Neon-js 来源于NEOWallet 网页版，由于这个版本体验不好，实际上已经雪藏，但是他的代码，早期叫做 antshares -tssdk，被neon-js继承了。

所以ts/js 开发者可以使用neon-js进行开发。我们将Neon-js的文档简单翻译如下：

**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

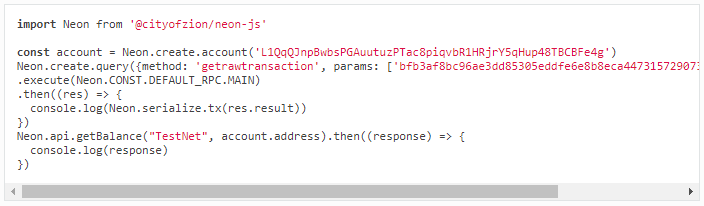
* 内置的RPC查询
* 交易创建，序列化和反序列化
* 钱包操纵
* 智能合约脚本生成器
* 第三方API支持

## 用法

Neon可以用两种方式使用：

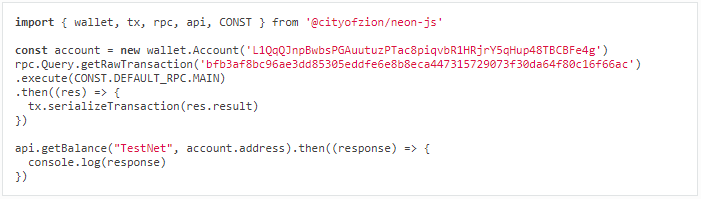
### 一、全部导入

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。



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。

### 二、精确导入



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

neon-js 项目被放在NPM中，在cityofzion组织下。

## 安装

安装方法:

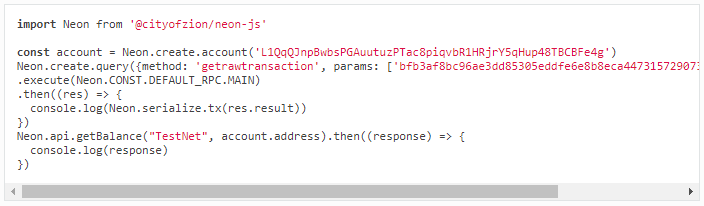
$ npm install @cityofzion/neon-js

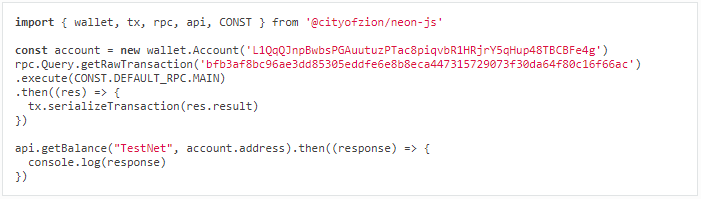
对于特定的提交（commit）或发布（release），请在后面添加提交哈希/标签/分支：

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

**导入neonjs 使用Import**

Neon-js支持两种Import方式。





**导入neonjs 使用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)

## 导入neonjs 使用Script标签

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（Balance）

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。

## UTXO（Coin）

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*

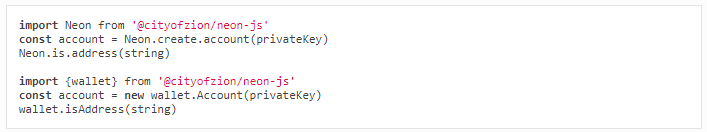
}

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

**钱包模块**

钱包模块被暴露为：

钱包模块包含操作密钥，创建签名和验证密钥的核心方法。



**帐户(Account)**

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：

加密私钥（NEP2）

私钥（HEX或WIF）

公钥

脚本散列

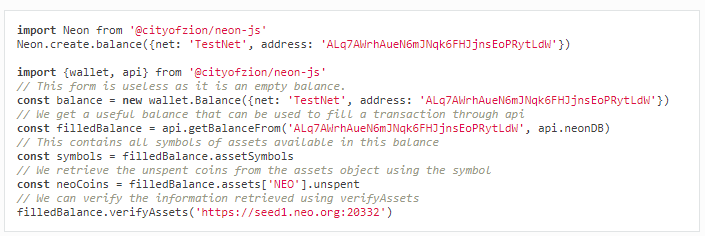
地址

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）

**余额(Balance)**

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。



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。

## Claims

The Claims class is a collection of claims data belonging to an account. It is usually retrieved from a 3rd part API. We do not recommend you craft your own Claims manually. This class is here for completeness in terms of high level objects.Claims类是属于一个账户的提取（claim)数据的集合。 它通常从第三方API检索。 我们不建议您手动制作自己的Claims。 这个类在这里是为了高层次的对象的完整性。

Like Balance, the constructor is the way to convert a Claims-like object into a neon-js Claims object. This is the primary method we use to convert the claims object we get from 3rd party API和Balance一样，构造函数是将类似Claim的对象转换为neon-js Claim对象的方法。 这是我们用来转换我们从第三方API获得的Claim对象的主要方法。

## 核心(Core)

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。

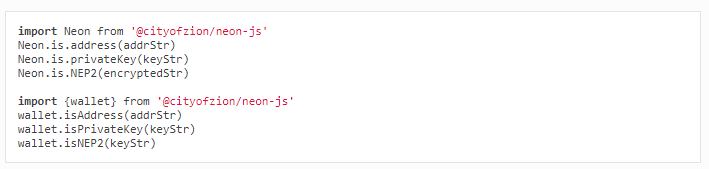
## NEP2（NEO2号提案）

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。

## 验证（Verify）

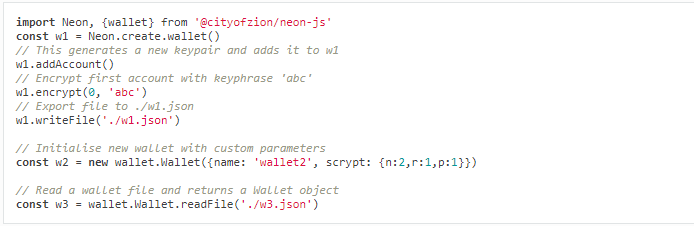
各种密钥格式的验证方法都是可用的：



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（wallet）**

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。



# 交易模块

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）

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，随意地评论，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:

## 交易组件

交易由以下部分组成：

1. 类型（type）

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（contract）
2. 提取（Claim）
3. 调用（Invocation）
4. 版本（version）

这决定了交易的版本。版本不同，协议可能延迟。

1. 属性（attribute）

**TransactionAttribute()**

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

附加到交易的附加属性。一个例子是一个备注。

4.输入（input）

**TransactionInput()**

|  |  |
| --- | --- |
| **Arguments:** | * **prevHash** ([string](http://cityofzion.io/neon-js/reference/sc.html" \l "string" \o "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”。

5 输出（output）

**TransactionOutput()**

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

6见证人（Witness）

**Witness()**

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

1. 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)

# RPC模块

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模块

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，以简化交易创建和发送。

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)})

这些方法是neon-js将要维护的核心功能。有内置的API选择，它将基于过去的交易在neonDB 和neoscan之间选择更可靠的API。

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claimable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。这是在api 内暴露为cmc。

**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)

# 智能合约模块

该 sc 模块暴露为

**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。结果将按顺序被返回。

**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。

# 工具模块（Utilities）

工具模块暴露为：

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

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

该工具（utility）模块包含：

* 格式操作方法
* 散列方法
* 工具（utility）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)

## 工具utility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'*

Fixed8是一个基于bignumber.js的类，用于存储和精确计算值。 它被扩展为具有用于在十进制和十六进制表示之间转换的辅助方法。

**import** Neon from '@cityofzion/neon-js'**const** a **=** **new** Neon.u.Fixed8(1)a.toHex() *// '0000000005f5e100'*a.toReverseHex() *// '00e1f50500000000'*

**const** b **=** Neon.u.Fixed8.fromHex('0000000005f5e100') *// 1*

**import** {u} from '@cityofzion/neon-js'**const** c **=** **new** u.Fixed8('2')**const** d **=** u.Fixed8.fromReverseHex('00e1f50500000000')

# 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。

# Log模块

The logging module is exposed only as a named import :

Logging模块仅作为一个命名导出被暴露为：

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

logging.logger.setAll('info') *// sets logging level of neon-js to 'info'*

**const** apiLogger **=** logging.logger.getLogger('api') *// gets the logger for the api package*

apiLogger.setLevel('warn') *// sets logging level only on the logger for the api package*

All logs are piped towards stdout and stderr. Each named package within neon-js will have its own logger. The initial setting for all loggers is ‘slient’.

所有的日志都是通过stdout和stderr传送的。 neon-js中的每个命名包都有自己的记录器。 所有记录器的初始设置是“slient”。

# 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
* 工具
* 钱包