交易是区块链中最基本也是最核心的一个概念，在以太坊中，交易更是重中之重，因为以太坊是一个智能合约平台，以太坊上的应用都是通过智能合约与区块链进行交互，而智能合约的执行是由交易触发的，没有交易，智能合约就是一段死的代码，可以说在以太坊中，一切都源于交易。下面就来看看在以太坊中交易是什么样的，交易里面都有什么。

交易的数据结构

在 core/types/transaction.go 中定义了交易的数据结构：

```
type Transaction struct {
    data txdata
    // caches
    hash atomic.Value
    size atomic.Value
    from atomic.Value}
```

在这个结构体里面只有一个 data 字段，它是 txdata 类型的，其他的三个字段 hash size from 是缓存字段，txdata 也是一个结构体，它里面定义了交易的具体的字段：

```
type txdata struct {
    AccountNonce       uint64
    Price, GasLimit *big.Int
    Recipient          *common.Address `rlp:"nil"` // nil means contract creation
    Amount               *big.Int
    Payload             []byte
    V                    *big.Int // signature
    R, S                *big.Int // signature}
```

各字段的含义如下：

AccountNonce：此交易的发送者已发送过的交易数

Price：此交易的 gas price

GasLimit：本交易允许消耗的最大 gas 数量

Recipient：交易的接收者，是一个地址

Amount：交易转移的以太币数量，单位是 wei

Payload：交易可以携带的数据，在不同类型的交易中有不同的含义

V R S：交易的签名数据

注意：这里并没有一个字段来指明交易的发送者，因为交易的发送者地址可以从签名中得到。

在 transaction.go 中还定义了一个 jsonTransaction 结构体，这个结构体用于将交易进行 json 序列化和反序列化，具体的序列化和反序列化可以参照 MarshalJSON 和 UnmarshalJSON 函数。以太坊节点会向外部提供 JSON RPC 服务，供外部调用，RPC 服务通过 json 格式传输数据，节点收到 json 数据后，会转换成内部的数据结构来使用。jsonTransaction 结构体使用 go 语言的 struct tag 特性指定了内部数据结构与 json 数据各字段的对应关系，例如内部的 AccountNonce 对应 json 的 nonce，Amount 对应 json 的 value。web3.js 的 eth.getTransaction() 和 eth.sendTransaction()使用的数据就是 json 格式的，根据这个结构体就可以知道在 web3.js 中交易的各个字段与程序内部的各个字段的对应关系。

```
type jsonTransaction struct {
    Hash             *common.Hash      `json:"hash"`
    AccountNonce *hexutil.Uint64 `json:"nonce"`
    Price            *hexutil.Big      `json:"gasPrice"`
    GasLimit         *hexutil.Big      `json:"gas"`
```

```
    Recipient       *common.Address `json:"to"`
    Amount          *hexutil.Big    `json:"value"`
    Payload         *hexutil.Bytes  `json:"input"`
    V               *hexutil.Big    `json:"v"`
    R               *hexutil.Big    `json:"r"`
    S               *hexutil.Big    `json:"s"`}
```

注：Payload 这个字段在 eth.sendTransaction()中对应的是 data 字段，在 eth.getTransaction()中对应的是 input 字段。

交易的 Hash

下面是计算交易 Hash 的函数，它是先从缓存 tx.hash 中取，如果取到，就直接返回，如果缓存中没有，就调用 rlpHash 计算 hash，然后把 hash 值加入到缓存中。

```
// Hash hashes the RLP encoding of tx.// It uniquely identifies the transaction.
func (tx *Transaction) Hash() common.Hash {
    if hash := tx.hash.Load(); hash != nil {
        return hash.(common.Hash)
    }
    v := rlpHash(tx)
    tx.hash.Store(v)
    return v}
```

rlpHash 的代码在 core/types/block.go 中：

```
func rlpHash(x interface{}) (h common.Hash) {
    hw := sha3.NewKeccak256()
    rlp.Encode(hw, x)
    hw.Sum(h[:0])
    return h}
```

从 rlpHash 函数可以看出，计算 hash 的方法是先对交易进行 RLP 编码，然后计算 RLP 编码数据的 hash，具体的 hash 算法是 Keccak256。

那么到底是对交易中的哪些字段计算的 hash 呢？这就要看 rlp.Encode 对哪些字段进行了编码。rlp.Encode 代码在 rlp/encode.go 中，不用看具体的实现，在注释中有这么一段：

```
// If the type implements the Encoder interface, Encode calls// EncodeRLP. This is true even for nil pointers, please see the// documentation for Encoder.
```

就是说如果一个类型实现了 Encoder 接口，那么 Encode 函数就会调用那个类型所实现的 EncodeRLP 函数。所以我们就要看 Transaction 这个结构体是否实现了 EncodeRLP 函数。回到 core/types/transaction.go 中，可以看到 Transaction 确实实现了 EncodeRLP 函数：

```
// DecodeRLP implements rlp.Encoder
func (tx *Transaction) EncodeRLP(w io.Writer) error {
    return rlp.Encode(w, &tx.data)}
```

从这可以看出交易的 hash 实际上是对 tx.data 进行 hash 计算得到的：txhash=Keccak256(rlpEncode(tx.data))。

交易的类型

在源码中交易只有一种数据结构，如果非要给交易分个类的话，我认为交易可以分为三种：转账的交易、创建合约的交易、执行合约的交易。web3.js 提供了发送交易的接口：

```
web3.eth.sendTransaction(transactionObject [, callback])
```

参数是一个对象，在发送交易的时候指定不同的字段，区块链节点就可以识别出对应类型的

交易。

转账的交易

转账是最简单的一种交易，这里转账是指从一个账户向另一个账户发送以太币。发送转账交易的时候只需要指定交易的发送者、接收者、转币的数量。使用 web3.js 发送转账交易应该像这样：

```
web3.eth.sendTransaction({
    from: "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
    to: "0xd46e8dd67c5d32be8058bb8eb970870f07244567",
    value: 10000000000000000});
```

value 是转移的以太币数量，单位是 wei，对应的是源码中的 Amount 字段。to 对应的是源码中的 Recipient。

创建合约的交易

创建合约指的是将合约部署到区块链上，这也是通过发送交易来实现。在创建合约的交易中，to 字段要留空不填，在 data 字段中指定合约的二进制代码，from 字段是交易的发送者也是合约的创建者。

```
web3.eth.sendTransaction({
    from: "contract creator's address",
    data: "contract binary code"});
```

data 字段对应的是源码中的 Payload 字段。

执行合约的交易

调用合约中的方法，需要将交易的 to 字段指定为要调用的合约的地址，通过 data 字段指定要调用的方法以及向该方法传递的参数。

```
web3.eth.sendTransaction({
    from: "sender's address",
    to: "contract address",
    data: "hash of the invoked method signature and encoded parameters"});
```

data 字段需要特殊的编码规则，具体细节可以参考 Ethereum Contract ABI。自己拼接字段既不方便又容易出错，所以一般都使用封装好的 SDK（比如 web3.js）来调用合约。