

1 Chainlink PriceFeeds 基本使用

PriceFeeds主要用于为Defi(去中心化金融)项目提供链下的资产价格参考，也就是说用户或项目方可以通过Chainlink的PriceFeeds合约获取到链下的资产价格数据。

Chainlink 官网给了一个PriceFeeds的使用例子:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

import "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";

contract PriceConsumerV3 {
    AggregatorV3Interface internal priceFeed;

    /**
     * Network: Goerli
     * Aggregator: ETH/USD
     * Address: 0xD4a33860578De61DBAbDc8BfDb98FD742fA7028e
     */
    constructor() {
        priceFeed = AggregatorV3Interface( //实例化AggregatorV3Interface接口
            0xD4a33860578De61DBAbDc8BfDb98FD742fA7028e //指向ETH/USD价格参考合约
        );
    }

    /**
     * Returns the latest price
     */
    function getLatestPrice() public view returns (int) {
        (
            uint80 roundID, //当前喂价轮次的ID
            int price, //ETH/USD币对价格
            uint startedAt, //本轮喂价开始的时间戳
            uint timeStamp, //获得最终聚合价格的时间戳
            uint80 answeredInRound //价格被计算出来时的轮次ID
        ) = priceFeed.latestRoundData();
        return price;
    }
}
```























PriceFeeds官方示例代码

Chainlink在各条公链生态上为每个资产币对分别创建了一个价格参考合约，当我们需要获得特定资产的价格数据时，只需要在constructor()函数中将接口实例化的地址修改成对应资产的价格参考合约地址然后将合约部署上链，再调用getLatestPrice()函数即可获取到相应的资产价格。各种资产的参考价格合约地址可以在Chainlink官网进行查询。

[参考价格合约地址查询网站](#)

Ethereum Mainnet

☐ Show more details

Pair	Asset	Type	Address
 1INCH / ETH	1inch	Crypto	 0x72AFAECF99C9d9C8215fF44C77B94
 1INCH / USD	1inch	Crypto	 0xc929ad75B72593967DE83E7F7Cda0
 AAPL / USD	Apple	Equities	 0x139C8512Cde1778e9b9a8e721ce1aEb
 AAVE / ETH	Aave	Crypto	 0x6Df09E975c830ECae5bd4eD9d90f3
 AAVE / USD	Aave	Crypto	 0x547a514d5e3769680Ce22B2361c10E
 ADA / USD	Cardano	Crypto	 0xAE48c91dF1fE419994FFDa27da09D5
 ADX / USD	Adex	Crypto	 0x231e764B44b2C1b7Ca171fa8021A24e
 ALBT / USD	AllianceBlock	Crypto	 0x057e52Fb830318E096CD96F369f0D
 ALCX / ETH	Alchemix	Crypto	 0x194a9AaF2e0b67c35915cD01101585
 ALCX / USD	Alchemix	Crypto	 0xc355e4C0B3ff4Ed0B49EaACD55FE2
 ALPHA / ETH	Alpha Finance	Crypto	 0x89c7926c7c15fD5BFDB1edcFf7E7fC8

官网资产价格参考合约地址示意图

2 PriceFeeds合约分析

在上面官方给的样例合约中可以看到获取价格数据的功能由价格参考合约接口AggregatorV3Interface实现，样例合约只是调用了价格参考合约里的latestRoundData()函数，因此若需要对PriceFeeds的链上合约部分进行分析，我们就需要进入到价格参考合约中看latestRoundData()的具体实现。

为了解PriceFeeds功能在生产环境中的具体实现，我们直接找一个在以太坊主网中真正提供服务的价格参考合约进行分析。

 BTC / ETH	Bitcoin	Crypto	 0xdeb288F737066589598e9214E782fa5A8eD689e8
-----------------------------------------------------------------------------------------------	---------	--------	------------------------------------------------------------------------------------------------------------------------------------------------

BTC/ETH价格参考合约地址

这里选的是主网中的BTC/ETH价格参考合约，我们在[以太坊区块链浏览器](#)中搜索该合约地址，点击“Contract”按钮即可查看合约源码。价格参考合约在以太坊区块链浏览器中的合约名称为“EACAggregatorProxy”。

主网BTC/ETH价格参考合约地址：0xdeb288F737066589598e9214E782fa5A8eD689e8

Transactions Internal Txns Erc20 Token Txns **Contract** Events Analytics Comments

Code Read Contract Write Contract Search Source Code

✓ Contract Source Code Verified (Exact Match)

Contract Name: **EACAggregatorProxy** Optimization Enabled: Yes with 1000000 runs

Compiler Version: v0.6.6+commit.6c089d02 Other Settings: default evmVersion, MIT license

Contract Source Code (Solidity)

```
27     owner = msg.sender;
28 }
29
30 /**
31  * @dev Allows an owner to begin transferring ownership to a new address,
32  * pending.
33  */
34 function transferOwnership(address _to)
35     external
36     onlyOwner()
37 {
38     pendingOwner = _to;
39     emit OwnershipTransferRequested(owner, _to);
40 }
41
42 /**
43  * @dev Allows an ownership transfer to be completed by the recipient.
44  */
45 function acceptOwnership()
46     external
47 {
48     require(msg.sender == pendingOwner, "Must be proposed owner");
49     address oldOwner = owner;
50 }
51
```

[etherscan中的合约查看页面](#)

由于合约源码较长，下面只截取相关的部分合约实现源码进行描述。

2.1 EACAggregatorProxy合约

```
/**
 * @title External Access Controlled Aggregator Proxy
 * @notice A trusted proxy for updating where current answers are read from
 * @notice This contract provides a consistent address for the
 * Aggregator and AggregatorV3Interface but delegates where it reads from to the
 owner, who is
 * trusted to update it.
 * @notice Only access enabled addresses are allowed to access getters for
 * aggregated answers and round information.
 */
contract EACAggregatorProxy is AggregatorProxy { //继承AggregatorProxy合约

    AccessControllerInterface public accessController; //定义一个合约接口类型

    constructor(
        address _aggregator, //聚合器合约地址
        address _accessController //权限控制地址，用于权限判定
    )
    public
        AggregatorProxy(_aggregator) //实例化父合约
    {
        setController(_accessController);
    }

    /**
     * @notice Allows the owner to update the accessController contract address.

```

```

    * @param _accessController The new address for the accessController contract
    */
function setController(address _accessController)
    public
    onlyOwner()
{
    //实例化权限控制合约
    accessController = AccessControllerInterface(_accessController);
}
}

```

价格参考合约EACAggregatorProxy初始化代码

先来看看价格参考合约的一些基本定义，我们可以看到价格参考合约EACAggregatorProxy继承了父合约AggregatorProxy，并且在构造函数中将聚合器合约作为传入参数将父合约实例化，然后调用setController函数将权限控制合约实例化并将合约对象赋予accessController。

```

interface AccessControllerInterface {
    function hasAccess(address user, bytes calldata data) external view returns
    (bool);
}

```

权限控制合约接口

权限控制合约accessController是通过接口AccessControllerInterface实例化的，而该接口中只有一个hasAccess函数，入参是名为user的地址类型以及一个calldata，初步判定这个函数的功能是判断某个地址是否有权限。

```

/**
 * @notice get data about the latest round. Consumers are encouraged to check
 * that they're receiving fresh data by inspecting the updatedAt and
 * answeredInRound return values.
 * Note that different underlying implementations of AggregatorV3Interface
 * have slightly different semantics for some of the return values. Consumers
 * should determine what implementations they expect to receive
 * data from and validate that they can properly handle return data from all
 * of them.
 * @return roundId is the round ID from the aggregator for which the data was
 * retrieved combined with a phase to ensure that round IDs get larger as
 * time moves forward.
 * @return answer is the answer for the given round
 * @return startedAt is the timestamp when the round was started.
 * (Only some AggregatorV3Interface implementations return meaningful values)
 * @return updatedAt is the timestamp when the round last was updated (i.e.
 * answer was last computed)
 * @return answeredInRound is the round ID of the round in which the answer
 * was computed.
 * (Only some AggregatorV3Interface implementations return meaningful values)
 * @dev Note that answer and updatedAt may change between queries.
 */
function latestRoundData()

```

```

public
view
checkAccess()
override
returns (
    uint80 roundId,    //聚合器进行数据聚合的轮次ID
    int256 answer,     //最终聚合得到的价格数据
    uint256 startedAt, //聚合开始的时间戳
    uint256 updatedAt, //聚合结束的时间戳(算出最终answer并更新的时间戳)
    uint80 answeredInRound //answer被计算出来时的轮次ID
)
{
    return super.latestRoundData();
}

```

价格参考合约的latestRoundData()函数实现

官方示例中的合约就是调用了价格参考合约的latestRoundData函数获取价格数据的，我们可以看到latestRoundData函数调用了函数修饰器checkAccess用于权限判断，判断调用者是否有获取价格数据的权限。而在函数的实现中，该函数通过super字段调用了父合约的latestRoundData函数来获取相应的价格数据。因此我们需要进一步查看checkAccess和父合约AggregatorProxy的实现。

先看看checkAccess修饰器的实现：

```

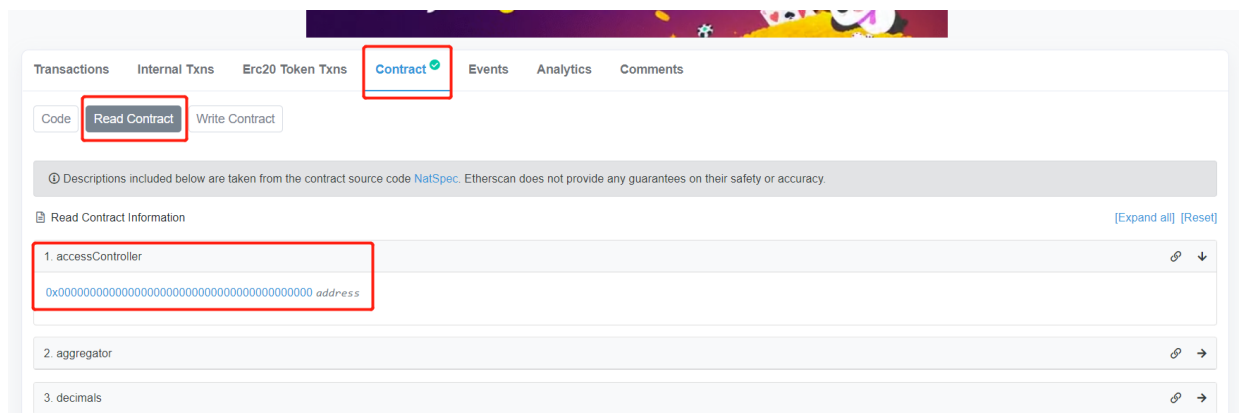
modifier checkAccess() {
    AccessControllerInterface ac = accessController;
    require(address(ac) == address(0) || ac.hasAccess(msg.sender, msg.data), "No
access");
    _;
}

```

价格参考合约中的checkAccess函数修饰器

修饰器checkAccess中实例化了一个ac用于接收权限控制合约accessController,然后该修饰器判断通过的条件是当ac合约地址为0地址或者是使用该修饰器的函数的调用者地址能够通过ac合约内hasAccess函数的权限认证。因此若accessController为0地址，即没有实例化权限控制合约时，checkAccess能无条件通过，否则则需要通过hasAccess的权限认证判断调用者是否有权限。

为了了解Chainlink在checkAccess修饰器中所作的权限控制，我们需要到权限控制合约中查看具体实现。



etherscan中accessController变量的最新值

在etherscan中可以直接查看已上链合约中变量的最新值，我们可以在BTC/ETH价格参考合约中通过查看该合约中accessController合约对象的地址找到Chainlink部署上链的权限控制合约，但是查看完之后发现目前accessController为0地址，这也就意味着Chainlink当前并没有给价格参考合约设置权限，任何地址都能通过checkAccess的权限判断。

既然不能通过这里找到权限控制合约的源码，那我们也可以通过github寻找Chainlink项目方的开源代码库，并从中寻找权限控制合约。

```
/**
 * @title SimplewriteAccessController
 * @notice Gives access to accounts explicitly added to an access list by the
 * controller's owner.
 * @dev does not make any special permissions for externally, see
 * SimpleReadAccessController for that.
 */
contract SimplewriteAccessController is AccessControllerInterface, ConfirmedOwner {
    bool public checkEnabled;
    mapping(address => bool) internal accessList;

    constructor() ConfirmedOwner(msg.sender) {
        checkEnabled = true;
    }

    /**
     * @notice Returns the access of an address
     * @param _user The address to query
     */
    function hasAccess(address _user, bytes memory _calldata) public view virtual
    override returns (bool) {
        return accessList[_user] || !checkEnabled;
    }

    /**
     * @title SimpleReadAccessController
     * @notice Gives access to:
     * - any externally owned account (note that off-chain actors can always read
     * any contract storage regardless of on-chain access control measures, so this
     * does not weaken the access control while improving usability)
     * - accounts explicitly added to an access list
     * @dev SimpleReadAccessController is not suitable for access controlling writes
     * since it grants any externally owned account access! See
     * SimplewriteAccessController for that.
     */
    contract SimpleReadAccessController is SimplewriteAccessController {
        /**
         * @notice Returns the access of an address
         * @param _user The address to query
         */
        function hasAccess(address _user, bytes memory _calldata) public view virtual
        override returns (bool) {
```

```
        return super.hasAccess(_user, _calldata) || _user == tx.origin;
    }
}
```

权限控制合约部分代码

权限控制合约SimpleReadAccessController，其继承了父合约SimpleWriteAccessController，SimpleReadAccessController中的hasAccess函数中调用到了父合约的hasAccess函数，我们先来看看父合约中的hasAccess函数。

父合约的hasAccess函数的入参是address类型的_user以及一个bytes类型的calldata，返回值则是bool类型。该函数根据传入的地址是否在accessList权限清单中来判断该地址是否有访问权限，而checkEnabled则是一个判断开关，当checkEnabled被设置为false时hasAccess无条件返回true，所以实际上父合约中的hasAccess就是一个白名单查询函数。这里可以看到传入的calldata并没有被使用，推测可能留作后续合约升级。

接在再来看看子合约SimpleReadAccessController中的hasAccess函数。这个hasAccess函数除了调用父合约的hasAccess函数判断白名单权限外，还对user是否为tx.origin进行判断，当调用hasAccess的地址就是这次交易的最初发起者时该函数也返回true，这也就意味着当外部个人账户直接与价格参考合约交互获取价格数据时也能通过checkAccess修饰器的权限判断。因此在用含有hasAccess的checkAccess修饰器进行权限判断时，只有外部个人账户地址或在白名单内的地址可以通过权限认证，而若是其他DeFi项目想要通过价格参考合约获取资产价格数据，则需要先联系Chainlink官方将该DeFi项目中的合约地址加入白名单。

当然前面也提到价格参考合约中的accessController合约目前是0地址，也就说明目前Chainlink官方并没有在一些主流资产价格参考合约上设置权限控制进行如上面所描述的权限判断，个人推断可能在一些DeFi项目因需要一些非主流资产币对的链下价格数据找Chainlink团队定制数据聚合服务时会启用该权限控制合约。

2.2 AggregatorProxy合约

分析完了checkAccess权限控制的实现，现在还需要看看价格参考合约的父合约AggregatorProxy中的latestRoundData函数实现。

```
/**
 * @notice get data about the latest round. Consumers are encouraged to check
 * that they're receiving fresh data by inspecting the updatedAt and
 * answeredInRound return values.
 * Note that different underlying implementations of AggregatorV3Interface
 * have slightly different semantics for some of the return values. Consumers
 * should determine what implementations they expect to receive
 * data from and validate that they can properly handle return data from all
 * of them.
 * @return roundId is the round ID from the aggregator for which the data was
 * retrieved combined with an phase to ensure that round IDs get larger as
 * time moves forward.
 * @return answer is the answer for the given round
 * @return startedAt is the timestamp when the round was started.
 * (Only some AggregatorV3Interface implementations return meaningful values)
 * @return updatedAt is the timestamp when the round last was updated (i.e.
```

```

* answer was last computed)
* @return answeredInRound is the round ID of the round in which the answer
* was computed.
* (Only some AggregatorV3Interface implementations return meaningful values)
* @dev Note that answer and updatedAt may change between queries.
*/
function latestRoundData()
    public
    view
    virtual
    override
    returns (
        uint80 roundId,    //聚合器进行数据聚合的轮次ID
        int256 answer,     //最终聚合得到的价格数据
        uint256 startedAt, //聚合开始的时间戳
        uint256 updatedAt, //聚合结束的时间戳(算出最终answer并更新的时间戳)
        uint80 answeredInRound //answer被计算出来时的轮次ID
    )
{
    Phase memory current = currentPhase; // cache storage reads

    (
        uint80 roundId,
        int256 answer,
        uint256 startedAt,
        uint256 updatedAt,
        uint80 ansIn
    ) = current.aggregator.latestRoundData(); //从current.aggregator中获取返回值

    return addPhaseIds(roundId, answer, startedAt, updatedAt, ansIn, current.id);
}

```

latestRoundData函数在AggregatorProxy父合约中的实现

父合约AggregatorProxy中latestRoundData函数中定义了一个Phase结构体current，并从currentPhase中接收值，这里currentPhase是该合约的storage存储类型数据，也就是全局状态变量。current从currentPhase中接收到值后，roundId、answer、startedAt、updatedAt和ansIn这5个参数就从current.aggregator的latestRoundData()函数中获取返回值，因此我们可以判断current.aggregator是负责聚合价格数据的聚合器合约。

当前面的步骤都完成后，最后latestRoundData()函数会调用addPhaseIds对前面获取的5个返回参数进行二次处理并最终返回。

2.2.1 Phase结构体


```

struct Phase {
    uint16 id; //该Phase的id号
    AggregatorV2V3Interface aggregator; //该Phase的聚合器合约(接口实例化)
}
Phase private currentPhase; //用于存储最新的Phase
AggregatorV2V3Interface public proposedAggregator; //用于提议新的聚合器合约

//Phase结构体中id到对应聚合器合约的映射 phaseAggregators, 方便根据id查找对应阶段的聚合器
mapping(uint16 => AggregatorV2V3Interface) public phaseAggregators;

```

Phase结构体定义

先来看看Phase结构体的定义，Phase结构体内定义了一个id和一个AggregatorV2V3Interface合约接口aggregator，在实例化Phase结构体的时候项目方会传入一个聚合器合约的地址和其对应的id，用于标识某个Phase的id及该Phase对应使用的聚合器合约，现在我们暂且把Phase理解为阶段。当把一个聚合器合约地址传给aggregator时aggregator会用AggregatorV2V3Interface接口对合约进行实例化，用以表示这个地址所对应的运行在以太坊虚拟机中的合约对象。

定义完Phase结构体后紧接着用这个结构体声明一个全局私有变量currentPhase，用于标识当前最新阶段的id和该阶段的聚合器合约。

同时该合约还定义了两个变量，一个是聚合器接口类型proposedAggregator，这个变量主要是用于提议新的聚合器合约；另一个是mapping类型的phaseAggregators，这个变量主要是存储从阶段id到该阶段所对应的聚合器合约的映射，便于根据id查找对应阶段的聚合器合约。

```

/*
 * Internal
 */

function setAggregator(address _aggregator)
    internal
{
    uint16 id = currentPhase.id + 1; //id自增
    //更新currentPhase
    currentPhase = Phase(id, AggregatorV2V3Interface(_aggregator));
    //将最新阶段的信息存入phaseAggregators
    phaseAggregators[id] = AggregatorV2V3Interface(_aggregator);
}

```

setAggregator函数实现

合约中用一个setAggregator函数来设置currentPhase，传入的参数是当前阶段的聚合器合约地址。每当需要设置一个新的currentPhase时，setAggregator函数都会将更新前的currentPhase的id加1，然后连同传入的参数aggregator一起将值传给currentPhase以对currentPhase进行更新。最后新生成的currentPhase的相关信息会被存进phaseAggregators映射中。

从这里我们可以得知每个阶段的id是递增的。

```

/**
 * @notice Allows the owner to propose a new address for the aggregator
 * @param _aggregator The new address for the aggregator contract

```

```

    */
    //提议一个聚合器合约
    function proposeAggregator(address _aggregator)
        external
        onlyOwner()
    {
        proposedAggregator = AggregatorV2V3Interface(_aggregator);
    }

    /**
     * @notice Allows the owner to confirm and change the address
     * to the proposed aggregator
     * @dev Reverts if the given address doesn't match what was previously
     * proposed
     * @param _aggregator The new address for the aggregator contract
     */
    //确认在最新的阶段使用已经提议的聚合器合约
    function confirmAggregator(address _aggregator)
        external
        onlyOwner()
    {
        require(_aggregator == address(proposedAggregator), "Invalid proposed aggregator"); //判断传入的要确认的地址是否是已经提议的聚合器合约地址，不是的话则报错
        delete proposedAggregator; //地址确认完成，删除提议的聚合器合约
        setAggregator(_aggregator); //确认提议的聚合器合约，生成对应的currentPhase
    }

```

进行聚合器合约替换的部分代码

我们可以看到前面的setAggregator函数的可见性是internal，这也就意味着这个函数会在合约里的其它地方被调用。对合约进行查找后，能够发现proposeAggregator和confirmAggregator函数与设置新阶段的聚合器合约有关系。

proposeAggregator函数用于提议一个新的聚合器合约，confirmAggregator合约则是对这个提议进行确认，我们可以看到当对提议进行确认时，储存提议的聚合器合约对象会被删除，然后再调用setAggregator函数对currentPhase进行更新，currentPhase中的aggregator字段就存储着已确认的最新的聚合器合约对象。

proposeAggregator和confirmAggregator两个函数的可见性都是external，是因为这两个函数是用于被外部调用的，当价格参考合约的管理者想要用新的聚合器合约去聚合价格数据时，就用从外部调用这两个函数。

就目前掌握的信息而言，还不知道Phase中的Id是否与roundId有联系，也不确定是否每一轮数据聚合都对应着一个新的Phase，是否每一轮数据聚合都会换一个聚合器合约，因此我们可以去etherscan查看currentPhase当前最新的状态辅助我们进行判断。

[Code](#)[Read Contract](#)[Write Contract](#)

11. latestTimestamp

12. owner

13. phaseAggregators

14. phaseId

returns the current phase's ID.

4 uint16

currentPhase的ID

可以看到BTC/ETH价格参考合约中currentPhase当前的id是4，也就是说从该合约上链到现在其用于聚合数据的聚合器合约只被更换了4次，因此我们可以判断聚合器合约的更换与数据聚合轮次没有联系，可能是当Chainlink官方需要对聚合器合约的业务进行升级的时候才会替换聚合器合约。所以相比于将Phase翻译为阶段，这里翻译成版本会更合适。

2.2.2 addPhaseIds函数

2.2.1部分主要是对BTC/ETH价格参考合约中的Phase结构体和currentPhase变量及其相应的操作函数进行分析描述，而在价格参考合约的latestRoundData函数中在return时还调用了addPhaseIds函数对返回值进行了处理，2.2.2部分会对addPhaseIds函数进行展开。

```
function addPhaseIds(
    uint80 roundId,    //聚合器进行数据聚合的轮次ID
    int256 answer,     //最终聚合得到的价格数据
    uint256 startedAt, //聚合开始的时间戳
    uint256 updatedAt, //聚合结束的时间戳(算出最终answer并更新的时间戳)
    uint80 answeredInRound, //answer被计算出来时的轮次ID
    uint16 phaseId     //currentPhase中的id，与聚合器聚合数据的轮次id不同
)
internal
view
returns (uint80, int256, uint256, uint256, uint80)
{
    return (
        addPhase(phaseId, uint64(roundId)),
        answer,
        startedAt,
```

```

        updatedAt,
        addPhase(phaseId, uint64(answeredInRound))
    );
}

```

addPhaseIds函数实现

在2.2开头父合约AggregatorProxy的latestRoundData函数中我们可以看到roundId、answer、startedAt、updatedAt和ansIn这5个参数的值是在currentPhase结构体中的聚合器合约里获取的，而addPhaseIds函数的作用则是对这些参数进行二次加工，并且currentPhase的id也被作为入参传了进去。

从上面addPhaseIds函数的具体实现中我们可以看到answer、startedAt和updatedAt三个参数被原封不动地return了回去，只有roundId和answeredInRound以及phaseId三个参数被addPhase函数进行了处理。也就是说，最后返回给用户的latestRoundData函数中的roundId是addPhase(phaseId, uint64(roundId))，answeredInRound则是addPhase(phaseId, uint64(answeredInRound))。

```

function addPhase(
    uint16 _phase,
    uint64 _originalId
)
    internal
    view
    returns (uint80)
{
    //用位运算拼接phaseId和roundId
    return uint80(uint256(_phase) << PHASE_OFFSET | _originalId); //位运算
}

```

addPhase函数实现

所以再来看看addPhase函数，该函数的入参是phaseId以及roundId或answerInRound，其中传入roundId或answerInRound只截取了右边的64位。addPhase函数在对两个入参进行位运算后将结果返回，其中<<是左偏移运算，a<<b的意思就是将a的二进制全部向左偏移b位，超出左边的部分直接舍弃，右边空出的部分补0，a<<b的本质是通过将a乘以2的b次方来对a向左进行偏移，不过这是二进制运算的知识点这里不做展开；|是按位或运算，比如二进制11001和10011的|运算就是两个数从左到右分别按位进行或运算，当两个数字对应位置有任意一个1时，则结果中对应的位置为1，因此这两个数的|运算结果为11011。

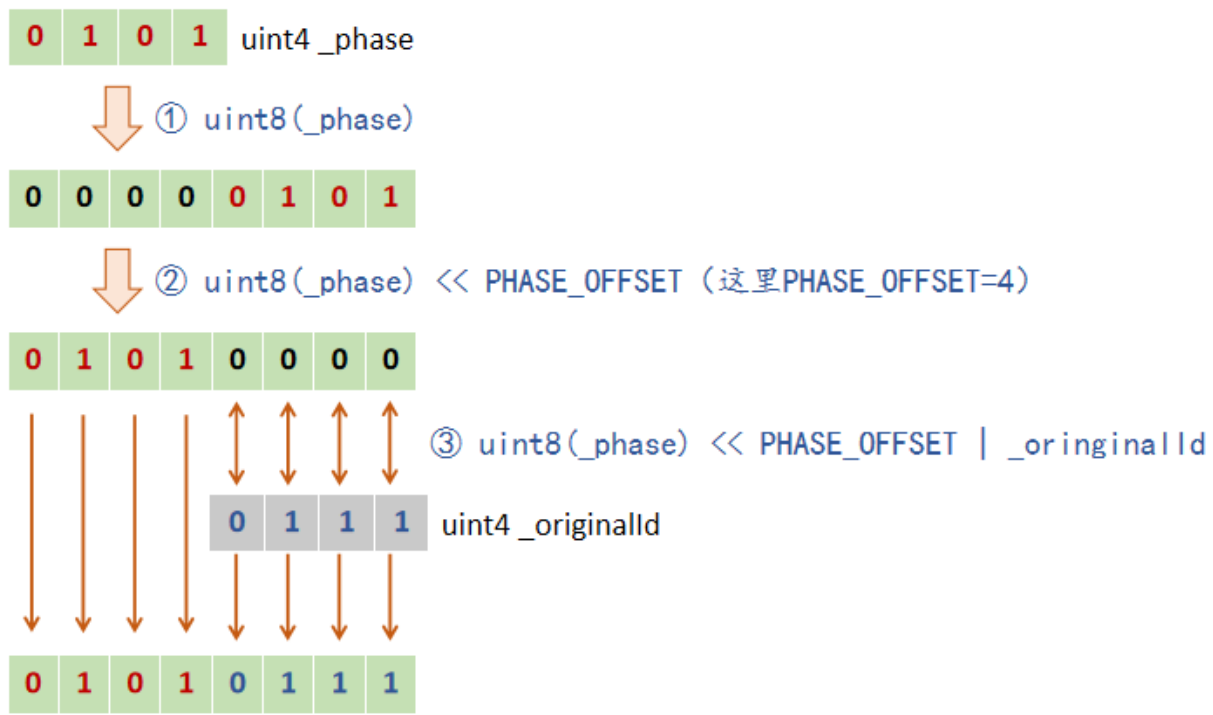
因为<<运算符的优先级高于|，所以addPhase函数中是先将phase向左偏移PHASE_OFFSET位然后再与originalId按位进行或运算。

```
uint256 constant private PHASE_OFFSET = 64;
```

PHASE_OFFSET变量

合约中PHASE_OFFSET被设置为64，所以当phase向左偏移后右边会补64个0(phase原长度为16为，因此需要将它转换成256位的uint256才有位置进行偏移)，而originalId的长度也为64位，当originalId与phase右边的64个0进行或运算时结果就是originalId本身，因此addPhase函数中位运算的意义就是将长度位64字节的originalId拼接在长度位16的phaseId的右边，最后返回一个长度为80个字节的uint80数据。

为了更加清晰地表达这个过程，可以简单地画个图(长度都进行了缩减)：



addPhase位运算示意图

当位运算结束后，`uint256(_phase) << PHASE_OFFSET | _originalId`的长度为256，而有实际意义的长度为80，所以最后再强制转换成`uint80`。(从`uint256`转为`uint80`会保留右边80位)

所以最后返回给`latestRoundData()`函数的`roundId`由`phaseId`和聚合器合约返回的`roundId`的右边64位拼接而成，`answerInRound`则是将`PhaseId`和聚合器合约返回的`answerInRound`的右边64位进行拼接后获得。

为了验证上面关于`addPhase()`函数分析的正确性，我们可以看看当前最新的`latestRoundData()`返回的`roundId`：

10. latestRoundData

get data about the latest round. Consumers are encouraged to check that they're receiving fresh data by ins implementations of AggregatorV3Interface have slightly different semantics for some of the return values. C they can properly handle return data from all of them.

Note that answer and updatedAt may change between queries.

Query

↳ `roundId uint80, answer int256, startedAt uint256, updatedAt uint256, answeredInRound uint80`

[latestRoundData method Response]

```
>> roundId uint80: 73786976294838207617
>> answer int256: 13625530630000000000
>> startedAt uint256: 1671044267
>> updatedAt uint256: 1671044267
>> answeredInRound uint80: 73786976294838207617
```

latestRoundData函数返回的roundId值

最新一轮价格聚合的roundId为73786976294838207617，而前面我们已经知道currentPhase.Id为4，4的二进制表达左移64位后的十进制为73786976294838206464，因此我们可以得知roundId是currentPhase.Id左移64位后加上1153获得，则1153即为聚合器合约返回的roundId，符合addPhase函数应该返回的结果。

同时从上面latestRoundData函数返回的值中可以看到roundId和answeredInRound是一样的，这也就意味着价格聚合器从开始进行价格聚合到最后得出结果都在同一个轮次内。

当然addPhase函数的分析是否正确还得查看聚合器合约返回的roundId是否真为1153，因此接着往下看。

2.2.3 currentPhase.aggregator 聚合器合约

从2.2开头的代码中我们可以得知最终核心的价格数据是从聚合器合约中的latestRoundData函数中获得的，价格参考合约的主要作用是可以选择不同的聚合器合约来提供价格数据，以及对从聚合器合约获得的价格数据进行处理等，因此我们还需要分析一下聚合器合约的具体实现。

2. aggregator

returns the current phase's aggregator address.

0x81076d6Ff2620Ea9Dd7bA9c1015f0d09A3A732E6 address

聚合器合约地址

在etherscan上可以直接看到价格合约中的currentPhase.aggregator地址为0x81076d6Ff2620Ea9Dd7bA9c1015f0d09A3A732E6，我们根据这个地址来查看聚合器合约的代码以及合约最新的状态。

Transactions

Internal Txns

Erc20 Token Txns

Contract ✓

Events

Analytics

Code

Read Contract

Write Contract

✓ **Contract Source Code Verified** (Exact Match)

Contract Name:

AccessControlledOffchainAggregator

Compiler Version

v0.7.6+commit.7338295f

聚合器合约名称

聚合器合约名称为AccessControllerOffchainAggregator，光看名称就可以判断出这个合约和EACAggregatorProxy合约一样都是封装在业务合约外面用于权限判断的代理合约。

2.3 AccessControlledOffchainAggregator合约

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.1;

import "./OffchainAggregator.sol";
import "./SimpleReadAccessController.sol";

/**
 * @notice Wrapper of OffchainAggregator which checks read access on Aggregator-
 * interface methods
 */
contract AccessControlledOffchainAggregator is OffchainAggregator,
SimpleReadAccessController {

    /// @inheritdoc OffchainAggregator
    function latestRoundData()
        public
        override
        view
        checkAccess() //在SimpleWriteAccessController合约中实现
        returns (
            uint80 roundId,
            int256 answer,
            uint256 startedAt,
            uint256 updatedAt,
            uint80 answeredInRound
        )
    {
        return super.latestRoundData();
    }
}
```

AccessControlledOffchainAggregator合约部分代码

AccessControlledOffchainAggregator合约继承了父合约OffchainAggregator和SimpleReadAccessController，在2.1中我们已经得知SimpleReadAccessController合约继承了其父合约SimpleWriteAccessController，主要的作用就是进行访问控制，判断msg.sender是否在合约的白名单内。

而AccessControlledOffchainAggregator合约中的latestRoundData函数则是调用了其父合约OffchainAggregator的同名函数来获取价格数据，并在调用前用修饰器checkAccess判断调用该函数的合约地址是否在权限控制合约的白名单内。

10. hasAccess

_user (address)

0xdeb288F737066589598e9214E782fa5A8eD689e8

_calldata (bytes)

0x0

Query

↳ bool

[hasAccess(address bytes) method Response]

➤ bool: true

SimpleWriteAccessController 合约中的权限判断函数

在etherscan中我们也可以看到当调用权限控制合约的hasAccess函数判断价格参考合约的权限时返回true，只有在白名单内的合约才能从聚合器合约获取价格数据。

2.4 OffchainAggregator父合约

OffchainAggregator合约就是从Chainlink网络获取价格数据的聚合器合约。

```
/**
 * @notice aggregator details for the most recently transmitted report
 * @return roundId aggregator round of latest report (NOT OCR round)
 * @return answer median of latest report
 * @return startedAt timestamp of block containing latest report
 * @return updatedAt timestamp of block containing latest report
 * @return answeredInRound aggregator round of latest report
 */
function latestRoundData()
    public
    override
    view
    virtual
    returns (
        uint80 roundId,
        int256 answer,
        uint256 startedAt,
        uint256 updatedAt,
        uint80 answeredInRound
    )
```



```

{
    roundId = s_hotVars.latestAggregatorRoundId;

    // Skipped for compatability with existing FluxAggregator in which
    latestRoundData never reverts.
    // require(roundId != 0, V3_NO_DATA_ERROR);

    //根据roundId查找相应的价格数据链下报告(也就是Transmission)
    Transmission memory transmission = s_transmissions[uint32(roundId)];
    return (
        roundId,
        transmission.answer, //从transmission中获取最后的answer
        transmission.timestamp,
        transmission.timestamp,
        roundId //直接将roundId的值返回给answeredInRound
    );
}

```

OffchainAggregator合约中的latestRoundData函数

从上面的函数中我们可以看到roundId是从s_hotVars结构体中获得的,而其他价格数据则是从transmission结构体和s_transmissions映射中获得, 因此接下来我们需要分析上面三个变量。

值得一提的是这里的latestRoundData函数在最后返回值的时候直接将roundId作为answeredInRound返回, 因此我们在2.2.2节的最后在etherscan中查看返回值时roundId和answeredInRound才会一致。

14. latestRoundData

aggregator details for the most recently transmitted report

Query

↳ roundId uint80, answer int256, startedAt uint256, updatedAt uint256, answeredInRound uint80

[latestRoundData method Response]

```

>> roundId uint80: 1153
>> answer int256: 13625530630000000000
>> startedAt uint256: 1671044267
>> updatedAt uint256: 1671044267
>> answeredInRound uint80: 1153

```

OffchainAggregator合约中的latestRoundData函数返回值

在etherscan查询该合约的latestRoundData函数返回值后可以看到roundId和answeredInRound返回值一致。同时roundId为1153, 证明2.2.1和2.2.2关于addPhase函数的分析正确。

2.4.1 s_hotVars结构体

```
// Storing these fields used on the hot path in a HotVars variable reduces the
// retrieval of all of them to a single SLOAD. If any further fields are
// added, make sure that storage of the struct still takes at most 32 bytes.
struct HotVars {
    // Provides 128 bits of security against 2nd pre-image attacks, but only
    // 64 bits against collisions. This is acceptable, since a malicious owner has
    // easier way of messing up the protocol than to find hash collisions.
    // 最新的数据聚合配置参数，出于安全考虑用于对抗碰撞
    bytes16 latestConfigDigest;
    // 最新一轮所处的阶段和轮次，前32字节代表阶段，后8个字节用于表明轮次
    uint40 latestEpochAndRound; // 32 most sig bits for epoch, 8 least sig bits for
round
    // Current bound assumed on number of faulty/dishonest oracles participating
    // in the protocol, this value is referred to as f in the design
    // 预言机网络中能容忍的不诚实节点或错误节点的最大数量阈值
    uint8 threshold;
    // Chainlink Aggregators expose a roundId to consumers. The offchain reporting
    // protocol does not use this id anywhere. We increment it whenever a new
    // transmission is made to provide callers with contiguous ids for successive
    // reports.
    // 最新一轮数据聚合的轮次ID
    uint32 latestAggregatorRoundId;
}
HotVars internal s_hotVars; //HotVars结构体的实例化s_hotVars
```

HotVars结构体定义

从HotVars结构体的定义中我们可以看到几个变量，其中最重要的就是latestAggregatorRoundId，该变量表示的是最新一轮数据聚合所处的轮次，latestRoundData在获取最新的价格数据时就是根据这里的轮次ID获取。

s_hotVars是HotVars结构体的实例化。

```
/**
 * @notice immediately requests a new round
 * @return the aggregatorRoundId of the next round. Note: The report for this
round may have been
 * transmitted (but not yet mined) *before* requestNewRound() was even called.
There is *no*
 * guarantee of causality between the request and the report at aggregatorRoundId.
 */
function requestNewRound() external returns (uint80) {
    require(msg.sender == owner || s_requesterAccessController.hasAccess(msg.sender,
msg.data),
    "Only owner&requester can call"); //权限控制，只有管理者或聚合器合约拥有者才能发起新的
轮次请求

    HotVars memory hotVars = s_hotVars;
```

```
emit RoundRequested( //广播新的轮次需求事件，预言机在接收到新的事件后就开始聚合数据
    msg.sender,
    hotVars.latestConfigDigest,
    uint32(s_hotVars.latestEpochAndRound >> 8),
    uint8(s_hotVars.latestEpochAndRound)
);
return hotVars.latestAggregatorRoundId + 1; //返回新请求的id数
}
```

requestNewRound函数

当预言机网络需要发起一轮新的数据聚合时就会从链下调用requestNewRound函数，该函数会向外界广播新轮次的请求事件，然后返回s_hotVars结构体中latestAggregatorRoundId+1，而当链外的预言机在监听到新的事件时就会开始新一轮的数据聚合。

需要注意的是当requestNewRound函数被调用后latestAggregatorRoundId实际上还没有+1，只有当链下完成价格数据聚合并将链下报告上链后该值才会更新。

我们可以看该函数在开头判断了调用者是否有发起新轮次请求的权限，用的还是hasAccess函数，也就是说判断权限的逻辑一致，但是我现在想知道这里用的权限控制合约和之前是不是同一个合约。

为了便于区分，我将该函数所使用的权限控制合约命名为“新轮次请求权限判断合约”，2.3中的权限控制合约命名为“价格参考权限控制合约”。(从实现层面来看本质上这两个合约都是SimpleWriteAccessController合约)

23. requesterAccessController

address of the requester access controller contract

0x641B698aD1C6E503470520B0EeCb472c0589dfE6 address

新轮次请求权限判断合约地址

在etherscan中可以查看到新轮次请求权限判断合约地址为0x641B698aD1C6E503470520B0EeCb472c0589dfE6，而2.3节中的价格参考权限控制合约是子合约AccessControlledOffchainAggregator直接继承然后才被部署到链上的，也就是说聚合器子合约中的权限控制合约地址与其自身一致，为0x81076d6Ff2620Ea9Dd7bA9c1015f0d09A3A732E6。

因此聚合器的父合约和子合约所使用的权限控制合约虽然判断逻辑一致，但是却是两份不同的合约，其权限控制的目的也不一致。

2. hasAccess

_user (address)

0xdeb288F737066589598e9214E782fa5A8eD689e8

<input> (bytes)

0x0

Query

↳ bool

[hasAccess(address,bytes) method Response]

>> bool: false

新一轮请求权限判断合约中的权限判断

此时用新一轮请求权限判断合约的权限判断函数查询价格参考合约的权限时返回的是false(2.3中返回的是true)。

2.4.2 s_transmissions结构体映射及transmit函数

在2.4的latestRoundData函数中可以看到价格数据会被存放在Transmission结构体中，而每一轮得到的Transmission结构体都会被存入s_transmissions结构体数组，并以roundId为索引。

```
// Transmission records the median answer from the transmit transaction at
// time timestamp
struct Transmission {
    int192 answer; // 192 bits ought to be enough for anyone
    uint64 timestamp; //时间戳
}
//存放每一轮的Transmission，以roundId为索引
mapping(uint32 /* aggregator round ID */ => Transmission) internal
s_transmissions;
```

Transmission和s_transmissions定义

在Transmission结构体中只定义了两个变量，一个是answer最终的价格数据answer，另一个则是获得这个数据的时间戳。而s_transmissions则是从roundId到Transmission的mapping映射，可以根据id找到对应轮次的价格。

```
/**
 * @notice transmit is called to post a new report to the contract
```

```

    * @param _report serialized report, which the signatures are signing. See parsing
    code below for format. The ith element of the observers component must be the index
    in s_signers of the address for the ith signature
    * @param _rs ith element is the R components of the ith signature on report. Must
    have at most maxNumOracles entries
    * @param _ss ith element is the S components of the ith signature on report. Must
    have at most maxNumOracles entries
    * @param _rawVs ith element is the the V component of the ith signature
    */
function transmit(
    // NOTE: If these parameters are changed, expectedMsgDataLength and/or
    // TRANSMIT_MSGDATA_CONSTANT_LENGTH_COMPONENT need to be changed accordingly
    // 链下报告内容
    bytes calldata _report,
    // 报告附带的预言机节点的签名
    bytes32[] calldata _rs, bytes32[] calldata _ss, bytes32 _rawVs // signatures
)
    external
{
    // 获取当前剩余可用的gas，一般来说gasleft()函数被放在函数的开头和结尾用于判断整个函数使用了多
    少gas
    // 这里是为了计算报告提交者提交报告所使用的gas，然后在函数结尾根据gas消耗返还提交者gas费
    uint256 initialGas = gasleft(); // This line must come first
    // Make sure the transmit message-length matches the inputs. Otherwise, the
    // transmitter could append an arbitrarily long (up to gas-block limit)
    // string of 0 bytes, which we would reimburse at a rate of 16 gas/byte, but
    // which would only cost the transmitter 4 gas/byte. (Appendix G of the
    // yellow paper, p. 25, for G_txdatazero and EIP 2028 for G_txdatanonzero.)
    // This could amount to reimbursement profit of 36 million gas, given a 3MB
    // zero tail.
    // 报告长度合理性判断，包括签名长度
    // 因为在最后返还gas费时有一部分是根据传入参数长度进行偿还，偿还价格为16gas/字节
    // 而提交者这部分的实际花费为4gas/字节，因此为避免提交者恶意套取gas的偿还费用需要下列判断
    require(msg.data.length == expectedMsgDataLength(_report, _rs, _ss),
        "transmit message too long");
    // 定义一个ReportData结构体r
    ReportData memory r; // Relieves stack pressure
    {
        r.hotVars = s_hotVars; // cache read from storage 从s_hotVars直接获取数据初始化

        bytes32 rawObservers; // 定义一个bytes32类型的rawObservers
        // 对链下报告进行解码成报告内容，数据提供节点索引和价格数据集
        (r.rawReportContext, rawObservers, r.observations) = abi.decode(
            _report, (bytes32, bytes32, int192[]))
    };
    // 链下报告内容包括11字节的0字符填充，16字节本轮聚合配置参数，4字节报告所处阶段及1字节报告所
    处轮次
    // rawReportContext consists of:
    // 11-byte zero padding
    // 16-byte configDigest
    // 4-byte epoch
    // 1-byte round

```

```

// 将报告内容左移88位获得聚合配置参数(将左边11字节的0填充移除以截取配置参数)
bytes16 configDigest = bytes16(r.rawReportContext << 88);
// 判断报告的配置参数和hotVars的配置参数是否一致, 不一致则停止上传报告
require(
    r.hotVars.latestConfigDigest == configDigest,
    "configDigest mismatch"
);

// 截取报告的阶段和轮次信息
uint40 epochAndRound = uint40(uint256(r.rawReportContext));

// direct numerical comparison works here, because
//
// ((e,r) <= (e',r')) implies (epochAndRound <= epochAndRound')
//
// because alphabetic ordering implies e <= e', and if e = e', then r<=r',
// so e*256+r <= e'*256+r', because r, r' < 256
// 判断该报告轮次是否是最新为最新轮次
require(r.hotVars.latestEpochAndRound < epochAndRound, "stale report");
// 判断签名数量是否大于最大可容忍不诚实节点的数量
require(_rs.length > r.hotVars.threshold, "not enough signatures");
// 判断签名数量是否小于最大的预言机节点数量(预言机节点数量在父合约中被定义为31个)
require(_rs.length <= maxNumOracles, "too many signatures");
// 判断签名的R,S组件数量是否匹配
require(_ss.length == _rs.length, "signatures out of registration");
// 判断价格数据集的数据数量是否小于最大的预言机节点数量
require(r.observations.length <= maxNumOracles,
    "num observations out of bounds");
// 判断价格数据集的数据数量是否大于2倍的最大可容忍不诚实节点的数量
require(r.observations.length > 2 * r.hotVars.threshold,
    "too few values to trust median");

// 获取签名集合的V组件集合并传给r结构体
// Copy signature parities in bytes32 _rawVs to bytes r.v
r.vs = new bytes(_rs.length);
for (uint8 i = 0; i < _rs.length; i++) {
    r.vs[i] = _rawVs[i];
}

// 获取价格数据提供节点的索引数据并传给r结构体
// Copy observer identities in bytes32 rawObservers to bytes r.observers
r.observers = new bytes(r.observations.length); //有多少价格数据就有多少数据提供
节点

bool[maxNumOracles] memory seen; //辅助bool数组, 用于判断有无出现重复的数据提供节点
for (uint8 i = 0; i < r.observations.length; i++) {
    uint8 observerIdx = uint8(rawObservers[i]); //获取提供第i个价格数据的节点索引
    require(!seen[observerIdx], "observer index repeated"); //有重复节点则停止报告
    seen[observerIdx] = true; //标记某个节点在本轮报告中提供了价格数据
    r.observers[i] = rawObservers[i]; //将索引信息复制到结构体r中
}

```

```

    oracle memory transmitter = s_oracles[msg.sender];    //获取此次报告的提交者
    require( // Check that sender is authorized to report
        transmitter.role == Role.Transmitter && //检查该提交者是否有提交报告的权限
        msg.sender == s_transmitters[transmitter.index],    //检查提交者的索引是否正确
        "unauthorized transmitter"
    );
    // 获取此次报告的阶段及轮次并传给r结构体
    // record epochAndRound here, so that we don't have to carry the local
    // variable in transmit. The change is reverted if something fails later.
    r.hotVars.latestEpochAndRound = epochAndRound;
}

{ // Verify signatures attached to report
    bytes32 h = keccak256(_report);    //加密报告
    bool[maxNumOracles] memory signed;    //辅助数组，用于判断是否有重复签名

    oracle memory o;
    for (uint i = 0; i < _rs.length; i++) {
        //利用预言机节点各自私钥对_report签名后生成的V、S、R组件生成私钥对应的公钥(密码学知识)
        address signer = ecrecover(h, uint8(r.vs[i])+27, _rs[i], _ss[i]);
        o = s_oracles[signer];
        //判断签名生成的公钥地址是否为s_oracles中的签名者(判断有无签名权限)
        require(o.role == Role.Signer, "address not authorized to sign");
        require(!signed[o.index], "non-unique signature"); //判断有无重复签名
        signed[o.index] = true; //标记
    }
}

{ // Check the report contents, and record the result
    // 检查价格数据集observations中的价格数据是否已经按照从小到大排序，方便后面取中位数作为结果
    for (uint i = 0; i < r.observations.length - 1; i++) {
        bool inOrder = r.observations[i] <= r.observations[i+1];
        require(inOrder, "observations not sorted");
    }
    // 取价格数据集集合中的中位数作为该轮BTC/ETH价格聚合中的最终结果
    int192 median = r.observations[r.observations.length/2];
    // 最终价格需要在预设的合理区间内
    require(minAnswer <= median && median <= maxAnswer, "median is out of min-max
range");
    // 2.4.1的requestNewRound函数发出新价格数据请求时没有变更latestAggregatorRoundId
    // 获取最终价格数据后才将latestAggregatorRoundId变量+1
    r.hotVars.latestAggregatorRoundId++;
    // 将结果存入s_transmissions映射中
    s_transmissions[r.hotVars.latestAggregatorRoundId] =
        Transmission(median, uint64(block.timestamp));
    //广播新一轮聚合数据摘要
    emit NewTransmission(
        r.hotVars.latestAggregatorRoundId,
        median,
        msg.sender,
        r.observations,

```



```

        r.observers,
        r.rawReportContext
    );
    // Emit these for backwards compatability with offchain consumers
    // that only support legacy events
    // 广播新一轮次ID
    emit NewRound(
        r.hotVars.latestAggregatorRoundId,
        address(0x0), // use zero address since we don't have anybody "starting" the
round here
        block.timestamp
    );
    // 广播新价格数据
    emit AnswerUpdated(
        median,
        r.hotVars.latestAggregatorRoundId,
        block.timestamp
    );
    // 数据校验
    validateAnswer(r.hotVars.latestAggregatorRoundId, median);
}
s_hotVars = r.hotVars; //更新s_hotVars
assert(initialGas < maxUint32); //断言
// 为参与价格聚合的节点发放link代币作为激励(实际发放过程更复杂一些, 在父合约实现)
// 并且偿还报告提交者的gas消耗, initialGas在transmit函数开头通过gasleft()获得
reimburseAndRewardOracles(uint32(initialGas), r.observers);
}

```

transmit函数(链下报告上链函数)

s_transmissions映射和s_hotVars会在transmit函数中被赋值, transmit函数就是预言机网络将链下的价格数据报告提交上链所要调用的函数, 可以理解为预言机网络和链上预言机生态的数据接口。transmit函数的传入参数为价格数据报告和提交数据的预言机节点的签名集合, 报告中的数据会被解码提取并存入到s_transmissions映射中, 而签名部分则会被用来做报告合理性的验证。

由于transmit函数代码较长, 下面先对代码进行拆分然后再分析。

2.4.3 transmit函数片段1(gasleft和expectedMsgDataLength)

```

/**
 * @notice transmit is called to post a new report to the contract
 * @param _report serialized report, which the signatures are signing. See parsing
code      below for format. The ith element of the observers component must be the
index in   s_signers of the address for the ith signature
 * @param _rs ith element is the R components of the ith signature on report. Must
have       at most maxNumOracles entries
 * @param _ss ith element is the S components of the ith signature on report. Must
have       at most maxNumOracles entries
 * @param _rawVs ith element is the the v component of the ith signature
 */
function transmit(
    // NOTE: If these parameters are changed, expectedMsgDataLength and/or

```



```

// TRANSMIT_MSGDATA_CONSTANT_LENGTH_COMPONENT need to be changed accordingly
// 链下报告内容
bytes calldata _report,
// 报告附带的预言机节点的签名
bytes32[] calldata _rs, bytes32[] calldata _ss, bytes32 _rawVs // signatures
)
external
{
    // 获取当前剩余可用的gas，一般来说gasleft()函数被放在函数的开头和结尾用于判断整个函数使用了多
    少gas
    // 这里是为了计算报告提交者提交报告所使用的gas，然后在函数结尾根据gas消耗返还提交者gas费
    uint256 initialGas = gasleft(); // This line must come first
    // Make sure the transmit message-length matches the inputs. Otherwise, the
    // transmitter could append an arbitrarily long (up to gas-block limit)
    // string of 0 bytes, which we would reimburse at a rate of 16 gas/byte, but
    // which would only cost the transmitter 4 gas/byte. (Appendix G of the
    // yellow paper, p. 25, for G_txdatazero and EIP 2028 for G_txdataanonzero.)
    // This could amount to reimbursement profit of 36 million gas, given a 3MB
    // zero tail.
    // 报告长度合理性判断，包括签名长度
    // 因为在最后返还gas费时有一部分是根据传入参数长度进行偿还，偿还价格为16gas/字节
    // 而提交者这部分的实际花费为4gas/字节，因此为避免提交者恶意套取gas的偿还费用需要下列判断
    require(msg.data.length == expectedMsgDataLength(_report, _rs, _ss),
        "transmit message too long");
    .....
}

```

transmit函数片段1

transmit函数的入参主要分为两个部分，第一部分是包含价格数据的report，第二部分则是用于签名验证的可组成签名集合的签名组件集合rs、ss和rawVs，接下来我们分析的重点会侧重report这一块。

函数开头用gasleft()函数获取整个函数当前仍可使用的gas，在这里的目的主要是用于计算报告提交者提交报告所花费的gas数量并以此偿还提交者gas消耗(相当于Chainlink官方出发送报告的手续费)。接着函数用require判断传入的参数的长度和有效长度是否一致，因为chainlink是按照传入的数据长度来进行gas补偿(根据报告长度超额补充)，因此为了避免有节点发送无效输入来恶意获取补偿而进行这项判断。

判断过程中所使用的函数为expectedMsgDataLength。

```

function expectedMsgDataLength(
    bytes calldata _report, bytes32[] calldata _rs, bytes32[] calldata _ss
) private pure returns (uint256 length)
{
    // calldata will never be big enough to make this overflow
    return uint256(TRANSMIT_MSGDATA_CONSTANT_LENGTH_COMPONENT) +
        _report.length + // one byte pure entry in _report
        _rs.length * 32 + // 32 bytes per entry in _rs
        _ss.length * 32 + // 32 bytes per entry in _ss
        0; // placeholder
}

//TRANSMIT_MSGDATA_CONSTANT_LENGTH_COMPONENT定义

```

```
// The constant-length components of the msg.data sent to transmit.
// See the "If we wanted to call sam" example on for example reasoning
// https://solidity.readthedocs.io/en/v0.7.2/abi-spec.html
uint16 private constant TRANSMIT_MSGDATA_CONSTANT_LENGTH_COMPONENT =
    4 + // function selector
    32 + // word containing start location of abiencoded _report value
    32 + // word containing location start of abiencoded _rs value
    32 + // word containing start location of abiencoded _ss value
    32 + // _rawVs value
    32 + // word containing length of _report
    32 + // word containing length _rs
    32 + // word containing length of _ss
    0; // placeholder
```

expectedMsgDataLength函数实现及相关变量定义

在expectedMsgDataLength函数中可以看到有效输入的长度实际上就是发起交易时入参为report、rs、ss和rawVs的abi编码长度(abi编码为16进制串, 包括函数选择器+变长数据位置和长度标识+边长数据内容, 这是以太坊底层和solidity的知识点, 这里不过多赘述), 目的就是防止交易发起者在msg.data中加入除了上述4个入参之外的其他参数。这里report中的每个实体长度为1个字节, 签名rs和ss中的每个实体长度为32个字节。

2.4.4 transmit函数片段2(ReportData、configDigest和s_oracles)

```
//定义一个ReportData结构体r
ReportData memory r; // Relieves stack pressure
{
    r.hotVars = s_hotVars; // cache read from storage 从s_hotVars直接获取数据初始化

    bytes32 rawObservers; // 定义一个bytes32类型的rawObservers
    // 对链下报告进行解码成报告内容, 数据提供节点索引和价格数据集
    (r.rawReportContext, rawObservers, r.observations) = abi.decode(
        _report, (bytes32, bytes32, int192[]))
    );
    // 链下报告内容包括11字节的0字符填充, 16字节本轮聚合配置参数, 4字节报告所处阶段及1字节报告所处轮次
    // rawReportContext consists of:
    // 11-byte zero padding
    // 16-byte configDigest
    // 4-byte epoch
    // 1-byte round

    // 将报告内容左移88位获得聚合配置参数(将左边11字节的0填充移除以截取配置参数)
    bytes16 configDigest = bytes16(r.rawReportContext << 88);
    // 判断报告的配置参数和hotVars的配置参数是否一致, 不一致则停止上传报告
    require(
        r.hotVars.latestConfigDigest == configDigest,
        "configDigest mismatch"
    );

    // 截取报告的阶段和轮次信息
```

```

uint40 epochAndRound = uint40(uint256(r.rawReportContext));

// direct numerical comparison works here, because
//
// ((e,r) <= (e',r')) implies (epochAndRound <= epochAndRound')
//
// because alphabetic ordering implies e <= e', and if e = e', then r<=r',
// so e*256+r <= e'*256+r', because r, r' < 256
// 判断该报告轮次是否是最新为最新轮次
require(r.hotVars.latestEpochAndRound < epochAndRound, "stale report");
// 判断签名数量是否大于最大可容忍不诚实节点的数量
require(_rs.length > r.hotVars.threshold, "not enough signatures");
// 判断签名数量是否小于最大的预言机节点数量(预言机节点数量在父合约中被定义为31个)
require(_rs.length <= maxNumOracles, "too many signatures");
// 判断签名的R,S组件数量是否匹配
require(_ss.length == _rs.length, "signatures out of registration");
// 判断价格数据集的数据数量是否小于最大的预言机节点数量
require(r.observations.length <= maxNumOracles,
        "num observations out of bounds");
// 判断价格数据集的数据数量是否大于2倍的最大可容忍不诚实节点的数量
require(r.observations.length > 2 * r.hotVars.threshold,
        "too few values to trust median");

// 获取签名集合的v组件集合并传给r结构体
// Copy signature parities in bytes32 _rawVs to bytes r.v
r.vs = new bytes(_rs.length);
for (uint8 i = 0; i < _rs.length; i++) {
    r.vs[i] = _rawVs[i];
}

// 获取价格数据提供节点的索引数据并传给r结构体
// Copy observer identities in bytes32 rawObservers to bytes r.observers
r.observers = new bytes(r.observations.length); //有多少价格数据就有多少数据提供节点
bool[maxNumOracles] memory seen; //辅助bool数组，用于判断有无出现重复的数据提供节点
for (uint8 i = 0; i < r.observations.length; i++) {
    uint8 observerIdx = uint8(rawObservers[i]); //获取提供第i个价格数据的节点索引
    require(!seen[observerIdx], "observer index repeated"); //有重复节点则停止报告
    seen[observerIdx] = true; //标记某个节点在本轮报告中提供了价格数据
    r.observers[i] = rawObservers[i]; //将索引信息复制到结构体r中
}

Oracle memory transmitter = s_oracles[msg.sender]; //获取此次报告的提交者
require( // Check that sender is authorized to report
    transmitter.role == Role.Transmitter && //检查该提交者是否有提交报告的权限
    msg.sender == s_transmitters[transmitter.index], //检查提交者的索引是否正确
    "unauthorized transmitter"
);
// 获取此次报告的阶段及轮次并传给r结构体
// record epochAndRound here, so that we don't have to carry the local
// variable in transmit. The change is reverted if something fails later.
r.hotVars.latestEpochAndRound = epochAndRound;
}

```

transmit函数片段2

由于第2个片段较长，因此每一行代码的作用我在注释中给出，下面只对一些关键部分展开描述。
transmit定义了ReportData结构体的实例化r用于接收报告内容。

```
//ReportData定义
// Used to relieve stack pressure in transmit
struct ReportData {
    //最新轮次报告数据
    HotVars hotVars; // Only read from storage once
    //价格数据提供节点集合的索引数据
    bytes observers; // ith element is the index of the ith observer
    //价格数据集
    int192[] observations; // ith element is the ith observation
    //签名集合v组件
    bytes vs; // jth element is the v component of the jth signature
    //链下报告内容
    bytes32 rawReportContext;
}
```

ReportData定义

r中hotVars通过获取上一轮的s_hotVars进行初始化，rawReportContext和observations则是将入参_report进行解码后获得。rawReportContext为原生的链下报告内容，主要包括报告配置configDigest和阶段轮次数据epochAndRound，阶段轮次数据这里不再过多说明，报告配置数据则主要是由预言机节点信息，最大可容忍不诚实节点数量，版本信息等数据加密后获得，用于判断报告中所使用的预言机网络节点配置是否符合提前设置的需求。

```
function configDigestFromConfigData(
    address _contractAddress, //合约地址
    uint64 _configCount, //配置版本号，每次配置变更+1
    address[] calldata _signers, //签名节点地址集合
    address[] calldata _transmitters, //报告提交节点地址集合
    uint8 _threshold, //最大可容忍诚实节点数量
    uint64 _encodedConfigVersion, //链下编码版本号
    bytes calldata _encodedConfig //链下编码配置
) internal pure returns (bytes16) {
    return bytes16(keccak256(abi.encode(_contractAddress, _configCount,
        _signers, _transmitters, _threshold, _encodedConfigVersion, _encodedConfig
    )));
}
```

configDigest生成函数configDigestFromConfigData

从报告配置参数configDigest的生成函数中可以看到该参数是对合约地址、配置版本号、签名节点和报告提交节点地址集合等信息进行keccak256编码后获得的，这也就意味着一旦报告中节点或者地址等数据发送变动，该报告就无法通过require对configDigest的判断，以此降低报告的错误率和增加报告的抗碰撞性。

transmit函数片段2的中间部分主要是对一些数据和签名的合理性进行判断，这一部分涉及到数字签名及拜占庭容错等知识，感兴趣的可以在Chainlink白皮书或者上网自行查阅资料了解，这里不做赘述。我们还可以看到合约中定义了一个最大的预言机节点数量maxNumOracles，这个变量在该合约的父合约OffchainAggregatorBilling中被定义为31。

```
// Maximum number of oracles the offchain reporting protocol is designed for
uint256 constant internal maxNumOracles = 31;
```

OffchainAggregatorBilling 父合约中maxNumOracles的定义

函数片段2的最后对价格数据提供节点索引rawObservers和报告提交者transmitter进行了处理，先分析rawObservers。**rawObservers从_report中直接解码获得，其代表了所有提供价格数据的节点的索引，比如rawObservers的第2位是3，那么它就代表observations价格集合中的第2个价格数据是由索引为3的预言机节点提供。**Chainlink不允许一个节点在一份报告中提供两个数据，因此设置了bool[maxNumOracles]数组用于辅助判断，当发现observations集中有两个数据都由同一个索引提供时，则会驳回这次报告。

接着分析报告提交者transmitter。transmit函数需要记录此次报告的提交者以方便发放提交奖励，并且为了达到这一目的合约中还需要有从链下预言机节点到链上提交地址的映射来确认是哪个预言机对应的地址获得该奖励从而方便后继的奖励发放和数据统计。聚合器合约使用s_oracles映射和s_transmitters数组来实现上述功能，这两个数据结构被定义在OffchainAggregator合约的父合约OffchainAggregatorBilling中。

```
mapping (address=> Oracle) internal s_oracles;

struct Oracle {
    // 预言机节点索引号
    uint8 index; // Index of oracle in s_signers/s_transmitters
    // 在报告中扮演的角色
    Role role; // Role of the address which mapped to this struct
}

// Used for s_oracles[a].role, where a is an address, to track the purpose
// of the address, or to indicate that the address is unset.
enum Role {
    // No oracle role has been set for address a
    Unset, // 没有角色
    // Signing address for the s_oracles[a].index'th oracle. I.e., report
    // signatures from this oracle should ecrecover back to address a.
    Signer, // 签名者
    // Transmission address for the s_oracles[a].index'th oracle. I.e., if a
    // report is received by OffchainAggregator.transmit in which msg.sender is
    // a, it is attributed to the s_oracles[a].index'th oracle.
    Transmitter // 提交者
}
```

OffchainAggregatorBilling 父合约中s_oracles的相关定义

s_oracles是从address到Oracle结构体的映射，而Oracle结构体中包含用于标识预言机的索引号和该预言机在某次报告中所扮演的角色Role。Role是枚举类型包含三种角色，包括Unset(未分配角色)、Signer(报告签名者)和Transmitter(报告提交者)。从s_oracles的定义中可以看出，我们可以通过address找到该地址所对应的预言机索引，并且查看该预言机在此次报告中所扮演的角色，从而根据所扮演的角色执行特定操作。

```
// s_transmitters contains the transmission address of each oracle,  
// i.e. the address the oracle actually sends transactions to the contract from  
address[] internal s_transmitters;
```

OffchainAggregatorBilling 父合约中s_transmitters的定义

s_transmitters是一个地址数组，存放的是每个预言机节点将链下报告发送至聚合器合约所使用的地址，其数组索引对应的是s_oracles中Oracle结构体的index。

了解完s_oracles和s_transmitters的定义后再来看transmit函数片段2最后一段代码就能发现这段代码实际上是在判断此次报告的提交者是否为当前轮次被预言机网络选出的提交者，是否是登记在s_transmitters中的预言机地址，以此来判断链下报告的有效性。

2.4.5 transmit函数片段3(获取最终价格数据median)

```
{ // Verify signatures attached to report  
  bytes32 h = keccak256(_report); //加密报告  
  bool[maxNumOracles] memory signed; //辅助数组，用于判断是否有重复签名  
  
  Oracle memory o;  
  for (uint i = 0; i < _rs.length; i++) {  
    //利用预言机节点各自私钥对_report签名后生成的V、S、R组件生成私钥对应的公钥(密码学知识)  
    address signer = ecrecover(h, uint8(r.vs[i])+27, _rs[i], _ss[i]);  
    o = s_oracles[signer];  
    //判断签名生成的公钥地址是否为s_oracles中的签名者(判断有无签名权限)  
    require(o.role == Role.Signer, "address not authorized to sign");  
    require(!signed[o.index], "non-unique signature"); //判断有无重复签名  
    signed[o.index] = true; //标记  
  }  
}  
  
{ // Check the report contents, and record the result  
  // 检查价格数据集observations中的价格数据是否已经按照从小到大排序，方便后面取中位数作为结果  
  for (uint i = 0; i < r.observations.length - 1; i++) {  
    bool inOrder = r.observations[i] <= r.observations[i+1];  
    require(inOrder, "observations not sorted");  
  }  
  // 取价格数据集中的中位数作为该轮BTC/ETH价格聚合中的最终结果  
  int192 median = r.observations[r.observations.length/2];  
  // 最终价格需要在预设的合理区间内  
  require(minAnswer <= median && median <= maxAnswer, "median is out of min-max range");  
  
  // 2.4.1的requestNewRound函数发出新价格数据请求时没有变更latestAggregatorRoundId  
  // 获取最终价格数据后才将latestAggregatorRoundId变量+1  
  r.hotVars.latestAggregatorRoundId++;  
  // 将结果存入s_transmissions映射中  
  s_transmissions[r.hotVars.latestAggregatorRoundId] =  
    Transmission(median, uint64(block.timestamp));  
  //广播新一轮聚合数据摘要  
  emit NewTransmission(
```



```

        r.hotVars.latestAggregatorRoundId,
        median,
        msg.sender,
        r.observations,
        r.observers,
        r.rawReportContext
    );
    // Emit these for backwards compatability with offchain consumers
    // that only support legacy events
    // 广播新一轮次ID
    emit NewRound(
        r.hotVars.latestAggregatorRoundId,
        address(0x0), // use zero address since we don't have anybody "starting" the
round here
        block.timestamp
    );
    // 广播新价格数据
    emit AnswerUpdated(
        median,
        r.hotVars.latestAggregatorRoundId,
        block.timestamp
    );
    // 数据校验
    validateAnswer(r.hotVars.latestAggregatorRoundId, median);
}
s_hotVars = r.hotVars;    //更新s_hotVars
assert(initialGas < maxUint32);    //断言
// 为参与价格聚合的节点发放link代币作为激励(实际发放过程更复杂一些, 在父合约实现)
// 并且偿还报告提交者的gas消耗, initialGas在transmit函数开头通过gasleft()获得
reimburseAndRewardOracles(uint32(initialGas), r.observers);

```

transmit函数片段3

函数片段3的代码功能已经在注释中给出, 开头部分是在验证报告里签名的合理性, 验证签名者的公钥是否为s_oracles里登记的signer或transmitter, 这里用到ecrecover函数来生成公钥, 大概思路就是“用私钥签名的报告签名”+“报告”=“私钥对应的公钥”, 其中报告签名可以由R、S、V三个组件组成。

片段3的第二部分主要是从价格数据集合observations中获取最终的BTC/ETH价格数据, 而获取的方法则是直接从集合中取中位数。报告提交节点在收集到其它节点提交的价格数据后会先将数据从小到大排序再打包发送至聚合器合约, 然后聚合器合约就可以直接在集合的中间位置(observations.length/2)获取到最终价格。取中位数可以避免价格集合中最大值或最小值偏差过大所带来的影响, 比取平均值有更强的稳定性。(实际上个人认为这是因为chainlink网络中节点的规模相对较小, 单纯取平均值的话最值的影响相对较大, 当然每个节点提交的数据本身就是多数据源聚合后所得这一点也保证了取中位数的有较高的可靠性)

取到的最终价格median会被保存在s_transmissions中供价格参考合约获取, 值得一提的是当获取到最终价格后, 聚合器合约才对hotVars的latestAggregatorRoundId变量进行更新, 以此避免roundId更新但获取不到对应价格的情况。而获取到最终价格并且对latestAggregatorRoundId更新后, transmit函数会将这一轮聚合的结果进行广播。

广播完后transmit函数还用validateAnswer对最终价格进行校验，这里的校验主要是将最终价格和上一轮价格作为入参进行比对，但是即使出现异常情况validateAnswer也不会阻止该报告上链，仅会将异常抛出。数据校验功能由专门的validator合约实现，不过目前已上链的聚合器合约中validateAnswer函数内validator合约地址被置为0地址，因此该函数实际上不起作用，这里不过多解析。

当transmit函数完成对链下报告的处理并将价格数据存入合约后，它会调用reimburseAndRewardOracles函数对参与此轮价格聚合的节点发放link代币奖励，并且报销报告提交节点提交报告所花费的手续费。reimburseAndRewardOracles函数由OffchainAggregatorBilling合约实现。

2.5 OffchainAggregatorBilling合约

OffchainAggregatorBilling合约负责向参与PriceFeeds的预言机节点发放代币，包括link激励和gas偿还。这里主要分析一下和reimburseAndRewardOracles函数相关的部分代码。

```
function reimburseAndRewardOracles(
    uint32 initialGas, //发送链下报告时最开始记录下的剩余可用gas，用于计算发送链下报告的总gas
    bytes memory observers //发送价格数据集合的所有节点索引，每一位代表一个节点的索引
)
{
    internal
    {
        Oracle memory txOracle = s_oracles[msg.sender]; //记录提交报告的预言机节点
        Billing memory billing = s_billing; //账单相关参数，记录固定激励金额以及每单位gas报销额
        // Reward oracles for providing observations. Oracles are not rewarded
        // for providing signatures, because signing is essentially free.
        // 获得预言机提交价格数据的次数以用于发放link代币奖励
        // link代币不会在提交数据后立即发放，而是可以在多次提交价格数据后由预言机主动领取(节约gas)
        // oracleRewards函数会对本轮提交过数据的节点的提交数据次数进行更新，以方便后续奖励发放
        s_oracleObservationsCounts =
            oracleRewards(observers, s_oracleObservationsCounts);
        // Reimburse transmitter of the report for gas usage
        require(txOracle.role == Role.Transmitter, //报告需要由本轮被选定出的transmitter提交
            "sent by undesignated transmitter"
        );
        uint256 gasPrice = impliedGasPrice( //设置合理的gasPrice
            tx.gasprice / (1 gwei), // convert to ETH-gwei units
            billing.reasonableGasPrice,
            billing.maximumGasPrice
        );
        // 计算callData的gas开销
        // The following is only an upper bound, as it ignores the cheaper cost for
        // 0 bytes. Safe from overflow, because calldata just isn't that long.
        uint256 callDataGasCost = 16 * msg.data.length;
        // If any changes are made to subsequent calculations, accountingGasCost
        // needs to change, too.
        uint256 gasLeft = gasleft(); //获取当前交易剩余可用gas
        uint256 gasCostEthWei = transmitterGasCostEthWei( // 获取提交该报告所花费的gas总额
            uint256(initialGas),
            gasPrice,
```



```

    callDataGasCost,
    gasLeft
);

// microLinkPerEth is 1e-6LINK/ETH units, gasCostEthWei is 1e-18ETH units
// (ETH-wei), product is 1e-24LINK-wei units, dividing by 1e6 gives
// 1e-18LINK units, i.e. LINK-wei units
// Safe from over/underflow, since all components are non-negative,
// gasCostEthWei will always fit into uint128 and microLinkPerEth is a
// uint32 (128+32 < 256!).
// 计算所需补偿的link代币
uint256 gasCostLinkWei = (gasCostEthWei * billing.microLinkPerEth)/ 1e6;

// Safe from overflow, because gasCostLinkWei < 2**160 and
// billing.linkGweiPerTransmission * (1 gwei) < 2**64 and we increment
// s_gasReimbursementsLinkWei[txOracle.index] at most 2**40 times.
// 计算需要发送给该报告提交者的总link代币数量(gas报销+发送报告奖励)
s_gasReimbursementsLinkWei[txOracle.index] =
    s_gasReimbursementsLinkWei[txOracle.index] + gasCostLinkWei +
    uint256(billing.linkGweiPerTransmission) * (1 gwei); // convert from linkGwei
to linkWei

// Uncomment next line to compute the remaining gas cost after above gasLeft().
// See OffchainAggregatorBilling.accountingGasCost docstring for more
information.
//
// gasUsedInAccounting = gasLeft - gasLeft();
}

```

reimburseAndRewardOracles函数

reimburseAndRewardOracles函数实际上可以看作是一个奖励发放函数，用于向PriceFeeds参与节点发放link代币奖励。需要注意的是，reimburseAndRewardOracles不直接向节点转账，而仅是做一个记录，每当一个报告被提交时，reimburseAndRewardOracles会更新价格数据提交者提交的次数和报告提交者的报销和奖励数额，然后节点可以根据记录领取奖励。这样设计的好处是不需要频繁地进行转账操作，预言机节点可以一次性领取多次价格聚合的奖励，从而降低gas开销。

reimburseAndRewardOracles函数主要就做了三件事，一是获取报告提交节点的相关信息txOracle和账单设置billing；二是用s_oracleObservationsCounts记录节点提交价格数据的次数，作为发放提交价格数据奖励的依据；三是用s_gasReimbursementsLinkWei记录节点的提交报告开销和提交报告奖励，作为发放提交报告奖励的依据。

下面也会根据这三件事展开分析。

2.5.1 txOracle和billing

txOracle实际上就是通过报告提交者的地址获取到该节点对应的s_oracles对象，s_oracles相关的部分在2.4.4节中有提及这里不做重复。

billing则是Billing结构体的实例化对象s_billing，里面主要记录发放奖励时需要设置的参数，如固定激励金额以及每单位gas报销等。

```
// Parameters for oracle payments
struct Billing {

    // Highest compensated gas price, in ETH-gwei units
    uint32 maximumGasPrice; // 发送报告的最大可接受gasPrice

    // If gas price is less (in ETH-gwei units), transmitter gets half the savings
    // 合理的gasPrice, 如果报告提交者transmitter提交报告时的gasPrice低于reasonableGasPrice
    // 则报告提交者可以获得比实际gasPrice更多的gasPrice补偿
    uint32 reasonableGasPrice;

    // Pay transmitter back this much LINK per unit eth spent on gas
    // (1e-6LINK/ETH units)
    uint32 microLinkPerEth; //根据花费在gas的每单位eth开销而返还给报告提交者的link代币

    // Fixed LINK reward for each observer, in LINK-gwei units
    uint32 linkGweiPerObservation; //提交一次价格数据的固定link代币奖励

    // Fixed reward for transmitter, in linkGweiPerObservation units
    uint32 linkGweiPerTransmission; //提交一次报告的固定link代币奖励
}
Billing internal s_billing;
```

Billing结构体定义

Billing结构体中字段的函数如上面注释所示，上述字段在OffchainAggregatorBilling合约构建的时候会进行初始化，并且可以通过setBilling函数进行设置(该函数这里不做说明)。

2.5.2 s_oracleObservationsCounts和oracleRewards

s_oracleObservationsCounts是一个用于记录每个预言机节点提交价格数据次数的数组，其定义如下：

```
// ith element is number of observation rewards due to ith process, plus one.
// This is expected to saturate after an oracle has submitted 65,535
// observations, or about 65535/(3*24*20) = 45 days, given a transmission
// every 3 minutes.
//
// This is always one greater than the actual value, so that when the value is
// reset to zero, we don't end up with a zero value in storage (which would
// result in a higher gas cost, the next time the value is incremented.)
// Calculations using this variable need to take that offset into account.
uint16[maxNumOracles] internal s_oracleObsrvationsCounts;
```

s_oracleObservationsCounts定义

s_oracleObservationsCounts是个uint16的数组，长度为最大的预言机数量maxNumOracles，也就是31。其数组下标对应各个预言机节点的索引Oracle.index，而下标对应的值就是该预言机在过去一段时间已提交且尚未领取奖励的价格数据的个数(一次聚合中一个节点只能提交一个价格数据)，当该节点领取完奖励后该值会被置为1(不置为0是为了节约gas开销，以太坊虚拟机中将状态从零值变为非零值会有额外的gas开销)。

reimburseAndRewardOracles函数中通过调用oracleRewards函数对s_oracleObservationsCounts数组进行更新。

```
function oracleRewards(
  bytes memory observers,
  uint16[maxNumOracles] memory observations
)
  internal
  pure
  returns (uint16[maxNumOracles] memory)
{
  // reward each observer-participant with the observer reward
  for (uint obsIdx = 0; obsIdx < observers.length; obsIdx++) {
    uint8 observer = uint8(observers[obsIdx]); //获取预言机节点下标index
    // observer对应下标的预言机节点的价格数据提交个数+1, 为防止溢出进行了特殊处理
    observations[observer] = saturatingAddUint16(observations[observer], 1);
  }
  return observations;
}
```

oracleRewards函数定义

oracleRewards函数的传入参数是observers和observations, 其中observers对应的是2.4.4中提到的rawObservers或r.observers, observations对应的则是s_oracleObservationsCounts(**特别注意, 这里的observations不是2.4节transmit函数里的observations**)。

从上面的函数定义可以看出oracleRewards函数的作用就是通过传入的observers获得该轮参与聚合的节点下标, 然后根据下标将s_oracleObservationsCounts中所有参与节点对应的数据提交个数+1。为了防止数据提交个数溢出(超过65536), oracleRewards函数还调用了saturatingAddUint16对每次的累加进行处理。

```
function saturatingAddUint16(uint16 _x, uint16 _y)
  internal
  pure
  returns (uint16)
{
  return uint16(min(uint256(_x)+uint256(_y), maxUint16));
}
```

saturatingAddUint16函数定义

saturatingAddUint16函数的实现比较简单, 就是返回入参x, y之和以及65536之间的较小值。也就是说当某个预言机提交过65536个数据且没有领取奖励后, 即使再提交新的数据其数据提交个数也不会再增加, 依旧为65536。

简而言之, 节点可以根据s_oracleObservationsCounts内的记录领取价格数据提交的奖励, 而每次提交报告时报告提交者都会通过oracleRewards函数对s_oracleObservationsCounts内的数据进行更新。

2.5.3 gas报销计算和s_gasReimbursementsLinkWei

在2.5开头我们可以看到，reimburseAndRewardOracles函数在对s_oracleObservationsCounts进行更新后的后续代码主要就是在计算transmitter提交报告时的gas开销，以方便对transmitter进行gas报销和报告提交奖励。

首先第一步是用impliedGasPrice函数计算一个用于gas报销的gasPrice。

```
// Gas price at which the transmitter should be reimbursed, in ETH-gwei/gas
function impliedGasPrice(
    uint256 txGasPrice,           // ETH-gwei/gas units    //实际的gasPrice
    uint256 reasonableGasPrice, // ETH-gwei/gas units    //官方设置的合理的gasPrice
    uint256 maximumGasPrice      // ETH-gwei/gas units    //可接受的最大gasPrice
)
    internal
    pure
    returns (uint256)
{
    // Reward the transmitter for choosing an efficient gas price: if they manage
    // to come in lower than considered reasonable, give them half the savings.
    //
    // The following calculations are all in units of gwei/gas, i.e. 1e-9ETH/gas
    // chainlink鼓励报告提交者选择低gasPrice，并会对该行为进行奖励
    uint256 gasPrice = txGasPrice; //获取实际的gasPrice
    if (txGasPrice < reasonableGasPrice) { //如果提交报告时的gasPrice较低则给予奖励
        // Give transmitter half the savings for coming in under the reasonable gas
        price
        gasPrice += (reasonableGasPrice - txGasPrice) / 2; //多奖励两者差额的一半
    }
    // Don't reimburse a gas price higher than maximumGasPrice
    // 如果提交报告时gasPrice过高，只会报销低于maximumGasPrice的部分
    return min(gasPrice, maximumGasPrice); //返回两者之间较小值
}
```

impliedGasPrice函数定义

第二步计算传入参数的gas开销callDataGasCost，这一步比较简单，chainlink对传入的报告每个字节报销16gas(实际开销为4gas/字节)。第三步是用transmitterGasCostEthWei函数计算运行整个transmit函数所需的gas开销。

```
// gas reimbursement due the transmitter, in ETH-wei
//
// If this function is changed, accountingGasCost needs to change, too. See
// its docstring
function transmitterGasCostEthWei(
    uint256 initialGas, // transmit函数开头记录的剩余可用gas
    uint256 gasPrice,   // ETH-gwei/gas units //用impliedGasPrice函数算出的gasPrice
    uint256 callDataCost, // gas units //传入参数callData的gas开销
    uint256 gasLeft // 调用transmitterGasCostEthWei函数前剩余可用gas
)
    internal
```

```

    pure
    returns (uint128 gasCostEthWei)
{
    require(initialGas >= gasLeft, "gasLeft cannot exceed initialGas");
    uint256 gasUsed = // gas units //计算总共使用的gas
        initialGas - gasLeft + // observed gas usage //transmit函数开始到该函数前的总gas
        开销
        // accountingGasCost为reimburseAndRewardOracles函数后续语句的gas开销
        // 因为调用transmitterGasCostEthWei函数后还会执行其它语句，所以要计算
        accountingGasCost
        callDataCost + accountingGasCost; // estimated gas usage
        // gasUsed is in gas units, gasPrice is in ETH-gwei/gas units; convert to ETH-
        wei
        uint256 fullGasCostEthWei = gasUsed * gasPrice * (1 gwei); //计算总gas开销
        assert(fullGasCostEthWei < maxUint128); // the entire ETH supply fits in a
        uint128...
        return uint128(fullGasCostEthWei); //返回总gas开销
}

```

transmitterGasCostEthWei函数定义

transmitterGasCostEthWei函数的语句作用如上所示，从函数可以看到发送报告所使用的总gas数量gasUsed由两部分组成，第一部分是传入calldata所需要的gas数量，第二部分是整个transmit函数运行所需要的gas数量。而第二部分在transmitterGasCostEthWei函数中又被分为transmitterGasCostEthWei函数前使用gas数量(initialGas - gasLeft)和transmitterGasCostEthWei函数后使用gas数量accountingGasCost。

获得gasUsed后就可以用impliedGasPrice函数得到的gasPrice来计算最后的总gas开销gasCostEthWei。

```

// This value needs to change if maxNumOracles is increased, or the accounting
// calculations at the bottom of reimburseAndRewardOracles change.
//
// To recalculate it, run the profiler as described in
// ../../profile/README.md, and add up the gas-usage values reported for the
// lines in reimburseAndRewardOracles following the "gasLeft = gasleft()"
// line. E.g., you will see output like this:
//
//      7      uint256 gasLeft = gasleft();
//     29      uint256 gasCostEthWei = transmitterGasCostEthWei(
//      9          uint256(initialGas),
//      3          gasPrice,
//      3          callDataGasCost,
//      3          gasLeft
//      .
//      .
//      .
//     59      uint256 gasCostLinkWei = (gasCostEthWei *
// billing.microLinkPerEth)/ 1e6;
//      .
//      .
//      .
//    5047      s_gasReimbursementsLinkWei[txOracle.index] =

```

```
//      856          s_gasReimbursementsLinkWei[txOracle.index] + gasCostLinkWei +
//      26          uint256(billing.linkGweiPerTransmission) * (1 gwei);
//
// If those were the only lines to be accounted for, you would add up
// 29+9+3+3+3+59+5047+856+26=6035.
uint256 internal constant accountingGasCost = 6035;
```

accountingGasCost定义和计算

逐句计算最后部分所示用的gas数量后得到的accountingGasCost为6035，后续如果chainlink对调用transmitterGasCostEthWei函数后的语句进行更改，则accountingGasCost也要重新计算并更改。

计算完总gas开销gasCostEthWei后，第四步就是根据gasCostEthWei计算需要报销给transmitter的link代币(chainlink用link代币来结算报销和奖励)。每eth的gas开销所补偿的link代币数量被定义在Billing结构体的microLinkPerEth字段中，最后算出来的总link代币报销数量为gasCostLinkWei。

第五步，也就是最后一步则是计算总共需要发送给transmitter的总link代币数量(发送报告本身也有link代币奖励)，然后累加进s_gasReimbursementsLinkWei数组中。总link代币发放数量为gasCostLinkWei加上提交报告奖励billing.linkGweiPerTransmission。s_gasReimbursementsLinkWei和s_oracleObservationsCounts类似，多次累加，一次性领取，初始值为1。

2.5.4 节点领取link代币奖励(payOracle和owedPayment)

从2.5.1到2.5.3这三小节可以看出，reimburseAndRewardOracles实际上就是对节点应该领取的奖励和应该返还给报告提交节点的报销进行记录，但还没有真正得将奖励发放下去。节点可以通过withdrawPayment函数提取link代币。

节点领取的link代币分为两部分，一部分是提供价格数据的奖励，被记录在s_oracleObservationsCounts中，另一部分是节点作为transmitter时的gas报销及提交报告奖励，被记录在s_gasReimbursementsLinkWei中。

```
function withdrawPayment(address _transmitter)
    external
{
    // 需要发起link代币提现的地址为预先登记在案的收款地址
    require(msg.sender == s_payees[_transmitter], "Only payee can withdraw");
    payOracle(_transmitter);    //往收款地址支付link代币
}
```

withdrawPayment函数实现

调用withdrawPayment函数需要传入预言机节点的transmitter地址，然后withdrawPayment函数会判断该地址对应的收款地址是否为交易发起地址(只有收款地址才能调用该函数)，最后该函数会调用payOracle向收款地址支付link代币。

```
// Addresses at which oracles want to receive payments, by transmitter address
mapping (address /* transmitter */ => address /* payment address */)
    internal
    s_payees;
```

s_payees定义

s_payees是从节点地址到收款地址的映射，节点提交报告的地址和收款地址可以不同。

```
// payOracle pays out _transmitter's balance to the corresponding payee, and zeros it out
function payOracle(address _transmitter)
    internal
{
    Oracle memory oracle = s_oracles[_transmitter]; //获取Oracle信息
    // 用owedPayment函数获取该预言机节点可获取的link代币数量
    uint256 linkWeiAmount = owedPayment(_transmitter);
    if (linkWeiAmount > 0) {
        address payee = s_payees[_transmitter]; //获取该预言机收款地址
        // Poses no re-entrancy issues, because LINK.transfer does not yield control flow.
        // 调用link代币合约向收款地址转账
        require(LINK.transfer(payee, linkWeiAmount), "insufficient funds");
        // 初始化s_oracleObservationsCounts中对应预言机的提交数据数量
        s_oracleObservationsCounts[oracle.index] = 1; // "zero" the counts. see var's docstring
        // 初始化s_gasReimbursementsLinkWei中对应预言机的可获取报销和奖励的link代币数量
        s_gasReimbursementsLinkWei[oracle.index] = 1; // "zero" the counts. see var's docstring
        emit OraclePaid(_transmitter, payee, linkWeiAmount); //广播
    }
}
```

payOracle函数实现

payOracle函数通过owedPayment获取transmitter对应预言机可提取的link代币总量linkWeiAmount，然后从s_payees映射获取收款地址并转账，转完帐后将该预言机对应的剩余未领取奖励记录进行初始化，最后再对提现结果进行广播。

```
/**
 * @notice query an oracle's payment amount
 * @param _transmitter the transmitter address of the oracle
 */
function owedPayment(address _transmitter)
    public
    view
    returns (uint256)
{
    Oracle memory oracle = s_oracles[_transmitter]; //获取Oracle信息
    if (oracle.role == Role.Unset) { return 0; } //预言机需要已注册在案
    Billing memory billing = s_billing; //获取billing账单设置
    // 获取数据提交奖励, = 数据提交个数 * 每个数据的link奖励数量
    uint256 linkWeiAmount =
        uint256(s_oracleObservationsCounts[oracle.index] - 1) * //初始值为1, 需减去
        uint256(billing.linkGweiPerObservation) *
        (1 gwei);
    // 加上该预言机作为transmitter提交报告时的报销和奖励
    linkWeiAmount += s_gasReimbursementsLinkWei[oracle.index] - 1;
```



```
return linkweiAmount;    // = 数据提交奖励 + 报告提交奖励 + gas报销
}
```

owedPayment函数实现

owedPayment函数的作用是获取某个预言机当前可领取的link代币数量。当前可领取代币数量=提交数据个数*提交每个数据的奖励+报告提交奖励和gas报销。

2.6 PriceFeeds合约部署时constructor赋予的初始值

PriceFeeds采用的是多合约文件集中部署的形式，所以2.1到2.5提到的所有合约中BTC/ETH价格参考合约EACAgregatorProxy及其父合约AggregatorProxy被部署在同一合约地址下，它们依赖的价格数据来源聚合器合约AccessControlledOffchainAggregator和其父合约OffchainAggregator、OffchainAggregatorBilling被部署在同一合约地址下。

为了加深对于合约中定义的各种状态变量的理解，下面从etherscan中分别查询两个合约地址部署时构造函数为某些变量赋予的初始值(是函数部署时赋予的值，与当前可能不同)。

🔗 **Constructor Arguments** (ABI-Encoded and is the last bytes of the Contract Creation Code above)

```
-----Decoded View-----  
Arg [0] : _aggregator (address): 0x0133Aa47B6197D0BA090Bf2CD96626Eb71fFd13c  
Arg [1] : _accessController (address): 0x00000000000000000000000000000000
```

价格参考合约constructor赋予的初始值

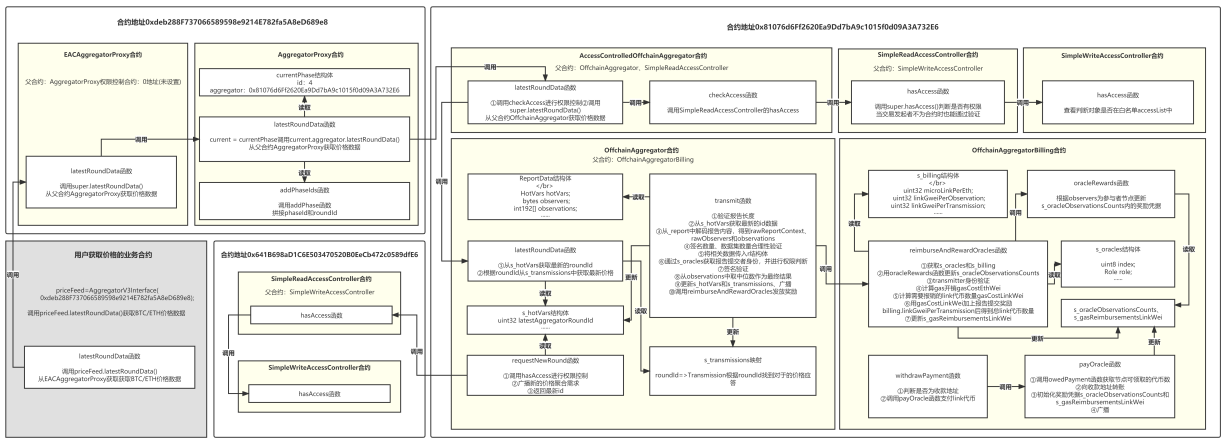
 Constructor Arguments (ABI-Encoded and is the last bytes of the Contract Creation Code above)

[illegible]

```
-----Decoded View-----  
Arg [0] : _maximumGasPrice (uint32): 1000  
Arg [1] : _reasonableGasPrice (uint32): 100  
Arg [2] : _microLinkPerEth (uint32): 65000000  
Arg [3] : _linkGweiPerObservation (uint32): 60000000  
Arg [4] : _linkGweiPerTransmission (uint32): 300000000  
Arg [5] : _link (address): 0x514910771AF9Ca656af840dff83E8264EcF986CA  
Arg [6] : _validator (address): 0x0000000000000000000000000000000000000000  
Arg [7] : _minAnswer (int192): 1000000000000000000  
Arg [8] : _maxAnswer (int192): 10000000000000000000000000000000000000000  
Arg [9] : _billingAccessController (address): 0x9db83CEf9f68b63989E4E82D65D549e7ff2aCda9  
Arg [10] : _requesterAccessController (address): 0x641B698aD1C6E503470520B0EeCb472c0589dfE6  
Arg [11] : _decimals (uint8): 18  
Arg [12] : description (string): BTC / ETH
```

聚合器合约constructor赋予的初始值

3 PriceFeeds合约调用示意图



PriceFeeds 合约调用示意图

4 最后

撰写这篇文章的目的主要是加深自己对于Chainlink PriceFeeds合约的理解，并作为记录以方便后续回头查阅。为了降低阅读门槛会有些许地方稍显啰嗦，各位挑选可能对自己有用的地方来看即可(整篇文章是照着我看代码的顺序梳理的，因此按顺序看可能更容易理解)。

合约分析中只讲了我觉得有必要讲的地方，并没有列出所有的函数及变量。这篇文章可以作为入门，但是真正要深入理解PriceFeeds的链上部分还是得去看白皮书和源码。

文章撰写的时间跨度较长，内容也较多，并且代码解析部分也是基于我的理解写的，因此肯定会有理解错误的地方，欢迎提出错误，我会以最快的速度修改。

有疑惑的地方也可以提出来大家一起讨论。