Leo Zhang Follow

Oct 20, 2019 · 7 min read ★ · Listen

Ethereum Standard ERC165 Explained

In smart contract development, a standard called ERC165 often appears when contract to contract interaction is needed. Solidity already provides a way for a contract to call functions from another contract. But why is ERC165 needed?

In this tutorial, I'll go through a simple example and explain the motivations behind ERC165, and how it works.

What is ERC165 for?

Let's say we have a simple Store contract that stores a uint256 value which can be queried or updated:

```
// Store.sol

pragma solidity 0.5.8;

contract Store {
  uint256 internal value;

  function setValue(uint256 v) external {
    value = v;
  }

  function getValue() external view returns (uint256) {
    return value;
  }
}
```

In solidity, we can implement it in two steps:

Step 1, define a contract interface *StoreInterface* which includes the definition of the function to call. We have to name the function *getValue* so that it matches the function name in the *Store* contract we are trying to call:

```
// StoreInterface.sol

pragma solidity 0.5.8;

interface StoreInterface {
  function getValue() external view returns (uint256);
}
```

Step 2, import this interface into the *StoreReader* contract, and in the readStoreValue function, cast the input address into the *StoreInterface*. Since the *getValue* function was specified in the *StoreInterface* definition, the *getValue* function from the *Store* address is exposed to us. We can call the *getValue* function directly:

```
// StoreReader.sol

import “./StoreInterface.sol”;

contract StoreReader {
  function readStoreValue(address store)
    external view returns (uint256) {
    return StoreInterface(store).getValue();
  }
}
```

This works when we pass in an address of a *Store* contract. But there is a problem: what if the address is not a *Store* address? It’s possible that the address is a user account or some contract that doesn’t have a *getValue* function, in which case the transaction will fail and revert.

We don’t like that *readStoreValue* can be called with an invalid *Store* address. However, there is no way to prevent that. Since an input address could be any account address or any contract address, we have to have a way to handle them

 

How do we add this validation check?

That is what ERC165 is for — a standard to publish and detect what interfaces a smart contract implements.

This standard will help us implement this check. Let's see how it works.

ERC165 defines an interface, which includes a single function definition -*supportsInterface*:

```
// ERC165/ERC165.sol

pragma solidity 0.5.8;

interface ERC165 {
  function supportsInterface(bytes4 interfaceID)
    external view returns (bool);
}
```

If a contract follows the ERC165 standard, it will publish what interfaces it supports. Then, other contracts can utilize the published information to avoid calling unsupported functions.

Why does EC165 define supportsInterface instead of supportsFunction?

A *supportsFunction* function could tell you whether a contract supports certain functions, but the problem is that some interfaces might happen to define the exact same function that your interface defines, in which case *supportsFunction* can't distinguish the "supported" function from your expected interface or some other unexpected interface.

Another factor is that supporting an interface implies supporting all functions that the interface includes, which is a more effective way to publish what functions a contract supports.

So in order for the *Store* contract to publish that it supports *StoreInterface*, it needs to implement the *supportsInterface* function to return *true* if the input *interfaceID* is the interface ID of *StoreInterface*, and return *false* if the interface ID is something else.

But wait, what is interface ID? And why it's a bytes4 value?

Interface ID is a unique identifier for an interface. We will explain how it's calculated later. For now,


```
// Store.sol

pragma solidity 0.5.8;

import "./StoreInterface.sol";
import "./ERC165/ERC165.sol";

contract Store is ERC165, StoreInterface {
  uint256 internal value;

  function setValue(uint256 v) external {
    value = v;
  }

  function getValue() external view returns (uint256) {
    return value;
  }

  function supportsInterface(bytes4 interfaceId) external view returns (bool)
{
    return interfaceId == 0x01ffc9a7 ||
           interfaceId == 0x75b24222;
  }
}
```

If a contract supports multiple interfaces, then it can let *supportsInterface* return *true* for any supported interface IDs. That's why the *supportsInterface* also return *true* for ERC165's interface ID, which is *0x01ffc9a7*.

OK. Now we can refactor the *StoreReader* contract, and let the *readStoreValue* function check if the remote contract publishes that it supports *StoreInterface*, and revert if it doesn't support:

```
// contracts/StoreReader.sol

pragma solidity 0.5.8;

import "./StoreInterface.sol";

contract StoreReader is StoreInterfaceId {
  function getStoreValue(address store) external view returns (uint256) {
    if (ERC165(storeAddress).supportsInterface(0x75b24222)) {
      return store.getValue();
```


Although this works, there are a few places we can improve:

1. ERC165 provides a recommended function `*doesContractImplementInterface*` for the check, which requires that the call uses less than 30000 gas when calling `*supportsInterface*`. We'd better use that in production.
2. If `*getStoreValue*` will be called multiple times for the same store address, then the check will be performed every time we call, which is unnecessary and expensive in terms of gas. We can optimize it by making a constructor function to check only once and cache a valid Store address into the contract's storage, so that the *getStoreValue* doesn't need to check that again.
3. We are repeating the hardcoded interface ID *0x75b24222* in both *StoreReader* and *Store*, we can move it into a *StoreInterfaceId* contract, and let both *Store* and StoreReader share this value by inheriting from *StoreInterfaceId* contract.

With the above improvements, we can change *StoreInterface* from an interface into an abstract contract.

```
// StoreInterface.sol
pragma solidity 0.5.8;

contract StoreInterfaceId {
  bytes4 internal constant STORE_INTERFACE_ID = 0x75b24222;
}

contract StoreInterface is StoreInterfaceId {
  function getValue() external view returns (uint256);
  function setValue(uint256 v) external;
}
```

And refactor *StoreReader* to add the *constructor* function, and use *doesContractImplementInterface* in it:

```
// StoreReader.sol

pragma solidity 0.5.8;

import "./StoreInterface.sol";
```

```
  constructor (address storeAddress) public {
    require(doesContractImplementInterface(
      storeAddress, STORE_INTERFACE_ID),
      "Doesn't support StoreInterface");

    store = StoreInterface(storeAddress);
  }

  function readStoreValue() external view returns (uint256) {
    return store.getValue();
  }
}
```

OK, great! With the above implementation, `StoreReader` is able to avoid reading values from contracts that don't support `StoreInterface`.

Calculating the Interface ID

Now let's go back to the question we had before: Why is the interface ID of *StoreInterface* *0x75b24222*?

That's actually another standard that ERC165 defines. ERC165 defines that an interface ID can be calculated as the XOR of all function selectors in the interface.

Let's say our StoreInterface has two functions: *getValue* and *setValue*:

```
// StoreInterfaceId

pragma solidity 0.5.8;

contract StoreInterfaceId {
  // StoreInterface.getValue.selector ^
  // StoreInterface.setValue.selector
  bytes4 internal constant STORE_INTERFACE_ID = 0x75b24222;
}

contract StoreInterface is StoreInterfaceId {
  function getValue() external view returns (uint256);
  function setValue(uint256 v) external;
}
```


```
pragma solidity 0.5.8;

import "./StoreInterface.sol";

contract Selector {
  // 0x75b24222
  function calcStoreInterfaceId() external pure returns (bytes4) {
    StoreInterface i;
    return i.getValue.selector ^ i.setValue.selector;
  }
}
```

Then, the *calcStoreInterfaceId* function can return the interface ID for *StoreInterface*. Running it in a test case as below, it will print the value *0x75b24222*.

Now we have a way to calculate the interface ID. However, you might still have a few questions:

1. Why using a bytes4 value as the interface ID, instead of something like a string or a bytes32 value?
2. What is a function selector?
3. Why do we use XOR to calculate the interface ID?

For question 1, a string or bytes32 value could distinguish interfaces, but the problem is they would take way more than 4 bytes to store, which is too costly. Note that an Ethereum full node needs to store all the data, including interface IDs for all interface defined. A standard needs to balance both the cost of storage and the chance of collision. The chance of collision means the chance for two different interfaces having the same interface ID. So bytes4 was chosen with the consideration of such balance.

For question 2, function selector is a bytes4 value that allows you to perform dynamic invocation of a function, based on the name of the function and the type of each one of the input arguments. You can think of it as the identifier of a function. In solidity, we can get a function selector by reading the *selector* property of a function. For example, we can make a *Selector* contract as below to print the function selector:

```
// Selector.sol
```

 

```
contract Selector {
  // 0x20965255
  function getValueSelector() external pure returns (bytes4) {
    StoreInterface i;
    return i.getValue.selector;
  }
}
```

For question 3, XOR has a nice property that any change to the function defined in the interface (i.e, function name, or arguments type) will result the calculated interface ID to be changed, and meanwhile the length of the total bytes stays unchanged, no matter how many functions are included in the interface.

Once we calculate the interface ID value, we can use it in both the *Store* contract and *StoreReader* contract. The *Store* contract uses it in the *supportsInterface* function to compare the input with the *StoreInterface's* interface ID. And the *StoreReader* contract uses it in the *readStoreValue* function to call a remote contract's *supportsInterface* function with the *StoreInterface*'s interface ID.

Summary

In this tutorial, we used two smart contracts, *Store* and *StoreReader*, as an example to explain how to make calls between contracts. The problem that ERC165 solves is that Ethereum doesn't have a built-in mechanism for a contract to know whether a given contract is an expected contract that has certain functions to call. ERC165's solution is to define a standard for contracts to publish what interfaces they support, so that other contracts can follow the same standard to detect whether it supports certain interfaces, and only call the interface's function if the interface is supported.

For more detail about ERC-165, you can also refer to the documentation here: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-165.md

As an exercise, you can implement a *StoreWriter* that saves the value to a Store contract.

The complete source code and test cases for this tutorial can be found on GitHub here.