映射的现实应用:

身份证 ------- 具体的人

手机号 -------- 账号或昵称

定义方式为：mapping（键类型=>值类型），
例如 mapping(address=>uint) public balances，这个映射的名字是 balances，权限类型为 public，键的类型是地址 address，值的类型是整型 uint，在 solidity 中这个映射的作用一般是通过地址查询余额。键的类型允许除映射外的所有类型。

键: 除了映射，变长数组，合约，枚举，结构体以外的任意类型
值: 允许任意类型，甚至是映射

例如：映射 balances 中包括三个键值对（user1:100,user2:145,user3:195）,输入 user2即可得到 145

下面来看一个例子：

```
contract MappingExample{
    mapping(address => uint) public balances;

    function update(uint amount) returns (address addr){
        balances[msg.sender] = amount;
        return msg.sender;
    }
}
```

说明：定义 balances 为一个映射，msg.sender 是合约创建者的地址，函数 update 有一个整型参数 amount（数量），

balances[msg.sender]=amount 的意思是将参数 amount 的值和 msg.sender 这个地址对应起来。

同一个映射中，可以有多个相同的值，但是键必须具备唯一性

映射类型，仅能用来作为状态变量，或在内部函数中作为 storage 类型的引用
可以通过将映射标记为 public，来让 Solidity 创建一个 getter。通过提供一个键做为参数来访问它，将返回对应的值。

映射的值类型也可以是映射，使用 getter 访问时，要提供这个映射值所对应的键。

```
pragma solidity >=0.4.0 <0.8.0;

contract MappingExample {
    mapping(address => uint) public balances;

    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}

contract MappingLBC {
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(this);
    }
}

pragma solidity >=0.4.22 <0.8.0;

contract MappingExample {

    mapping (address => uint256) private _balances;
    mapping (address => mapping (address => uint256)) private _allowances;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    function allowance(address owner, address spender) public view returns (uint256) {
        return _allowances[owner][spender];
    }

    function transferFrom(address sender, address recipient, uint256 amount) public returns (bool) {
        _transfer(sender, recipient, amount);
        approve(sender, msg.sender, amount);
        return true;
    }

    function approve(address owner, address spender, uint256 amount) public returns (bool) {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
        return true;
    }
```

```
    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _balances[sender] -= amount;
        _balances[recipient] += amount;
        emit Transfer(sender, recipient, amount);
    }
}
```

可迭代映射

映射本身是无法遍历的，即无法枚举所有的键。不过，可以在它们之上实现一个数据结构来进行迭代。 例如，以下代码实现了 IterableMapping 库，然后 User 合约可以添加数据，sum 函数迭代求和所有值。

```
pragma solidity >=0.6.0 <0.8.0;

struct IndexValue { uint keyIndex; uint value; }
struct KeyFlag { uint key; bool deleted; }

struct itmap {
    mapping(uint => IndexValue) data;
    KeyFlag[] keys;
    uint size;
}

library IterableMapping {
    function insert(itmap storage self, uint key, uint value) internal returns (bool replaced) {
        uint keyIndex = self.data[key].keyIndex;
        self.data[key].value = value;
        if (keyIndex > 0)
            return true;
        else {
            keyIndex = self.keys.length;

            self.keys.push();
            self.data[key].keyIndex = keyIndex + 1;
            self.keys[keyIndex].key = key;
            self.size++;
            return false;
        }
    }

    function remove(itmap storage self, uint key) internal returns (bool success) {
```

```
        uint keyIndex = self.data[key].keyIndex;
        if (keyIndex == 0)
            return false;
        delete self.data[key];
        self.keys[keyIndex - 1].deleted = true;
        self.size --;
    }

    function contains(itmap storage self, uint key) internal view returns (bool) {
        return self.data[key].keyIndex > 0;
    }

    function iterate_start(itmap storage self) internal view returns (uint keyIndex) {
        return iterate_next(self, uint(-1));
    }

    function iterate_valid(itmap storage self, uint keyIndex) internal view returns (bool) {
        return keyIndex < self.keys.length;
    }

    function iterate_next(itmap storage self, uint keyIndex) internal view returns (uint r_keyIndex) {
        keyIndex++;
        while (keyIndex < self.keys.length && self.keys[keyIndex].deleted)
            keyIndex++;
        return keyIndex;
    }

    function iterate_get(itmap storage self, uint keyIndex) internal view returns (uint key, uint value)
{
        key = self.keys[keyIndex].key;
        value = self.data[key].value;
    }
}

// 如何使用
contract User {
    // Just a struct holding our data.
    itmap data;
    // Apply library functions to the data type.
    using IterableMapping for itmap;

    // Insert something
    function insert(uint k, uint v) public returns (uint size) {
        // This calls IterableMapping.insert(data, k, v)
        data.insert(k, v);
```

```
        // We can still access members of the struct,
        // but we should take care not to mess with them.
        return data.size;
    }

    // Computes the sum of all stored data.
    function sum() public view returns (uint s) {
        for (
            uint i = data.iterate_start();
            data.iterate_valid(i);
            i = data.iterate_next(i)
        ) {
            (, uint value) = data.iterate_get(i);
            s += value;
        }
    }
}
```