

了解Vyper：灵活的New Ethereum语言

关于Vyper的全面指南，这是一种新的以太坊语言。

Vyper的是一个通用的，实验性的编程语言，编译成EVM（Ethereum Virtual Machine）编码，和Solidity一样。但是，Vyper旨在大规模简化流程，以便创建更易于理解的智能合约，这些合约对所有相关方更加透明，并且攻击的入口点更少。

任何针对EVM的代码都必须是超高效的，以最大限度地减少智能合约执行所需的gas，因为与低效代码的合约实际上需要更多的ether执行，并且可能很快变得非常昂贵，特别是在微处理器等用例中交易。最终的结果是Vyper看起来在逻辑上类似于Solidity，在语法上类似于Python，但没有很多面向对象编程范例 - 可能需要一个新的范式定义用于事务编程。

现在学习这些逻辑和语法上的差异将有助于您成为世界级的Vyper开发人员，因为截至2018年6月，Vyper仍处于v0.1.0-beta.1版本！



了解Vyper

0.对比Python， Vyper和Solidity

在这里，我们介绍了Vyper的高级“原因” - 为我们分析和编写Vyper代码提供了一个重点，包括[智能合约](#)。

关键改进1：简单

Vyper不包含大多数程序员熟悉的许多构造：类继承，函数重载，运算符重载和递归。对于图灵完备语言而言，这些都不是技术上必需的，并且它们通过增加复杂性来代表安全风险。由于这种复杂性，这些结构将使得[智能合约](#)难以理解并由一个非[专业人员](#)进行审计，如在[Solidity合约](#)中所见。

不太常见的构造和修饰符（这使得编写错误代码变得容易），联机汇编（这会破坏Ctrl + F）和二进制修复点（二进制修复通常需要近似值）。

关键改进2：安全

用Vyper开发者自己的话说，Vyper

“如果目标为了提高安全性，它会故意禁止一些事情或使事情变得更难。”

因此，Vyper并不是Solidity的全部替代品，而是一种在安全性至关重要的地方使用的优秀语言，例如用于处理患者健康元数据的智能合约或用于[去中心化的](#) AI渐变模型。

Vyper代码和语法差异

Vyper的构建尽可能与Python类似，同时努力实现安全性和简单性的目标，因此语言的整体感觉是相同的，尽管有许多不同的地方需要注意。

执行文件

虽然Python脚本执行为

```
python file_name.PY
```

，使用编译vyper脚本

vyper file_name.vy

状态变量

状态变量是永久存储在合同存储中的值，可以是任何类型，例如：

```
exampleStateVariable: int256
```

映射

Vyper合约包含的第一件事是合约存储字段，例如令牌余额映射：

```
balance: public(wei_value[address])
```

这是一个定义键和相应值的状态变量。

Vyper映射基本上是初始化的哈希表，如上所示，所以“每个可能的密钥都存在，并映射到一个字节表示全为零的值：一个类型的默认值。”

关键数据不存储在映射中，而只是存储在keccak256哈希中以查找其值。

在定义balance时，给出了类型**public**（），然后是映射语法：首先给出**wei_value**的值类型，然后是**key (address)** 在方括号中 - 类似于Python对数组的处理。

定义

您会注意到Vyper在定义名称（如balance）时使用冒号而不是Python的等号，尽管Python 3.6包含与变量注释相同的语法：

```
context = {} # empty dictionary context ["a"]: 2 #  
annotate dictionary variable
```

冒号语法用于变量注释，冒号用作赋值运算符，仅指定类型注释。Vyper使用此语法进行真值赋值。

整数类型

Vyper只有两种整数类型：**uint256**（对于非负整数）和**int128**（对于有符号整数） - 与Solidity的uint8到uint256的阶跃为8，而对于int8到int256则相同（这意味着int类型有64个不同的关键字） ）。

布尔运算符，运算符，比较和函数

对大多数运算符而言，Vyper与Python的语法几乎相同，包括：

```
true and false booleans; not, and, or, ==, and != operators;  
<, <=, ==, !=, >=, and > comparisons; and +, -, *, /, **, and  
% arithmetic operators (only for int128)
```

以及一些类似的内置函数：

`len(x)` to **return** the length of an **int**; `floor(x)` to **round** a decimal down to nearest **int**; **and** `ceil(x)` to **round** a decimal up to the nearest **int**

还有一些新的：

`sha3(x)` to **return** the sha3 **hash as bytes** **32**; `concat(x, ...)` to concatenate multiple inputs; `slice(x, start=_start, len=_len)` to **return slice** of `_len` starting at `_start`

列表

Vyper中的列表使用格式 `_name: _ValueType [_Integer]` 声明，而设置值和返回语句的语法与Python相同。

例如：

```
lst: int128 [ 3 ] #define a list
lst = [ 1,2,3 ] #set values
lst [2] = 5 # 通过索引设置一个值
return lst [0] # 返回1
```

结构

结构是您定义的类型，哪些群组变量，并使用 `struct.argname` 访问，因此（有点类似于Python词典）：

```
struct: { # define the struct
    arg1: int128, arg2: decimal
}
struct.arg1 = 1 #access arg1 in struct
```

定义方法

方法（Vyper中的合约方法）在Python和Vyper中以相同的方式定义：

```
def method():
```

```
do_something()
```

除了Python提供的功能之外，Vyper还包括ETH特定的修饰符，例如**@payable**和**@assert** - 前者用于使合约能够进行事务处理，而后者用于布尔表达式：

注意**def function_name (arg1, arg2, ..., argx) -> output**：用于定义函数的语法。与Python不同，Vyper在 ->之后明确定义def行中的输出类型。

构造函数

构造函数遵循与Python相同的约定，并在区块链上实例化给定的合约和参数。init初始化程序并且只执行一次。例如：

```
@public def __init__(_name: bytes32, _decimals: uint256, _initialSupply: uint256):
```

```
self.name = _name self.decimals = _decimals
self.totalSupply = uint256_mul(_initialSupply,
uint256_exp(convert(5, 'uint256'), _decimals))
```

与在Python中一样，self用于判断实例变量。上面的函数使用**@public装饰符**进行修饰，以使其具有公共可见性，并允许外部实体调用它（与默认值相反 - 或者省略装饰符 - 这是私有的）。

修饰符**@constant**用于修饰只读取状态的方法，而**@payable**使任何方法都可以通过payment来调用。

事件

您可以在索引结构中使用**__log__**记录事件，如下所示：

```
payment: __log__({amount: uint256, param2:
indexed(address)}) tot_payment: uint256 @public def
pay():

self.tot_payment += msg.value log.payment(msg.value,
msg.sender)
```


编写Vyper合约

现在，让我们写几个简单的智能合约。以下代码段允许合约接收NFT（不可互换的令牌）并能够针对该令牌发送。

```
@public def safeTransferFrom(_from: address, _to:
address, _tokenId: uint256):

self._validateTransferFrom(_from, _to, _tokenId,
msg.sender) self._doTransfer(_from, _to, _tokenId)
if(_to.codesize > 0):
returnValue: bytes[4] = raw_call(_to, '\xf0\x95\xba',
outsized=4, gas=msg.gas)
assert returnValue == '\xf0\x95\xba'
```

下面演示了@public修饰符，定义了一个具有明确给定类型的单个参数的函数，以及一个简单的代码体，使用判断语句来验证用户是否有权作为“委托投票”程序的一部分进行投票：

```
# Give a voter the right to vote on this ballot # This may
only be called by the chairperson @public def
give_right_to_vote(voter: address):
```

```
assert msg.sender == self.chairperson # throw if sender is
not chairperson assert not self.voters[voter].voted # throw
if voter has already voted assert self.voters[voter].weight
== 0 # throw if voter's voting weight isn't 0
```

```
self.voters[voter].weight = 1 self.voter_count += 1
```

在讨论了语法和逻辑区别之后，代码并没有太吓人。vyper.online提供“使用委托投票”程序的完整源代码，使用构造选民和提案，以及以下适当的命名函数：

```
def delegated(addr: address) -> bool def
directly_voted(addr: address) -> bool def
__init__(_proposalNames: bytes32[2]) def
give_right_to_vote(voter: address) def
forward_weight(delegate_with_weight_to_forward:
address) def delegate(to: address) def vote(proposal:
int128) def winning_proposal() -> int128 def
winner_name() -> bytes32
```

与任何编程语言一样，事先规划出主要结构（在本例中为合约函数）会使编程变得更加容易。要记住Vyper的主要区别是缺乏OOP范例。在当前的开发阶段，您还无法进行外部代码调用。

允许外部代码调用的注意事项可以在以下开发建议中看到：

外部合约A：

```
def foo(): constant def bar(): modifying # This contract  
contract B: a: A def baz(): a.foo() a.bar()
```

合约B调用合约A，包括A中的方法，最简单的例子。

运行Vyper

要继续编写代码，请转到vyper.online，并在“源代码”选项卡下编写代码示例，并在准备好后单击“编译”。

Vyper实现和测试执行最常用的客户端（虽然在pre-alpha中）是Py-EVM，最初由Vitalik自己开发，允许用户在不更改核心库的情况下添加操作码或修改现有操作码，从而实现更大的模块化和可扩展性相比传统的客户端。

要获得Py-EVM，只需使用`pip install py-evm == 0.2.0a16`。

3A。部署Vyper合同

虽然Py-EVM目前处于pre-alpha状态并且可能难以启动和运行，但有两种更简单的替代方法可以将Vyper合同部署到公共testnet（以及奖励）：

- 1) 将从vyper.online生成的字节码粘贴到Mist或geth中
- 2) 使用myetherwallet合约菜单在当前浏览器中部署3)（即将推出）

在未来，*Vyper*将与 *populus* 集成，允许您轻松部署 *Vyper*合同

为简单起见，我们将使用选项（1）和Mist（基于geth的新生的用户界面而不是基于终端的geth）部署合同。由于Vyper编译为与Solidity相同的字节码，因此我们不需要任何特定于Vyper的客户端，并且可以遵循这些稍微绕道的步骤：

1. 转到vyper.online并在预先填写的投票“源代码”上单击“编译”
2. 复制“字节码”选项卡下的所有内容
3. 安装Mist，如果你的系统还没有
4. 允许节点下载和同步（这会自动发生）
5. 在Mist设置中选择“使用测试网络”

6. 创建一个密码（记住它.....）
7. 输入合约
8. 在Mist界面中选择“Contracts”
9. 选择“部署新合约”
10. 转到“合约字节代码”选项卡
11. 粘贴您从vyper.online复制的字节码

部署合约

1. 选择“DEPLOY”并输入之前的密码
2. 确认已部署Vyper合约
3. 转到Mist中的“Wallets”选项卡
4. 向下滚动到“最新交易”
5. 你应该看到我们刚刚部署的合约！

* 虽然处于“创建合约”状态，因为它没有被挖掘和验证

结论

本指南提供了对Vyper的逻辑和语法介绍，允许我们开始编程和部署合约。根据本指南的知识，您应该能够为Vyper及其文档的开发做出贡献，并继续通过vyper.online编码来学习。

同样，Vyper并不是要取代Solidity，但是正如一项研究发现超过34,000份易受攻击的合约，在这个空间中对更强安全性的需求比以往任何时候都更大，这使得Vyper成为以太坊的重要未来。

进一步阅读和路线图

由于Vyper仍处于实验开发阶段，官方文档和GitHub是最全面的资源，以下提供参考地址：

01.[Vyper's Community Gitter](#)

02.[Vyper Tools and Resources](#)

03.["Ethereum Book" pages on Vyper](#)

04.[Study: "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale"](#)

05.["Step-by-Step Guide: Getting Started with Ethereum Mist Wallet"](#)

07.[Testing and Deploying Vyper Contracts](#)

08.["Build Your First Ethereum Smart Contract with Solidity — Tutorial"](#)

[Generalizing the steps to fit Vyper is fairly straightforward]

Vyper的1.0版开发步骤侧重于接口（以及内部和外部调用的改进等），它们定义了约束，因此您可以与实现该接口的任何对象进行通信。接口支持升级智能合约的替代解决方案，它们不是基本功能所必需的，您可以在Vyper中开始编码，尽管语言不完整。

版本1.0的开发路线图，从Vyper的gitter中检索和编辑：

- 01.两种类型的文件：接口（一个接口一个文件）和合约（一个合约一个文件）。
- 02.您可以在接口文件中定义类似ERC721Metadata的接口合约文件。
- 03.接口文件是与Ethereum ABI 一对一完全兼容的。
- 04.从Solidity到Vyper接口编写一个翻译器。
- 05.创建所有最终ERC接口的库，即使您必须手动制作他们。
- 06.Import接口文件到合约的接口。

07.接口是一种修饰地址的类型。

08.接口可以继承其他接口。

09.仔细研究ERC-165的接口ID，并重现上面给出的例子ERC-721这涉及接口如何继承其他接口。

10.接口可以具有可选功能。（一次突破从solidity.）

11.合约可以实现接口。

12.实现接口但未实现必需的合约功能是一个错误。

13.实现接口但未实现可选的合约函数既不是ERROR也不是警告。

14.将@public重命名为@external以匹配Solidity。

15.引入一个新的函数修饰符@internal，它允许调用一个内部函数。

16.重新引入当前用于外部调用的函数调用语法（删除步骤14）但它适用于内部调用。

17.像这样实现外部调用：**外部跳转调用表 ->**

LOADCALLDATA

unpack ->将函数参数添加到堆栈 ->调用内部函数 ->执行函数的东西。

18.像这样实现内部调用：向stack添加函数参数 -> **调用internal功能 ->做功能的东西。**

从我们编写的代码中可以看出，Vyper在开发过程中取得了很大的进步，并且在1.0发布之前只进行了一些重大更新(分解为上面的小步骤)!

来源：

<https://blockgeeks.com/guides/understanding-vyper/>

翻译：

爱上平顶山@慢雾团队