Homework 5

Using Chainlink Oracles

108321033 吳明騰

108321056 唐泳烽

108321062 許恩慈

Consuming Data Feeds

Introduction: In this chapter, we will demonstrate the utilization of Chainlink Oracle to obtain the token prices from various blockchains and convert them into their corresponding values in US dollars. This process involves referencing an informative article on Consuming Data Feeds that provides valuable insights and guidelines for extracting and utilizing the required data. By following the instructions outlined in the article, we will showcase how to effectively leverage Chainlink Oracle to capture token prices accurately across different blockchains and convert them into US dollar values.

constructor(): The constructor function plays a pivotal role in our code, as it establishes the interaction with the Chainlink Oracle using the AggregatorV3Interface(). By providing the relevant address associated with the desired coins, we enable the ability to make queries and retrieve data. For more detailed information regarding this process, please refer to the specified website.

In the accompanying figure below, we illustrate how we feed the BTC/USD, ETH/USD, JPY/USD, and LINK/USD data respectively into our system. This ensures that we have access to the latest and accurate price information for these specific cryptocurrency pairs.

```
constructor() {
        priceFeed_BTC_USD = AggregatorV3Interface(
                0x1b44F3514812d835EB1BDB0acB33d3fA3351Ee43
        );
        priceFeed_ETH_USD = AggregatorV3Interface(
                0x694AA1769357215DE4FAC081bf1f309aDC325306
        );
        priceFeed_JPY_USD = AggregatorV3Interface(
                0x8A6af2B75F23831ADc973ce6288e5329F63D86c6
        );
        priceFeed_LINK_USD = AggregatorV3Interface(
                0xc59E3633BAAC79493d908e63626716e204A45EdF
        );
}
```

getLatestPrice(): The getLatestPrice() function serves the purpose of fetching data from the Chainlink Oracle and storing it in designated variables. Within this function, we extract essential information such as the roundID and answerInRound, which represent the identification numbers associated with the data request. Additionally, the price variable captures the prices of two distinct tokens, and requiring multiplication by a factor of 10^{8} to obtain the accurate value.

Moreover, the startAt and timeStamp variables denote the specific time when this data was retrieved from the Oracle. These timestamps are measured in seconds and may require conversion to a more suitable format for presentation or further analysis.

```
function getLatestPrice_BTC_USD() public view returns (uint80, int , uint, uint, uint80) {
        (
                uint80 roundID ,
                int price,
                uint startedAt,
                uint timeStamp,
                uint80 answeredInRound
        ) = priceFeed_BTC_USD.latestRoundData();
        return (roundID, price, startedAt, timeStamp, answeredInRound);
}
```

Result: The figure below is the result of our retrieval of data

	roundID / answeredInRound	price	startedAt / timeStamp
BTC_USD	18446744073709556000	26766.33	2023-05-22 14:04:36
ETH_USD	18446744073709556000	1803.54	2023-05-22 13:19:12
JPY_USD	18446744073709552000	0.00725364	2023-05-21 23:18:48
LINK_USD	18446744073709556000	6.442603	2023-05-22 13:39:00

Random Numbers: Using Chainlink VRF

Introduction: In this chapter, our focus shifts to the generation of random numbers within the context of blockchain. While blockchain is often considered a deterministic machine, where querying nodes simultaneously yields the same results, this predictability conflicts with the inherent nature of random numbers, which require unpredictability.

To address this challenge, Chainlink offers an alternative API called Chainlink VRF (Verifiable Random Function). This API enables the generation of random numbers within the blockchain ecosystem while ensuring their verifiability. To delve deeper into this topic, we recommend referring to the article titled "Random Numbers: Using Chainlink VRF," which provides valuable insights and guidelines on incorporating Chainlink VRF for random number generation.

In addition, this chapter includes the implementation code, available through Remix Code, which offers a practical demonstration of the process involved in utilizing Chainlink VRF for generating random numbers. By following the instructions and studying the provided code, you will gain a clear understanding of how to effectively integrate Chainlink VRF into your blockchain projects for secure and verifiable random number generation.

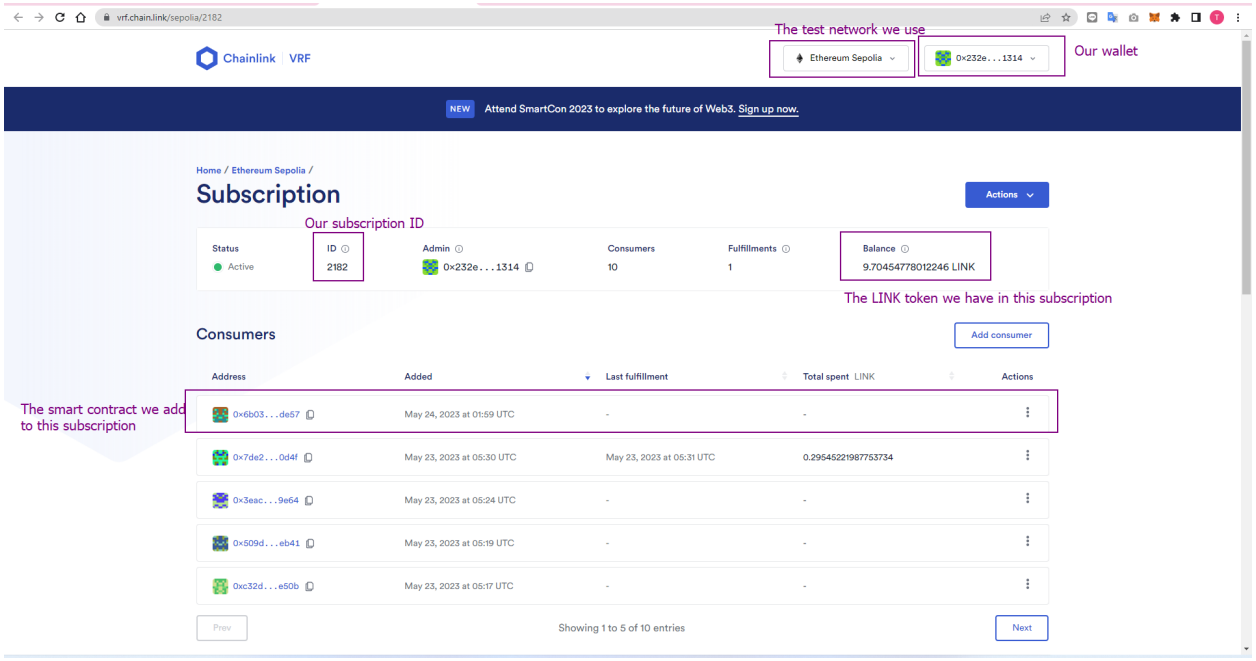
Subscription: The concept of subscriptions plays a significant role in managing the funding for VRF (Verifiable Random Function) requests. Subscription accounts are utilized to provide the necessary funds for VRF requests. To simplify the management of funding for your application requests, Chainlink offers the Subscription Manager. This tool allows you to create an account and pre-pay for VRF requests, ensuring that all funding requirements are centrally managed.

In the context of our discussion, the accompanying picture below displays an example of a subscription. The ID associated with this Subscription is 2182. To initiate the subscription, we begin by transferring some LINK tokens, the native currency of the Chainlink network, to the subscription account. This allocation of funds ensures that there are sufficient resources available to support the VRF requests made by our application.

Additionally, we take the next step of adding the deployed smart contract to the subscription. This integration allows the subscription account to manage the funding for

VRF requests originating from our specific smart contract, streamlining the process and ensuring efficient management of resources.

By leveraging subscriptions and utilizing the Subscription Manager, we can conveniently allocate and manage funds for VRF requests within a single, centralized location, simplifying the overall process and enhancing the operational efficiency of our application.



Variables: The figure below shows some setting variables, among which the values of vrfCoordinator and s_keyHash we refer to [subscription supported-networks configurations], we use Sepolia test network

```
uint64 s_subscriptionId;
address vrfCoordinator = 0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625;
bytes32 s_keyHash=0x474e34a077df58807dbe9c96d3c009b23b3c6d0cce433e59bbf5b34f823bc56c;
uint32 callbackGasLimit = 2500000;
uint16 requestConfirmations = 3;
uint32 numWords =  1;
uint256 private constant ROLL_IN_PROGRESS = 42;
```

constructor(): In the function constructor() in the figure below, we use the suite VRFConsumerBaseV2.sol provided by Chainlink

```
// constructor
  constructor(uint64 subscriptionId) VRFConsumerBaseV2(vrfCoordinator) {
    COORDINATOR = VRFCoordinatorV2Interface(vrfCoordinator);
    s_owner = msg.sender;
    s_subscriptionId = subscriptionId;
  }
```

rollDice(): The function rollDice() in the figure below is the random number that the actual call comes from the blockchain

```
// rollDice function
  function rollDice(address roller) public onlyOwner payable returns (uint256 requestId) {
    require(s_results[roller] == 0, "Already rolled");
    // Will revert if subscription is not set and funded.
    requestId = COORDINATOR.requestRandomWords(
    s_keyHash,
    s_subscriptionId,
    requestConfirmations,
    callbackGasLimit,
    numWords
   );

    s_rollers[requestId] = roller;
    s_results[roller] = ROLL_IN_PROGRESS;
    emit DiceRolled(requestId, roller);
    // return requestId;
  }
```

API Calls: Using Any API

Introduction: To begin, we will explore the role of Tasks and External adapters in the context of smart contracts. Tasks are specific actions or functions that smart contracts need to perform, while External adapters act as bridges between the smart contract and external data sources, such as APIs. Understanding these components is crucial to effectively working with Oracle Jobs.

Speaking of Oracle Jobs, you will learn how they leverage Tasks and External adapters to facilitate the retrieval of data from external APIs. Oracle Jobs are specialized components that allow smart contracts to interact with and fetch data from these external sources. By grasping the mechanics of Oracle Jobs, you will gain the necessary knowledge to seamlessly incorporate data requests into your smart contract.

We all refer to this article API Calls: Using Any API , and implement the code Remix Code.

Using Terminal:

```
$ curl -X 'GET' \
        'https://min-api.cryptocompare.com/data/pricemultifull?fsyms=ETH&tsyms=USD' \
         -H 'accept: application/json'
```


```
{
        "RAW": {
                "ETH": {
                        "USD": {
                                "TYPE": "5",
                                "MARKET":"CCCAGG",
                                "FROMSYMBOL":"ETH",
                                "TOSYMBOL": "USD",
                                "FLAGS": "2052",
                                "PRICE":1904.51,
                                "LASTUPDATE":1685338133,
                                "MEDIAN":1904.45,
                                "LASTVOLUME": 0.01710677,
                                "LASTVOLUMETO": 32.5774485172,
                                "LASTTRADEID":"453152290",
                                "VOLUMEDAY":54196.43703646803,
                                "VOLUMEDAYTO":103683250.82123952,
                                "VOLUME24HOUR": 158743.2615551,
                                "VOLUME24HOURTO": 299891826.69759256,
```

```
                    "OPENDAY": 1909.21,
                    ...
}
```

Constructor(): We are going to use Sepolia Testnet. Here are some details settings

* Link Token: 0x779877A7B0D9E8603169DdbD7836e478b4624789
* Oracle: 0x6090149792dAAeE9D1D568c9f9a6F6B46AA29eFD (Chainlink DevRel)
* jobId: ca98366cc7314957b8c012c72f05aeeb

The corresponding table of jobId and each function is in this website , we choose "7da2702f37fd48e5b1b9a5715e3509b6" this time, where tasks are Http, JsonParse, Ethabiencode.

```
constructor() ConfirmedOwner(msg.sender) {
        setChainlinkToken(0x779877A7B0D9E8603169DdbD7836e478b4624789);
        setChainlinkOracle(0x6090149792dAAeE9D1D568c9f9a6F6B46AA29eFD);
        jobId = "7da2702f37fd48e5b1b9a5715e3509b6";
        fee = (1 * LINK_DIVISIBILITY) / 10; // 0,1 * 10**18 (Varies by network and job)
}
```

request(): This function serves the purpose of configuring specific parameters and establishing a connection with the blockchain. Within this function, we utilize the req.add() method to set up HTTP calls to the desired API. It is crucial to ensure that the method is set to "GET" to retrieve data from the API successfully.

To process the obtained JSON data, we utilize JSON Parse, which enables the extraction of specific values at designated key paths within the JSON structure. This functionality is essential for retrieving the precise data required for our application.

Furthermore, the multiply() function plays a crucial role in manipulating the retrieved data. By multiplying the input value by a predetermined multiplier, we can eliminate decimal places and work with whole numbers, if necessary. This step is particularly useful in cases where decimal precision is not required or when working with specific data formats that do not accommodate decimal values.

By utilizing these functions and techniques, we can effectively configure API calls, retrieve and parse JSON data, and perform necessary data manipulations to ensure the accurate and appropriate utilization of the retrieved information within our blockchain application.

```
function request() public {
        Chainlink.Request memory req = buildChainlinkRequest(
                jobId,
                address(this),
                this.fulfill.selector
        );
        req.add(
                'get',
                'https://min-api.cryptocompare.com/data/pricemultifull?fsyms=ETH&tsyms=USD'
        );
        req.add('path', 'RAW,ETH,USD,VOLUME24HOUR');
        req.addInt('times', 10**18);
        sendChainlinkRequest(req, (1 * LINK_DIVISIBILITY) / 10); // 0,1*10**18 LINK
}
```

fulfill(): This function plays a crucial role in the Oracle Job's workflow, as it serves as the designated destination where the result is sent upon the completion of the job. Once the Oracle Job successfully retrieves the requested data or completes the assigned task, it forwards the result to the fulfill function.

The fulfill function acts as a recipient and handles the received result, allowing further processing or utilization within the smart contract or application. It serves as a pivotal component in the interaction between the Oracle Job and the smart contract, ensuring the seamless transfer of data and facilitating the execution of subsequent actions based on the obtained result.

```
uint256 public volume;
function fulfill (
        bytes32 _requestId,
        uint256 _volume
) public recordChainlinkFulfillment(_requestId) {
                volume = _volume;
}
```