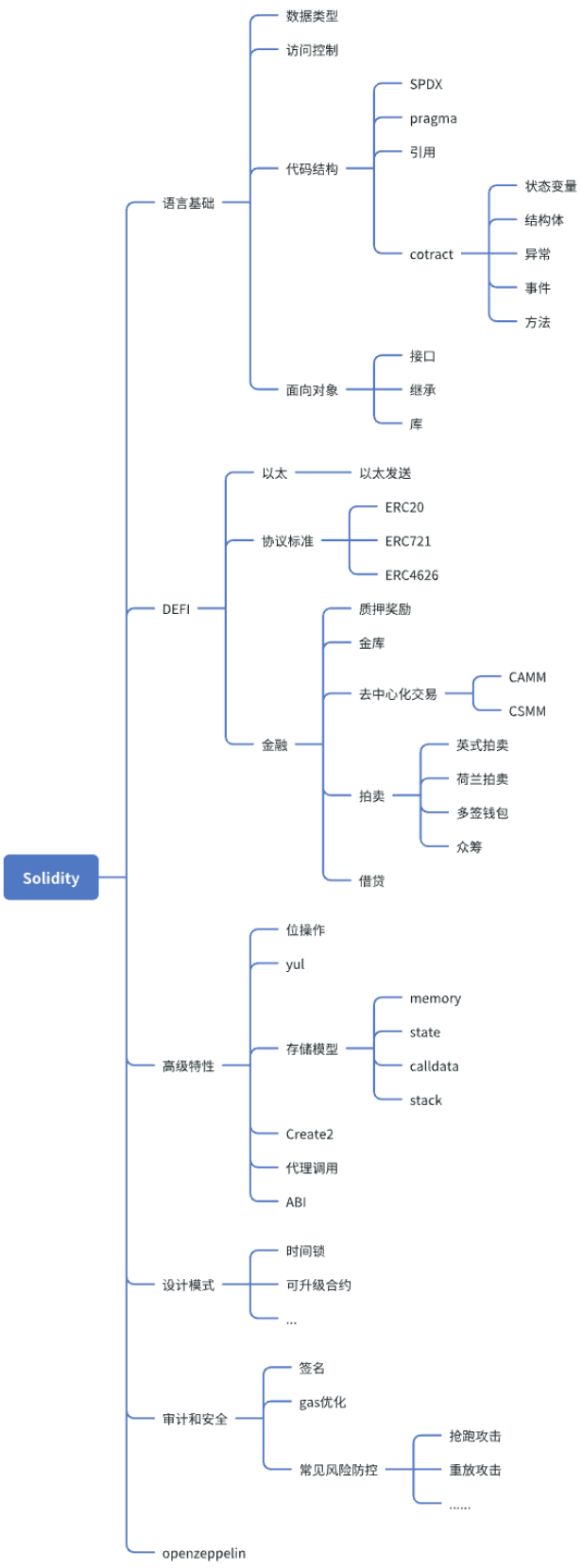
下半节

1、Solidity的学习内容


Solidity
语言基础
数据类型
访问控制
代码结构
SPDX
pragma
引用
cotract
状态变量
结构体
异常
事件
方法
面向对象
接口
继承
库
DEFI
以太
以太发送
协议标准
ERC20
ERC721
ERC4626
金融
质押奖励
金库
去中心化交易
CAMM
CSMM
拍卖
英式拍卖
荷兰拍卖
多签钱包
众筹
借贷
高级特性
位操作
yul
存储模型
memory
state
calldata
stack
Create2
代理调用
ABI
设计模式
时间锁
可升级合约
...
审计和安全
签名
gas优化
常见风险防控
抢跑攻击
重放攻击
.....
openzeppelin

2、讲解合约里的重点关键字

阅读NTF

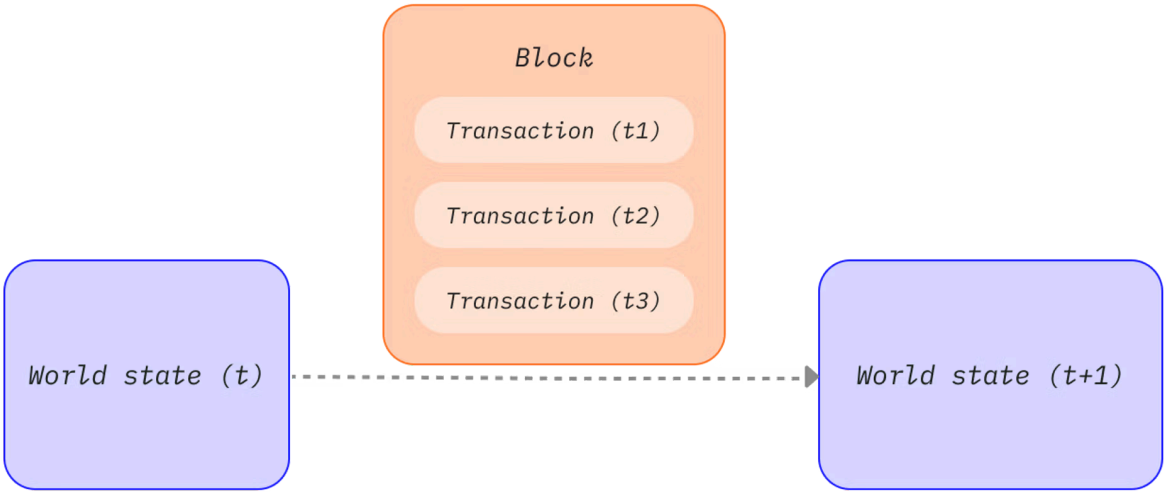
（1）require，revert，assert的区别

三者的主要区别

函数	使用场景	错误类型	Gas 处理
require	输入验证、权限检查等预期错误	可恢复错误（Error）	剩余 gas 退回
revert	复杂条件下的显式错误触发	可恢复错误（Error）	剩余 gas 退回
assert	检查代码逻辑中的严重错误	不可恢复错误（Panic）	所有剩余 gas 消耗

（2）局部变量、 状态变量、全局变量

- 状态变量：存储在区块链上，合约级别的变量，持久化存储。
- 局部变量：函数内部的临时变量，存储在内存或栈中，函数执行结束后销毁。
- 全局变量：Solidity 提供的特殊变量，用于访问区块链和交易信息。



（2）pure，view

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.27;
import "hardhat/console.sol";
contract PureView{
```

```
    uint256 _var;

    function showGlobalVar()  payable  external{

        console.log(msg.sender);
        console.log(block.timestamp);
        console.logUint(msg.value);


    }
    function notAccess(uint256 a,uint256 b) external pure returns(uint256){

        return a+b;
    }
        //这里不能是pure，因为msg.sender.balance就已经读取链上状态
    function notModifiy(uint256 a,uint256 b) external view returns(uint256){

        return a+b+msg.sender.balance;
    }

    function modifiyState() external  {
        _var++;
    }

}
```

🧪实验：测试脚本

通过修改notModifiy 删除view，可以看到不再是读取结果，而是一个transactionResponse

```
const { ethers } = require("hardhat");
async function show() {
  const _contract = await ethers.getContractFactory("PureView");
  const _pureView = await _contract.deploy();
  await _pureView.waitForDeployment();
```

```
  console.log(await _pureView.notModifiy(1, 2));
}

show();
```

👨‍🎓提问：不是public的数据就不能被外部访问么

```
const { ethers } = require("hardhat");
const hre = require("hardhat");
async function show() {
  const _contract = await ethers.getContractFactory("PureView");
  const _pureView = await _contract.deploy();
  await _pureView.waitForDeployment();
  console.log(await _pureView.notModifiy(1, 2));
  const storageValue = await ethers.provider.getStorage(
    await _pureView.getAddress(),
    0
  );

  console.log(`Storage value at slot`, storageValue);
}

show();
```

如果状态变量被定义为public，那么编译器会自动为public的变量生成get(),但是依然不能通过区块链直接修改他，要通过函数去修改。

（3）函数修饰符

public=external+internal>internal>private

```
contract SubPureView is PureView{
    //默认是internal
    function test() view external {
        console.log(_var);
```

```
    }
}
```

（4）多重继承

在Solidity中，继承是一种面向对象编程的概念，允许一个合约（子合约）从另一个合约（父合约）继承属性和方法。这种机制有以下几个特点：

- 代码复用：子合约可以重用父合约的代码，减少重复编写
- 功能扩展：子合约可以在父合约基础上添加新的功能
- 多重继承：Solidity支持多重继承，一个合约可以继承多个父合约
- 重写：子合约可以重写父合约的函数，实现自定义行为

在Solidity中使用 `is` 关键字来实现继承。例如：

```
contract ChildContract is ParentContract {
    // 子合约的代码
}
```

示例：构造函数按照is 后面的继承顺序

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.26;
import "hardhat/console.sol";

contract Parent{
    constructor(){
        console.log("Parent");
    }

    function foo()  virtual pure public {
        console.log("fool Parent");
    }
}
```

```
contract Base1 is Parent{

    constructor(){
      console.log("Base1");
    }

    function foo()  override virtual pure public {
      console.log("fool base1");
    }
}
contract Base2 is Parent{

    constructor(){
      console.log("Base2");
    }

    function foo() override  virtual  pure public {
      console.log("fool base2");
    }
}


contract Inherited is Base2, Base1{

   constructor(){

     console.log("Inherited");
   }


    function foo()  override(Base1,Base2)  pure  public {
      console.log("fool");
      super.foo();
    }
}
```

在上面这个例子中，Inherited合约构造函数调用顺序是 Base2,Base1，但是foo的调用顺序是Base1，Base2

> Another simplifying way to explain this is that when a function is called that is defined multiple times in different contracts, the given bases are searched from right to left (left to right in Python) in a depth-first manner, stopping at the first match. If a base contract has already been searched, it is skipped.

```
contract Parent{
     uint256  public i;
     constructor(){
            console.log("parent()");
     }
}
contract Base1 is Parent
{
   constructor(){
      console.log("Base1()");
   }
   function foo() virtual   public {
      i=i+2;
      console.log("foo1()");
   }
}

contract Base2 is Parent
{
   constructor(){
      console.log("Base2()");
   }
   function foo() virtual public {
       i=i*2;
```

```
        console.log("foo2()");
    }
}

contract Inherited is Base2, Base1
{
    // Derives from multiple bases defining foo(), so we must explicitly
    // override it
    //rsult 2 0+2
    function foo() override(Base2,Base1)  public {
        console.log("foo()");
        super.foo();
    }
}
```

演示super的调用顺序是 Inherited→Base1.foo()→Base2.foo()

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.26;
import "hardhat/console.sol";

contract Parent{
    constructor(){
        console.log("Parent");
    }

    function foo()  virtual pure public {
        console.log("fool Parent");
    }
}
contract Base1 is Parent{

    constructor(){
        console.log("Base1");
    }

    function foo()  override virtual pure public {
        console.log("fool base1");
```

```
    }
}
contract Base2 is Parent{

    constructor(){
      console.log("Base2");
    }

    function foo() override  virtual  pure public {
      console.log("fool base2");
    }
}


contract Inherited is Base2, Base1{

   constructor(){

     console.log("Inherited");
   }


    function foo()  override(Base1,Base2)  pure  public {
      console.log("fool");
      super.foo();
    }
}
```

（5）数据的存储位置

Solidity storage is like an array of length 2^256. Each slot in the array can store 32 bytes.

```
//SPDX-License-Identifier:MIT
pragma solidity ^0.8.24;
contract TestStorage{

   uint256 a=123;
```

```
    function retreiveSlotContent(uint256 index) external view returns(bytes3
2 content){
        assembly{
            content:=sload(index)
        }
    }
}
```