以太坊改进提案EIP1014（2018-04-20）新增了一个操作码CREATE2，可用于创建合约并控制合约地址的生成。引入CREATE2的目的是要“占住”该合约将来可能会部署的地址，因为原始的CREATE在计算合约地址时依赖于nonce，而nonce只能单向递增无法控制，所以很难通过CREATE来“占住”某个地址，即无法保证未来该合约一定能部署在某个地址上。

之所以要“占住”合约地址主要时为了进行“**反事实实例化**”，即在某种特定的业务需求下，我们可能需要创建一个还没部署上链，但满足有可能部署上链这一事实条条件的合约。具体定义可以在《**反事实的广义状态通道白皮书**》中查看。

提案内容

在**0xf5**处添加了一个新的操作码（**CREATE2**），该操作码接受4个栈堆参数，分别是endowment，memory_start，memory_length，salt。CREATE2能像CREATE（**0xf0**）一样创建新合约，但是CREATE是通过对合约创建者地址和合约创建交易的nonce进行哈希来生成合约地址，而CREATE2则是使用**keccak256(0xff ++ address ++ salt ++ keccak256(init_code))[12:]**来计算合约地址，因此用CREATE2创建合约时合约地址相对更可控。

在上面的计算公式中，**0xff**占1个字节，**address**为合约创建者地址，占20个字节，**salt**被定为32字节（刚好占用一个栈），因此使用CREATE2的最后一次哈希计算的长度为85字节。

CREATE2在使用时需要传入的参数有4个：

endowment：部署合约时需要给合约发送的ETH数量，单位是wei；

memory_start：待部署合约字节码在内存中的起始位置；

memory_length：待部署合约字节码的长度；

salt：用于做区别的进行哈希计算的盐分，可用来控制最终的地址生成。

使用CREATE2的时候需要确保计算出来的地址不与CREATE使用**keccak256(rlp([sender, nonce]))**计算出来的地址发生哈希碰撞。

合约地址的计算依赖于对init_code的哈希计算，所以如果重复对大段的init_code进行哈希计算则有可能造成对客户端的DDoS攻击。

CREATE2有和CREATE相同的gas消耗模式，但是CREATE2需要一个额外的哈希计算gas成本，为**GSHA3WORD * ceil(len(init_code) / 32)**。在计算最终的合约地址和执行init_code之前，这部分额外的哈希gas会与内存拓展gas和CreateGas一起被扣除。

本提案的平均每字gas成本与SHA3操作码一致。

说明

使用CREATE2可能会引发错误，即当CREATE2进行合约部署时，若合约地址已经被使用，则CREATE2会抛出异常。具体表现为合约地址的**nonce**或代码空间不为0，EIP684中有具体样例。

EIP161中提到，在执行**init_code**之前，账户创建交易和CREATE操作的nonce要增加1。这也就意味着如果在交易中创建合约，则合约的nonce立刻就会变为非0。

还需要注意的是，合约自毁操作码SELFDESTRUCT（0xff）在对合约进行自毁时不会立刻清空合约地址对应的nonce和code，因此不能在一个交易内进行同一合约地址的销毁和重新创建。

用例

官方用例

用例 1

- address `0x0000000000000000000000000000000000000000`
- salt `0x0000000000000000000000000000000000000000000000000000000000000000`
- init_code `0x00`
- gas (assuming no mem expansion): `32006`
- result: `0x4D1A2e2bB4F88F0250f26Ffff098B0b30B26BF38`

用例 2

- address `0xdeadbeef0000000000000000000000000000000`
- salt `0x0000000000000000000000000000000000000000000000000000000000000000`
- init_code `0x00`
- gas (assuming no mem expansion): `32006`
- result: `0xB928f69Bb1D91Cd65274e3c79d8986362984fDA3`

用例 3

- address `0xdeadbeef0000000000000000000000000000000`
- salt `0x000000000000000000000000feed000000000000000000000000000000000000`
- init_code `0x00`
- gas (assuming no mem expansion): `32006`
- result: `0xD04116cDd17beBE565EB2422F2497E06cC1C9833`

用例 4

- address `0x0000000000000000000000000000000000000000`
- salt `0x0000000000000000000000000000000000000000000000000000000000000000`
- init_code `0xdeadbeef`
- gas (assuming no mem expansion): `32006`
- result: `0x70f2b2914A2a4b783FaEFb75f459A580616Fcb5e`

用例 5

- address `0x000000000000000000000000000000000000deadbeef`
- salt `0x0000000000000000000000000000000000000000000000000000000000000000cafebabe`

- init_code `0xdeadbeef`
- gas (assuming no mem expansion): `32006`
- result: `0x60f3f640a8508fC6a86d45DF051962668E1e8AC7`

用例 6

- address `0x0000000000000000000000000000000000deadbeef`
- salt `0x0000000000000000000000000000000000000000000000000000000000cafebabe`
- init_code `0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef`
- gas (assuming no mem expansion): `32012`
- result: `0x1d8bfDC5D46DC4f61D6b6115972536eBE6A8854C`

用例 7

- address `0x0000000000000000000000000000000000000000`
- salt `0x0000000000000000000000000000000000000000000000000000000000000000`
- init_code `0x`
- gas (assuming no mem expansion): `32000`
- result: `0xE33C0C7F7df4809055C3ebA6c09CFe4BaF1BD9e0`

官方给的例子是基于原理层面的，下面我给一个基于合约层面实现的CREATE2使用例子。该例子改编自Uniswap V2-core中的部分代码。

```
pragma solidity 0.8.10;

contract Template {
    uint256 public a;
    string public b;
    address payable owner;

    constructor(uint256 _a, string memory _b) payable {
        a = _a;
        b = _b;
        owner = payable(msg.sender);
    }

    function getValue() public view returns(uint256){
        return address(this).balance;
    }

    //函数自毁后把ETH转移到owner，自毁后create2才能再次把地址部署到同一个地址
    function toSuicide() public{
        selfdestruct(owner);
    }
}

```

```
contract Create2Test {
    address public targetAddress;

    function create22(string calldata salt) external payable returns (address contractAddress) {
        //获取要发送给新合约的以太
        uint256 value = msg.value;
        //获取待部署的合约字节码
        bytes memory bytecode = type(Template).creationCode;
        //若待部署合约在部署时需要传入参数，则需要用encodePacked将字节码和传入参数进行编码
        bytes memory bytecodeAndParams = abi.encodePacked(bytecode, abi.encode(1,"lls"));
        //添加计算地址的盐，用于控制最终生成的地址，长度需要为32字节
        bytes32 theSalt = keccak256(abi.encodePacked(salt));
        //创建合约
        //value为部署合约时需要给合约发送的ETH数量(wei)，需要待部署合约的构造函数声明payable
        //并且部署合约的交易中msg.value的值要大于create2里value的值，不然会部署失败，若有多余ETH，则会被Create2Test合约本身接收
        //第二个参数是合约字节码在内存中的起始位置，因为byte数据由“长度+内容”组成，长度部分占32字节，
        //因此add对字节码地址偏移32个字节来获得字节码内容开始的地址
        //第三个参数为字节码长度，直接用mload获取字节码的前32个字节即可得到长度
        //第四个参数是盐，长度为32字节
        assembly {
            contractAddress := create2(value, add(bytecodeAndParams, 0x20), mload(bytecodeAndParams), theSalt)
        }
        targetAddress = contractAddress;
    }

    function getValue() public view returns(uint256){
        return address(this).balance;
    }
}
```

需要注意的点就是，如果需要在同一个地址下部署新的合约，则需要先将原合约销毁，否则会部署失败。

资料来源

EIP-1014: Skinny CREATE2 (ethereum.org)

教程 | 充分利用 CREATE2 (careerengine.us)

v2-core/UniswapV2Factory.sol at master · Uniswap/v2-core (github.com)