根据例子学习Solidity

本章将为你介绍几个例子，让我们理解 Solidity 如何编写智能合约。 本章有5个合约例子，由浅如深。

投票合约

以下的合约有一些复杂，但展示了很多Solidity的语言特性。它实现了一个投票合约。 当然，电子投票的主要问题是如何将投票权分配给正确的人员以及如何防止被操纵。 我们不会在这里解决所有的问题，但至少我们会展示如何进行委托投票，同时，计票又是 **自动和完全透明的** 。

我们的想法是为每个（投票）表决创建一份合约，为每个选项提供简称。 然后作为合约的创造者——即主席，将给予每个独立的地址以投票权。

地址后面的人可以选择自己投票，或者委托给他们信任的人来投票。

在投票时间结束时， `winningProposal()` 将返回获得最多投票的提案。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

/// @title 委托投票
contract Ballot {
    // 这里声明了一个新的复合类型用于稍后的变量
    // 它用来表示一个选民
    struct Voter {
        uint weight; // 计票的权重
        bool voted;  // 若为真，代表该人已投票
        address delegate; // 被委托人
        uint vote;   // 投票提案的索引
    }

    // 提案的类型
    struct Proposal {
        bytes32 name;   // 简称（最长32个字节）
        uint voteCount; // 得票数
    }

    address public chairperson;

    // 这声明了一个状态变量，为每个可能的地址存储一个 `Voter`。
    mapping(address => Voter) public voters;

    // 一个 `Proposal` 结构类型的动态数组
    Proposal[] public proposals;

    /// 为 `proposalNames` 中的每个提案，创建一个新的（投票）表决
    constructor(bytes32[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;
        //对于提供的每个提案名称，
        //创建一个新的 Proposal 对象并把它添加到数组的末尾。
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` 创建一个临时 Proposal 对象，
            // `proposals.push(...)` 将其添加到 `proposals` 的末尾
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // 授权 `voter` 对这个（投票）表决进行投票
    // 只有 `chairperson` 可以调用该函数。
    function giveRightToVote(address voter) public {
        // 若 `require` 的第一个参数的计算结果为 `false`，
        // 则终止执行，撤销所有对状态和以太币余额的改动。
        // 在旧版的 EVM 中这曾经会消耗所有 gas，但现在不会了。
        // 使用 require 来检查函数是否被正确地调用，是一个好习惯。
        // 你也可以在 require 的第二个参数中提供一个对错误情况的解释。
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }
```

```
    /// 把你的投票委托到投票者 `to`。
    function delegate(address to) public {
        // 传引用
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");

        require(to != msg.sender, "Self-delegation is disallowed.");

        // 委托是可以传递的，只要被委托者 `to` 也设置了委托。
        // 一般来说，这种循环委托是危险的。因为，如果传递的链条太长，
        // 则可能需消耗的gas要多于区块中剩余的（大于区块设置的gasLimit），
        // 这种情况下，委托不会被执行。
        // 而在另一些情况下，如果形成闭环，则会让合约完全卡住。
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // 不允许闭环委托
            require(to != msg.sender, "Found loop in delegation.");
        }

        // `sender` 是一个引用, 相当于对 `voters[msg.sender].voted` 进行修改
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];
        if (delegate_.voted) {
            // 若被委托者已经投过票了，直接增加得票数
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // 若被委托者还没投票，增加委托者的权重
            delegate_.weight += sender.weight;
        }
    }

    /// 把你的票(包括委托给你的票)，
    /// 投给提案 `proposals[proposal].name`.
    function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;

        // 如果 `proposal` 超过了数组的范围，则会自动抛出异常，并恢复所有的改动
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev 结合之前所有的投票，计算出最终胜出的提案
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // 调用 winningProposal() 函数以获取提案数组中获胜者的索引，并以此返回获胜者的名称
    function winnerName() public view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
```

```
        }
    }
```

可能的优化

当前，为了把投票权分配给所有参与者，需要执行很多交易。你有没有更好的主意?

秘密竞价（盲拍）合约

在本节中，我们将展示如何轻松地在以太坊上创建一个秘密竞价的合约。 我们将从公开拍卖开始，每个人都可以看到出价，然后将此合约扩展到盲拍合约， 在竞标期结束之前无法看到实际出价。

简单的公开拍卖

以下简单的拍卖合约的总体思路是每个人都可以在投标期内发送他们的出价。 出价已经包含了资金/以太币，来将投标人与他们的投标绑定。 如果最高出价提高了（被其他出价者的出价超过），之前出价最高的出价者可以拿回她的钱。 在投标期结束后，受益人需要手动调用合约来接收他的钱 - 合约不能自己激活接收。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.7.0;

contract SimpleAuction {
    // 拍卖的参数。
    address payable public beneficiary;
    // 时间是unix的绝对时间戳（自1970-01-01以来的秒数）
    // 或以秒为单位的时间段。
    uint public auctionEnd;

    // 拍卖的当前状态
    address public highestBidder;
    uint public highestBid;

    //可以取回的之前的出价
    mapping(address => uint) pendingReturns;

    // 拍卖结束后设为 true，将禁止所有的变更
    bool ended;

    // 变更触发的事件
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // 以下是所谓的 natspec 注释，可以通过三个斜杠来识别。
    // 当用户被要求确认交易时将显示。

    /// 以受益者地址 `_beneficiary` 的名义，
    /// 创建一个简单的拍卖，拍卖时间为 `_biddingTime` 秒。
    constructor(
        uint _biddingTime,
        address payable _beneficiary
    ) {
        beneficiary = _beneficiary;
        auctionEnd = block.timestamp + _biddingTime;
    }

    /// 对拍卖进行出价，具体的出价随交易一起发送。
    /// 如果没有在拍卖中胜出，则返还出价。
    function bid() public payable {
        // 参数不是必要的。因为所有的信息已经包含在了交易中。
        // 对于能接收以太币的函数，关键字 payable 是必须的。

        // 如果拍卖已结束，撤销函数的调用。
        require(
            block.timestamp <= auctionEnd,
            "Auction already ended."
        );

        // 如果出价不够高，返还你的钱
        require(
            msg.value > highestBid,
            "There already is a higher bid."
        );

        if (highestBid != 0) {
            // 返还出价时，简单地直接调用 highestBidder.send(highestBid) 函数，
            // 是有安全风险的，因为它有可能执行一个非信任合约。
            // 更为安全的做法是让接收方自己提取金钱。
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
```

```
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    /// 取回出价（当该出价已被超越）
    function withdraw() public returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // 这里很重要，首先要设零值。
            // 因为，作为接收调用的一部分，
            // 接收者可以在 `send` 返回之前，重新调用该函数。
            pendingReturns[msg.sender] = 0;

            if (!payable(msg.sender).send(amount)) {
                // 这里不需抛出异常，只需重置未付款
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// 结束拍卖，并把最高的出价发送给受益人
    function auctionEnd() public {
        // 对于可与其他合约交互的函数（意味着它会调用其他函数或发送以太币），
        // 一个好的指导方针是将其结构分为三个阶段：
        // 1. 检查条件
        // 2. 执行动作 (可能会改变条件)
        // 3. 与其他合约交互
        // 如果这些阶段相混合，其他的合约可能会回调当前合约并修改状态，
        // 或者导致某些效果（比如支付以太币）多次生效。
        // 如果合约内调用的函数包含了与外部合约的交互，
        // 则它也会被认为是与外部合约有交互的。

        // 1. 条件
        require(block.timestamp >= auctionEnd, "Auction not yet ended.");
        require(!ended, "auctionEnd has already been called.");

        // 2. 生效
        ended = true;
        emit AuctionEnded(highestBidder, highestBid);

        // 3. 交互
        beneficiary.transfer(highestBid);
    }
}
```

秘密竞拍（盲拍）

上面的公开拍卖接下来将被扩展为一个秘密竞拍。 秘密竞拍的好处是在投标结束前不会有时间压力。 在一个透明的计算平台上进行秘密竞拍听起来像是自相矛盾，但密码学可以实现它。

在 **投标期间** ，投标人实际上并没有发送她的出价，而只是发送一个哈希版本的出价。 由于目前几乎不可能找到两个（足够长的）值，其哈希值是相等的，因此投标人可通过该方式提交报价。 在投标结束后，投标人必须公开他们的出价：他们不加密的发送他们的出价，合约检查出价的哈希值是否与投标期间提供的相同。

另一个挑战是如何使拍卖同时做到 **绑定和秘密** : 唯一能阻止投标者在她赢得拍卖后不付款的方式是，让她将钱连同出价一起发出。 但由于资金转移在以太坊中不能被隐藏，因此任何人都可以看到转移的资金。

下面的合约通过接受任何大于最高出价的值来解决这个问题。 当然，因为这只能在披露阶段进行检查，有些出价可能是 **无效** 的， 并且，这是故意的(与高出价一起，它甚至提供了一个明确的标志来标识无效的出价): 投标人可以通过设置几个或高或低的无效出价来迷惑竞争对手。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }

    address payable public beneficiary;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;

    mapping(address => Bid[]) public bids;

    address public highestBidder;
    uint public highestBid;

    // 可以取回的之前的出价
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    /// 使用 modifier 可以更便捷的校验函数的入参。
    /// `onlyBefore` 会被用于后面的 `bid` 函数：
    /// 新的函数体是由 modifier 本身的函数体，并用原函数体替换 `_;` 语句来组成的。
    modifier onlyBefore(uint _time) { require(block.timestamp < _time); _; }
    modifier onlyAfter(uint _time) { require(block.timestamp > _time); _; }

    constructor(
        uint _biddingTime,
        uint _revealTime,
        address payable _beneficiary
    ) public {
        beneficiary = _beneficiary;
        biddingEnd = block.timestamp + _biddingTime;
        revealEnd = biddingEnd + _revealTime;
    }

    /// 可以通过 `_blindedBid` = keccak256(value, fake, secret)
    /// 设置一个秘密竞拍。
    /// 只有在出价披露阶段被正确披露，已发送的以太币才会被退还。
    /// 如果与出价一起发送的以太币至少为 “value” 且 “fake” 不为真，则出价有效。
    /// 将 “fake” 设置为 true ，然后发送满足订金金额但又不与出价相同的金额是隐藏实际出价的方法。
    /// 同一个地址可以放置多个出价。
    function bid(bytes32 _blindedBid)
        public
        payable
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: _blindedBid,
            deposit: msg.value
        }));
    }

    /// 披露你的秘密竞拍出价。
    /// 对于所有正确披露的无效出价以及除最高出价以外的所有出价，你都将获得退款。
    function reveal(
        uint[] _values,
        bool[] _fake,
        bytes32[] _secret
```

```
    )
        public
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        require(_values.length == length);
        require(_fake.length == length);
        require(_secret.length == length);

        uint refund;
        for (uint i = 0; i < length; i++) {
            Bid storage bid = bids[msg.sender][i];
            (uint value, bool fake, bytes32 secret) =
                    (_values[i], _fake[i], _secret[i]);
            if (bid.blindedBid != keccak256(value, fake, secret)) {
                // 出价未能正确披露
                // 不返还订金
                continue;
            }
            refund += bid.deposit;
            if (!fake && bid.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // 使发送者不可能再次认领同一笔订金
            bid.blindedBid = bytes32(0);
        }
        msg.sender.transfer(refund);
    }

    // 这是一个 "internal" 函数， 意味着它只能在本合约（或继承合约）内被调用
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != address(0)) {
            // 返还之前的最高出价
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }

    /// 取回出价（当该出价已被超越）
    function withdraw() public {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // 这里很重要，首先要设零值。
            // 因为，作为接收调用的一部分，
            // 接收者可以在 `transfer` 返回之前重新调用该函数。（可查看上面关于‘条件 -> 影响 -> 交互’的标注）
            pendingReturns[msg.sender] = 0;

            msg.sender.transfer(amount);
        }
    }

    /// 结束拍卖，并把最高的出价发送给受益人
    function auctionEnd()
        public
        onlyAfter(revealEnd)
```

```
    {
        require(!ended);
        emit AuctionEnded(highestBidder, highestBid);
        ended = true;
        beneficiary.transfer(highestBid);
    }
}
```

安全的远程购买合约

Purchasing goods remotely currently requires multiple parties that need to trust each other. The simplest configuration involves a seller and a buyer. The buyer would like to receive an item from the seller and the seller would like to get money (or an equivalent) in return. The problematic part is the shipment here: There is no way to determine for sure that the item arrived at the buyer.

There are multiple ways to solve this problem, but all fall short in one or the other way. In the following example, both parties have to put twice the value of the item into the contract as escrow. As soon as this happened, the money will stay locked inside the contract until the buyer confirms that they received the item. After that, the buyer is returned the value (half of their deposit) and the seller gets three times the value (their deposit plus the value). The idea behind this is that both parties have an incentive to resolve the situation or otherwise their money is locked forever.

This contract of course does not solve the problem, but gives an overview of how you can use state machine-like constructs inside a contract.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0  <0.9.0;

contract Purchase {
    uint public value;
    address payable public seller;
    address payable public buyer;

    enum State { Created, Locked, Release, Inactive }

    State public state;


    modifier condition(bool _condition) {
        require(_condition);
        _;
    }

    modifier onlyBuyer() {
        require(
            msg.sender == buyer,
            "Only buyer can call this."
        );
        _;
    }

    modifier onlySeller() {
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    modifier inState(State _state) {
        require(
            state == _state,
            "Invalid state."
        );
        _;
    }

    event Aborted();
    event PurchaseConfirmed();
    event ItemReceived();
    event SellerRefunded();

    //确保 `msg.value` 是一个偶数。
    //如果它是一个奇数，则它将被截断。
    //通过乘法检查它不是奇数。
    constructor() payable {
        seller = payable(msg.sender);
        value = msg.value / 2;
        require((2 * value) == msg.value, "Value has to be even.");
    }


    ///中止购买并回收以太币。
    ///只能在合约被锁定之前由卖家调用。
    function abort()
        public
        onlySeller
        inState(State.Created)
```

```
    {
        emit Aborted();
        state = State.Inactive;
        seller.transfer(address(this).balance);
    }

    /// 买家确认购买。
    /// 交易必须包含 `2 * value` 个以太币。
    /// 以太币会被锁定，直到 confirmReceived 被调用。
    function confirmPurchase()
        public
        inState(State.Created)
        condition(msg.value == (2 * value))
        payable
    {
        emit PurchaseConfirmed();
        buyer = payable(msg.sender);
        state = State.Locked;
    }

    /// 确认你（买家）已经收到商品。
    /// 这会释放被锁定的以太币。
    function confirmReceived()
        public
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        state = State.Release;

        buyer.transfer(value);
    }

    /// This function refunds the seller, i.e.
    /// pays back the locked funds of the seller.
    function refundSeller()
        public
        onlySeller
        inState(State.Release)
    {
        emit SellerRefunded();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        state = State.Inactive;

        seller.transfer(3 * value);
    }
}



}
```

微支付通道合约

译者注：本文其实和很多 线下扩容 的思路很类似，有兴趣可延伸阅读。

来看看如何在以太坊上实现一个支付通道。 通过使用密码签名技术可以在相同的参与者之间 **安全的、重复的、免手续费** 的转移以太币。学习这个示例子，我们需要先了解签名和验证签名以及如何建立支付通道。

创建及验证签名

想象一下 Alice 想发送一些以太币给 Bob, 即 Alice 发送者，而 Bob 是接收者。

Alice仅仅需要发送一条在链下密码学签名后的信息给Bob（比如通过消息），编写检查也是类似的。

Alice 和 Bob 用签名去授权交易，这可以通过以太坊智能合约来实现。Alice 将创建一个简单的智能合约来发送他的以太币，发送的函数不再是她在发起交易的时候执行，她将让 Bob 来执行并支付交易费。

合约工作有以下几步：

1. Alice 部署 `ReceiverPays` 合约, 并附上足够的以太来负担支付通道的付款。
2. Alice 通过自己的私钥签名来授权一个支付。
3. Alice 发送签名信息给Bob，这个信息是不需要保密的（稍后解释），用什么发送也无关紧要。
4. Bob 通过把签名信息提交给合约来索取这笔支付， 合约将验证信息的真实性并发送金额。

创建签名

Alice 不需要和以太坊网络进行交互就可以完成签名，这个过程是完全离线的。 在这个指引里, 我们将通过使用 web3.js and MetaMask 在浏览器里完成签名, 方法在 EIP-762 有描述。

```
/// 先计算一个hash
var hash = web3.utils.sha3("message to sign");
web3.eth.personal.sign(hash, web3.eth.defaultAccount, function () { console.log("Signed"); });
```

注解

`web3.eth.personal.sign` 会关注待签名信息的长度， 因为我们先计算了hash，这个信息将总是32字节，因此长度前缀也总是相同。

哪些内容需要签名

为了合约能实现支付功能，签名消息必须包括：

1. 收款人地址
2. 发送金额
3. 能够保护重放攻击的信息

所谓重放攻击是指一个被授权的支付消息被重复使用，为了避免重放攻击，我们引入一个 nonce (以太坊链上交易也是使用这个方式来防止重放攻击), 它表示一个账号已经发送交易的次数。智能合约将检查 nonce 是否使用过。

另外一种重放攻击可能发生的情形是这样的：所有者部署 `ReceiverPays` 合约之后，进行了一些支付，然后其销毁了合约， 随后又再次部署 `ReceiverPays` 合约， 这时新的合约无法知道先前部署合约的 nonce，所以攻击者可以再次利用先前的支付信息。 Alice 可以通过在签名信息中加入合约地址来阻止这个攻击。

下面的 `claimPayment()` 前两行，就是用来防止重放攻击。

打包参数

我们已经知道哪些信息需要包含到签名消息里，我们需要把这些信息合并在一起，计算 hash 然后 签名。很简单，先拼接数据，然后 ethereumjs-abi 库提供了 `soliditySHA3` that mimics the behaviour of 函数类似于 Solidity 的 `keccak256` 函数应用在 `abi.encodePacked` 的输出结果上，下面是JavaScript 为 `ReceiverPays` 实现签名的代码：

```
// recipient 表示向谁付款.
// amount, 单位 wei, 指定发送金额数量.
// nonce 保护重放攻击
// contractAddress 保护跨合约重放攻击
function signPayment(recipient, amount, nonce, contractAddress, callback) {
    var hash = "0x" + abi.soliditySHA3(
        ["address", "uint256", "uint256", "address"],
        [recipient, amount, nonce, contractAddress]
    ).toString("hex");

    web3.eth.personal.sign(hash, web3.eth.defaultAccount, callback);
}
```

在Solidity中还原消息签名者

通常, ECDSA（椭圆曲线数字签名算法） 包含两个参数, `r` and `s`. 在以太坊中签名包含第三个参数 `v`, 它可以用于验证哪一个账号的私钥签署了这个消息。 Solidity 提供了一个内建函数 ecrecover 它接受 `r`, `s` and `v` 作为参数并且返回签名这的地址。

提取签名参数

使用 web3.js 签名的数据， `r`, `s` 和 `v` 是连接在一起的，第一步是把各部分分离出来。 我们可以在客户端这这个操作，但是在合约上实现就仅仅需要一个参数而不是三个参数， 分离一个大的直接数组到各个部分工作量比较大，所以我们在 `splitSignature` 函数（在本节的结尾可以看到这个函数）里使用 内联汇编 来完成这个工作。

计算信息的Hash

合约需要知道哪些参数被签名了，以便它可以从参数中重建信息用来验证签名。在函数 `claimPayment` 中的 `prefixed` 和 `recoverSigner` 就是用来做这个事情。

ReceiverPays 完整合约代码

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract ReceiverPays {
    address owner = msg.sender;

    mapping(uint256 => bool) usedNonces;

    constructor() payable {}

    // 收款方认领付款
    function claimPayment(uint256 amount, uint256 nonce, bytes memory signature) public {
        require(!usedNonces[nonce]);
        usedNonces[nonce] = true;

        // 重建在客户端签名的信息
        bytes32 message = prefixed(keccak256(abi.encodePacked(msg.sender, amount, nonce, this)));

        require(recoverSigner(message, signature) == owner);

        payable(msg.sender).transfer(amount);
    }

    /// destroy the contract and reclaim the leftover funds.
    function kill() public {
        require(msg.sender == owner);
        selfdestruct(payable(msg.sender));
    }

    /// 第三方方法，分离签名信息的 v r s
    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
        require(sig.length == 65);

        assembly {
            // first 32 bytes, after the length prefix.
            r := mload(add(sig, 32))
            // second 32 bytes.
            s := mload(add(sig, 64))
            // final byte (first byte of the next 32 bytes).
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
        pure
        returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

        return ecrecover(message, v, r, s);
    }

    /// 加入一个前缀，因为在eth_sign签名的时候会加上。
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
```

```
    }
}
```

编写一个简单的支付通道

Alice 现在可以创建一个简单但完整支付通道，支付通道通过加密签名可以重复安全的转移以太币，并且无需付费。

什么是支付通道?

支付通道允许在无需发生交易的情况下多次转移以太。这意味着可以避免与交易相关的延迟和费用。 我们将探讨两方（Alice和Bob）之间的简单单向支付通道。 它涉及三个步骤：

1. Alice 附加一些以太创建智能合约，可以称为“打开”了支付通道
2. Alice会签署一些消息指明给接收者付款金额。 每次付款都会重复此步骤。
3. Bob“关闭”支付通道，取回以太币，并将剩余部分发送回发送者。

> **注解**
>
> 只有步骤1和3需要以太坊交易，步骤2意味着发送者通过离线方法（例如电子消息）将加密签名的消息发送给接收者。 这意味着只需要两个交易就可以支持任意数量（次数）的以太币转账。

Bob 保证会收到资金，因为智能合约托管以太并根据合法的签名消息来执行。 合约还可以强制超时执行，即使收款人拒绝关闭通道，Alice也能保证最终收回资金。 付款通道的参与者可以决定支付通道打开的持续时间。 对于短期交易，例如为网络访问的每一分钟支付一次网费，或者是长期的，例如向员工支付小时工资，支付可能持续数月或数年。

打开支付通道

要打开支付通道，Alice 需要部署智能合约，附加要托管的以太币并指定预期的收款人，以及通道存在有效时间。 合约的 `SimplePaymentChannel` 函数就是来做这个事情，代码在本节末尾。

进行支付

Alice 通过向 Bob 发送签名消息来付款。该步骤完全在以太坊网络之外执行。 消息由发送者以加密方式签名，然后直接传输给收款人。

每条消息都包含以下信息：

- 智能合约的地址，用于防止交叉合约重放攻击。
- 到目前为止所发送的以太总量。

在一系列转账结束时，付款通道仅需关闭一次。因此，只有一条消息被兑换。 这就是为什么每条消息都指定了以太的累计总量，而不是每次的微支付金额。 收款人自然而然的会选择兑换最新消息，因为这是以太总数最高的消息。 每条信息包含的nonce 将不再需要，因为智能合约仅执行一条信息。

包含合约地址用于防止一个支付通道的消息被用于不同的通道。

以下是修改后的JavaScript代码，用于对上一节中的消息进行加密签名：

```
function constructPaymentMessage(contractAddress, amount) {
    return abi.soliditySHA3(
        ["address", "uint256"],
        [contractAddress, amount]
    );
}

function signMessage(message, callback) {
    web3.eth.personal.sign(
        "0x" + message.toString("hex"),
        web3.eth.defaultAccount,
        callback
    );
}

// contractAddress is used to prevent cross-contract replay attacks.
// amount, in wei, specifies how much Ether should be sent.

function signPayment(contractAddress, amount, callback) {
    var message = constructPaymentMessage(contractAddress, amount);
    signMessage(message, callback);
}
```

关闭状态通道

当Bob准备好收到他们的资金时，就可以通过调用智能合约上的 `关闭` 功能来关闭支付通道。 关闭通道会向接收方支付所欠的以太币并销毁合约，剩余的以太币返回Alice。为了关闭通道，Bob需要提供 Alice 签名过的消息。

智能合约必须验证信息是否包含发送者的有效签名。执行此验证的过程与上面收款人使用的方法相同。Solidity函数 `isValidSignature` 和 `recoverSigner` 就是完成这个工作。

只有付款通道收款人可以调用 `close` 函数，其会选择最近的付款消息，因为该消息有最高的付款总额。如果允许发送者调用此函数，他们可以提供较低金额的消息，来欺骗收款人。

函数会验证签名的消息是否与给定的参数匹配，如果匹配，收款人将收到应得的部分，余下的部分通过 `selfdestruct` 返还给发送者。 可以在完整的合约代码中看到 `close` 函数。

通道有效期

Bob可以随时关闭支付通道，但如果他没有这样做，Alice 需要一种方法来收回他们托管的资金。 一个方法是在合约部署时设置 *到期时间* ，一旦达到那个时间，Alice 就可以调用 `claimTimeout` 收回他们的资金。 可以在完整的合约代码中查看 `claimTimeout` 函数。

调用此功能后，Bob无法再接收任何以太币，因此，Bob必须在到期前关闭频道。

完整合约代码

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract SimplePaymentChannel {
    address payable public sender;      // The account sending payments.
    address payable public recipient;   // The account receiving the payments.
    uint256 public expiration;  // Timeout in case the recipient never closes.

    constructor (address payable _recipient, uint256 duration)
        public
        payable
    {
        sender = payable(msg.sender);
        recipient = _recipient;
        expiration = block.timestamp + duration;
    }

    function isValidSignature(uint256 amount, bytes memory signature)
        internal
        view
        returns (bool)
    {
        bytes32 message = prefixed(keccak256(abi.encodePacked(this, amount)));

        // check that the signature is from the payment sender
        return recoverSigner(message, signature) == sender;
    }

    /// the recipient can close the channel at any time by presenting a
    /// signed amount from the sender. the recipient will be sent that amount,
    /// and the remainder will go back to the sender
    function close(uint256 amount, bytes memory signature) public {
        require(msg.sender == recipient);
        require(isValidSignature(amount, signature));

        recipient.transfer(amount);
        selfdestruct(sender);
    }

    /// the sender can extend the expiration at any time
    function extend(uint256 newExpiration) public {
        require(msg.sender == sender);
        require(newExpiration > expiration);

        expiration = newExpiration;
    }

    /// 如果过期过期时间已到，而收款人没有关闭通道，可执行此函数，销毁合约并返还余额
    function claimTimeout() public {
        require(block.timestamp >= expiration);
        selfdestruct(sender);
    }

    /// All functions below this are just taken from the chapter
    /// 'creating and verifying signatures' chapter.

    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
        require(sig.length == 65);
```

```
        assembly {
            // first 32 bytes, after the length prefix
            r := mload(add(sig, 32))
            // second 32 bytes
            s := mload(add(sig, 64))
            // final byte (first byte of the next 32 bytes)
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
        pure
        returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

        return ecrecover(message, v, r, s);
    }

    /// builds a prefixed hash to mimic the behavior of eth_sign.
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

注解

函数 `splitSignature` 没有做足够的安全检查，完整的产品里应该使用严格测试的库，如：openzepplin 的版本 。

验证支付

与上一节不同，付款通道中的消息不是马上赎回。 收款人会跟踪最新消息及在关闭付款通道时兑换它。 这意味着接收者对每条消息进行验证就至关重要。 否则，无法保证收款人能够最终获得付款。

收款人使用以下过程验证每条消息：

1. 验证信息中的合约地址是否与付款通道匹配。
2. 验证新金额是否为预期金额。
3. 确认新金额不超过托管的以太币总额。
4. 验证签名是否有效并来自通道的付款方。

我们使用 ethereumjs-util 库来编写验证过程，这里使用 JavaScript ， 当然实现的方式有很多。下面的代码借鉴了 上面的 *constructMessage* 函数:

```
// this mimics the prefixing behavior of the eth_sign JSON-RPC method.
function prefixed(hash) {
    return ethereumjs.ABI.soliditySHA3(
        ["string", "bytes32"],
        ["\x19Ethereum Signed Message:\n32", hash]
    );
}

function recoverSigner(message, signature) {
    var split = ethereumjs.Util.fromRpcSig(signature);
    var publicKey = ethereumjs.Util.ecrecover(message, split.v, split.r, split.s);
    var signer = ethereumjs.Util.pubToAddress(publicKey).toString("hex");
    return signer;
}

function isValidSignature(contractAddress, amount, signature, expectedSigner) {
    var message = prefixed(constructPaymentMessage(contractAddress, amount));
    var signer = recoverSigner(message, signature);
    return signer.toLowerCase() ==
        ethereumjs.Util.stripHexPrefix(expectedSigner).toLowerCase();
}
```

库合约使用

通过在合约中引入模块化方法（），可以帮助我们减少溢出等风险。 在下面的合约例子中，引入 `Balances` 库 合约使用了 `move` 方法， 去检查地址余额是否符合预期。

现在大家只需了解下库的作用，后面的文档有 更多关于库的使用。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0  <0.9.0;

library Balances {
    function move(mapping(address => uint256) storage balances, address from, address to, uint amount)
internal {
        require(balances[from] >= amount);
        require(balances[to] + amount >= balances[to]);
        balances[from] -= amount;
        balances[to] += amount;
    }
}

contract Token {
    mapping(address => uint256) balances;
    using Balances for *;   // 引入库
    mapping(address => mapping (address => uint256)) allowed;

    event Transfer(address from, address to, uint amount);
    event Approval(address owner, address spender, uint amount);

    function balanceOf(address tokenOwner) public view returns (uint balance) {
        return balances[tokenOwner];
    }
    function transfer(address to, uint amount) public returns (bool success) {
        balances.move(msg.sender, to, amount);
        emit Transfer(msg.sender, to, amount);
        return true;

    }

    function transferFrom(address from, address to, uint amount) public returns (bool success) {
        require(allowed[from][msg.sender] >= amount);
        allowed[from][msg.sender] -= amount;
        balances.move(from, to, amount);   // 使用了库方法
        emit Transfer(from, to, amount);
        return true;
    }

    function approve(address spender, uint tokens) public returns (bool success) {
        require(allowed[msg.sender][spender] == 0, "");
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }
}
```