4. 솔리디티의 프록시 패턴(Proxy Pattern)에 대해서 설명하고, 이에 대한 활용 예를 적어주세요.

1) Proxy Pattern 소개

한 번 배포된 컨트랙트 코드는 수정할 수 없기 때문에 컨트랙트의 기능을 업그레이드하기 위해서는 새로운 컨트랙트를 작성하여 배포해야 한다. 이 때 `Proxy Pattern` 을 사용하면 기존에 해당 컨트랙트를 사용하던 클라이언트 쪽에서 어떠한 변경도 하지 않고 `logic` 컨트랙트를 업그레이드하여 기능을 변경할 수 있다. `proxy` 컨트랙트는 `logic` 컨트랙트의 주소를 참조하여 `delegatecall()` 을 호출하여 `logic` 컨트랙트에서 코드를 실행할 수 있다. 만약 새로운 버전의 `logic` 컨트랙트가 배포되면 `proxy` 컨트랙트에서 참조하던 기존의 `logic` 컨트랙트 주소를 새로운 버전의 `logic` 컨트랙트 주소로 변경한다. 그러면 사용자는 기존의 동일한 `proxy` 컨트랙트를 호출하면서도 새로운 기능의 `logic` 컨트랙트를 사용할 수 있게 된다.

`Zeppelin` 에서는 `logic` 컨트랙트의 업그레이드가 가능한 3가지 Proxy Patterns(`Inherited Storage` , `Unstructured Storage` , `Eternal Storage`)을 제안하고 있다. 3가지 패턴이 해결하고자 하는 문제는 모두 어떻게 하면 `logic` 컨트랙트가 `upgradeability` 를 위해 사용하는 중요한 `state variables` (e.g. 가장 최신 logic 컨트랙트의 주소)를 변경하지 않을 수 있을까하는 것이다.

`Inherited Storage` 패턴은 `proxy` 에게 필요한 `storage structure` 를 `logic` 컨트랙트가 상속하는 것이다. 즉 `proxy` 컨트랙트와 `logic` 컨트랙트가 동일한 `storage structure` 를 공유하며 동일한 필수 `state variables` 를 서로 갖는 것이다.

`Eternal Storage` 패턴에서는 `proxy` 와 `logic` 컨트랙트가 함께 상속하는 별도의 컨트랙트에 `storage schema` 를 정의하는 방식이다. 그리고 이 `storage` 컨트랙트에는 `logic` 컨트랙트가 필요한 모든 `state variables` 가 정의되어 있으며 `proxy` 와 공유한다.

`Unstructured Storage` 패턴에서는 `logic` 컨트랙트가 업그레이드에 필요한 어떠한 `state variables` 도 상속할 필요가 없다. 대신에 `proxy` 컨트랙트 안에 정의된 `unstructured storage slot` 을 사용한다. 이 때 충분히 랜덤한 해쉬를 값을 `constant` 로 선언하여 `logic` 컨트랙트의 주소를 저장하는 `slot` 으로 사용한다. 이렇게 하면 `constant` 는 `state variable` 이 아니므로 상태가 변경되는 문제가 발생하지 않는다.

2) Proxy Pattern 컨트랙트 예시

`proxy` 컨트랙트를 이용한 토큰을 관리하는 `logic` 컨트랙트 예시를 작성해보도록 할 것이다. 이 때 우리는 `Eternal Storage` 방식을 사용하여 `proxy` 와 `logic` 컨트랙트가 함께 상속하는 `storage structure` 컨트랙트를 활용할 것이다.

(1) StorageStructure 컨트랙트

`StorageStructure` 컨트랙트는 `proxy` 와 `logic` 컨트랙트가 동시에 알아야 하는 중요한 `state variables` 를 갖고 있다. `StorageStructure` 컨트랙트 코드는 다음과 같다.

StorageStructure.sol

```
pragma solidity ^0.4.18;

contract StorageStructure {
  address public implementation;
  address public owner;
  bool public isInit = false;
  mapping (address => uint) internal balances;
  uint internal totalSupply;
}
```

`implementation` 변수에는 구현된 `lgoci` 컨트랙트의 주소가 저장되며 나머지는 기본적인 토큰 관리를 위한 변수들이다.

(2) Logic 컨트랙트

이번 예제에서는 `logic` 컨트랙트로서 간단한 토큰 컨트랙트를 사용할 것이다. `TokenLogic` 컨트랙트에는 토큰을 초기 설정는 `init()` 함수와 사용자 간에 토큰을 주고받을 수 있는 `transfer()` 함수가 있다. 이 함수들은 나중에 `proxy` 를 통해 `delegatecall` 로 호출될 것이다.

TokenLogic.sol

```
pragma solidity ^0.4.18;

import "./StorageStructure.sol";

contract TokenLogic is StorageStructure {
  function init(uint _initialSupply) public {
    require(msg.sender == owner);
    require(isInit == false);
    isInit = true;
    balances[msg.sender] = totalSupply = _initialSupply;
  }

  function transfer(address _to, uint _value) public returns (bool) {
    require(balances[msg.sender] >= _value);
    balances[msg.sender] = balances[msg.sender] - _value;
    balances[_to] = balances[_to] + _value;
    return true;
  }

  function balanceOf(address _owner) public constant returns (uint balance) {
    return balances[_owner];
  }
}
```

(3) Proxy 컨트랙트

`proxy` 컨트랙트는 `logic` 컨트랙트와 동일하게 `StorageStructure`를 상속하며, 구현된 `logic` 컨트랙트의 주소를 저장하고 변경할 수 있는 기능이 있다.

Proxy.sol

```
pragma solidity ^0.4.18;

import "./StorageStructure.sol";

contract Proxy is StorageStructure {
  function Proxy() public {
    owner = msg.sender;
  }

  function upgradeTo(address _newImplementation) external {
    require(msg.sender == owner);
    require(implementation != _newImplementation);
    _setImplementation(_newImplementation);
  }

  function () payable public {
    address impl = implementation;
    require(impl != address(0));
    assembly {
      let ptr := mload(0x40)
      calldatacopy(ptr, 0, calldatasize)
      let result := delegatecall(gas, impl, ptr, calldatasize, 0, 0)
      let size := returndatasize
      returndatacopy(ptr, 0, size)

      switch result
      case 0 { revert(ptr, size) }
      default { return(ptr, size) }
    }
  }

  function _setImplementation(address _newImp) internal {
    implementation = _newImp;
  }
}
```

`proxy` 컨트랙트에는 `upgradeTo()` 함수가 있는데 이 함수를 통해 배포된 `TokenLogic` 컨트랙트의 주소를 저장한다. 이 함수를 이용하면 나중에 새로운 버전의 `logic` 컨트랙트가 배포되었을 때 손쉽게 업그레이드를 수행할 수 있다.

```
function upgradeTo(address _newImplementation) external {
  require(msg.sender == owner);
  require(implementation != _newImplementation);
  _setImplementation(_newImplementation);
}
```

그리고 `proxy` 컨트랙트에서 핵심 역할을 하는 `fallback` 함수 안에 `delegatecall` 을 호출하는 `assembly` 코드는 다음과 같다.

```
assembly {
  let ptr := mload(0x40)
  calldatacopy(ptr, 0, calldatasize)
  let result := delegatecall(gas, impl, ptr, calldatasize, 0, 0)
  let size := returndatasize
  returndatacopy(ptr, 0, size)

  switch result
  case 0 { revert(ptr, size) }
  default { return(ptr, size) }
}
```

`ptr` 변수가 가리키는 `memory slot` 의 `0x40` 에는 사용가능한 `memory` 주소의 `pointer` 를 담고 있다. 따라서 `ptr` 가 가리키는 현재 사용가능한 `memory` 위치에 전송받은 `calldata` 의 크기 만큼 인데스 0부터 값을 복사한다.

```
let ptr := mload(0x40)
calldatacopy(ptr, 0, calldatasize)
```

그 다음 외부 컨트랙트에게 `delegatecall` opcode를 사용하여 외부 호출을 한다. 이 때 전달되는 파라미터들은 다음과 같다.

```
let result := delegatecall(gas, impl, ptr, calldatasize, 0, 0)
```

- `gas` : 함수를 실행하기 위해 필요한 `gas`
- `impl` : 호출할 외부의 `logic` 컨트랙트 주소
- `ptr` : 전달된 `data` 가 저장된 `pointer`
- `0` : 출력 데이터. 어떤 값이 출려될지 결정되지 않았기 때문에 사용하지 않음. 대신 나중에 `returndata` 사용
- `0` : 출력 데이터의 크기. 마찬가지로 사용하지 않음. 대신 나중에 `returndatasize` 를 사용

다음 줄에서는 `returndatasize` opcode를 사용하여 리턴된 데이터의 크기를 저장한다.

```
let size := returndatasize
```

그리고 전달된 데이터 크기 만큼의 데이터를 `ptr` 이 가리키는 위치에 저장한다.

```
returndatacopy(ptr, 0, size)
```

마지막으로 switch 구문은 전달된 데이터를 리턴하거나 또는 오류가 있을 경우 `revert()` 한다.

이제 모든 컨트랙트 작성이 완료되었으며, 다음과 같이 `proxy` 컨트랙트를 사용할 수 있다.

(Stores address of the logic contract)

1. `TokenLogic` 컨트랙트와 `Proxy` 컨트랙트를 생성한다.
2. `upgradeTo()` 함수를 호출하여 생성된 `TokenLogic` 컨트랙트의 주소를 저장한다.
3. `delegatecall` 로 `init()` 함수를 호출하여 토큰을 생성한다.
4. `transfer()` 함수를 호출하여 다른 주소로 토큰을 전송한다.
5. 만약 `logic` 컨트랙트를 업그레이드 할 경우 새로 컨트랙트를 생성하고 2번 과정을 반복한다.

References

[1] *OpenZeppelin. (2018, Apr 19).* *Proxy Patterns* *[Web Blog Post]*