Proxy Delegate

Intent

Introduce the possibility to upgrade smart contracts without breaking any dependencies.

Motivation

Mutability in Ethereum is hard to achieve, but necessary. It allows developers to adapt to a changing environment and to react to bugs and other errors. To overcome the limitations introduced by the immutability of contract code, a contract can be split up into modules, which are then virtually upgradeable. They are only virtually upgradeable, because existing contracts still cannot be changed. However, a new version of the contract can be deployed and its address replaces the old one in storage. To avoid breaking dependencies of other contracts that are referencing the upgraded contract, or users who do not know about the release of a new contract version (that comes with a new address), we make use of a proxy (sometimes also called dispatcher) contract that delegates calls to the specific modules. These modules are also called delegates, as work is delegated to them by the proxy. A first functional version of this pattern was introduced in 2016.

This example makes use of a special message call, named `delegatecall`. Using this new message call allows a contract to pass on the function call to the delegate without having to explicitly know the function signature, a crucial point for upgradeability. Another difference to a regular message call is, that with a `delegatecall` the code at the target address is executed in the context of the calling contract. This means that the storage and state of the calling contract are used. Additionally, transaction properties like `msg.sender` and `msg.value` will remain the ones of the initial caller.

This pattern often goes hand in hand with the Eternal Storage pattern to further decouple storage from contract logic.

Applicability

Use the Proxy Delegate pattern when

- you want to delegate function calls to other contracts.
- you need upgradeable delegates, without breaking dependencies.
- you are familiar with advanced concepts like delegatecalls and inline assembly.

Participants & Collaborations

There are several participants interacting with each other in this pattern. The basic idea is that a caller (external or contract address) makes a function call to the proxy, which delegates the call to the delegate, where the function code is located. The result is then returned to the proxy, which forwards it to the caller. To know at which address the current version of the delegate resides, the proxy can either store it itself in a variable, or in case the Eternal Storage pattern is used, consult the external storage for the current address.

Because `delegatecall` is used to delegate the call, the called function is executed in the context of the proxy. This further means that the storage of the proxy is used for function execution, which results in the limitation that the storage of the delegate contract has to be append only. What this means is, that in case of an upgrade, existing storage variables cannot be omitted or changed, only new variables are allowed to be added. This is because changing the storage structure in the delegate would mess up storage in the proxy, which is expecting the previous structure. An example for this behavior can be found in the GitHub repository.

Implementation

The implementation of the Proxy part of the pattern is more complex than most of the other patterns presented in this document. A `delegatecall` is used to execute functions at a delegate in the context of the proxy and without having to know the function identifiers, because `delegatecall` forwards the `msg.data`, containing the function identifier in the first four bytes. In order to trigger the forwarding

mechanism for every function call, it is placed in the proxy contract's fallback function. Unfortunately a `delegatecall` only returns a boolean variable, signaling the execution outcome. When using the call in the context of a proxy, however, we are interested in the actual return value of the function call. To overcome this limitation, inline assembly (inline assembly allows for more precise control over the stack machine, with a language similar to the one used by the EVM and can be used within solidity code) has to be used. With the help of inline assembly we are able to dissect the return value of the `delegatecall` and return the actual result to the caller. Due to the complexity of inline assembly, any further explanation on the implemented functionality, will be done with the help of an example in the Sample Code section of this pattern. One way of circumventing the need for inline assembly would be returning the result to the caller via events. While events cannot be accessed from within a contract, it would be possible to listen to them from the front end and act according to the result from there on. This method, however, will not be discussed in this pattern.

As stated in the Participants & Collaborations section, the upgrading mechanism, hence, storing of the current version of the delegate, can either happen in external storage or in the proxy itself. In case the address is stored in the proxy, a guarded function has to be implemented, which lets an authorized address update the delegate address.

The delegate can be implemented in the same way as any regular contract and no special precautions have to be taken, as the delegate does not have to know about the proxy using its code. The only thing that has to be taken into account is, that while upgrading the contract, storage sequence has to be the same; only additions are permitted.

Sample Code

This generic example of a Proxy contract is inspired by this post and stores the current version of the delegate in its own storage. Because the design of the Delegate contract can take many forms, there is no explicit example given.

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

    address delegate;
    address owner = msg.sender;

    function upgradeDelegate(address newDelegateAddress) public {
        require(msg.sender == owner);
        delegate = newDelegateAddress;
    }

    function() external payable {
        assembly {
            let _target := sload(0)
            calldatacopy(0x0, 0x0, calldatasize)
            let result := delegatecall(gas, _target, 0x0, calldatasize, 0x0, 0)
            returndatacopy(0x0, 0x0, returndatasize)
            switch result case 0 {revert(0, 0)} default {return (0, returndatasize)}
        }
    }
}
```

The address variables in line 3 and 4 store the address of the delegate and the owner, respectively. The `upgradeDelegate(..)` function is the mechanism that allows a new version of the delegate being used, without the caller of the proxy having to worry about it. An authorized entity, in this case the owner (checked with a simple form of the Access Restriction pattern in line 7) is able to provide the address of a new delegate version, which replaces the old one (line 8).

The actual forwarding functionality is implemented in the function starting from line 11. The function does not have a name and is therefore the fallback function, which is being called for every unknown function identifier. Therefore, every function call to the proxy (besides the ones to `upgradeDelegate(..)`) will trigger the fallback function and execute the following inline assembly code: Line 13 loads the first variable in storage, in this case the address of the delegate, and stores it in the memory variable `_target`. Line 14 copies the function signature and any parameters into memory. In line 15 the `delegatecall` to the `_target` address is made, including the function data that has been stored in memory. A boolean containing the execution outcome is returned and stored in the `result` variable. Line 16 copies the actual return value into memory. The switch in line 17 checks whether the execution outcome was negative, in which case any state changes are reverted, or positive, in which case the result is returned to the caller of the proxy.

Consequences

There are several implications that should be considered when using the Proxy Delegate pattern for achieving upgradeability. With its implementation, complexity is increased drastically and especially developers new to smart contract development with Solidity, might find it difficult to understand the concepts of delegatecalls and inline assembly. This increases the chance of introducing bugs or other unintended behavior. Another point are the limitations on storage changes: fields cannot be deleted nor rearranged. While this is not an insurmountable problem, it is important to be aware of, in order to not accidentally break contract storage. An important negative consequence from a social perspecive is the potential loss in trust from users. With upgradeable contracts, immutability as one of the key benefits of blockchains, can be avoided. Users have to trust the responsible entities to not introduce any unwanted functionality with one of their upgrades. A solution to this caveat could be strategies that only allow for partial upgrades. Core features could be non-upgradeable, while other, less essential, features are implemented with the option for upgrades. If this approach is not applicable, a trust loss could also be mitigated by introducing a test period, during which upgrades can be carried out. After the expiration of the test period, the contract cannot be changed any longer.

Besides these negative consequences, the Proxy Delegate pattern is an efficient way to separate the upgrading mechanism from contract design. It allows for upgradeability, without breaking any dependencies.

Known Uses

Implementations of the Proxy Delegate pattern are more likely to be found in bigger DApps, containing a large number of contracts. One example for this is Augur, a prediction market that lets users bet on the outcome of future events. In this case the address of the upgradeable contract is not stored in the proxy itself, but in some kind of address resolver.

< Back
