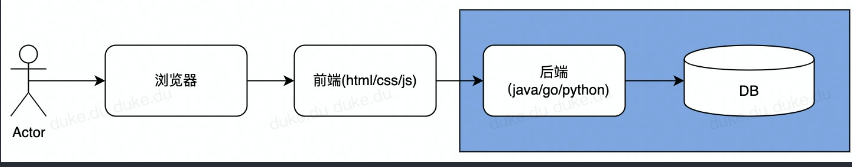
# APP与DAPP

APP架构

B/S或C/S的架构大体如此，从前端到后台，从机房到云原生，技术在迭代，效率在提高，一切越来越丝滑。底层的系统大致分为：从Unix、Linux、Dos、MacOs、Windows、安卓、IOS等等。

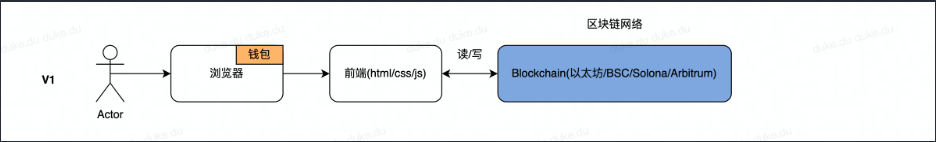


主流的dapp一般不称自己为app，而是称为：protocol（协议），如：aave、compound它们称自己为借贷协议，原因是：app是封闭的，而协议则是可以集成拓展的。

DAPP架构

而去中心化应用的架构却与App完全不同，因为没有专门的后台，它的区块链网络就是底层系统：如以太坊网络、BSC网络等。

一个最简单的Dapp应用可以直接描述如下，我们称为：V1版本架构

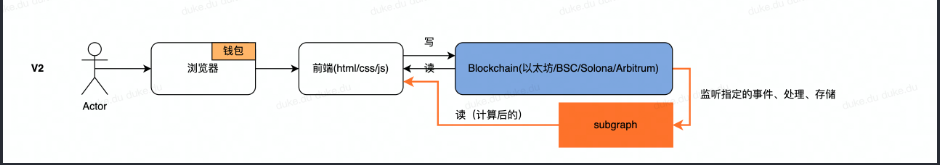


V1：我们可以看到，此时浏览器多了一个钱包模块，钱包是打开区块链的身份钥匙🔑，整个区块链上世界都是建立在密码学之上的（非对称加密，不懂也罢），而图中蓝色部分就是区块链网络，我们的程序就运行在其中，它既是后端服务，也是数据库。我们的前端直接与区块链网络进行交互，包括：

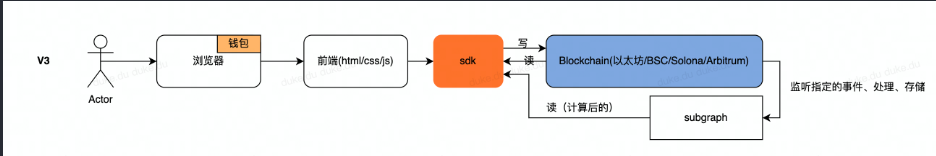
\*读数据（读取区块链账本数据），免费的

\*写数据（向区块链网络提交数据，永久保存，此时需要矿工打包执行），付费的（这点 记 住即可，后续语言学习中，陆续会讲解）

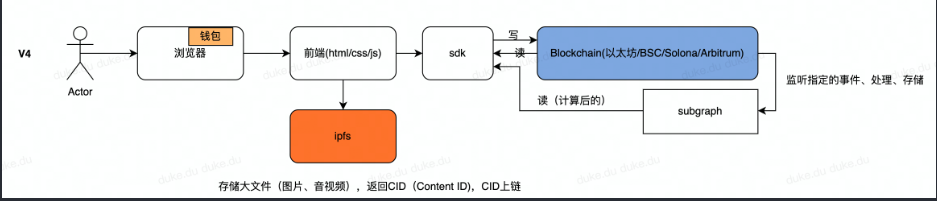
V2：随着区块链网络生态的发展，人们渐渐发现了优化空间，即我们可以将读操作预先存储到一个服务器中，然后前端直接从中获取数据，这既可以保证系统安全，又可以提高效率，因此subgraph（捕捉链上事件，链下存储，支持逻辑处理）这个基建得到飞速发展，图中橘黄色部分。我们称之为：V2版本架构



V3：当然，如果业务过于复杂，与链上合约交互的部分，我们可以独立封装出来，让前端专注于展示，合约专注于逻辑，所以引入了SDK，这部分专门处理和链上合约打交道的所有逻辑，并做相应的业务处理，时刻准备喂给前端，此时我们称为：V3版本架构



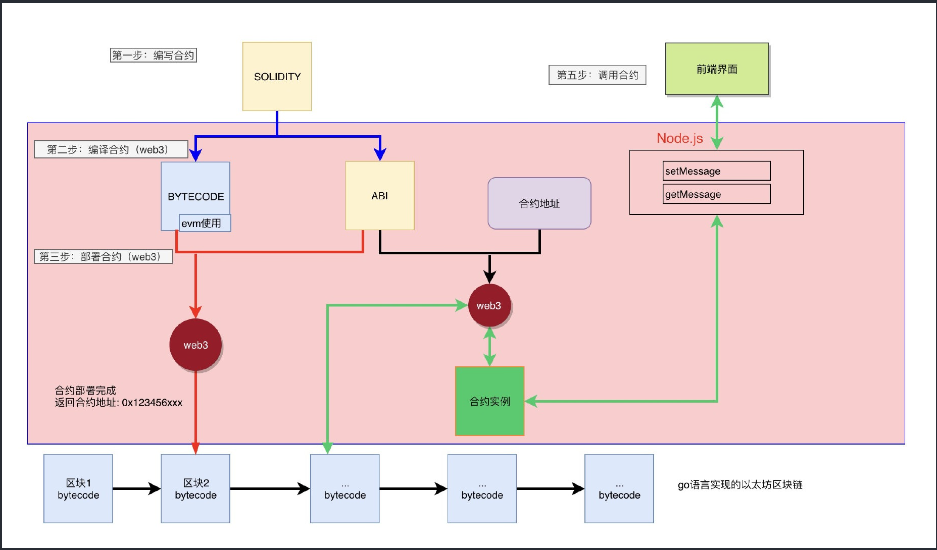
V4：再后来，区块链存储项目也陆续诞生，其中最为人知的便是：ipfs，它是一个致力于大文件存储的项目，最近很火的nft项目中，所有的图片都是要上传到ipfs上，从而节约以太坊网络费用，所以此时的设计更改如下，我们称之为：V4版本架构



# DAPP技术栈

web3的技术栈并不复杂，传统的互联网后端技术对于web3开发是非必须的，而是作为可选项，你需要掌握：

1. 区块链基本知识：地址、token、非对称加密、哈希、共识；
2. Solidity：编写智能合约，这是整个dapp最核心部分；
3. nodejs：这部分必须手到擒来，因为框架、单测、脚本等都是使用js来操作的（js是最主流的，python、java等也有）；
4. web3 sdk：主流的web3库为web3.js或ethers.js，所有与链上交互的操作都是通过这两个库进行的；
5. Subgraph: 这是一个三方基建服务，可以在链下对事件进行捕捉（扫链、计算、存储postgres），从而对进行数据整合，被前端调用，快速返回数据，极大的方便了开发者。
6. 前端：react（大项目使用，知名协议原生支持的库更全面）、vue
7. 后端（可选项）：机器人、周期性调用合约、监听扫链（钱包）、其他需求



开发流程

需求分析->开发->联调-> 测试网测试->主网测试->合约审计->上线->合约升级

设计原则

1. 相比于高效率更注重确定性（不可篡改）

（1）数据与逻辑分离，方便升级

2. 代码开源，数据完全公开（透明）

（1）拒绝链上存储任何敏感信息（包括修饰为private的字段）

（2）逻辑清晰，做好权限控制

（3）单元测试

3. 随时准备应对攻击（安全性）

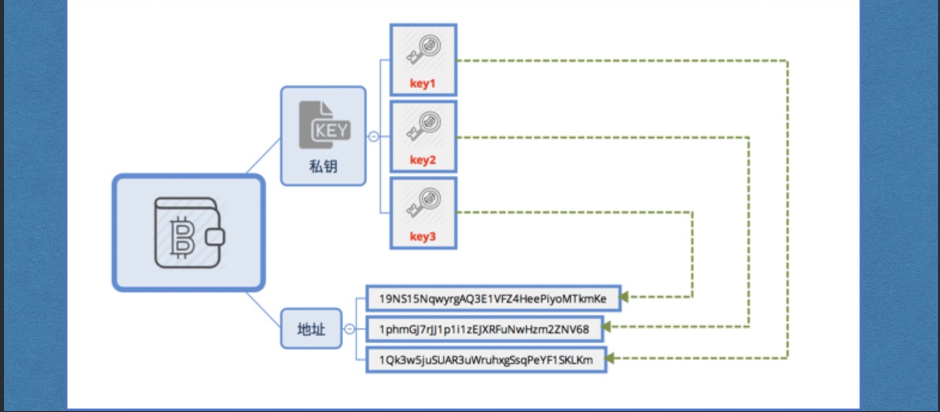
（1）总假设合约是被攻击成功的，有补救措施（紧急暂停、储备金）

（2）不重复造车轮子，使用经过检验的代码（类似于加密算法）

4. 交互付费（写数据时）

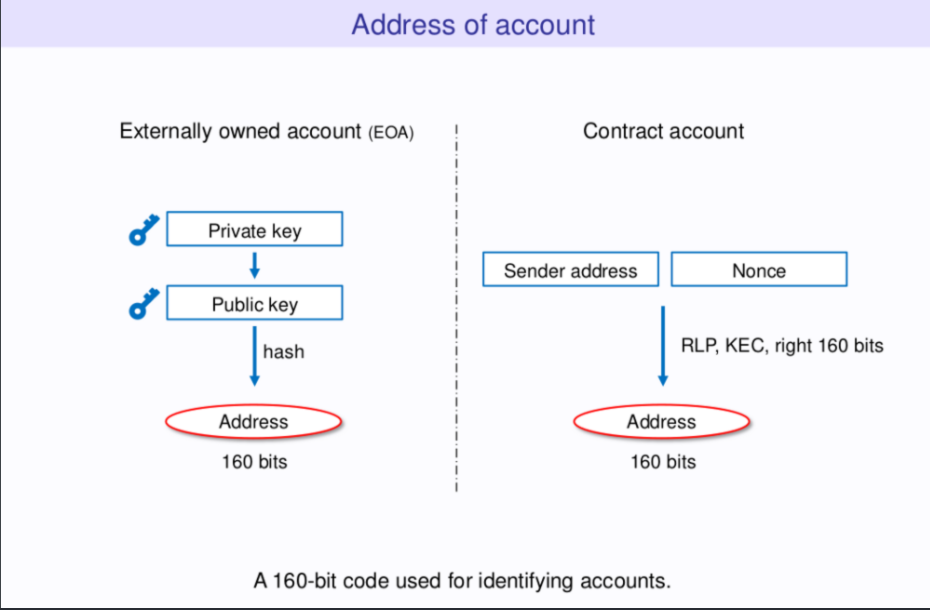
（1）非必要的数据不要上链（on-chain vs off-chain)

# 钱包与地址

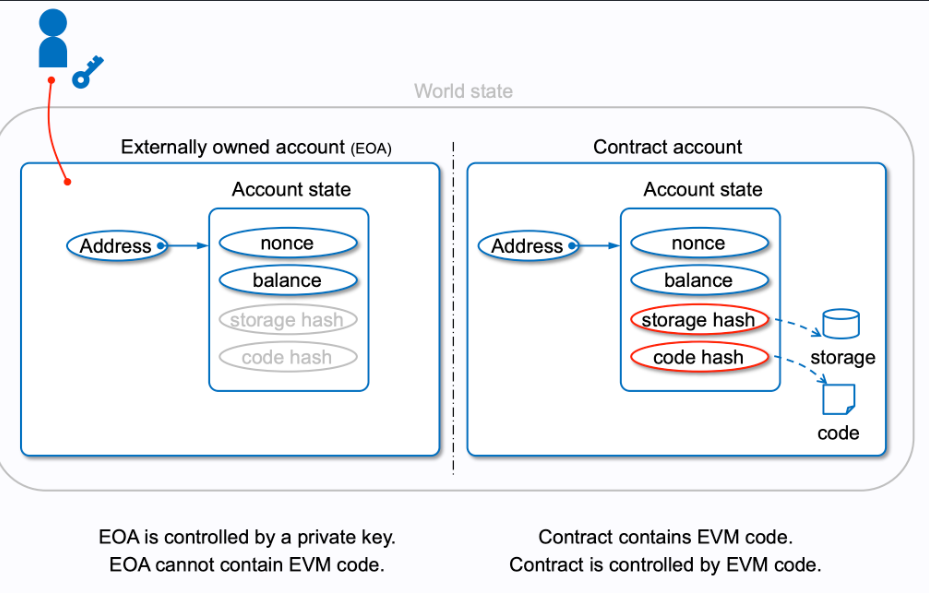


地址类型

* 小狐狸这种的地址，一般叫做EOA：External Of Account，外部账户，由私钥控制
* 部署合约后生成的地址，一般叫做CA：Contract Account，合约地址，生成规则取决于：部署地址和当前的Nonce

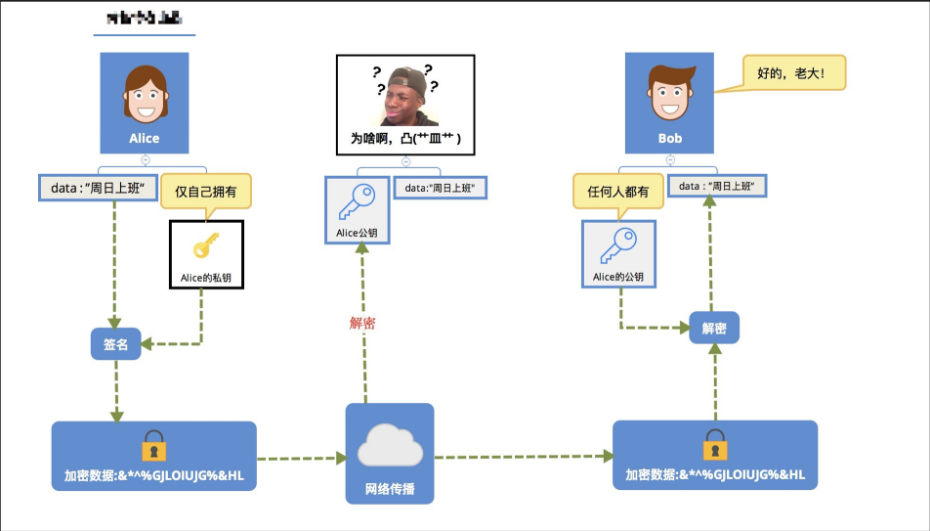


区别

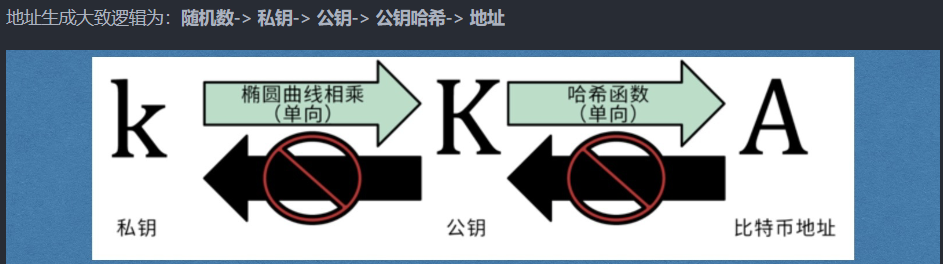


公私钥

* 私钥：是保密的，一般用于签名，证明你是资产的持有者；
* 公钥：是公开的，一般用于验证，证明私钥签名的数据是有效的



比特币地址生成规则



以太坊地址生成规则

1. 生成 256 位随机数作为私钥。
2. 将私钥转化为 secp256k1 非压缩格式的公钥，即 512 位的公钥。
3. 使用散列算法 Keccak256 计算公钥的哈希值，转化为十六进制字符串。
4. 取十六进制字符串的后 40 个字母，开头加上 0x 作为地址。

助记词

* 作用：帮助用户来更加方便的记住自己的私钥，管理财产（一套助记词可以派生N个钱包地址，小狐狸上可以点创建钱包，会自动生成新地址）。
* 原理：将随机数通过特定编码转化为词库中的单词。

# Solidity补缺

clone合约时，如果合约内有初始值，必须使用constant，否则clone的新合约初始值为空值。

Clone合约

普通工厂（我的理解就是使用用户的输入new一个类型的合约）

普通工厂模式的主要缺点是[气体成本高](https://link.juejin.cn/?target=https://ethereum.stackexchange.com/q/84764/33305" \o "https://ethereum.stackexchange.com/q/84764/33305" \t "https://juejin.cn/post/_blank)。而这正是克隆工厂模式的用武之地。

克隆工厂模式

为什么采用克隆工厂模式？

因为我们部署的是同一个合约，合约的每个实例都会有相同的字节码。所以，为每个部署反复存储所有字节码，会促进字节码的气体成本浪费。

解决这个问题的机制是只部署一个Foundation 合同的实例，并让所有其他Foundation 合同的实例作为代理，将调用委托给Foundation 合同的第一个实例，并允许函数在代理合同的上下文中运行。有了这个，Foundation 合同的每个实例将有自己的状态，并简单地使用由我们建立的Foundation 合同的实例作为一个库。

由[Peter Murray](https://link.juejin.cn?target=https://github.com/yarrumretep" \o "https://github.com/yarrumretep" \t "_blank),[Nate Welch](https://link.juejin.cn?target=https://github.com/nathanwelch" \o "https://github.com/nathanwelch" \t "_blank),[Joe Messerman](https://link.juejin.cn?target=https://github.com/JAMesserman" \o "https://github.com/JAMesserman" \t "_blank)创建的[eip-1167](https://link.juejin.cn?target=https://eips.ethereum.org/EIPS/eip-1167" \o "https://eips.ethereum.org/EIPS/eip-1167" \t "_blank)提供了这种机制。根据它的[文档](https://link.juejin.cn?target=https://eips.ethereum.org/EIPS/eip-1167" \o "https://eips.ethereum.org/EIPS/eip-1167" \t "_blank)，"这个标准指定了一个最小的字节码实现，它将所有的调用委托给一个已知的、固定的地址，以一种不可改变的方式简单而廉价地克隆合约功能。"

克隆[工厂合约](https://link.juejin.cn?target=https://github.com/optionality/clone-factory" \o "https://github.com/optionality/clone-factory" \t "_blank)是[eip-1167](https://link.juejin.cn?target=https://eips.ethereum.org/EIPS/eip-1167" \o "https://eips.ethereum.org/EIPS/eip-1167" \t "_blank)标准的一个参考实现。

变量

* 状态变量（state）

（1）定义在合约内，函数外

（2）存储在链上

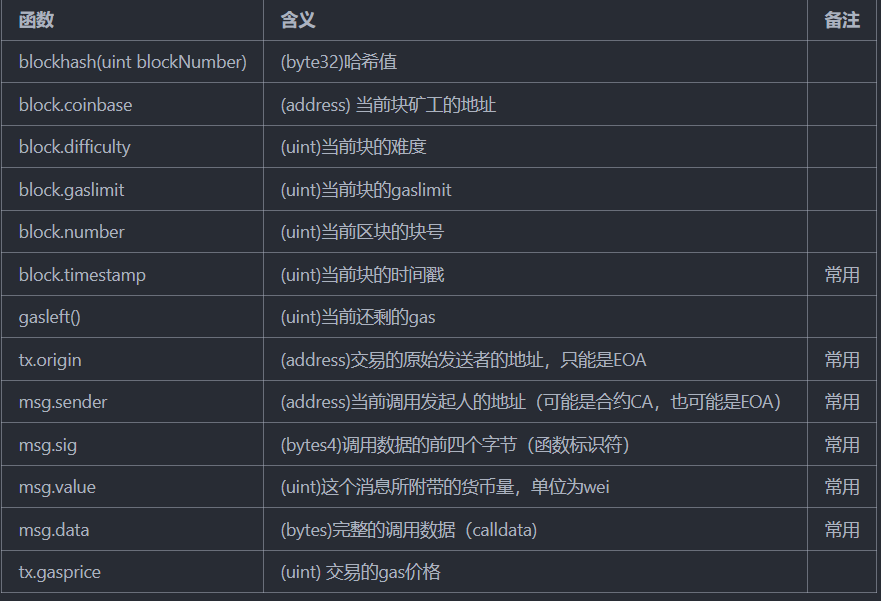
* 本地变量（local）

（1）定义在函数内

（2）不会存储在链上

* 全局变量（global）

1. 与当前合约无关，描述整个区块链的信息（时间、块高等）



不可变量

1. 与常量类似，但是不必硬编码，可以在构造函数时传值，部署后无法改变。
2. immutable仅支持值类型（如：int，address，bytes8），不支持非值类型（如：string，bytes）

bytes,string,byteN的关系



Bytes：

Bytes是动态数组，相当与byte数组

支持push方法添加

可以与string互相转换

* storage：属于状态变量，数据会存储在链上，仅适用于所有引用类型：string，bytes，数组，结构体，mapping等；
* memory：仅存储在内存中，供函数使用，数据不上链，适用于所有类型，包括：
* 值类型（int，bool，bytes8等）
* 引用类型（string，bytes，数组，结构体，mapping）
* calldata：存储函数的参数的位置，是只读的（只有calldata支持数组切片，状态变量不可以直接使用切片，需要new新数组，然后使用for循环解决）
* 其他：Solidity 变量中 memory 、calldata 2 个表示作用非常类似，都是函数内部临时变量，它们最大的区别就是 calldata 是不可修改的，在某些只读的情况比较省 Gas.
* 局部变量（此处指引用类型）默认是Storage类型的，只能将使用storage类型赋值，不能使用memory类型来赋值。

不能基于memory对象创建storage对象

# 错误Error

合约中发生错误时，整个交易状态都会进行回滚，一共有三个错误处理方式，具体如下：

（1）require：一般用于参数有效性校验，最常用。消耗的gas不会退回，剩余的gas退回；（条件为真，继续向下执行）

（2）revert：与require类似，适用于校验条件复杂时使用；

（3）assert：用于断言绝对不改出错的地方，注意：（条件为真，继续向下执行;一般用于校验内部错误）

i: 一般用于程序异常处理，触发了assert意味着存在bug；

Ii: 不提供错误信息；

0.8.0之前，Asset会消耗掉所有提供的gaslimit，剩余的gas也不会返回。 [（0.8.0之后已经不会再消耗了）](https://goerli.etherscan.io/tx/0x4f23d669924b8a7462427399c81f6e125d91065f08658804fdbadb587689987a" \o "https://goerli.etherscan.io/tx/0x4f23d669924b8a7462427399c81f6e125d91065f08658804fdbadb587689987a)

1. 也可以自定义error，可以节约gas
2. [错误消息的长度会影响：](https://medium.com/@chebyk.in/how-big-is-solidity-custom-error-messages-overhead-1e915724b450" \o "https://medium.com/@chebyk.in/how-big-is-solidity-custom-error-messages-overhead-1e915724b450)

（1）gas消耗数量

（2）单个合约的大小

# 事件

事件是区块链上的日志，每当用户发起操作的时候，可以发送相应的事件，常用于：

（1）监听用户对合约的调用

（2）便宜的存储（用合约存储更加昂贵）

通过链下程序（如：subgraph）对合约进行事件监听，可以对Event进行搜集整理，从而做好数据统计，常用方式：

（1）合约触发后发送事件

（2）subgraph对合约事件进行监听，计算（如：统计用户数量）

（3）前端程序直接访问subgraph的服务，获得统计数据（这避免了在合约层面统计数据的费用，并且获取速度更快）

一个事件内可以最多将三个字段修饰为indexed，当使用indexed关键字时，更加方便索引，并且：

（1）如果修饰的是值类型的，则直接展示；

（2）如果是非值类型，如：array，string等，则使用keccak256哈希值。

（3）indexed：方便索引，加了inexed是topics

（4）non-indexed：没有解码，需要使用abi解码后才知道内容，不加indexd是data

其他：

Log也在区块链账本中，和合约存储在不同的结构中；

Log是由交易执行产生的数据，它是不需要共识的，可以通过重新执行交易生成；

Log是经由链上校验的，无法造假，因为一笔交易的ReceiptHash是存在链上的（Header中）

# 修饰器modifier

修饰器用于修饰函数，在函数执行前或执行后进行调用，经常用于：

（1）权限控制；

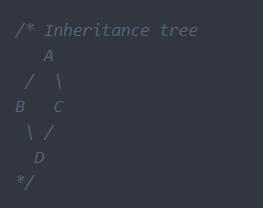
（2）参数校验；

（3）防止重入攻击等；

重写（见继承的方法\_重写）

1. 继承的顺序很重要，遵循最远继承，即后面继承的合约会覆盖前面父合约的方法
2. super会调用继承链条上的每一个合约的相关函数，而不仅仅是最近的父合约

简单例子



1. D调用foo的时候，由于B，C中的foo都没有使用super，所以只是覆盖问题，根据最远继承，C 覆盖了B，所以执行顺序为：D -> C -> A；
2. D调用bar的时候，由于B，C中的bar使用了super，此时D的两个parent都需要执行一遍，因此为D-> C -> B -> A，整个过程中A合约只会被调用一次。具体原因是Solidity借鉴了Python的方式，强制一个由基类构成的DAG（有向无环图）使其保证一个特定的顺序。

继承的状态变量覆盖

1. 状态变量无法进行重写，只能继承父类状态变量
2. 但是可以通过在子合约的构造函数中进行覆盖，从而达到重写目的

contract A {

    string public name = "Contract A";

    function getName() public view returns (string memory) {

        return name;

    }

}

// Shadowing is disallowed in Solidity 0.6 覆盖在0.6是不行的

// This will not compile

// contract B is A {

//     string public name = "Contract B";

// }

contract C is A {

    // This is the correct way to override inherited state variables.

    constructor() {

        name = "Contract C";

    }

    // C.getName returns "Contract C"

}

构造函数

1. 构造函数是可选的，在部署合约时会自动被调用
2. 在合约继承的时候，如果父合约有构造函数，则需要显示的对父合约进行构造

父构造函数总是按照继承顺序调用，而不管子合约的构造函数中列出的父合约的顺序如何。

Abstract

抽象合约的作用是将函数定义和具体实现分离，从而实现解耦、可拓展性，其使用规则为：

（1）当合约中有未实现的函数时，则合约必须修饰为abstract；

（2）当合约继承的base合约中有构造函数，但是当前合约并没有对其进行传参时，则必须 修饰为abstract；

（3）abstract合约中未实现的函数必须在子合约中实现，即所有在abstract中定义的函数都 必须有实现；

（4）abstract合约不能单独部署，必须被继承后才能部署；

Interface

可以使用Interface完成多个合约之间进行交互，interface有如下特性：

（1）接口中定义的function不能存在具体实现；

（2）接口可以继承；

（3）所有的function必须定义为external；

（4）接口中不能存在constructor函数；

（5）接口中不能定义状态变量。

（6）[abstract和interface的区别](https://medium.com/upstate-interactive/solidity-how-to-know-when-to-use-abstract-contracts-vs-interfaces-874cab860c56" \o "https://medium.com/upstate-interactive/solidity-how-to-know-when-to-use-abstract-contracts-vs-interfaces-874cab860c56)

Library

库与合约类似，限制：不能在库中定义状态变量，不能向库地址中转入ether；

库有两种存在形式：

1. 内嵌（embedded）：当库中所有的方法都是internal时，此时会将库代码内嵌在调用合约中，不会单独部署库合约；
2. 链接（linked）：当库中含有external或public方法时，此时会单独将库合约部署，并在调用合约部署时链接link到库合约。

（1）可以复用的代码可以编写到库中，不同的调用者可以linked到相同的库，因此会更加节约gas；

（2）对于linked库合约，调用合约使用delegatecall进行调用，所以上下文为调用合约；

（3）部署工具（如remix）会帮我们自动部署&链接合约库。

参考link：

Stack Overflow提问出和我相同的疑问：[https://ethereum.stackexchange.com/questions/106574/why-can-i-use-linked-libraries-on-remix-ide-without-deploying-them-first](https://ethereum.stackexchange.com/questions/106574/why-can-i-use-linked-libraries-on-remix-ide-without-deploying-them-first" \o "https://ethereum.stackexchange.com/questions/106574/why-can-i-use-linked-libraries-on-remix-ide-without-deploying-them-first)

讲解library：[https://www.youtube.com/watch?v=25MLAnIzXRw](https://www.youtube.com/watch?v=25MLAnIzXRw" \o "https://www.youtube.com/watch?v=25MLAnIzXRw)

讲解library（画了一个图）：[https://www.youtube.com/watch?v=iIMSMfArTiE](https://www.youtube.com/watch?v=iIMSMfArTiE" \o "https://www.youtube.com/watch?v=iIMSMfArTiE)

remix部署合约和库详解：[https://medium.com/remix-ide/deploying-with-libraries-on-remix-ide-24f5f7423b60](https://medium.com/remix-ide/deploying-with-libraries-on-remix-ide-24f5f7423b60" \o "https://medium.com/remix-ide/deploying-with-libraries-on-remix-ide-24f5f7423b60)

进阶

abi.encode、abi.decode、abi.encodePacked

（1）abi.encode：可以将data编码成bytes，生成的bytes总是32字节的倍数，不足32为会自动填充（用于给合约调用）；

（2）abi.decode：可以将bytes解码成data（可以只解析部分字段）

（3）abi.encodePacked：与abi.encode类似，但是生成的bytes是压缩过的（有些类型不会自动填充，无法传递给合约调用）。

（4）手册：[https://docs.soliditylang.org/en/v0.8.13/abi-spec.html?highlight=abi.encodePacked#non-standard-packed-mode](https://docs.soliditylang.org/en/v0.8.13/abi-spec.html?highlight=abi.encodePacked" \l "non-standard-packed-mode" \o "https://docs.soliditylang.org/en/v0.8.13/abi-spec.html?highlight=abi.encodePacked#non-standard-packed-mode)

Selector

方式1：abi.encodeWithSignature

拼装selector和函数参数，我们可以在A合约中得到calldata，并在A合约中通过call方法去调用B合约中的方法，从而实现合约间的调用。举例，下面的代码功能是：在当前合约中使用call调用addr地址中的transfer方法：

// 在合约中，一个function的4字节selector可以通过abi.encodeWithSignature(...)来获取

// "0xa9059cbb"

bytes memory transferSelector = abi.encodeWithSignature("transfer(address,uint256)");

// 调用合约

addr.call(transferSelector, 0xSomeAddress, 100);

// 一般会写成一行，简写如下：

// addr.call(abi.encodeWithSignature("transfer(address,uint256)"), 0xSomeAddress, 100);

方式2：keccak256方法（sha3哈希算法）

//注意我们这里做hash时，仅处理函数名与参数，并不会计算函数返回值 bytes4(keccak256(bytes("transfer(address,uint256)")))

方式3：在合约内部也可以直接获取selector

// 假设当前合约内有transfer函数

this.transfer.selector

方式4：也可以使用abi.encodeCall方式获取

interface IERC20 {

function transfer(address, uint) external;

}

function encodeCall(address to, uint amount) external pure returns (bytes memory) {

// Typo and type errors will not compile

return abi.encodeCall(IERC20.transfer, (to, amount));

}

Encode

Call

Fallback

1. fallback是特殊的函数，无参数，无返回值；

2. 何时会被调用：

（1）当被调用的方法不存在时，fallback会被调用，属于default函数；

（2）当向合约转ether但是合约不存在receive函数时；

（3）当向合约转ether但是msg.data不为空时。（即使receive存在）

3. 当使用transfer或者send对合约进行转账时，fallback函数的gaslimit限定为2300 gas

合约之间的调用

（1）使用合约实例调用合约（常规）：A.foo(argument)

（2）使用call调用合约: A.call(calldata)

（3）使用delegate调用合约：A.delegatecall(calldata)

Create

在EVM层面，一共有两个操作码（OPCODE）可以用来创建合约：

create：

原理：新生成地址 = hash(创建者地址, nonce)

特点：不可预测，因为nonce是变化的

create2：

原理：新生成地址 = hash("0xFF",创建者地址, salt, bytecodeHash)

特点：可以预测，因为没有变量

Try/catch

try/catch仅可以捕捉在调用external函数或创建合约中抛出的异常信息。

节约gas

（1）使用calldata替换memory

（2）将状态变量加载到memory中

（3）使用++i替换i++

（4）对变量进行缓存

（5）短路效应

节约gas的十种方法

见项目的solidity进阶，节约gas

Type

type(x) 可以返回x类型的对象信息，例如：

type(x).name: 合约的名字；

type(x).creattionCode: 合约部署时的bytecode；

type(x).runtimeCode: 合约运行时的bytecode，一般是构造函数数据，但是当constructor中有汇编时会有不同

剩余

Assembly

Merkletree

Signature

Permit

Math

Timelock

Upgrade

Event

Permit2

Signature-eip712

EIP协议

ERC20（见OZgame）

ERC721（见OZgame）

ERC721A对ERC721改进

1. 单笔交易种铸造多个NFT可节省大量gas
2. Azuki合约允许铸造多个NFT，其成本于铸造当NFT的成本基本相同

如何工作的？

1. 从OpenZepplin（OZ）ERC721Enumerable种删除重复存储

IERC721Enumerable的OZ实现包括每个令牌元数据的冗余存储。这种非规范化的方法对读函数进行了优化，但却付出了编写函数的巨大代价，考虑到用户不太可能为读函数付费，这种方法并不理想。此外，我们的令牌从0开始按顺序编号，这使我们可以从基本实现中删除一些冗余存储。

1. 每批铸币请求更新一次所有这的余额，而不是每次铸造的NFT

假设A有2个代币，想在购买5个，在solidity中，更新存储需要花费gas，所以通过一次将A的代币从2更新到7会更便宜，而不是2到3，3到4.

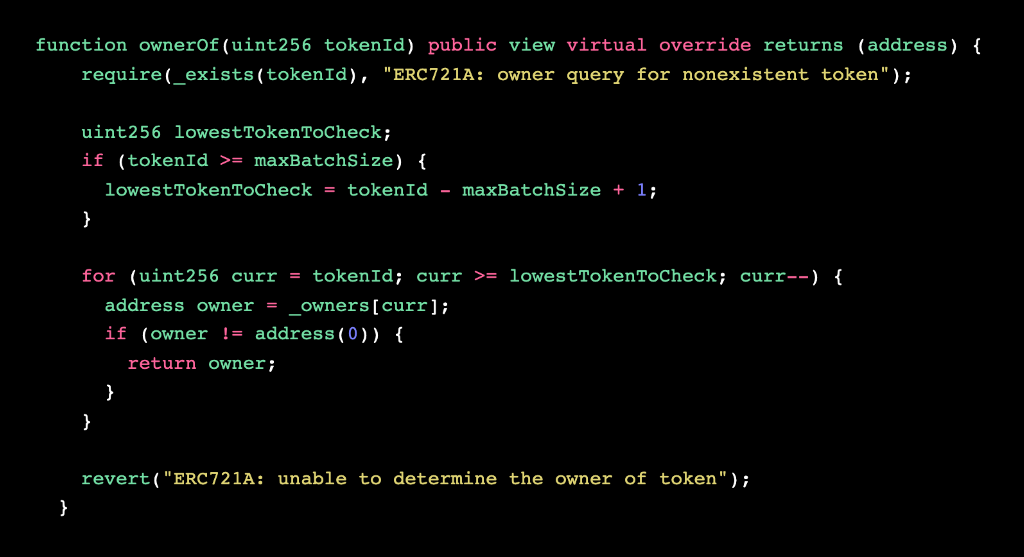
1. 每批铸币的请求更新一次所有者的余额，而不是每个铸造的NFT更新

这在精神上与优化 2 相似。假设 Alice 想购买 3 个代币 - 代币 #100、#101 和 #102。与其将 Alice 保存为 3 次所有者（每次都花费我们的 gas），不如以语义上暗示 Alice 拥有所有 3 个代币的方式保存所有者值一次。

如何？假设 Alice 铸造代币 #100、#101 和 #102，Bob 铸造代币 #103 和 #104。内部所有者跟踪器如下所示：



这里的关键是，如果我们想看看谁拥有 #102，我们实际上不需要将 Alice 明确设置为 #102 的显式所有者来这样做。我们可以更改 ownerOf 函数来执行以下操作：



ERC165（识别合约类型）

在opensea中，我们能够知道一个NFT的具体协议类型，即这个collection到底是ERC721还是ERC1155，

而之所以能够判断出NFT的协议类型，是因为标准的NFT的代码都会遵循一个标准，即[EIP-165]([https://eips.ethereum.org/EIPS/eip-165#how-interfaces-are-identified](https://eips.ethereum.org/EIPS/eip-165" \l "how-interfaces-are-identified" \o "https://eips.ethereum.org/EIPS/eip-165#how-interfaces-are-identified)[EIP section])。

EIP-165接口定义

EIP165提供了一种检测智能合约类型的方法，在实现上它只定义了一个接口，同时也明确了计算interfaceId的规则：对该协议（如NFT721）的所有接口selector的hash值做^运算（异或，XOR，^），再取前四字节作为interfaceId。

总结

（1）当想检查一个NFT721是否实现了EIP165标准时，只需要查询：supportsInterface(0x01ffc9a7)的返回值即可，true表示支持，false表示不支持；

（2）同样的道理：0x5b5e139f返回true表示支持IERC721Metadata，0x80ac58cd返回true表示为标准NFT721协议。

EIP2612

EIP712

EIP5114

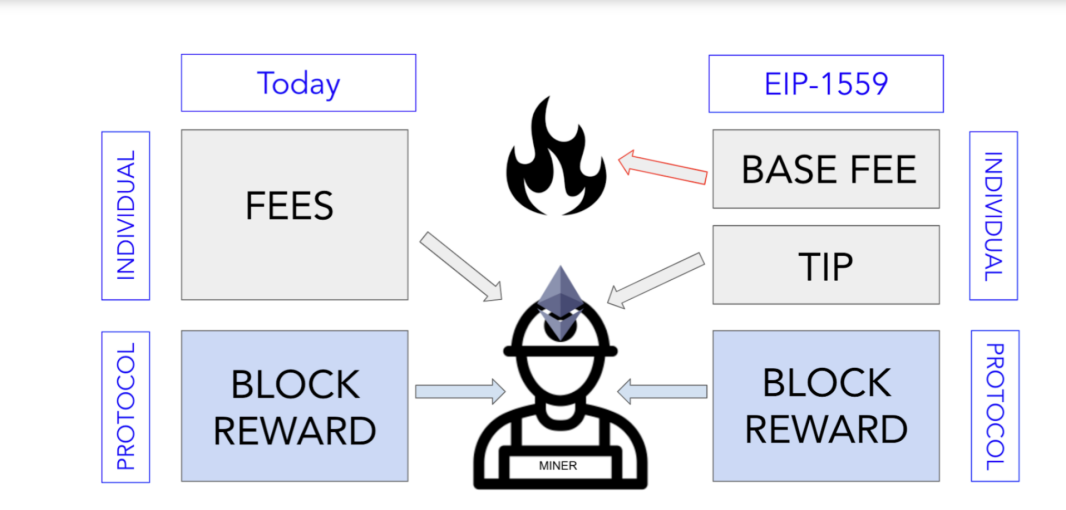
EIP1820

EIP4626

？

EIP1559

EIP1559修改了当前的矿工费计算模型，将其由原来的：用户任意出价矿工费（导致gas price过高的）模式改为现在的：base fee + tip模式（baseFee是当前交易打包时的基础gas price），tip是由用户自行定义的，并且引入了容量变化等其他因素，这种新模式使得gas price不会高的太离谱，gas fee可以动态调节。（并且其中：base fee部分系统会燃烧掉，矿工只会获取tip）



安全方面

1. 重入攻击

（恶意合约）内的函数反向调用A（原合约）的函数实现的攻击。

1. 返回值校验

调用外部适合函数时，有些函数调用失败不会抛出错误回滚交易而是返回false，如果忘记检查函数返回值会导致以为调用成功。

1. 合约自杀地dos

合约自杀时，会将合约自身持有地ether全部转入到指定地地址之中。

典型：

1. 合约使用了balance方法来进行校验。
2. 合约的receive函数或者fallback函数中存在一些函数调用，将状态变量修改为自毁合约的地址时。
3. 读取私有变量

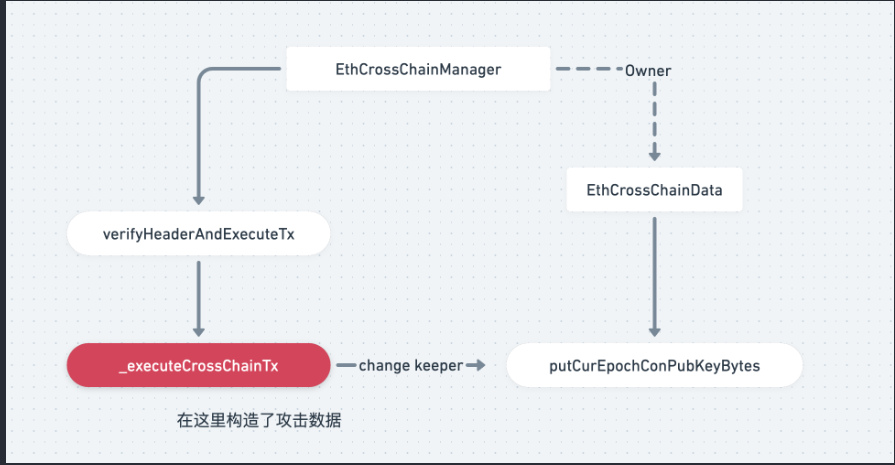
所有链上的数据都是可以被访问，不要吧任何敏感信息存储在链上。

Private修饰符表示链上合约无法访问该数据，但是可以通过链下（读取slot的方式读取私有数据）

安全事故分析

1. poly网络哈希碰撞攻击

通过构造特定的合约calldata（主要是四字节的sig），可以实现攻击，这个四字节sig可以通过hash碰撞来实现



1. Op代币重放攻击

当前的区块链系统中，用户对交易的签名有两种形式：

（1）包含6个字段信息（RLP编码元素）：nonce, gasprice, startgas, to, value, data。

（2）包含9个字段信息（RLP编码元素）nonce, gasprice, startgas, to, value, data, chainid, 0, 0，多了后面三个字段。（主要区别是chainid）

对于第一种交易，签名v值为：{0,1} + 27，第二种交易形式（符合EIP1155标准）：签名v值为：{0,1} + chainid \* 2 + 35

根据ECDSA签名原理我们知道（r，s，v）：在v值不相同的时候，即使r和s相同，还原出来的公钥也是不相同的。所以满足EIP1155标准的交易可以阻止重放攻击。

当前的区块链世界是一个多链并存的世界，并且很多链都是EVM兼容的，对于不带chainid的签名交易，我们可以把这条链上的交易信息读取出来，然后在另外一条链上执行，这样就会导致交易被再次执行（因为签名都是有效的），这就是所谓的重放攻击，所以使用chainid可以保证无法被重放，因为每条链的chainid不同。

1. FTX因手续费提币攻击

Xen Token最近太火了，算是熊市中的一股清流，让死寂的币圈又活跃了起来。

这个项目的参与算是0门槛，只需要支付gas费就可以mint，然后claim，越早进入mint的token越多，收益越高。

这么火🔥的项目自然会走入黑客的视野，果然，一场风暴就此降临。只不过受害者不是Xen Token项目方，而是无辜的FTX交易所，具体攻击结果为：

（1）黑客通过FTX的免手续费提币功能，通过技术手段，让FTX支付手续费， 疯狂调用Xen mint 1.7万次，获得大量的Xen Token，并通过Uniswap等dex 将Xen兑换为61个ETH，完成套现。

（2）FTX交易所前后损失：80多个ETH。

攻击流程

1. 攻击者在FTX上进行小额提币ETH，收款地址为自己的攻击合约（实现了fallback方法）；
2. fallback内部创建1～3个子合约；
3. 在子合约中：先调用xen的claimRank得到xen；
4. 在子合约中：再调用claimMintRewardAndShare方法，将子合约持有的xen转给黑客的EOA；
5. 子合约自杀；
6. Profanity弱随机数暴力破解攻击
7. 出块相关攻击

区块链中的交易排序是由矿工决定的，这使得矿工有套利空间，针对同一区块中的交易，主要有三种方式进行套利：

（1）抢跑：指通过让特定交易排在目标交易前而获利，主要针对清算和套利交易；

（2）尾随：指通过让特定交易排在目标交易之后而获利，主要针对预言机交易或大单交易；

（3）三明治攻击：上述两种攻击形式的结合，让目标交易恰好加载两笔特定构造交易中间，从而获利。

\*\*这种攻击大大拓宽了可攻击的范围，哪怕是一笔普通的AMM交易，都有可能成为针对对象。

\*\*攻击者的第一笔构造交易制造更大的交易价格波动，待目标交易执行完成之后，紧接着执行第二笔构造交易，换回发动攻击的代币完成获利。

（4）全节点矿工在交易确认前已经知道交易信息，所以它可以：

\*\*获取对自己有利的信息，构造高gasfee交易，提前成交（谜底等协议）

\*\*扩大approve授权额度：被授权人在委托人的新授权额度确认之前，可以构造一笔交易将原始的授权转移给别人，这样最终相当于委托人授权了：旧额度+新额度。

交易排序问题进一步导致了MEV（矿工可提取手续）问题，这是区块链发展的重要研究方向之一。

1. Msg.value持久化问题

在如下代码中，存在msg.value被重复使用的问题：例如：在delegatecall中，msg.value与当前batch函数中msg.value一致，这可能导致错误。或者其它的情况，例如：一个函数循环调用另一个函数。

function batch(bytes[] calldata calls, bool revertOnFail) external payable returns(bool[] memory success, bytes[] memory results) {

successes = new bool[](calls.length);

results = new bytes[](calls.length);

for (uint256 i=0; i< calls.length; i++) {

(bool success, bytes memory result) = address(this).delegatecall(calls[i]);

require(success || !revertOnFail, \_getReyerMsg(result));

successes[i] = success;

results[i] = result;

}

}

1. 用EOA来call方法攻击

合约地址与EOA地址的调用区别：

1. 注意，当使用合约进行call，delegatecall调用函数时，是永远都不会抛出异常的，哪怕被调用的函数不存在，它只能返回true或false。

I：当被调用的函数存在时，返回true，不存在时返回false；（fallback不存在情况下）

II：如果被调用函数不存在，但是实现了fallback，则返回true；

（2）对于非合约地址（EOA）调用函数时（这个是恶意行为，因为EOA根本没有函数），只要这个EOA的gas足够即可，返回的bool值居然永远都是true！

（3）攻击案例：[QBridege安全事件](https://halborn.com/explained-the-qubit-hack-january-2022/" \o "https://halborn.com/explained-the-qubit-hack-january-2022/)

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.13;

import "hardhat/console.sol";

contract Test {

    function TestEOA(uint \_num) public payable {

      address EOA = 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2;

      (bool success, bytes memory data) = address(EOA).call(

        abi.encodeWithSignature("mint(uint256)", \_num)

      );

            // 这个居然是true，这个存在攻击可能

      console.log("result:", success);

    }

    function TestNotExist(uint \_num) public payable {

      (bool success, bytes memory data) = address(this).call(

        abi.encodeWithSignature("notExist(uint256)", \_num)

      );

            // true，因为执行了fallback

            // 如果fallback不存在，则返回false，但是不会抛异常

      console.log("result:", success);

    }

    function TestExist(uint \_num) public payable {

      (bool success, bytes memory data) = address(this).call(

          abi.encodeWithSignature("mint(uint256)", \_num)

      );

      // true

      console.log("result:", success);

    }

    function mint(uint \_num) public payable {

        console.log("mint called:", \_num);

    }

    fallback() external payable {

      console.log("fallback called!");

    }

}

这里三个都是返回true

1. 可燃烧代币攻击

有些代币在转账时会销毁一部分转账费用，导致实际收到的代币余额偏少，如果开发者没考虑到这一点，以转账值计算，会导致出现偏差。

我们可以通过校验转账前后的token余额来进行校验，确保转账之后确实收到了预期数量的token，从而防止作恶行为。

1. 签名验证攻击

签名被重复使用，或者利用椭圆曲线签名算法的对称性，根据已有签名构造合法签名。

\*\*1. 案例：\*\*2021 年 7 月 12 日 AnySwap 被盗 800 万美金

2. 事故原因：对交易签名除了私钥外需要一个随机数 R，但是 Anyswap 部署新合约失误，导致在 BSC 上的 V3 路由器 MPC 帐户下有两个交易具有相同的 R 值签名，攻击者反推到这个 MPC 账户的私钥转走了被盗资金。

\*\*3. 解决方案：\*\*使用EIP-712标准验证签名，参考OpenZeppelin的实现：[https://docs.openzeppelin.com/contracts/3.x/api/drafts。](https://docs.openzeppelin.com/contracts/3.x/api/drafts%E3%80%82" \o "https://docs.openzeppelin.com/contracts/3.x/api/drafts%E3%80%82)

1. Delegatecall攻击

当合约中存在delegatecall操作时，要格外小心，我们可以把使用delegatecall的合约当成代理合约，可以通过业务合约（Attack合约）来对代理合约进行升级，从而修改了代理合约中的数据。

1. Tx.origin攻击

类似诱导你去转账，然后通过攻击合约的receiver再次调用你的转账合约，直到将你的钱包掏空

1. Dos
2. 非预期回滚，例如：向没有接收函数的合约转账
3. 权限操作，例如：将错误的地址授权导致无法正常工作
4. encodePacked攻击

abi.encodePacked() 采用非填充序列化，当序列化参数包含多个变长数组时，攻击者可以在保持所有元素顺序不变的前提下，改变两个变长数组的元素，如此序列化的结果相同。

1. 短地址攻击

如果传入的 to 是末端缺省的短地址，EVM 会将后面字节补足地址，而最后的 value 值不足则用 0 填充，导致实际转出的代币数值倍增，在执行transfer函数时，原型如下：

function transfer(address \_to, uint256 \_value) returns (bool success)`