实验二报告

NFT 智能合约与所有权经济

Lab 2: NFT Smart Contract & Ownership Economy

姓名：范舒舰

学号：23121677

专业：网络空间安全

2026 年 4 月 3 日

目录

1 实验目标

1. **掌握 ERC-721 标准**
   理解非同质化代币的行业标准。

2. **理解数字稀缺性**
   探索代码如何强制执行唯一性和限量供应。

3. **进阶数据映射**
   深化对 mapping 追踪资产所有权的理解。

4. **IPFS 与元数据**
   学习区块链如何链接链外资产。

2 实验背景与实验要求

2.1 实验背景

在实验一中，你为可替代的 ETH 建立了一个透明账本。然而，某些资产是唯一的（如艺术品）。NFT 允许我们在链上表示这些唯一资产。本次实验你将创建一个“数字画廊”。

2.2 实验要求

2.2.1 基础功能（必做）

使用 `OpenZeppelin` 库编写名为 `SimpleNFT` 的合约。

2.2.1.1. **铸造机制**
实现 `mint()` 函数，允许外部账户创建新的 NFT。

2.2.1.2. **所有权追踪**
每个 NFT 必须有唯一 `ID` ，并使用计数器累加。

2.2.1.3. **元数据链接**
为每个代币分配 `URI` 以指向其图像。

2.2.2 进阶功能（选做）

2.2.2.1. **铸造费**
每次铸造收取少量 `ETH` 作为手续费。

2.2.2.2. **总量限制**
设置总量限制，达到后停止铸造。

2.2.2.3. **版权费**

实现版税逻辑，使未来的交易分成回流给创作者。

3 开发环境说明

本实验使用浏览器中的 Remix IDE 作为智能合约开发与测试环境，利用 OpenZeppelin 官方库实现 ERC-721 NFT 合约，并在 Remix VM 提供的虚拟链环境中完成部署和交互测试。

本实验开发环境如下：

- 开发工具：Remix IDE
- 编程语言：Solidity
- 依赖库：OpenZeppelin Contracts
- 编译器版本：latest local version - soljson-v0.8.34+commit.80d5c536.js
- 部署环境：Remix VM（Osaka）

4 理论基础

4.1 ERC-721 与 NFT 唯一性

ERC-721 是以太坊生态中用于表示非同质化代币的标准接口。与 ERC-20 不同，ERC-721 中的每个代币都具有唯一的 `tokenId`，因此更适合表示艺术品、数字藏品、证书等具有唯一性的链上资产。

在本实验中，NFT 的唯一性由递增计数器 `nextTokenId` 实现。每调用一次 `mint()`，合约便取出当前计数值作为新的 `tokenId`，并在铸造完成后将计数器加一，从而确保后续铸造得到不同编号。

4.2 所有权追踪与元数据绑定

ERC-721 标准内部维护了 `tokenId` 到所有者地址的映射关系，因此用户可通过 `ownerOf(tokenId)` 查询某个 NFT 当前属于哪个地址。这一机制是 NFT 所有权表达的核心。

同时，NFT 本身通常不直接将图片内容存储在链上，而是通过 `tokenURI(tokenId)` 返回一个指向链外元数据文件的 URI。该元数据文件中可以进一步描述作品名称、图片链接、属性等内容。本实验中通过 `_setTokenURI(tokenId, uri)` 为每个 NFT 建立链上到链外资源的连接。

4.3 铸造费与合约余额

当 `mint()` 被声明为 `payable` 后，调用者即可在铸造 NFT 的同时向合约发送 ETH。本实验在进阶部分规定调用者必须支付 `0.01 ether` (设定) 作为铸造费。发送的 ETH 会累积到合约余额中，之后由合约拥有者通过 `withdraw()` 提取。

4.4 ERC-2981 与版税机制

ERC-2981 是 NFT 版税标准，用于声明某个 NFT 在后续二级市场交易中应向哪个地址支付多少比例的版税。该标准并不直接强制完成付款，而是通过 `royaltyInfo(tokenId, salePrice)` 返回接收者地址与版税金额，从而使支持该标准的市场能够读取并执行相应规则。

5 合约设计说明

5.1 基础版合约主要状态变量

基础版 `SimpleNFT` 合约实现 NFT 铸造、唯一编号和 URI 绑定，其主要状态变量如下：

表 1: 基础版主要状态变量说明

变量名	作用说明
`nextTokenId`	记录下一个将要分配的 NFT 编号

5.2 最终版合约主要状态变量

在进阶部分中，合约被扩展为支持铸造费、总量限制和版税机制，其主要状态变量如下：

表 2: 最终版主要状态变量说明

变量名	作用说明
`owner`	保存合约部署者地址，用于提款权限控制与默认版税接收
`nextTokenId`	记录下一个将要分配的 NFT 编号
`MINT_FEE`	每次铸造所需支付的固定费用，设为 `0.01 ether`
`MAX_SUPPLY`	NFT 最大供应量，设为 100

5.3 合约功能说明

1. **基础版铸造功能**
   基础版通过 `mint(string memory uri)` 为调用者铸造一个新的 NFT，并使用 `_safeMint()` 与 `_setTokenURI()` 完成所有权登记和元数据绑定。

2. **唯一编号管理**
   通过 `nextTokenId` 递增分配 NFT 编号，每次铸造成功后执行 `nextTokenId++`。

3. **铸造费机制**
   最终版将 `mint()` 声明为 `payable`，并要求调用者发送的 ETH 必须等于 `MINT_FEE`。

4. **总量限制机制**
   最终版通过 `require(nextTokenId < MAX_SUPPLY, "Max supply reached")` 在代码层面限制 NFT 的总铸造数量。

5. **版税机制**
   最终版额外继承 `ERC2981`，并在构造函数中通过 `_setDefaultRoyalty(msg.sender, 500)` 将默认版税比例设置为 5%。

6. **提款功能**
   合约部署者可通过 `withdraw()` 提取合约中累积的铸造费用，非部署者调用时会被回滚。

6 实验步骤（基础部分）

6.1 步骤一：编写基础功能代码

首先在 Remix IDE 中新建 `SimpleNFT.sol` 文件，导入 OpenZeppelin 提供的 `ERC721URIStorage` 扩展合约，并完成基础版 `SimpleNFT` 的实现。基础版合约需要支持以下最小功能：通过 `nextTokenId` 实现递增编号；通过 `mint()` 铸造 NFT；通过 `_setTokenURI()` 为 NFT 绑定 URI；并通过 `ownerOf(tokenId)` 验证所有权归属。

6.2 步骤二：编译基础功能代码

进入 `Solidity Compiler` 面板，选择编译器版本 `soljson-v0.8.34+commit.80d5c536.js`，对基础版 `SimpleNFT.sol` 进行编译。编译成功后，Remix 显示绿色对勾，表明代码语法正确并可部署。

6.3 步骤三：部署基础版合约并完成首次铸造

进入 `Deploy & Run Transactions` 面板，环境设置为 `Remix VM (Osaka)`，使用第一个测试账户部署基础版合约。部署完成后，在 `mint` 输入框中传入 URI：`https:`

`//example.com/nft/1.json`，随后调用 `mint()`。由图1可见，交易成功执行，说明基础版合约已能够完成 NFT 的首次铸造。

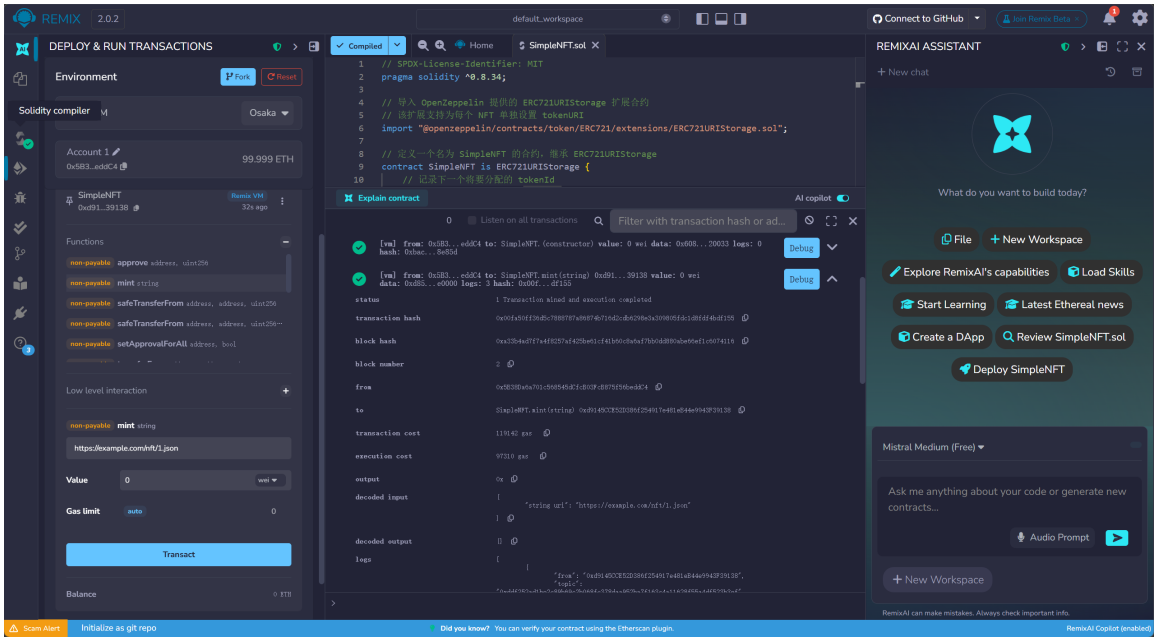

图 1: 基础版合约部署成功并完成首次铸造

6.4 步骤四：测试基础功能

1. **验证编号计数器是否递增**

   调用 `nextTokenId()`，返回值为 1，说明第一次 NFT 铸造完成后，计数器已由初始值 0 正确增加至 1，从而保证后续 NFT 可获得新的唯一编号，如图2所示。

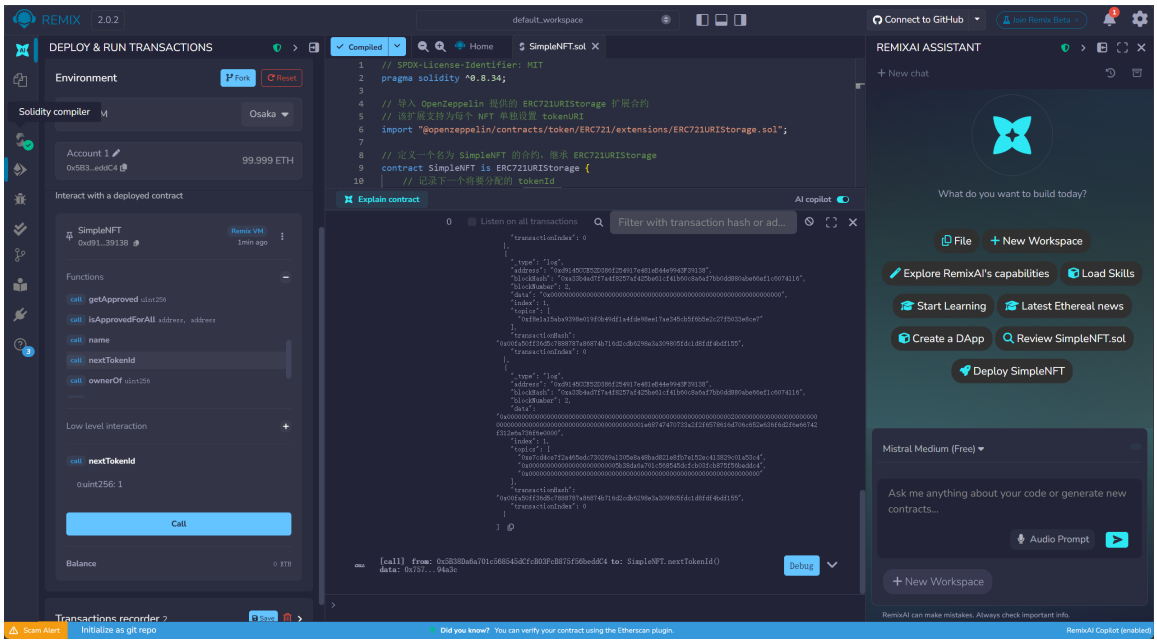

图 2: 基础版调用 `nextTokenId()` 返回 1

2. **验证 NFT 所有权归属**

   调用 `ownerOf(0)`，返回地址为 `0x5B38Da6a701c568545dCfcB03FcB875f56beddC4`，与部署并执行铸造操作的账户一致，说明 `tokenId = 0` 的 NFT 已被正确登记到当前调用者名下，如图3所示。

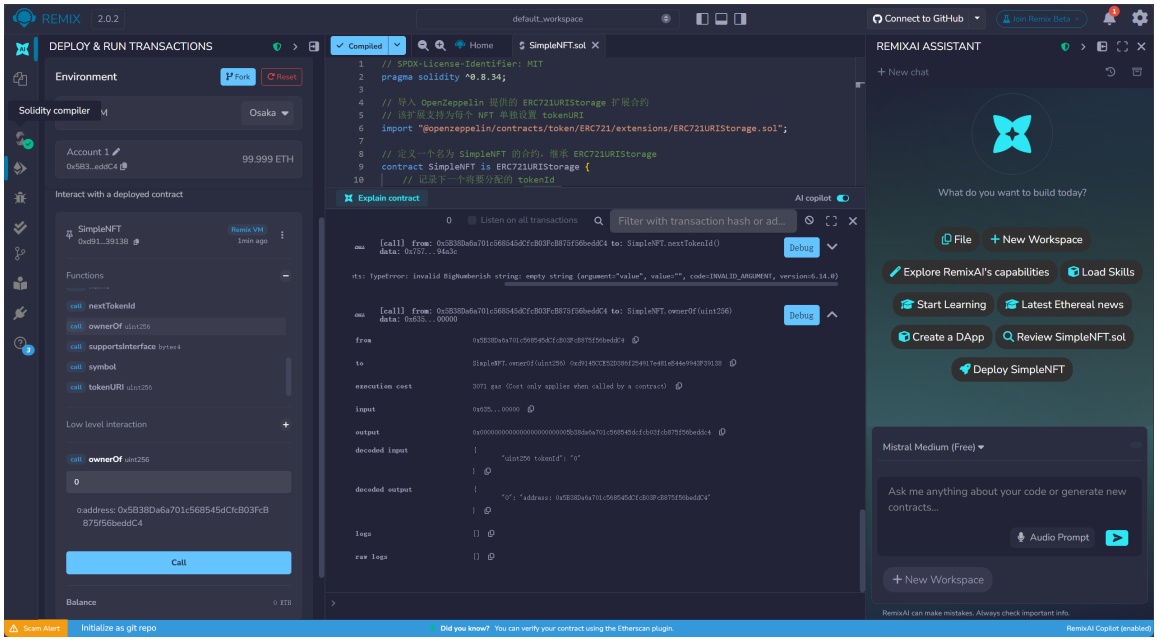

图 3: 基础版调用 `ownerOf(0)` 验证所有权归属

3. **验证元数据 URI 绑定**

   调用 `tokenURI(0)`，返回值为 `https://example.com/nft/1.json`，与首次铸造时输入的 URI 一致，说明该 NFT 的元数据链接已被正确写入合约状态中，如图4所示。

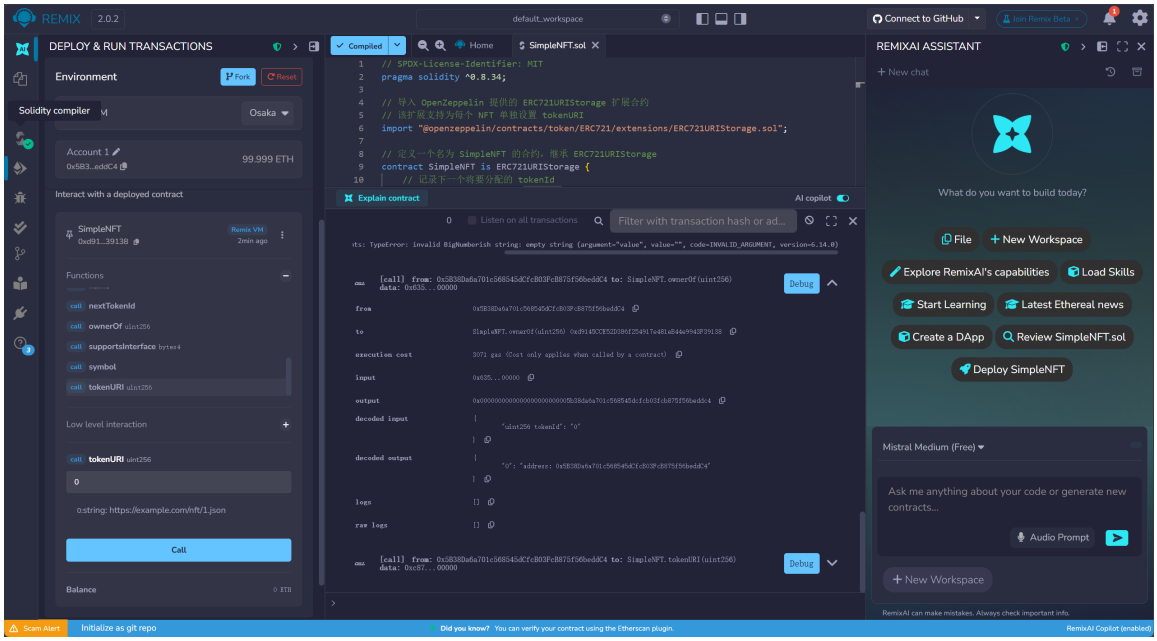

图 4: 基础版调用 `tokenURI(0)` 验证 URI 绑定

6.5 基础部分实验现象记录

根据实验截图，基础功能测试结果如下：

- 首次铸造输入的 URI：`https://example.com/nft/1.json`
- 第一次铸造成功后对应的 `tokenId`：`0`
- 调用 `nextTokenId()` 的返回值：`1`
- 调用 `ownerOf(0)` 的返回地址：`0x5B38Da6a701c568545dCfcB03FcB875f56beddC4`
- 调用 `tokenURI(0)` 的返回值：`https://example.com/nft/1.json`

7 实验步骤（进阶部分）

7.1 步骤一：扩展为最终版本合约

在基础版验证完成后，继续在 `SimpleNFT` 合约中加入铸造费、总量限制、版税与提款权限控制，形成最终版本合约。最终版额外引入 `ERC2981`，并加入如下机制：

- 铸造费用常量：`uint256 public constant MINT_FEE = 0.01 ether;`
- 最大供应量：`uint256 public constant MAX_SUPPLY = 100;`
- 合约拥有者：`address public owner;`
- 提款函数：`withdraw()`
- 默认版税：`_setDefaultRoyalty(msg.sender, 500)`

7.2 步骤二：编译并部署最终版本合约

在加入扩展功能后，再次使用 `soljson-v0.8.34+commit.80d5c536.js` 编译最终版 `SimpleNFT.sol`。编译通过后重新部署合约，为后续进阶功能测试提供独立实例。

7.3 步骤三：测试进阶功能

1. **测试铸造费机制**
   将 `mint()` 扩展为 `payable` 后，先使用错误铸造费发起调用，交易被回滚并显示 `Incorrect mint fee`；随后将 `Value` 正确设置为 `0.01 ether` 再次调用 `mint()`，交易成功执行，说明铸造费机制已正确生效，如图5所示。

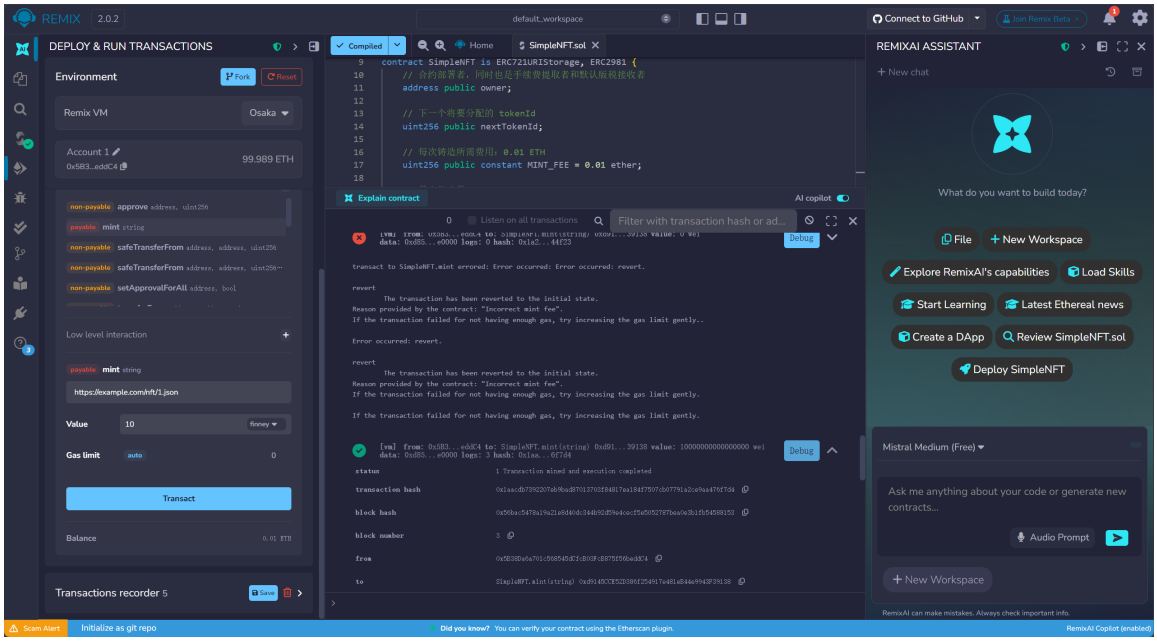

图 5: 最终版合约测试铸造费：错误费用失败，正确费用成功

2. **再次验证最终版 NFT 所有权**
   在最终版中调用 `ownerOf(0)`，返回地址仍为部署者账户 `0x5B38Da6a701c568545dCfcB03FcB875f56beddC4`，说明在增加铸造费和版税等扩展逻辑后，NFT 的所有权追踪功能仍能保持正确，如图6所示。

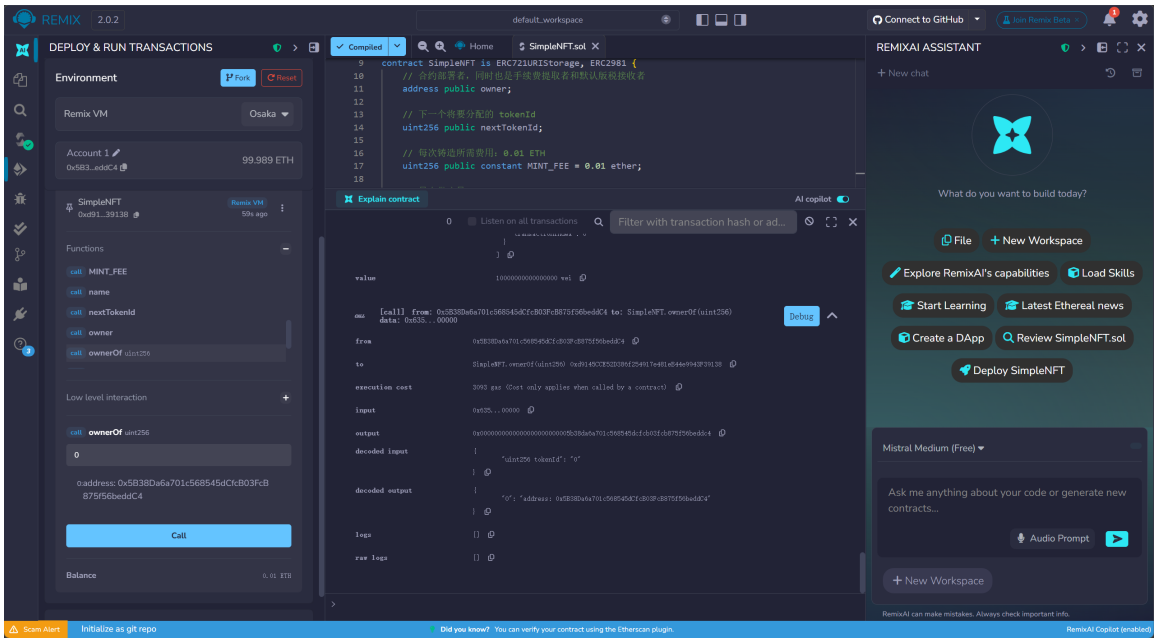

图 6: 最终版调用 `ownerOf(0)` 验证所有权归属

3. **再次验证最终版 URI 绑定**
   在最终版中调用 `tokenURI(0)`，返回值仍为 `https://example.com/nft/1.json`，

说明在加入扩展功能后，NFT 与链外元数据之间的绑定关系没有被破坏，如图7所示。

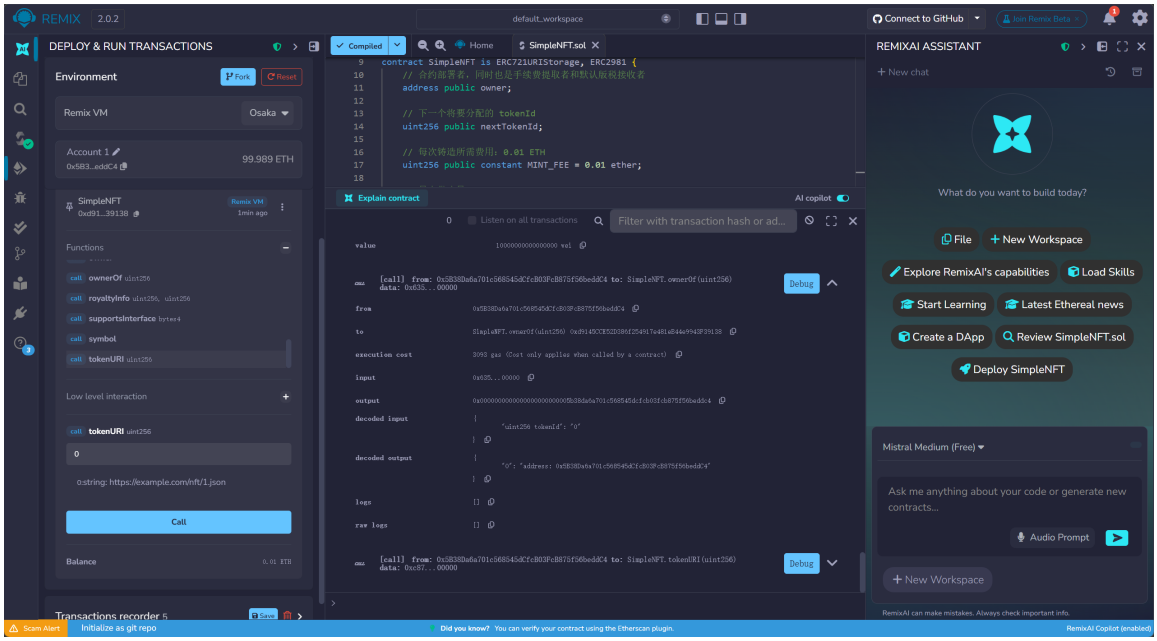

图 7: 最终版调用 `tokenURI(0)` 验证 URI 绑定

4. **测试版税信息查询**

   调用 `royaltyInfo(0, 1000000000000000000)`，即假设 `tokenId = 0` 的 NFT 以 1 ether 的价格在二级市场出售。查询结果显示版税接收者为 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4，版税金额为 5000000000000000 wei，即 0.05 ether，与构造函数中设置的 5% 默认版税完全一致，如图8所示。

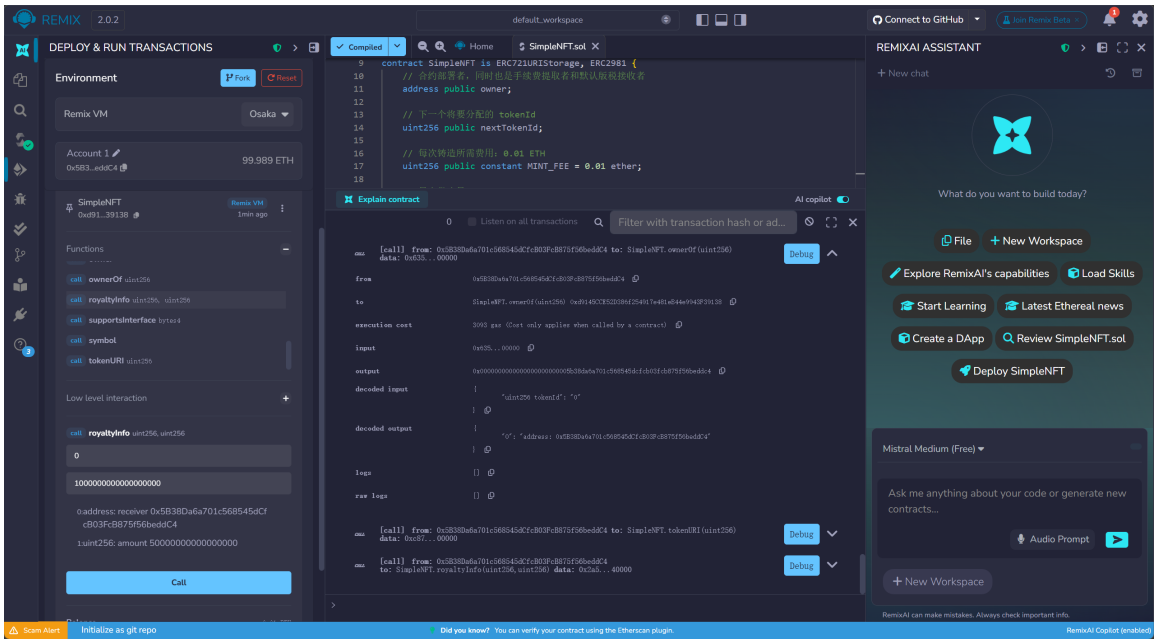

图 8: 最终版调用 `royaltyInfo()` 查询 5% 版税信息

5. **测试 owner 提取铸造费用成功**
   在至少一次成功铸造后，合约中累计保存了铸造费用。部署者账户调用 `withdraw()` 后，交易成功执行，且合约余额变为 `0 ETH`，说明拥有者具有提取铸造费用的权限，如图9所示。

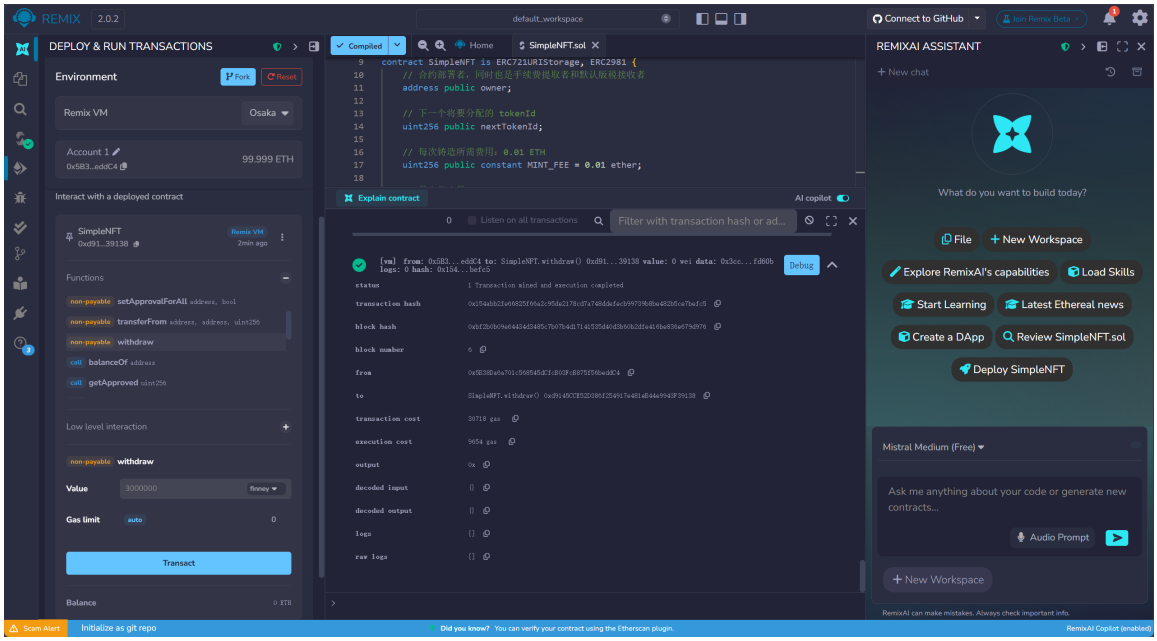

图 9: 最终版中 owner 成功调用 `withdraw()`

6. **测试非 owner 提款失败**
   将账户切换为第二个测试账户后，再次调用 `withdraw()`，交易被回滚并显示 `Not contract owner`，说明最终版中对提款权限的访问控制逻辑已经正确生效，如图10所示。

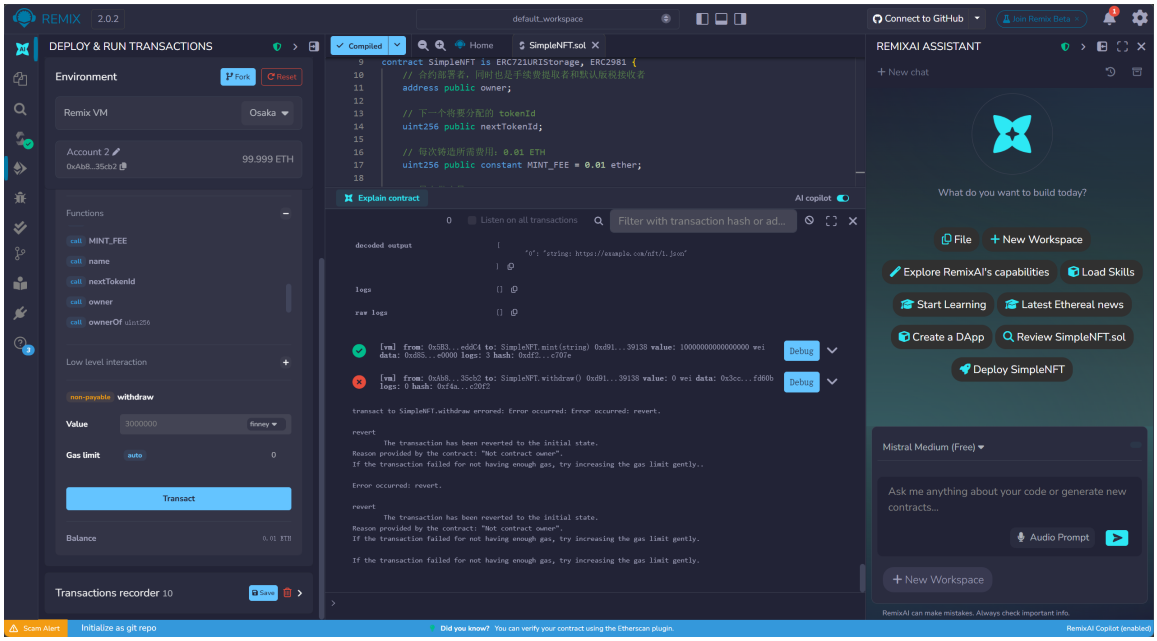

图 10: 非 owner 账户调用 `withdraw()` 失败

7.4 关于总量限制的实现说明

最终版代码中已通过常量 `MAX_SUPPLY = 100` 以及条件 `require(nextTokenId < MAX_SUPPLY, "Max supply reached")` 实现总量限制逻辑。该机制意味着当 NFT 数量达到 100 个后，后续铸造将被拒绝并回滚。由于当前提供的实验截图主要用于展示铸造费、所有权、URI、版税与提款权限控制，因此未单独展示达到供应上限后的回滚截图，但其限制逻辑已经在最终版代码中完整实现。

7.5 进阶部分实验现象记录

根据实验截图，进阶功能测试结果如下：

- 错误铸造费调用结果：失败，错误提示为 `Incorrect mint fee`
- 正确铸造费设置：`0.01 ether`
- 最终版铸造成功后合约余额：`0.01 ETH`
- 最终版 `ownerOf(0)` 返回地址：`0x5B38Da6a701c568545dCfcB03FcB875f56beddC4`
- 最终版 `tokenURI(0)` 返回值：`https://example.com/nft/1.json`
- `royaltyInfo(0, 1 ether)` 返回的接收者地址：`0x5B38Da6a701c568545dCfcB03FcB875f56beddC4`
- `royaltyInfo(0, 1 ether)` 返回的版税金额：`50000000000000000` wei（即 `0.05 ether`）
- owner 调用 `withdraw()` 的结果：成功
- 非 owner 调用 `withdraw()` 的结果：失败
- 非 owner 调用失败提示信息：`Not contract owner`

8 功能验证

8.1 基础版部署与铸造验证

图11展示了基础版 `SimpleNFT` 在 Remix VM 中成功部署并完成首次铸造后的界面。截图中可以看到部署成功记录、`mint()` 调用成功记录以及传入的 URI 参数，说明基础版合约已经具备最基本的 NFT 创建能力。

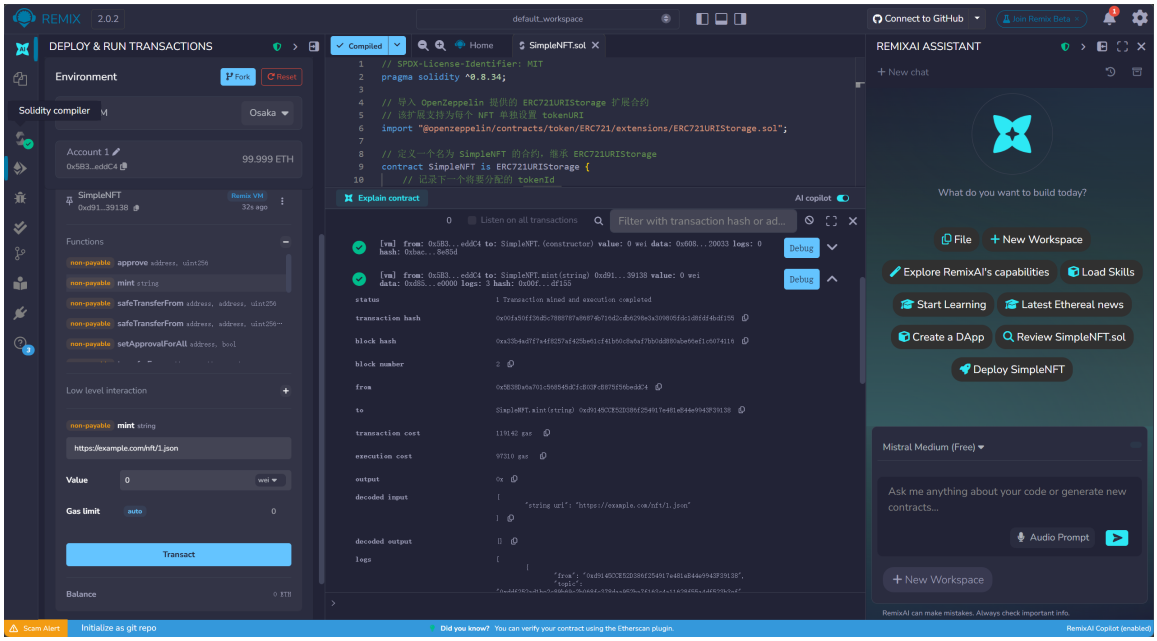

图 11: 基础版合约部署并完成首次铸造

8.2 基础版所有权与 URI 验证

图12与图13分别展示了基础版对 `ownerOf(0)` 与 `tokenURI(0)` 的调用结果。前者证明 NFT 的所有权已被正确登记到部署者账户，后者证明元数据 URI 已与该 NFT 建立正确绑定。

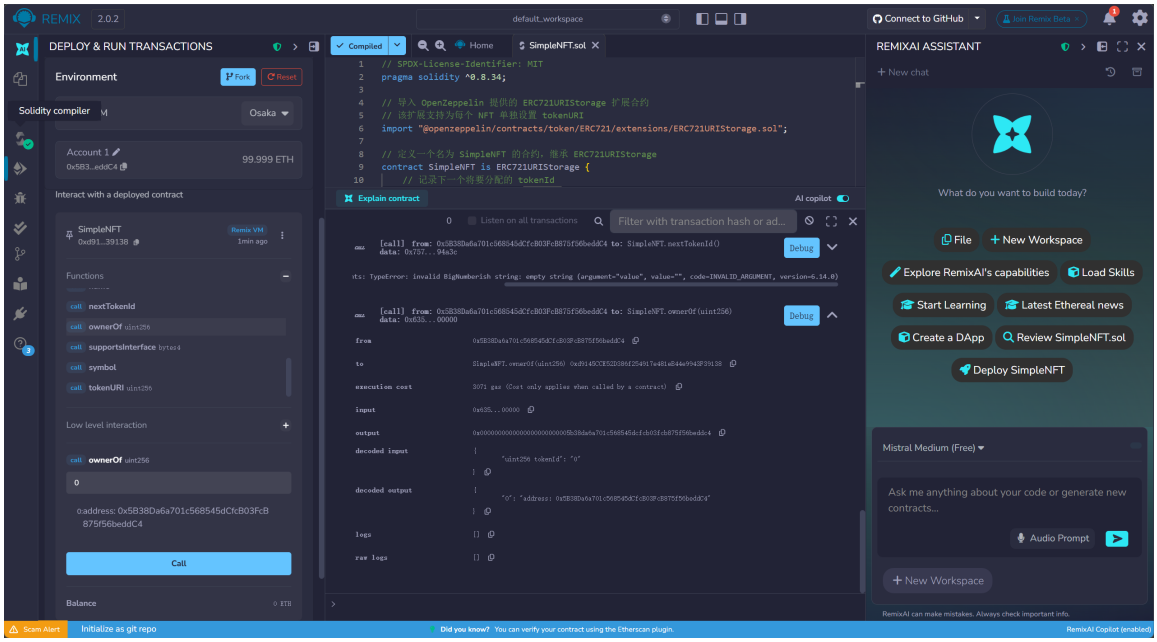

图 12: 基础版 `ownerOf(0)` 调用结果

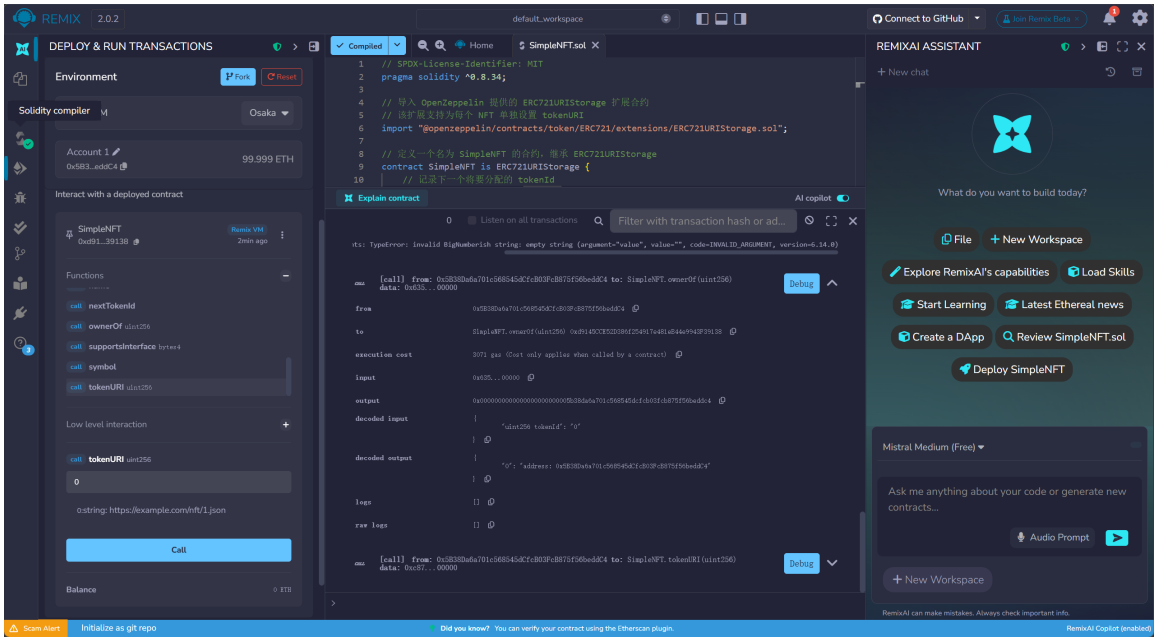

图 13: 基础版 `tokenURI(0)` 调用结果

8.3 进阶功能验证

图14、图15、图16 与图17 分别验证了铸造费、版税与提款权限控制机制。实验结果表明：错误费用不能完成铸造；`royaltyInfo()` 能返回正确的 5% 版税；部署者可以成功提取合约中的铸造费用；非部署者则无法调用 `withdraw()`。

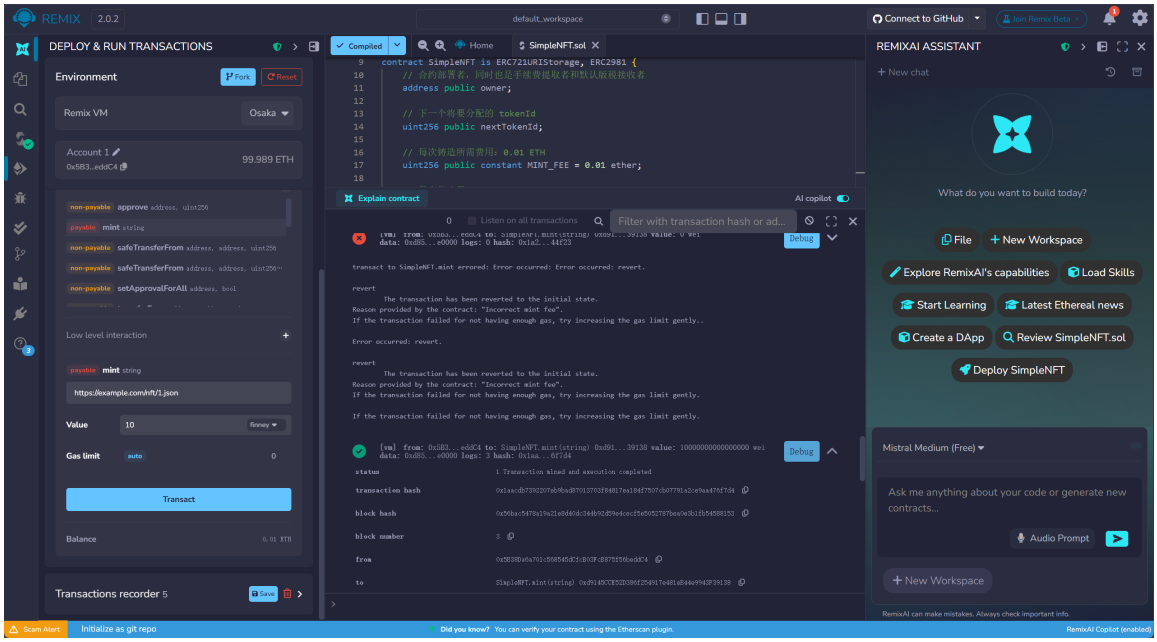

图 14: 最终版铸造费机制验证

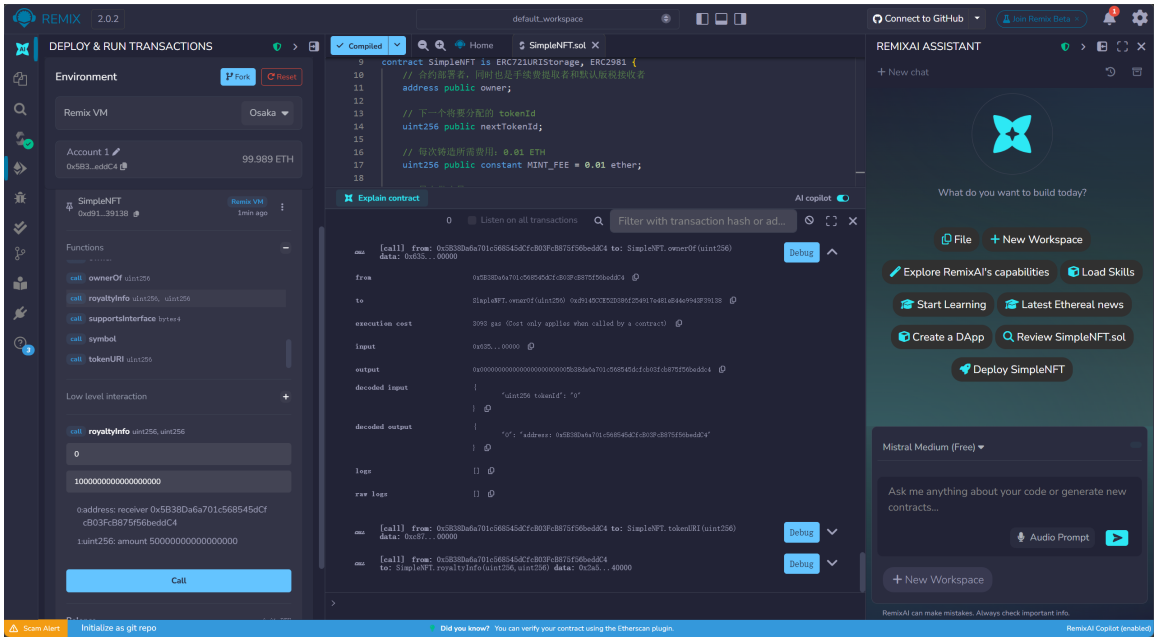

图 15: 最终版版税机制验证

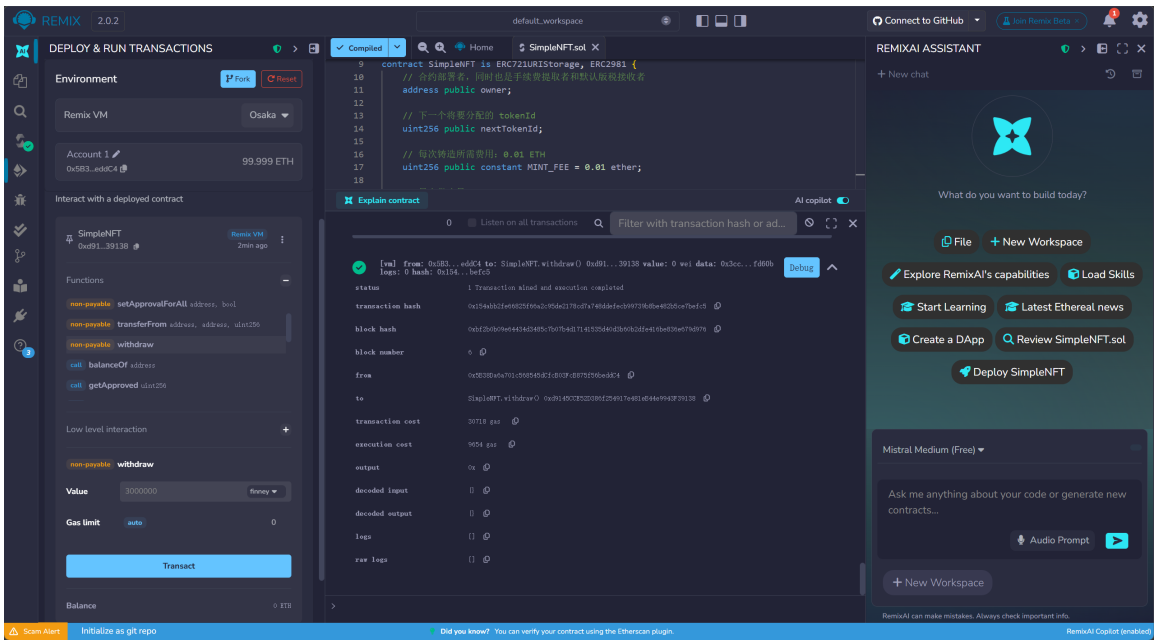

图 16: 最终版 owner 提款成功

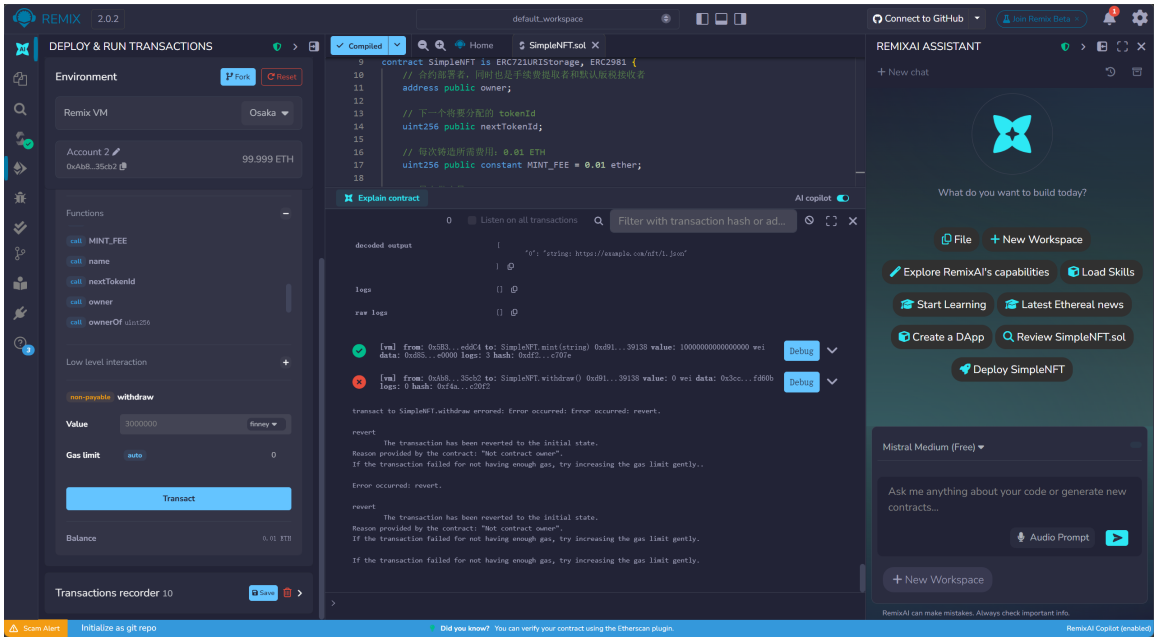

图 17: 最终版非 owner 提款失败

8.4 总量限制说明

最终版中通过 `MAX_SUPPLY` 与 `require(nextTokenId < MAX_SUPPLY)` 实现了总量限制机制。虽然本次截图未单独给出达到上限后的失败示例，但从最终代码逻辑可以判断，当铸造数量达到 100 个后，后续 `mint()` 将回滚并提示 `Max supply reached`。

A 附录A：基础版 Solidity 源代码

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.34;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";

contract SimpleNFT is ERC721URIStorage {

    uint256 public nextTokenId;

    event Minted(address indexed to, uint256 indexed tokenId, string uri);

    constructor() ERC721("MyArt", "MART") {}

    function mint(string memory uri) public {

        uint256 tokenId = nextTokenId;

                 msg.sender
        _safeMint(msg.sender, tokenId);

        _setTokenURI(tokenId, uri);

        emit Minted(msg.sender, tokenId, uri);

        nextTokenId++;
    }
}
```

B 附录B：最终版 Solidity 源代码

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.34;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";

import "@openzeppelin/contracts/token/common/ERC2981.sol";

contract SimpleNFT is ERC721URIStorage, ERC2981 {
```

```
    address public owner;

    uint256 public nextTokenId;

    uint256 public constant MINT_FEE = 0.01 ether;

    uint256 public constant MAX_SUPPLY = 100;

    event Minted(address indexed to, uint256 indexed tokenId, string uri);

    modifier onlyOwner() {
        require(msg.sender == owner, "Not contract owner");
        _;
    }

    constructor() ERC721("MyArt", "MART") {
        owner = msg.sender;

        _setDefaultRoyalty(msg.sender, 500);
    }

    function mint(string memory uri) public payable {
        require(msg.value == MINT_FEE, "Incorrect mint fee");

        require(nextTokenId < MAX_SUPPLY, "Max supply reached");

        uint256 tokenId = nextTokenId;

        _safeMint(msg.sender, tokenId);

        _setTokenURI(tokenId, uri);

        emit Minted(msg.sender, tokenId, uri);

        nextTokenId++;
    }

    function withdraw() public onlyOwner {
        uint256 balance = address(this).balance;
        require(balance > 0, "No ETH to withdraw");
```

```
        (bool success, ) = payable(owner).call{value: balance}("");
        require(success, "Withdraw failed");
    }

    function supportsInterface(bytes4 interfaceId)
        public
        view
        override(ERC721URIStorage, ERC2981)
        returns (bool)
    {
        return super.supportsInterface(interfaceId);
    }
}
```