**neon-js**

neon-js是与NEO区块链交互的JS库。它支持：

**-** Wallet methods such as key generation, manipulation and encryption.

**-** Transaction creation, serialization and deserialization.

**-** Smart contract calling **for** NEP**-**5.

**-** Various APIs needed to support a light wallet.

**快速入门**

以语义形式导入默认模块以提供全部功能：

**import** Neon from '@cityofzion/neon-js'

**const** account **=** Neon.create.account('L1QqQJnpBwbsPGAuutuzPTac8piqvbR1HRjrY5qHup48TBCBFe4g')

Neon.create.query({method**:** 'getrawtransaction', params**:** ['bfb3af8bc96ae3dd85305eddfe6e8b8eca447315729073f30da64f80c16f66ac']})

.execute(Neon.CONST.DEFAULT\_RPC.MAIN)

.then((res) => {

console.log(Neon.serialize.tx(res.result))

})

Neon.api.getBalance("TestNet", account.address).then((response) => {

console.log(response)

})

使用命名导入进行细粒度控制和摇树（tree-shaking）：

**import** { wallet, tx, rpc, api, CONST } from '@cityofzion/neon-js'

**const** account **=** **new** wallet.Account('L1QqQJnpBwbsPGAuutuzPTac8piqvbR1HRjrY5qHup48TBCBFe4g')

rpc.Query.getRawTransaction('bfb3af8bc96ae3dd85305eddfe6e8b8eca447315729073f30da64f80c16f66ac')

.execute(CONST.DEFAULT\_RPC.MAIN)

.then((res) => {

tx.serializeTransaction(res.result)

})

api.getBalance("TestNet", account.address).then((response) => {

console.log(response)

})

### 格式

本文档旨在用作库的高级解释。这个文档的布局将严格遵循源代码。我们强烈建议您阅读源代码，因为它也被评论。

**目录:**

* 概述
  + 特征
  + 用法
* 安装
  + 安装
  + Import
  + Require
  + 网络
* 界面
  + 余额
  + 币
  + [Claim](http://cityofzion.io/neon-js/interface.html#claim)
* 钱包
  + 账号
  + 余额
  + Core
  + [NEP2](http://cityofzion.io/neon-js/wallet.html#nep2)
  + 验证
  + 钱包
* 交易
  + 交易
  + 组件
  + 类型
* [节点 RPC](http://cityofzion.io/neon-js/rpc/index.html)
  + [RPC 客](http://cityofzion.io/neon-js/rpc/index.html#rpc-client)户端
  + 查询
* [API](http://cityofzion.io/neon-js/api/index.html)
  + 核
  + [NeonDB](http://cityofzion.io/neon-js/api/index.html#neondb)
  + [Neoscan](http://cityofzion.io/neon-js/api/index.html#neoscan)
  + [CoinMarketCap](http://cityofzion.io/neon-js/api/index.html#coinmarketcap)
  + [NEP5](http://cityofzion.io/neon-js/api/index.html#nep5)
* 智能合约
  + [Script Builder](http://cityofzion.io/neon-js/smart_contracts/index.html#script-builder)
  + [ContractParam](http://cityofzion.io/neon-js/smart_contracts/index.html#contractparam)
* 效用
  + 格式
  + [Hashing](http://cityofzion.io/neon-js/utility.html#hashing)
  + 效用类
* 常量
* 参考
  + [API](http://cityofzion.io/neon-js/reference/api.html)
  + [RPC](http://cityofzion.io/neon-js/reference/rpc.html)
  + 智能合约
  + 交易
  + 效用
  + 钱包
* [Changelog](http://cityofzion.io/neon-js/changelog.html)
  + [2.3.0](http://cityofzion.io/neon-js/changelog.html#id1)
  + [2.2.2](http://cityofzion.io/neon-js/changelog.html#id2)
  + [2.2.1](http://cityofzion.io/neon-js/changelog.html#id3)
  + [2.2.0](http://cityofzion.io/neon-js/changelog.html#id4)
  + [2.1.0](http://cityofzion.io/neon-js/changelog.html#id5)
  + [2.0.0](http://cityofzion.io/neon-js/changelog.html#id6)
  + [1.1.x](http://cityofzion.io/neon-js/changelog.html#x)
  + [1.0.x](http://cityofzion.io/neon-js/changelog.html#id9)

**Overview概述**

neon-js 是一个与NEO区块链接口的Javascript库，提供快速简单的方法发送RPC调用，创建交易和简单的合约调用。

**特征**

* 内置的RPC查询
* 交易创建，序列化和反序列化
* 钱包钥操纵
* 智能合约脚本生成器
* 第三方API支持

## 用法

Neon可以用两种方式使用：

语义

Neon的默认导入是一个Javascript对象，其功能按照动词-名词约定以语义方式排列。如果一个方法超过了2个级别，名字的其余部分是在名词级别用驼峰命名法。

**import** Neon from '@cityofzion/neon-js'

Neon.create.privateKey()

Neon.serialize.tx(transactionObj)

Neon.get.publicKeyFromPrivateKey(privateKey)

这种风格是建议初学者或任何只是希望使用Neon没有麻烦的人。

命名

命名的导入是传统的JS导入。Neon中的模块是：

* api
* CONST
* rpc
* sc
* tx
* u
* wallet

这种风格提供更多的控制和灵活性。请务必参阅每个模块导出的源代码。

**安装**

neon-js住在cityofzion组织下的NPM系统中。

**安装**

去安装:

$ npm install @cityofzion/neon-js

对于特定的提交（commit）或发布（release），请在后面添加提交哈希/标签/分支：

$ npm install @cityofzion/neon-js#dev //dev branch

**导入**

Neon-js支持两种导入。

默认导入将导入neon的语义版本。如果你是新的,或者只是想要使用整个包，请使用它。

**import** Neon from '@cityofzion/neon-js'

Neon.create.claimTx(...args)

**const** query **=** Neon.create.query()

模块通过命名导入（named imports）来暴露。这允许更细粒度的控制和访问单个模块。

**import** {rpc, tx} from '@cityofzion/neon-js'

Neon.tx.createClaimTx(...args)

**const** query **=** **new** rpc.Query()

**Require**

由于neon-js包使用ES6模块约定，require因此需要指定他们想要的模块：

**var** neon**-**js **=** require('@cityofzion/neon-js')

*// Semantic Style by using default import*

**var** Neon **=** neon**-**js.**default**

**const** query **=** Neon.create.query()

*// Named imports are available too*

**var** wallet **=** neon**-**js.wallet

**var** tx **=** neon**-**js.tx

**const** account **=** **new** wallet.Account(privateKey)

## Web

Neos-js也为Web打包。您可以通过脚本标签添加它：

<script src =“./ lib / browser.js”> </ script>

该库将作为全局变量Neon被提供。与require样式类似，您将具有default下的语义样式和在暴露自同一级别下其余的命名模块。

# 接口

neon-js仅仅是一个与区块链建立互动的库。它依靠外部数据源来获取构建交易所需的数据。但是，这些信息不能从官方NEO节点轻松获得。因此，为了填补这个空白，已经在努力建立第三方节点。

在这里，我们将描述来自外部数据源的信息。

余额描述了一个地址拥有的资产，以及列出未花费的币。未花费的币用于需要转移资产的大部分交易。neon-js目前需要这种格式：

{*// Balance*

address**:** string, *// The address*

assets**:** {

NEO**:** {

balance**:** number,

unspent**:** Coin[]

},

GAS**:** {

balance**:** number,

unspent**:** Coin[]

}

}

}

地址属性充当此对象的ID，用于计算更改以返回给帐户时用于派生scriptHash。

资产存储在一个对象中，并使用它们的符号进行检索。目前，只有2个可用资产（NEO / GAS），它们都有相似的符号名称。符号应使用大写字母，长度为3-4个字母。

每个符号将包含balance和unspent属性。balance告诉我们这个可用资产的总量，并作为一个简单的检查，如果有足够的可用资产可用来完成发送意图。unspent包含用作交易输入的Coin的列表。

## Coin

## 币

这个币本质上是一个可以花费的交易输出：

{*// Coin*

index**:** number,

txid**:** string,

value**:** number

}

这个币描述了一个可以在新交易中用作TransactionInput的未花费的TransactionOutput。

## Claim

该Claim对象专门用于构造ClaimTransaction：

{

claim**:** number, *// Amount claimable, multipled by 10^8*

end**:** number, *// Block which this coin was spent*

index**:** number, *// array index of transaction in outputs*

start**:** number, *// Block which this coin was created*

sysfee**:** number, *// System fees involved*

txid**:** string, *// Transaction ID of the coin*

value**:** number *// Amount of NEO held in this coin*

}

NEO在被持有时产生GAS。 当NEO被花费时，它产生的GAS通过ClaimTransaction被解锁，并被要求提取。这个Claim对象使得我们有可能计算出这个币所要求的GAS的数量。公式是：

claim **=** ((start **-** end) **\*** 8 **+** sysfee) **\*** value

**钱包**

钱包模块被暴露为：

**import** Neon from '@cityofzion/neon-js'

**const** account **=** Neon.create.account(privateKey)

Neon.is.address(string)

**import** {wallet} from '@cityofzion/neon-js'

**const** account **=** **new** wallet.Account(privateKey)

wallet.isAddress(string)

钱包模块包含操作密钥，创建签名和验证密钥的核心方法。

**import** neon from '@cityofzion/neon-js'

*// Key generation and conversion*

**const** privateKey **=** Neon.create.privateKey()

**const** wif **=** Neon.get.WIFFromPrivateKey(privateKey)

*// Encryption / Decryption*

**const** encrypted **=** Neon.encrypt.privateKey(privateKey, 'myPassword')

**const** decrypted **=** Neon.decrypt.privateKey(encrypted, 'myPassword')

*// Verify keys*

Neon.is.wif(wif) *// true*

Neon.is.publicKey('randomphrase') *// false*

*// Transaction signing*

**const** signature **=** Neon.create.signature(tx, privateKey)

**帐户**

Account类是从一个给定的密钥构造而来的，并提供了从给定的密钥中派生所有其他密钥格式的方法。请务必注意，您无法从较高级别派生较低级别的密钥。

钥匙排列如下：

1. 加密密钥（NEP2）
2. 私钥（HEX或WIF）
3. 公钥
4. 脚本散列
5. 地址

Account类只能由私钥，公钥或地址创建。ScriptHash和NEP2不被接受。

您不能从使用公钥创建的帐户派生私钥。（帐户将发生错误）

**import** Neon from '@cityofzion/neon-js'

**const** account **=** Neon.create.account(privateKey)

**let** publicKey **=** account.publicKey

**let** address **=** account.address

**const** account2 **=** Neon.create.account(publicKey)

account.privateKey *// Throws error*

**余额**

余额类存储帐户的余额。它通常使用第三方API检索，因为NEO节点没有RPC调用，无法只用一次调用就可轻松检索这些信息。

**import** Neon from '@cityofzion/neon-js'

Neon.create.balance({net**:** 'TestNet', address**:** 'ALq7AWrhAueN6mJNqk6FHJjnsEoPRytLdW'})

**import** {wallet, api} from '@cityofzion/neon-js'

*// This form is useless as it is an empty balance.*

**const** balance **=** **new** wallet.Balance({net**:** 'TestNet', address**:** 'ALq7AWrhAueN6mJNqk6FHJjnsEoPRytLdW'})

*// We get a useful balance that can be used to fill a transaction through api*

**const** filledBalance **=** api.getBalanceFrom('ALq7AWrhAueN6mJNqk6FHJjnsEoPRytLdW', api.neonDB)

*// This contains all symbols of assets available in this balance*

**const** symbols **=** filledBalance.assetSymbols

*// We retrieve the unspent coins from the assets object using the symbol*

**const** neoCoins **=** filledBalance.assets['NEO'].unspent

*// We can verify the information retrieved using verifyAssets*

filledBalance.verifyAssets('https://seed1.neo.org:20332')

余额类用于追踪可用于构建交易的未花费的币。verifyAssets是一个方便的方法，以确保由第三方API提供的未花费的币是通过比照一个NEO节点验证他们真正没有被花费掉。但是，这是一个昂贵的操作，所以谨慎使用。

## 核心

可用的核心方法是转换密钥格式并生成新的私钥的方法。

请务必注意，可用的方法不是全套，但只有最低要求。一般来说，有一种方法可以从较高的密钥中重新得到较低的密钥。例如，getPublicKeyFromPrivateKey存在但不是getAddressFromPrivatKey或getPrivateKeyFromPublicKey。对于所有格式的转换，鼓励您使用Account类。

**import** Neon from '@cityofzion/neon-js'

**const** privateKey **=** Neon.create.privateKey()

**const** publicKey **=** Neon.get.publicKeyFromPrivateKey(publicKey)

**const** scriptHash **=** Neon.get.scriptHashFromPublicKey(publicKey)

**const** address **=** Neon.get.addressFromScriptHash(scriptHash)

**import** { wallet } from '@cityofzion/neon-js'

**const** privateKey **=** wallet.generatePrivateKey()

**const** publicKey **=** wallet.getPublicKeyFromPrivateKey(privateKey)

**const** scriptHash **=** wallet.getScriptHashFromPublicKey(publicKey)

**const** address **=** wallet.getAddressFromScriptHash(scriptHash)

## NEP2

NEP2标准描述了加密或解密私钥的过程。加密方法接受WIF或HEX私钥。但是，解密方法将始终返回WIF以保持一致性。

请务必注意，加密/解密需要很长时间，在浏览器中可能无法很好地运行。

**import** Neon from '@cityofzion/neon-js'

**const** privateKey **=** Neon.create.privateKey()

**const** WIF **=** Neon.get.WIFFromPrivateKey(privateKey)

**const** nep2Key **=** Neon.encrypt(privateKey, 'myPassword')

**const** decryptedKey **=** Neon.decrypt(nep2Key, 'myPassword')

WIF **===** decryptedKey *// true*

**import** { wallet }

**const** privateKey **=** wallet.generatePrivateKey()

**const** WIF **=** **new** wallet.Account(privateKey).WIF

**const** nep2Key **=** wallet.encrypt(WIF, 'myPassword')

**const** decryptedKey **=** wallet.decrypt(nep2Key, 'myPassword')

WIF **===** decryptedKey *// true*

**验证**

各种密钥格式的验证方法都是可用的：

**import** Neon from '@cityofzion/neon-js'

Neon.is.address(addrStr)

Neon.is.privateKey(keyStr)

Neon.is.NEP2(encryptedStr)

**import** {wallet} from '@cityofzion/neon-js'

wallet.isAddress(addrStr)

wallet.isPrivateKey(keyStr)

wallet.isNEP2(keyStr)

这些方法将返回一个关于密钥格式的布尔值。没有错误将被抛出。

**钱包**

钱包类管理一系列帐户。它为用户提供了对多个账户的高级控制。例如，在单个交易中同时从多个地址发送资产。这是通过导出到文件来保留帐户数据的方法。

**import** Neon, {wallet} from '@cityofzion/neon-js'

**const** w1 **=** Neon.create.wallet()

*// This generates a new keypair and adds it to w1*

w1.addAccount()

*// Encrypt first account with keyphrase 'abc'*

w1.encrypt(0, 'abc')

*// Export file to ./w1.json*

w1.writeFile('./w1.json')

*// Initialise new wallet with custom parameters*

**const** w2 **=** **new** wallet.Wallet({name**:** 'wallet2', scrypt**:** {n**:**2,r**:**1,p**:**1}})

*// Read a wallet file and returns a Wallet object*

**const** w3 **=** wallet.Wallet.readFile('./w3.json')

[Next](http://cityofzion.io/neon-js/transactions/index.html)[Previous](http://cityofzion.io/neon-js/interface.html)

# 交易

The Transactions module is exposed as:

交易模块暴露为：

**import** Neon from '@cityofzion/neon-js'**let** transaction1 **=** Neon.create.claimTx(...args)transaction1 **=** Neon.sign.transaction(transaction1, privateKey)**let** serialized1 **=** Neon.serialize.tx(transaction1)**let** txid1 **=** Neon.get.transactionHash(transaction1)

**import** {tx} from '@cityofzion/neon-js'**let** transaction2 **=** tx.createClaimTx(...args)transaction2 **=** tx.signTransaction(transaction2, privateKey)**let** serialized2 **=** tx.serializeTransaction(transaction2)**let** txid2 **=** tx.getTransactionHash(transaction2)

交易形成了与区块链交互的核心。为了在链上实现任何状态变化，交易需要由共识节点发送并处理成块。

## 交易

Transaction类是一个包装类，它包含操作和构建交易所需的所有工具。这允许我们动态地添加意图，随意的评论，而不是将所有东西都塞进一个单一的方法中。

**import** Neon from '@cityofzion/neon-js'*// Let us create a ContractTransaction with a custom version***let** tx **=** Neon.create.tx({type**:** 128, version**:**2})*// Now let us add an intention to send 1 NEO to someone*tx.addOutput('NEO',1,someAddress).addRemark('I am sending 1 NEO to someAddress') *// Add an remark*.calculate(balance) *// Now we add in the balance we retrieve from an external API and calculate the required inputs.*.sign(privateKey) *// Sign with the private key of the balance*

**const** hash **=** tx.hash *// Store the hash so we can use it to query a block explorer.*

*// Now we can use this serializedTx string and send it through sendrawtransaction RPC call.***const** serializedTx **=** tx.serialize()

Transactions are composed of the following parts:

## 组件

交易由以下部分组成：

1. 类型

这决定了交易类型。这决定了交易如何被序列化或反序列化。目前该库仅支持以下几种类型：

1. 合约
2. 提取（Claim）
3. 调用
4. 版本

这决定了交易的版本。版本不同，协议可能延迟。

1. 属性

**TransactionAttribute()**

|  |  |
| --- | --- |
| **Arguments:** | * **usage** (number) – Identifying byte * **data** ([string](http://cityofzion.io/neon-js/reference/sc.html#string)) – Data |

附加到交易的附加属性。一个例子是一个备注。

1. 输入

**TransactionInput()**

|  |  |
| --- | --- |
| **Arguments:** | * **prevHash** ([string](http://cityofzion.io/neon-js/reference/sc.html#string)) – Transaction hash, Uint256 * **prevIndex** (number) – Index of the coin in the previous transaction, Uint16 |

交易的输入。这是这笔交易正在“花费”的资产。系统费用也包括在这里。交易处理后，输入被视为“花费”。

1. 输出

**TransactionOutput()**

|  |  |
| --- | --- |
| **Arguments:** | * **assetId** ([string](http://cityofzion.io/neon-js/reference/sc.html#string)) – assetId, Uint256 * **value** (number) – value of output, Fixed8 * **scriptHash** ([string](http://cityofzion.io/neon-js/reference/sc.html#string)) – Uint160 |

The outputs of the transaction. This indicates the unspent assets created by this transaction. These outputs are ‘unspent’ and can be referenced as inputs in future transactions.

交易的输出。这表示由此交易创建的未花费资产。这些产出是“未花费”的，可以在未来的交易中作为输入参考。

1. Witness

**Witness()**

|  |  |
| --- | --- |
| **Arguments:** | * **invocationScript** ([string](http://cityofzion.io/neon-js/reference/sc.html#string)) – This data is stored as is (Little Endian) * **verificationScript** ([string](http://cityofzion.io/neon-js/reference/sc.html#string)) – This data is stored as is (Little Endian) |

交易的见证人。这些是授权交易的签名。通常使用输入资产所有者的私钥来生成签名。请务必注意，虽然这个组件是被命名的**Witness**，但它在Transaction对象中的关键是**scripts**（我们试图保持C＃回购中描述的原始名称）。

独占数据（每种交易类型都是唯一的）

各项交易所需的各种数据。例如，ClaimTransaction将具有claims包含所有可提取的（claimable）交易的字段。InvocationTransaction将具有该script字段来代替智能合同调用。

## Types类型

**目录:**

* [合同交易](http://cityofzion.io/neon-js/transactions/contract.html)
* [提取交易](http://cityofzion.io/neon-js/transactions/claim.html)
* [调用交易](http://cityofzion.io/neon-js/transactions/invocation.html)

# 节点RPC

该rpc模块暴露为：

**import** Neon from '@cityofzion/neon-js'**const** client **=** Neon.create.rpcClient(URL)

**import** {rpc} from '@cityofzion/neon-js'**const** client **=** **new** rpc.rpcClient(URL)

## RPC客户端

RPC客户端充当特定NEO节点的模型。RPC调用是外部应用程序可以轻松地与NEO网络交互而不发送交易的方法。

它提供了内置的RPC方法以方便调用。以前的查询可以从history属性中检索。

RPC方法镜像官方NEO文档中的API参考。所有的RPC方法都返回一个Promise。

请务必注意，方法名称遵循camelCase的JS约定，而不是全部小写。

**import** Neon from '@cityofzion/neon-js'*// Creates a RPCClient with URL of version 2.3.2***const** client **=** Neon.create.rpcClient('http://seed1.neo.org:10332', '2.3.2')*// Returns block number*client.getBlockCount()client.getRawTransaction('f5412dba662ec8023e6fc93dba23e7b62679e0a7bebed52a0c3f70795cbb51d2', 1)

*// This will throw an error as invokefunction is not supported @ 2.3.2*client.invokeFunction(contractAddr,'name')

*// Custom query***let** query **=** Neon.create.query({method**:** 'custommethod'}client.execute(query)

## 查询

查询对象是一个围绕在请求/响应对周围的简单的包装。它使我们能够快速生成查询，而不依赖于客户端。

自定义查询可以通过传入必要的参数来创建。

还有支持生成受支持的RPC方法的静态方法。

**import** Neon from '@cityofzion/neon-js'

*// Custom query***const** query **=** Neon.create.query({method**:** 'newmethod', params**:** [arg1, arg2]})**const** response **=** query.execute('http://mycustomneonode.com:10332')

**import** { rpc } from '@cityofzion/neon-js'*// Simple query creation and execution***const** response **=** rpc.Query.getBlock(1).execute('http://seed1.neo.org:10332')

# API

该api模块显示为：

**import** Neon from '@cityofzion/neon-js'Neon.get.balance('TestNet', address)Neon.get.tokenBalance(contractScriptHash)

**import** {api} from '@cityofzion/neon-js'api.getBalance('TestNet', address)

该api模块包含所有与Neon一起使用的第三方API。主要亮点是提供必要的信息来构建ClaimTransaction或ContractTransaction的NeonDB API。一个正常的NEO节点不能为我们提供通过RPC检索余额或可提取（claimable）交易的简便方法。

但是请未必注意，这些API依赖于由第三方托管的节点，因此使用它们需要您自担风险。

## Core

这些核心方法有助于捆绑不同的第三方API，以简化事务创建和发送。

core方法暴露在顶层api。3个高级方法是：

1. claimGas
2. sendAsset
3. doInvoke

**import** Neon from '@cityofzion/neon-js'**const** config **=** {

net**:** 'TestNet'

address**:** 'ALq7AWrhAueN6mJNqk6FHJjnsEoPRytLdW',

privateKey**:** '7d128a6d096f0c14c3a25a2b0c41cf79661bfcb4a8cc95aaaea28bde4d732344'}Neon.api.claimGas(config).then((conf) => {

console.log(conf.response)})

**import** {api} from '@cityofzion/neon-js'api.claimGas(config).then((conf) => {

console.log(conf.response)})

这些方法主要依赖NeonDB作为Neoscan的信息作为回退。因此，他们通常更可靠。

这些方法围绕着传递一个包含沿着链的所有信息的配置对象。每种方法都会消化配置对象中的必要信息以执行其任务，并向其传递添加了新信息的配置对象。

**import** {api} from '@cityofzion/neon-js'*// This chain is basically api.claimGas*api.getClaimsFrom(config, api.neonDB).then((c) => api.createTx(c, 'claim')).then((c) => api.signTx(c)).then((c) => api.sendTx(c))

## NeonDB

NeonDB API暴露如下：

**import** Neon from '@cityofzion/neon-js'Neon.get.balance('TestNet', address)Neon.**do**.claimAllGas('TestNet', privateKey)

**import** {api} from '@cityofzion/neon-js'api.neonDB.getBalance('TestNet', address)api.neonDB.doClaimAllGas('TestNet', privateKey)

NeonDB API描述了由[neon-wallet-db](https://github.com/CityOfZion/neon-wallet-db)公开的API集合以及其它方便的方法。该节点由CityOfZion托管。

该API返回有用的信息，这些信息并非内置于标准的NEO节点，例如可索偿的交易或可花费的币。这些信息被用来构建交易。

例如，该getBalance方法返回某个具体地址的可花费资产的列表。然后这被用来构造一个ContractTransaction。

## Neoscan

NeoSan API作为一旦NeonDB失败时的备份。它没有暴露在语义导出中。而是使用命名的导出：

**import** {api} from '@cityofzion/neon-js'api.neoscan.getBalance('TestNet', address)api.neoscan.getClaims('MainNet', address)

这里找到的方法类似于NeonDB，但不包括所有的东西。方法将返回NeonDB的预期的类似的数据结构。

## CoinMarketCap

直接调用CoinMarketCap API来检索最新的价格信息。这是暴露cmc在内api。

**import** Neon from '@cityofzion/neon-js'Neon.get.price('NEO', 'EUR')Neon.get.price('GAS') *// defaults to USD*Neon.get.prices(['NEO', 'GAS'], 'EUR')Neon.get.prices(['NEO', 'GAS']) *// defaults to USD*

**import** { api } from '@cityofzion/neon-js'api.cmc.getPrice('NEO', 'SGD')api.cmc.getPrices(['NEO', 'GAS'], 'SGD')

## NEP5

NEP5标准描述了一组在智能合约中作为令牌实施的方法。这是以太坊中ERC-20标记标准的NEO等效物。

这组方法依赖于版本> = 2.3.3的NEO节点。该方法使用常量中DEFAULT\_RPC的默认节点。

**import** Neon from '@cityofzion/neon-js'**const** rpxScriptHash **=** Neon.CONST.CONTRACTS.TEST\_RPXNeon.get.tokenInfo('http://seed1.neo.org:20332', rpxScriptHash)Neon.get.tokenBalance('http://seed1.neo.org:20332', rpxScriptHash, address)

**import** { api } from '@cityofzion/neon-js'api.nep5.getTokenInfo('http://seed1.neo.org:20332', rpxScriptHash)api.nep5.getTokenBalance('http://seed1.neo.org:20332', rpxScriptHash)*// This is a combination of both info and balance within a single call*api.nep5.getToken('http://seed1.neo.org:20332', rpxScriptHash, address)

# 智能合约

该 sc 模块显示为

**import** Neon from '@cityofzion/neon-js'**const** sb **=** Neon.create.scriptBuilder()

**import** {sc} from '@cityofzion/neon-js'**const** sb **=** **new** sc.scriptBuilder()

在NEO中，用户通过InvocationTransactions与智能合约交互。这些交易将scriptBuilder的十六进制输出和涉及到的资产携带到网络中进行处理。

要测试智能合约，最好使用RPC调用：

* 调用
* invokefunction
* invokescript

这些在v2.3.3中实现。这些RPC调用执行提供的脚本，并根据当前的区块链状态返回结果。但是，它并没有被记录在链上。因此，他们的目的是测试脚本，以确保有效性，并找出所需的gas成本。

例如，在NEP5令牌标准中，我们不要求实际的交易来检索令牌的名称或符号。因此，最好使用invokeRPC调用而不是实际的invocationTransaction。

当我们想要改变状态时，我们将使用一个交易。例如，我们希望将地址A的令牌传递给B.我们将在发送实际交易之前使用invoke来确保脚本是有效的。

## 脚本生成器

脚本生成器是一个对象，它将智能合约方法调用转换为一个可以通过InvocationTransaction发送到网络的十六进制字符串。

**import** Neon, {rpc} from '@cityofzion/neon-js'**const** sb **=** Neon.create.scriptBuilder()*// Build script to call 'name' from contract at 5b7074e873973a6ed3708862f219a6fbf4d1c411*sb.emitAppCall('5b7074e873973a6ed3708862f219a6fbf4d1c411', 'name')

*// Test the script with invokescript*rpc.Query.invokeScript(sb.str).execute(nodeURL)

*// Create InvocationTransaction for real execution***const** tx **=** Neon.create.invocationTx(publicKey, {}, {}, sb.str, 0)

您可以将多个调用链接在一个VM脚本中。结果将按顺序返回。

**import** Neon, {rpc} from '@cityofzion/neon-js'**const** sb **=** Neon.create.scriptBuilder()sb.emitAppCall(scriptHash, 'name').emitAppCall(scriptHash, 'symbol')

*// Returns name, symbol*rpc.Query.invokeScript(sb.str)

.execute(Neon.CONST.DEFAULT\_RPC.MAIN)

.then((res) => {

console.log(res)

})

为了方便，提供一个简单的包装方法。

**import** Neon from '@cityofzion/neon-js'**const** props **=** {

scriptHash**:** Neon.CONST.CONTRACTS.TEST\_RPX,

operation**:** 'name',

args**:** []}*// Returns a hexstring***const** vmScript **=** Neon.create.script(props)

## ContractParam

ContractParam对象提供了一种构建invoke和invokefunction的简便方法。这些RPC调用使用一个JSON结构来进行参数传递，而且手工创建可能会混乱：

{

type**:** String,

value**:** 'this is a string'}

ContractParam目前支持创建字符串，布尔值，整数，bytearray和数组：

**import** Neon, {sc, rpc, CONST} from '@cityofzion/neon-js'**const** param1 **=** Neon.create.contractParam('String', 'balanceOf')*// This is a convenient way to convert an address to a reversed scriptHash that smart contracts use.***const** param2 **=** sc.ContractParam.byteArray('AVf4UGKevVrMR1j3UkPsuoYKSC4ocoAkKx', 'address')

rpc.Query.invoke(CONST.CONTRACTS.TEST\_RPX, param1, sc.ContractParam.array(param2))

.then((res) => {

console.log(res)

})

ContractParams与ScriptBuilder兼容，因此可以直接将它们作为参数传递。

# 效用

效用模块暴露为：

**import** Neon from '@cityofzion/neon-js'Neon.u.reverseHex(hexstring)

**import** { u } from '@cityofzion/neon-js'u.reverseHex(hexstring)

该效用程序模块包含：

* 格式操作方法
* 散列方法
* 效用类

## 格式

虽然Neon中的大多数方法都会接受字符串并输出字符串，但底层逻辑需要进行大量的格式转换。

**import** Neon from '@cityofzion/neon-js'Neon.u.reverseHex(hexstring)Neon.u.num2fixed8(1)Neon.u.ab2str(arrayBuffer)

最常见的格式是十六进制字符串。这是一个字符串，每2个字符代表一个字节数组中的一个字节。neon-js故意使用十六进制字符串，因为字符串很容易打印和操作。

NEO中使用的特殊格式是fixed8数字格式。这是一个精度为8位小数的定点浮点数。它通常是从一个hexstring接收的getrawtransaction。neon-js具有将其转换为和来自JS编号类型的功能。

## 散列法

这些方法是围绕在CryptoJS函数的方便包装。他们接受字符串并返回字符串。

**import** Neon from '@cityofzion/neon-js'*// Performs a single SHA*Neon.u.sha256(item)*// Performs a SHA followed by a SHA*Neon.u.hash256(item)*// Performs a SHA followed by a RIPEMD160*Neon.u.hash160(item)

## 效用类

StringStream是一个简单的流对象，允许我们逐字节读取一个十六进制字符串。这不是一个实际的流，但假装为流接口以实现更好的操作。它存储整个字符串和一个指针，以跟踪字符串上的当前位置。

它用于序列化和反序列化事务对象。用于智能合约的ScriptBuilder类从StringStream继承。

**import** Neon from '@cityofzion/neon-js'**const** ss **=** **new** Neon.u.StringStream('abcdefgh')ss.read(1) *// 'ab'*ss.read(2) *// 'cdef'*ss.isEmpty() *// false*ss.read(1) *// 'gh'*ss.isEmpty() *// true*ss.str *// 'abcdefgh'*

# 常量

常量被定义和暴露为：

**import** Neon from '@cityofzion/neon-js'**const** assets **=** Neon.CONST.ASSETS**const** defaultRPC **=** Neon.CONST.DEFAULT\_RPC

**import** { CONST } from '@cityofzion/neon-js'**const** rpcVersion **=** CONST.RPC\_VERSION

请务必参考所有可用常量的源代码。

# 参考

* [API](http://cityofzion.io/neon-js/reference/api.html)
* [RPC](http://cityofzion.io/neon-js/reference/rpc.html)
* 智能合约
* 交易
* 效用
* 钱包

# Changelog更改日志

这详细介绍了从以前的记录版本中所做的更改。

## 2.3.0

智能合约

添加OpCodes APPEND和REVERSE。

添加getPrices用一次API调用来获得多个令牌价格

更新解析api/getToken包含空字符串解析decimals字段的情况。

更新doMintToken以包含未来调用所需的额外信息。

* 这是为了准备即将发生的NEP5令牌变更。

实施API开关。

这个内部开关允许控制API服务器的优先级。

* 优先级设置为0，neonDB优先级设置为1。在中间设置会导致随机选择。
* 开关将动态选择哪个服务器响应更好。一个故障将开始倾斜到其他服务器的交换机。冻结交换机将防止这种动态行为。

目前这个问题还没有完全暴露出来，但将来还会有

**import** {api} from '@cityofzion/neon-js'api.setApiSwitch(0)api.sendAsset(config) *// sendAsset, claimGas and doInvoke will default to use neoscan first*api.setApiSwitch(1)api.doInvoke(config) *// This call will default to use neonDB first*

*// This freezes the switch, preventing it from changing dynamically.// You still can change it with setApiSwitch.*api.setSwitchFreeze(**true**)

修复

* 修复core.signTransaction检查输入是否是一个十六进制私钥。
* 修复NEP5令牌按decimals字段解析。
* 修复调用独占组件的默认值。

其它

* 将文档添加到自述文件。

## 2.2.2

* 修复nep5/doTransferToken

## 2.2.0

钱包

* 执行NEP-6标准 wallet.Wallet
* 将NEP2常数移到CONST
* 加密/解密有一个额外的可选scrypt参数
* 弃用wallet.encryptWifAccount，wallet.generateEncryptedWif，wallet.encryptWIF，wallet.decryptWIF

**import** Neon, {wallet} from '@cityofzion/neon-js'**const** w1 **=** Neon.create.wallet()**const** w2 **=** **new** wallet.Wallet()

账户

* 添加标签和额外的帐户
* 添加功能加密和解密到帐户

交易

* 交易创建现在将使用过的币从unspent移动到spent和在unconfirmed里添加新币。
* api.sendTx现在把币从unconfirmed移到unspent。
* 这意味着我们可以在一个块中创建2个交易，而不会相互阻塞。以前，这两笔交易将尝试使用相同的币。

API

* 添加api.getToken这个api.getTokenInfo和api.getTokenBalance的组合，允许单个调用内的简单信息检索。这是从语义上暴露为Neon.get.token。
* 漏洞修复CoinMarketCap将价格截断为整数。
* 漏洞修复 doTransferToken发送gas到错误地址，并在成功时将错误的项目追加到txid（改为附加完整的tx）。
* 使用没有余额的地址时，捕获getTokenBalance错误。

RPC

* 添加VMZip方法。这允许单独解析VM结果。请务必注意，这个方法产生一个解析函数。这不是直接使用。

**import** {rpc, u} from '@cityofzion/neon-js'**const** parsingFunc **=** rpc.VMZip(u.hexstring2ab, u.fixed82num)rpc.Query.invoke(script).parseWith(parsingFunc)

Utils

* 添加hexstring2str方法。

## 2.1.0

作为ES6类的余额

* verifyAssets根据给定的NEO节点验证未花费的币。用于检查余额是否完全同步和可用。
* applyTx申请交易支出用于余额。允许使用Balance来构建另一个Transaction，而不用等待同步。
* 数据结构重写。AssetBalances现在被藏起来了assets。使用assetSymbols来发现查找私钥。

*// This array contains all the symbols of the assets available in this Balance*balance.assetSymbols **=** ['NEO', 'GAS']*// Lookup assets using their symbols*balance.assets **=** {

NEO**:** {balance**:** 1, unspent**:** [ Object ], spent**:** [], unconfirmed**:** []}

GAS**:** {balance**:** 25.1, unspent**:** [ Object ], spent**:** [], unconfirmed**:** []}}

添加doTransferToken到api/nep5

 utils单元测试

打字稿类型修复

## 2.0.0

包导出语义风格

* 默认导出现在是遵循动词-名词惯例的语义对象。
* 可用的动词有：get，create，serialize，deserialize，is。阅读index.js每个模块的文件知道它们导出什么。
* 模块可单独作为命名导出来使用。如果您只需要帐户方法，import { wallet } from '@cityofzion/neon-js'

常量和util方法现在导出为

**import** Neon from '@cityofzion/neon-js'Neon.CONST.DEFAULT\_RPCNeon.u.reverseHex

**import** { CONST, u } from '@cityofzion/neon-js'CONST.DEFAULT\_RPCu.reverseHex

钱包

* 帐户现在可以作为类而不是JS对象。现在推荐帐户这种方法来管理密钥。
* 删除getAccountFromWIFKey和getAccountFromPrivateKey
* 密钥操作方法精简到最低限度。不再有getAddressFromPrivateKey。现在的方法只能把密钥转化一个等级。
* 密钥验证方法已经完全针对每种可能的密钥格式实施 按照isKeyFormat惯例。

交易

* 交易现在是一个ES6类而不是一个JS对象。交易现在是建立和操作交易的推荐方式。
* 重构方法来利用新的Transaction类。
* 从创建交易方法中删除.publicKey参数，因为地址足以生成scriptHash。
* 添加用于创建TransactionOutput的人性化的方法。
* 能够通过addRemark给交易添加评论

RPC

* RPCClient类建模一个NEO节点。使用Neon.create.rpcClient(url)举例说明。伴随来自RPC调用的内置方法。
* 查询类建模一个RPC调用。使用Neon.create.query()举例说吗。伴随来自RPC调用的内置方法。

API

* neon-wallet-db API被转移到api文件夹。
* 增加了coinmarketcap查询支持，方便查询价格。
* 令牌查询（NEP5）转移到这里。
* 添加了Neoscan支持。
* 硬件支持集成为外部签名函数提供的参数。
* 新的核心api方法：sendAsset，claimGas和doInvoke。

SC

* Sc中的generateDeployScript 是用于生成部署脚本的包装器。
* ContractParam添加到支持invoke和invokefunctionRPC调用

## 2.2.1

修正fixed82num不接受空字符串

## 1.1.x

### 1.1.1

* 总账支持
* 增加使用neonDB API的外部函数进行签名的功能。
* 针对\_emitNum的漏洞修复

### 1.1.0

交易大修

* 交易现在用动词-名词的惯例在语义上暴露。
* 交易创建被暴露为create.claim，create.contract和create.invocation

交易可以使用serializeTransaction和deserializeTransaction进行序列化或反序列化。

交易签名是现在signTransaction，它返回签名的交易，而不必手动附加签名。

* 交易哈希可以使用getTransactionHash在交易对象中的传送来计算。

用于智能合同调用的ScriptBuilder

* 类是一个用于构建VM脚本的对象，该脚本映射了C＃回购中找到的ScriptBuilder。
* buildScript是一个围绕在ScriptBuilder方便的的包装，和接受args的operation来调用合约。

getAccount方法重命名为getAccount并返回一个单一的Account对象而不是一个数组

getAccountsFromWIFKey -> getAccountFromWIFKey

getAccountsFromPrivateKey -> getAccountFromPrivateKey

## 1.0.x

### 1.0.4

* 附加的NEP2包装方法（简单的加密的WIF创建）
* 地址验证，以防止发送到非NEO地址。

### 1.0.2

引入NEP2支持（加密/解密WIF）

### 1.0.1

* 为neon-wallet-db将API支持升级到v2