📖 Solidity: interazioni con altri contratti

- call
- staticcall
- delegatecall
- abi.*
- Interagire con token non-ERC20 compliant
- Multisig
- Multicall

Esecuzione di uno smart contract

Quando la EVM esegue uno smart contract viene creato un **execution context** che consiste di varie regioni di memoria:

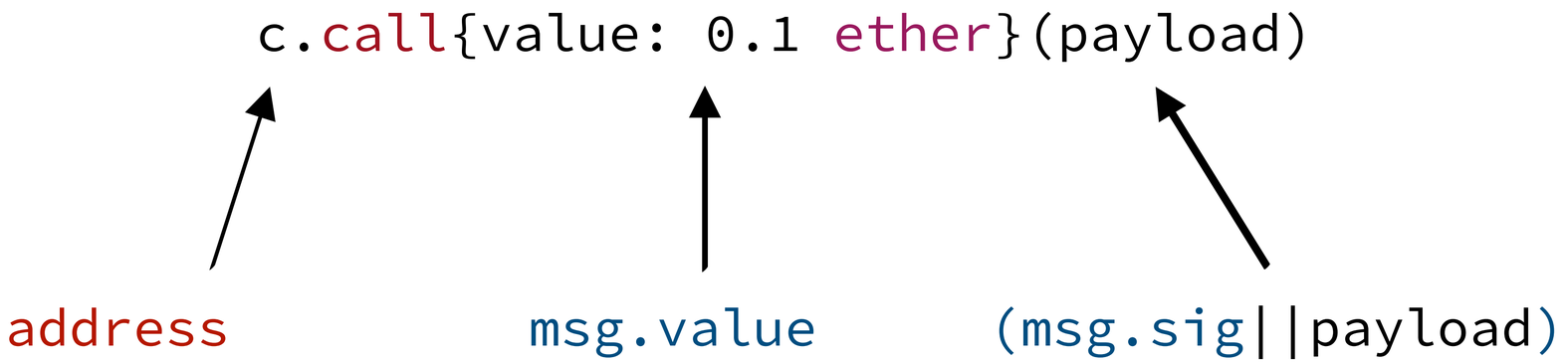
- **Code**: la regione di memoria in cui vengono conservate le istruzioni
- **Stack**: una lista di elementi di 32 byte (word). Utilizzato per gestire gli argomenti che serviranno alle istruzioni successive e viene manipolato dalle istruzioni stesse.
- **Memory**: una regione che esiste durante l'esecuzione dello smart contract, inizializzata a 0. È possibile accedere tramite MLOAD e MSTORE scegliendo un offset
- **Storage**: memoria persistente di uno smart contract. È una mappa con chiavi di 32 byte e valori di 32 byte. Un valore non inizializzato è di default 0.
- **Calldata**: memoria immutabile che contiene i dati che vengono propagati con una transazione. Nel caso della creazione di uno smart contract contiene il codice per il costruttore (init code).
- **Return data:** utilizzata per ritornare un valore dopo una chiamata. Viene manipolata dai contratti tramite gli opcode RETURN e REVERT.

Esecuzione di uno smart contract

Il **context** di un contratto è valido fino alla fine dell'esecuzione del contratto stesso.

Quando viene effettuata una chiamata ad un contratto esterno viene creato un nuovo **context** per eseguire il codice chiamato.

In Solidity è possibile fare una chiamata di *basso livello* utilizzando call.



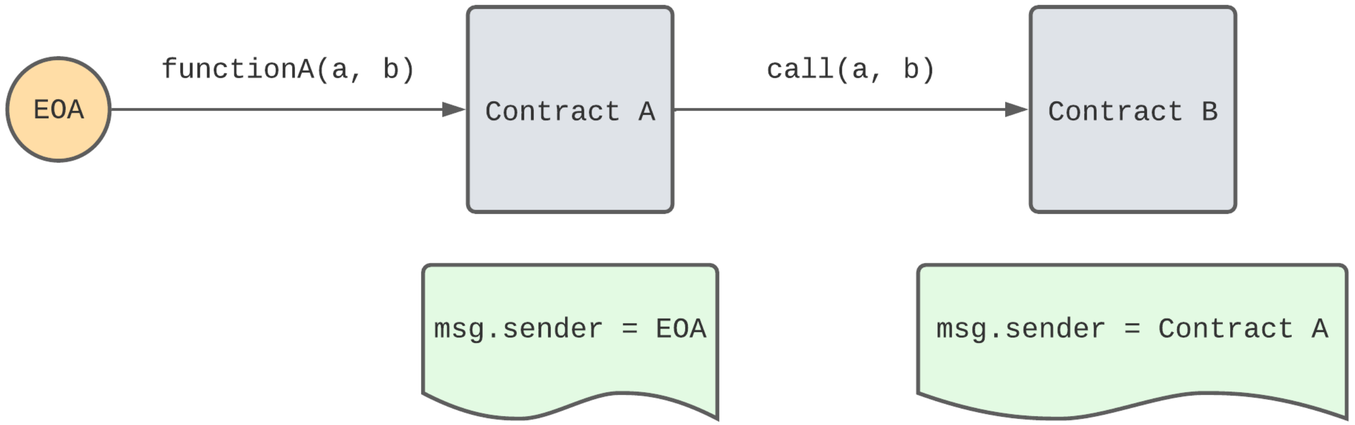
call

L’istruzione call ritorna una tupla (bool, bytes memory).

Il primo valore indica la corretta esecuzione del contratto (false nel caso di revert da parte del contratto chiamato).

Il secondo valore indica il valore di ritorno del contratto chiamato (eg. un bool nel caso del metodo transfer di un ERC20).

Il cambio di contesto implica che i campi della variabile d’ambiente msg cambieranno.



Esempio di call in Solidity

```
dai = address(0x6B175474E89094C44Da98b954EedeAC495271d0F); // indirizzo di DAI
vb = address(0xAb5801a7D398351b8bE11C439e05C5B3259aeC9B); // vitalik b.

// costruiamo la chiamata dai.transfer(vb, 1e18)
payload = abi.encodeWithSignature("transfer(address, uint256)", vb, 1e18);

// payload = 0xa9059cbb000000000000000000000000ab5801a7d398351b8be11c439e05
c5b3259aec9b0000000000000000000000000000000000000000000000000de0b6b3a7640000;
// se la chiamata va a buon fine, success = true

bool success = dai.transfer(vb, 1e18); //

(bool success, bytes memory ret) = dai.call(payload); // effettuiamo la chiamata
```

Utilizzo di call per gestire ERC20 non-compliant

Alcuni dei token pre-EIP20 non seguono lo standard di ritornare un valore bool sulle funzioni transfer, transferFrom, approve.

È possibile gestire questi edge case utilizzando call.

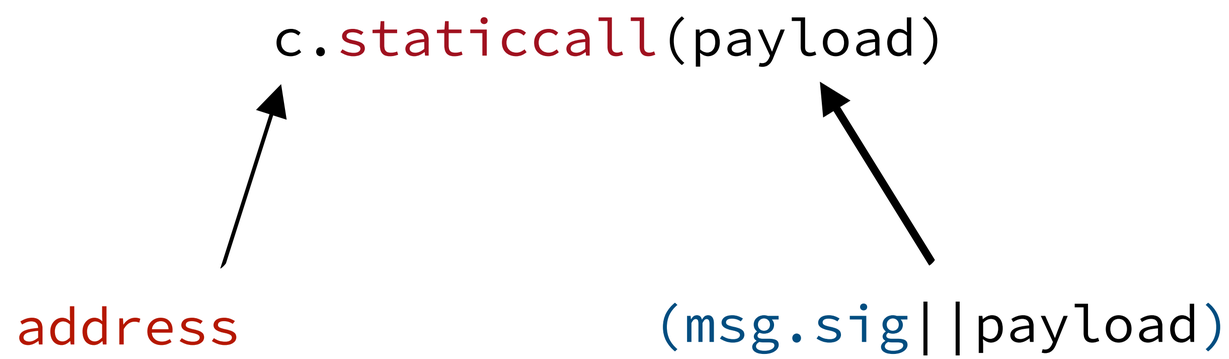
```
function safeTransfer(address token, uint256 amount) external {
    (bool success, bytes memory ret) = token.call(
        abi.encodeWithSignature(
            “transfer(address, uint256)”, msg.sender, amount)
        )
    );

    require(success);

    if(ret.length > 0) {
        require(abi.decode(ret, (bool)), “operation failed”);
    }
}
```

staticcall

L’istruzione staticcall è una variante di call che non prevede però nessun cambio di stato all’interno della EVM (nessun cambio allo storage).

Il comportamento di staticcall è uguale a quello di call tranne per il fatto che solleverà un’eccezione nel caso di cambio di stato.

È utile per chiamare metodi view di altri contratti.

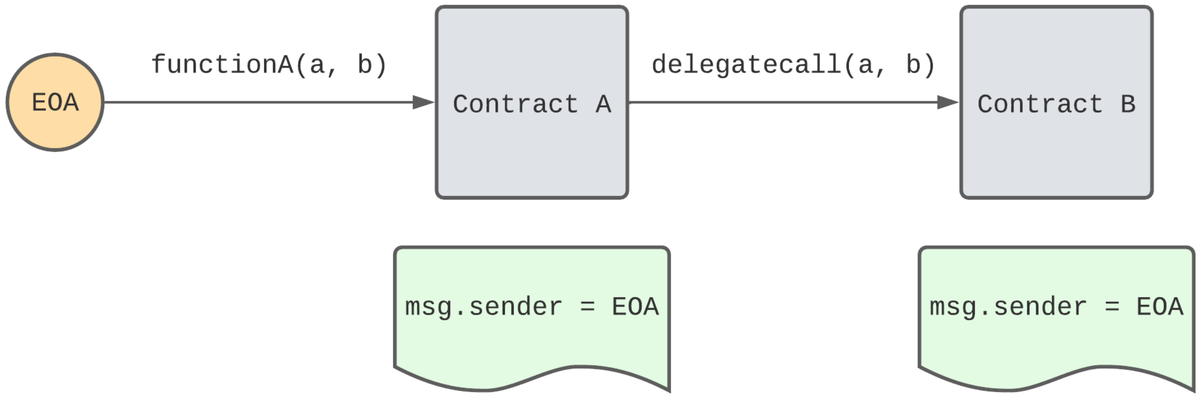


delegatecall

L’istruzione delegatecall permette di chiamare un contratto con il context del contratto chiamante.

Ciò comporta che non verrà creato un nuovo context. Verrà utilizzato il context del contratto chiamante (storage, msg) tranne che per il codice.

È utile per: multicall, proxy (upgrade di smart contract).



abi.*

Il payload della chiamata di un metodo di uno smart contract è serializzato utilizzando ABI encoding.

In Solidity è possibile (de)serializzare utilizzando ABI encoding tramite un gruppo di metodi messi a disposizione dal linguaggio:

- abi.decode(bytes memory, (..)) può essere usato per decodificare dei parametri data una tupla di tipi
- abi.encode(..) può essere usato per codificare una lista di valori (ritorna bytes memory)
- abi.encodePacked(..): come abi.encode ma i valori sono codificati in modo *packed* (vengono usati i byte richiesti dal valore stesso, ritorna bytes memory)
- abi.encodeWithSelector(bytes4, (..)) permette di codificare una chiamata ad un metodo
- abi.encodeWithSignature(string memory, (..)) permette di codificare una chiamata ad un metodo (non richiede il selector codificato, ma la stringa, eg. “transfer(address,uint256)”)
- abi.encodeCall(function, (..)) permette di codificare una chiamata ad un metodo (richiede il puntatore alla funzione da chiamare)

Multisig

Un multisig (multi-signature) permette di eseguire transazioni una volta raggiunto il consenso tra n parti off-chain, le quali autorizzeranno una transazione tramite una firma crittografica (eg. una transazione).

In particolare un multisig può:

- mantenere una lista di *signer*
- eseguire transazioni una volta raggiunta la quota di firme necessaria

Multicall

Un contratto multicall permette di eseguire una sequenza di azioni definite in una sola transazione (fallendo nel caso in cui una delle azioni fallisse).

Multisig e Multicall: uso di delegatecall e call per creare batch di transazioni