(context.Background(), account, nil)
if err != nil {
  log.Fatal(err)
}

fmt.Println(balance) // 25893180161173005034
```

传区块号能让您读取该区块时的账户余额。区块号必须是 `big.Int` 类型。

```
blockNumber := big.NewInt(5532993)
balance, err := client.BalanceAt(context.Background(), account, blockNumbe
if err != nil {
  log.Fatal(err)
}

fmt.Println(balance) // 25729324269165216042
```

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。

```
fbalance := new(big.Float)
fbalance.SetString(balance.String())
ethValue := new(big.Float).Quo(fbalance, big.NewFloat(math.Pow10(18)))

fmt.Println(ethValue) // 25.729324269165216041
```

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。

```
pendingBalance, err := client.PendingBalanceAt(context.Background(), acco
fmt.Println(pendingBalance) // 25729324269165216042
```

完整代码

account_balance.go

```
package main

import (
    "context"
    "fmt"
    "log"
    "math"
    "math/big"

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

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    account := common.HexToAddress("0x71c7656ec7ab88b098defb751b740
    balance, err := client.BalanceAt(context.Background(), account, nil)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(balance) // 25893180161173005034

    blockNumber := big.NewInt(5532993)
    balanceAt, err := client.BalanceAt(context.Background(), account, blockNu
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(balanceAt) // 25729324269165216042

    fbalance := new(big.Float)
    fbalance.SetString(balanceAt.String())
    ethValue := new(big.Float).Quo(fbalance, big.NewFloat(math.Pow10(18)))
    fmt.Println(ethValue) // 25.729324269165216041

    pendingBalance, err := client.PendingBalanceAt(context.Background(), ac
    fmt.Println(pendingBalance) // 25729324269165216042
}
```

账户代币余额

要学习如何读取账户代币(ECR20)余额，请前往ECR20代币智能合约章节。

生成新钱包

要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。

```
privateKey, err := crypto.GenerateKey()
if err != nil {
  log.Fatal(err)
}
```

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。

```
privateKeyBytes := crypto.FromECDSA(privateKey)
```

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”。

```
fmt.Println(hexutil.Encode(privateKeyBytes)[2:]) // fad9c8855b740a0b7ed4c
```

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` ，它始终是EC前缀，不是必需的。

```
publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
if !ok {
  log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey")
}

publicKeyBytes := crypto.FromECDSAPub(publicKeyECDSA)
fmt.Println(hexutil.Encode(publicKeyBytes)[4:]) // 9a7df67f79246283fdc93af
```

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。

```
address := crypto.PubkeyToAddress(*publicKeyECDSA).Hex()
fmt.Println(address) // 0x96216849c49358B10257cb55b28eA603c874b05E
```

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。

```
hash := sha3.NewLegacyKeccak256()
hash.Write(publicKeyBytes[1:])
fmt.Println(hexutil.Encode(hash.Sum(nil)[12:])) // 0x96216849c49358b10257
```

完整代码

generate_wallet.go

```
package main

import (
    "crypto/ecdsa"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/crypto"
    "golang.org/x/crypto/sha3"
)

func main() {
    privateKey, err := crypto.GenerateKey()
    if err != nil {
        log.Fatal(err)
    }

    privateKeyBytes := crypto.FromECDSA(privateKey)
    fmt.Println(hexutil.Encode(privateKeyBytes)[2:]) // 0xfad9c8855b740a0b7

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey
    }

    publicKeyBytes := crypto.FromECDSAPub(publicKeyECDSA)
    fmt.Println(hexutil.Encode(publicKeyBytes)[4:]) // 0x049a7df67f79246283f

    address := crypto.PubkeyToAddress(*publicKeyECDSA).Hex()
    fmt.Println(address) // 0x96216849c49358B10257cb55b28eA603c874b05

    hash := sha3.NewLegacyKeccak256()
    hash.Write(publicKeyBytes[1:])
    fmt.Println(hexutil.Encode(hash.Sum(nil)[12:])) // 0x96216849c49358b102
}
```

Keystores

keystore是一个包含经过加密了的钱包私钥。go-ethereum中的keystore，每个文件只能包含一个钱包密钥对。要生成keystore，首先您必须调用 NewKeyStore ，给它提供保存keystore的目录路径。然后，您可调用 NewAccount 方法创建新的钱包，并给它传入一个用于加密的口令。您每次调用 NewAccount ，它将在磁盘上生成新的keystore文件。

这是一个完整的生成新的keystore账户的示例。

```
ks := keystore.NewKeyStore("./wallets", keystore.StandardScryptN, keystore
password := "secret"
account, err := ks.NewAccount(password)
if err != nil {
  log.Fatal(err)
}

fmt.Println(account.Address.Hex()) // 0x20F8D42FB0F667F2E53930fed426f2
```

现在要导入您的keystore，您基本上像往常一样再次调用 NewKeyStore ，然后调用 Import 方法，该方法接收keystore的JSON数据作为字节。第二个参数是用于加密私钥的口令。第三个参数是指定一个新的加密口令，但我们在示例中使用一样的口令。导入账户将允许您按期访问该账户，但它将生成新keystore文件！有两个相同的事物是没有意义的，所以我们将删除旧的。

这是一个导入keystore和访问账户的示例。

```
file := "./wallets/UTC--2018-07-04T09-58-30.122808598Z--20f8d42fb0f667f2e
ks := keystore.NewKeyStore("./tmp", keystore.StandardScryptN, keystore.Sta
jsonBytes, err := ioutil.ReadFile(file)
if err != nil {
  log.Fatal(err)
}

password := "secret"
account, err := ks.Import(jsonBytes, password, password)
if err != nil {
  log.Fatal(err)
}

fmt.Println(account.Address.Hex()) // 0x20F8D42FB0F667F2E53930fed426f2

if err := os.Remove(file); err != nil {
  log.Fatal(err)
}
```

完整代码

keystore.go

```
package main

import (
    "fmt"
    "io/ioutil"
    "log"
    "os"

    "github.com/ethereum/go-ethereum/accounts/keystore"
)

func createKs() {
    ks := keystore.NewKeyStore("./tmp", keystore.StandardScryptN, keystore.
    password := "secret"
    account, err := ks.NewAccount(password)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(account.Address.Hex()) // 0x20F8D42FB0F667F2E53930fed426
}

func importKs() {
    file := "./tmp/UTC--2018-07-04T09-58-30.122808598Z--20f8d42fb0f667f2e
    ks := keystore.NewKeyStore("./tmp", keystore.StandardScryptN, keystore.
    jsonBytes, err := ioutil.ReadFile(file)
    if err != nil {
        log.Fatal(err)
    }

    password := "secret"
    account, err := ks.Import(jsonBytes, password, password)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(account.Address.Hex()) // 0x20F8D42FB0F667F2E53930fed426

    if err := os.Remove(file); err != nil {
        log.Fatal(err)
    }
}

func main() {
    createKs()
    //importKs()
}
```

分层确定性(HD)Wallet

关于创建或使用一个分层确定性(HD)钱包，请参考Go包:
https://github.com/miguelmota/go-ethereum-hdwallet

地址检查

本节将介绍如何确认一个地址并确定其是否为智能合约地址。

检查地址是否有效

我们可以使用简单的正则表达式来检查以太坊地址是否有效：

```
re := regexp.MustCompile("^0x[0-9a-fA-F]{40}$")

fmt.Printf("is valid: %v\n", re.MatchString("0x323b5d4c32345ced77393b353
fmt.Printf("is valid: %v\n", re.MatchString("0xZYXb5d4c32345ced77393b353
```

检查地址是否为账户或智能合约

我们可以确定，若在该地址存储了字节码，该地址是智能合约。这是一个示例，在例子中，我们获取一个代币智能合约的字节码并检查其长度以验证它是一个智能合约：

```
// 0x Protocol Token (ZRX) smart contract address
address := common.HexToAddress("0xe41d2489571d322189246dafa5ebde
bytecode, err := client.CodeAt(context.Background(), address, nil) // nil is lat
if err != nil {
  log.Fatal(err)
}

isContract := len(bytecode) > 0

fmt.Printf("is contract: %v\n", isContract) // is contract: true
```

当地址上没有字节码时，我们知道它不是一个智能合约，它是一个标准的以太坊账户。

```
// a random user account address
address := common.HexToAddress("0x8e215d06ea7ec1fdb4fc5fd21768f4b3
bytecode, err := client.CodeAt(context.Background(), address, nil) // nil is lat
if err != nil {
  log.Fatal(err)
}

isContract = len(bytecode) > 0

fmt.Printf("is contract: %v\n", isContract) // is contract: false
```

完整代码

address_check.go

```
package main

import (
    "context"
    "fmt"
    "log"
    "regexp"

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

func main() {
    re := regexp.MustCompile("^0x[0-9a-fA-F]{40}$")

    fmt.Printf("is valid: %v\n", re.MatchString("0x323b5d4c32345ced77393b3
    fmt.Printf("is valid: %v\n", re.MatchString("0xZYXb5d4c32345ced77393b3

    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    // 0x Protocol Token (ZRX) smart contract address
    address := common.HexToAddress("0xe41d2489571d322189246dafa5eb
    bytecode, err := client.CodeAt(context.Background(), address, nil) // nil is l
    if err != nil {
        log.Fatal(err)
    }

    isContract := len(bytecode) > 0

    fmt.Printf("is contract: %v\n", isContract) // is contract: true

    // a random user account address
    address = common.HexToAddress("0x8e215d06ea7ec1fdb4fc5fd21768f4
    bytecode, err = client.CodeAt(context.Background(), address, nil) // nil is la
    if err != nil {
        log.Fatal(err)
    }

    isContract = len(bytecode) > 0

    fmt.Printf("is contract: %v\n", isContract) // is contract: false
}
```

交易(Transaction)

这些部分将讨论如何使用go-ethereum ethclient 包在以太坊上查询和发送交易。注意这里的交易 transaction 是指广义的对以太坊状态的更改，它既可以指具体的以太币转账，代币的转账，或者其他对智能合约的创建或者调用。而不仅仅是传统意义的买卖交易。

查询区块

正如我们所见，您可以有两种方式查询区块信息。

区块头

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

```
header, err := client.HeaderByNumber(context.Background(), nil)
if err != nil {
  log.Fatal(err)
}

fmt.Println(header.Number.String()) // 5671744
```

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。

```
blockNumber := big.NewInt(5671744)
block, err := client.BlockByNumber(context.Background(), blockNumber)
if err != nil {
  log.Fatal(err)
}

fmt.Println(block.Number().Uint64())     // 5671744
fmt.Println(block.Time().Uint64())       // 1527211625
fmt.Println(block.Difficulty().Uint64()) // 3217000136609065
fmt.Println(block.Hash().Hex())          // 0x9e8751ebb5069389b855bba72d94
fmt.Println(len(block.Transactions()))   // 144
```

调用 `Transaction` 只返回一个区块的交易数目。

```
count, err := client.TransactionCount(context.Background(), block.Hash())
if err != nil {
  log.Fatal(err)
}

fmt.Println(count) // 144
```

在下个章节，我们将学习查询区块中的交易。

完整代码

blocks.go

```
package main

import (
    "context"
    "fmt"
    "log"
    "math/big"

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

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    header, err := client.HeaderByNumber(context.Background(), nil)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(header.Number.String()) // 5671744

    blockNumber := big.NewInt(5671744)
    block, err := client.BlockByNumber(context.Background(), blockNumber)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(block.Number().Uint64())     // 5671744
    fmt.Println(block.Time().Uint64())       // 1527211625
    fmt.Println(block.Difficulty().Uint64()) // 3217000136609065
    fmt.Println(block.Hash().Hex())          // 0x9e8751ebb5069389b855bba72d9
    fmt.Println(len(block.Transactions()))   // 144

    count, err := client.TransactionCount(context.Background(), block.Hash())
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(count) // 144
}
```

查询交易

在上个章节 我们学习了如何在给定区块编号的情况下读取块及其所有数据。 我们可以通过调用 Transactions 方法来读取块中的事务，该方法返回一个 Transaction 类型的列表。 然后，重复遍历集合并获取有关事务的任何信息就变得简单了。

```
for _, tx := range block.Transactions() {
  fmt.Println(tx.Hash().Hex())       // 0x5d49fcaa394c97ec8a9c3e7bd9e8388d
  fmt.Println(tx.Value().String())   // 10000000000000000
  fmt.Println(tx.Gas())              // 105000
  fmt.Println(tx.GasPrice().Uint64()) // 102000000000
  fmt.Println(tx.Nonce())            // 110644
  fmt.Println(tx.Data())             // []
  fmt.Println(tx.To().Hex())         // 0x55fE59D8Ad77035154dDd0AD0388D09D
}
```

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，这个我们从客户端拿到链ID。

```
chainID, err := client.NetworkID(context.Background())
if err != nil {
  log.Fatal(err)
}

if msg, err := tx.AsMessage(types.NewEIP155Signer(chainID)); err != nil {
  fmt.Println(msg.From().Hex()) // 0x0fD081e3Bb178dc45c0cb23202069ddA5
}
```

每个事务都有一个收据，其中包含执行事务的结果，例如任何返回值和日志，以及为“1”（成功）或“0”（失败）的事件结果状态。

```
receipt, err := client.TransactionReceipt(context.Background(), tx.Hash())
if err != nil {
  log.Fatal(err)
}

fmt.Println(receipt.Status) // 1
fmt.Println(receipt.Logs) // ...
```

在不获取块的情况下遍历事务的另一种方法是调用客户端的 TransactionInBlock 方法。 此方法仅接受块哈希和块内事务的索引值。您可以调用 TransactionCount 来了解块中有多少个事务。

```
blockHash := common.HexToHash("0x9e8751ebb5069389b855bba72d9490
count, err := client.TransactionCount(context.Background(), blockHash)
if err != nil {
  log.Fatal(err)
}

for idx := uint(0); idx < count; idx++ {
  tx, err := client.TransactionInBlock(context.Background(), blockHash, idx)
  if err != nil {
    log.Fatal(err)
  }

  fmt.Println(tx.Hash().Hex()) // 0x5d49fcaa394c97ec8a9c3e7bd9e8388d420f
}
```

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

```
txHash := common.HexToHash("0x5d49fcaa394c97ec8a9c3e7bd9e8388d42
tx, isPending, err := client.TransactionByHash(context.Background(), txHash
if err != nil {
  log.Fatal(err)
}

fmt.Println(tx.Hash().Hex()) // 0x5d49fcaa394c97ec8a9c3e7bd9e8388d420fb
fmt.Println(isPending)       // false
```

完整代码

transactions.go

```
package main

import (
    "context"
    "fmt"
    "log"
    "math/big"

    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    blockNumber := big.NewInt(5671744)
    block, err := client.BlockByNumber(context.Background(), blockNumber)
    if err != nil {
        log.Fatal(err)
    }

    for _, tx := range block.Transactions() {
        fmt.Println(tx.Hash().Hex())       // 0x5d49fcaa394c97ec8a9c3e7bd9e838
        fmt.Println(tx.Value().String())   // 10000000000000000
        fmt.Println(tx.Gas())              // 105000
        fmt.Println(tx.GasPrice().Uint64()) // 102000000000
        fmt.Println(tx.Nonce())            // 110644
        fmt.Println(tx.Data())             // []
        fmt.Println(tx.To().Hex())         // 0x55fE59D8Ad77035154dDd0AD0388D

        chainID, err := client.NetworkID(context.Background())
        if err != nil {
            log.Fatal(err)
        }

        if msg, err := tx.AsMessage(types.NewEIP155Signer(chainID)); err == ni
            fmt.Println(msg.From().Hex()) // 0x0fD081e3Bb178dc45c0cb2320206
        }

        receipt, err := client.TransactionReceipt(context.Background(), tx.Hash()
        if err != nil {
            log.Fatal(err)
        }
```

```
    fmt.Println(receipt.Status) // 1
}

blockHash := common.HexToHash("0x9e8751ebb5069389b855bba72d94
count, err := client.TransactionCount(context.Background(), blockHash)
if err != nil {
    log.Fatal(err)
}

for idx := uint(0); idx < count; idx++ {
    tx, err := client.TransactionInBlock(context.Background(), blockHash, id
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(tx.Hash().Hex()) // 0x5d49fcaa394c97ec8a9c3e7bd9e8388d4
}

txHash := common.HexToHash("0x5d49fcaa394c97ec8a9c3e7bd9e8388d
tx, isPending, err := client.TransactionByHash(context.Background(), txHa
if err != nil {
    log.Fatal(err)
}

fmt.Println(tx.Hash().Hex()) // 0x5d49fcaa394c97ec8a9c3e7bd9e8388d420
fmt.Println(isPending)       // false
}
```

转账以太币ETH

在本课程中，您将学习如何将ETH从一个帐户转移到另一个帐户。如果您已熟悉以太坊，那么您就知道如何交易包括您打算转账的以太币数量量，燃气限额，燃气价格，一个随机数(nonce)，接收地址以及可选择性的添加的数据。 在广告发送到网络之前，必须使用发送方的私钥对该交易进行签名。

假设您已经连接了客户端，下一步就是加载您的私钥。

```
privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad0f3
if err != nil {
  log.Fatal(err)
}
```

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。

```
publicKey := privateKey.Public()
publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
if !ok {
  log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey")
}

fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
```

接下来我们可以读取我们应该用于帐户交易的随机数。

```
nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
if err != nil {
  log.Fatal(err)
}
```

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。 这里有一个小工具可以帮助您在ETH和wei之

间进行转换: https://etherconverter.netlify.com

```
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。

```
toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4fa1
```

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。

```
tx := types.NewTransaction(nonce, toAddress, value, gasLimit, gasPrice, nil)
```

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。

```
chainID, err := client.NetworkID(context.Background())
if err != nil {
  log.Fatal(err)
}

signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), privateKe
if err != nil {
  log.Fatal(err)
}
```

现在我们终于准备通过在客户端上调用“SendTransaction”来将已签名的事务广播到整个网络。

```
err = client.SendTransaction(context.Background(), signedTx)
if err != nil {
  log.Fatal(err)
}

fmt.Printf("tx sent: %s", signedTx.Hash().Hex()) // tx sent: 0x77006fcb3938f6
```

然后你可以去Etherscan看交易的确认过程:
https://rinkeby.etherscan.io/tx/0x77006fcb3938f648e2cc65bafd27dec30b9bfbe9df41f78498b9c8b7322a249e

---

完整代码

transfer_eth.go

```
package main

import (
    "context"
    "crypto/ecdsa"
    "fmt"
    "log"
    "math/big"

    "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"
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad
    if err != nil {
        log.Fatal(err)
    }

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey
    }

    fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
    nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
    if err != nil {
        log.Fatal(err)
    }

    value := big.NewInt(1000000000000000000) // in wei (1 eth)
    gasLimit := uint64(21000)                // in units
    gasPrice, err := client.SuggestGasPrice(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4f
    var data []byte
    tx := types.NewTransaction(nonce, toAddress, value, gasLimit, gasPrice, d
```

```
    chainID, err := client.NetworkID(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), private
    if err != nil {
        log.Fatal(err)
    }

    err = client.SendTransaction(context.Background(), signedTx)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("tx sent: %s", signedTx.Hash().Hex())
}
```

代币的转账

本节将向你介绍如何转移ERC-20代币。 了解如何转移非ERC-20兼容的其他类型的代币请查阅智能合约的章节 来了解如何与智能合约交互。

假设您已连接客户端，加载私钥并配置燃气价格，下一步是设置具体的交易数据字段。 如果你完全不明白我刚讲的这些，请先复习 以太币转账的章节。

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。

```
toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4fa1
```

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(HelloToken HTN)，这个可以用代币工厂服务来完成https://tokenfactory.surge.sh, 代币我部署到了Rinkeby测试网。

让我们将代币合约地址分配给变量。

```
tokenAddress := common.HexToAddress("0x28b149020d2152179873ec60b
```

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。

```
transferFnSignature := []byte("transfer(address,uint256)")
```

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。

```
hash := sha3.NewKeccak256()
hash.Write(transferFnSignature)
methodID := hash.Sum(nil)[:4]
fmt.Println(hexutil.Encode(methodID)) // 0xa9059cbb
```

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。

```
paddedAddress := common.LeftPadBytes(toAddress.Bytes(), 32)
fmt.Println(hexutil.Encode(paddedAddress)) // 0x000000000000000000000
```

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。

```
amount := new(big.Int)
amount.SetString("1000000000000000000000", 10) // 1000 tokens
```

代币量也需要左填充到32个字节。

```
paddedAmount := common.LeftPadBytes(amount.Bytes(), 32)
fmt.Println(hexutil.Encode(paddedAmount))  // 0x00000000000000000000
```

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。

```
var data []byte
data = append(data, methodID...)
data = append(data, paddedAddress...)
data = append(data, paddedAmount...)
```

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。

```
gasLimit, err := client.EstimateGas(context.Background(), ethereum.CallMsg
  To:   &toAddress,
  Data: data,
})
if err != nil {
  log.Fatal(err)
}

fmt.Println(gasLimit) // 23256
```

接下来我们需要做的是构建交易事务类型，这类似于您在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。

```
tx := types.NewTransaction(nonce, tokenAddress, value, gasLimit, gasPrice,
```

下一步是使用发件人的私钥对事务进行签名。 `SignTx`方法需要EIP155签名器(EIP155 signer)，这需要我们从客户端拿到链ID。

```
chainID, err := client.NetworkID(context.Background())
if err != nil {
  log.Fatal(err)
}

signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), privateKe
if err != nil {
  log.Fatal(err)
}
```

最后广播交易。

```
err = client.SendTransaction(context.Background(), signedTx)
if err != nil {
  log.Fatal(err)
}

fmt.Printf("tx sent: %s", signedTx.Hash().Hex()) // tx sent: 0xa56316b637a94
```

你可以去Etherscan看交易的确认过程:
https://rinkeby.etherscan.io/tx/0xa56316b637a94c4cc0331c73ef26389d6c097506d581073f927275e7a6ece0bc

要了解更多如何加载ERC20智能合约并与之互动的内容，可以查看
ERC20代币的智能合约章节.

完整代码

transfer_tokens.go

```
package main

import (
    "context"
    "crypto/ecdsa"
    "fmt"
    "log"
    "math/big"

    "github.com/ethereum/go-ethereum"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/crypto"
    "github.com/ethereum/go-ethereum/crypto/sha3"
    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad
    if err != nil {
        log.Fatal(err)
    }

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey
    }

    fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
    nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
    if err != nil {
        log.Fatal(err)
    }

    value := big.NewInt(0) // in wei (0 eth)
    gasPrice, err := client.SuggestGasPrice(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4f
```

```
tokenAddress := common.HexToAddress("0x28b149020d2152179873ec6(

transferFnSignature := []byte("transfer(address,uint256)")
hash := sha3.NewKeccak256()
hash.Write(transferFnSignature)
methodID := hash.Sum(nil)[:4]
fmt.Println(hexutil.Encode(methodID)) // 0xa9059cbb

paddedAddress := common.LeftPadBytes(toAddress.Bytes(), 32)
fmt.Println(hexutil.Encode(paddedAddress)) // 0x000000000000000000000

amount := new(big.Int)
amount.SetString("10000000000000000000", 10) // 1000 tokens
paddedAmount := common.LeftPadBytes(amount.Bytes(), 32)
fmt.Println(hexutil.Encode(paddedAmount)) // 0x000000000000000000000

var data []byte
data = append(data, methodID...)
data = append(data, paddedAddress...)
data = append(data, paddedAmount...)

gasLimit, err := client.EstimateGas(context.Background(), ethereum.CallM
    To:   &toAddress,
    Data: data,
})
if err != nil {
    log.Fatal(err)
}
fmt.Println(gasLimit) // 23256

tx := types.NewTransaction(nonce, tokenAddress, value, gasLimit, gasPric

chainID, err := client.NetworkID(context.Background())
if err != nil {
    log.Fatal(err)
}

signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), private
if err != nil {
    log.Fatal(err)
}

err = client.SendTransaction(context.Background(), signedTx)
if err != nil {
    log.Fatal(err)
}
```

```
    fmt.Printf("tx sent: %s", signedTx.Hash().Hex()) // tx sent: 0xa56316b637a
}
```

订阅新区块

在本节中，我们将讨论如何设置订阅以便在新区块被开采时获取事件。首先，我们需要一个支持websocket RPC的以太坊服务提供者。在示例中，我们将使用infura 的websocket端点。

```
client, err := ethclient.Dial("wss://ropsten.infura.io/ws")
if err != nil {
  log.Fatal(err)
}
```

接下来，我们将创建一个新的通道，用于接收最新的区块头。

```
headers := make(chan *types.Header)
```

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。

```
sub, err := client.SubscribeNewHead(context.Background(), headers)
if err != nil {
  log.Fatal(err)
}
```

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。

```
for {
  select {
  case err := <-sub.Err():
    log.Fatal(err)
  case header := <-headers:
    fmt.Println(header.Hash().Hex()) // 0xbc10defa8dda384c96a17640d84de5
  }
}
```

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。

```
block, err := client.BlockByHash(context.Background(), header.Hash())
if err != nil {
  log.Fatal(err)
}

fmt.Println(block.Hash().Hex())       // 0xbc10defa8dda384c96a17640d84de5
fmt.Println(block.Number().Uint64())   // 3477413
fmt.Println(block.Time().Uint64())     // 1529525947
fmt.Println(block.Nonce())            // 130524141876765836
fmt.Println(len(block.Transactions())) // 7
```

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。

完整代码

block_subscribe.go

```
package main

import (
    "context"
    "fmt"
    "log"

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

func main() {
    client, err := ethclient.Dial("wss://ropsten.infura.io/ws")
    if err != nil {
        log.Fatal(err)
    }

    headers := make(chan *types.Header)
    sub, err := client.SubscribeNewHead(context.Background(), headers)
    if err != nil {
        log.Fatal(err)
    }

    for {
        select {
        case err := <-sub.Err():
            log.Fatal(err)
        case header := <-headers:
            fmt.Println(header.Hash().Hex()) // 0xbc10defa8dda384c96a17640d8

            block, err := client.BlockByHash(context.Background(), header.Hash()
            if err != nil {
                log.Fatal(err)
            }

            fmt.Println(block.Hash().Hex())       // 0xbc10defa8dda384c96a17640c
            fmt.Println(block.Number().Uint64())  // 3477413
            fmt.Println(block.Time().Uint64())    // 1529525947
            fmt.Println(block.Nonce())            // 130524141876765836
            fmt.Println(len(block.Transactions())) // 7
        }
    }
}
```

构建原始交易（Raw Transaction）

如果你看过上个章节, 那么你知道如何加载你的私钥来签名交易。 我们现在假设你知道如何做到这一点，现在你想让原始交易数据能够在以后广播它。

首先构造事务对象并对其进行签名，例如：

```
tx := types.NewTransaction(nonce, toAddress, value, gasLimit, gasPrice, dat

signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), privateKe
if err != nil {
  log.Fatal(err)
}
```

现在，在我们以原始字节格式获取事务之前，我们需要初始化一个 types.Transactions 类型，并将签名后的交易作为第一个值。

```
ts := types.Transactions{signedTx}
```

这样做的原因是因为 Transactions 类型提供了一个 GetRlp 方法，用于以RLP编码格式返回事务。 RLP是以太坊用于序列化对象的特殊编码方法。结果是原始字节。

```
rawTxBytes := ts.GetRlp(0)
```

最后，我们可以非常轻松地将原始字节转换为十六进制字符串。

```
rawTxHex := hex.EncodeToString(rawTxBytes)

fmt.Printf(rawTxHex)
// f86d8202b384773594008252089445 92d8f8d7b001e72cb26a73e4fa1806a
```

接下来，你就可以广播原始交易数据。在下一章 我们将学习如何广播一个原始交易。

完整代码

transaction_raw_create.go

```
package main

import (
    "context"
    "crypto/ecdsa"
    "encoding/hex"
    "fmt"
    "log"
    "math/big"

    "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"
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad
    if err != nil {
        log.Fatal(err)
    }

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey
    }

    fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
    nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
    if err != nil {
        log.Fatal(err)
    }

    value := big.NewInt(1000000000000000000) // in wei (1 eth)
    gasLimit := uint64(21000)                // in units
    gasPrice, err := client.SuggestGasPrice(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4f
    var data []byte
```

```
    tx := types.NewTransaction(nonce, toAddress, value, gasLimit, gasPrice, d

    chainID, err := client.NetworkID(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), private
    if err != nil {
        log.Fatal(err)
    }

    ts := types.Transactions{signedTx}
    rawTxBytes := ts.GetRlp(0)
    rawTxHex := hex.EncodeToString(rawTxBytes)

    fmt.Printf(rawTxHex) // f86...772
}
```

发送原始交易事务

在上个章节中 我们学会了如何创建原始事务。 现在，我们将学习如何将其广播到以太坊网络，以便最终被处理和被矿工打包到区块。

首先将原始事务十六进制解码为字节格式。

```
rawTx := "f86d8202b28477359400825208944592d8f8d7b001e72cb26a73e4

rawTxBytes, err := hex.DecodeString(rawTx)
```

接下来初始化一个新的 types.Transaction 指针并从go-ethereum rlp 包中调用 DecodeBytes ，将原始事务字节和指针传递给以太坊事务类型。 RLP是以太坊用于序列化和反序列化数据的编码方法。

```
tx := new(types.Transaction)
rlp.DecodeBytes(rawTxBytes, &tx)
```

现在，我们可以使用我们的以太坊客户端轻松地广播交易。

```
err := client.SendTransaction(context.Background(), tx)
if err != nil {
  log.Fatal(err)
}

fmt.Printf("tx sent: %s", tx.Hash().Hex()) // tx sent: 0xc429e5f128387d224ba8
```

然后你可以去Etherscan看交易的确认过程:
https://rinkeby.etherscan.io/tx/0xc429e5f128387d224ba8bed6885e86525e14bfdc2eb24b5e9c3351a1176fd81f

---

完整代码

transaction_raw_sendreate.go

```
package main

import (
    "context"
    "encoding/hex"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/ethclient"
    "github.com/ethereum/go-ethereum/rlp"
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    rawTx := "f86d8202b28477359400825208944592d8f8d7b001e72cb26a73

    rawTxBytes, err := hex.DecodeString(rawTx)

    tx := new(types.Transaction)
    rlp.DecodeBytes(rawTxBytes, &tx)

    err = client.SendTransaction(context.Background(), tx)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("tx sent: %s", tx.Hash().Hex()) // tx sent: 0xc429e5f128387d224b
}
```

智能合约

在下个章节中我们会介绍如何用Go来编译，部署，写入和读取智能合约。

智能合约的编译与ABI

与智能合约交互，我们要先生成相应智能合约的应用二进制接口ABI(application binary interface)，并把ABI编译成我们可以在Go应用中调用的格式。

第一步是安装 Solidity编译器 (`solc`).

Solc 在Ubuntu上有snapcraft包。

```
sudo snap install solc --edge
```

Solc在macOS上有Homebrew的包。

```
brew update
brew tap ethereum/ethereum
brew install solidity
```

其他的平台或者从源码编译的教程请查阅官方solidity文档install guide.

我们还得安装一个叫`abigen`的工具，来从solidity智能合约生成ABI。

假设您已经在计算机上设置了Go，只需运行以下命令即可安装`abigen`工具。

```
go get -u github.com/ethereum/go-ethereum
cd $GOPATH/src/github.com/ethereum/go-ethereum/
make
make devtools
```

我们将创建一个简单的智能合约来测试。 学习更复杂的智能合约，或者智能合约的开发的内容则超出了本书的范围。 我强烈建议您查看truffle framework 来学习开发和测试智能合约。

这里只是一个简单的合约，就是一个键/值存储，只有一个外部方法来设置任何人的键/值对。 我们还在设置值后添加了要发出的事件。

```
pragma solidity ^0.4.24;

contract Store {
 event ItemSet(bytes32 key, bytes32 value);

 string public version;
 mapping (bytes32 => bytes32) public items;

 constructor(string _version) public {
  version = _version;
 }

 function setItem(bytes32 key, bytes32 value) external {
  items[key] = value;
  emit ItemSet(key, value);
 }
}
```

虽然这个智能合约很简单，但它将适用于这个例子。

现在我们可以从一个solidity文件生成ABI。

```
solc --abi Store.sol
```

它会将其写入名为“Store_sol_Store.abi”的文件中

现在让我们用 abigen 将ABI转换为我们可以导入的Go文件。 这个新文件将包含我们可以用来与Go应用程序中的智能合约进行交互的所有可用方法。

```
abigen --abi=Store_sol_Store.abi --pkg=store --out=Store.go
```

为了从Go部署智能合约，我们还需要将solidity智能合约编译为EVM字节码。 EVM字节码将在事务的数据字段中发送。 在Go文件上生成部署方法需要bin文件。

```
solc --bin Store.sol
```

现在我们编译Go合约文件，其中包括deploy方法，因为我们包含了bin文件。

```
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out
```

在接下来的课程中，我们将学习如何部署智能合约，然后与之交互。

完整代码

Commands

```
go get -u github.com/ethereum/go-ethereum
cd $GOPATH/src/github.com/ethereum/go-ethereum/
make
make devtools

solc --abi Store.sol
solc --bin Store.sol
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out
```

Store.sol

```
pragma solidity ^0.4.24;

contract Store {
 event ItemSet(bytes32 key, bytes32 value);

 string public version;
 mapping (bytes32 => bytes32) public items;

 constructor(string _version) public {
  version = _version;
 }

 function setItem(bytes32 key, bytes32 value) external {
  items[key] = value;
  emit ItemSet(key, value);
 }
}
```

solc version used for these examples

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

部署智能合约

如果你还没看之前的章节，请先学习编译智能合约的章节因为这节内容，需要先了解如何将智能合约编译为Go文件。

假设你已经导入从 `abigen` 生成的新创建的Go包文件，并设置ethclient，加载您的私钥，下一步是创建一个有配置密匙的交易发送器(tansactor)。首先从go-ethereum导入 `accounts/abi/bind` 包，然后调用传入私钥的 `NewKeyedTransactor`。然后设置通常的属性，如nonce，燃气价格，燃气上线限制和ETH值。

```
auth := bind.NewKeyedTransactor(privateKey)
auth.Nonce = big.NewInt(int64(nonce))
auth.Value = big.NewInt(0)     // in wei
auth.GasLimit = uint64(300000) // in units
auth.GasPrice = gasPrice
```

如果你还记得上个章节的内容, 我们创建了一个非常简单的“Store”合约，用于设置和存储键/值对。 生成的Go合约文件提供了部署方法。 部署方法名称始终以单词*Deploy*开头，后跟合约名称，在本例中为*Store*。

deploy函数接受有密匙的事务处理器，ethclient，以及智能合约构造函数可能接受的任何输入参数。我们测试的智能合约接受一个版本号的字符串参数。 此函数将返回新部署的合约地址，事务对象，我们可以交互的合约实例，还有错误（如果有）。

```
input := "1.0"
address, tx, instance, err := store.DeployStore(auth, client, input)
if err != nil {
  log.Fatal(err)
}

fmt.Println(address.Hex())   // 0x147B8eb97fD247D06C4006D269c90C1908F
fmt.Println(tx.Hash().Hex()) // 0xdae8ba5444eefdc99f4d45cd0c4f24056cba6

_ = instance // will be using the instance in the 下个章节
```

就这么简单：）你可以用事务哈希来在Etherscan上查询合约的部署状态: https://rinkeby.etherscan.io/tx/0xdae8ba5444eefdc99f4d45cd0c4f24056cba6a02cefbf78066ef9f4188ff7dc0

完整代码

Commands

```
solc --abi Store.sol
solc --bin Store.sol
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out
```

Store.sol

```
pragma solidity ^0.4.24;

contract Store {
  event ItemSet(bytes32 key, bytes32 value);

  string public version;
  mapping (bytes32 => bytes32) public items;

  constructor(string _version) public {
    version = _version;
  }

  function setItem(bytes32 key, bytes32 value) external {
    items[key] = value;
    emit ItemSet(key, value);
  }
}
```

contract_deploy.go

```
package main

import (
    "context"
    "crypto/ecdsa"
    "fmt"
    "log"
    "math/big"

    "github.com/ethereum/go-ethereum/accounts/abi/bind"
    "github.com/ethereum/go-ethereum/crypto"
    "github.com/ethereum/go-ethereum/ethclient"

    store "./contracts" // for demo
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad
    if err != nil {
        log.Fatal(err)
    }

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey
    }

    fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
    nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
    if err != nil {
        log.Fatal(err)
    }

    gasPrice, err := client.SuggestGasPrice(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    auth := bind.NewKeyedTransactor(privateKey)
    auth.Nonce = big.NewInt(int64(nonce))
    auth.Value = big.NewInt(0)     // in wei
    auth.GasLimit = uint64(300000) // in units
```

```
    auth.GasPrice = gasPrice

    input := "1.0"
    address, tx, instance, err := store.DeployStore(auth, client, input)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(address.Hex())   // 0x147B8eb97fD247D06C4006D269c90C190
    fmt.Println(tx.Hash().Hex()) // 0xdae8ba5444eefdc99f4d45cd0c4f24056cb

    _ = instance
}
```

solc version used for these examples

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

加载智能合约

这写章节需要了解如何将智能合约的ABI编译成Go的合约文件。如果你还没看， 前先读上一个章节 。

一旦使用 abigen 工具将智能合约的ABI编译为Go包，下一步就是调用“New”方法，其格式为“New”，所以在我们的例子中如果你 回想一下它将是*NewStore*。此初始化方法接收智能合约的地址，并返回可以开始与之交互的合约实例。

```
address := common.HexToAddress("0x147B8eb97fD247D06C4006D269c90C
instance, err := store.NewStore(address, client)
if err != nil {
  log.Fatal(err)
}

_ = instance // we'll be using this in the 下个章节
```

完整代码

Commands

```
solc --abi Store.sol
solc --bin Store.sol
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out
```

Store.sol

```
pragma solidity ^0.4.24;

contract Store {
  event ItemSet(bytes32 key, bytes32 value);

  string public version;
  mapping (bytes32 => bytes32) public items;

  constructor(string _version) public {
    version = _version;
  }

  function setItem(bytes32 key, bytes32 value) external {
    items[key] = value;
    emit ItemSet(key, value);
  }
}
```

contract_load.go

```
package main

import (
    "fmt"
    "log"

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

    store "./contracts" // for demo
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    address := common.HexToAddress("0x147B8eb97fD247D06C4006D269c9
    instance, err := store.NewStore(address, client)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("contract is loaded")
    _ = instance
}
```

solc version used for these examples

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

Querying a Smart Contract

这写章节需要了解如何将智能合约的ABI编译成Go的合约文件。如果你还没看， 前先读上一个章节 。

在上个章节我们学习了如何在Go应用程序中初始化合约实例。 现在我们将使用新合约实例提供的方法来阅读智能合约。 如果你还记得我们在部署过程中设置的合约中有一个名为 version 的全局变量。 因为它是公开的，这意味着它们将成为我们自动创建的getter函数。 常量和view函数也接受 bind.CallOpts 作为第一个参数。了解可用的具体选项要看相应类的文档 一般情况下我们可以用 nil 。

```
version, err := instance.Version(nil)
if err != nil {
  log.Fatal(err)
}

fmt.Println(version) // "1.0"
```

完整代码

Commands

```
solc --abi Store.sol
solc --bin Store.sol
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out
```

Store.sol

```
pragma solidity ^0.4.24;

contract Store {
  event ItemSet(bytes32 key, bytes32 value);

  string public version;
  mapping (bytes32 => bytes32) public items;

  constructor(string _version) public {
    version = _version;
  }

  function setItem(bytes32 key, bytes32 value) external {
    items[key] = value;
    emit ItemSet(key, value);
  }
}
```

contract_read.go

```
package main

import (
    "fmt"
    "log"

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

    store "./contracts" // for demo
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    address := common.HexToAddress("0x147B8eb97fD247D06C4006D269c9
    instance, err := store.NewStore(address, client)
    if err != nil {
        log.Fatal(err)
    }

    version, err := instance.Version(nil)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(version) // "1.0"
}
```

solc version used for these examples

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

写入智能合约

这写章节需要了解如何将智能合约的ABI编译成Go的合约文件。如果你还没看， 前先读上一个章节 。

写入智能合约需要我们用私钥来对交易事务进行签名。

```
privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad0f3
if err != nil {
  log.Fatal(err)
}

publicKey := privateKey.Public()
publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
if !ok {
  log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey")
}

fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
```

我们还需要先查到nonce和燃气价格。

```
nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
if err != nil {
  log.Fatal(err)
}

gasPrice, err := client.SuggestGasPrice(context.Background())
if err != nil {
  log.Fatal(err)
}
```

接下来，我们创建一个新的keyed transactor，它接收私钥。

```
auth := bind.NewKeyedTransactor(privateKey)
```

然后我们需要设置keyed transactor的标准交易选项。

```
auth.Nonce = big.NewInt(int64(nonce))
auth.Value = big.NewInt(0)     // in wei
auth.GasLimit = uint64(300000) // in units
auth.GasPrice = gasPrice
```

现在我们加载一个智能合约的实例。如果你还记得上个章节 我们创建一个名为*Store*的合约，并使用 abigen 工具生成一个Go文件。 要初始化它，我们只需调用合约包的*New*方法，并提供智能合约地址和ethclient，它返回我们可以使用的合约实例。

```
address := common.HexToAddress("0x147B8eb97fD247D06C4006D269c90(
instance, err := store.NewStore(address, client)
if err != nil {
  log.Fatal(err)
}
```

我们创建的智能合约有一个名为*SetItem*的外部方法，它接受solidity“bytes32”格式的两个参数（key，value）。 这意味着Go合约包要求我们传递一个长度为32个字节的字节数组。 调用*SetItem*方法需要我们传递我们之前创建的 auth 对象（keyed transactor）。 在幕后，此方法将使用它的参数对此函数调用进行编码，将其设置为事务的 data 属性，并使用私钥对其进行签名。 结果将是一个已签名的事务对象。

```
key := [32]byte{}
value := [32]byte{}
copy(key[:], []byte("foo"))
copy(value[:], []byte("bar"))

tx, err := instance.SetItem(auth, key, value)
if err != nil {
  log.Fatal(err)
}

fmt.Printf("tx sent: %s", tx.Hash().Hex()) // tx sent: 0x8d490e535678e9a2436
```

现在我就可以看到交易已经成功被发送到了以太坊网络了: https://rinkeby.etherscan.io/tx/0x8d490e535678e9a24360e955d75b27ad307bdfb97a1dca51d0f3035dcee3e870

要验证键/值是否已设置，我们可以读取智能合约中的值。

```
result, err := instance.Items(nil, key)
if err != nil {
  log.Fatal(err)
}

fmt.Println(string(result[:])) // "bar"
```

搞定！

完整代码

Commands

```
solc --abi Store.sol
solc --bin Store.sol
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out
```

Store.sol

```
pragma solidity ^0.4.24;

contract Store {
  event ItemSet(bytes32 key, bytes32 value);

  string public version;
  mapping (bytes32 => bytes32) public items;

  constructor(string _version) public {
    version = _version;
  }

  function setItem(bytes32 key, bytes32 value) external {
    items[key] = value;
    emit ItemSet(key, value);
  }
}
```

contract_write.go

```
package main

import (
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/accounts/abi/bind"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/ethclient"

    store "./contracts" // for demo
)

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad
    if err != nil {
        log.Fatal(err)
    }

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey
    }

    fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
    nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
    if err != nil {
        log.Fatal(err)
    }

    gasPrice, err := client.SuggestGasPrice(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    auth := bind.NewKeyedTransactor(privateKey)
    auth.Nonce = big.NewInt(int64(nonce))
    auth.Value = big.NewInt(0)     // in wei
    auth.GasLimit = uint64(300000) // in units
    auth.GasPrice = gasPrice

    address := common.HexToAddress("0x147B8eb97fD247D06C4006D269c9
```

```
    instance, err := store.NewStore(address, client)
    if err != nil {
        log.Fatal(err)
    }

    key := [32]byte{}
    value := [32]byte{}
    copy(key[:], []byte("foo"))
    copy(value[:], []byte("bar"))

    tx, err := instance.SetItem(auth, key, value)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("tx sent: %s", tx.Hash().Hex()) // tx sent: 0x8d490e535678e9a24

    result, err := instance.Items(nil, key)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(string(result[:])) // "bar"
}
```

solc version used for these examples

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

读取智能合约的字节码

有时您需要读取已部署的智能合约的字节码。 由于所有智能合约字节码都存在于区块链中，因此我们可以轻松获取它。

首先设置客户端和要读取的字节码的智能合约地址。

```
client, err := ethclient.Dial("https://rinkeby.infura.io")
if err != nil {
  log.Fatal(err)
}

contractAddress := common.HexToAddress("0x147B8eb97fD247D06C4006D
```

现在你需要调用客户端的 `codeAt` 方法。 `codeAt` 方法接受智能合约地址和可选的块编号，并以字节格式返回字节码。

```
bytecode, err := client.CodeAt(context.Background(), contractAddress, nil) //
if err != nil {
  log.Fatal(err)
}

fmt.Println(hex.EncodeToString(bytecode)) // 60806...10029
```

你也可以在etherscan上查询16进制格式的字节码
https://rinkeby.etherscan.io/address/0x147b8eb97fd247d06c4006d269c90c1908fb5d54#code

---

完整代码

contract_bytecode.go

```
package main

import (
    "context"
    "encoding/hex"
    "fmt"
    "log"

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

func main() {
    client, err := ethclient.Dial("https://rinkeby.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    contractAddress := common.HexToAddress("0x147B8eb97fD247D06C400
    bytecode, err := client.CodeAt(context.Background(), contractAddress, nil)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(hex.EncodeToString(bytecode)) // 60806...10029
}
```

查询ERC20代币智能合约

首先创建一个ERC20智能合约interface。这只是与您可以调用的函数的函数定义的契约。

```
pragma solidity ^0.4.24;

contract ERC20 {
    string public constant name = "";
    string public constant symbol = "";
    uint8 public constant decimals = 0;

    function totalSupply() public constant returns (uint);
    function balanceOf(address tokenOwner) public constant returns (uint ba
    function allowance(address tokenOwner, address spender) public consta
    function transfer(address to, uint tokens) public returns (bool success);
    function approve(address spender, uint tokens) public returns (bool succ
    function transferFrom(address from, address to, uint tokens) public retur

    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, u
}
```

然后将interface智能合约编译为JSON ABI，并使用 `abigen` 从ABI创建Go包。

```
solc --abi erc20.sol
abigen --abi=erc20_sol_ERC20.abi --pkg=token --out=erc20.go
```

假设我们已经像往常一样设置了以太坊客户端，我们现在可以将新的 *token*包导入我们的应用程序并实例化它。这个例子里我们用Golem 代币的地址.

```
tokenAddress := common.HexToAddress("0xa74476443119A942dE498590F
instance, err := token.NewToken(tokenAddress, client)
if err != nil {
  log.Fatal(err)
}
```

我们现在可以调用任何ERC20的方法。 例如，我们可以查询用户的代币余额。

```
address := common.HexToAddress("0x0536806df512d6cdde913cf95c9886f
bal, err := instance.BalanceOf(&bind.CallOpts{}, address)
if err != nil {
  log.Fatal(err)
}

fmt.Printf("wei: %s\n", bal) // "wei: 74605500647408739782407023"
```

我们还可以读ERC20智能合约的公共变量。

```
name, err := instance.Name(&bind.CallOpts{})
if err != nil {
  log.Fatal(err)
}

symbol, err := instance.Symbol(&bind.CallOpts{})
if err != nil {
  log.Fatal(err)
}

decimals, err := instance.Decimals(&bind.CallOpts{})
if err != nil {
  log.Fatal(err)
}

fmt.Printf("name: %s\n", name)        // "name: Golem Network"
fmt.Printf("symbol: %s\n", symbol)    // "symbol: GNT"
fmt.Printf("decimals: %v\n", decimals) // "decimals: 18"
```

我们可以做一些简单的数学运算将余额转换为可读的十进制格式。

```
fbal := new(big.Float)
fbal.SetString(bal.String())
value := new(big.Float).Quo(fbal, big.NewFloat(math.Pow10(int(decimals))))

fmt.Printf("balance: %f", value) // "balance: 74605500.647409"
```

同样的信息也可以在etherscan上查询:
https://etherscan.io/token/0xa74476443119a942de498590fe1f2454d7d4ac0d?a=0x0536806df512d6cdde913cf95c9886f65b1d3462

完整代码

Commands

```
solc --abi erc20.sol
abigen --abi=erc20_sol_ERC20.abi --pkg=token --out=erc20.go
```

erc20.sol

```
pragma solidity ^0.4.24;

contract ERC20 {
    string public constant name = "";
    string public constant symbol = "";
    uint8 public constant decimals = 0;

    function totalSupply() public constant returns (uint);
    function balanceOf(address tokenOwner) public constant returns (uint ba
    function allowance(address tokenOwner, address spender) public consta
    function transfer(address to, uint tokens) public returns (bool success);
    function approve(address spender, uint tokens) public returns (bool succ
    function transferFrom(address from, address to, uint tokens) public retur

    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, u
}
```

contract_read_erc20.go

```
package main

import (
    "fmt"
    "log"
    "math"
    "math/big"

    "github.com/ethereum/go-ethereum/accounts/abi/bind"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/ethclient"

    token "./contracts_erc20" // for demo
)

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    // Golem (GNT) Address
    tokenAddress := common.HexToAddress("0xa74476443119A942dE49859
    instance, err := token.NewToken(tokenAddress, client)
    if err != nil {
        log.Fatal(err)
    }

    address := common.HexToAddress("0x0536806df512d6cdde913cf95c988
    bal, err := instance.BalanceOf(&bind.CallOpts{}, address)
    if err != nil {
        log.Fatal(err)
    }

    name, err := instance.Name(&bind.CallOpts{})
    if err != nil {
        log.Fatal(err)
    }

    symbol, err := instance.Symbol(&bind.CallOpts{})
    if err != nil {
        log.Fatal(err)
    }

    decimals, err := instance.Decimals(&bind.CallOpts{})
    if err != nil {
        log.Fatal(err)
    }
```

```
    fmt.Printf("name: %s\n", name)        // "name: Golem Network"
    fmt.Printf("symbol: %s\n", symbol)    // "symbol: GNT"
    fmt.Printf("decimals: %v\n", decimals) // "decimals: 18"

    fmt.Printf("wei: %s\n", bal) // "wei: 746055006474087397824070 23"

    fbal := new(big.Float)
    fbal.SetString(bal.String())
    value := new(big.Float).Quo(fbal, big.NewFloat(math.Pow10(int(decimals))

    fmt.Printf("balance: %f", value) // "balance: 74605500.647409"
}
```

solc version used for these examples

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

事件

智能合约具有在执行期间“发出”事件的能力。 事件在以太坊中也称为“日志”。事件的输出存储在日志部分下的事务处理中。 事件已经在以太坊智能合约中被广泛使用，以便在发生相对重要的动作时记录，特别是在代币合约（即ERC-20）中，以指示代币转账已经发生。 这些部分将引导您完成从区块链中读取事件以及订阅事件的过程，以便交易事务被矿工打包入块的时候及时收到通知。

订阅事件日志

为了订阅事件日志，我们需要做的第一件事就是拨打启用websocket的以太坊客户端。 幸运的是，Infura支持websockets。

```
client, err := ethclient.Dial("wss://rinkeby.infura.io/ws")
if err != nil {
  log.Fatal(err)
}
```

下一步是创建筛选查询。 在这个例子中，我们将阅读来自我们在之前课程中创建的示例合约中的所有事件。

```
contractAddress := common.HexToAddress("0x147B8eb97fD247D06C4006D
query := ethereum.FilterQuery{
  Addresses: []common.Address{contractAddress},
}
```

我们接收事件的方式是通过Go channel。让我们从go-ethereum `core/types` 包创建一个类型为 `Log` 的channel。

```
logs := make(chan types.Log)
```

现在我们所要做的就是通过从客户端调用 `SubscribeFilterLogs` 来订阅，它接收查询选项和输出通道。 这将返回包含unsubscribe和error方法的订阅结构。

```
sub, err := client.SubscribeFilterLogs(context.Background(), query, logs)
if err != nil {
  log.Fatal(err)
}
```

最后，我们要做的就是使用select语句设置一个连续循环来读入新的日志事件或订阅错误。

```
for {
  select {
  case err := <-sub.Err():
    log.Fatal(err)
  case vLog := <-logs:
    fmt.Println(vLog) // pointer to event log
  }
}
```

我们会在下个章节介绍如何解析日志。

完整代码

Commands

```
solc --abi Store.sol
solc --bin Store.sol
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out
```

Store.sol

```
pragma solidity ^0.4.24;

contract Store {
  event ItemSet(bytes32 key, bytes32 value);

  string public version;
  mapping (bytes32 => bytes32) public items;

  constructor(string _version) public {
    version = _version;
  }

  function setItem(bytes32 key, bytes32 value) external {
    items[key] = value;
    emit ItemSet(key, value);
  }
}
```

event_subscribe.go

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/ethclient"
)

func main() {
    client, err := ethclient.Dial("wss://rinkeby.infura.io/ws")
    if err != nil {
        log.Fatal(err)
    }

    contractAddress := common.HexToAddress("0x147B8eb97fD247D06C400
    query := ethereum.FilterQuery{
        Addresses: []common.Address{contractAddress},
    }

    logs := make(chan types.Log)
    sub, err := client.SubscribeFilterLogs(context.Background(), query, logs)
    if err != nil {
        log.Fatal(err)
    }

    for {
        select {
        case err := <-sub.Err():
            log.Fatal(err)
        case vLog := <-logs:
            fmt.Println(vLog) // pointer to event log
        }
    }
}
```

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

读取事件日志

智能合约可以可选地释放“事件”，其作为交易收据的一部分存储日志。读取这些事件相当简单。首先我们需要构造一个过滤查询。我们从go-ethereum包中导入 FilterQuery 结构体并用过滤选项初始化它。我们告诉它我们想过滤的区块范围并指定从中读取此日志的合约地址。在示例中，我们将从在智能合约章节)创建的智能合约中读取特定区块所有日志。

```
query := ethereum.FilterQuery{
  FromBlock: big.NewInt(2394201),
  ToBlock:   big.NewInt(2394201),
  Addresses: []common.Address{
    contractAddress,
  },
}
```

下一步是调用ethclient的 FilterLogs ，它接收我们的查询并将返回所有的匹配事件日志。

```
logs, err := client.FilterLogs(context.Background(), query)
if err != nil {
  log.Fatal(err)
}
```

返回的所有日志将是ABI编码，因此它们本身不会非常易读。为了解码日志，我们需要导入我们智能合约的ABI。为此，我们导入编译好的智能合约Go包，它将包含名称格式为 <Contract>ABI 的外部属性。之后，我们使用go-ethereum中的 accounts/abi 包的 abi.JSON 函数返回一个我们可以在Go应用程序中使用的解析过的ABI接口。

```
contractAbi, err := abi.JSON(strings.NewReader(string(store.StoreABI)))
if err != nil {
  log.Fatal(err)
}
```

现在我们可以通过日志进行迭代并将它们解码为我么可以使用的类型。若您回忆起我们的样例合约释放的日志在Solidity中是类型为 bytes32 ，那么Go中的等价物将是 [32]byte 。我们可以使用这些类型创建一个匿名结构体，并将指针作为第一个参数传递给解析后的ABI接口的 Unpack 函数，以解码原始的日志数据。第二个参数是我们尝试解码的事件名称，最后一个参数是编码的日志数据。

```
for _, vLog := range logs {
  event := struct {
    Key   [32]byte
    Value [32]byte
  }{}
  err := contractAbi.Unpack(&event, "ItemSet", vLog.Data)
  if err != nil {
    log.Fatal(err)
  }

  fmt.Println(string(event.Key[:]))   // foo
  fmt.Println(string(event.Value[:])) // bar
}
```

此外，日志结构体包含附加信息，例如，区块摘要，区块号和交易摘要。

```
fmt.Println(vLog.BlockHash.Hex()) // 0x3404b8c050aa0aacd0223e91b5c32fe
fmt.Println(vLog.BlockNumber)     // 2394201
fmt.Println(vLog.TxHash.Hex())    // 0x280201eda63c9ff6f305fcee51d5eb861
```

主题(Topics)

若您的solidity事件包含 indexed 事件类型，那么它们将成为*主题*而不是日志的数据属性的一部分。在solidity中您最多只能有4个主题，但只有3个可索引的事件类型。第一个主题总是事件的签名。我们的示例合约不包含可索引的事件，但如果它确实包含，这是如何读取事件主题。

```
var topics [4]string
for i := range vLog.Topics {
  topics[i] = vLog.Topics[i].Hex()
}

fmt.Println(topics[0]) // 0xe79e73da417710ae99aa2088575580a60415d359a
```

正如您所见，首个主题只是被哈希过的事件签名。

```
eventSignature := []byte("ItemSet(bytes32,bytes32)")
hash := crypto.Keccak256Hash(eventSignature)
fmt.Println(hash.Hex()) // 0xe79e73da417710ae99aa2088575580a60415d35
```

这就是阅读和解析日志的全部内容。要学习如何订阅日志，阅读上个章节)。

完整代码

命令

```
solc --abi Store.sol
solc --bin Store.sol
abigen --bin=Store_sol_Store.bin --abi=Store_sol_Store.abi --pkg=store --out
```

Store.sol

```
pragma solidity ^0.4.24;

contract Store {
  event ItemSet(bytes32 key, bytes32 value);

  string public version;
  mapping (bytes32 => bytes32) public items;

  constructor(string _version) public {
    version = _version;
  }

  function setItem(bytes32 key, bytes32 value) external {
    items[key] = value;
    emit ItemSet(key, value);
  }
}
```

event_read.go

```
package main

import (
    "context"
    "fmt"
    "log"
    "math/big"
    "strings"

    "github.com/ethereum/go-ethereum"
    "github.com/ethereum/go-ethereum/accounts/abi"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/crypto"
    "github.com/ethereum/go-ethereum/ethclient"

    store "./contracts" // for demo
)

func main() {
    client, err := ethclient.Dial("wss://rinkeby.infura.io/ws")
    if err != nil {
        log.Fatal(err)
    }

    contractAddress := common.HexToAddress("0x147B8eb97fD247D06C400
    query := ethereum.FilterQuery{
        FromBlock: big.NewInt(2394201),
        ToBlock:   big.NewInt(2394201),
        Addresses: []common.Address{
            contractAddress,
        },
    }

    logs, err := client.FilterLogs(context.Background(), query)
    if err != nil {
        log.Fatal(err)
    }

    contractAbi, err := abi.JSON(strings.NewReader(string(store.StoreABI)))
    if err != nil {
        log.Fatal(err)
    }

    for _, vLog := range logs {
        fmt.Println(vLog.BlockHash.Hex()) // 0x3404b8c050aa0aacd0223e91b5
        fmt.Println(vLog.BlockNumber)     // 2394201
        fmt.Println(vLog.TxHash.Hex())    // 0x280201eda63c9ff6f305fcee51d5e
```

```
        event := struct {
            Key   [32]byte
            Value [32]byte
        }{}
        err := contractAbi.Unpack(&event, "ItemSet", vLog.Data)
        if err != nil {
            log.Fatal(err)
        }

        fmt.Println(string(event.Key[:]))   // foo
        fmt.Println(string(event.Value[:])) // bar

        var topics [4]string
        for i := range vLog.Topics {
            topics[i] = vLog.Topics[i].Hex()
        }

        fmt.Println(topics[0]) // 0xe79e73da417710ae99aa2088575580a60415d
    }

    eventSignature := []byte("ItemSet(bytes32,bytes32)")
    hash := crypto.Keccak256Hash(eventSignature)
    fmt.Println(hash.Hex()) // 0xe79e73da417710ae99aa2088575580a60415d
}
```

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

读取ERC-20代币的事件日志

首先，创建ERC-20智能合约的事件日志的interface文件 `erc20.sol`：

```
pragma solidity ^0.4.24;

contract ERC20 {
    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, u
}
```

然后在给定abi使用 `abigen` 创建Go包

```
solc --abi erc20.sol
abigen --abi=erc20_sol_ERC20.abi --pkg=token --out=erc20.go
```

现在在我们的Go应用程序中，让我们创建与ERC-20事件日志签名类型相匹配的结构类型：

```
type LogTransfer struct {
    From   common.Address
    To     common.Address
    Tokens *big.Int
}

type LogApproval struct {
    TokenOwner common.Address
    Spender    common.Address
    Tokens     *big.Int
}
```

初始化以太坊客户端

```
client, err := ethclient.Dial("https://mainnet.infura.io")
if err != nil {
  log.Fatal(err)
}
```

按照ERC-20智能合约地址和所需的块范围创建一个“FilterQuery”。这个例子我们会用ZRX 代币:

```
// 0x Protocol (ZRX) token address
contractAddress := common.HexToAddress("0xe41d2489571d322189246da
query := ethereum.FilterQuery{
  FromBlock: big.NewInt(6383820),
  ToBlock:   big.NewInt(6383840),
  Addresses: []common.Address{
    contractAddress,
  },
}
```

用 FilterLogs 来过滤日志：

```
logs, err := client.FilterLogs(context.Background(), query)
if err != nil {
  log.Fatal(err)
}
```

接下来我们将解析JSON abi，稍后我们将使用解压缩原始日志数据：

```
contractAbi, err := abi.JSON(strings.NewReader(string(token.TokenABI)))
if err != nil {
  log.Fatal(err)
}
```

为了按某种日志类型进行过滤，我们需要弄清楚每个事件日志函数签名的keccak256哈希值。 事件日志函数签名哈希始终是 topic [0]，我们很快就会看到。 以下是使用go-ethereum crypto 包计算keccak256哈希的方法：

```
logTransferSig := []byte("Transfer(address,address,uint256)")
LogApprovalSig := []byte("Approval(address,address,uint256)")
logTransferSigHash := crypto.Keccak256Hash(logTransferSig)
logApprovalSigHash := crypto.Keccak256Hash(LogApprovalSig)
```

现在我们将遍历所有日志并设置switch语句以按事件日志类型进行过滤：

```
for _, vLog := range logs {
  fmt.Printf("Log Block Number: %d\n", vLog.BlockNumber)
  fmt.Printf("Log Index: %d\n", vLog.Index)

  switch vLog.Topics[0].Hex() {
  case logTransferSigHash.Hex():
    //
  case logApprovalSigHash.Hex():
    //
  }
}
```

现在要解析 Transfer 事件日志，我们将使用 abi.Unpack 将原始日志数据解析为我们的日志类型结构。 解包不会解析 indexed 事件类型，因为它们存储在 topics 下，所以对于那些我们必须单独解析，如下例所示：

```
fmt.Printf("Log Name: Transfer\n")

var transferEvent LogTransfer

err := contractAbi.Unpack(&transferEvent, "Transfer", vLog.Data)
if err != nil {
  log.Fatal(err)
}

transferEvent.From = common.HexToAddress(vLog.Topics[1].Hex())
transferEvent.To = common.HexToAddress(vLog.Topics[2].Hex())

fmt.Printf("From: %s\n", transferEvent.From.Hex())
fmt.Printf("To: %s\n", transferEvent.To.Hex())
fmt.Printf("Tokens: %s\n", transferEvent.Tokens.String())
```

Approval 日志也是类似的方法：

```
fmt.Printf("Log Name: Approval\n")

var approvalEvent LogApproval

err := contractAbi.Unpack(&approvalEvent, "Approval", vLog.Data)
if err != nil {
  log.Fatal(err)
}

approvalEvent.TokenOwner = common.HexToAddress(vLog.Topics[1].Hex())
approvalEvent.Spender = common.HexToAddress(vLog.Topics[2].Hex())

fmt.Printf("Token Owner: %s\n", approvalEvent.TokenOwner.Hex())
fmt.Printf("Spender: %s\n", approvalEvent.Spender.Hex())
fmt.Printf("Tokens: %s\n", approvalEvent.Tokens.String())
```

最后，把所有的步骤放一起：

```
Log Block Number: 6383829
Log Index: 20
Log Name: Transfer
From: 0xd03dB9CF89A9b1f856a8E1650cFD78FAF2338eB2
To: 0x924CD9b60F4173DCDd5254ddD38C4F9CAB68FE6b
Tokens: 2804000000000000000000


Log Block Number: 6383831
Log Index: 62
Log Name: Approval
Token Owner: 0xDD3b9186Da521AbE707B48B8f805Fb3Cd5EEe0EE
Spender: 0xCf67d7A481CEEca0a77f658991A00366FED558F7
Tokens: 10000000000000000000000000000000000000000000000000000000000000


Log Block Number: 6383838
Log Index: 13
Log Name: Transfer
From: 0xBA826fEc90CEFdf6706858E5FbaFcb27A290Fbe0
To: 0x4aEE792A88eDDA29932254099b9d1e06D537883f
Tokens: 2863452144424379687066
```

我们可以把解析的日志与etherscan的数据对比:
https://etherscan.io/tx/0x0c3b6cf604275c7e44dc7db400428c1a39f33f0c6cbc19ff625f6057a5cb32c0#eventlog

完整代码

Commands

```
solc --abi erc20.sol
abigen --abi=erc20_sol_ERC20.abi --pkg=token --out=erc20.go
```

erc20.sol

```
pragma solidity ^0.4.24;

contract ERC20 {
    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, u
}
```

event_read_erc20.go

```
package main

import (
    "context"
    "fmt"
    "log"
    "math/big"
    "strings"

    token "./contracts_erc20" // for demo
    "github.com/ethereum/go-ethereum"
    "github.com/ethereum/go-ethereum/accounts/abi"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/crypto"
    "github.com/ethereum/go-ethereum/ethclient"
)

// LogTransfer ..
type LogTransfer struct {
    From   common.Address
    To     common.Address
    Tokens *big.Int
}

// LogApproval ..
type LogApproval struct {
    TokenOwner common.Address
    Spender    common.Address
    Tokens     *big.Int
}

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    // 0x Protocol (ZRX) token address
    contractAddress := common.HexToAddress("0xe41d2489571d322189246
    query := ethereum.FilterQuery{
        FromBlock: big.NewInt(6383820),
        ToBlock:   big.NewInt(6383840),
        Addresses: []common.Address{
            contractAddress,
        },
    }

    logs, err := client.FilterLogs(context.Background(), query)
```

```
if err != nil {
    log.Fatal(err)
}

contractAbi, err := abi.JSON(strings.NewReader(string(token.TokenABI)))
if err != nil {
    log.Fatal(err)
}

logTransferSig := []byte("Transfer(address,address,uint256)")
LogApprovalSig := []byte("Approval(address,address,uint256)")
logTransferSigHash := crypto.Keccak256Hash(logTransferSig)
logApprovalSigHash := crypto.Keccak256Hash(LogApprovalSig)

for _, vLog := range logs {
    fmt.Printf("Log Block Number: %d\n", vLog.BlockNumber)
    fmt.Printf("Log Index: %d\n", vLog.Index)

    switch vLog.Topics[0].Hex() {
    case logTransferSigHash.Hex():
        fmt.Printf("Log Name: Transfer\n")

        var transferEvent LogTransfer

        err := contractAbi.Unpack(&transferEvent, "Transfer", vLog.Data)
        if err != nil {
            log.Fatal(err)
        }

        transferEvent.From = common.HexToAddress(vLog.Topics[1].Hex())
        transferEvent.To = common.HexToAddress(vLog.Topics[2].Hex())

        fmt.Printf("From: %s\n", transferEvent.From.Hex())
        fmt.Printf("To: %s\n", transferEvent.To.Hex())
        fmt.Printf("Tokens: %s\n", transferEvent.Tokens.String())

    case logApprovalSigHash.Hex():
        fmt.Printf("Log Name: Approval\n")

        var approvalEvent LogApproval

        err := contractAbi.Unpack(&approvalEvent, "Approval", vLog.Data)
        if err != nil {
            log.Fatal(err)
        }

        approvalEvent.TokenOwner = common.HexToAddress(vLog.Topics[1]
        approvalEvent.Spender = common.HexToAddress(vLog.Topics[2].Hex
```

```
            fmt.Printf("Token Owner: %s\n", approvalEvent.TokenOwner.Hex())
            fmt.Printf("Spender: %s\n", approvalEvent.Spender.Hex())
            fmt.Printf("Tokens: %s\n", approvalEvent.Tokens.String())
        }

        fmt.Printf("\n\n")
    }
}
```

solc version used for these examples

```
$ solc --version
0.4.24+commit.e67f0147.Emscripten.clang
```

读取0x Protocol事件日志

要读取0x Protocol事件日志，我们必须首先将solidity智能合约编译为一个Go包。

安装solc版本 `0.4.11`

```
npm i -g solc@0.4.11
```

为例如 `Exchange.sol` 的事件日志创建0x Protocol交易所智能合约接口:

Create the 0x protocol exchange smart contract interface for event logs as `Exchange.sol`:

```
pragma solidity 0.4.11;

contract Exchange {
    event LogFill(
        address indexed maker,
        address taker,
        address indexed feeRecipient,
        address makerToken,
        address takerToken,
        uint filledMakerTokenAmount,
        uint filledTakerTokenAmount,
        uint paidMakerFee,
        uint paidTakerFee,
        bytes32 indexed tokens, // keccak256(makerToken, takerToken), allows
        bytes32 orderHash
    );

    event LogCancel(
        address indexed maker,
        address indexed feeRecipient,
        address makerToken,
        address takerToken,
        uint cancelledMakerTokenAmount,
        uint cancelledTakerTokenAmount,
        bytes32 indexed tokens,
        bytes32 orderHash
    );

    event LogError(uint8 indexed errorId, bytes32 indexed orderHash);
}
```

接着给定abi，使用 abigen 来创建Go exchange 包：

Then use abigen to create the Go exchange package given the abi:

```
solc --abi Exchange.sol
abigen --abi="Exchange.sol:Exchange.abi" --pkg=exchange --out=Exchange.
```

现在在我们的Go应用程序中，让我们创建与0xProtocol事件日志签名类型匹配的结构体类型：

```
type LogFill struct {
    Maker                common.Address
    Taker                common.Address
    FeeRecipient         common.Address
    MakerToken           common.Address
    TakerToken           common.Address
    FilledMakerTokenAmount *big.Int
    FilledTakerTokenAmount *big.Int
    PaidMakerFee         *big.Int
    PaidTakerFee         *big.Int
    Tokens               [32]byte
    OrderHash            [32]byte
}

type LogCancel struct {
    Maker                  common.Address
    FeeRecipient           common.Address
    MakerToken             common.Address
    TakerToken             common.Address
    CancelledMakerTokenAmount *big.Int
    CancelledTakerTokenAmount *big.Int
    Tokens                 [32]byte
    OrderHash              [32]byte
}

type LogError struct {
    ErrorID   uint8
    OrderHash [32]byte
}
```

初始化以太坊客户端：

```
client, err := ethclient.Dial("https://mainnet.infura.io")
if err != nil {
  log.Fatal(err)
}
```

创建一个 FilterQuery，并为其传递0x Protocol智能合约地址和所需的区块范围：

```
// 0x Protocol Exchange smart contract address
contractAddress := common.HexToAddress("0x12459C951127e0c374FF9105
query := ethereum.FilterQuery{
  FromBlock: big.NewInt(6383482),
  ToBlock:   big.NewInt(6383488),
  Addresses: []common.Address{
    contractAddress,
  },
}
```

用 FilterLogs 查询日志：

```
logs, err := client.FilterLogs(context.Background(), query)
if err != nil {
  log.Fatal(err)
}
```

接下来我们将解析JSON abi，我们后续将使用解压缩原始日志数据：

```
contractAbi, err := abi.JSON(strings.NewReader(string(exchange.ExchangeA
if err != nil {
  log.Fatal(err)
}
```

为了按某种日志类型过滤，我们需要知晓每个事件日志函数签名的keccak256摘要。正如我们很快所见到的那样，事件日志函数签名摘要总是 topic[0]：

```
// NOTE: keccak256("LogFill(address,address,address,address,address,uint2
logFillEvent := common.HexToHash("0d0b9391970d9a25552f37d436d2aae2

// NOTE: keccak256("LogCancel(address,address,address,address,uint256,u
logCancelEvent := common.HexToHash("67d66f160bc93d925d05dae1794c9

// NOTE: keccak256("LogError(uint8,bytes32)")
logErrorEvent := common.HexToHash("36d86c59e00bd73dc19ba3adfe068e
```

现在我们迭代所有的日志并设置一个switch语句来按事件日志类型过滤：

```
for _, vLog := range logs {
 fmt.Printf("Log Block Number: %d\n", vLog.BlockNumber)
 fmt.Printf("Log Index: %d\n", vLog.Index)

 switch vLog.Topics[0].Hex() {
 case logFillEvent.Hex():
  //
 case logCancelEvent.Hex():
  //
 case logErrorEvent.Hex():
  //
 }
}
```

现在要解析 LogFill，我们将使用 abi.Unpack 将原始数据类型解析为我们自定义的日志类型结构体。Unpack不会解析 indexed 事件类型，因为这些它们存储在 topics 下，所以对于那些我们必须单独解析，如下例所示：

```
fmt.Printf("Log Name: LogFill\n")

var fillEvent LogFill

err := contractAbi.Unpack(&fillEvent, "LogFill", vLog.Data)
if err != nil {
 log.Fatal(err)
}

fillEvent.Maker = common.HexToAddress(vLog.Topics[1].Hex())
fillEvent.FeeRecipient = common.HexToAddress(vLog.Topics[2].Hex())
fillEvent.Tokens = vLog.Topics[3]

fmt.Printf("Maker: %s\n", fillEvent.Maker.Hex())
fmt.Printf("Taker: %s\n", fillEvent.Taker.Hex())
fmt.Printf("Fee Recipient: %s\n", fillEvent.FeeRecipient.Hex())
fmt.Printf("Maker Token: %s\n", fillEvent.MakerToken.Hex())
fmt.Printf("Taker Token: %s\n", fillEvent.TakerToken.Hex())
fmt.Printf("Filled Maker Token Amount: %s\n", fillEvent.FilledMakerTokenAn
fmt.Printf("Filled Taker Token Amount: %s\n", fillEvent.FilledTakerTokenAmc
fmt.Printf("Paid Maker Fee: %s\n", fillEvent.PaidMakerFee.String())
fmt.Printf("Paid Taker Fee: %s\n", fillEvent.PaidTakerFee.String())
fmt.Printf("Tokens: %s\n", hexutil.Encode(fillEvent.Tokens[:]))
fmt.Printf("Order Hash: %s\n", hexutil.Encode(fillEvent.OrderHash[:]))
```

对于 LogCancel 类似：

```
fmt.Printf("Log Name: LogCancel\n")

var cancelEvent LogCancel

err := contractAbi.Unpack(&cancelEvent, "LogCancel", vLog.Data)
if err != nil {
  log.Fatal(err)
}

cancelEvent.Maker = common.HexToAddress(vLog.Topics[1].Hex())
cancelEvent.FeeRecipient = common.HexToAddress(vLog.Topics[2].Hex())
cancelEvent.Tokens = vLog.Topics[3]

fmt.Printf("Maker: %s\n", cancelEvent.Maker.Hex())
fmt.Printf("Fee Recipient: %s\n", cancelEvent.FeeRecipient.Hex())
fmt.Printf("Maker Token: %s\n", cancelEvent.MakerToken.Hex())
fmt.Printf("Taker Token: %s\n", cancelEvent.TakerToken.Hex())
fmt.Printf("Cancelled Maker Token Amount: %s\n", cancelEvent.CancelledM
fmt.Printf("Cancelled Taker Token Amount: %s\n", cancelEvent.CancelledTak
fmt.Printf("Tokens: %s\n", hexutil.Encode(cancelEvent.Tokens[:]))
fmt.Printf("Order Hash: %s\n", hexutil.Encode(cancelEvent.OrderHash[:]))
```

最后是 `LogError`：

```
fmt.Printf("Log Name: LogError\n")

errorID, err := strconv.ParseInt(vLog.Topics[1].Hex(), 16, 64)
if err != nil {
  log.Fatal(err)
}

errorEvent := &LogError{
  ErrorID:   uint8(errorID),
  OrderHash: vLog.Topics[2],
}

fmt.Printf("Error ID: %d\n", errorEvent.ErrorID)
fmt.Printf("Order Hash: %s\n", hexutil.Encode(errorEvent.OrderHash[:]))
```

将它们放在一起并运行我们将看到以下输出：

```
Log Block Number: 6383482
Log Index: 35
Log Name: LogFill
Maker: 0x8dd688660ec0BaBD0B8a2f2DE3232645F73cC5eb
Taker: 0xe269E891A2Ec8585a378882fFA531141205e92E9
Fee Recipient: 0xe269E891A2Ec8585a378882fFA531141205e92E9
Maker Token: 0xD7732e3783b0047aa251928960063f863AD022D8
Taker Token: 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
Filled Maker Token Amount: 240000000000000000000000
Filled Taker Token Amount: 6930282000000000000
Paid Maker Fee: 0
Paid Taker Fee: 0
Tokens: 0xf08499c9e419ea8c08c4b991f88632593fb36baf4124c62758acb21
Order Hash: 0x306a9a7ecbd9446559a2c650b4cfc16d1fb615aa2b3f4f63078


Log Block Number: 6383482
Log Index: 38
Log Name: LogFill
Maker: 0x04aa059b2e31B5898fAB5aB24761e67E8a196AB8
Taker: 0xe269E891A2Ec8585a378882fFA531141205e92E9
Fee Recipient: 0xe269E891A2Ec8585a378882fFA531141205e92E9
Maker Token: 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
Taker Token: 0xD7732e3783b0047aa251928960063f863AD022D8
Filled Maker Token Amount: 6941718000000000000
Filled Taker Token Amount: 240000000000000000000000
Paid Maker Fee: 0
Paid Taker Fee: 0
Tokens: 0x97ef123f2b566f36ab1e6f5d462a8079fbe34fa667b4eae67194b3f
Order Hash: 0xac270e88ce27b6bb78ee5b68ebaef666a77195020a6ab8922


Log Block Number: 6383488
Log Index: 43
Log Name: LogCancel
Maker: 0x0004E79C978B95974dCa16F56B516bE0c50CC652
Fee Recipient: 0xA258b39954ceF5cB142fd567A46cDdB31a670124
Maker Token: 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
Taker Token: 0x89d24A6b4CcB1B6fAA2625fE562bDD9a23260359
Cancelled Maker Token Amount: 30000000000000000000
Cancelled Taker Token Amount: 727484842500000000000
Tokens: 0x9dd48110dcc444fdc242510c09bbbbe21a5975cac061d82f7b843b
Order Hash: 0xe43eff38dc27af046bfbd431926926c072bbc7a509d56f6f1a7a
```

将解析后的日志输出与etherscan上的内容进行比较：
https://etherscan.io/tx/0xb73a4492c5db1f67930b25ce3869c1e6b9bdbccb239a23b6454925a5bc0e03c5

完整代码

命令

```
solc --abi Exchange.sol
abigen --abi="Exchange.sol:Exchange.abi" --pkg=exchange --out=Exchange.
```

Exchange.sol

```
pragma solidity 0.4.11;

contract Exchange {
    event LogFill(
        address indexed maker,
        address taker,
        address indexed feeRecipient,
        address makerToken,
        address takerToken,
        uint filledMakerTokenAmount,
        uint filledTakerTokenAmount,
        uint paidMakerFee,
        uint paidTakerFee,
        bytes32 indexed tokens, // keccak256(makerToken, takerToken), allows
        bytes32 orderHash
    );

    event LogCancel(
        address indexed maker,
        address indexed feeRecipient,
        address makerToken,
        address takerToken,
        uint cancelledMakerTokenAmount,
        uint cancelledTakerTokenAmount,
        bytes32 indexed tokens,
        bytes32 orderHash
    );

    event LogError(uint8 indexed errorId, bytes32 indexed orderHash);
}
```

event_read_0xprotocol.go

```
package main

import (
    "context"
    "fmt"
    "log"
    "math/big"
    "strconv"
    "strings"

    exchange "./contracts_0xprotocol" // for demo
    "github.com/ethereum/go-ethereum"
    "github.com/ethereum/go-ethereum/accounts/abi"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/ethclient"
)

// LogFill ...
type LogFill struct {
    Maker               common.Address
    Taker               common.Address
    FeeRecipient        common.Address
    MakerToken          common.Address
    TakerToken          common.Address
    FilledMakerTokenAmount *big.Int
    FilledTakerTokenAmount *big.Int
    PaidMakerFee        *big.Int
    PaidTakerFee        *big.Int
    Tokens              [32]byte
    OrderHash           [32]byte
}

// LogCancel ...
type LogCancel struct {
    Maker                  common.Address
    FeeRecipient           common.Address
    MakerToken             common.Address
    TakerToken             common.Address
    CancelledMakerTokenAmount *big.Int
    CancelledTakerTokenAmount *big.Int
    Tokens                 [32]byte
    OrderHash              [32]byte
}

// LogError ...
type LogError struct {
    ErrorID   uint8
```

```
    OrderHash [32]byte
}

func main() {
    client, err := ethclient.Dial("https://mainnet.infura.io")
    if err != nil {
        log.Fatal(err)
    }

    // 0x Protocol Exchange smart contract address
    contractAddress := common.HexToAddress("0x12459C951127e0c374FF91
    query := ethereum.FilterQuery{
        FromBlock: big.NewInt(6383482),
        ToBlock:   big.NewInt(6383488),
        Addresses: []common.Address{
            contractAddress,
        },
    }

    logs, err := client.FilterLogs(context.Background(), query)
    if err != nil {
        log.Fatal(err)
    }

    contractAbi, err := abi.JSON(strings.NewReader(string(exchange.Exchang
    if err != nil {
        log.Fatal(err)
    }

    // NOTE: keccak256("LogFill(address,address,address,address,address,uir
    logFillEvent := common.HexToHash("0d0b9391970d9a25552f37d436d2aa

    // NOTE: keccak256("LogCancel(address,address,address,address,uint256
    logCancelEvent := common.HexToHash("67d66f160bc93d925d05dae1794

    // NOTE: keccak256("LogError(uint8,bytes32)")
    logErrorEvent := common.HexToHash("36d86c59e00bd73dc19ba3adfe06

    for _, vLog := range logs {
        fmt.Printf("Log Block Number: %d\n", vLog.BlockNumber)
        fmt.Printf("Log Index: %d\n", vLog.Index)

        switch vLog.Topics[0].Hex() {
        case logFillEvent.Hex():
            fmt.Printf("Log Name: LogFill\n")

            var fillEvent LogFill
```

```
        err := contractAbi.Unpack(&fillEvent, "LogFill", vLog.Data)
        if err != nil {
            log.Fatal(err)
        }

        fillEvent.Maker = common.HexToAddress(vLog.Topics[1].Hex())
        fillEvent.FeeRecipient = common.HexToAddress(vLog.Topics[2].Hex())
        fillEvent.Tokens = vLog.Topics[3]

        fmt.Printf("Maker: %s\n", fillEvent.Maker.Hex())
        fmt.Printf("Taker: %s\n", fillEvent.Taker.Hex())
        fmt.Printf("Fee Recipient: %s\n", fillEvent.FeeRecipient.Hex())
        fmt.Printf("Maker Token: %s\n", fillEvent.MakerToken.Hex())
        fmt.Printf("Taker Token: %s\n", fillEvent.TakerToken.Hex())
        fmt.Printf("Filled Maker Token Amount: %s\n", fillEvent.FilledMakerTo
        fmt.Printf("Filled Taker Token Amount: %s\n", fillEvent.FilledTakerTok
        fmt.Printf("Paid Maker Fee: %s\n", fillEvent.PaidMakerFee.String())
        fmt.Printf("Paid Taker Fee: %s\n", fillEvent.PaidTakerFee.String())
        fmt.Printf("Tokens: %s\n", hexutil.Encode(fillEvent.Tokens[:]))
        fmt.Printf("Order Hash: %s\n", hexutil.Encode(fillEvent.OrderHash[:]))

    case logCancelEvent.Hex():
        fmt.Printf("Log Name: LogCancel\n")

        var cancelEvent LogCancel

        err := contractAbi.Unpack(&cancelEvent, "LogCancel", vLog.Data)
        if err != nil {
            log.Fatal(err)
        }

        cancelEvent.Maker = common.HexToAddress(vLog.Topics[1].Hex())
        cancelEvent.FeeRecipient = common.HexToAddress(vLog.Topics[2].H
        cancelEvent.Tokens = vLog.Topics[3]

        fmt.Printf("Maker: %s\n", cancelEvent.Maker.Hex())
        fmt.Printf("Fee Recipient: %s\n", cancelEvent.FeeRecipient.Hex())
        fmt.Printf("Maker Token: %s\n", cancelEvent.MakerToken.Hex())
        fmt.Printf("Taker Token: %s\n", cancelEvent.TakerToken.Hex())
        fmt.Printf("Cancelled Maker Token Amount: %s\n", cancelEvent.Canc
        fmt.Printf("Cancelled Taker Token Amount: %s\n", cancelEvent.Cance
        fmt.Printf("Tokens: %s\n", hexutil.Encode(cancelEvent.Tokens[:]))
        fmt.Printf("Order Hash: %s\n", hexutil.Encode(cancelEvent.OrderHas

    case logErrorEvent.Hex():
        fmt.Printf("Log Name: LogError\n")

        errorID, err := strconv.ParseInt(vLog.Topics[1].Hex(), 16, 64)
```

```
            if err != nil {
                log.Fatal(err)
            }

            errorEvent := &LogError{
                ErrorID:   uint8(errorID),
                OrderHash: vLog.Topics[2],
            }

            fmt.Printf("Error ID: %d\n", errorEvent.ErrorID)
            fmt.Printf("Order Hash: %s\n", hexutil.Encode(errorEvent.OrderHash
        }

        fmt.Printf("\n\n")
    }
}
```

这些示例使用的solc版本

```
$ solc --version
0.4.11+commit.68ef5810.Emscripten.clang
```

签名

数字签名允许不可否认性，因为这意味着签署消息的人必须拥有私钥，来证明消息是真实的。 任何人都可以验证消息的真实性，只要它们具有原始数据的散列和签名者的公钥即可。 签名是区块链的基本组成部分，我们将在接下来的几节课中学习如何生成和验证签名。

生成一个签名

用于生成签名的组件是：签名者私钥，以及将要签名的数据的哈希。 只要输出为32字节，就可以使用任何哈希算法。 我们将使用Keccak-256作为哈希算法，这是以太坊常常使用的算法。

首先，我们将加载私钥。

```
privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad0f3
if err != nil {
  log.Fatal(err)
}
```

接下来我们将获取我们希望签名的数据的Keccak-256，在这个例子里，它将是*hello*。go-ethereum `crypto` 包提供了一个方便的 `Keccak256Hash` 方法来实现这一目的。

```
data := []byte("hello")
hash := crypto.Keccak256Hash(data)
fmt.Println(hash.Hex()) // 0x1c8aff950685c2ed4bc3174f3472287b56d9517b
```

最后，我们使用私钥签名哈希，得到签名。

```
signature, err := crypto.Sign(hash.Bytes(), privateKey)
if err != nil {
  log.Fatal(err)
}

fmt.Println(hexutil.Encode(signature)) // 0x789a80053e4927d0a898db8e065
```

现在我们已经成功生成了签名，在下个章节中，我们将学习如何验证签名确实是由该私钥的持有者签名的。

完整代码

signature_generate.go

```
package main

import (
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/crypto"
)

func main() {
    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad
    if err != nil {
        log.Fatal(err)
    }

    data := []byte("hello")
    hash := crypto.Keccak256Hash(data)
    fmt.Println(hash.Hex()) // 0x1c8aff950685c2ed4bc3174f3472287b56d9517

    signature, err := crypto.Sign(hash.Bytes(), privateKey)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(hexutil.Encode(signature)) // 0x789a80053e4927d0a898db8e0
}
```

验证签名

在上个章节中，我们学习了如何使用私钥对一段数据进行签名以生成签名。 现在我们将学习如何验证签名的真实性。

我们需要有3件事来验证签名：签名，原始数据的哈希以及签名者的公钥。 利用该信息，我们可以确定公钥对的私钥持有者是否确实签署了该消息。

首先，我们需要以字节格式的公钥。

```
publicKeyBytes := crypto.FromECDSAPub(publicKeyECDSA)
```

接下来我们将需要原始数据哈希。 在上一课中，我们使用Keccak-256生成哈希，因此我们将执行相同的操作以验证签名。

```
data := []byte("hello")
hash := crypto.Keccak256Hash(data)
fmt.Println(hash.Hex()) // 0x1c8aff950685c2ed4bc3174f3472287b56d9517b
```

现在假设我们有字节格式的签名，我们可以从go-ethereum `crypto` 包调用 Ecrecover （椭圆曲线签名恢复）来检索签名者的公钥。 此函数采用字节格式的哈希和签名。

```
sigPublicKey, err := crypto.Ecrecover(hash.Bytes(), signature)
if err != nil {
  log.Fatal(err)
}
```

为了验证我们现在必须将签名的公钥与期望的公钥进行比较，如果它们匹配，那么预期的公钥持有者确实是原始消息的签名者。

```
matches := bytes.Equal(sigPublicKey, publicKeyBytes)
fmt.Println(matches) // true
```

还有 `SigToPub` 方法做同样的事情，区别是它将返回ECDSA类型中的签名公钥。

```
sigPublicKeyECDSA, err := crypto.SigToPub(hash.Bytes(), signature)
if err != nil {
  log.Fatal(err)
}

sigPublicKeyBytes := crypto.FromECDSAPub(sigPublicKeyECDSA)
matches = bytes.Equal(sigPublicKeyBytes, publicKeyBytes)
fmt.Println(matches) // true
```

为方便起见， go-ethereum/crypto 包提供了 VerifySignature 函数，该函数接收原始数据的签名，哈希值和字节格式的公钥。 它返回一个布尔值，如果公钥与签名的签名者匹配，则为true。一个重要的问题是我们必须首先删除signture的最后一个字节，因为它是ECDSA恢复ID，不能包含它。

```
signatureNoRecoverID := signature[:len(signature)-1] // remove recovery ID
verified := crypto.VerifySignature(publicKeyBytes, hash.Bytes(), signatureNo
fmt.Println(verified) // true
```

这些就是使用go-ethereum软件包生成和验证ECDSA签名的基础知识。

完整代码

signature_verify.go

```
package main

import (
    "bytes"
    "crypto/ecdsa"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/crypto"
)

func main() {
    privateKey, err := crypto.HexToECDSA("fad9c8855b740a0b7ed4c221dbad
    if err != nil {
        log.Fatal(err)
    }

    publicKey := privateKey.Public()
    publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
    if !ok {
        log.Fatal("cannot assert type: publicKey is not of type *ecdsa.PublicKey
    }

    publicKeyBytes := crypto.FromECDSAPub(publicKeyECDSA)

    data := []byte("hello")
    hash := crypto.Keccak256Hash(data)
    fmt.Println(hash.Hex()) // 0x1c8aff950685c2ed4bc3174f3472287b56d9517

    signature, err := crypto.Sign(hash.Bytes(), privateKey)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(hexutil.Encode(signature)) // 0x789a80053e4927d0a898db8e0

    sigPublicKey, err := crypto.Ecrecover(hash.Bytes(), signature)
    if err != nil {
        log.Fatal(err)
    }

    matches := bytes.Equal(sigPublicKey, publicKeyBytes)
    fmt.Println(matches) // true

    sigPublicKeyECDSA, err := crypto.SigToPub(hash.Bytes(), signature)
    if err != nil {
        log.Fatal(err)
```

```
    }

    sigPublicKeyBytes := crypto.FromECDSAPub(sigPublicKeyECDSA)
    matches = bytes.Equal(sigPublicKeyBytes, publicKeyBytes)
    fmt.Println(matches) // true

    signatureNoRecoverID := signature[:len(signature)-1] // remove recovery
    verified := crypto.VerifySignature(publicKeyBytes, hash.Bytes(), signatureI
    fmt.Println(verified) // true
}
```

Testing

- 发币水龙头
- 使用模拟客户端

水龙头

水龙头是免费在[测试网]获得ETH的方式。

这里罗列了各个测试网的龙头。

- Ropsten testnet - https://faucet.ropsten.be
- Rinkeby testnet - https://faucet.rinkeby.io
- Kovan testnet - https://gitter.im/kovan-testnet/faucet
- Sokol testnet - https://faucet-sokol.herokuapp.com

使用模拟客户端

您可以使用模拟客户端来快速轻松地在本地测试您的交易，非常适合单元测试。为了开始，我们需要一个带有初始ETH的账户。为此，首先生成一个账户私钥。

```
privateKey, err := crypto.GenerateKey()
if err != nil {
  log.Fatal(err)
}
```

接着从 `accounts/abi/bind` 包创建一个 `NewKeyedTransactor`，并为其传递私钥。

```
auth := bind.NewKeyedTransactor(privateKey)
```

下一步是创建一个创世账户并为其分配初始余额。我们将使用 `core` 包的 `GenesisAccount` 类型。

```
balance := new(big.Int)
balance.SetString("10000000000000000000", 10) // 10 eth in wei

address := auth.From
genesisAlloc := map[common.Address]core.GenesisAccount{
  address: {
    Balance: balance,
  },
}
```

现在我们将创世分配结构体和配置好的汽油上限传给 `account/abi/bind/backends` 包的 `NewSimulatedBackend` 方法，该方法将返回一个新的模拟以太坊客户端。

```
blockGasLimit := uint64(4712388)
client := backends.NewSimulatedBackend(genesisAlloc, blockGasLimit)
```

您可以像往常一样使用此客户端。作为一个示例，我们将构造一个新的交易并进行广播。

```
fromAddress := auth.From
nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
if err != nil {
  log.Fatal(err)
}

value := big.NewInt(1000000000000000000) // in wei (1 eth)
gasLimit := uint64(21000)                // in units
gasPrice, err := client.SuggestGasPrice(context.Background())
if err != nil {
  log.Fatal(err)
}

toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4fa1
var data []byte
tx := types.NewTransaction(nonce, toAddress, value, gasLimit, gasPrice, dat
chainID := big.NewInt(1)
signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), privateKe
if err != nil {
  log.Fatal(err)
}

err = client.SendTransaction(context.Background(), signedTx)
if err != nil {
  log.Fatal(err)
}

fmt.Printf("tx sent: %s\n", signedTx.Hash().Hex()) // tx sent: 0xec3ceb05642c
```

到现在为止，您可能想知道交易何时才会被开采。为了“开采”它，您还必须做一件额外的事情，在客户端调用 `Commit` 提交新开采的区块。

```
client.Commit()
```

现在您可以获取交易收据并看见其已被处理。

```
receipt, err := client.TransactionReceipt(context.Background(), signedTx.Has
if err != nil {
  log.Fatal(err)
}
if receipt == nil {
  log.Fatal("receipt is nil. Forgot to commit?")
}

fmt.Printf("status: %v\n", receipt.Status) // status: 1
```

因此，请记住：模拟客户端允许您使用模拟客户端的`Commit`方法手动开采区块。

完整代码

client_simulated.go

```
package main

import (
    "context"
    "fmt"
    "log"
    "math/big"

    "github.com/ethereum/go-ethereum/accounts/abi/bind"
    "github.com/ethereum/go-ethereum/accounts/abi/bind/backends"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/core"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/crypto"
)

func main() {
    privateKey, err := crypto.GenerateKey()
    if err != nil {
        log.Fatal(err)
    }

    auth := bind.NewKeyedTransactor(privateKey)

    balance := new(big.Int)
    balance.SetString("10000000000000000000", 10) // 10 eth in wei

    address := auth.From
    genesisAlloc := map[common.Address]core.GenesisAccount{
        address: {
            Balance: balance,
        },
    }

    blockGasLimit := uint64(4712388)
    client := backends.NewSimulatedBackend(genesisAlloc, blockGasLimit)

    fromAddress := auth.From
    nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
    if err != nil {
        log.Fatal(err)
    }

    value := big.NewInt(1000000000000000000) // in wei (1 eth)
    gasLimit := uint64(21000)                // in units
    gasPrice, err := client.SuggestGasPrice(context.Background())
    if err != nil {
        log.Fatal(err)
```

```
    }

    toAddress := common.HexToAddress("0x4592d8f8d7b001e72cb26a73e4f
    var data []byte
    tx := types.NewTransaction(nonce, toAddress, value, gasLimit, gasPrice, d

    chainID := big.NewInt(1)
    signedTx, err := types.SignTx(tx, types.NewEIP155Signer(chainID), private
    if err != nil {
        log.Fatal(err)
    }

    err = client.SendTransaction(context.Background(), signedTx)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("tx sent: %s\n", signedTx.Hash().Hex()) // tx sent: 0xec3ceb0564

    client.Commit()

    receipt, err := client.TransactionReceipt(context.Background(), signedTx.H
    if err != nil {
        log.Fatal(err)
    }
    if receipt == nil {
        log.Fatal("receipt is nil. Forgot to commit?")
    }

    fmt.Printf("status: %v\n", receipt.Status) // status: 1
}
```

Swarm

Swarm是以太坊的去中心化和分布式的存储解决方案，与IPFS类似。Swarm是一种点对点数据共享网络，其中文件通过其内容的哈希来寻址。与Bittorrent类似，可以同时从多个节点获取数据，只要单个节点承载分发数据，它就可以随处被访问。这种方法可以在不必依靠托管任何类型服务器的情况下分发数据 - 数据可访问性与位置无关。可以激励网络中的其他节点自己复制和存储数据，从而在原节点未连接到网络时避免了对托管服务的依赖。

Swarm的激励机制Swap（Swarm Accounting Protocol）是一种协议，通过该协议，Swarm网络中的个体可以跟踪传送和接收的数据块，以及由此产生相应的（微）付款。 SWAP本身可以在更广泛的背景下运行，但它通常表现为适用于点对点之间成对会计的通用微支付方案。虽然设计通用，但它的第一个用途是将带宽计算作为Swarm去中心化的点对点存储网络中数据传输的激励的一部分。

搭建 Swarm 节点

要运行swarm，首先需要安装 geth 和 bzzd ，这是swarm背景进程。

```
go get -d github.com/ethereum/go-ethereum
go install github.com/ethereum/go-ethereum/cmd/geth
go install github.com/ethereum/go-ethereum/cmd/swarm
```

然后我们将生成一个新的geth帐户。

```
$ geth account new

Your new account is locked with a password. Please give a password. Do no
Passphrase:
Repeat passphrase:
Address: {970ef9790b54425bea2c02e25cab01e48cf92573}
```

将环境变量 BZZKEY 导出，并设定为我们刚刚生成的geth帐户地址。

```
export BZZKEY=970ef9790b54425bea2c02e25cab01e48cf92573
```

然后使用设定的帐户运行swarm，并作为我们的swarm帐户。 默认情况下，Swarm将在端口“8500”上运行。

```
$ swarm --bzzaccount $BZZKEY
Unlocking swarm account 0x970EF9790B54425BEA2C02e25cAb01E48CF925
Passphrase:
WARN [06-12|13:11:41] Starting Swarm service
```

现在swarm进程已经可以运行了，那么我们会在下个章节学习如何上传文件。

完整代码

Commands

```
go get -d github.com/ethereum/go-ethereum
go install github.com/ethereum/go-ethereum/cmd/geth
go install github.com/ethereum/go-ethereum/cmd/swarm
geth account new
export BZZKEY=970ef9790b54425bea2c02e25cab01e48cf92573
swarm --bzzaccount $BZZKEY
```

上传文件到Swarm

在上个章节 我们在端口“8500”上运行了一个作为背景进程的swarm节点。接下来就导入swarm包go-ethereum `swearm/api/client`。我将把包装别名为 `bzzclient`。

```
import (
  bzzclient "github.com/ethereum/go-ethereum/swarm/api/client"
)
```

调用 `NewClient` 函数向它传递swarm背景程序的url。

```
client := bzzclient.NewClient("http://127.0.0.1:8500")
```

用内容 *hello world* 创建示例文本文件 `hello.txt`。我们将会把这个文件上传到swarm。

```
hello world
```

在我们的Go应用程序中，我们将使用Swarm客户端软件包中的“Open”打开我们刚刚创建的文件。 该函数将返回一个 `File` 类型，它表示swarm清单中的文件，用于上传和下载swarm内容。

```
file, err := bzzclient.Open("hello.txt")
if err != nil {
  log.Fatal(err)
}
```

现在我们可以从客户端实例调用 `Upload` 函数，为它提供文件对象。 第二个参数是一个可选添的现有内容清单字符串，用于添加文件，否则它将为我们创建。 第三个参数是我们是否希望我们的数据被加密。

返回的哈希值是文件的内容清单的哈希值，其中包含hello.txt文件作为其唯一条目。 默认情况下，主要内容和清单都会上传。 清单确保您可以使用正确的mime类型检索文件。

```
manifestHash, err := client.Upload(file, "", false)
if err != nil {
  log.Fatal(err)
}

fmt.Println(manifestHash) // 2e0849490b62e706a5f1cb8e7219db7b01677f2
```

然后我们就可以在这里查看上传的文件 `bzz://2e0849490b62e706a5f1cb8e7219db7b01677f2a859bac4b5f522afd2a5f02c0`，具体如何下载，我们会在下个章节介绍。

完整代码

Commands

```
geth account new
export BZZKEY=970ef9790b54425bea2c02e25cab01e48cf92573
swarm --bzzaccount $BZZKEY
```

hello.txt

```
hello world
```

swarm_upload.go

```
package main

import (
    "fmt"
    "log"

    bzzclient "github.com/ethereum/go-ethereum/swarm/api/client"
)

func main() {
    client := bzzclient.NewClient("http://127.0.0.1:8500")

    file, err := bzzclient.Open("hello.txt")
    if err != nil {
        log.Fatal(err)
    }

    manifestHash, err := client.Upload(file, "", false)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(manifestHash) // 2e0849490b62e706a5f1cb8e7219db7b01677
}
```

从Swarm下载文件

在上个章节 我们将一个hello.txt文件上传到swarm，作为返回值，我们得到了一个内容清单哈希。

```
manifestHash := "f9192507e2e8e118bfedac428c3aa1dec4ae156e954128ec5
```

让我们首先通过调用“DownloadManfest”来下载它，并检查清单的内容。

```
manifest, isEncrypted, err := client.DownloadManifest(manifestHash)
if err != nil {
  log.Fatal(err)
}
```

我们可以遍历清单条目，看看内容类型，大小和内容哈希是什么。

```
for _, entry := range manifest.Entries {
  fmt.Println(entry.Hash)       // 42179060941352ba7b400b16c40f1e1290423
  fmt.Println(entry.ContentType) // text/plain; charset=utf-8
  fmt.Println(entry.Path)       // ""
}
```

如果您熟悉swarm url，它们的格式为 `bzz:/<hash>/<path>`，因此为了下载文件，我们指定了清单哈希和路径。 在这个例子里，路径是一个空字符串。 我们将这些数据传递给 `Download` 函数并返回一个文件对象。

```
file, err := client.Download(manifestHash, "")
if err != nil {
  log.Fatal(err)
}
```

我们现在可以阅读并打印返回的文件阅读器的内容。

```
content, err := ioutil.ReadAll(file)
if err != nil {
  log.Fatal(err)
}

fmt.Println(string(content)) // hello world
```

正如预期的那样，它记录了我们原始文件所包含的 *hello world*。

完整代码

Commands

```
geth account new
export BZZKEY=970ef9790b54425bea2c02e25cab01e48cf92573
swarm --bzzaccount $BZZKEY
```

swarm_download.go

```
package main

import (
    "fmt"
    "io/ioutil"
    "log"

    bzzclient "github.com/ethereum/go-ethereum/swarm/api/client"
)

func main() {
    client := bzzclient.NewClient("http://127.0.0.1:8500")
    manifestHash := "2e0849490b62e706a5f1cb8e7219db7b01677f2a859bac
    manifest, isEncrypted, err := client.DownloadManifest(manifestHash)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(isEncrypted) // false

    for _, entry := range manifest.Entries {
        fmt.Println(entry.Hash)       // 42179060941352ba7b400b16c40f1e1290
        fmt.Println(entry.ContentType) // text/plain; charset=utf-8
        fmt.Println(entry.Size)       // 12
        fmt.Println(entry.Path)       // ""
    }

    file, err := client.Download(manifestHash, "")
    if err != nil {
        log.Fatal(err)
    }

    content, err := ioutil.ReadAll(file)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(string(content)) // hello world
}
```

Whisper

Whisper是一种简单的基于点对点身份的消息传递系统，旨在成为下一去中心化的应用程序的构建块。 它旨在以相当的代价提供弹性和隐私。 在接下来的部分中，我们将设置一个支持Whisper的以太坊节点，然后我们将学习如何在Whisper协议上发送和接收加密消息。

连接Whisper客户端

要使用连接Whisper客户端，我们必须首先连接到运行whisper的以太坊节点。 不幸的是，诸如infura之类的公共网关不支持whisper，因为没有金钱动力免费处理这些消息。 Infura可能会在不久的将来支持whisper，但现在我们必须运行我们自己的 `geth` 节点。一旦你安装 geth, 运行geth的时候加 `--shh` flag来支持whisper协议, 并且加 `--ws` flag和 `--rpc`，来支持websocket来接收实时信息，

```
geth --rpc --shh --ws
```

现在在我们的Go应用程序中，我们将导入在 `whisper/shhclient` 中找到的go-ethereum whisper客户端软件包并初始化客户端，使用默认的websocket端口“8546”通过websockets连接我们的本地geth节点。

```
client, err := shhclient.Dial("ws://127.0.0.1:8546")
if err != nil {
  log.Fatal(err)
}

_ = client // we'll be using this in the 下个章节
```

现在我们已经拨打了，让我们创建一个密钥对来加密消息，然后再发送消息 在下一章节.

完整代码

Commands

```
geth --rpc --shh --ws
```

whisper_client.go

```
package main

import (
    "log"

    "github.com/ethereum/go-ethereum/whisper/shhclient"
)

func main() {
    client, err := shhclient.Dial("ws://127.0.0.1:8546")
    if err != nil {
        log.Fatal(err)
    }

    _ = client // we'll be using this in the 下个章节
    fmt.Println("we have a whisper connection")
}
```

生成 Whisper 密匙对

在Whisper中，消息必须使用对称或非对称密钥加密，以防止除预期接收者以外的任何人读取消息。

在连接到Whisper客户端后，您需要调用客户端的 `NewKeyPair` 方法来生成该节点将管理的新公共和私有对。 此函数的结果将是一个唯一的ID，它引用我们将在接下来的几节中用于加密和解密消息的密钥对。

```
keyID, err := client.NewKeyPair(context.Background())
if err != nil {
  log.Fatal(err)
}

fmt.Println(keyID) // 0ec5cfe4e215239756054992dbc2e10f011db1cdfc88b9
```

在下一章节 让我们学习如何发送一个加密的消息。

完整代码

Commands

```
geth --rpc --shh --ws
```

whisper_keypair.go

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/whisper/shhclient"
)

func main() {
    client, err := shhclient.Dial("ws://127.0.0.1:8546")
    if err != nil {
        log.Fatal(err)
    }

    keyID, err := client.NewKeyPair(context.Background())
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(keyID) // 0ec5cfe4e215239756054992dbc2e10f011db1cdfc88b
}
```

在Whisper上发送消息

在我们能够创建消息之前，我们必须首先使用公钥来加密消息。在上个章节中，我们学习了如何使用 NewKeyPair 函数生成公钥和私钥对，该函数返回了引用该密钥对的密钥ID。我们现在必须调用 PublicKey 函数以字节格式读取密钥对的公钥，我们将使用它来加密消息。

```
publicKey, err := client.PublicKey(context.Background(), keyID)
if err != nil {
  log.Print(err)
}

fmt.Println(hexutil.Encode(publicKey)) // 0x04f17356fd52b0d13e5ede84f998
```

现在我们将通过从go-ethereum whisper/whisperv6 包中初始化 NewMessage 结构来构造我们的私语消息，这需要以下属性：

- Payload 字节格式的消息内容
- PublicKey 加密的公钥
- TTL 消息的活跃时间
- PowTime 做工证明的时间上限
- PowTarget 做工证明的时间下限

```
message := whisperv6.NewMessage{
  Payload:   []byte("Hello"),
  PublicKey: publicKey,
  TTL:       60,
  PowTime:   2,
  PowTarget: 2.5,
}
```

我们现在可以通过调用客户端的 Post 函数向网络广播，给它消息，它是否会返回消息的哈希值。

```
messageHash, err := client.Post(context.Background(), message)
if err != nil {
  log.Fatal(err)
}

fmt.Println(messageHash) // 0xdbfc815d3d122a90d7fb44d1fc6a46f3d76ec7
```

在下个章节中我们将看到如何创建消息订阅以便能够实时接收消息。

完整代码

Commands

```
geth --shh --rpc --ws
```

whisper_send.go

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/whisper/shhclient"
    "github.com/ethereum/go-ethereum/whisper/whisperv6"
)

func main() {
    client, err := shhclient.Dial("ws://127.0.0.1:8546")
    if err != nil {
        log.Fatal(err)
    }

    keyID, err := client.NewKeyPair(context.Background())
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(keyID) // 0ec5cfe4e215239756054992dbc2e10f011db1cdfc88b

    publicKey, err := client.PublicKey(context.Background(), keyID)
    if err != nil {
        log.Print(err)
    }
    fmt.Println(hexutil.Encode(publicKey)) // 0x04f17356fd52b0d13e5ede84f9

    message := whisperv6.NewMessage{
        Payload:   []byte("Hello"),
        PublicKey: publicKey,
        TTL:       60,
        PowTime:   2,
        PowTarget: 2.5,
    }
    messageHash, err := client.Post(context.Background(), message)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(messageHash) // 0xdbfc815d3d122a90d7fb44d1fc6a46f3d76e
}
```

监听/订阅Whisper消息

在本节中，我们将订阅websockets上的Whisper消息。 我们首先需要的是一个通道，它将从 whisper/whisperv6 包中的 Message 类型接收Whispe消息。

```
messages := make(chan *whisperv6.Message)
```

在我们调用订阅之前，我们首先需要确定消息的过滤标准。 从whisperv6包中初始化一个新的 Criteria 对象。 由于我们只对定位到我们的消息感兴趣，因此我们将条件对象上的 PrivateKeyID 属性设置为我们用于加密消息的相同密钥ID。

```
criteria := whisperv6.Criteria{
  PrivateKeyID: keyID,
}
```

接下来，我们调用客户端的 SubscribeMessages 方法，该方法订阅符合给定条件的消息。 HTTP不支持此方法; 仅支持双向连接，例如websockets和IPC。最后一个参数是我们之前创建的消息通道。

```
sub, err := client.SubscribeMessages(context.Background(), criteria, messag
if err != nil {
  log.Fatal(err)
}
```

现在我们已经订阅了，我们可以使用 select 语句来读取消息，并处理订阅中的错误。 如果您从上一节回忆起来，消息内容在 Payload 属性中作为字节切片，我们可以将其转换回人类可读的字符串。

```
for {
  select {
  case err := <-sub.Err():
    log.Fatal(err)
  case message := <-messages:
    fmt.Printf(string(message.Payload)) // "Hello"
  }
}
```

查看下面的完整代码，获取完整的栗子。 这就是消息订阅的所有内容。

完整代码

Commands

```
geth --shh --rpc --ws
```

whisper_subscribe.go

```
package main

import (
    "context"
    "fmt"
    "log"
    "os"
    "runtime"

    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/ethereum/go-ethereum/whisper/shhclient"
    "github.com/ethereum/go-ethereum/whisper/whisperv6"
)

func main() {
    client, err := shhclient.Dial("ws://127.0.0.1:8546")
    if err != nil {
        log.Fatal(err)
    }

    keyID, err := client.NewKeyPair(context.Background())
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(keyID) // 0ec5cfe4e215239756054992dbc2e10f011db1cdfc88b

    messages := make(chan *whisperv6.Message)
    criteria := whisperv6.Criteria{
        PrivateKeyID: keyID,
    }
    sub, err := client.SubscribeMessages(context.Background(), criteria, mess
    if err != nil {
        log.Fatal(err)
    }

    go func() {
        for {
            select {
            case err := <-sub.Err():
                log.Fatal(err)
            case message := <-messages:
                fmt.Printf(string(message.Payload)) // "Hello"
                os.Exit(0)
            }
        }
    }()

    publicKey, err := client.PublicKey(context.Background(), keyID)
```

```
    if err != nil {
        log.Print(err)
    }
    fmt.Println(hexutil.Encode(publicKey)) // 0x04f17356fd52b0d13e5ede84f9

    message := whisperv6.NewMessage{
        Payload:   []byte("Hello"),
        PublicKey: publicKey,
        TTL:       60,
        PowTime:   2,
        PowTarget: 2.5,
    }

    messageHash, err := client.Post(context.Background(), message)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(messageHash) // 0xdbfc815d3d122a90d7fb44d1fc6a46f3d76e

    runtime.Goexit() // wait for goroutines to finish
}
```

工具函数

- 工具函数集合

工具函数集

函数的实现可以在这里. 它们一般接口比较通用. 这里我们先看几个例子。

检查地址是否是有效的以太坊地址:

```
valid := util.IsValidAddress("0x323b5d4c32345ced77393b3530b1eed0f3464
fmt.Println(valid) // true
```

检查地址是否为零地址:

```
zeroed := util.IsZeroAddress("0x0")
fmt.Println(zeroed) // true
```

将小数转换为wei(整数）。 第二个参数是小数位数。

```
wei := util.ToWei(0.02, 18)
fmt.Println(wei) // 20000000000000000
```

将wei（整数）转换为小数。 第二个参数是小数位数。

```
wei := new(big.Int)
wei.SetString("20000000000000000", 10)
eth := util.ToDecimal(wei, 18)
fmt.Println(eth) // 0.02
```

根据燃气上限和燃气价格计算燃气花费。

```
gasLimit := uint64(21000)
gasPrice := new(big.Int)
gasPrice.SetString("2000000000", 10)
gasCost := util.CalcGasCost(gasLimit, gasPrice)
fmt.Println(gasCost) // 42000000000000
```

从签名中提取R，S和V值。

```
sig := "0x789a80053e4927d0a898db8e065e948f5cf086e32f9ccaa54c1908e2
r, s, v := util.SigRSV(sig)
fmt.Println(hexutil.Encode(r[:])[2:]) // 789a80053e4927d0a898db8e065e948
fmt.Println(hexutil.Encode(s[:])[2:]) // 2621578113ddbb62d509bf6049b8fb54
fmt.Println(v)                        // 28
```

完整代码

util.go

```
package util

import (
    "math/big"
    "reflect"
    "regexp"
    "strconv"

    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/common/hexutil"
    "github.com/shopspring/decimal"
)

// IsValidAddress validate hex address
func IsValidAddress(iaddress interface{}) bool {
    re := regexp.MustCompile("^0x[0-9a-fA-F]{40}$")
    switch v := iaddress.(type) {
    case string:
        return re.MatchString(v)
    case common.Address:
        return re.MatchString(v.Hex())
    default:
        return false
    }
}

// IsZeroAddress validate if it's a 0 address
func IsZeroAddress(iaddress interface{}) bool {
    var address common.Address
    switch v := iaddress.(type) {
    case string:
        address = common.HexToAddress(v)
    case common.Address:
        address = v
    default:
        return false
    }

    zeroAddressBytes := common.FromHex("0x0000000000000000000000000
    addressBytes := address.Bytes()
    return reflect.DeepEqual(addressBytes, zeroAddressBytes)
}

// ToDecimal wei to decimals
func ToDecimal(ivalue interface{}, decimals int) decimal.Decimal {
    value := new(big.Int)
    switch v := ivalue.(type) {
    case string:
```

```
        value.SetString(v, 10)
    case *big.Int:
        value = v
    }

    mul := decimal.NewFromFloat(float64(10)).Pow(decimal.NewFromFloat(flo
    num, _ := decimal.NewFromString(value.String())
    result := num.Div(mul)

    return result
}

// ToWei decimals to wei
func ToWei(iamount interface{}, decimals int) *big.Int {
    amount := decimal.NewFromFloat(0)
    switch v := iamount.(type) {
    case string:
        amount, _ = decimal.NewFromString(v)
    case float64:
        amount = decimal.NewFromFloat(v)
    case int64:
        amount = decimal.NewFromFloat(float64(v))
    case decimal.Decimal:
        amount = v
    case *decimal.Decimal:
        amount = *v
    }

    mul := decimal.NewFromFloat(float64(10)).Pow(decimal.NewFromFloat(flo
    result := amount.Mul(mul)

    wei := new(big.Int)
    wei.SetString(result.String(), 10)

    return wei
}

// CalcGasCost calculate gas cost given gas limit (units) and gas price (wei)
func CalcGasCost(gasLimit uint64, gasPrice *big.Int) *big.Int {
    gasLimitBig := big.NewInt(int64(gasLimit))
    return gasLimitBig.Mul(gasLimitBig, gasPrice)
}

// SigRSV signatures R S V returned as arrays
func SigRSV(isig interface{}) ([32]byte, [32]byte, uint8) {
    var sig []byte
    switch v := isig.(type) {
    case []byte:
```

```
        sig = v
    case string:
        sig, _ = hexutil.Decode(v)
    }

    sigstr := common.Bytes2Hex(sig)
    rS := sigstr[0:64]
    sS := sigstr[64:128]
    R := [32]byte{}
    S := [32]byte{}
    copy(R[:], common.FromHex(rS))
    copy(S[:], common.FromHex(sS))
    vStr := sigstr[128:130]
    vI, _ := strconv.Atoi(vStr)
    V := uint8(vI + 27)

    return R, S, V
}
```

test file: util_test.go

词汇表

Addresses 地址

以来在以太坊中收发交易的地址，通常有数字和字母组成，也可以表示为一个二维码，由ECDSA密匙对衍生。

Agreement Ledgers 共识账本

由两方或者多方来进行协商和达成共识的记账本。

Altcoin 代币

比特币替代品的缩写。目前大部分的代币都是比特币经过一些对做工证明微小改动的的分叉。其中最引人瞩目的是莱特币。莱特币对原来比特币的改动包括减少出块时间，增加最大的币的数量，和一个不同的哈希算法。

Attestation Ledgers 查证账本

一个专门为协议，承诺或者声明等，提供证明存在证据的（用来认证）的分布式账本。

ASIC 应用特制集成电路

"Application Specific Integrated Circuit"应用特制集成电路的缩写。ASICs是专门为某个单独的应用设计的硅芯片。就比特币来说，它们通常指专门用来计算SHA-256来进行挖比特币的芯片。

Bitcoin 比特币

目前最著名的虚拟货币，基于做工证明（proof-of-work）。

Blockchain 区块链

一种不可篡改，用一个叫区块的结构数字化存储数据的分布式账本。每个区块再被数字签名“链”到下一个区块。这是的区块链可以用来做记账本，并切可以被有合适权限的个体分享，和使用。

Block Ciphers 区块加密器

一种对一整个区块的数据一起作为一个组群来进行用密匙和算法进行文本加密的的方法。而不是一个比特一个比特的算。

Block Height 区块高度

用来指区块链中链在一起的区块的个数。比如，高度0就是指第一个区块，通常也叫创世区块。

Block Rewards 区块奖励

当矿工成功算出一个块得到的奖励。取决于具体的币种，还有是否所有的币都已经被挖完，区块奖励可以是币和交易费的混合。目前比特币网络的区块奖励是25个比特币每个块。

Central Ledger 中央记账本

特制由一个中样机构维护的记账本。

Chain Linking 跨链

吧两天不同区块链连到一起，并且允许跨链交易的的过程。这会允许像比特币这类的区块链与其他侧链通信并进行资产交易。

Cipher 加密器

用来加密和解密信息的算法。通常讲，“cipher”也可以指一个加密消息，或者“编码”。

Confirmation 确认

区块种的交易会被区块链网络确认。在一个做工证明的系统（如比特币）中，这个确认过程也被称为挖矿。一旦一笔交易被确认，它就不可以被逆转或者双花。交易被区块记录后的块数的个数，也叫确认的次数。比如，交易是3个区块前被确认的，那么我们就说交易有三个“确认”。确认的数量越多，双花攻击的难度就越大。

Consensus Process 共识过程

一群负责维护一个分布式记账本的节点对记账本状态和内容达成共识的过程。

Consortium Blockchain 联盟链

一个共识过程有一组提前选出的节点控制的区块链。比如，你可以想象一个由15个金融机构组成的联盟，每个机构运营自己的节点，但是每个区块一定要有15个机构中的10个签名才能生效。也有混合模式，比如区块的根哈希是公开的，并且提供一个允许公共成员进行查询，并且获取加密证明部分区块链状态的的API。这类的区块链可以称为“半去中心化”。

Cryptoanalysis 加密分析学

一门通过分析，在不用私钥的情况下获取加密信息的学科。

Cryptocurrency 加密货币

一种基于数学的数字货币形式，其中加密技术用于规范货币单位的生成并验证资金的转移。 此外，加密货币独立于中央银行运作。

Cryptography 加密学

指加密和解密信息的过程。

dApp 去中心应用

去中心化的应用程序必须是完全开源的，它必须自主运行，并且没有实体控制其大部分代币。

DAO 去中心独立组织

（分散的自治组织）可以被认为是在不可破坏的业务规则的控制下的，没有任何人为控制运行的组织。

The DAO（一个风投基金的名字）

一个基于以太坊的风险投机构。它们造成了很多的软分叉和硬分叉。

Decryption 解密

将密文重新转换为明文的过程。

Encryption 加密

将明文消息（明文）转换为看起来像无意义的随机比特序列数据流（密文）的过程。

ERC 以太坊协商意见请求（Ethereum Request for Comments）的缩写

ERC是Ethereum Request for Comments的缩写. 一个ERC是一个对Ethereum的提案。

ERC-20 第20号ERC，目前大部分以太坊代币都遵守的规范

一个以太坊代币的标准。

Ether 以太币

以太坊区块链的原生代币，用于支付网络上的交易费，矿工奖励和其他服务。

Ethereum 以太坊

基于区块链技术的开放式软件平台，使开发人员能够编写智能合约，构建和部署分散式应用程序。

Ethereum Classic 以太坊经典版

一个经过硬分叉后的以太坊。

EVM 以太坊虚拟机

用来运行以太坊二进制码的以太坊虚拟机。

EVM Bytecode 以太坊虚拟机二进制码

以太坊区块链上的帐户可以包含的编程语言代码。每次向帐户发送消息时，都会执行与帐户关联的EVM代码，并且能够读取/写入存储，并自行发送消息。

Digital Commodity 数字商品

通常指稀缺，电子可转让，无形，具有市场价值的数字商品。

Digital Identity 数字身份

由个人，组织或电子设备在网络空间中采用或声明的在线或网络身份。

Distributed Ledgers 分布式记账本

一种分布在多个站点，国家或机构的数据库。 记录一个接一个地存储在连续的分类帐中。 分布式分类帐数据可以是“需许可”或“不需许可”。这指以控制（谁可以查看）账本的方式区分。

Difficulty 难度

指在Proof-of-Work挖矿中，挖掘区块链网络中的块的难度。 在比特币网络中，挖掘的难度每2016块都会调整。 这是为了将块验证时间保持在十分钟。快了就更难一点，慢了就简单一点。

Double Spend 双花

指比特币网络中的某个场景，有人试图同时向两个不同的收件人发送比特币交易。 但是，一旦比特币交易得到确认，就几乎不可能再花费它。 交易的确认越多，双倍花费比特币变得越困难。

Fiat currency 法币

是政府宣布为履行财务义务，如美元或欧元。

Fork 分叉

通过在网络的不同部分同时创建两个块，创建区块链的持续替代版本。
这会在分叉的节点之后创建了两个并行的区块链，其中一个是获胜的区块链。

Gas 以太坊“燃气”

一直以太坊的计量方式，大致相当于计算量步骤（对于以太坊）。 每项交易都必须包括燃气限额和愿意支付的单位燃气费用; 矿工会根据费用，去选择包括的交易。 每项操作都有燃气开支; 对于大多数操作而言，它是在3-10，尽管一些昂贵的操作有高达700的支出，而以太币交易本身的支出为21000个燃气。

Gas Cost “燃气” 花费

燃气费用是燃气上限乘以燃气价格。

Gas Limit ”燃气“上限

事务在智能合约执行中应该耗尽的最大燃气上限。

Gas Price ”燃气“ 价格

The price per computational unit.

Geth Go以太坊客户端的简称

一个Go实现的以太坊客户端。 https://github.com/ethereum/go-ethereum

Go 狗语言（笑）

Go是由Robert Griesemer，Rob Pike和Ken Thompson于2009年在Google创建的一种编程语言。

Golang Go编程语言

Go编程语言。

go-ethereum Go实现的以太坊客户端

Go实现的以太坊客户端项目名。

Halving （收益）减半

比特币的供应量有限，这使得它们成为稀缺的数字商品。 发行的比特币总量为2100万。 每个块产生的比特币数量每四年减少50%。 这称为“减半”。最后一半将在2140年进行。

Hard fork 硬分叉

通过对区块链协议的更改使先前无效的块/事务有效，因此需要所有用户升级其客户端。

Hashcash 哈希现金

用于限制电子邮件垃圾邮件和拒绝服务攻击的基于工作量证明的区块链系统，并且最近因其在比特币（和其他加密货币）中作为挖掘算法的一部分使用而闻名。

Hashrate 哈希速率（一种对算力的描述）

比特币矿工在给定时间内（通常是一秒钟）可以计算的哈希数。

HD Wallet 硬件钱包

硬件钱包或分级确定性钱包是一种新的数字钱包，它自动生成私有/公共地址（或密钥）的树状结构，从而解决用户必须自己生成它们的问题。

Infura 一个提供以太坊全节点服务

Infura为以太坊网络提供安全，可靠和可扩展的网关。 https://infura.io/

Initial Coin Offering 发币

（ICO）是一种新的加密货币从其整体币库中销售预付代币的事件，以换取前期资本。 ICO经常被用于新加密货币的开发者以筹集资金。

IPFS 星际文件系统的缩写

行星际文件系统（IPFS）是一种协议和网络，旨在创建一种在分布式文件系统中点对点的存储和共享，内容可寻址的，超媒体的的对等方法。

Keccak-256 一个以太坊常用加密哈希算法的名词

以太坊中使用的哈希算法。

Keystore 密匙库

包含加密钱包私钥和钱包元数据的文件。

Kovan 科文（一个以太坊测试网的名称）

一个基于权威证明的以太坊测试网. 只有Parity支持.

Ledger 记账本

仅允许附加记录存储（append only），其中记录是不可变的，并且记录除财务记录外更多的一般信息。

Litecoin 莱特币

一个基于Scrypt算法和做工证明的点对点加密货币。有时被称为加密货币的白银（如果比特币是黄金的话）。

Mining 挖矿

不断寻找符合标准的特殊哈希值，并验证交易并将其添加到区块链的过程。 使用计算硬件计算加密问题的这一过程也会触发加密货币的生成（挖矿奖励）。

Mnemonic 助记符（通常用来恢复密匙）

助记符短语，助记符恢复短语或助记符种子是用作生成HD钱包的主私钥和主链代码的种子的单词列表。

Multi-signature 多签名

（multisig）多签名地址允许对多方要求密钥来授权交易。 在创建地址时决定所需数量的签名。 多签名地址具有更大的防盗能力。

Node 节点

任何连接到区块链网络的计算机。

Nonce 去重数（一个只能用一次的数字）

一个只使用一次的数字。

Full node 全节点

一个完全执行区块链所有规则的节点。

Parity 奇偶（一个Rust实现的以太坊客户端）

用Rust语言编写的以太坊实现。 https://github.com/paritytech/parity

P2P 点对点的缩写

P2P 是Peer to Peer（点对点）的缩写.

Peer-to-peer 点对点

指在高度互连的网络中至少两方之间发生的去中心化的交互。 P2P参与者通过一个中介点直接相互交往。

Permissioned Ledger 需要请求权限的记账本

是一个分类帐，其中参与者必须具有访问分类帐的权限。 许可的分类帐可能有一个或多个所有者。 添加新记录时，有限共识流程会检查分类帐的完整性。 这是由可信赖的参与者 - 例如政府部门或银行 - 进行的，这使得保持共享记录比未经许可的分类账使用的共识过程简单得多。

Permissioned Blockchains 需要请求权限的区块链

提供高度可验证的数据集，因为共识流程创建了数字签名，所有各方都可以看到。

Private Key 私钥

一串数据，显示您可以支配特定钱包中的比特币。 私钥可以被认为是密码; 除了您之外，不要向任何人泄露私钥，因为它们允许您通过加密签名从比特币钱包中使用比特币。

Proof of Authority 权威证明

私有区块链中的共识机制，它基本上为一个客户端（或特定数量的客户端）提供了一个特定私钥，以便在区块链中生成所有块。

Proof of Stake 利权证明

工作量证明系统的替代方案，其中您使用加密货币的现有股份（您持有的货币金额）来计算您可以开采的货币金额。

Proof of Work 做工证明

将采矿能力与计算能力联系起来的系统。 必须对块进行散列，这本身就是一个简单的计算过程，但是在散列过程中添加了一个额外的变量，使其更加困难。 成功散列块时，散列必须花费一些时间和计算量。 因此，散列块被认为是工作的证据。

Protocols 协议

描述如何传输或交换数据的正式规则集，尤其是在网络中。

Rinkeby 一个以太坊测试网的名称

以太坊区块链上的权威证明的测试网。 仅由Geth支持。

RLP 递归长度前缀编码标准的缩写

Recursive Length Prefix (RLP) 是一种编码任意嵌套的二进制数据数组的标准。 RLP是用于序列化以太坊中对象的主要编码方法。

Ropsten 一个做工证明的以太坊测试网

以太坊区块链上的工作证明的测试网，可以最好地模拟生产环境。 由Geth和Parity支持。

Scrypt 一个做工证明的哈希算法

SHA-256的另一种工作系统证明，旨在对CPU和GPU矿工特别友好，同时为尽力无视ASIC矿工提的优势。

SHA256 一个加密哈希方程的名称（应用于比特币）

一个加密函数，比特币工作系统证明的基础。

Signature 签名

数字签名是用于呈现数字消息或文档的真实性的数学方案。

Smart contract 智能合约

合约的条款以计算机语言或代码，而非法律语言记录。 智能合约可以由计算系统自动执行，例如合适的分布式分类帐系统。

Soft fork 软分叉

对比特币协议的改变，其中仅使先前有效的块/事务无效。 由于旧节点将新块识别为有效，因此softfork是向后兼容的。 这种分支只需要大多数矿工升级来执行新规则。

Sokol 一个只有Parity 客户端支持的以太坊测试网

一个以太坊区块链上的权威证明的测试网。 仅受Parity客户端支持。

Stream ciphers 数据流加密器

一种加密文本（密文）的方法，其中加密密钥和算法一次一位地应用于数据流中的每个二进制数字。

Swarm 直译为虫群，以太坊中的分布式文件存储组件

去中心化的文件存储，以太坊的一部分。

Token 代币

是可以拥有的东西的数字身份。

Tokenless Ledger 无代币记账本

指的是不需要本地货币运行的分布式分类帐。

Transaction Block 交易块

比特币网络上的一系列交易，聚集到一个块中，然后可以对其进行哈希处理并添加到区块链中。

Transaction Fees 交易费

通过比特币网络发送的一些交易收取的小额费用。 交易费用将授予成功包含相关交易的区块的矿工。

Unpermissioned ledgers 无权限记账本

没有单一所有者的区块链; 不能被任何个体拥有。 无需许可的分类帐的目的是允许任何人向分类帐和拥有分类帐的每个人提供相同副本的数据。

Wallet 钱包

包含私钥集合的文件。

Whisper 直译为悄悄话，以太坊中的一个点对点消息系统

作为以太坊的一部分的，点对点消息传递系统。

资源

Ethereum, Solidity 还有 Go的学习资源列表。

参考规范

- 智能合约参考规范
- 安全考虑
- Solidity的一些坑

寻求帮助和问题解答

- StackExchange
  - 以太坊
- Reddit
  - ethdev
  - golang
- Gitter
  - List of Gitter channels

社区

- Reddit
  - ethereum
  - ethtrader
- Twitter
  - ethereum

库

- go-ethereum
- go-solidity-sha3
- go-ethereum-hdwallet
- go-ethutil

开发者工具

- Truffle
- Infura
- Remix IDE
- Keccak-256 Online