Master Z. DEV.

Blockchain Developer

Lezione 14

Contratti upgreadabili

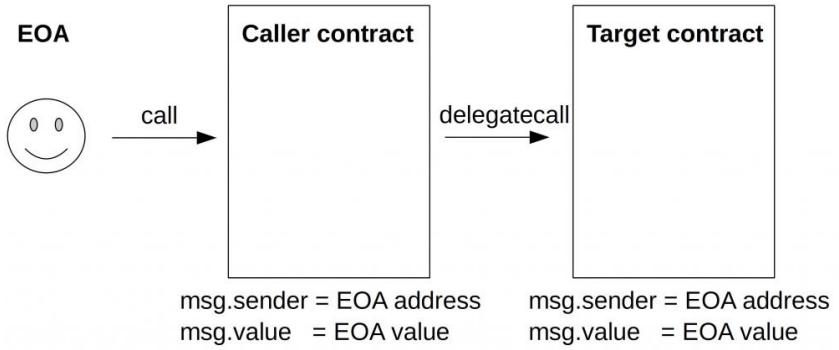
Upgrade di contratti

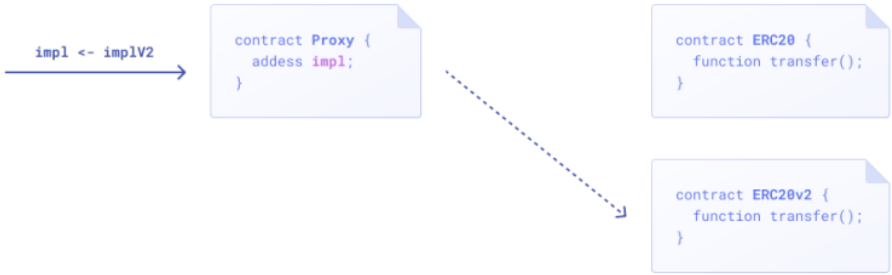
Perché?

- Bug fixes
- Nuove funzionalità

Modalità 1, nessun upgrade al codice:
Cambio di parametri => cambio di funzionamento (percentuali, indirizzi, ecc..)
Cambio contratto (migrazione) => cambio indirizzo, informazioni a tutti gli utenti, ecc…

Modalità 2 upgrade del codice ma nessun cambio di indirizzo:
Proxies, sfruttando funzioni come fallback e delegateCall.





Master DEV. Blockchain Developer

Delegate calls

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

// NOTE: Deploy this contract first
contract B {
    // NOTE: storage layout must be the same as contract A
    uint public num;
    address public sender;
    uint public value;

    function setVars(uint _num) public payable {
        num = _num;
        sender = msg.sender;
        value = msg.value;
    }
}

contract A {
    uint public num;
    address public sender;
    uint public value;

    function setVars(address _contract, uint _num) public payable {
        // A's storage is set, B is not modified.
        (bool success, bytes memory data) = _contract.delegatecall(
            abi.encodeWithSignature("setVars(uint256)", _num)
        );
    }
}
```

In this example, when someone will call *function setVars* from contract A, *delegatecall* will be called. Inside *delegatecall* we see that it will call a *function setVars(uint256)* from contract with address equal to _contract.
The value which will be sent to this function is equal to _num.
Then, *delegatecall* setVars function from contract B will start execute. Here is the core of it.
If this function change the value of any variable, it will be change not in contract B, but in contract A. How the function setVars from contract B will know which variable change? By name? No. To understand this, we should know that Solidity saves every variable in storage. Function called by *delegatecall* works on numbers of slot.
So, in contract B *function setVars* set num (variable in slot number 0) to _num, sender (variable in slot number 1) to msg.sender and value (variable in slot number 2) to msg.value.
When we call this function by *delegatecall* we set the variable in slot number 0 in contract A to _num, the variable in slot number 1 in contract A to msg.sender and so on. Therefore, it is very important to have the **same storage layout in both contract**.

How you see, *delegatecall* has a great power. But with great power comes great responsibility. So we have to be careful when we use it.

Master Z. DEV. Blockchain Developer

Upgrade di contratti

Terminologia:

- Implementation contract: il codice del contratto. Quando aggiornato, il deploy definirà un nuovo indirizzo per questo contratto
- Proxy contract: punta al corretto implementation contract e redirige le chiamate a questo contratto
- Le chiamate degli utenti avverranno attraverso il proxy
- Admin: utente/i che possono aggiornare il contratto di implementazione
- Tutte le variabili dello storage saranno immagazzinate nel contratto proxy!!
- La logica sarà contenuta nell'implementazione

Ci sono diversi modi per implementare queste logiche, ma ognuna di esse ha diversi contro, principalmente:

- Storage clashes => le variabili nello storage delle diverse implementazioni **non possono essere cambiate di posto** e bisogna procedere aggiungendo le nuove eventuali variabili appendendole alle esistenti (la locazione nello storage non può essere cambiata)
- Function selector clashes => le funzioni sono rappresentate da soli 4 bytes:

  bytes4 fSel = keccak256('nome(parametri)')

  Siccome sono pochi, può capitare che funzioni diverse abbiano lo stesso selettore, ad esempio 0x42966c68 viene generate sia da 'burn(uint256)' che da 'collate_propagate_storage(bytes16)'

https://www.4byte.directory/

Master Z. DEV. Blockchain Developer

Upgrade di contratti

Metodo 1: Transparent Proxy Pattern (https://blog.openzeppelin.com/the-transparent-proxy-pattern)

- I proxy admin non possono chiamare funzioni contenute nel contratto di implementazione, ma solo funzioni riservate agli admin, che sono quelle che regolano gli aggiornamenti.
- Gli user non hanno nessun potere sulle funzioni di aggiornamento

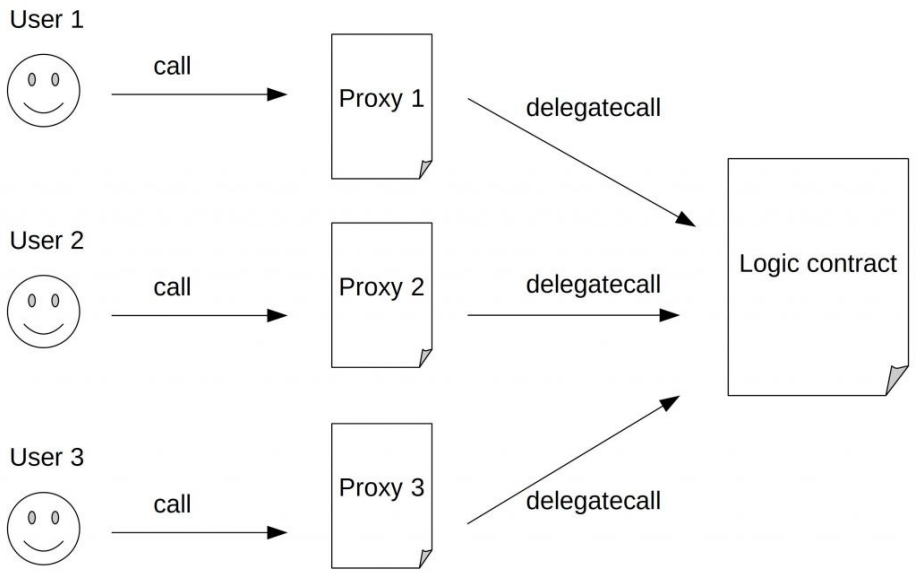
Metodo 2: Universal Upgradeable Proxies (UUPS) (https://eips.ethereum.org/EIPS/eip-1822)

- Le funzioni che regolano gli aggiornamenti sono messe nel contratto di implementazione (logic contract), e le funzioni con la stessa signature sono così eliminate dal compilatore stesso, e questo metodo permette anche di risparmiare gas quando avviene l'aggiornamento della logica del contratto rispetto al metodo precedente.
- Sembra tutto bello, ma se si deploya un contratto senza le funzioni di aggiornamento, il contratto non è più upgradabile! Oppure se si sbagliano i permessi di amministrazione il contratto diventa vulnerabile!

Metodo 3: Diamond Pattern (https://eips.ethereum.org/EIPS/eip-2535)
Permette di implementare diversi contratti nello stesso schema di proxy, usato molto spesso per aggiornare contratti davvero molto grandi in termini di spazio, tali da superare i 24kB di bytecode

Metodo 4: Beacon and upgradeable beacon (https://eips.ethereum.org/EIPS/eip-1967)

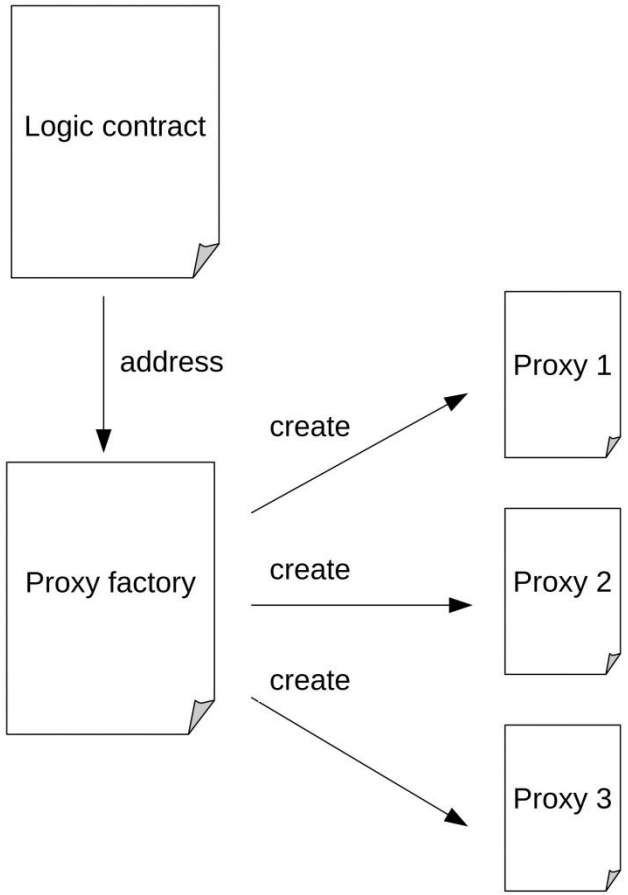
Proxy



A solution to the high gas costs mentioned in the normal factory approach is to use a minimal proxy. Instead of deploying a new contract with all its logic, every time we need a new instance, we deploy a tiny (minimal) contract that simply forwards our function call with the delegateCall statement to a contract at a fixed address that contains all logic.
We call the contract that does all the logic logic contract. The contract that is there for forwarding the call is called proxy.
With the delegate call statement, the called target contract (logic contract) is executed in the context of the calling (proxy) contract. All changes that are made during a delegate call are stored in the caller's realm.

https://eips.ethereum.org/EIPS/eip-1167

Proxy



Basic Concept
The proxy contract has four tasks:

1. Taking the arguments from the call
2. Forward the arguments via delegatecall to the logic contract
3. Receive the return values from the logic contracts
4. Return the values from 3. if successful

Our proxy contract only needs one function that forwards all calls. The arguments contain the function name and parameters of the target function in the logic contract. All variables and balances are stored under the address of the proxy contract.
In order to set up the minimal proxy, we need a logic contract, a proxy factory, and the proxy contract. The address of the logic contract is used as the target address in the proxy contracts.

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/proxy/Clones.sol

Master Z. DEV. Blockchain Developer

Proxy Libraries

A seconda della suite che state usando (Truffle, Hardhat, Foundry), si possono usare delle librerie come quelle di OpenZeppelin, di cui trovate la documentazione qui:
https://docs.openzeppelin.com/upgrades-plugins/1.x/

Documentazione su come scrivere contratti upgradeabili:
https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable

Documentazione sui pattern da seguire per scrivere contratti upgradeabili:
https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies

Github repo con esempi:
https://github.com/HashHaran/hardhat-upgrades/tree/main

Grazie per l’attenzione