Bridging L1 and L2 (Standard Bridge)

Titan에서는 L1과 L2간의 자산 이동을 위해 Standard Bridge를 지원합니다. 사용자는 L2에서 사용하려는 ETH, ERC20 토큰을 Standard Brdige 컨트랙트를 통해 L2로 보낼 수 있습니다. (deposit) 반대로, L2에서 사용하던 ETH, ERC20 토큰을 L1으로 출금하는 것 또한 Standard Bridge로 할 수 있습니다. (withdraw)
첫 번째로 deposit은 자산을 L1에서 L2으로 이동하는 것을 의미합니다. ETH의 deposit은 `L1StandardBridge` 컨트랙트의 `depositETH`, `depositETHTo` 함수를 통해 할 수 있습니다. ERC20 토큰의 deposit은 `L1StandardBridge` 컨트랙트의 `depositERC20`, `depositERC20To` 함수를 통해 할 수 있습니다. 자산의 deposit은 deposit 트랜잭션이 마이닝되고 몇 분 가량이 지난 이후 L2에서 처리됩니다.
두 번째로 withdraw는 L2에서 L1으로 자산을 이동하는 것을 뜻합니다. L2에서 ETH와 ERC20 토큰의 withdrawal은 `L2StandardBridge` 컨트랙트의 `withdraw`, `withdrawTo` 함수를 통해 할 수 있습니다. L1의 경우와 다르게 L2에서의 출금은 ETH와 ERC20별로 나누어지지 않는데, L2에서 ETH는 Native coin이 아니라 토큰으로 처리되기 때문입니다. Titan에서 자산의 출금은 1주일의 기간이 소요됩니다. 이것은 Optimistic Rollup의 특성상 L2의 트랜잭션이 검증되기 위해 필요한 기간입니다.
본 섹션에서는 Titan Contacts 패키지를 import하여 ETH를 deposit/withdraw하는 방법을 Javascript 코드 예제를 통해 알아보겠습니다.
예제 코드는 여기에서 확인하실 수 있습니다. Titan SDK를 사용하여 L1-L2 간에 ETH를 전송하는 방법을 배우기 위해서는 직접 코드를 실행해 보는 것이 좋습니다.

Setup

- Ethereum 블록체인과 상호작용하기 위한 기능을 지원하는 `ethers` 라이브러리를 가져옵니다. `@tokamak-network/titan-contracts` 패키지를 import하여 사전 배포 및 L1StandardBridge와 L2StandardBridge의 ABI와 바이트코드를 가져옵니다.

```
import { ethers } from "ethers";
import { predeploys } from "@tokamak-network/titan-contracts";

import l1StandardBridgeArtifact from "@tokamak-network/titan-contracts/artifacts/contra
import l2StandardBridgeArtifact from "@tokamak-network/titan-contracts/artifacts/contra
```

- `@tokamak-network/titan-contracts` 패키지를 통해 가져온 L1StandardBridge와 L2StandardBridge의 아티팩트를 파라미터로 각 컨트랙트의 팩토리 인스턴스를 생성합니다.

```
const factory__L1StandardBridge = new ethers.ContractFactory(
  l1StandardBridgeArtifact.abi,
  l1StandardBridgeArtifact.bytecode
)
const factory__L2StandardBridge = new ethers.ContractFactory(
  l2StandardBridgeArtifact.abi,
  l2StandardBridgeArtifact.bytecode
)
```

- `ethers` 라이브러리를 이용하여 L1과 L2의 provider 객체와 wallet 객체를 생성합니다. 그리고 다음과 같이 L1StandardBridge와 L2StandardBridge 인스턴스를 만듭니다.

1. `L2StandardBridge` 인스턴스 생성:
   - `factory__L2StandardBridge` 를 사용하여 컨트랙트 팩토리 인스턴스의 메서드를 활용
   - `l2Wallet` 를 컨트랙트에 연결
   - `predeploys.L2StandardBridge` 를 컨트랙트 주소로 사용하여 `attach` 함수를 호출하여 `L2StandardBridge` 인스턴스를 생성

2. `L1StandardBridge` 인스턴스 생성:
   - `factory__L1StandardBridge` 를 사용하여 컨트랙트 팩토리 인스턴스의 메서드를 활용
   - `L2StandardBridge` 의 `l1TokenBridge()` 함수를 호출하여 L1 Standard Bridge 컨트랙트 주소인 `L1StandardBridgeAddress` 를 가져옴
   - `l1Wallet` 를 컨트랙트에 연결
   - `L1StandardBridgeAddress` 를 컨트랙트 주소로 사용하여 `attach` 함수를 호출하여 `L1StandardBridge` 인스턴스를 생성

```
const l1RpcProvider = new ethers.providers.JsonRpcProvider(l1Url)
const l2RpcProvider = new ethers.providers.JsonRpcProvider(l2Url)

const l1Wallet = new ethers.Wallet(key, l1RpcProvider)
const l2Wallet = new ethers.Wallet(key, l2RpcProvider)

const L2StandardBridge = factory__L2StandardBridge
    .connect(l2Wallet)
    .attach(predeploys.L2StandardBridge)
const L1StandardBridgeAddress = await L2StandardBridge.l1TokenBridge()
const L1StandardBridge = factory__L1StandardBridge
  .connect(l1Wallet)
  .attach(L1StandardBridgeAddress)
```

Deposit

- Setup 단계에서 생성한 `L1StandardBridge` 컨트랙트의 `depositETH` 함수를 호출하여 L1에서 L2로 원하는 양의 ETH를 이동시킬 수 있습니다. (L1에는 deposit하는 수량 이상의 ETH + 트랜잭션 전송을 위한 적정 가스비가 준비되어야 합니다.)
- depositETH 함수의 첫 번째 매개변수는 L2 트랜잭션에 사용될 가스량, 두 번째 매개변수는 추가적인 데이터로, 배열 형태로 전달됩니다. 예제에서는 빈 배열을 사용합니다. 세 번째 매개변수는 `value` 속성을 통해 예치할 ETH의 양을 설정합니다. `ethers.utils.parseEther(balance)` 를 통해 `balance` 값을 이더 단위로 변환하여 설정합니다.
- deposit 트랜잭션이 L1에서 처리될 때까지 대기하고, 트랜잭션의 수신 정보를 나타내는 `receipt` 객체를 얻습니다. 트랜잭션의 `receipt` 객체의 `status` 속성을 확인하여 트랜잭션이 성공적으로 처리되었는지 확인합니다. `status` 값이 1이 아닐 경우, 즉 실패한 경우 `Error` 객체를 throw합니다.
- L1에서 요청한 deposit 트랜잭션은 `L2StandardBridge` 를 목적지로 하는 message를 생성하여 L2로 전달합니다. `L1CrossDomainMessenger` , `L2CrossDomainMessenger` 를 거쳐 전달된 message는 `L2StandardBridge` 의 `finalizeDeposit` 함수를 호출하여 L2로 deposit한 양만큼의 ETH를 전송합니다.

```
const tx = await L1StandardBridge.depositETH(
  200000, // Gas for L2 transaction
  [],
  {
    value: ethers.utils.parseEther(balance),
  }
)
console.log(`TX Hash: ${tx.hash}`)

const receipt = await tx.wait()
if (receipt.status !== 1) {
```

```
  throw(new Error('transaction is failed'));
}
```

Withdraw

- `L2StandardBridge` 의 `withdraw` 함수를 호출하여 L2에서 L1으로 원하는 수량의 ETH를 브릿징할 수 있습니다. (L2에는 withdraw하는 수량 이상의 ETH + 트랜잭션 전송을 위한 적정 가스비가 준비되어야 합니다.)
- 함수의 첫 번째 매개변수는 인출할 자산을 지정하는 토큰 주소를 지정합니다. 예제에서는 `predeploys.OVM_ETH` 를 사용하여 토큰으로 처리할 수 있도록 OVM_ETH를 사용합니다. 두 번째 매개변수는 인출할 ETH의 양을 설정합니다.
- `ethers.utils.parseEther(balance)` 를 통해 `balance` 값을 이더 단위로 변환하여 설정합니다. 세 번째 매개변수는 토큰의 인출 소유권을 나타내는 값입니다. 예제에서는 0을 사용합니다. 네 번째 매개변수는 인출에 대한 추가적인 데이터로, 문자열 형태로 전달됩니다. 예제에서는 ' `0xFFFF` '를 사용합니다.
- withdraw 트랜잭션이 L2에서 처리될 때까지 대기하고 트랜잭션의 수신 정보를 나타내는 `receipt` 객체를 얻습니다. 트랜잭션의 `receipt` 객체의 `status` 속성을 확인하여 트랜잭션이 성공적으로 처리되었는지 확인합니다.
- L2에서 전송한 withdraw 트랜잭션은 `L1StandardBridge` 를 목적지로 하는 message를 생성하여 L1으로 전달합니다. 옵티미스틱 롤업에서 L2에서 생성된 트랜잭션과 State Root는 L1으로 롤업됩니다. L1에 성공적으로 State Root가 롤업되었는지 확인이 되면 message relayer 서비스가 `L1CrossDomainMessenger` 를 호출하여 L1으로 message를 전달합니다. Titan에서는 여러 개의 메시지를 하나의 트랜잭션에 포함하여 전달하는 배치 릴레이를 지원하여 트랜잭션 수수료를 낮추고 릴레이 속도를 높였습니다. 그 다음 `L1StandardBridge` 의 `finalizeETHWithdrawal` 함수를 호출하여 L1에 ETH가 전송되면 withdraw가 완료됩니다.

```
const tx = await L2StandardBridge.withdraw(
  predeploys.OVM_ETH,
  ethers.utils.parseEther(balance),
  0,
  '0xFFFF'
)

console.log(`TX Hash: ${tx.hash}`)

const receipt = await tx.wait()
if (receipt.status !== 1) {
  throw(new Error('transaction is failed'));
}
```