Eternal Storage

A compilation of patterns and best practices for the smart contract programming language Solidity

View on GitHub

Eternal Storage

Intent

Keep contract storage after a smart contract upgrade.

Motivation

When upgrading a smart contract, e.g. with the help of the Proxy Delegate pattern, what really happens is that a new version of the contract is deployed to the network and coexists with the old version. Because the old contract is not actually updated to a new version, the accumulated storage still resides at the old address. This usually includes important data, like user information, account balances or references to other contracts, which are still needed in the new version of the contract. One option would be to implement functionality to read every item from the old storage and store it in the new one. There are at least two issues with this approach: Firstly, writing to storage is one of the most expensive operations in Ethereum. Successively reading every single storage entry and storing it at the new address, every time a contract is upgraded, would be unreasonable from an economic point of view. There would even be a chance that the transaction carrying out the storage migration would run out of gas, in case there are too many entries to store. Secondly, the whole storage migration would need to be already planned during creation time and a lot of additional logic would need to be included, to carry out the migration.

A more practical solution was first proposed in 2016. It solves the problems of migrating storage by separating the storage from contract logic. A separate contract, with the sole purpose of acting as a storage to another contract, is introduced. It should be as flexible as possible, in order to avoid the need for an upgrade of the storage structure, as this would introduce the same problems as explained before. The storage is supposed to last over the whole lifetime of the initial contract, therefore the name eternal storage. A new version of the smart contract can simply use the same storage contract as its predecessor, after it has been registered.

Applicability

Use the Eternal Storage pattern when

- your contract is upgradeable and should retain storage after an upgrade.
- you want to avoid problems with migrating storage after a contract upgrade.
- you can accept a slightly more complex syntax for storing data.

Participants & Collaborations

There are three entities involved in this pattern. The central point is the smart contract implementing the Eternal Storage pattern. It provides its storage for another contract. The administrative work is done by an owner, which could be the person responsible for the DApp, or an autonomous organization. The administrator can set the address of the latest version of the contract using the storage, and update it once the address has changed due to an upgrade. The last remaining entity is the contract in need of the storage. This is the contract at the address set by the administrator, and has the authorization to send and retrieve elements from the storage contract.

Implementation

It would be possible to implement this pattern with a rigid representation of the needed storage, by implementing only the currently used data types in the eternal storage. To avoid upgrades to the data store, however, it should be designed as flexible as possible. This flexibility is achieved by implementing several mappings, one for each data type, in which data can be stored. These mappings map the abstracted down value to a certain sha3 hash, acting as a key-value storage. A sha3 hash is used as the key, in order to allow identifiers of arbitrary length to be used as keys. Using hashes as keys also enables the storage of complicated data types, like mappings (e.g. using `keccak256("balances", "UserID123")` as the key for storing the balance of the user with the ID 123).

Each mapping should be equipped with three functions to manage storage, retrieval and deletion. The storage function stores a provided value and the associated key in the respective mapping, depending on the data type of the value. The retrieval function returns the value for a provided key. To delete an existing entry in a mapping, the deletion function is called with the key of the item to be deleted as an input parameter. Because the functions for storage and deletion are affecting the contract state, they should be guarded by the Access Restriction pattern, so access is only allowed from the most recent version of the contract using the eternal storage. The address of the current version can be stored in a variable and should only be changeable by authorized addresses.

The hashing of the storage key should take place at the calling contract in order to have a uniform function interface for the eternal storage that always expects keys of the `bytes32` data type. Since the proposed approach is more complex than regular value storing, wrappers can help to reduce complexity and make code more readable.

Sample Code

The following sample code showcases a possible implementation of the Eternal Storage pattern and is inspired by this post. For the sake of space, this implementation only features the two data types `uint` and `address`. The remaining data types are implemented accordingly. The full implementation with a total of six data types can be found in the GitHub repository.

```
// This code has not been professionally audited, therefore I cannot make any promises about
// safety or correctness. Use at own risk.
contract EternalStorage {

    address owner = msg.sender;
    address latestVersion;

    mapping(bytes32 => uint) uIntStorage;
    mapping(bytes32 => address) addressStorage;

    modifier onlyLatestVersion() {
       require(msg.sender == latestVersion);
        _;
    }

    function upgradeVersion(address _newVersion) public {
        require(msg.sender == owner);
        latestVersion = _newVersion;
    }

    // *** Getter Methods ***
    function getUint(bytes32 _key) external view returns(uint) {
        return uIntStorage[_key];
    }

    function getAddress(bytes32 _key) external view returns(address) {
        return addressStorage[_key];
    }

    // *** Setter Methods ***
    function setUint(bytes32 _key, uint _value) onlyLatestVersion external {
        uIntStorage[_key] = _value;
    }

    function setAddress(bytes32 _key, address _value) onlyLatestVersion external {
        addressStorage[_key] = _value;
    }

    // *** Delete Methods ***
    function deleteUint(bytes32 _key) onlyLatestVersion external {
        delete uIntStorage[_key];
    }

    function deleteAddress(bytes32 _key) onlyLatestVersion external {
        delete addressStorage[_key];
    }
}
```

In line 3 the variable `owner` is initialized with `msg.sender` to specify an authorized entity. Administration could be implemented in several other ways, for example to authorize several addresses, or let an autonomous organization take the role of the owner. Line 4 stores the current address of the latest version of the contract using this storage. The mappings in line 6 and 7 are acting as the key-value store, where the actual data is stored. The `onlyLatestVersion()` modifier from line 9 makes sure, that only the address of the latest version is able to insert and delete values from storage. The `upgradeVersion(..)` function in line 14 is only callable by the owner (this is achieved with a simple `require` statement instead of a modifier, because it is the only function in the contract that should only be callable by the owner) and lets him update the address of the latest contract version.

The two functions from line 20 onward are the getters and return the value for a provided key. The two following functions in line 29 and 33 are setters that take a hash and a value as input parameters and store them in the respective mapping. The last two functions in line 38 and 42 are responsible for deleting entries from the mappings when provided with a key. These two functions, as well as the two setters right before, are only callable by the latest version of the contract, because they make use of the `onlyLatestVersion()` modifier.

The upgradeable contract that uses the eternal storage, can implement wrappers to facilitate dealing with the unfamiliar syntax using hashes as keys. The following code shows three exemplary wrappers to help manage user balances.

```
function getBalance(address balanceHolder) public view returns(uint) {
    return eternalStorageAdr.getUint(keccak256("balances", balanceHolder));
}

function setBalance(address balanceHolder, uint amount) internal {
    eternalStorageAdr.setUint(keccak256("balances", balanceHolder), amount);
}

function addBalance(address balanceHolder, uint amount) internal {
    setBalance(balanceHolder, getBalance(balanceHolder) + amount);
}
```

The `getBalance(..)` function retrieves the balance for a key, which is generated as the hash of the string `"balances"` combined with the address of the balance holder. The `setBalance(..)` function works in a similar way, only setting a balance instead of getting it. In both functions a combination of variables is hashed in order to generate a unique key that can be used to access the mappings. The last function `addBalance(..)` makes use of the previous two by first getting the balance of a user, then adding an amount to it and storing it again in the end. It is a good example for how wrappers can help reduce complexity, for example by concealing the hashing mechanism.

Consequences

The obvious advantage of the Eternal Storage pattern is the elimination of the need for storage migration after upgrading a smart contract. A newly deployed contract version, can call the same storage contract that its predecessor used, after it has been registered. It can read from it or store new key-value pairs. Positive consequences specific to the proposed approach, with the use of hashes as keys in the key-value store, revolve around flexibility. The eternal storage is flexible, because virtually every data type can be stored. That in turn makes the eternal storage flexible against any possible changes in the data scheme of the calling contract, without having to upgrade it.

Several negative consequences have to be considered before implementing this pattern as a storage solution. The separation of logic and storage increases complexity, because external calls have to be made. External calls should always be handled with caution, as they can cause unintended behavior. Further complexity is introduced by additional syntax, including the need for hash functions, which can be mitigated to some extend by the use of wrappers. As already addressed in the Proxy Delegate pattern, the bypass of immutability can influence the trust users are willing to put into a DApp. Depending on the exact implementation, an authorized entity could be able to set a malicious contract as the latest version and alter the storage to his advantage. This issue can be mitigated by a proper rule set for version changes, or decentralized ownership.

Known Uses

The Eternal Storage pattern is used in the upgradeability strategy of Rocket Pool, a decentralized proof of stake pool. They try to overcome the issue of trust loss by disabling the direct access of the owner to the contract, after he has initialized it.

< Back