SemiOS Contract Development

Version:1.0

https://semios.io/

SemiOS team

2023.11

Contents

1 Terms

- SubNodes: Basic Unit of SemiOS Governance. A SubNodes belongs to a SeedNodes.
- SeedNodes: A collection of SubNodes, representing a governance series.
- Builder: The work uploader. Multiple works may share a same builder. A builder belongs to a SubNodes.
- Starter:SubNodes Creator. When creating SubNodes, an existing SeedNodes can be specified, or a new SeedNodes can be created for affiliation.
- Work: Users (called Builders) can upload pictures to SemiOS website for a SubNodes, which are available for purchase by other users. When a work is purchased it generates an (on-chain) NFT. This purchase process is also called minting an NFT, and the buyer is also referred to as the Minter.
- NFT: Non-Fungible Token, is facilitated by the ERC721 contract which provides a standard framework. In SemiOS, a NFT is assosiate with a SubNodes and a Builder. A NFT identifier includes a ERC721 address and a tokenId.
- Minter: The user that mints a NFT, who is also the NFT's owner.
- ERC20: Fungible Token.
- OutputToken: An endogenous ERC20 token associated with a SeedNodes, as assets generated by this governance series.
- InputToken: An exogenous token associated with a SeedNodes, as external assets attracted by the SeedNodes, usually uses ETH or other known ERC20-tokens.
- NodesAssetPool: A fund pool associated with a SubNodes.
- NodesRedeemPool: A special fund pool associated with a SeedNodes, providing the function of exchanging OutputTokens for InputTokens.
- Treasury: A special fund pool associated with a SeedNodes, used to store initial OutputTokens.
- RoyaltyFee: Fee charged on secondary trades of an NFT.
- RoyaltySplitter: A contract for distributing Royalty Fees.
- ProtocolFeePool: An address to collect commissions for the SemiOS project owner.

- Pass Card: Reserved NFTs under a SubNodes.
- URI: A string type, usually containing a link. SubNodes, NFT, and Builder each have their corresponding URI.
- Fixed Price/Unified Fixed Price: Refers to the use of a fixed price for minting NFTs.
- Block: Associate to a SubNodes, also known as a block, it is a time window unit for reward distribution and price change strategies.
- Active Block: A block in which any NFT has been minted in the associate SubNodes.
- MintWindowDuration: The duration time of one round, counted by the block number.
- MintFee (NFT price): The token amount to pay for minting an NFT, usually uses InputToken.
- Block Reward: Assets flowing out of a NodesAssetPool in a round.
- TopUp Mode: A special SubNodes mode for fundraising purposes.
- Lottery Mode: A special SubNodes mode where non-active rounds' earnings accumulate.
- Infinite Mode: A special SubNodes mode with infinite rounds, where each round's total block reward equals the total assets of the NodesAssetPool.
- OutputToken Payment Mode: A special SubNodes mode where MintFee is paid by OutputToken.
- Self-Selected InputToken Mode: A special SeedNodes mode where the ETH used for MintFee is replaced with an ERC20 token chosen by the user.
- Import OutputToken Mode: A special SeedNodes mode where OutputToken is an existing ERC20 token imported by the creator, instead of SeedNodes generated ERC20 token.
- TopUp Account: An account used by users for investment in TopUp mode.
- Maker: The owner of a TopUp account.

2 SemiOS Introduction

The SemiOS, previously known as DAO For Art and later ProtoDao, has been ongoing over two years since its first commit on June 15, 2022. It is an integrated product that encompasses NFT uploading, minting, governance, and reward distribution. This section will provide a brief overview of the project's functionalities based on version 1.3. Features from new versions are also introduced. Starting from version 1.6, the project name was changed to SemiOS.

2.1 SubNodes

A SubNodes is the governance unit used in this project. Users can create a SubNodes and engage in activities such as NFT minting, reward claim, and parameter modification based on this SubNodes. The user who create the SubNodes is called Starter, also known as SubNodes owner, who has authority to set SubNodes attributes and is one of the roles that receive profits. Each SubNodes is identified in the contract by a unique *daoId* (of types bytes32).

2.2 SubNodes Associated Contracts

2.2.1 ERC721 Contract

When each SubNodes is created, an ERC721 contract is also established and associated with it. This contract provides functionalities related to the minting of NFTs.

2.2.2 NodesAssetPool Contract

Each SubNodes, upon creation, is associated with a fund pool contract known as the NodesAssetPool. This contract is used to store the InputTokens and OutputTokens of the SubNodes.

2.2.3 RoyaltySplitter Contract

After an NFT is minted and begins circulating on the secondary market, major exchanges transfer a portion of the transaction fees from secondary sales to a specific

address, as stated in the ERC721 contract. The information includes the fee percentage and the recipient address. In SemiOS, the fee percentage set by the Starter is restricted within system-defined maximum and minimum limits. This part of the fee is known as the RoyaltyFee. Note that this action is not mandatory, but exchanges generally adhere to this rule voluntarily. This rule does not apply to peer-to-peer transfers between users.

The SemiOS system wants the RoyaltyFee to be directed to two addresses: one being the NodesRedeemPool and the other the project's address, known as the ProtocolFeePool. However, the standard for RoyaltyFee in the ERC721 only allows for one declared recipient address. Therefore, a RoyaltySplitter contract is introduced to facilitate the automatic distribution of RoyaltyFees.

Each SubNodes, upon creation, sets up a RoyaltySplitter contract, which is declared in the ERC721 contract as the recipient of the RoyaltyFee. When this contract receives ETH, it automatically distributes the received ETH to specific split addresses according to predefined proportions, through the fallback() function.

- If the SubNodes has not enabled the Import OutputToken mode, the addresses for the split payment are the ProtocolFeePool and the NodesRedeemPool.
- If the SubNodes has enabled the Import OutputToken mode, the addresses for the split payment are the ProtocolFeePool and the NodesAssetPool, as the import ERC20 does not support the Redeem function.
- Currently, the ProtocolFeePool collects a 2.5% share of the RoyaltyFee.

The RoyaltySplitter contract also supports the distribution of ERC20 tokens, but this must be triggered manually or via a Chainlink Keeper.

2.2.4 GrantAssetPoolNFT Contract

In version 1.6, a new feature was added involving an ERC721 contract created with each SubNodes. When a user makes a grant to the NodesAssetPool, this contract mints an NFT as proof of the donation.

2.3 SeedNodes

SeedNodes is a collection of SubNodes which belong to the same governance series. When creating a SubNodes, an existing SeedNodes should be specified for affiliation.

If no association is made, a new SeedNodes is created for affiliation. In this case, the SubNodes Starter is the owner of the SeedNodes.

2.4 SeedNodes Associated Contracts

SeedNodes Associated Contracts means that all associate SubNodes share the same contract.

2.4.1 ERC20 Contract

Each SeedNodes is associated with an ERC20 contract created concurrently with its inception. This contract serves as the governance token for the series, known as the OutputToken. All SubNodes within the series share the same OutputToken.

New in version 1.3: Upon creation, a SeedNodes can opt to associate with an existing ERC20 address, known as the Import OutputToken mode. In this mode, there is no need to create a new ERC20 contract.

2.4.2 NodesRedeemPool Contract

Each SeedNodes is also associated with an additional fund pool contract known as the NodesRedeemPool upon its creation,. This contract is used to store a portion of the InputToken revenues from all SubNodes in the series and provides users with the functionality to exchange their OutputTokens for InputTokens.

2.4.3 Treasury Contract

This feature, added in 1.6, involves each SeedNodes creating an additional fund pool upon its establishments. Additionally, a preset total number of OutputToken is generated for the system. The default total number is set at 1 billion.

2.4.4 GrantTreasuryNFT Contract

Additionally, a similar feature was introduced in version 1.6 for each SeedNodes. When a user makes a grant to the Treasury, this ERC721 contract mints an NFT as a certificate of the transaction.

In conclusion, SeedNodes and SubNodes have following relationships:

- All SubNodes shares the same OutputToken (as well as InputToken) as its associated SeedNodes, but SubNodes still maintains its own ERC721 contract. As of version 1.3, whether a SubNodes adopts Import OutputToken mode is consistent with its associated SeedNodes.
- All SubNodes shares the same NodesRedeemPool with its associated SeedNodes, but still possesses its own NodesAssetPool.
- Compared to SubNodes starters, SeedNodes creators have more permissions, mainly in setting treasury assets and rights. These rights will be represented as NFTs starting from version 1.6.

2.5 Builder

Builder represents the uploader of work and is identified in the contract by a unique *canvasId* (of type bytes32). Each work belongs to a Builder, and each Builder is associated with a SubNodes. The *canvasId* is generated off-chain and must be provided to the contract at the time of minting the NFT. Builder is also one of the beneficiaries.

When a SubNodes is created, the first Builder is automatically generated, which is the SubNodes Starter. All reserved NFTs (pass cards) of the SubNodes belong to this Builder.

In the current version, the NFT pricing strategy is associated with the Builder. Future versions simplify to SubNodes.

2.6 SubNodes Block

The SubNodes Block serves as the minimum calculation period for NFT pricing strategies and reward distribution strategies. Many rules are based on rounds. For example, rewards distributed within a block are calculated and disbursed when the round is over. Unique rules apply to NFT pricing strategies within a round, which will be discussed further below.

Each SubNodes can specify a parameter called Block duration, which represents the duration of a block counted by Ethereum block number. Each time the MintWindowDuration is reset, the current round is recalculated.

If any works are minted within a SubNodes during a block, that block is considered active.

Each SubNodes has an attribute called remaining blocks. With each passing block, SubNodes' remaining blocks decrease by one. The remaining rounds of a SubNodes can be set by owner. When a SubNodes' remaining blocks reach zero, the SubNodes enters a dead state, during which no NFT can be minted. The owner can revive a dead SubNodes by setting a non-zero remaining blocks (or switching to infinite mode), after which all data on active blocks and pricing strategies are reset and recalculated.

2.7 Work and NFT

Builders upload works to the website. Before being minted in the contract, it has nothing to do with the blockchain. When minters chooses to purchase a work, an on-chain NFT is minted. Each NFT is assigned to a specific Builder (*canvasId*) and SubNodes (*daoId*).

Users can mint an NFT under an existing Builder, or choose to create a new Builder and mint an NFT belongs to it simultaneously, depending on whether the specified *canvasId* already exists.

Once an NFT is minted, its owner becomes the initiator of the minting transaction, and it can subsequently be traded on the secondary market.

2.7.1 NFT Price

When the NFT minting function is called, a payment called MintFee in InputToken is required (added in version 1.4: in OutputTokenpayment mode, OutputToken is used for MintFee) as the price for minting the NFT. Typically, the InputToken is ETH or well-known third-party ERC-20 tokens such as USDT and USDC.

The pricing for NFTs can follow these models:

- Unified Fixed Price

  After a SubNodes activates this mode, its owner will set a fixed price. In this case, all NFTs under this SubNodes are minted at this price, and minting does not require a signature. This mode cannot be switched.

  The unified fixed price can be set to zero initially. If so, it can not be set to a non-zero value. If initial unified fixed price is not zero, owner can set it to any non-zero value at any time.

- Fixed Price

When the unified fixed price is disabled, the Builder can designate a fixed price for his work by providing a signature. When users mint NFT for purchase this work, they must upload the Builder's signature on the price, thus ensuring the price data cannot be falsified.

- Non-fixed price (system pricing) mode

  When the unified fixed price is disabled, the Builder can designate his work as system pricing item by providing a signature. When users mint NFT for purchase this work, they must upload the Builder's signature on system pricing mode, thus ensuring the price mode cannot be falsified.

Note: The attribute that asset used for minting NFT has been changed from ETH to a user-defined InputToken is introduced in version 1.7, called Self-Selected InputToken Mode which is SeedNodes attribute.

2.7.2 System Pricing Strategy

The system pricing strategy is quite complex and supports the introduction of new templates. However, generally, it follows these rules:

- Each SubNodes has a floor price to be set. Typically, the first NFT minted in a SubNodes is at this floor price. The floor price cannot be zero.
- If an NFT under a certain Builder is minted multiple times within a round, each minting price will be twice the price of the previous minting (w.r.t. exponential pricing template, the factor of 2 can be set to other coefficients) or increased by a fixed amount. (w.r.t. linear pricing template) (hereafter, multiplying or dividing by 2 is used to represent price step changes).
- When a block ends, the minting price of a NFT is halved, but the price reduction stops at the base price.
- If the prices of all Builders' NFTs under the SubNodes are reduced to the floor price, the minting price for the NFTs in the next round will be half the floor price (this part is not affected by the template). During this period, if any work under the SubNodes is minted, the prices of all NFTs in that SubNodes revert to the floor price.

The factor of 2 mentioned above can be defined by templates into other pricing methods. Currently, the project supports exponential and linear pricing templates, and it

is possible to set the pricing coefficients. For the exponential pricing template, price changes by multiplying or dividing by the pricing coefficient; for the linear pricing template, price changes by adding or subtracting the pricing coefficient; in all cases, the price cannot drop below the floor price.

The pricing strategy and floor price can be set by owner.

2.7.3 Pass Card

When a SubNodes is created, a certain number of NFTs can be reserved as special NFTs. Their tokenIds range from 1 to the reserved quantity, and they have a specific URI prefix. The price of a Pass card, when the unified fixed price is not activated, is the floor price of the DAO. Pass cards belong to the Builder that is automatically created at the time of the SubNodes creation.

The tokenIds of all non-special NFTs start from the reserved quantity + 1.

2.8 MintFee Distribution Rules

When a user (minter) mints an NFT, they must pay InputToken for MintFee. MintFee will be distributed to the following address for the first time, where the distribution ratio is set by the owner.

- ProtocolFeePool, the commission pool of the SemiOS project. The proportion is specified by the SemiOS admin. In version 1.7 it is changed to 0.
- Builder, the uploader of the work (NFT mint for purchase).
- NodesRedeemPool, the redemption pool (OutputToken–>InputToken) for the SeedNodes.
- NodesAssetPool, the affiliated fund pool of the SubNodes.

Except the ProtocolFeePool ratio, other ratios can be set with two different values for system pricing and non-system pricing respectively.

In TopUp mode, no diversion occurs: all MintFee go directly into the Minter's TopUp account.

2.9 Block Reward Issuance Rules

When an NFT is minted within a SubNodes for the first time in a block, the block becomes active and triggers a block reward issuance. This means that all assets of the NodesAssetPool, including OutputToken and InputToken, are distributed according to the following rules:

2.9.1 Total Block Issuance Determination

Whether the assets are InputToken or OutputToken, the total amount of block reward issuance is always determined according to the following rules:

If the SubNodes has not activated the Lottery mode:

$$\text{Amount issued for current block} = \frac{\text{Total amount in NodesAssetPool}}{\text{Remaining blocks}}$$

The sources of assets in NodesAssetPool can be from the redirection of MintFee, the block rewards from other SubNodes, or assets directly injected by users. Notably, if the asset is OutputToken, after version 1.6, the SeedNodes owner can inject treasury funds into the NodesAssetPool.

Remaining blocks include the current block.

If the SubNodes has activated the Lottery mode:

$$X = Y * \frac{Z}{M + Z - 1}$$

- X: Amount issued for current block.
- Y: Total Asset in NodesAssetPool.
- Z: Lottery block.
- M: Remaining block.

The lottery block Z is the current block serial number minus the serial number of the last active block (if there is no active period, then it is 0), which is the total number of block for the accumulation of rewards (including the current period).

If the SubNodes activates the Infinite mode, then the total amount issued X will be the total assets.

2.9.2 Distribution Rules

After determining the total amount for current block reward issuance, the assets are allocated for the following purposes, according to ratios set by SubNodes owner (Note: different ratios can be set for InputToken and OutputToken separately):

- Transfer to the NodesAssetPool of several other SubNodes under the same SeedNodes, meaning that this SubNodes can designate several SubNodes (within the same SeedNodes) as the recipient SubNodes.
- Transfer to the NodesRedeemPool, which can only be set for InputToken.
- Used for internal distribution within this round.
- The portion not used in block reward and remains in the NodesAssetPool of this SubNodes, called reserved.

If the SubNodes activates the TopUp mode, then 100% of the total OutputToken block issuance is used for internal reward distribution, and all InputToken in the NodesAssetPool(under normal circumstances, neither MintFee nor block rewards enter the asset pool in TopUp mode) are transferred into the NodesRedeemPool.

2.10 Internal Distribution Rules

The portion of the block rewards used for internal distribution will be allocated to the following roles according to the set distribution coefficients and weights:

- The NFT minters in this block.
- The Builders of the NFTs minted in this block.
- The Starter (SubNodes creator).
- The ProtocolFeePool, in version 1.7 this ratio is set to be 0.

Among these, the distribution coefficient is a SubNodes parameter, and roles include the Minter, Builder, and Starter. Coefficients can be set separately for these roles as well as for InputToken and OutputToken. For a single NFT minting, the added weight for a role is the amount of the MintFee that flows into the NodesAssetPool multiplied by the role coefficient (when the role is Minter or Builder, only the case that user

mints the NFT or is the Builder of the NFT is included); the total weight increase is the amount of the MintFee that flows into the NodesAssetPool. It means that an NFT's contribution is related to its MintFee that distributes to the NodesAssetPool.

The final distribution ratio is the user's weight divided by the total weight.

$$X = Y * \frac{\Sigma_{\mathbf{Z}}(C * F)}{\Sigma_R F}$$

- X: User's claimable reward quantity for the block.
- Y: Total internal distribution for the block.
- Z: NFTs minted in the block.
- C: Role coefficient.
- R: All NFTs in the block.
- F: Amount of MintFee flowing into the NodesAssetPool.

If the SubNodes activates the TopUp mode, then by default, all internal distributions are allocated to Minter (i.e., the Minter role coefficient is 100%, and the formula remains unchanged). Moreover, the aforementioned amount of minting fees flowing into the NodesAssetPool(definition of F) is changed to the total MintFee, because under TopUp mode, InputToken does not flow into the NodesAssetPool.

If the SubNodes' global fixed price is set to zero, then the aforementioned F is regarded as 1, meaning that all NFTs contribute the same.

2.10.1 Reward Collection

The rewards from internal distribution for the current block can only be collected after the block ends. The contract supports a one-click collection where users can collect earnings from all roles, across all SubNodes, for all blocks where rewards have not yet been claimed.

The diagram below summarizes the basic process of asset flow within the DAO.

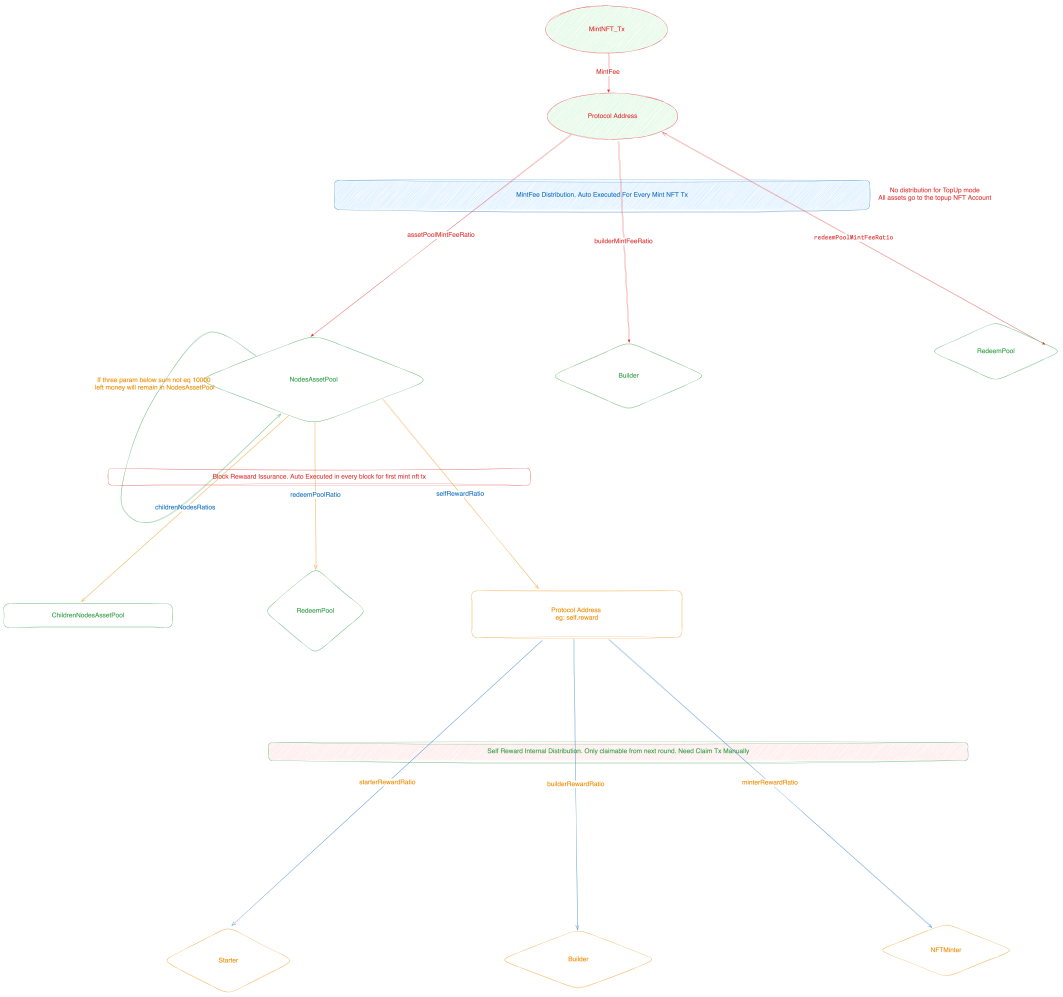

Figure 1: SubNodes asset flow

2.11 Redeem Rule

Users can exchange the received OutputToken for InputToken. The redemption rules do not apply to Import OutputTokenMode.

The circulating amount of OutputToken is defined as the total supply of OutputToken minus the OutputToken balance in NodesAssetPools of all series SubNodes, then minus the OutputToken balance in the NodesRedeemPool.

After version 1.6, the circulating amount of OutputToken need to be further reduced by the OutputToken balance in the treasury.

The amount of redeemed InputToken equals the number of OutputToken used for redemption multiplied by the InputToken balance in the NodesRedeemPool, then divided by the circulating amount of OutputToken.

2.12 Granting the Asset Pool

This is a new logic added in version 1.6.

Users can make grants to the NodesAssetPool or the treasury.

- If the choice is to make a grant to the NodesAssetPool, users can opt to use funds from their own wallet or from the treasury. Only the owner the SeedNodes can use the latter option. After making the payment, the user receives a "grantAssetPoolNFT" as proof of payment.
- If the choice is to make a grant to the treasury, users must use funds from their own wallet. After making the payment, the user receives a "grantTreasuryNFT" as proof of payment.

2.13 Permission Judgment

2.13.1 NFT Minting Permission Judgment

A SubNodes has complex settings for determining whether a user has the permission to mint NFTs. The overall process follows these steps:

- BlockMintCap (SubNodes attribute): the maximum number of NFTs that can be minted each block.
- Blacklist (SubNodes attribute): if a user's address is on this blacklist, their mint request is denied.
- NodesMintCap (SubNodes attribute): SubNodes global mint limit. If the Whitelist attribute is not enabled, this requires that the total number of mints of a user cannot exceed this limit. If met, minting is allowed.
- Whitelist (SubNodes attribute): If activated, it is divided into Minter Whitelist, NFT whitelist and a ERC721 Whitelist.
- If a user is on the Minter Whitelist:
- UserMintcap[] (SubNodes attribute): each whitelist user's mint limit (which can be infinite). In this case, the total number of mints a user makes cannot exceed the UserMintcap[the user] value. If met, minting is allowed. Note: Merkle tree system is used to determine whether a user is in minter Whitelist.

- If the user is not on the minter whitelist, go to NFT Whitelist:
- NFT Whitelist (SubNodes attribute, newly added in version 1.7) : a list of NFTs, each with a minting limit (which can be infinite). If a user owns any NFTs in the NFT Whitelist, the total number of mints the user makes cannot exceed the minting limit associate with the NFT. If met, minting is allowed. Note: If the user owns multiple NFTs in the NFT Whitelist, the smallest limit is used for evaluation.
- If the user is not on the NFT Whitelist, go to ERC721 Whitelist:
- ERC721 Whitelist (SubNodes attribute): a lists of ERC721 contract addresses, each with a minting limit (which can be infinite). If a user owns any NFTs from ERC721 contracts on the ERC721 Whitelist, the total number of mints the user makes cannot exceed the minting limit associate with the ERC721 contract; if met, minting is allowed; otherwise, it is not allowed. Note: If the user owns multiple NFTs from the ERC721 whitelist, the smallest limit is used for evaluation.
- If none of the above conditions are met, the mint request is denied.

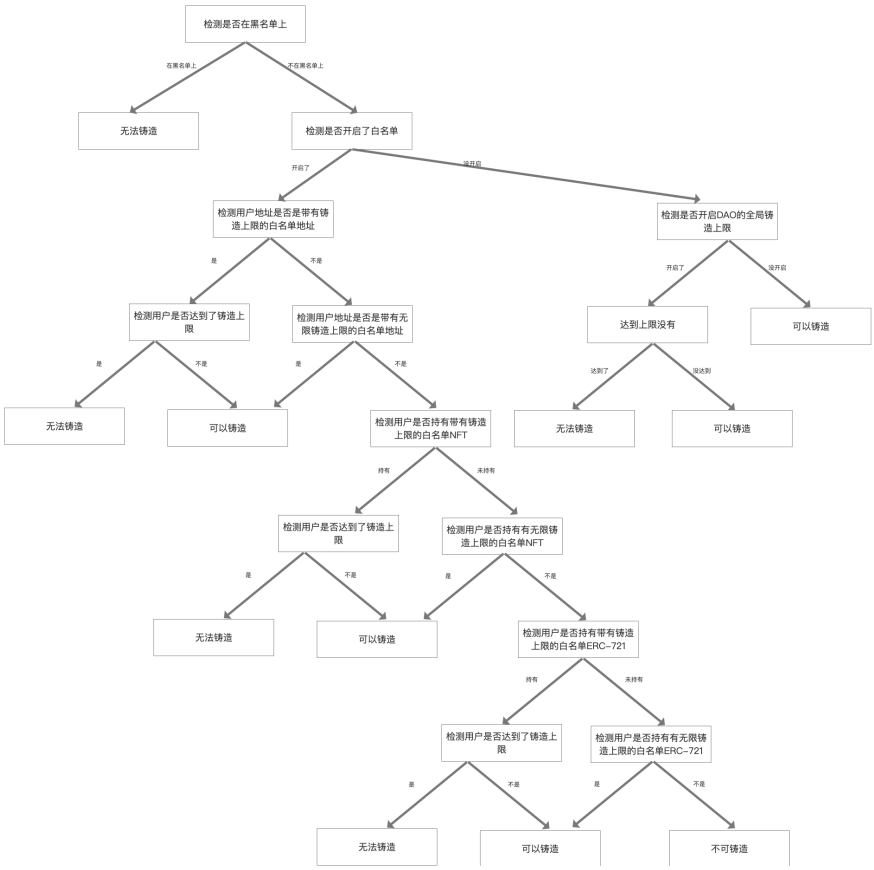

Figure 2: minter permission

2.13.2 Builder Permission Assessment

When a new Builder is to be created (identified by minting), the SubNodes assesses the permissions of the Builder's address as follows:

- Builder Blacklist (SubNodes attribute): If the Builder's address is on this blacklist, the minting is denied.
- Builder Whitelist, Builder NFT Whitelist and Builder ERC721 WhiteList (SubNodes attribute). If all attributes are disabled, the minting is allowed.
- If the Builder is on the Builder Whitelist, the minting is allowed. Note: Merkle tree system is used to determine whether a user is in Builder Whitelist.
- If the Builder owns any NFTs in Builder NFT Whitelist (a list of NFTs) or from ERC721 contracts in Builder ERC721 WhiteList, the minting is allowed.
- If none of these criteria are met, the minting is denied.

2.14 SubNodes Rights Tied to NFT

This is a feature proposed for version 1.6, and is implemented in 1.10 in the business layer. Specifically, when a SubNodes is created, its rights can be divided as follows:

- The "Edit Information" right. This right is unrelated to the contract itself and is used to modify parameters displayed on the website. The contract only records the owner of this right.
- The right to"Edit On-chain Parameters". This includes modifying on-chain properties of the SubNodes, such as changing the remaining block, entering or exiting infinite mode, setting the global NodesMintCap, the BlockMintCap, the floor price, the Unified Fixed Price, adjusting the NFT pricing strategy, setting all recipient SubNodes and their corresponding block reward distribution ratios, including NodesRedeemPool ratio (InputTokenonly), internal distribution ratio, and reserved ratio, as well as setting all role coefficients with respect to internal distribution.
- The right to "Edit Strategies". This involves modifying various blacklists and whitelists of the SubNodes.

- Revenue rights, which designate the one who can collect the reward of the Starter.

Additionally, if a SeedNodes is created, it introduces the following three additional rights:

- The right to "Edit SubNodes Information". This right is unrelated to the contract and is used to modify parameters displayed on the website. This right allows for editing the information of "SeedNodes website".
- The "Treasury Transfer Asset" right. This right allows for the transfer of treasury funds into the NodesAssetPool of any SubNodes within it, i.e. allows granting NodesAssetPool using funds from the treasury.
- The right to set the "TopUp Unlock Distribution Ratio", which includes setting a default ratio and setting ratios for one or more SubNodes within it. See the following subsections for detail.

All the aforementioned rights are tied to NFTs, meaning that only the owner of the corresponding NFT possesses these rights (in previous versions, all these rights were owned by the Starter). Additionally, the owner of each right has the option to bind this right to another NFT. When a SubNodes is initially created, its associate ERC721 contract will create an NFT with tokenId 0, and all four rights (seven rights for also creating a SeedNodes) are bound to this NFT, which is owned by the Starter. Subsequently, the Starter can distribute different rights to different NFTs by binding them accordingly.

2.15 Supplementary Explanation for Special Mode

2.15.1 Infinite Mode

- In Infinite Mode, the total amount of block reward issuance in each block equals all the assets in the NodesAssetPool.
- After turning Infinite Mode on or off, the data of the pricing strategy remains unchanged, and the block does not reset. For instance, if the price of the last non-fixed price NFT under a certain Builder in the current block is 0.01 ETH, then even if Infinite Mode is toggled, the minting price for the next NFT under that Builder would be 0.02 ETH (assuming an exponential pricing strategy with a pricing coefficient of 2).

- After turning off Infinite Mode, the lottery block are reset. Note: Turning on Infinite Mode is unrelated to the lottery round in the lottery mode, because Infinite Mode distributes all rewards.
- When Turning off Infinite Mode, it is necessary to re-specify the remaining block for the SubNodes.

2.15.2 TopUp Mode

New Concept: Savings Account, introduced for SubNodes operating in TopUp mode. A user's savings account is shared across the SeedNodes. A SubNodes in TopUp mode can be thought of as a fund where LPs inject capital by minting artworks. Using money from the savings account is akin to the fund making an investment, and successful investments unlock OutputToken as rewards proportionally.

In a SubNodes with TopUp mode, the InputToken for MintFee is transferred to their savings account at the end of the current block. The total block reward for each block in a TopUp SubNodes remains calculated as before, but theoretically, there would be no InputToken in the NodesAssetPool. The distribution rule for OutputToken is 100% for internal allocation. The OutputToken that users can claim are transferred to their savings accounts at the end of the round.

Using the savings account: When users mint any NFT in a non-TopUp SubNodes within the same SeedNodes, the required InputToken is first deducted from their savings account. Each portion of InputToken used from the savings account unlocks a corresponding proportion of OutputToken directly into the user's wallet. For example, if a user's savings account contains 1 ETH (as InputToken) and 2000 OutputToken, spending 0.3 ETH to mint an NFT would result in 600 OutputToken being transferred into the user's wallet (unlocking 30%).

Minting NFTs under a TopUp SubNodes does not allow the use of the savings account.

Version 1.4 update: When users use OutputToken from their savings account to mint NFT (only in SubNodes that have enabled OutputToken Payment Mode), the corresponding proportion of InputTokenis unlocked (implementing a refund logic).

Abstract Behavior of the Savings Account

- Minting an NFT in a TopUp SubNodes is equivalent to depositing money (In-

putToken) into the savings account; the source of funds can only be the user's wallet (cannot use money from another saving account).

- Minting an NFT in a non-TopUp SubNodes is equivalent to withdrawing/spending money from the savings account. Spending InputToken can unlock OutputToken in the saving account, and spending OutputTokencan unlock InputToken in the savings account (unlocking means directly transferring to the user's wallet).

Savings Account Bound to NFT This is a new logic added in version 1.5. The savings account is bound to an NFT's owner rather than a user's address, thus allowing the savings account to circulate as an asset package in the secondary market. Only the owner of the bound NFT can use the funds in the corresponding savings account.

When a user deposits money into the savings account by minting artwork in a TopUp SubNodes, they must specify an NFT identifier (including ERC721 address and tokenId) to indicate that the money is to be deposited into the savings account associated with that NFT. If the ERC721 address field of the NFT identifier specified by the user is zero address, then the NFT minted by the user becomes the NFT bound to the new savings account, essentially creating a new savings account. When a user spends money from the savings account, they can specify at most one NFT address, indicating that they are spending money from the savings account bound to that NFT, with the requirement that the user is the owner of this NFT. If the ERC721 address field of the specified NFT address is the zero address, it indicates that the user is not using any savings account.

NFT Locking This is new logic added in version 1.5. The owner of an NFT bound to a savings account can lock the NFT, specifying the block.number in Ethereum until which it is locked. During the lock period, the savings account bound to the NFT cannot be used for spending, and if that NFT's identifier is specified during minting, assets from the user's wallet will be used directly. Deposits into the savings account are still allowed during the lock period. Only the owner of the NFT can lock it and even if the NFT is transferred to others, it still remains locked until the end of the lock-up period. NFT locking applies to all SeedNodes in the project.

Savings Account Unlocking Diversion This feature was added in version 1.6. In previous versions, 100% of the unlocked assets from a savings account were trans-

ferred directly into the user's wallet. Now, a portion of the unlocked InputToken/OutputToken must be transferred to the NodesRedeemPool/NodesAssetPool. The proportion of this transfer is predetermined by the creator of the SeedNodes and is not specified in the SubNodes creation parameters.

The creator of the SeedNodes can set a default proportion for the unlocking diversion, which every newly created SubNodes will follow. Additionally, the SeedNodes creator can set different proportions across multiple affiliated SubNodes.

2.15.3 OutputToken Payment Mode

In this mode, the SubNodes requires users to mint NFTs using OutputTokens. This requires prior approval, or the OutputTokenmust support the permit method. When minting NFT using the savings account in this mode, a corresponding proportion of InputTokenis unlocked.

OutputToken payment mode and TopUp mode cannot be enabled simultaneously.

2.15.4 Self-Selected InputToken Mode

This is a SeedNodes feature introduced in version 1.7, which allows InputToken to be any ERC20 token selected by SeedNodes owner, instead of just ETH.

2.15.5 Import OutputToken Mode

This is a SeedNodes feature, which allows OutputToken to be any ERC20 token selected by SeedNodes owner, instead of SeedNodes generated ERC20 token.

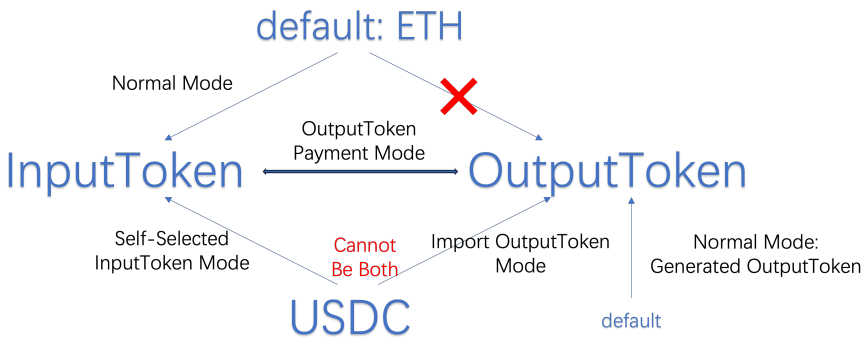

Figure 3: different SubNodes modes

2.15.6 Incentive Plan

This is new feature in version 1.8. Users can create an incentive Plan to reward saving accounts holders. A Plan is related to a SeedNodes and has following properties specified by its creator:

- Related SeedNodes id.
- The start time of the Plan, counted by Ethereum block.number.
- The number of blocks for which the Plan continues.
- The duration of each block, counted by Ethereum block.number.
- Total amount of tokens allocated for plan reward.
- The reward token address.
- Incentives InputToken or OutputToken, called incentive token.
- Whether the reward token is from the treasury. If so, the plan creator needs to have the "Treasury Transfer Asset" authorization, and the asset from treasury is used for the Plan. Otherwise, the assets from the creator's wallet are used.

Currently, the *DynamicPlan* template is used, where the total plan reward is uniform distributed in each block, and the reward in each round distributed to saving accounts holders, proportional to the incentive token amount in it at the end of the block.

It is notable that for TopUp SubNodes, the block rewards will not automatically enter the savings account after current block ends. Users need to manually trigger the contract to post the rewards to the savings account, only then will the Plan record it.

In conclusion, for a Plan that starts simultaneously with a TopUp SubNodes and shares the same block duration, in order to earn the PLAN reward, a user can mint a NFT in the TopUp SubNodes in block 1, and manually trigger the posing in block 2. Then the user is able to claim plan reward in block 3, to earn block 2's plan reward.

2.16 Nodes Default Template

The Nodes is currently created with the following default parameters:

- Floor price set at 0.01 ETH.

- Global fixed price set at 0.01 ETH.
- Reserved pass card quantity set at 1000.
- Pricing strategy set to exponential mode, with a pricing coefficient of 2.
- Initially mint 1,000,000,000 OutputTokenfor the treasury.
- Start time is the current block, with a round duration of one day.
- Starter is added to the Minter Whitelist, with a minting limit of 5.
- Associated ERC721 is added to the ERC721 Whitelist, with a minting limit of 5 for its holders. (Fission effect)
- All special modes are turned off.
- Other parameters are to be added...

```
struct CreateSemiDaoParam {
  bytes32 existDaoId;
  DaoMetadataParam daoMetadataParam;
  Whitelist whitelist;
  Blacklist blacklist;
  DaoMintCapParam daoMintCapParam;
  NftMinterCapInfo[] nftMinterCapInfo;
  NftMinterCapIdInfo[] nftMinterCapIdInfo;
  TemplateParam templateParam;
  BasicDaoParam basicDaoParam;
  ContinuousDaoParam continuousDaoParam;
  AllRatioParam allRatioParam;
  uint256 actionType;
}
```

3 SemiOS Contract

Due to version iterations, there may be two versions of contracts with prefixes D4A or PD, which may have an inheritance relationship. The current version primarily uses the PD prefix.

3.1 Authorizations

Node: All authorized address on the test network are assigned to the SemiOS developer address. The main network is a multisig address:
0x064D35db3f037149ed2c35c118a3bd79Fa4fE323

Key authorizations include:

- *DEFAULT_ADMIN_ROLE* of the main contract, which can assign addresses for all other authorizations.
- *PROTOCOL_ROLE* of the main contract, which can set all global attributes.
- *OPERATION_ROLE* of the main contract, which can set special uri prefix for pass cards.
- *SIGNER_ROLE* of the main contract, which can sign the price of the work like Builders, and the signature is recognized by the contract. This function is used for the SemiOS developer to assist builders with bulk signing.
- *DAO_ROLE* for the main contract, which can pause a specific SubNodes or Builder, that disables their NFT minting.
- *Owner* of the main contract, which can set facets for the diamond structure. The transfer of this ownership requires a second confirmation from the recipient.
- *royaltySplitterOwner* of the main contract, which is set as the owner of all RoyaltySplitters, with the ability to set distribution ratios. (It is set in *setRoyaltySplitterAndSwapFactoryAddress* function with *PROTOCOL_ROLE*)
- Owner of the *ProxyAdmin* of all pools (include NodesAssetPools and treasuries), which can change their implementation logic.
- *assetOwner* of the main contract, which is an important role that is:
  - The *DEFAULT_ADMIN_ROLE* of all generated OutputToken contracts.
  - The *DEFAULT_ADMIN_ROLE* of all NodesRedeemPooland NodesAssetPool contracts.
  - The *DEFAULT_ADMIN_ROLE* of all treasury contracts.
  - The *DEFAULT_ADMIN_ROLE* of all (three) generated ERC721-contracts.
- protocolFeePool of the main contract, the commission pool of SemiOS project.

- Owner of the *ProxyAdmin* for *NaiveOwner* and *PermissionControl* contracts.
- *DEFAULT_ADMIN_ROLE* of the *NaiveOwner* contract.

All of the following contracts inherit their corresponding interface (named I+ContractName), but currently, interface implementations do not use the override keyword.

3.2 Diamond Contract Structure

The main contract, *D4ADiamond*, adopts the diamond structure and inherits from the @solidstate contract library[1]. Hereafter, we will refer to the diamond structure's main contract as the *protocol*. Nearly all core business operations are executed by calling the *protocol* contract address.

The following diamond facets currently exist.

- *PDProtocol* serves as the default facet (*fallbackAddress*) and provides users with functions such as minting NFTs, claiming single-role rewards, and redeeming InputToken. After a user mints an NFT, the price template and reward template are invoked to update the pricing strategy and reward weight data. If it is the first minting within a block, it will also trigger the reward block process.

  This contract inherits from EIP-721 and is used to verify the Builder's signature on the NFT price. Its main parameters include:

  - *type = "MintNFT(bytes32 canvasID, bytes32 tokenURIHash, uint256 flatPrice)"*
  - *name= "ProtoDaoProtocol"*
  - *version = "1"*

- *PDCreate* provides users with the functionality to create a SubNodes, along with all associated components. These include:

  - The ERC-721 contract for works.
  - The ERC-721 contract for grantAssetPoolNFTs
  - The NodesAssetPool contract.
  - The RoyaltySplitter.

[1] `https://github.com/solidstate-network/solidstate-solidity`

If a seed node is also created, it also includes

  - The OutputToken contract
  - Treasury contract and the ERC-721 contracts for grant treasury NFTs

The core method is *createDao*. It returns a *daoId* as SubNodes ID.

- *PDProtocolReadable* provides all query methods, which are all view methods.
- *PDProtocolSettter* provides methods for setting parameters and attributes for the DAO, all of which are write methods and typically require authorization.
- *D4ASettings*, provides the setting of all global attributes and project authorization addresses, as well as corresponding query methods. This contract inherits from *AccessControl* (from @solidstate), and invoking relevant setting methods requires the *PROTOCOL_ROLE* authorization.
- *PDRound* calculates the block number in which the SubNodes is currently located and provides query methods. The core method is *getDaoCurrentRound*.
- *PDGrant* provides functionalities for granting funds to the treasury and NodesAssetPool, as well as issuing commemorative NFTs.
- *PDLock* provides functionalities for locking saving account bounded NFT and querying the NFT lock status.
- *PDPlan* provides functionalities for creating incentive plan and claim plan reward.

  The core method is *createPlan*. It returns a *planId* as Plan ID.

3.3 Contract Storage

In this project, storage libraries are categorized based on business logic. A single facet may access multiple storage libraries.

Each storage library's core variable is a *mapping* structure (key $\Rightarrow$ value), where the value is a *struct* containing relevant variables. Unless otherwise stated, all storage libraries use the SubNodes ID as the key.

- *DaoStorage* includes the fundamental attributes of the SubNodes, focusing on:

  - meta information: start block, remaining block, uri, index, RoyaltyFee ratio.
  - Associated contract addresses: NodesRedeemPool, OutputToken, InputToken.
  - NFT maximum supply and current NFT supply.
  - Pricing and Reward strategy templates
  - NFT-related limits

- *BasicDaoStorage* includes additional SubNodes attributes, focusing on:
  - NodesAssetPool address
  - Unified Fixed Price Mode status and unified fixed price.
  - Pass card numbers.
  - Special mode statuses.
  - Saving account unlocking diversion ratio.

  This storage is often used for new attributes introduced in updated versions.

- *CanvasStorage*, includes Builder related attributes, focusing on:
  - All belonging tokenIds to this Builder.
  - Index of the Builder.
  - The ID of the associated SubNodes.
  - Uri of the Builder.

- *PoolStorage*, uses the NodesRedeemPool address as the key (so the same key is shared across the SeedNodes), focusing on:
  - Records the asset amount of saving accounts.
  - Default ratio of saving account unlocking diversion.
  - Address of treasury and ERC721 for *grantTreasuryNFT*.
  - The ID list of all incentive plans.

- *TreeStroage* primarily records inheritance attributes between SubNodes, including:
  - The ID list of all recipient SubNodes for block reward distribution.

  - The distribution ratio of block rewards.
  - The ID list of all SubNodes in the same SeedNodes.
  - The distribution ratio of MintFee.
  - Role coefficients for internal distribution.

- *ProtocolStorage*, record some global attributes includes:
  - Whether an Uri has been used.
  - Latest SubNodes index.
  - NFT locking information, e.g., start time and duration.

- *PriceStorage* records:
  - The current minting price information for the Builder, using Builder ID as key.
  - The maximum minting price information under the SubNodes(considering depreciation)

- *RewardStroage*, records intermediate variables needed to calculate user reward, e.g., user's weight for internal distribution.

- *SettingsStorage*, records global attributes, including:
  - Factory contracts and other related contracts addresses.
  - Maximum and minimum RoyaltyFee ratio.
  - Pause status, can take any ID as key or just the project status.
  - *assetPoolOwner* and *royaltySplitterOwner*, the important authorization role.
  - All reward/price strategy templates.

- *RoundStorage*, records the SubNodes' block duration and the last modification details.

- *OwnerStorage*, records the NFTs that SubNodes rights tie. SubNodes rights take SubNodes ID as key and SeedNodes rights take NodesRedeemPool address as key.

- *PlanStorage*, takes Plan ID as key, records the plan's meta information and intermediate variables needed to calculate user's plan reward.

- *GrantStorage*, records the grant info for "grantAssetPoolNFT" and "grantTreasuryNFT".

3.4 Factory Contract

This project includes the following factory contracts:

- *PD4AERC721WithFilterFactory*, used for creating ERC721 contracts (*PD4AERC721WithFilter*) associated with the SubNodes. The introduction of the Filter is to ensure compatibility with OpenSea functionality. The creation process utilizes the minimal proxy contract pattern (EIP1167), which means *PD4AERC721WithFilter* contracts are non-upgradable.
- *D4AERC20Factory*, used for creating ERC20 contracts (*D4AERC20*) associated with the SeedNodes. The minimal proxy contract pattern is employed here, meaning the resulting *D4AERC20* contracts are non-upgradable.
- *D4AFeePoolFactory* used for creating SubNodes related fund pools (*D4AFeePool*). The *TransparentUpgradeableProxy* pattern is employed here, allowing the created *D4AFeePool* to be upgradable. Additionally, the factory contract will create **one** *ProxyAdmin* contract to facilitate the upgrade of all *D4AFeePool*. The *TransparentUpgradeableProxy* contract is inherited from OpenZeppelin 4.0[2].
- *D4ARoyaltySplitterFactory* is used for creating Royalty Fee splitting contracts (*D4ARoyaltySplitter*). The minimal proxy contract pattern is employed here, which means the resulting *D4ARoyaltySplitter* contracts are non-upgradable.
- *UniswapV2Factory* is used for creating Uniswap V2 trading pools. When creating a SubNodes, it is possible to choose to create associated trading pools for OutputToken and InputToken. It is an external contract.

3.5 SubNodes Generated Contract

This project includes the following SubNodes creation (and associated) contracts:

- *D4AERC20*, the previously mentioned ERC20 contract associated with the SeedNodes. This contract inherits from *AccessControlUpgradeable*[3], where the *DEFAULT_ADMIN_ROLE*

[2] `https://docs.openzeppelin.com/contracts/4.x/api/proxy`. It is notable that in OpenZeppelin 5.0, each transparent proxy contract has a separate *ProxyAdmin*, which is not suitable here

[3] `https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/access/AccessControlUpgradeable.sol`. It has slight differences from that of @solidstate.

authorization is transferred to the *assetOwner* of the protocol (via the *changeAdmin* function called in *PDCreate*), while the *MINTER* and *BURNER* authorizations are assigned to the protocol. The contract supports the "permit" feature, allowing users to perform approve operations by providing signatures.

- *PDERC721WithFilter*, the previously mentioned ERC721 contract associated with the SubNodes:
  - It inherits from *AccessControlUpgradeable*, where the *DEFAULT_ADMIN_ROLE* authorization is transferred to the *assetOwner* of the protocol, the *MINTER* and *ROYALTY* authorization are assigned to the protocol. The later has the authority to set the royalty fee information for NFTs (EIP-2981).
  - The method for minting NFTs in this contract is *mintItem*, which requires the parameter *tokenId*. If *tokenId* is 0, it means the NFT is not a pass card, and its number will increment sequentially from *startTokenId*, where *startTokenId* is set to the number of pass cards. If tokenId is non-zero, it means the NFT is a pass card, and its number will be equal to the provided tokenId.
  - To comply with OpenSea's requirements, this contract inherits from *Ownable*, with the owner being transferred to the Builder. The owner has the authority to set the *contractUri* (initially set to *DaoUri*).
- *D4AFeePool*, the previously mentioned fund pool contract, encompasses the implementations of all NodesAssetPool and NodesRedeemPool. This contract inherits from *AccessControlUpgradeable*, with the *DEFAULT_ADMIN_ROLE* transferred to *assetOwner* of the protocol, and the *AUTO_TRANSFER* permission assigned to the protocol. Both *DEFAULT_ADMIN_ROLE* and *AUTO_TRANSFER* have the authority to transfer funds from the pool. The contracts are upgradeable (*TransparentUpgradeableProxy*) and share a same *ProxyAdmin*.
- D4ARoyaltySplitter, the previously mentioned royalty splitting contract,
  - This contract inherits from *Ownable* and can set attributes such as the splitting ratios. The owner is transferred to *royaltySplitterOwner* of the protocol.
  - This contract inherits the Chainlink Keeper interface and can automatically trigger contract methods (via *performUpkeep*) when certain conditions are met (i.e., *checkUpkeep* returns true). The contract can automatically distribute ETH through the fallback function. However, as ERC20 tokens cannot be automatically distributed, received ERC20 tokens will be temporarily

held in this contract. When the value of any ERC20 token in the list (converted to ETH via the Uniswap router's *getAmountsOut* function) exceeds a set threshold (currently 5 ETH), the contract will convert all ERC20 tokens in the list to ETH using the Uniswap router (with a slippage setting of 99.5%, and prices sourced from Chainlink *oracleRegistry*). Distribution will then be handled through the fallback function.

Note: For contracts with non-empty fallback functions, it is not possible to directly call the withdraw method of the WETH contract. This is because the WETH contract uses the *transfer* method to transfer ETH, which limits the gas to 2300 and does not support the execution of any other logic within the fallback function. Consequently, this contract will store received WETH until performUpkeep is triggered to handle the distribution.

3.6 Template Contract

The template contracts are primarily used for calculating system pricing and internal block rewards. They are invoked via *delegate_call*. The project supports the ability to add new templates at any time.

- *LinearPriceVariation*, the linear pricing template, with the main method *getCanvasNextPrice*, is used to obtain the system price for the next system pricing NFT.
- *ExponentialPriceVariation*, the exponential pricing template, with the main method *getCanvasNextPrice,* is used to obtain the price for the next system pricing NFT.
- *UniformDistributionRewardIssuance* is currently the only block reward issuance template. Its methods include:
  - *getDaoRoundDistributeAmount*: Calculates the total block reward for a currently non-active block if someone mint NFT.
  - *getRoundReward*: Records the total reward for internal distribution in an active block.
  - *updateReward*: Updates the state based on block reward issuance.
  - *claimDaoCreatorReward* and other methods: Handle the reward claiming logic.

3.7 Independent Contract

The creation of independent contracts is due to historical reasons. In the future, consider merging all independent contracts into facets.

3.7.1 PermissionControl

This contract is used to determine permissions for minting NFTs and creating Builder.

The address of this contract is stored in *SettingsStorage*.

The contract introduces a structure *WhiteList*, which includes two Merkel Roots and two *address[ ]* arrays, corresponding to the permissions for minters and Builders. The Merkel Roots are used to determine if an address is on the WhiteList, while the address arrays are used to check if a user is a holder of any of the listed ERC721 tokens.

The contract also introduces a *BlackList* structure, which consists of two *address[ ]* arrays for minters and Builders, respectively. Blacklist does not involve Merkel Tree.

The contract inherits from EIP712 and supports setting permissions via signatures. Currently, this feature is not in use.

3.8 NaiveOwner

This contract is used to record and verify the initial owner of all SubNodesand all Builders, using their IDs as keys.

This contract supports *transferOwnership*.

This contract is upgradable.

It inherits from *AccessControlUpgradeable*, where *DEFAULT_ADMIN_ROLE* is transferred to the multisig address on the mainnet, while *INITIALIZER_ROLE* is assigned to the protocol, which is responsible for initializing the owners of each SubNodes and Builders.

3.9 D4AUniversalClaimer

This contract provides a one-click function to claim all rewards, including those as Starters, Builders, and minters under all SubNodes.

3.10 Supporting Contract

3.10.1 CutFacetFunction

This contract consists entirely of *pure* methods and is designed to retrieve all the selectors for a given *Interface* (returning a *bytes4[]* structure). This facilitates the construction of parameters for the diamondCut function in the diamond main contract.

A small trick is used where a dynamically-sized array bytes4 is pre-allocated with a length of 256. Elements are gradually added to selectors when the total number is unknown. Finally, the length of the array is reset to the actual number of selectors using assembly: *assembly* { *mstore(selectors, selectorIndex)* }.

Each time a new interface is added, its selectors need to be added to this contract. (Since each function is appended with an integrity check, there's no need to worry about missing or extra additions.)