如何用OpenZeppelin构建安全的智能合约

安全是区块链技术的核心基础之一。广为人知的网络安全漏洞实际上被归咎于与使用区块链的系统相连接，而不是区块链本身。不过，对于与去中心化应用互动的用户来说，安全往往是他们的首要考虑因素。他们可能想知道。"网络的安全性如何？"或者，"我的交易或资产的安全性如何？"OpenZeppelin是一家为去中心化应用提供安全产品的供应商，旨在解决这些问题。

OpenZeppelin提供开源的OpenZeppelin Contracts，用Solidity编写，用于构建安全的智能合约。OpenZeppelin Contracts使用ERC标准，用于基于Ethereum的代币，可用于许多类型的项目。为了尽量减少与在以太坊或其他区块链上构建安全智能合约有关的网络风险，OpenZeppelin Contracts不断地被审计和测试。

在本教程中，我们将研究如何使用OpenZeppelin合约和Truffle框架来构建和测试安全智能合约。此外，我们将演示如何在Remix IDE上使用OpenZeppelin Contracts。

在Truffle项目中使用OpenZeppelin Contracts

让我们探讨一下使用Truffle框架和OpenZeppelin Contracts构建智能合约。OpenZeppelin Contracts包含Solidity代码，被导入到包含我们Solidity代码的源文件中。这减少了必须手动编写的代码量。

我们将回顾如何设置Truffle开发环境，以及如何在Truffle项目中安装和实施OpenZeppelin。

设置Truffle开发环境

让我们使用Truffle框架为我们的项目建立一个开发环境。

要在全局范围内安装该框架，运行以下命令。

```
npm install truffle -g
```

接下来，创建一个新的目录以包含Solidity项目，`openzeppelin-contracts`。

```
mkdir openzeppelin-contracts
```

现在，cd进入新创建的文件夹，并运行以下Truffle命令。

```
truffle init
```

这将创建一个包含文件夹、源文件和配置文件的Truffle模板项目。我们可以对现有的源代码进行调整和/或添加更多的源代码，以建立我们特定的项目。

接下来，让我们用npm安装OpenZeppelin合同包。

```
npm init -y
npm install @openzeppelin/contracts
```

现在，我们将旋转一些自定义的token项目，使用OpenZeppelin Contracts来节省开发时间，也确保项目的安全性。在我们开发项目的过程中，我们将介绍该包所暴露的一些接口，以便在我们的代码中使用。

使用OpenZeppelin合约和Truffle建立一个ERC20项目

让我们建立一个ERC20可替换代币项目。

首先，cd进入包含solidity源文件的 `./contracts` 文件夹。在这个目录下，创建一个 `.sol` 文件， `Rocket.sol` 。

```
touch Rocket.sol
```

使用你喜欢的IDE，打开项目，用以下代码更新 `Rocket.sol` 文件，以实现一个自定义的令牌。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import '@openzeppelin/contracts/token/ERC20/ERC20.sol';

contract Rocket is ERC20 {
  constructor(uint256 initialSupply) ERC20("ROCKET", "ROC") {
    _mint(msg.sender, initialSupply * (10 ** decimals()));
  }
}

```

这个示例代码代表了实现一个自定义令牌所需的最小数量的OpenZeppelin合同包接口导入。使用 `ERC20` 函数，我们为令牌定义了一个自定义的名称， `ROCKET` ，以及一个符号， `ROC` 。

`constructor` 函数上的参数 `initialSupply` ，将容纳打算用于该令牌的总供应量。访问GitHub上的OpenZeppelin合约，查看从 `ERC20.sol` 文件中继承的其他接口。

现在，让我们为我们的智能合约写一个单元测试。我们将实现迁移文件，在本地开发智能合约，然后编写测试套件。

打开 `/migrations` 文件夹，并创建一个新的 `.js` 文件。然后，运行以下命令来创建 `2_customToken.js` 文件。

```
cd migrations
touch 2_customToken.js
```

在新文件内，添加以下JavaScript代码，为我们的自定义代币项目设置一个迁移过程。

```
const Rocket = artifacts.require("Rocket");

module.exports = function(deployer) {
  deployer.deploy(Rocket, 1000000);
}
```

接下来，让我们在 `/test` 文件夹内创建一个 `.test.js` 文件。我们将使用这个 `.test.js` 文件来测试我们的项目。

```
cd test
touch customToken.test.js
```

在新文件中，添加以下JavaScript代码，为我们的自定义令牌项目实现一个测试套件。

```
const Rocket = artifacts.require("contracts/Rocket.sol");

contract("Rocket", (accounts) => {
  before(async () => {
    rocket = await Rocket.deployed();
  });

  it("mint 1M worth of tokens", async () => {
    let balance = await rocket.balanceOf(accounts[0]);
    balance = web3.utils.fromWei(balance);
    assert.equal(balance, 1000000, "Initial supply of token is 1000000");
  });

  it("transfer token to another account", async () => {
    let amount = web3.utils.toWei("10000", "ether");
```

```
    await rocket.transfer(accounts[1], amount, { from: accounts[0] });
    let balance = await rocket.balanceOf(accounts[1]);
    balance = web3.utils.fromWei(balance);
    assert.equal(balance, 10000, "token balance is 10000");
  });
});
```

在Truffle的帮助下，我们定义了单元测试，以检查代币账户余额，并确认初始供应量与指定金额相匹配。

我们还定义了持有代币的第一个账户 `accounts[0]` ，和另一个账户 `account[1]` 之间的转账交易。这些是由Truffle框架提供给我们的假账户。我们将价值10,000的 `ROC` 代币从 `account[0]` 转到 `account[1]` 。

现在，让我们来运行测试。

```
truffle test

Compiling your contracts...
===========================
> Compiling ./contracts/Migrations.sol
> Compiling ./contracts/Rocket.sol
> Compiling @openzeppelin/contracts/token/ERC20/ERC20.sol
> Compiling @openzeppelin/contracts/token/ERC20/IERC20.sol
> Compiling @openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol
> Compiling @openzeppelin/contracts/utils/Context.sol
> Artifacts written to /var/folders/z2/963_51js5f370fvrx34q8cr40000gn/T/test-
-7499-6UUMhtN9GNHv
> Compiled successfully using:
   - solc: 0.8.11+commit.d7f03943.Emscripten.clang


  Contract: Rocket
    ✓ mint 1M worth of tokens
    ✓ transfer token to another account (69ms)


  2 passing (126ms)

```

`truffle test` 命令编译并将我们的智能合约迁移到由Truffle框架旋转的临时本地区块链上。接下来，Truffle针对我们智能合约中定义的功能和特性运行我们的测试套件。

使用OpenZeppelin合约和Truffle建立一个ERC721项目

现在，让我们建立一个ERC721非可替换代币项目。ERC721是指NFT智能合约项目的定义标准。非fungible意味着代币是唯一的。

让我们在 `/contracts` 文件夹中创建一个 `.sol` 文件， `SpaceShip.sol` ，。

接下来，我们将在 `SpaceShip.sol` 文件中添加以下源代码。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import"@openzeppelin/contracts/token/ERC721/ERC721.sol";
import"@openzeppelin/contracts/utils/Counters.sol";
contract SpaceShip is ERC721 {
    using Counters for Counters.Counter;
    Counters.Counter private _tokenIds;
    constructor() ERC721("SpaceShip","SPSP") {}
    function _mintToken(address account, string memory tokenURI) public
returns (uint256){
        _tokenIds.increment();
        uint256 newTokenId = _tokenIds.current();
        _mint(account, newTokenId);
        return newTokenId;
    }
}
```

在这段代码中，我们导入两个对象： `ERC721` 类对象和 `Counters` 库。我们继承了所有在 `ERC721` 类上定义的暴露接口。使用 `Counters` 库，我们为每个使用我们的智能合约铸造的NFT定义唯一的ID。

我们的铸币功能， `_mintToken` ，是每个新铸币将被链接到的账户。我们将接收者的账户ID， `address account` ，作为参数传递给这个铸币函数。

根据以太坊社区同意的元数据规范， `tokenURI` 参数指向代币资源，如图片、视频、访问或任何其他有价值的资源。在 `_mintToken` 函数内，我们调用我们的智能合约所暴露的 `_mint` 函数，以进行实际的计算和存储NFTs。

使用OpenZeppelin合约和Remix构建ERC1155项目

作为我们参观OpenZeppelin合约的最后一站，我们将在Ethereum的IDE，Remix上建立一个ERC1155 fungibility-agnostic，gas-efficient的代币项目。ERC1155项目可以用一个合约同时代表多个代币。

访问Remix - Ethereum IDE，获得一个在线IDE的实例，我们将用它来成功构建和编译我们的项目。

注意，本教程假定你有一些使用Remix的经验。

让我们在 `/contracts` 文件夹中创建一个 `.sol` 文件， `lemonade.sol` 。

接下来，我们将在 `lemonade.sol` 文件中添加以下源代码。

```
pragma solidity ^0.8.0;

import "https://github.com/OpenZeppelin/openzeppelin-contracts/master/contracts/token/ERC1155/ERC1155.sol";
import "https://github.com/OpenZeppelin/openzeppelin-contracts/master/contracts/access/Ownable.sol";

contract Lemonade is ERC1155, Ownable {
    uint256 public const LEMONADE = 0;
    constructor() ERC1155("") {
        _mint(msg.sender, LEMONADE, 12, "");
    }

    function mint(address account, uint256 tokenID, uint256 supply) public onlyOwner {
        _mint(account, tokenID, supply, "");
    }

    function burn(address account, uint256 tokenID, uint256 amount) public onlyOwner {
        _burn(account, tokenID, amount);
    }
}
```

在这段代码中，我们通过GitHub链接导入Remix中的OpenZeplin合约的源文件， `ERC1155.sol` 和 `Ownable.sol` 。我们在智能合约中实现了 `_mint()` 和 `_burn()` 两个函数。 `_burn()` 函数接口消除了对ERC721规范的限制因素，因为一旦智能合约部署在区块链上，铸币就立即受到限制。我们还扩展了智能合约，以便我们在未来可以铸造更多的NFT。

总结

在本教程中，我们演示了如何使用OpenZeppelin合约和Truffle框架来构建和测试ERC20可替换代币和ERC721不可替换代币的安全智能合约。我们还演示了如何使用OpenZeppelin Contracts和Remix为ERC1155可替代性、气体效率高的代币构建智能合约。

OpenZeppelin合约可用于构建去中心化的应用，范围包括DeFi、治理、访问和安全智能合约，以及NFT游戏、赌注或玩赚项目等定制项目。

为了更好地了解OpenZeppelin合约如何提高DApp开发的效率，请阅读EIP规范，并在该公司的文档或GitHub上研究OpenZeppelin代码库。