前言

合约逻辑

合约结构

两阶段提交核心逻辑

错误处理逻辑

通用参数校验

事件机制

总结

参考资料

通过状态锁在 Solidity 智能合约中实现两阶段提交

前言

在一些牵扯到多个系统或合约交互的智能合约应用场景中，尤其是一些资产/数据准确性较为敏感的业务中，我们需要保证在整个业务流程中数据的原子性。因此，我们需要在合约层面实现类似多阶段提交的机制，即将合约中的状态更改过程分解为预提交和正式提交两个阶段。

本文通过状态锁的机制实现了一个最小化的两阶段提交模型，完整合约代码参见 TwoPhaseCommit.sol，下文将对本合约核心逻辑进行讲解，并尽量遵循风格指南与最佳实践。

> 注：本合约因初始场景主要考虑的是联盟链中的业务用途，未对 Gas fee 等进行特定优化，仅供学习参考。

合约逻辑

合约结构

两阶段提交场景包含以下方法：

1. set: 两阶段 - 预提交
2. commit: 两阶段 - 正式提交
3. rollback: 两阶段 - 回滚

因 Solidity 语言对于字符串长度判断/比较等有一些限制，为了提升合约代码的可读性，本合约提供了部分辅助方法，主要包含以下方法：

1. isValidKey: 检查 key 是否合法
2. isValidValue: 检查 value 是否合法
3. isEqualString: 比较两个字符串是否相等

两阶段提交核心逻辑

在两阶段提交场景中，本合约提供了一套简易的 `set`，`commit`，`rollback` 方法实现，实现了将合约调用传入的 key-value 键值对存储到链上。我们通过状态锁的机制来实现跨链交易的原子性。我们定义了如下数据结构：

```
enum State {
    UNLOCKED,
    LOCKED
}

struct Payload {
    State state;
    string value;
    string lockValue;
}
```

其中， `State` 为枚举类型，记录了链上 key 值的锁定状态，而 `Payload` 结构则会对锁定状态、当前值与正在锁定的值进行存储，并通过如下 `mapping` 结构与 key 进行绑定：

```
mapping (string => Payload) keyToPayload;
```

因此，我们可以根据 `keyToPaylaod` 对合约调用中的每一个 key 进行状态跟踪，并在下述 `cc_set`，`cc_commit`，`cc_rollback` 方法中对 key 的状态进行检查，进行一些异常处理。

cc_set()

在 `cc_set()` 方法中，我们会检查 key 的状态，如为 `State.LOCKED`，则不会进行存储并抛出异常：

```
if (keyToPayload[_key].state == State.LOCKED) {
    revert TwoPhaseCommit__DataIsLocked();
}
```

如为 `State.UNLOCKED`，则会将合约调用传入的值存储至 lockValue 中，并将其状态设置为 `LOCKED`，等待后续 `cc_commit` 或 `cc_rollback` 进行解锁。

```
keyToPayload[_key].state = State.LOCKED;
keyToPayload[_key].lockValue = _value;
```

cc_commit()

在 `cc_commit()` 方法中，我们会检查 key 的状态，如为 `State.UNLOCKED`，则不会对该 key 进行操作，并抛出异常：

```
if (keyToPayload[_key].state == State.UNLOCKED) {
    revert TwoPhaseCommit__DataIsNotLocked();
}
```

如为 `State.LOCKED`，我们检查合约调用传入的值是否与 lockValue 相等，如不相等，则抛出异常：

```
if (!isEqualString(keyToPayload[_key].lockValue, _value)) {
    revert TwoPhaseCommit__DataIsInconsistent();
}
```

如值相等，则会将该 key 所对应的 value 存储上链，将 key 的状态设置为 `UNLOCKED`，更新当前值 `value`，同时将 `lockValue` 置空：

```
store[_key] = _value;
keyToPayload[_key].state = State.UNLOCKED;
keyToPayload[_key].value = _value;
keyToPayload[_key].lockValue = "";
```

cc_rollback()

在 `cc_rollback()` 方法中，我们会检查 key 的状态，如为 `State.UNLOCKED`，则不会对该 key 进行操作，并抛出异常：

```
if (keyToPayload[_key].state == State.UNLOCKED) {
    revert TwoPhaseCommit__DataIsNotLocked();
}
```

如为 `State.LOCKED`，我们检查合约调用传入的值是否与 lockValue 相等，如不相等，则抛出异常：

```
if (!isEqualString(keyToPayload[_key].lockValue, _value)) {
    revert TwoPhaseCommit__DataIsInconsistent();
}
```

如值相等，则会将该 key 所对应的 value 存储上链，将 key 的状态设置为 `UNLOCKED`，并将 `lockValue` 置空：

```
keyToPayload[_key].state = State.UNLOCKED;
keyToPayload[_key].lockValue = "";
```

错误处理逻辑

在合约执行异常场景中，我们会抛出错误并进行回滚。为了更好地提升错误消息的可读性并方便上层应用人员进行错误捕获与处理，我们采用了错误类型定义的方式，定义了各类异常场景，因为我在错误命名中已经包含了大部分信息，所以未定义错误类型额外参数值，可以根据需求自行定制。

```
error TwoPhaseCommit__DataKeyIsNull();
error TwoPhaseCommit__DataValueIsNull();
error TwoPhaseCommit__DataIsNotExist();
error TwoPhaseCommit__DataIsLocked();
error TwoPhaseCommit__DataIsNotLocked();
error TwoPhaseCommit__DataIsInconsistent();
```

在具体合约逻辑中，我们通过 `revert` 方法抛出异常，如：

```
if (!isValidKey(bytes(_key))) {
    revert TwoPhaseCommit__DataKeyIsNull();
}

if (!isValidValue(bytes(_value))) {
    revert TwoPhaseCommit__DataValueIsNull();
}

if (keyToPayload[_key].state == State.UNLOCKED) {
    revert TwoPhaseCommit__DataIsNotLocked();
}

if (!isEqualString(keyToPayload[_key].lockValue, _value)) {
    revert TwoPhaseCommit__DataIsInconsistent();
}
```

通用参数校验

我们会对传入参数进行一些合法性校验，为了提供拓展性，我们通过 `isValidKey()` 与 `isValidValue()` 方法对 key 与 value 进行独立校验：

```
/**
 * @notice 数据键格式校验
 * @param _key 数据 - 键
 */
function isValidKey(bytes memory _key) private pure returns (bool)
{
    bytes memory key = _key;

    if (key.length == 0) {
        return false;
    }
    return true;
}

/**
 * @notice 数据值格式校验
 * @param _value 数据 - 值
 */
function isValidValue(bytes memory _value) private pure returns (bool)
{
    bytes memory value = _value;

    if (value.length == 0) {
        return false;
    }
    return true;
}
```

本合约只进行了非空校验，可根据业务需要自行定制业务逻辑，在需要校验的地方调用即可，如：

```
if (!isValidKey(bytes(_key))) {
    revert TwoPhaseCommit__DataKeyIsNull();
}

if (!isValidValue(bytes(_value))) {
    revert TwoPhaseCommit__DataValueIsNull();
}

if (!isValidValue(bytes(store[_key]))) {
```

```
        revert TwoPhaseCommit__DataIsNotExist();
    }
```

事件机制

此外，我们定义了核心方法对应的 event，并为事件设置了 indexed 以方便上层应用进行监听和处理。

```
event setEvent(string indexed key, string indexed value);
event getEvent(string indexed key, string indexed value);
event commitEvent(string indexed key, string indexed value);
event rollbackEvent(string indexed key, string indexed value);
```

在合约方法中通过 `emit()` 方法抛出 event，如：

```
emit setEvent(_key, _value);
emit getEvent(_key, _value);
emit commitEvent(_key, _value);
emit rollbackEvent(_key, _value);
```

总结

以上就是我两阶段提交合约的一个最佳实践，关于 Solidity 基础语法可参看『Solidity 智能合约开发 - 基础』，后续我还会对更多合约场景进行实践与讲解，敬请关注。

参考资料

1. TwoPhaseCommit.sol 合约源码
2. Solidity 智能合约开发 - 基础
3. Solidity 官方文档