Message Execution Options

What are message_options ?

Because the source chain has no concept of the destination chain's state, you must specify the amount of gas you anticipate will be necessary for executing yourlzReceive orlzCompose method on the destination smart contract.

LayerZero provides robustMessage Execution Options , which allow you to specify arbitrary logic as part of the message transaction, such as the gas amount andmsg.value theExecutor pays for message delivery, the order of message execution, or dropping an amount of gas to a destination address.

The most common options you will use when building arelzReceiveOption ,lzComposeOption , andlzNativeDropOption .

Options Builders

A Solidity library and off-chain SDK have been provided to build specific Message Options for your application.

- OptionsBuilder.sol
- : Can be imported from@layerzerolabs/lz-evm-oapp-v2
- .
- options.ts
- : Can be imported from@layerzerolabs/lz-v2-utilities
- .

Generating Options

You can generate options depending on your OApp's development environment:

- Remix
- : for quick testing in Remix, you can deploy locally to the Remix VM a contract using theOptionsBuilder.sol
- library. See the example provided below.
- Open in Remix
- What is Remix?
- Foundry
- :_options
- can be generated directly in your Foundry unit tests using theOptionsBuilder.sol
- library. See theOmniCounter Test
- file as an example for how to properly invoke options.
- Hardhat
- : you can also locally declare options in Hardhat via theoptions.ts
- file.

All tools use the same method for packing the_options bytes array to simplify your experience when switching between environments.

Step 1: Import Options

All_options tools must be imported into your environment to be used.

Options Library

Import theOptionsBuilder from@layerzerolabs/lz-evm-oapp-v2 into either your Foundry test or smart contract to be deployed locally.

import

{ OptionsBuilder }

from

"@layerzerolabs/lz-evm-oapp-v2/contracts/oapp/libs/OptionsBuilder.sol" ;

Options SDK

Start by importingOptions from@layerzerolabs/lz-v2-utilities .

import

{ Options }

from

'@layerzerolabs/lz-v2-utilities' ;

Step 2: Initializing Options

ThenewOptions method is used to initialize a new bytes array.

const _options =

Options . newOptions ( ) ; It's a starting point to which you can add specific option configurations.

The command below in Solidity allows you to conveniently extend the bytes type with LayerZero'sOptionsBuilder library methods, simplifying the creation and manipulation of message execution options.

using

OptionsBuilder

for

bytes ; info Message Options start with a magic number to identify the options type, followed by type-specific instructions.

TYPE_1 andTYPE_2 are legacy options identifiers for handling Endpoint V1 messages (ie. Relayer Adapter Parameters).

For LayerZero V2, only useTYPE_3 as outlined in this page.

Step 3: Adding Option Types

When generating_options , you will want specific handling for different message types used in your smart contract. For instance,addExecutorLzReceiveOption is a method that can be used to specify how the Executor callslzReceive on the receiving chain.

const executorGas =

1000000 ;

// Gas limit for the executor const executorValue =

0 ;

// msg.value for the lzReceive() function on destination in wei

const _options =

Options . newOptions ( ) . addExecutorLzReceiveOption (executorGas , executorValue) ; You can continue appending newOptions methods to add more Executor message handling; all packed into a single call.

See below for allOption Types .

Step 4: Passing Options in Send

After generating_options , you will want to test them in asend call.

Options SDK

Using the Options SDK, this can be passed directly into a Hardhat task or unit test depending on your use case.

// Other parameters for the send function const _dstEid =

'someEndpointId' ;

// Destination endpoint ID const message =

'Your message here' ;

// The message you want to send

// Call the send function on the smart contract // Convert your options array toHex() const tx =

await yourOAppContract . send (destEndpointId , message , _options . toHex ( )) ; await tx . wait ( ) ; In this Typescript snippet, thesend function is being called on theYourOAppContract contract instance, passing in the destination endpoint ID, the message, and the_options that were constructed:

contract

YourOAppContract

{ // ... other functions and declarations

function

send (uint32 _dstEid ,

string

memory message ,

bytes

memory _options)

public

payable

{ // Logic to handle the sending of the message with the provided options // This might involve interacting with other contracts or internal logic bytes

memory _payload = abi . encode (message) ; _lzSend (_dstEid ,

// Destination chain's endpoint ID. _payload ,

// Encoded message payload being sent. _options ,

// Message execution options (e.g., gas to use on destination). MessagingFee (msg . value ,

0) ,

// Fee struct containing native gas and ZRO token. payable (msg . sender)

// The refund address in case the send call reverts.) ; }

// ... other functions and declarations } In this Solidity snippet, thesend function takes three parameters:_dstEid ,message , and_options . The function's logic would then use those parameters to to send a message cross-chain.

Options Library

Using theOptionsBuilder.sol library, these_options can be directly referenced in your Foundry tests for quick local testing.

import

{ MyOApp }

from

"../contracts/oapp/examples/MyOApp.sol" ; import

{ TestHelper }

from

"../contracts/tests/TestHelper.sol" ; import

{ OptionsBuilder }

from

"@layerzerolabs/lz-evm-oapp-v2/contracts/oapp/libs/OptionsBuilder.sol" ;

contract

MyOAppTest

is TestHelper { using

OptionsBuilder

for

bytes ; // ... other test setup functions function

test_increment ( )

public

{ bytes

memory options = OptionsBuilder . newOptions ( ) . addExecutorLzReceiveOption (200000 ,

0) ; (uint256 nativeFee ,

)

= aCounter . quote (bEid , MsgCodec . VANILLA_TYPE , options) ; aCounter . increment { value : nativeFee } (bEid , MsgCodec . VANILLA_TYPE , options) ; // ... other test logic } } The above example is taken from theOmniCounter Foundry Test in the LayerZero V2 repo.

info SeeTestHelper.sol for full Foundry testing support.

Option Types

There are multiple option types to take advantage of, each controlling specific handling of LayerZero messages.

lzReceive

Option

ThelzReceive option specifies the gas values the Executor uses when callinglzReceive on the destination chain.

Options . newOptions ( ) . addExecutorLzReceiveOption (200000 ,

0) ; It defines the amount of_gas andmsg.value to be used in thelzReceive call by the Executor on the destination chain:

OPTION_TYPE_LZRECEIVE contains(uint128 _gas, uint128 _value)

_gas : The amount of gas you'd provide for thelzReceive call in source chain native tokens.200000 should be enough for most transactions.

_value : Themsg.value for the call. This value is often included to fund any operations that need native gas on the destination chain, including sending another nested message.

lzCompose

Option

This option allows you to allocate some gas and value to yourComposed Message on the destination chainlzCompose is used when you want to call external contracts from yourlzReceive function.

Options . newOptions ( ) . addExecutorLzComposeOption (0 ,

1000000 ,

0) ; OPTION_TYPE_LZCOMPOSE contains(uint16 _index, uint128 _gas, uint128 _value)

_index : The index of thelzCompose() function call. When multiples of this option are added, they are summed PER index by the Executor on the remote chain. This can be useful for defining multiple composed message steps that happen sequentially.

_gas : The gas amount for the lzCompose call varies based on the destination's compose logic and the destination chain's characteristics (e.g., opcode pricing). Although1000000 is commonly used as a default, it's important to perform tailored testing to determine the optimal gas requirement for your specific transaction needs.

_value : Themsg.value for the call.

lzNativeDrop

Option

This option contains how much native gas you want to drop to the_receiver , this is often done to allow users or a contract to have some gas on a new chain.

Options . newOptions ( ) . addExecutorNativeDropOption (1000000000000 , receiverAddressInBytes32) ; OPTION_TYPE_LZNATIVEDROP contains(uint128 _amount, bytes32 _receiver)

_amount : The amount of gas in wei to drop for the receiver.

_receiver : Thebytes32 representation of the receiver address.

OrderedExecution

Option

By adding this option, the Executor will utilizeOrdered Message Delivery . This overrides the default behavior ofUnordered Message Delivery .

Options . newOptions ( ) . addExecutorOrderedExecutionOption (bytes ('')) ; For example, if nonce2 transaction fails, all subsequent transactions with this option will not be executed until the previous message has been resolved with.

bytes : The argument should always be initialized as an empty bytes array ("").

caution These message_options must be combined with in-app contract changes, listed underOrdered Message Delivery .

Duplicate Option Types

Multiple options of the same type can be passed and appended into the same options array. The logic on how multiple options of the same type are summed differs per option type:

bytes

memory options = OptionsBuilder . newOptions ( ) . addExecutorLzReceiveOption (200000 ,

0) . addExecutorLzComposeOption (0 ,

1000000 ,

0) . addExecutorLzComposeOption (1 ,

500000 ,

0) ; * lzReceive * : Both the_gas * and_value * parameters are summed. * lzCompose * : Both the_gas * and_value * parameters aresummed by index * . * lzNativeDrop * : The_amount * parameter is summed by unique_receiver * address.

Make sure that appending multiple options is the intended behavior of your unique OApp.Edit this page

Previous Executor Next Estimating Gas Fees