

# web3.js开发文档--前端

## 1.web3.js

<https://web3js.readthedocs.io/en/v1.5.2/>

以太坊Web3.js库，Web3.js库是一个javascript库，用于与以太坊区块链进行交互。

web3.js 库是一系列模块的集合，服务于以太坊生态系统的各个功能，如：

- web3-eth 用来与以太坊区块链及合约的交互；
- web3-shh Whisper 协议相关，进行p2p通信和广播；
- web3-bzz swarm 协议（去中心化文件存储）相关；
- web3-utils 包含一些对 DApp 开发者有用的方法。

## 2.开发工具

node@v16.13.1

vue/cli @v4.5.15

vue@2.0

web3@ 1.5.2

ethereumjs-tx:@1.3.7

## 3.部署工作

### 3.1准备MetaMask交易账号

<https://support.metamask.io/>

MetaMask是一款插件型（无需下载客户端）轻量级数字货币钱包，主要用于Chrome谷歌浏览器和火狐浏览器Firefox。

前往网站搜索MetaMask小狐狸钱包，下载后注册好钱包。（<https://zhuanlan.zhihu.com/p/368736357>）

#### 3.1.1MetaMask添加网络

- 1 后端提供：
- 2 矿机测试服：<http://192.168.11.120:8548/>
- 3 链ID：84350
- 4 合约地址：[0x22C628359aa70190fcaeb4009D2a7F6C61EF8a9](#)

< 网络 ×

网络名称

|自定义名称

新增 RPC URL

对应上方测试服地址

链 ID ⓘ

对应链ID

符号 (选填)

区块浏览器 URL (选填)

取消

保存

### 3.1.2MetaMask添加代币



contract ▾



# 添加代币

自定义代币

代币合约地址

| 对应上方合约地址

代币符号

自动识别

小数精度

0

取消

下一步

## 3.1.3 MetaMask 获取账号和私钥



## 3.2创建vue项目

- 1 vue create webdemo
- 2 下载相应需求:
- 3 yarn add web3.js@1.5.2//web3
- 4 yarn add ethereumjs-tx@1.3.7//ethereumjs-tx

### ethereumjs-tx:

根据区块链工作原理，创建交易时，会签署交易然后向网络广播。为了签署交易，我们使用JavaScript库ethereumjs-tx。

使用这个库的目的是，可以在本地签署交易。

要在本地签署交易，可以在本地运行自己的以太坊节点，这样就不必使用ethereumjs-tx库了。但是，如前所述，本地运行节点比较麻烦，需要同步区块链数据，相当繁琐。测试阶段使用自己的私网链接来管理。

如果在远程节点签署交易，就需要让远程节点管理我们的私钥，这是有风险的。所以最终我们选择了ethereumjs-tx来签署本地交易。

## 4.检查钱包

### 4.1检查是否安装钱包与连接网络

<http://cw.hubwiz.com/card/c/metamask-api/1/2/1/>

```
1  // 引入web3
2  import Web3 from "web3";
3
4  //写在 created()里面
5  //判断用户是否安装MetaMask钱包插件
6  if (typeof window.ethereum === "undefined") {
7    //没安装MetaMask
8    console.log("请安装MetaMask钱包")
9  } else {
10   //如果用户安装了MetaMask，你可以要求他们授权应用登录并获取其账号
11   window.ethereum.enable().catch(function (reason) {
12     //如果用户拒绝了登录请求
13     if (reason === "User rejected provider access") {
14       // 用户拒绝登录后执行语句；
15       console.log("请授权")
16     } else {
17       // 本不该执行到这里，但是真到这里了，说明发生了意外
18       console.log("There was a problem signing you in");
19     }
20   }).then(async (accounts) => {
21     // 判断是否连接以太或者自定义网络
22     //84350是私网的链ID
23     if (window.ethereum.networkVersion !== "84350") {
24       console.log("请连接xx网络", accounts);
25     } else {
26       // 获取账号和余额
27       const web3 = new Web3(window.web3.currentProvider);
28       const [acc] = await web3.eth.getAccounts(function () {});
29       web3.eth.getBalance(acc, (err, wei) => {
30         //这是MetaMask钱包目前第一个的账户
31         let account = acc;
32         //这里获取的余额是ETH的，不能获取智能合约上添加的，如需要要从智能合约中调用函数获取
```

```
33     //余额单位从wei转换为ether
34     let balance = web3.utils.fromWei(wei, "ether");
35     console.log(account,balance)
36   });
37 }
38 });
39 }
```

## 5.调用智能合约

### 5.1使用密钥进行本地签署交易

本次演示使用的ethereumjs-tx，安装的ethereumjs-tx就是本地签署作用，本地签署交易是使用密钥进行签署。

```
1  // 引入web3.js@1.5.1
2  import Web3 from "web3";
3  // 测试服地址
4  //const rpcURL = "http://192.168.11.120:8548";
5  // 建立Web3连接
6  //const web3 = new Web3(rpcURL);
7  //上方为老方法，使用web3最新规定：返回当前有效的通信服务提供器。
8  const web3 = new Web3(
9    Web3.givenProvider ||
10     new Web3.providers.WebsocketProvider("ws://192.168.11.88:8545")
11  );
12  // 智能合约生成的ABI参数，调用智能合约里的函数
13  // ABI参数由后端提供，一般为json格式
14  import { abi } from "../assets/ABI/ABI.json";
15  // 合约地址
16  const address = "0x22C628359aa70190fcaeb4009D2a7F6C61EF8a9";
17  // 通证智能合约的Javascript对象：
18  const contract = new web3.eth.Contract(abi, address);
19  // 签署交易的JavaScript库ethereumjs-tx@1.3.7
20  import Tx from "ethereumjs-tx";
21  // 本机账户，前面已经获取到了，直接传过来就可以用，这里作展示账号
22  const account1 = "0xE9f6d5F43b6D61d6cC146aafCB3E5Af3C8f774E5"
23  // 账户密钥，前面已经从账户中导出
24  const pk1 = "从账户中导出"; // 实际项目中应该从process.env.PRIVATE_KEY_1中读取
25  //密钥转码
```

```
26 const privateKey1 = Buffer.from(pk1, "hex");
27
```

### 5.1.1.call5.1.2

call 用来调用 view 和 pure 函数。它只运行在本地节点，不会在区块链上创建事务，也不会消耗任何 gas。用户也不会被要求用MetaMask对事务签名。

使用 Web3.js, 你可以如下 call 一个名为myMethod的方法并传入一个 123 作为参数:

```
1 // 参数from: 调用者的地址, call查到的返回值返回给这个地址
2 myContract.methods.myMethod("需要传入的参数").call({from: "你的账户"})
```

### 例：查询余额

```
1 //利用智能合约可以直接使用call方法查询自己添加的代币的余额,就不需要调用上面封装的contractCall方法
2 //balanceOf是智能合约中的方法, 直接调用。account1是我的账户, 可以直接查询账户余额
3 contract.methods.balanceOf(account1).call({ from: account1 }, (err, result) => {
4   if (err) {
5     console.log(err, "err");
6   }
7   console.log("balanceOf", err, result);
8   //获取到余额并赋值
9   let balance = result / 100;
10 });
```

### 5.1.2.send

send 将创建一个事务并改变区块链上的数据。你需要用 send 来调用任何非 view 或者 pure 的函数。

注意: send 一个事务将要求用户支付gas, 并会要求弹出对话框请求用户使用 Metamask 对事务签名。在我们使用 Metamask 作为我们的 web3 提供者时, 所有这一切都会在我们调用 send() 的时候自动发生。而我们自己无需在代码中操心这一切, 挺爽的吧。

使用 Web3.js, 你可以如下 send一个事物调用myMethod的方法并传入一个 123 作为参数:

```
1 myContract.methods.myMethod("需要传入的参数").send()
```

事例: 我们这里测试开发使用send调用智能合约遇到了问题, 他会报错unknown account, 原因应该是需要账户解锁或者交易没有签名造成的, 目前没有找到好的解决办法。所以, 我们使用的通过私钥来签署交易并发布, 具体操作如下:

## 例：质押押金

```
1 // 质押押金
2 async pledge(money1) {
3     //调用上方contractCall函数。因为使用send调用，需要私钥签署
4     // 传入合约需要参数data，首次调用函数，
5     let data1 = contract.methods.betForGame(money1 * 100).encodeABI();
6     await this.contractCall(data1, account1); //account1是本机账户
7     // 再次使用call调用函数查询交易结果
8     //注意，这里查询的交易结果假结果，因为交易结果是异步的，现在还没有上链，所以后端此时同步返回给你的是一个假结果，要上链之后在进行结果查询，下面会说到。
9     //这里查询该方法主要是用于错误回调，获取到catch里面的错误事件回调。
10    contract.methods.betForGame(money1 * 100).call({ from: account1 }).then((res, err) => {
11        // 交易结果成功,返回交易数据，可能为假数据
12        // console.log(res);
13        // console.log(err);
14        }).catch((err) => {
15            // 函数执行失败，可能重复质押或者押金不够
16            console.log("这里返回的失败:", err, err.message);
17        });
18    },
```

## 例：与合约游戏

```
1 //与合约游戏
2 async goContract() {
3     //调用上方contractCall函数。因为使用send调用，需要私钥签署
4     // 传入合约需要参数data，首次调用函数，
5     let data2 = contract.methods.startGameToContract().encodeABI();
6     // function startGameToContract() external returns (uint256,uint256,int256)
7     await this.contractCall(data2, account1);
8     // 查询交易结果，与上方解释一样
9     contract.methods.startGameToContract().call({ from: account1 }).then((res, err) => {
10        // 交易结果成功,返回交易数据，可能为假数据
11        console.log(res);
12        // console.log(err);
13    }).catch((err) => {
14        // 函数执行失败，可能没有质押或者已经游戏完成
```



```

15     console.log("这里返回的失败:", err);
16   });
17 },

```

## 例：与玩家游戏

```

1  async goOpponent() {
2    //调用上方contractCall函数。因为使用send调用，需要私钥签署
3    // 传入合约需要参数data，首次调用函数，
4    let data3 = contract.methods.startGameToOtherOne().encodeABI();
5    //function startGameToOtherOne() external returns (uint256,uint256,int256)
6    await this.contractCall(data3, account1);//account1是本机账户
7    // 查询交易结果，与上方解释一样
8    contract.methods.startGameToOtherOne().call({ from: account1 }).then((res, err) => {
9      // 交易结果成功,返回交易数据，可能为假数据
10     // console.log(res);
11     // console.log(err);
12     }).catch((err) => {
13       // 函数执行失败，可能没有质押或者已经游戏完成
14       console.log("这里返回的失败:", err);
15     });
16 },

```

## 5.1.3本地签署交易是使用密钥进行签署

```

1  // 合约函数
2  contractCall(data, account1) {
3    //现在为nonce变量赋值，可以使用web3.eth.getTransactionCount()函数获取交易nonce。
4    web3.eth.getTransactionCount(account1, async (err, txCount) => {
5      // 构建交易对象
6      const txObject = {
7        // nonce:这是账号的前一个交易计数。这个值必须是十六进制
8        nonce: web3.utils.toHex(txCount),
9        // gasLimit: 消耗Gas上限。部署合约比转账会消耗更多Gas，需要提高上限
10       gasLimit: web3.utils.toHex(200000),
11       // gasPrice: Gas价格，这里是 5 Gwei。
12       gasPrice: web3.utils.toHex(web3.utils.toWei("5", "gwei")),
13       // to:此参数将是已部署智能合约的地址
14       to: address,

```

```

15 // data:被调用的智能合约函数的十六进制表示, 使用web3.js函数encodeABI()
16 // 例如: contract.methods.函数方法("需要传入的参数").encodeABI()
17 // 这里会通过下方函数传过来
18 data: data,
19 // chainId:连接智能合约的主网
20 chainId: web3.utils.toHex("84350"),
21 };
22 // 签署交易
23 const tx = new Tx(txObject);
24 tx.sign(privateKey1);
25 const serializedTx = tx.serialize();
26 const raw = "0x" + serializedTx.toString("hex");
27 // console.log(raw);
28 // 广播交易, 将这个已签名的序列化交易发送到测试网络
29 // web3.eth.sendSignedTransaction(raw, (err, txHash) => {
30 //   console.log("txHash:", err, txHash);
31 // });
32 // web3.eth.sendTransaction(raw).on("receipt", console.log).on('error',
console.error);
33 await web3.eth
34   .sendSignedTransaction(raw)
35   .on("receipt", console.log)
36   .then((res) => {
37     // 成功回调, 获取到成功参数
38     // console.log(res, err);
39     // 通过订阅事件查询, 获取到blockNumber区块号和transactionHash值, 通过筛选出对应
    的值
40     // 然后将查询到的结果进行判断, 展示
41     contract.getPastEvents(
42       "AllEvents", // 过滤事件参数, 这里获取全部事件
43       {
44         filter: { // 可选, 按索引参数过滤事件 },
45         fromBlock: 0, // 起始块
46         toBlock: 'latest', // 终止块
47       },
48       (err, events) => {
49         // 获取到所有事件
50         console.log(events);
51 // 这里我们可以通过filter筛选再通过上方回调成功的res获取的值, 将起始块和终止块规定, 也和与将过滤
    事件参数传进去, 进行规范查询
52 // 再通过js遍历筛选出所要的那一条

```

```

53         }
54     );
55 })
56     .catch((err) => {
57         //错误回调 上链失败
58         console.log("上链失败:", err);
59     });
60 });
61 },
62

```

## 5.2使用钱包交易

本次演示使用的是**MetaMask小狐狸钱包**，通过调用合约或者发起交易，唤起钱包，帮我们进行本地签署，规范gasPrice，自动调节网络等功能。

但是每一次调用合约，就需要在小狐狸上面签字一次，用户体验不好，并且用户没有钱包怎么办?但是很大程度上解决了私钥的问题!

```

1  import Web3 from "web3";
2  import { abi } from "../assets/ABI/ABI.json";
3  // 合约地址
4  const address = "0xcAF35BA550ADFF051767f61bf8b1C6407CA2d150";
5  // 我们只要装了metamask插件，那么浏览器中就被注入了web3,就能使用此方法直接唤起钱包使用
6  // 不使用密钥签字，调用当前的钱包发起交易和调用智能合约
7  const web3 = new Web3(window.web3.currentProvider);
8  // 通证智能合约的Javascript对象:
9  const contract = new web3.eth.Contract(abi, address);
10 //本机账户,这里只做展示，前面已经获取到了
11 const account1 = "0xE9f6d5F43b6D61d6cC146aafCB3E5Af3C8f774E5"

```

### 5.2.1call

解释与方法参考上方本地签署的call方法

call方法其实和本地签署的调用是一样的，因为他不会改变区块链上的事件，只是查询事件，所以不用消耗油费和进行签署。

#### 例：查询余额

```

1  //利用智能合约可以直接使用call方法查询自己添加的代币的余额,就不需要调用上面封装的contractCall方法
2  //balanceOf是智能合约中的方法，直接调用。account1是我的账户，可以直接查询账户余额

```

```

3 contract.methods.balanceOf(account1).call({ from: account1 }, (err, result) => {
4   if (err) {
5     console.log(err, "err");
6   }
7   console.log("balanceOf", err, result);
8   //获取到余额并赋值
9   let balance = result / 100;
10 });

```

### 5.2.2.send

解释与方法参考上方本地签署的call方法

send方法和本地签署的调用是一样的，主要是因为钱包帮我们进行了本地签署等一系列事件，所以我们代码就很方便。

#### 例：质押押金

```

1 // 质押押金betForGame
2 async pledge(money1) {
3   //直接调用智能合约函数，使用send方法，传入需要参数，通过then接受到成功失败回调
4   await contract.methods.betForGame(money1 * 100).send({ from: account1 }).then((res,
5     err) => {
6     // 成功回调
7     console.log(res, err);
8     // console.log(err);
9   }).catch((res, err) => {
10     // 错误回调
11     console.log(res, err);
12   });

```

#### 例：与合约游戏

```

1 //与合约游戏
2 async goContract() {
3   //直接调用智能合约函数，使用send方法，传入需要参数，通过then接受到成功失败回调
4   await contract.methods.startGameToContract().send({ from: account1 }).then((res, err)
5     => {
6     // 交易结果成功,返回交易数据
7     console.log(res);
8     // console.log(err);
9   }).catch((err) => {

```

```

9      // 函数执行失败，可能没有质押或者已经游戏完成
10     console.log("这里返回的失败:", err);
11   });
12 },

```

## 例：与玩家游戏

```

1  async goOpponent() {
2    //直接调用智能合约函数，使用send方法，传入需要参数，通过then接受到成功失败回调
3    await contract.methods.startGameToOtherOne().send({ from: account1 }).then((res, err)
=> {
4      // 交易结果成功,返回交易数据
5      console.log(res);
6    }).catch((err) => {
7      // 函数执行失败，可能没有质押或者已经游戏完成
8      console.log("这里返回的失败:", err);
9    });
10 },

```

## 6.查询历史记录

### 6.1 web3.eth.getTransactionReceipt

返回指定交易的收据，使用哈希指定交易。

需要指出的是，挂起的交易其收据无效。

```

1  // 通过地址查询交易详情
2  web3.eth.getTransactionReceipt(
3    // transactionHash
4    "0x6980b2a6ac510b50f8b0a742e8369db5da07891fca7f3755ef49bc1d2b28f1e5"
5  )
6  .then((res) => {
7    // console.log(`Status: ${res.status}`);
8    console.log(res);
9  });

```

### 返回值

```

1  //解释说明：
2  Object - 交易收据对象，如果收据不存在则为null。交易对象的结构如下：
3  transactionHash: DATA, 32字节 - 交易哈希

```

[illegible]

[illegible]

## 6.2通过智能合约方法contract.getPastEvents查询历史事件

## 读取合约历史事件

```

1 import Web3 from "web3";
2 const web3 = new Web3(
3   Web3.givenProvider ||
4     new Web3.providers.WebsocketProvider("ws://192.168.11.88:8545")
5 );
6 import { abi } from "../assets/ABI/ABI.json";
7 const address = "0x22C628359aa70190fcaeb4009D2a7F6C61EF8a9";
8 const contract = new web3.eth.Contract(abi, address);
9
10 contract.getPastEvents(
11   "allEvents", // 过滤事件参数，这里获取全部事件
12   {
13     filter: {
14       //可选，按索引参数过滤事件
15     },
16     // 可选，仅读取从该编号开始的块中的历史事件,默认值为0
17     fromBlock: 0, // 起始块
18     // 可选，仅读取截止到该编号的块中的历史事件，默认值为"latest"
19     toBlock: "latest", // 终止块
20   },
21   (err, events) => {
22     //所有事件
23     console.log(events);
24   }
25 )
26

```

```
27 参数:
28 event - String: 事件名, 或者使用 "allEvents" 来读取所有的事件
29 options - Object: 用于部署的选项, 包含以下字段:
30 filter - Object : 可选, 按索引参数过滤事件, 例如 {filter: {myNumber: [12,13]}} 表示所有
    "myNumber" 为12 或 13的事件
31 fromBlock - Number : 可选, 仅读取从该编号开始的块中的历史事件。
32 toBlock - Number : 可选, 仅读取截止到该编号的块中的历史事件, 默认值为"latest"
33 topics - Array : 可选, 用来手动设置事件过滤器的主题。如果设置了filter属性和事件签名, 那么
    (topic[0])将不会自动设置
34 callback - Function : 可选的回调参数, 触发时其第一个参数为错误对象, 第二个参数为历史事件数组
```

返回值: 一个Promise对象, 其解析值为历史事件对象数组

```
1 //解释说明:
2 Object - 交易收据对象, 如果收据不存在则为null。交易对象的结构如下:
3 address: 合约哈希地址
4 blockHash: 上链区块哈希地址
5 blockNumber: 上链区块值
6 event: 返回事件
7 id: "log_3a834fcd"
8 logIndex: 0
9 raw: {data: '0x0000000000000000000000000000000000000000000000000000000000000000...
    0000000000000000000000000000000000000000000000000000000000000012c', topics: Array(1)}
10 removed: false
11 returnValues: 智能合约规定返回的参数 (可理解为后端传值)
12 signature: 签署哈希值
13 transactionHash: DATA, 32字节 - 交易哈希
14
15
16 //实例返回
17 address: "0xcAF35BA550ADFF051767f61bf8b1C6407CA2d150"
18 blockHash: "0xd320c0110a4f5249a02a8f9348cfdcde282a7b86a4a7fd4e17aee6c1b737857e"
19 blockNumber: 61499
20 event: "gameResult"
21 id: "log_3a834fcd"
22 logIndex: 0
23 raw: {data: '0x0000000000000000000000000000000000000000000000000000000000000000...
    0000000000000000000000000000000000000000000000000000000000000012c', topics: Array(1)}
24 removed: false
25 returnValues: Result {0: '9', 1: '4', 2: '700', 3: '300'}
26 signature: "0xbcb29f0ac81def54c5ba9670bc3f93ea72a0b715f4b37f65e540a37f6d5d9f85"
27 transactionHash: "0xea137ff3407489a6dc63744e6ab4c45ccbfc40a77ce4d8e051306243550023d7"
```



## 技术支持:

十二生肖网络科技有限公司

技术部: 总监: 吕磊

前端: 陆俊佑, 杨加骏