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 token ID.
- Minter: The user that mints a NFT, who is also the NFT's owner.
- Main ERC721 contract: An ERC721 contract associated with a SubNodes, which manages all NFTs in the SubNodes.
- 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: The price of the work is a fixed price and determined by the Builder/by a specific parameter of the SubNodes.
- System Pricing: The price of the work is determined by the system and is referred to as the System Price.
- Floor Price: The typical price of the first Work in a SubNodes with respect to System Pricing.
- Block: Associate to a SubNodes, it is a time window unit for reward distribution and price change strategies. The block number starts incrementing from 1.
- Active Block: A block in which any NFT has been minted in the associate SubNodes.
- MintWindowDuration: The duration time of one Block, counted by Ethereum 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 Block.
- Internal Distribution: Assets distributed for rewarding different roles in a Block.
- Unified Fixed Price Mode: A special SubNodes mode where all Works are priced at a single Unified Fixed Price.
- TopUp Mode: A special SubNodes mode for fundraising purposes.
- Lottery Mode: A special SubNodes mode where non-active Blocks' earnings accumulate.
- Infinite Mode: A special SubNodes mode with infinite Blocks, where the Block Reward always 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.
- Plan: An incentive program that lasts for a specified period, designed to incentivize makers by distributing rewards to them

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 Main ERC721 Contract

When each SubNodes is created, an ERC721 contract is also established and associated with it called Main ERC721 contract. 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 Main 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 EIP721 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 Main 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 another 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, and the ID of the SubNodesis alse referred as the ID of the SeedNodes.

2.4 SeedNodes Associated Contracts

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

2.4.1 OutputToken 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 by minting to the Treasury. 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 new 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 Main 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 Block. For example, rewards distributed within a Block are calculated and disbursed when the Block is over. Unique rules apply to NFT System Pricing strategies within a Block, 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 Block duration is reset, the current Block 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 Block. With each passing Block, SubNodes' remaining Block decrease by one. The remaining Block of a SubNodes can be set by owner. When a SubNodes' remaining Block reach zero, the SubNodes ends, during which no NFT can be minted. The owner can revive an ended SubNodes by setting a non-zero remaining Block (or switching to Infinite Mode), after which all data on active Block 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 Block, 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 Block 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 Mode 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 (The actual rules applied depend on the selected reward template. Currently, there is only one type of template with the following description):

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:

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

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.

The count of the Remaining Block 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 Block.
- 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.

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

- X: User's claimable reward quantity of the Block.
- Y: Total Internal Distribution of 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 Claim

The rewards from Internal Distribution for the current Block can only be claimed after the Block ends. The contract supports a one-click claim where users can claim rewards 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.

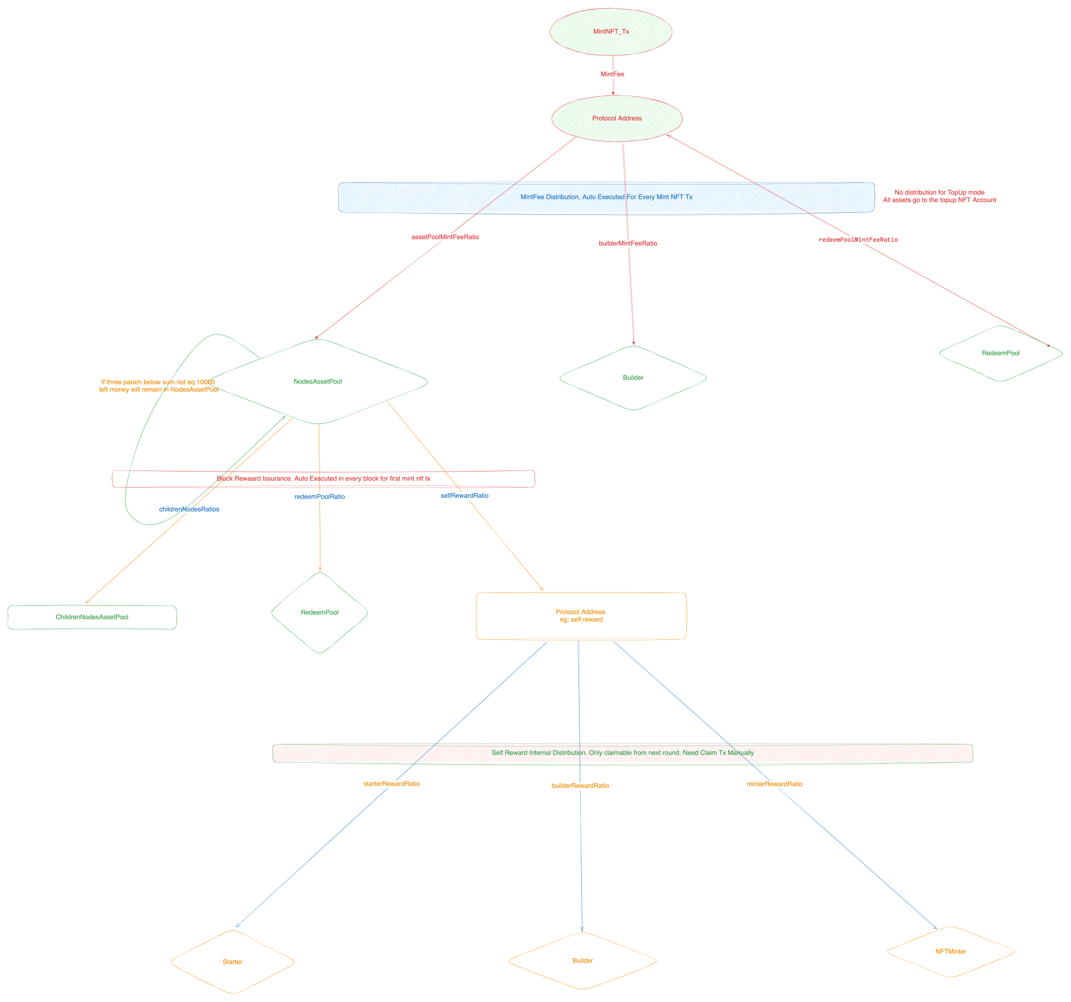
MintNFT_Tx
MintFee
Protocol Address
MintFee Distribution. Auto Executed For Every Mint NFT Tx
No distribution for TopUp mode
All assets go to the topup NFT Account
assetPoolMintFeeRatio
builderMintFeeRatio
redeemPoolMintFeeRatio
If three param below sum not eq 10000
left money will remain in NodesAssetPool
NodesAssetPool
Builder
RedeemPool
Block Rewaard Issurance. Auto Executed in every block for first mint nft tx
childrenNodesRatios
redeemPoolRatio
selfRewardRatio
ChildrenNodesAssetPool
RedeemPool
Protocol Address
eg: self.reward
Self Reward Internal Distribution. Only claimable from next round. Need Claim Tx Manually
starterRewardRatio
builderRewardRatio
minterRewardRatio
Starter
Builder
NFTMinter

Figure 1: SubNodes asset flow

2.11 Redeem Rule

Users can exchange the received OutputToken for InputTokenwith respect to a SeedNodes. The redemption rules do not apply to Import OutputTokenMode.

The circulating amount of OutputToken is defined as the total supply of OutputToken minus the total OutputToken balance in all NodesAssetPools of the SeedNodes, 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.

Granters can make grants to the NodesAssetPool or the Treasury.

- If the choice is to make a grant to the NodesAssetPool, granters can opt to use funds from their own wallet or from the Treasury. Only the has the “Treasury Transfer Asset" right 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, granters 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 Minter related 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.
- Minter related Whitelist (SubNodes attribute): If activated, it is divided into Minter Whitelist, Minter NFT whitelist and Minter 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 Minter NFT Whitelist:
- Minter 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 Minter 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 Minter NFT Whitelist, the smallest limit is used for evaluation.
- If the user is not on the Minter NFT Whitelist, go to Minter ERC721 Whitelist:
- Minter 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 Minter 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 Minter ERC721 whitelist, the smallest limit is used for evaluation.
- If none of the above conditions are met, the mint request is denied.

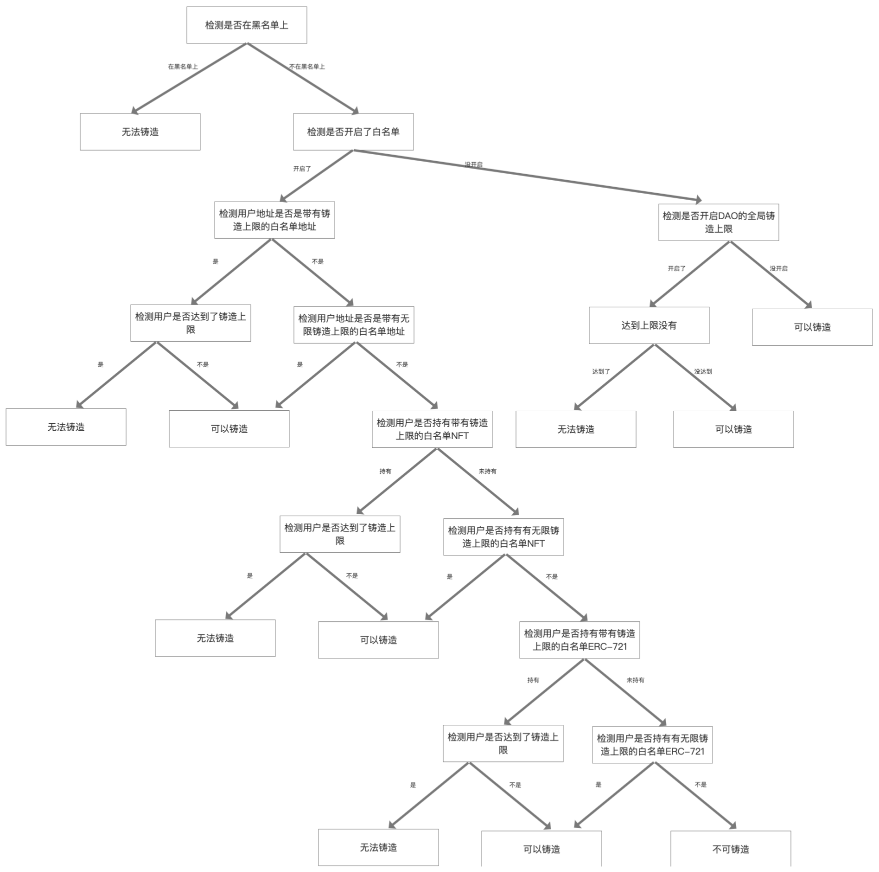
检测是否在黑名单上
在黑名单上
不在黑名单上
无法铸造
检测是否开启了白名单
开启了
没开启
检测用户地址是否是带有铸造上限的白名单地址
检测是否开启DAO的全局铸造上限
是
不是
开启了
没开启
检测用户是否达到了铸造上限
检测用户地址是否是带有无限铸造上限的白名单地址
达到上限没有
可以铸造
是
不是
是
不是
达到了
没达到
无法铸造
可以铸造
检测用户是否持有带有铸造上限的白名单NFT
无法铸造
可以铸造
持有
未持有
检测用户是否达到了铸造上限
检测用户是否持有无限铸造上限的白名单NFT
是
不是
是
不是
无法铸造
可以铸造
检测用户是否持有带有铸造上限的白名单ERC-721
持有
未持有
检测用户是否达到了铸造上限
检测用户是否持有无限铸造上限的白名单ERC-721
是
不是
是
不是
无法铸造
可以铸造
不可铸造

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.
- Starter reward 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 SeedNodes 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 “Transfer Treasury 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 TopUp Account Unlocking Distribution ratio for default or 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 Main 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. Note that if the NFT tied to Starter reward rights is transferred to someone else, all unclaimed Starter rewards will be transferred along with it.

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 Block 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: TopUp Account, introduced for SubNodes operating in TopUp mode. A user's TopUp 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 TopUp Account is akin to the fund making an investment, and successful investments unlock OutputToken as rewards proportionally.

In a SubNodes with TopUp mode, the 100% InputToken for MintFee is transferred to Minter specified TopUp 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 Block Reward issuance rule for OutputToken is 100% for Internal Distribution. The OutputToken that users can claim are transferred to their TopUp Accounts at the end of the Block.

Using the TopUp Account: When users mint any NFT in a non-TopUp SubNodes within the same SeedNodes, the required InputToken is first deducted from their TopUp Account. Each portion of InputToken used from the TopUp Account unlocks a corresponding proportion of OutputToken directly into the user's wallet. For example, if a user's TopUp 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 TopUp Account.

Version 1.4 update: When users use OutputToken from their TopUp 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 TopUp Account

- Minting an NFT in a TopUp SubNodes is equivalent to depositing money (InputToken) into the TopUp Account; the source of funds can only be the user's wallet (cannot use money from another TopUp Account).
- Minting an NFT in a non-TopUp SubNodes is equivalent to withdrawing/spending money from the TopUp Account. Spending InputToken can unlock OutputToken in the TopUp Account, and spending OutputTokencan unlock InputToken in the TopUp Account (unlocking means directly transferring to the user's wallet).

TopUp Account Bound to NFT This is a new logic added in version 1.5. The TopUp Account is bound to an NFT's owner rather than a user's address, thus allowing the TopUp 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 TopUp Account. ß When a user deposits money into the TopUp 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 TopUp 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 TopUp Account, essentially creating a new TopUp Account. When a user spends money from the TopUp Account, they can specify at most one NFT address, indicating that they are spending money from the TopUp 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 TopUp Account.

NFT Locking This is new logic added in version 1.5. The owner of an NFT bound to a TopUp Account can lock the NFT, specifying the duration of the lock, counted by Ethereum block.number. During the lock period, the TopUp 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 TopUp 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.

TopUp Account Unlocking Distribution This feature was added in version 1.6. In previous versions, 100% of the unlocked assets from a TopUp Account were transferred directly into the Maker's wallet. Now, a portion of the unlocked InputToken/OutputToken must be transferred to the NodesRedeemPool/Treasury. 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 TopUp 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.

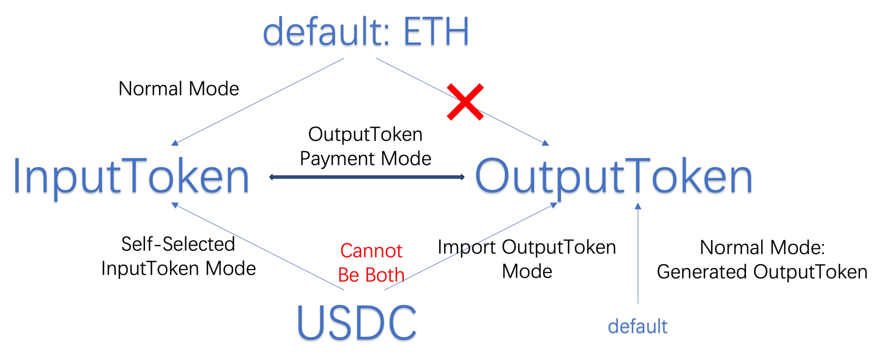

Figure 3: different SubNodes modes

2.16 Incentive Plan

This is new feature in version 1.8. Users can create an incentive Plan to reward Markers (TopUp 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. Note that The Block for plan and the Block for SubNodes are unrelated; it is used as the minimum cycle for distributing rewards in the Plan
- 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 Block distributed to TopUp Accounts holders, proportional to the incentive token amount in it at the end of the Block.

It is notable that for TopUp SubNodes, the MintFee and Block Rewards will not automatically enter the TopUp Account after current Block ends. Makers need to manually trigger the contract to post the rewards to the TopUp 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 maker 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.

2.17 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 Block duration of one day.
- Starter is added to the Minter Whitelist, with a minting limit of 5.
- Associated Main ERC721 address 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…

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. For the same reason, the contract code often uses 'DAO' to refer to SubNodes.

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.
- 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 Block Reward issuance 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"*

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

- *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.

  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. It inherits from *D4AProtocolReadable*, which contains the view methods for old versions.

- *PDProtocolSettter* provides methods for setting parameters and attributes for the DAO, all of which are write methods and typically require authorization. It inherits from *D4AProtocolSetter*, which contains the setting methods for old versions.

- *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. It also inherits from *D4ASettingsReadable*, which includes all the view methods.

- *PDRound* calculates the Ethereum 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.

[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

- *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 (OutputToken) associated with the SeedNodes. This contract inherits from *AccessControlUpgradeable*[3], where the *DEFAULT_ADMIN_ROLE* 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 contracts 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 Starter. 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

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

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 split the royalty fee of NFTs to two addresses with corresponding splitting ratios, which are set when initializing the 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 the System Price 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 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.
- *DynamicPlan* is currently the only Plan template. Its main function includes updating Plan status and claiming Plan rewards. Prefix sum algorithm is used for this template.

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.

3.7.2 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.

3.7.3 D4AUniversalClaimer

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

3.8 Supporting Contract

3.8.1 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.)

4 Contract Interface

In this section we introduce all public interfaces of SemiOS contract, excluding those requiring *PROTOCOL_ROLE*.

Note: All token-related quantity parameters in the contract, such as various prices and issuance amounts, are represented by shifting the decimal point to the left by the number of decimals corresponding to the token. For example, for prices of 1 USDC and 1 ETH, the values recorded in the contract and the parameters should passed when interacting with the contract are 1e6 and 1e18, respectively.

Note: All provided URIs must be unique, regardless of their type; otherwise, the contract will revert.

Note: All provided Builder IDs (*canvasId*) must be unique, regardless of SubNodes they belong; otherwise, the contract will revert.

4.1 IPDCreate

In this facet there is only one public function:

Interface Description:

```
function createDao(CreateSemiDaoParam calldata createDaoParam) external payable
  returns (bytes32 daoId);
```

This interface is used to create a SubNodes. The function takes a structure as a parameter and returns the ID of the newly created SubNodes(of type ‘bytes32‘).

Interface Parameters: createDaoParam

```
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;
  }
```

This structure contains all the necessary parameters for the creation of a SubNodes, including the existing SubNodes ID, metadata parameters, whitelist and blacklist information, mint cap parameters, template parameters, and more. In the following, all parameters are associated with the SubNodes that to be created.

existDaoId The ID of an existing SubNodes, used to link to the SeedNodes that the existing SubNodes belongs to, represented as a 'bytes32'. The provided existing SubNodes must be the first of an SeedNodes. If a new SeedNodes is to be create, the parameter can be any value.

daoMetadataParam

```
  struct DaoMetadataParam {
    uint256 startBlock;
    uint256 mintableRounds;
    uint256 duration; // blocknumber * 1e18
    uint256 floorPrice;
    uint256 maxNftRank;
    uint96 royaltyFee;
    string projectUri;
    uint256 projectIndex;
  }
```

Contains metadata related to the SubNodes. The detailed fields are described as follows:

startBlock The Ethereum block.number at which the SubNodes starts, represented as a 'uint256'. If zero value is provided, it means staring from the current Ethereum block of the transaction.

mintableRounds The number of initial Blocks, represented as a 'uint256'.

duration The duration of the SubNodes Block, expressed as the Ethereum block.number multiplied by 1×10^{18}, represented as a 'uint256'.

floorPrice The Floor Price, represented as a 'uint256'.

maxNftRank maxNftRank: The rank of total amount of NFTs that can be minted, with the current setup including five ranks corresponding to the following quantities: [1000, 5000, 10,000, 50,000, 100,000].

royaltyFee The RoyaltyFee, represented as a ‘uint96’.

projectUri A URI pointing to the SubNodes’ metadata or additional information, represented as a ‘string’.

projectIndex The SubNodes’ index , represented as a ‘uint256’. This is used to create a SubNodes with index less than a certain *reserved-DaoAmount*, and required to be triggered in a special action type. For regular SubNodes, the index increment by 1 with each creation and the variable has no effect.

whitelist

```
struct Whitelist {
  bytes32 minterMerkleRoot;
  address[] minterNFTHolderPasses;
  NftIdentifier[] minterNFTIdHolderPasses;
  bytes32 canvasCreatorMerkleRoot;
  address[] canvasCreatorNFTHolderPasses;
  NftIdentifier[] canvasCreatorNFTIdHolderPasses;
}
```

A list of addresses or entities that are permitted to interact with the SubNodes, including minting an NFT or becoming a builder. The detailed fields are described as follows:

minterMerkleRoot The Merkle root used to verify addresses that are in the Minter Whitelist with infinite minting limit, represent as a ‘bytes32’. It provides a compact proof mechanism to check if an address is part of the allowed list.

minterNFTHolderPasses An array of addresses that specifies the ERC721 Whitelist with infinite minting limit.

minterNFTIdHolderPasses An array of *NftIdentifier* struct that specifies the NFT WhiteList with infinite minting limit. The struct *NftIdentifier* can precisely identify a specific NFT, which is

```
struct NftIdentifier {
  address erc721Address;// represent the associated ERC721
address of the NFT
  uint256 tokenId;// represent the token ID of the NFT
}
```

canvasCreatorMerkleRoot The Merkle root used to verify addresses that are in the Builder WhiteList, represent as a 'bytes32'

canvasCreatorNFTHolderPasses An array of addresses that specifies the Builder ERC721 WhiteList.

canvasCreatorNFTIdHolderPasses An array of *NftIdentifier* struct that specifies the Builder NFT WhiteList.

blacklist

```
struct Blacklist {
  address[] minterAccounts;
  address[] canvasCreatorAccounts;
}
```

A list of addresses or entities that are prohibited from interacting with the SubNodes, including minting an NFT or becoming a builder. The detailed fields are described as follows:

minterAccounts An array of addresses that are prohibited from minting NFTs under the SubNodes.

canvasCreatorAccounts An array of addresses that are prohibited from becoming Builders under the SubNodes.

DaoMintCapParam

```
struct DaoMintCapParam {
  uint32 daoMintCap;
  UserMintCapParam[] userMintCapParams;
}
```

Parameters related to the minting NFT limits for the SubNodes. The detailed fields are described as follows:

daoMintCap The (global) maximum number of NFTs that can be minted by an address, represented by 'uint32'. It only works when the all kinds of minter related Whitelist is empty. If the provided value is zero, it means that no restriction is applied for this item.

userMintCapParams A struct array that specifies the Minter Whitelist with finite minting limit. The struct *UserMintCapParam* specifies the user addresses in the Minter Whitelist and the corresponding minting limit:

```
struct UserMintCapParam {
    address minter; // user address in the Minter Whitelist
    uint32 mintCap; // minting limit of the user
  }
```

nftMinterCapInfo A struct array that specifies the Minter ERC721 Whitelist with minting limit. The struct *NftMinterCapInfo* specifies the ERC721 addresses in the Minter ERC721 Whitelist and the corresponding minting limit:

```
struct NftMinterCapInfo {
  address nftAddress; //ERC721 address in the Minter ERC721 Whitelist
  uint256 nftMintCap; // minting limit of the owner of any NFT from
the ERC721 address
}
```

nftMinterCapIdInfo A struct array that specifies the Minter NFT Whitelist with minting limit. The struct *NftMinterCapIdInfo* specifies the *NftIdentifier* in the Minter NFT Whitelist and the corresponding minting limit:

```
struct NftMinterCapIdInfo {
  address nftAddress; // ERC721 address that the NFT in the Minter
NFT Whitelist from
  uint256 tokenId; // Token ID of the NFT
  uint256 nftMintCap; // minting limit of the NFT's owner
}
```

templateParam

```
struct TemplateParam {
  PriceTemplateType priceTemplateType;
  uint256 priceFactor;
  RewardTemplateType rewardTemplateType;
  uint256 rewardDecayFactor;
  bool isProgressiveJackpot;
}
```

Template contract type for System Pricing and Block Reward Issuance rule. The detailed fields are described as follows:

priceTemplateType This parameter specifies the type of System Pricing template. It is an enumeration type represented by *PriceTemplateType*:

```
enum PriceTemplateType {
  EXPONENTIAL_PRICE_VARIATION, // stands for LinearPriceVariation
  LINEAR_PRICE_VARIATION // stands for ExponentialPriceVariation
}
```

priceFactor This parameter represents the price variation range of type 'uint256'. For exponential price, It is the price multiplier multiplied by 10,000; for example, if the price doubles for each mint, this value would be 20,000. For linear price, it represents the specific amount of price change for each mint, usually related to the decimal of the InputToken.

rewardTemplateType This parameter specifies the type of Block Reward Issuance template. It is an enumeration type represented by *RewardTemplateType*:

```
  enum RewardTemplateType { UNIFORM_DISTRIBUTION_REWARD } //
stands for current rule
```

rewardDecayFactor This parameter represents the additional factor used for Block Reward Issuance template. Currently the rule does not use it and it can be any value of type 'uint256'.

isProgressiveJackpot This boolean parameter indicates whether Lottery mode is activated. If 'true', Lottery mode is active.

basicDaoParam

```
struct BasicDaoParam {
  bytes32 canvasId;
  string canvasUri;
  string daoName;
}
```

Additional parameters for the SubNodes. The detailed fields are described as follows:

canvasId The ID of the first Builder.

canvasUri A URI string representing the first Builder.

daoName A string that specifies the name of the SubNodes, which is also the name of the associated OutputToken contract (if created for SeedNodes) and all associate ERC721 contracts.

continuousDaoParam

```
struct ContinuousDaoParam {
 uint256 reserveNftNumber;
 bool unifiedPriceModeOff;
 uint256 unifiedPrice;
 bool needMintableWork;
 uint256 dailyMintCap;
 //1.3add-------------------//block reward distribution ratios
 bytes32[] childrenDaoId;
 uint256[] childrenDaoOutputRatios;
 uint256[] childrenDaoInputRatios;
 uint256 redeemPoolInputRatio;
 uint256 treasuryOutputRatio;
 uint256 treasuryInputRatio;
 uint256 selfRewardOutputRatio;
 uint256 selfRewardInputRatio;
 bool isAncestorDao;
 address daoToken;
 bool topUpMode;
 //1.4add--------------------
 bool infiniteMode;
 bool outputPaymentMode;
 //1.6add--------------------
 string ownershipUri;
 //1.7add--------------------
 address inputToken;
}
```

Parameters specific to newly added attributes with versions. The detailed fields are described as follows:

reserveNftNumber The number of reserved NFTs (Pass Cards), represented by 'uint256'. It can not be zero if the *needMintableWork* parameter is true.

unifiedPriceModeOff A boolean indicating whether the Unified Fixed Price mode is deactivated. If 'true', it is activated.

unifiedPrice The Unified Fixed Price for Unified Fixed Price mode, represented by 'uint256'.

needMintableWork A boolean that specifies whether Pass Cards are reserved. If 'false', there is no Pass Card and the *reserveNftNumber* parameter can be any number.

dailyMintCap The minting limit in each Block of the SubNodes, represented by ‘uint256’.

childrenDaoId An array of ‘bytes32’ representing the IDs of the children SubNodes (recipient SubNodes).

childrenDaoOutputRatios An array of ‘uint256’ representing the OutputToken ratios of children SubNodes for Block Reward Issuance, with *Basis_Point = 10000*. If means that if a ratio is 50%, then corresponding value should be 5000.

childrenDaoInputRatios An array of ‘uint256’ representing the InputToken ratios of children SubNodes for Block Reward Issuance, with *Basis_Point = 10000* (In the following all ratios are with *Basis_Point = 10000*).

redeemPoolInputRatio The InputToken ratio of the NodesRedeemPool for Block Reward Issuance, represented by ‘uint256’.

treasuryOutputRatio The OutputToken ratio of the Treasury for Block Reward Issuance, represented by ‘uint256’.

treasuryInputRatio The InputToken ratio of the Treasury for Block Reward Issuance, represented by ‘uint256’.

selfRewardOutputRatio The OutputToken ratio of Internal Distribution for Block Reward Issuance, represented by ‘uint256’.

selfRewardInputRatio The InputToken ratio of Internal Distribution for Block Reward Issuance, represented by ‘uint256’.

Note: The sum of all above InputToken/OutputToken ratios must be less or equal to 10000.

isAncestorDao A boolean indicating whether a new SeedNodes should created. If ‘true’, a SeedNodes is created and the SubNodes is the first of the SeedNodes.

daoToken The OutputToken address upon Import OutputToken Mode activated, represented by ‘address’. If zero address is provided, it means that Import OutputToken Mode is not activated and SubNodes generated ERC20 contract will be used as OutputToken.

topUpMode A boolean indicating whether the TopUp mode is activated. If ‘true’ it is activated.

infiniteMode A boolean that specifies whether Infinite Mode is activated. If ‘true’ it is activated.

outputPaymentMode A boolean indicating whether OutputToken Payment Mode is activated. If 'true' it is activated. It cannot be activated simultaneously with the TopUp mode.

ownershipUri The URI of the initial NFT associate with rights, that is, the NFT with zero token ID from the associate Main ERC721 contract of the SubNodes, represented by 'string'.

inputToken The OutputToken address upon Self-Selected InputToken Mode activated. If the value is zero address, it means that Self-Selected InputToken Mode is not activated and ETH will be used as InputToken.

allRatioParam

```
struct AllRatioParam {
  //mint fee ratios
  uint256 canvasCreatorMintFeeRatio;
  uint256 assetPoolMintFeeRatio;
  uint256 redeemPoolMintFeeRatio;
  uint256 treasuryMintFeeRatio;
  // add l.protocolMintFeeRatioInBps should be 10000
  uint256 canvasCreatorMintFeeRatioFiatPrice;
  uint256 assetPoolMintFeeRatioFiatPrice;
  uint256 redeemPoolMintFeeRatioFiatPrice;
  uint256 treasuryMintFeeRatioFiatPrice;
  // add l.protocolMintFeeRatioInBpsFiatPrice

  //output reward ratio
  uint256 minterOutputRewardRatio;
  uint256 canvasCreatorOutputRewardRatio;
  uint256 daoCreatorOutputRewardRatio;
  // add l.protocolOutputRewardRatio,

  //input reward ratio, add l.protocolInputRewardRatio
  uint256 minterInputRewardRatio;
  uint256 canvasCreatorInputRewardRatio;
  uint256 daoCreatorInputRewardRatio;
}
```

Parameters that MintFee distribution ratios and role coefficients for Internal Distribution. The detailed fields are described as follows, all represented as 'uint256':

canvasCreatorMintFeeRatio MintFee ratio of the Builder, with respect to System Pricing Mode (with *Basis_Point* = *10000*, and the same applies below).

assetPoolMintFeeRatio MintFee ratio of the NodesAssetPool, with respect to System Pricing Mode.

redeemPoolMintFeeRatio MintFee ratio of the NodesRedeemPool, with respect to System Pricing Mode.

treasuryMintFeeRatio MintFee ratio of the Treasury, with respect to System Pricing Mode.

Note: The sum of the above four parameters must be equal to 10000 minus the value returned by the *mintProtocolFeeRatio()* function.

canvasCreatorMintFeeRatioFiatPrice MintFee ratio of the Builder, with respect to non-System Pricing Mode.

assetPoolMintFeeRatioFiatPrice MintFee ratio of the NodesAssetPool, with respect to non-System Pricing Mode.

redeemPoolMintFeeRatioFiatPrice MintFee ratio of the NodesRedeem-Pool, with respect to non-System Pricing Mode.

treasuryMintFeeRatioFiatPrice MintFee ratio of the Treasury, with respect to non-System Pricing Mode.

Note: The sum of the above four parameters must be equal to 10000 minus the value returned by the *mintProtocolFeeRatio()* function.

minterOutputRewardRatio The role coefficient of Minter for Output-Token Internal Distribution (with *Basis_Point = 10000*, and the same applies below).

canvasCreatorOutputRewardRatio The role coefficient of Builder for OutputToken Internal Distribution.

daoCreatorOutputRewardRatio The role coefficient of Starter for Out-putToken Internal Distribution.

Note: The sum of the above three parameters must be equal to 10000 minus the value of *protocolOutputRewardRatio* in *SettingsStorage* (Currently, there is no public query interface available).

minterInputRewardRatio The role coefficient of Minter for InputTo-ken Internal Distribution

canvasCreatorInputRewardRatio The role coefficient of Starter for InputToken Internal Distribution.

daoCreatorInputRewardRatio The role coefficient of Starter for Out-putToken Internal Distribution.

Note: The sum of the above three parameters must be equal to 10000 minus the value of *protocolInputRewardRatio* in *SettingsStorage* (Currently, there is no public query interface available).

actionType The type of action or operation being performed, currently using the default value 20.

Interface Return Value: **DaoId** The return value is of type ‘bytes32’, representing the ID of the newly created SubNodes.

We recommend using the following JavaScript code to generate the Merkle Root and obtain the Merkle Proof:

getMerkleProofFoundry.js

```
const { StandardMerkleTree } = require("@openzeppelin/merkle-tree");
const { ethers } = require("ethers");

function buildMerkleTree(accounts) {
  const tree = StandardMerkleTree.of(
  accounts.map((account) => [account]),
  ["address"]
  );

  return tree;
}

function getProof(tree, account) {
  for (const [i, v] of tree.entries()) {
    if (v[0] === account) {
      const proof = tree.getProof(i);
      const abiCoder = ethers.AbiCoder.defaultAbiCoder();
      process.stdout.write(abiCoder.encode(["bytes32[]"], [proof]));
    }
  }
}

const tree = buildMerkleTree(
process.argv[2].includes(" ") ? process.argv[2].split(" ") : [process.argv[2]]
);
getProof(tree, process.argv[3]);
```

getMerkleRootFoundry.js

```
const { StandardMerkleTree } = require("@openzeppelin/merkle-tree");
const { ethers } = require("ethers");

function buildMerkleTree(accounts) {
  const tree = StandardMerkleTree.of(
  accounts.map((account) => [account]),
  ["address"]
  );

  return tree;
}

const tree = buildMerkleTree(
process.argv[2].includes(" ") ? process.argv[2].split(" ") : [process.argv[2]]
);

process.stdout.write(tree.root);
```

Example

Here is an example command to invoke the interface using ‘cast’:

```
cast send 0x60E771d7E4B7A8f8E7Fdc28d6E3852A4c556e546 "createDao((bytes32,(uint256,
  uint256,uint256,uint256,uint256,uint96,string,uint256),(bytes32,address[],(
  address,uint256)[],bytes32,address[],(address,uint256)[]),(address[],address[])
  ,(uint32,(address,uint32)[]),(address,uint256)[],(address,uint256,uint256)[],(
  uint8,uint256,uint8,uint256,bool),(bytes32,string,string),(uint256,bool,uint256,
  bool,uint256,bytes32[],uint256[],uint256[],uint256,uint256,uint256,uint256,
  uint256,bool,address,bool,bool,bool,string,address),(uint256,uint256,uint256,
  uint256,uint256,uint256,uint256,uint256,uint256,uint256,uint256,uint256,uint256,
  uint256),uint256))" '(0
  x0000000000000000000000000000000000000000000000000000000000000000
  ,(0,52,100,10000000000000000,2,1000,'http://project.uri',0),(0
  x0000000000000000000000000000000000000000000000000000000000000000,[],[],0
  x0000000000000000000000000000000000000000000000000000000000000000,[],[]),([],[])
  ,(0,[]),[],[],(0,20000,0,0,false),(0
  x73b3277727e0a49ccf380a360985688769d069601c919427422fafb0fe959ef3,'http://canvas
  .uri','MyDAO'),(1000,false,10000000000000000,false,100,[],[],[],0,0,0,0,0,true,0
  x0000000000000000000000000000000000000000,false,false,false,'https://example.com
  /ownership',0x0000000000000000000000000000000000000000)
  ,(1000,2000,7000,0,500,3500,6000,0,1000,2000,7000,1000,2000,7000),20)' --private
```

```
   -key *** --rpc-url http://127.0.0.1:8545
```

4.2 IPDProtocol

mintNFT

Interface Description:

```
function mintNFT(CreateCanvasAndMintNFTParam calldata vars) external payable
  returns (uint256);
```

This function is responsible for a Minter to purchase a work, a.k.a, mint a NFT. The function takes a structure as a parameter and returns the token ID of the newly minted NFT (of type 'uint256').

Interface Parameters:

vars

```
      struct CreateCanvasAndMintNFTParam {
        bytes32 daoId;
        bytes32 canvasId;
        string canvasUri; // be empty when not creating a canvas
        address canvasCreator; // be empty when not creating a canvas
        string tokenUri;
        bytes nftSignature;
        uint256 flatPrice;
        bytes32[] proof;
        bytes32[] canvasProof; // be empty when not creating a canvas
        address nftOwner;
        bytes erc20Signature;
        uint256 deadline;
        NftIdentifier nftIdentifier;
      }
```

The structure contains the necessary data for creating a Builder and minting an NFT. The detailed fields are described as follows:

daoId The ID of the SubNodes associated with the Work, represented by 'bytes32'.

canvasId The ID of the Builder associated with the Work, represented by 'bytes32'. If not creating a new Builder, this field should reference an existing Builder ID; otherwise, it should reference a new Builder ID that generated off-chain.

canvasUri The URI for the Builder metadata, represented by 'string'. It has no effect if not creating a new canvas.

canvasCreator The address of the Builder, represented by 'address'. It has no effect if not creating a new canvas. h

tokenUri The URI pointing to the metadata of the NFT being minted, represented by 'string'.

If the string has the special structure: *specialTokenUriPrefix* + *daoIndex* + *'-'* + *tokenIndex* + *".json*", where

- ***specialTokenUriPrefix*** is a predefined prefix ('string' type) returned by the interface *getSpecialTokenUriPrefix()*;
- ***daoIndex*** is the numerical index of the SubNodes ('uint256' type but transferred to 'string') returned by the interface *getDaoIndex(bytes32 daoId)*, with SubNodes ID, as parameter;
- ***tokenIndex*** is a number index ranging from 0 to the total number of Pass Cards (returned by the interface *getDaoReserveNftNumber(bytes32 daoId)*, with SubNodes ID, as parameter),

then it means that a Pass Card is minting. Note that the contract will fetch a suitable token ID according to current number of total minted Pass Cards, so the actual token ID of the newly minted Pass Card may differ to the provided *tokenIndex*.

nftSignature A signature that verifies the price of the NFT, represented by 'bytes'. It is signed by the Builder or the *SIGNER_ROLE*.

flatPrice The Fixed Price for the NFT, represented by 'uint256'. If Unified Fixed Price Mode is activated, the provided value must match the value returned by the *getDaoUnifiedPrice(bytes32 daoId)* function, where the parameter is the ID of the SubNodes; if Unified Fixed Price Mode is not activated, the value that should be provided is included in the *nftSignature* signature. A value of 0 indicates System Pricing is approved by the signer, while a non-zero value represents a Fixed Price approved by the signer.

proof A Merkle proof for verifying the Minter inclusion in the Minter Whitelist, represented by 'bytes32[]'.

canvasProof A Merkle proof for verifying the Builder inclusion in the Builder

Whitelist, represented by ‘bytes32[]’. It has no effect if not creating a new canvas.

nftOwner The address of the new NFT owner, represented by ‘address’.

erc20Signature The signature used for ERC-20 permit (EIP 2612), for ERC20 InputToken payment, represented by ‘bytes’.

deadline The deadline timestamp used for ERC-20 permit, represented by ‘uint256’. If zero value is provided, it means the minter is not using the permit function, and an approval for ERC20 InputToken payment is still required.

nftIdentifier The NFT identifier that indicates a TopUp Account.
If the SubNodes is TopUp Mode, the mint operation is used to top up the TopUp Account. If the *erc721Address* field in the *nftIdentifier* is zero address, the account to top up is the newly minted NFT’s associated TopUp Account (implying a new TopUp Account is created). If it is non-zero, the the account to top up is the TopUp Account indicated by the NFT identifier.
If the SubNodes is not TopUp Mode, the TopUp Account is used for spending. If the *erc721Address* field in the *nftIdentifier* is a zero address or the *nftIdentifier* associated TopUp Account is locked, it means the Minter is paying for the minting from their own wallet. If it is non-zero, the Minter is using funds from the TopUp Account indicated by the *nftIdentifier*. The Minter must be the owner of the NFT corresponding to the *nftIdentifier*. If the funds in the TopUp Account are insufficient to cover the cost of the work, the remaining amount will be deducted from the Minter’s wallet.

Interface Return Value: The return value is of type ‘bytes32’, representing the token ID of the newly minted NFT.

We recommend using the following JavaScript code to generate the EIP-721 signature.

getMerkleProofFoundry.js

```
const { ethers } = require("ethers");
async function generateMintNFTSignature(privateKey, protocolAddress, value) {
  const signer = new ethers.Wallet(privateKey);
  const provider = new ethers.providers.JsonRpcProvider('http://127.0.0.1:7545');
  // All properties on a domain are optional
  const domain = {
    name: "D4AProtocol",
    version: "2",
    chainId: (await provider.getNetwork()).chainId,
    verifyingContract: protocolAddress,
  };

  // The named list of all type definitions
  const types = {
    MintNFT: [
    { name: "canvasID", type: "bytes32" },
    { name: "tokenURIHash", type: "bytes32" },
    { name: "flatPrice", type: "uint256" },
    ],
  };

  const signature = await signer._signTypedData(domain, types, value);
  return signature;
}
```

exchangeOutputToInput

Interface Description:

```
function exchangeOutputToInput(bytes32 daoId, uint256 amount, address to)
  external returns (uint256);
```

This function is responsible for a user to redeem OutputToken to InputToken.

Interface Parameters:

daoId The ID of the SubNodes where this operation is performed, represented by ‘bytes32’.

amount The amount of OutputToken that the user wants to redeem, represented by ‘uint256’. This specifies how much of the OutputToken is being exchanged for the InputToken.

to The address to which the redeemed InputToken will be sent, represented by ‘address’. This is the recipient address where the InputTokenwill be transferred after the exchange.

Interface Return Value:

The return value is of type ‘uint256’, representing the amount of redeemed InputToken.

claimDaoNftOwnerReward

Interface Description:

```
function claimDaoNftOwnerReward(bytes32 daoId) external
  returns (uint256 daoNftOwnerOutputReward, uint256 daoNftOwnerInputReward);
```

This function is responsible for claiming reward of the Starter reward rights. The reward tokens will transferred to the owner of the NFT tied with the Starter reward right. This function can be called by any address.

Interface Parameters:

daoId The ID of the SubNodes where the Starter is the creator, represented by ‘bytes32’.

Interface Return Value:

daoNftOwnerOutputReward The first return value is of type ‘uint256’, representing the claimed OutputToken amount.

daoNftOwnerInputReward The second return value is of type ‘uint256’, representing the claimed InputToken amount.

claimCanvasReward

Interface Description:

```
function claimCanvasReward(bytes32 canvasId) external
  returns (uint256 canvasCreatorOutputReward, uint256 canvasCreatorInputReward);
```

This function is responsible for claiming reward of the Builder. The reward tokens will transferred to the Builder’s address. This function can be called by any address.

Interface Parameters:

canvasId The ID of the Builder, represented by ‘bytes32’.

Interface Return Value:

canvasCreatorOutputReward The first return value is of type 'uint256', representing the claimed OutputToken amount.

canvasCreatorInputReward The second return value is of type 'uint256', representing the claimed InputToken amount.

claimNftMinterReward

Interface Description:

```
  function claimNftMinterReward(bytes32 daoId, address minter) external
returns (uint256 nftMinterOutputReward, uint256 nftMinterInputReward);
```

This function is responsible for claiming reward of the Minter. The reward tokens will transferred to the *minter* parameter. This function can be called by any address.

Interface Parameters:

daoId The ID of the SubNodes where the claim is performed, represented by 'bytes32'.

minter The address of the minter who will receive the reward, represented by 'address'.

Interface Return Value:

nftMinterOutputReward The first return value is of type 'uint256', representing the claimed OutputToken amount.

nftMinterInputReward The second return value is of type 'uint256', representing the claimed InputToken amount.

4.3 IPDRound

getDaoCurrentRound

Interface Description:

```
function getDaoCurrentRound(bytes32 daoId) external view returns (uint256
  currentRound);
```

This function is responsible for retrieving the current Block number of a specific SubNodes.

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

currentRound The return value is of type 'uint256', representing the current Block number of the specified SubNodes.

setDaoDuration

Interface Description:

```
function setDaoDuration(bytes32 daoId, uint256 duration) external;
```

This function is responsible for setting the Block duration for a specific SubNodes.

Interface Parameters:

daoId The ID of the specific SubNodes whose duration is to be set, represented by 'bytes32'.

duration The new duration value to be set for the specific SubNodes, counted by Ethereum block.number time 1e18, represented by 'uint256'.

Interface Return Value:

This function does not return any value.

getDaoLastModifyBlock

Interface Description:

```
function getDaoLastModifyBlock(bytes32 daoId) external view returns (uint256);
```

This function retrieves the last modification time (conunted by Ethereum block.number) at which the specific SubNodes had its block duration modified or was restarted.

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

The return value is of type ‘uint256’, representing the last modification time to the specific SubNodes.

getDaoLastModifyRound

Interface Description:

```
function getDaoLastModifyRound(bytes32 daoId) external view returns (uint256);
```

This function retrieves the last Round number at which the specific SubNodes had its block duration modified or was restarted.

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

The return value is of type ‘uint256’, representing the Block number of the last modification to the specific SubNodes.

4.4 ID4ASettingsReadable

permissionControl

Interface Description:

```
function permissionControl() external view returns (IPermissionControl);
```

This function retrieves the address of the *PermissionControl* independent contract.

Interface Return Value:

The return value is of type ‘IPermissionControl’, representing the address of the *IPermissionControl* contract. (All interface type is equivalent to ‘address’ type).

ownerProxy

Interface Description:

```
function ownerProxy() external view returns (ID4AOwnerProxy);
```

This function retrieves the address of the *ID4AOwnerProxy* independent contract.

Interface Return Value:

> The return value is of type 'ID4AOwnerProxy', representing the address of the *ID4AOwnerProxy* contract.

mintProtocolFeeRatio

Interface Description:

```
function mintProtocolFeeRatio() external view returns (uint256);
```

This function retrieves the MintFee ratio of the ProtocolFeePool, with basis 10000.

Interface Return Value:

> The return value is of type 'uint256', representing the MintFee ratio of the ProtocolFeePool.

tradeProtocolFeeRatio

Interface Description:

```
function tradeProtocolFeeRatio() external view returns (uint256);
```

This function retrieves the Royalty Fee ratio received by the ProtocolFeePool, with basis 10000.

Interface Return Value:

> The return value is of type 'uint256', representing the Royalty Fee ratio received by the ProtocolFeePool.

ratioBase

Interface Description:

```
function ratioBase() external view returns (uint256);
```

This function retrieves the basis of the ratio, which is 10000.

Interface Return Value:

The return value is of type ‘uint256’, representing the basis of the ratio.

getPriceTemplates

Interface Description:

```
function getPriceTemplates() external view returns (address[] memory
  priceTemplates);
```

This function retrieves the addresses of the PriceTemplate contracts.

Interface Return Value:

priceTemplates The return value is of type ‘address[]’, representing the addresses of the PriceTemplate contracts.

getRewardTemplates

Interface Description:

```
function getRewardTemplates() external view returns (address[] memory
  rewardTemplates);
```

This function retrieves the addresses of the RewardTemplate contracts.

Interface Return Value:

rewardTemplates The return value is of type ‘address[]’, representing the addresses of the RewardTemplate contracts.

4.5 IPDGrant

grantDaoAssetPool

Interface Description:

```
function grantDaoAssetPool(bytes32 daoId, uint256 amount, bool useTreasury,
  string calldata tokenUri, address token) external;
```

This function is responsible for granting the NodesAssetPool of a specific SubNodes with a certain amount of ERC20 token. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

amount The amount of OutputToken to be granted to the NodesAssetPool, represented by 'uint256'.

useTreasury A boolean indicating whether the Treasury is used for the grant. If 'true', the grant's funds will be sourced from the treasury, and the granter is required to have the "Treasury Transfer Asset" right; otherwise, they will come from the granter's wallet.

tokenUri The URI of the 'grantAssetPoolNFT', represented by 'string'.

token The address of the ERC20 token to be granted, represented by 'address'.

Interface Return Value:

This function does not return any value.

grantDaoAssetPoolWithPermit

Interface Description:

```
function grantDaoAssetPoolWithPermit(bytes32 daoId, uint256 amount, string
  calldata tokenUri, address token, uint256 deadline, uint8 v, bytes32 r,
  bytes32 s)external;
```

This function is responsible for granting the NodesAssetPool of a specific SubNodes with a certain amount of token, using the ERC20 permit feature eliminates the need for prior approval. The grant's funds will be always sourced from the granter's wallet for this function. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

amount The amount of OutputToken to be granted to the NodesAssetPool, represented by 'uint256'.

tokenUri The URI of the 'grantAssetPoolNFT' minted to the granter, represented by 'string'.

token The address of the ERC20 token to be granted, represented by 'address'.

deadline The deadline timestamp used for ERC-20 permit, represented by 'uint256'.

v The 'v' value of the ERC-20 permit signature, represented by 'uint8'.

r The 'r' value of the ERC-20 permit signature, represented by 'bytes32'.

s The 's' value of the ERC-20 permit signature, represented by 'bytes32'.

Interface Return Value:

This function does not return any value.

grantTreasury

Interface Description:

```
function grantTreasury(bytes32 daoId, uint256 amount, string calldata tokenUri,
  address token) external;
```

This function is responsible for granting the Treasury of a specific SubNodes with a certain amount of ERC20 toknen. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the Treasury associated SeedNodes, represented by 'bytes32'.

amount The amount of ERC20 token to be granted to the Treasury, represented by 'uint256'.

tokenUri The URI of the ‘grantTreasuryNFT’, represented by ‘string’.

token The address of the ERC20 token to be granted, represented by ‘address’.

Interface Return Value:

This function does not return any value.

grantTreasuryWithPermit

Interface Description:

```
function grantTreasuryWithPermit(bytes32 daoId, uint256 amount, string calldata
  tokenUri, address token, uint256 deadline, uint8 v, bytes32 r, bytes32 s)
  external;
```

This function is responsible for granting the Treasury of a specific SubNodes with a certain amount of ERC20 token, using the ERC20 permit feature eliminates the need for prior approval. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the Treasury associated SeedNodes, represented by ‘bytes32’.

amount The amount of ERC20 token to be granted to the Treasury, represented by ‘uint256’.

tokenUri The URI of the ‘grantTreasuryNFT’, represented by ‘string’.

token The address of the ERC20 token to be granted, represented by ‘address’.

deadline The deadline timestamp used for ERC-20 permit, represented by ‘uint256’.

v The ‘v’ value of the ERC-20 permit signature, represented by ‘uint8’.

r The ‘r’ value of the ERC-20 permit signature, represented by ‘bytes32’.

s The ‘s’ value of the ERC-20 permit signature, represented by ‘bytes32’.

Interface Return Value:

This function does not return any value.

4.6 IPDLock

lockTopUpNFT

Interface Description:

```
function lockTopUpNFT(NftIdentifier calldata nft, uint256 duration) external;
```

This function is responsible for locking the TopUp Account associated with the NFT. The function takes the following parameters:

Interface Parameters:

nft The NFT identifier that indicates a TopUp Account, represented by 'NftIdentifier' struct.

duration The duration of the lock, counted by Ethereum block.number, represented by 'uint256'. The lock start at the current block.number.

Interface Return Value:

This function does not return any value.

checkTopUpNftLockedStatus

Interface Description:

```
function checkTopUpNftLockedStatus(NftIdentifier calldata nft) external view
  returns (bool locked);
```

This function is responsible for checking the lock status of the TopUp Account associated with the NFT. The function takes the following parameters:

Interface Parameters:

nft The NFT identifier that indicates a TopUp Account, represented by 'NftIdentifier' struct.

Interface Return Value:

locked The return value is of type 'bool', representing the lock status of the TopUp Account associated with the NFT.

getTopUpNftLockedInfo

Interface Description:

```
function getTopUpNftLockedInfo(NftIdentifier calldata nft) external view returns
    (uint256 lockStartBlock, uint256 duration);
```

This function is responsible for retrieving the lock information of the TopUp Account associated with the NFT. The function takes the following parameters:

Interface Parameters:

nft The NFT identifier that indicates a TopUp Account, represented by 'NftIdentifier' struct.

Interface Return Value:

lockStartBlock The return value is of type 'uint256', representing the block.number at which the lock started.

duration The return value is of type 'uint256', representing the duration of the lock.

Note:

If the TopUp Account has never been locked, the 'lockStartBlock' and 'duration' will be 0.

4.7 IPDPlan

createPlan

Interface Description:

```
function createPlan(CreatePlanParam calldata param) external payable returns (
  bytes32 planId);
```

This function is responsible for creating a Plan. The function takes the following parameters:

Interface Parameters:

param

```
struct CreatePlanParam {
  bytes32 daoId;
  uint256 startBlock;
  uint256 duration;
  uint256 totalRounds;
  uint256 totalReward;
  address rewardToken;
  bool useTreasury;
  bool io;
  string uri;
  PlanTemplateType planTemplateType;
}
```

The structure contains the necessary data for creating a Plan. The detailed fields are described as follows:

daoId Any SubNodes ID in the Plan associated SeedNodes, represented by 'bytes32'.

startBlock The Ethereum block.number at which the Plan starts, represented by 'uint256'. If zero value is provided, the Plan will start at the current block.number.

duration The duration of the Plan, counted by Ethereum block.number, represented by 'uint256'.

totalRounds The total number of Rounds in the Plan, represented by 'uint256'.

totalReward The total reward amount of the Plan, represented by 'uint256'.

rewardToken The address of the ERC20 token used for the reward, represented by 'address'. If zero address is provided, the reward will be in ETH.

useTreasury A boolean indicating whether the Treasury is used for the reward. If 'true', the reward's funds will be sourced from the treasury, and the caller is required to have the ’Treasury Transfer Asset’ right; otherwise, they will come from the creator's wallet.

io A boolean indicating whether the Plan incentizes InputToken or OutputToken. If 'true', the Plan allocates rewards based on the amount of InputToken in the TopUp Account; otherwise, it allocates rewards based on the OutputTokens.

uri The URI of the Plan, represented by 'string'.

planTemplateType The type of the Plan template, represented by 'PlanTemplateType'.

Interface Return Value:

planId The return value is of type 'bytes32', representing the ID of the created Plan.

addPlanTotalReward

Interface Description:

```
function addPlanTotalReward(bytes32 planId, uint256 amount, bool useTreasury)
  external payable;
```

This function is responsible for adding reward to the Plan. Any address can call this function. The function takes the following parameters:

Interface Parameters:

planId The ID of the Plan, represented by 'bytes32'.

amount The amount of reward to be added to the Plan, represented by 'uint256'.

useTreasury A boolean indicating whether the Treasury is used for the reward. If 'true', the reward's funds will be sourced from the treasury, and the caller is required to have the 'Treasury Transfer Asset' right; otherwise, they will come from the creator's wallet.

Interface Return Value:

This function does not return any value.

claimMultiPlanReward

Interface Description:

```
function claimMultiPlanReward(bytes32[] calldata planIds, NftIdentifier calldata
   nft) external returns (uint256);
```

This function is responsible for claiming rewards from multiple Plans for a specific NFT associated TopUp Account. The reward tokens will be transferred to the NFT owner. The function takes the following parameters:

Interface Parameters:

planIds An array of Plan IDs, represented by 'bytes32[]'.

nft The NFT identifier that indicates a TopUp Account, represented by 'NftIdentifier'.

Interface Return Value:

The return value is of type 'uint256', representing the total claimed reward amount.

claimDaoPlanReward

Interface Description:

```
function claimDaoPlanReward(bytes32 daoId, NftIdentifier calldata nft) external
  returns (uint256);
```

This function is responsible for claiming all (not deleted) Plan rewards in a specific SeedNodes for a specific NFT associated TopUp Account. The reward tokens will be transferred to the NFT owner. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by 'bytes32'.

nft The NFT identifier that indicates a TopUp Account, represented by 'NftIdentifier'.

Interface Return Value:

The return value is of type 'uint256', representing the total claimed reward amount.

claimDaoPlanRewardForMultiNft

Interface Description:

```
function claimDaoPlanRewardForMultiNft(bytes32 daoId, NftIdentifier[] calldata
  nfts) external returns (uint256);
```

This function is responsible for claiming all (not deleted) Plan rewards in a specific SeedNodes for multiple NFTs accociated TopUp Accounts. The reward tokens will be transferred to the NFT owners. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by 'bytes32'.

nfts An array of NFT identifiers that indicate the TopUp Accounts to claim the reward, represented by 'NftIdentifier[]'.

Interface Return Value:

The return value is of type 'uint256', representing the total claimed reward amount.

updateTopUpAccount

Interface Description:

```
function updateTopUpAccount(bytes32 daoId,NftIdentifier memory nft) external
  returns (uint256 topUpOutputQuota, uint256 topUpInputQuota);
```

This function is responsible for updating the specific TopUp Account in a specific SeedNodes by posting distributed Mintfee and Block Rewards to the TopUp Account, to be recorded by the Plan's storage state. It can also be called in a read-only manner to retrieve the total (posted plus unposted) balance of the TopUp Account through the return values. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by 'Bytes32'.

nft The NFT identifier that indicates a TopUp Account, represented by 'NftIdentifier'.

Interface Return Value:

topUpOutputQuota The OutputToken balance of the TopUp Account after updating.

topUpInputQuota The InputToken balance of the TopUp Account after updating.

updateMultiTopUpAccount

Interface Description:

```
function updateMultiTopUpAccount(bytes32 daoId, NftIdentifier[] calldata nfts)
  external;
```

This function is responsible for updating multiple TopUp Accounts in a specific SeedNodes by posting distributed Mintfee and Block Rewards to each TopUp Account, to be recorded by the Plan's storage state. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by 'Bytes32'.

nfts An array of NFT identifiers, each indicating a TopUp Account, represented by 'NftIdentifier[]'.

Interface Return Value:

This function does not return any values.

getTopUpBalance

Interface Description:

```
function getTopUpBalance(bytes32 daoId, NftIdentifier memory nft) external view
  returns (uint256,uint256);
```

This function is responsible for retrieving the (posted) balance of the TopUp Account in a specific SeedNodes. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by 'Bytes32'.

nft The NFT identifier that indicates a TopUp Account, represented by 'NftIdentifier'.

Interface Return Value:

The return values are of type 'uint256', representing the OutputToken and InputToken balance of the TopUp Account.

getPlanCumulatedReward

Interface Description:

```
function getPlanCumulatedReward(bytes32 planId) external returns (uint256);
```

This function is responsible for retrieving the total amount of all distributed rewards of the Plan, by calling in a read-only manner. It is a write function and it will also update the related stroage state of the plan. The function takes the following parameters:

Interface Parameters:

planId	The ID of the Plan, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the distributed reward amount of the Plan.

retriveUnclaimedToken

Interface Description:

```
function retriveUnclaimedToken(bytes32 planId) external;
```

This function is responsible for retrieving the left reward tokens of the Plan for its creator. Only the creator of the plan can call this function, and it can only be invoked after the plan has ended. The returned rewards do not include any unclaimed rewards by users. The rewards primarily come from cases where, at the end of a Block, no non-zero TopUp Account exists; in such cases, the plan does not distribute rewards for that Block. The function takes the following parameters:

Interface Parameters:

planId	The ID of the Plan, represented by 'bytes32'.

Interface Return Value:

This function does not return any value.

getPlanCurrentRound

Interface Description:

```
function getPlanCurrentRound(bytes32 planId) external view returns (uint256);
```

This function is responsible for retrieving the current Block number of the Plan. The function takes the following parameters:

Interface Parameters:

planId The ID of the Plan, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the current Block number of the Plan.

4.8 ID4AProtocolReadable

This interface includes legacy methods as well as newly added methods in old versions. All the methods are view functions.

Legacy Methods

getProjectInfo

Interface Description:

```
function getProjectInfo(bytes32 daoId) external view returns (uint256 startRound
  , uint256 mintableRound, uint256 maxNftAmount, address daoFeePool, uint96
  royaltyFeeRatioInBps, uint256 index,string memory daoUri, uint256
  outputTotalSupply);
```

This function is responsible for retrieving the information of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

startRound The return value is of type ‘uint256’, representing the start time of the specific SubNodes, counted by Ethereum block.number.

mintableRound The return value is of type ‘uint256’, representing the total number of Blocks in the specific SubNodes’ duation. It equals the number of passed Blocks plus the number of remaining Blocks.

maxNftAmount The return value is of type ‘uint256’, representing the total amount of NFTs that can be minted in the specific SubNodes.

daoFeePool The return value is of type ‘address’, representing the NodesAsset-Pool address of the specific SubNodes.

royaltyFeeRatioInBps The return value is of type ‘uint96’, representing the RoyaltyFee ratio of the specific SubNodes.

index The return value is of type ‘uint256’, representing the index of the specific SubNodes.

daoUri The return value is of type ‘string’, representing the URI of the specific SubNodes.

outputTotalSupply The return value is of type ‘uint256’, representing the total number of granted OutputToken of the specific SubNodes. In current version, this value is no longer in use.

getProjectCanvasAt

Interface Description:

```
function getProjectCanvasAt(bytes32 daoId, uint256 index) external view returns
  (bytes32);
```

This function is responsible for retrieving the Builder ID given the associated by SubNodes ID and index. The function takes the following parameters:

Interface Parameters:

daoId The ID of the SubNodes, represented by ‘bytes32’.

index The index of the Builder, represented by ‘uint256’, which means that the ID of the index-th created builder will be retrieved

Interface Return Value:

The return value is of type 'bytes32', representing the Builder ID.

getProjectTokens

Interface Description:

```
function getProjectTokens(bytes32 daoId) external view returns (address token,
  address nft);
```

This function is responsible for retrieving the OutputToken and Main ERC721 addresses of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

token The return value is of type 'address', representing the address of the OutputToken of the specific SubNodes.

nft The return value is of type 'address', representing the address of the Main ERC721 of the specific SubNodes.

getCanvasNFTCount

Interface Description:

```
function getCanvasNFTCount(bytes32 canvasId) external view returns (uint256);
```

This function is responsible for retrieving the total number of NFTs minted on the Builder. The function takes the following parameters:

Interface Parameters:

canvasId The ID of the Builder, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the total number of NFTs minted on the Builder.

getTokenIDAt

Interface Description:

```
function getTokenIDAt(bytes32 canvasId, uint256 index) external view returns (
  uint256);
```

This function is responsible for retrieving the NFT ID of the Builder at the specified index. The function takes the following parameters:

Interface Parameters:

canvasId The ID of the Builder, represented by ‘bytes32’.

index The index of the NFT, represented by ‘uint256’, which means that the *tokenID* of the index-th minted NFT on the Builder will be retrieved.

Interface Return Value:

The return value is of type ‘uint256’, representing the NFT’s *tokenId*.

getCanvasProject

Interface Description:

```
function getCanvasProject(bytes32 canvasId) external view returns (bytes32);
```

This function is responsible for retrieving the SubNodes ID associated with the Builder. The function takes the following parameters:

Interface Parameters:

canvasId The ID of the Builder, represented by ‘bytes32’.

Interface Return Value:

The return value is of type ‘bytes32’, representing the SubNodes ID associated with the Builder.

getCanvasURI

Interface Description:

```
function getCanvasURI(bytes32 canvasId) external view returns (string memory);
```

This function is responsible for retrieving the URI of the Builder. The function takes the following parameters:

Interface Parameters:

canvasId The ID of the Builder, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'string', representing the URI of the Builder.

getProjectCanvasCount

Interface Description:

```
function getProjectCanvasCount(bytes32 daoId) external view returns (uint256);
```

This function is responsible for retrieving the total number of Builders created in the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the total number of Builders created in the specific SubNodes.

Newly Added Methods in Old Versions

getDaoStartBlock

Interface Description:

```
function getDaoStartBlock(bytes32 daoId) external view returns (uint256
  startRound);
```

This function is responsible for retrieving the start time of the specific SubNodes, conunted by Ethereum block.number. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

startRound The return value is of type 'uint256', representing the start time of the specific SubNodes.

getDaoMintableRound

Interface Description:

```
function getDaoMintableRound(bytes32 daoId) external view returns (uint256
  mintableRound);
```

This function is responsible for retrieving the total number of Blocks in the specific SubNodes' duration. It equals the number of passed Blocks plus the number of remaining Blocks. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

mintableRound The return value is of type 'uint256', representing the total number of Blocks of the specific SubNodes duration.

getDaoIndex

Interface Description:

```
function getDaoIndex(bytes32 daoId) external view returns (uint256 index);
```

This function is responsible for retrieving the index of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

index The return value is of type 'uint256', representing the index of the specific SubNodes.

getDaoUri

Interface Description:

```
function getDaoUri(bytes32 daoId) external view returns (string memory daoUri);
```

This function is responsible for retrieving the URI of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

daoUri The return value is of type 'string', representing the URI of the specific SubNodes.

getDaoFeePool

Interface Description:

```
function getDaoFeePool(bytes32 daoId) external view returns (address daoFeePool)
  ;
```

This function is responsible for retrieving the NodesAssetPool address of the specific SubNodes. It is the same as the *getDaoAssetPool* interface. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

daoFeePool The return value is of type 'address', representing the NodesAsset-Pool address of the specific SubNodes.

getDaoToken

Interface Description:

```
function getDaoToken(bytes32 daoId) external view returns (address token);
```

This function is responsible for retrieving the OutputToken address of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

token The return value is of type 'address', representing the OutputToken address of the specific SubNodes.

getDaoNft

Interface Description:

```
function getDaoNft(bytes32 daoId) external view returns (address nft);
```

This function is responsible for retrieving the main ERC721 address of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

nft The return value is of type 'address', representing the main ERC721 address of the specific SubNodes.

getDaoNftMaxSupply

Interface Description:

```
function getDaoNftMaxSupply(bytes32 daoId) external view returns (uint256
  nftMaxSupply);
```

This function is responsible for retrieving the total amount of NFTs that can be minted in the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

nftMaxSupply The return value is of type 'uint256', representing the total amount of NFTs that can be minted in the specific SubNodes.

getDaoNftTotalSupply

Interface Description:

```
function getDaoNftTotalSupply(bytes32 daoId) external view returns (uint256
  nftTotalSupply);
```

This function is responsible for retrieving the total amount of NFTs that have been minted in the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

nftTotalSupply The return value is of type ‘uint256’, representing the total amount of NFTs that have been minted in the specific SubNodes.

getDaoRoyaltyFeeRatioInBps

Interface Description:

```
function getDaoRoyaltyFeeRatioInBps(bytes32 daoId) external view returns (uint96
    royaltyFeeRatioInBps);
```

This function is responsible for retrieving the RoyaltyFee ratio of the Main ERC721 of the specific SubNodes, represented by the number of basis points (BPS=10000). The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

royaltyFeeRatioInBps The return value is of type ‘uint96’, representing the RoyaltyFee ratio of the Main ERC721 of the specific SubNodes.

getDaoExist

Interface Description:

```
function getDaoExist(bytes32 daoId) external view returns (bool);
```

This function is responsible for checking whether the specific SubNodes exists. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

The return value is of type 'bool', representing whether the specific SubNodes exists.

getDaoCanvases

Interface Description:

```
function getDaoCanvases(bytes32 daoId) external view returns (bytes32[] memory
  canvases);
```

This function is responsible for retrieving the IDs of all Builders created in the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

canvases The return value is of type 'bytes32[]', representing the IDs of all Builders created in the specific SubNodes.

getDaoPriceTemplate

Interface Description:

```
function getDaoPriceTemplate(bytes32 daoId) external view returns (address
  priceTemplate);
```

This function is responsible for retrieving the address of the PriceTemplate contract of the specific SubNodes. The PriceTemplate contract is used to calculate the System Price of the NFTs minted in the SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

priceTemplate The return value is of type ‘address’, representing the address of the PriceTemplate contract of the specific SubNodes.

getDaoPriceFactor

Interface Description:

```
function getDaoPriceFactor(bytes32 daoId) external view returns (uint256
  priceFactor);
```

This function is responsible for retrieving the PriceFactor of the specific SubNodes. The PriceFactor is used as a parameter to calculate the System Price of the NFTs minted in the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

priceFactor The return value is of type ‘uint256’, representing the PriceFactor of the specific SubNodes.

getDaoRewardTemplate

Interface Description:

```
function getDaoRewardTemplate(bytes32 daoId) external view returns (address
  rewardTemplate);
```

This function is responsible for retrieving the address of the RewardTemplate contract of the specific SubNodes. The RewardTemplate contract is used to calculate and update the Block Reward issuance of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

rewardTemplate The return value is of type 'address', representing the address of the RewardTemplate contract of the specific SubNodes.

getDaoMintCap

Interface Description:

```
function getDaoMintCap(bytes32 daoId) external view returns (uint32);
```

This function is responsible for retrieving the maximum number of NFTs that can be minted in the specific SubNodes for each (Minter) address. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint32', representing the maximum number of NFTs that can be minted in the specific SubNodes for each (Minter) address.

getUserMintInfo

Interface Description:

```
function getUserMintInfo(bytes32 daoId, address account) external view returns (
  uint32 minted, uint32 userMintCap);
```

This function is responsible for retrieving the number of NFTs minted by the Minter in the specific SubNodes, as well as the maximum number of NFTs that can be minted by the Minter in the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

account The address of the Minter, represented by 'address'.

Interface Return Value:

minted The return value is of type 'uint32', representing the number of NFTs minted by the Minter in the specific SubNodes.

userMintCap The return value is of type 'uint32', representing the maximum number of NFTs that can be minted by the Minter in the specific SubNodes.

getDaoTag

Interface Description:

```
function getDaoTag(bytes32 daoId) external view returns (string memory);
```

This function is responsible for retrieving the tag of the specific SubNodes. The tag is used to differentiate between versions of the specific SubNodes. The tag for all SubNodes created under the current version is set to 'BASIC_DAO'. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'string', representing the tag of the specific SubNodes.

getCanvasDaoId

Interface Description:

```
function getCanvasDaoId(bytes32 canvasId) external view returns (bytes32 daoId);
```

This function is responsible for retrieving the SubNodes ID associated with the Builder. The function takes the following parameters:

Interface Parameters:

canvasId The ID of the Builder, represented by 'bytes32'.

Interface Return Value:

daoId The return value is of type 'bytes32', representing the SubNodes ID associated with the Builder.

getCanvasTokenIds

Interface Description:

```
function getCanvasTokenIds(bytes32 canvasId) external view returns (uint256[]
  memory tokenIds);
```

This function is responsible for retrieving the token IDs of all NFTs minted on the Builder. The function takes the following parameters:

Interface Parameters:

canvasId The ID of the Builder, represented by 'bytes32'.

Interface Return Value:

tokenIds The return value is of type 'uint256[]', representing the token IDs of all NFTs minted on the Builder.

getCanvasIndex

Interface Description:

```
function getCanvasIndex(bytes32 canvasId) external view returns (uint256);
```

This function is responsible for retrieving the index of the Builder. The function takes the following parameters:

Interface Parameters:

canvasId The ID of the Builder, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the index of the Builder.

getCanvasIndex

Interface Description:

```
function getCanvasIndex(bytes32 canvasId) external view returns (uint256);
```

This function is responsible for retrieving the index of the Builder, which means that it is the index-th Builder created in its associate SubNodes. The function takes the following parameters:

Interface Parameters:

canvasId The ID of the Builder, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the index of the Builder.

getCanvasUri

Interface Description:

```
function getCanvasUri(bytes32 canvasId) external view returns (string memory
  canvasUri);
```

This function is responsible for retrieving the URI of the Builder. The function takes the following parameters:

Interface Parameters:

canvasId The ID of the Builder, represented by 'bytes32'.

Interface Return Value:

canvasUri The return value is of type 'string', representing the URI of the Builder.

getCanvasExist

Interface Description:

```
function getCanvasExist(bytes32 canvasId) external view returns (bool);
```

This function is responsible for checking whether the Builder exists. The function takes the following parameters:

Interface Parameters:

canvasId The ID of the Builder, represented by ‘bytes32’.

Interface Return Value:

The return value is of type ‘bool’, representing whether the Builder exists.

getCanvasLastPrice

Interface Description:

```
function getCanvasLastPrice(bytes32 canvasId) external view returns (uint256
  round, uint256 price);
```

This function is responsible for retrieving the information of last minted **System Pricing** NFT on the Builder. The function takes the following parameters:

Interface Parameters:

canvasId The ID of the Builder, represented by ‘bytes32’.

Interface Return Value:

round The return value is of type ‘uint256’, representing the Block number of the last minting.

price The return value is of type ‘uint256’, representing the price of the last minting.

getCanvasNextPrice

Interface Description:

```
function getCanvasNextPrice(bytes32 canvasId) external view returns (uint256
  price);
```

This function is responsible for retrieving the price of the next System Pricing NFT on an existed Builder. The function takes the following parameters:

Interface Parameters:

canvasId The ID of the Builder, represented by ‘bytes32’.

Interface Return Value:

price The return value is of type ‘uint256’, representing the price of the next System Pricing NFT on the Builder.

getDaoMaxPriceInfo

Interface Description:

```
function getDaoMaxPriceInfo(bytes32 daoId) external view returns (uint256 round,
    uint256 price);
```

This function returns information about the highest-priced NFT minted under the entire specific SubNodes. Note that the highest price takes into account the period decay. Specifically, the Builder with the highest system price for the current period is first determined, and then the information of the most recently minted NFT from that Builder is retrieved. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

round The return value is of type ‘uint256’, representing the Block number at which the highest-priced NFT was minted.

price The return value is of type ‘uint256’, representing the price of the highest-priced NFT.

getDaoFloorPrice

Interface Description:

```
function getDaoFloorPrice(bytes32 daoId) external view returns (uint256
  floorPrice);
```

This function is responsible for retrieving the Floor Price of the specific SubNodes. It is the same as the *getProjectFloorPrice* interface. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

floorPrice The return value is of type 'uint256', representing the Floor Price of the specific SubNodes.

getDaoRoundMintCap

Interface Description:

```
function getDaoRoundMintCap(bytes32 daoId) external view returns (uint256);
```

This function is responsible for retrieving the maximum number of NFTs that can be minted in the specific SubNodes for each Block. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the maximum number of NFTs that can be minted in the specific SubNodes for each Block.

getDaoUnifiedPriceModeOff

Interface Description:

```
function getDaoUnifiedPriceModeOff(bytes32 daoId) external view returns (bool);
```

This function is responsible for checking whether the specific SubNodes is in the Unified Fixed Price Mode. The Unified Price Mode is a mode in which the price is the same for all NFTs in the SubNodes. If the SubNodes is in the Unified Fixed Price Mode, the function returns 'false'; otherwise, it returns 'true'. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'bool', representing whether the specific SubNodes is in the Unified Price Mode.

getDaoUnifiedPrice

Interface Description:

```
function getDaoUnifiedPrice(bytes32 daoId) external view returns (uint256);
```

This function is responsible for retrieving the Unified Price of the specific SubNodes. The Unified Price is the price of all NFTs in the SubNodes when the SubNodes is in the Unified Fixed Price Mode. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the Unified Price of the specific SubNodes.

getDaoReserveNftNumber

Interface Description:

```
function getDaoReserveNftNumber(bytes32 daoId) external view returns (uint256);
```

This function is responsible for retrieving the number of Pass Cards in the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type ‘uint256’, representing the number of Pass Cards in the specific SubNodes.

4.9 IPDProtocolReadable

This interface inherits from *ID4AProtocolReadable* and includes methods from new versions. This section will introduce all methods that do not belong to ID4AProtocolReadable.

getDaoId

Interface Description:

```
function getDaoId(uint8 daoTag, uint256 daoIndex) external view returns (bytes32
  );
```

This function is responsible for retrieving the specific SubNodes ID by the DAO tag and index. The function takes the following parameters:

Interface Parameters:

daoTag The tag of the specific SubNodes, represented by ‘uint8’. It is used to differentiate between versions of SubNodes. The tag for all SubNodes created under the current version is set to ‘BASIC_DAO’, with a corresponding uint8 value of 1.

daoIndex The index of the specific SubNodes, represented by ‘uint256’.

Interface Return Value:

The return value is of type ‘bytes32’, representing the specific SubNodes ID.

getNFTTokenCanvas

Interface Description:

```
function getNFTTokenCanvas(bytes32 daoId, uint256 tokenId) external view returns
    (bytes32);
```

This interface, given the specified SubNodes and tokenID, retrieves the Builder ID that the specific SubNodes' NFT with the specified tokenID associated. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

tokenId The token ID of the NFT, represented by 'uint256'.

Interface Return Value:

The return value is of type 'bytes32', representing the Builder ID that the NFT associated.

getLastestDaoIndex

Interface Description:

```
function getLastestDaoIndex(uint8 daoTag) external view returns (uint256);
```

This function is responsible for retrieving the index of the last created SubNodes by the DAO tag. The function takes the following parameters:

Interface Parameters:

daoTag The tag of the specific SubNodes, represented by 'uint8'. It is used to differentiate between versions of SubNodes. The tag for all SubNodes created under the current version is set to 'BASIC_DAO', with a corresponding uint8 value of 1.

Interface Return Value:

The return value is of type 'uint256', representing the latest index of the SubNodes.

getDaoAncestor

Interface Description:

```
function getDaoAncestor(bytes32 daoId) external view returns (bytes32);
```

This function is responsible for retrieving the ID of the SeedNodes (the ID of the SubNodes that was created together with the SeedNodes) to which the specific SubNodes belongs, also called the ancestor of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'bytes32', representing the ID of the ancestor of the specific SubNodes.

getDaoVersion

Interface Description:

```
function getDaoVersion(bytes32 daoId) external view returns (uint8);
```

This function is responsible for retrieving the version of the specific SubNodes. The version is used to differentiate between different versions of SubNodes'. The version for all SubNodes created under the current version is set to 16. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint8', representing the version of the specific SubNodes.

getCanvasCreatorMintFeeRatio

Interface Description:

```
function getCanvasCreatorMintFeeRatio(bytes32 daoId) external returns (uint256);
```

This function is responsible for retrieving the MintFee ratio of the Builder, which is the percentage of the MintFee that the Builder receives when a System Pricing NFT is minted on the Builder. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the MintFee ratio of the Builder.

getAssetPoolMintFeeRatio

Interface Description:

```
function getAssetPoolMintFeeRatio(bytes32 daoId) external returns (uint256);
```

This function is responsible for retrieving the MintFee ratio of the NodesAssetPool, which is the percentage of the Mintfee that the NodesAssetPool receives when a System Pricing NFT is minted on the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the MintFee ratio of the NodesAssetPool.

getRedeemPoolMintFeeRatio

Interface Description:

```
function getRedeemPoolMintFeeRatio(bytes32 daoId) external returns (uint256);
```

This function is responsible for retrieving the MintFee ratio of the NodesRedeemPool, which is the percentage of the Mintfee that the NodesRedeemPool receives when a System Pricing NFT is minted on the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

The return value is of type ‘uint256’, representing the MintFee ratio of the NodesRedeemPool.

getTreasuryMintFeeRatio

Interface Description:

```
function getTreasuryMintFeeRatio(bytes32 daoId) external returns (uint256);
```

This function is responsible for retrieving the MintFee ratio of the Treasury, which is the percentage of the Mintfee that the Treasury receives when a System Pricing NFT is minted on the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

The return value is of type ‘uint256’, representing the MintFee ratio of the Treasury.

getCanvasCreatorMintFeeRatioFiatPrice

Interface Description:

```
function getCanvasCreatorMintFeeRatioFiatPrice(bytes32 daoId) external returns (
  uint256);
```

This function is responsible for retrieving the MintFee ratio for Fixed Price of the Builder, which is the percentage of the Mintfee that the Builder receives when a non-System Pricing NFT is minted on the Builder. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the MintFee ratio for Fixed Price of the Builder.

getAssetPoolMintFeeRatioFiatPrice

Interface Description:

```
function getAssetPoolMintFeeRatioFiatPrice(bytes32 daoId) external returns (
  uint256);
```

This function is responsible for retrieving the MintFee ratio for Fixed Price of the NodesAssetPool, which is the percentage of the Mintfee that the NodesAssetPool receives when a non-System Pricing NFT is minted on the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the MintFee ratio for Fixed Price of the NodesAssetPool .

getRedeemPoolMintFeeRatioFiatPrice

Interface Description:

```
function getRedeemPoolMintFeeRatioFiatPrice(bytes32 daoId) external returns (
  uint256);
```

This function is responsible for retrieving the MintFee ratio for Fixed Price of the NodesRedeemPool, which is the percentage of the Mintfee that the NodesRedeem-Pool receives when a non-System Pricing NFT is minted on the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the MintFee ratio for Fixed Price of the NodesRedeemPool.

getTreasuryMintFeeRatioFiatPrice

Interface Description:

```
function getTreasuryMintFeeRatioFiatPrice(bytes32 daoId) external returns (
  uint256);
```

This function is responsible for retrieving the MintFee ratio for Fixed Price of the Treasury, which is the percentage of the Mintfee that the Treasury receives when a non-System Pricing NFT is minted on the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the MintFee ratio for Fixed Price of the Treasury.

getMinterOutputRewardRatio

Interface Description:

```
function getMinterOutputRewardRatio(bytes32 daoId) external view returns (
  uint256);
```

This function is responsible for retrieving the role coefficient of Minter for OutputToken Internal Distribution of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the role coefficient of Minter for OutputToken Internal Distribution.

getCanvasCreatorOutputRewardRatio

Interface Description:

```
function getCanvasCreatorOutputRewardRatio(bytes32 daoId) external view returns
  (uint256);
```

This function is responsible for retrieving the role coefficient of Builder for OutputToken Internal Distribution of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the role coefficient of Builder for OutputToken Internal Distribution.

getDaoCreatorOutputRewardRatio

Interface Description:

```
function getDaoCreatorOutputRewardRatio(bytes32 daoId) external view returns (
  uint256);
```

This function is responsible for retrieving the role coefficient of Starter for OutputToken Internal Distribution of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the role coefficient of Starter for OutputToken Internal Distribution.

getMinterInputRewardRatio

Interface Description:

```
function getMinterInputRewardRatio(bytes32 daoId) external view returns (uint256
  );
```

This function is responsible for retrieving the role coefficient of Minter for InputToken Internal Distribution of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the role coefficient of Minter for InputToken Internal Distribution.

getCanvasCreatorInputRewardRatio

Interface Description:

```
function getCanvasCreatorInputRewardRatio(bytes32 daoId) external view returns (
  uint256);
```

This function is responsible for retrieving the role coefficient of Builder for InputToken Internal Distribution of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

The return value is of type ‘uint256’, representing the role coefficient of Builder for InputToken Internal Distribution.

getDaoCreatorInputRewardRatio

Interface Description:

```
function getDaoCreatorInputRewardRatio(bytes32 daoId) external view returns (
  uint256);
```

This function is responsible for retrieving the role coefficient of Starter for InputToken Internal Distribution of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

The return value is of type ‘uint256’, representing the role coefficient of Starter for InputToken Internal Distribution.

getIsAncestorDao

Interface Description:

```
function getIsAncestorDao(bytes32 daoId) external view returns (bool);
```

This function is responsible for checking whether the specific SubNodes ID is also the ID of a SeedNodes, i.e., it is the first created SubNodes in the SeedNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'bool', representing whether the specific SubNodes ID is also the ID of a SeedNodes.

getDaoLastActiveRound

Interface Description:

```
function getDaoLastActiveRound(bytes32 daoId) external view returns (uint256);
```

This function is responsible for retrieving the last Active Block number of the specific SubNodes. The last active Block is the Block at which the last NFT was minted in the SubNodes (including the current Block). The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the last active round of the specific SubNodes.

getDaoPassedRound

Interface Description:

```
function getDaoPassedRound(bytes32 daoId) external view returns (uint256);
```

This function is responsible for retrieving the number of Blocks that have passed since the start of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the number of Blocks that have passed.

getDaoRemainingRound

Interface Description:

```
function getDaoRemainingRound(bytes32 daoId) external view returns (uint256);
```

This function is responsible for retrieving the number of remaining Blocks before the end of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the number of remaining Blocks.

getDaoChildren

Interface Description:

```
function getDaoChildren(bytes32 daoId) external view returns (bytes32[] memory);
```

This function is responsible for retrieving the IDs of the children SubNodes (recipient SubNodesfor Block Reward issuance) of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is an array of type 'bytes32', representing the IDs of the children SubNodes.

getDaoChildrenOutputRatios

Interface Description:

```
function getDaoChildrenOutputRatios(bytes32 daoId) external view returns (
  uint256[] memory);
```

This function is responsible for retrieving the OutputToken ratios of children SubNodes for Block Reward Issuance of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

An array of type 'uint256', representing the OutputToken ratios of children SubNodes.

getDaoChildrenInputRatios

Interface Description:

```
function getDaoChildrenInputRatios(bytes32 daoId) external view returns (uint256
  [] memory);
```

This function is responsible for retrieving the InputToken ratios of children SubNodes for Block Reward Issuance of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

An array of type ‘uint256’, representing the InputToken ratios of children SubNodes.

getDaoRedeemPoolInputRatio

Interface Description:

```
function getDaoRedeemPoolInputRatio(bytes32 daoId) external view returns (
  uint256);
```

This function is responsible for retrieving the InputToken ratio of the NodesRedeemPool for Block Reward Issuance of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

An array of type ‘uint256’, representing the InputToken ratio of the NodesRedeemPool.

getDaoTreasuryOutputRatio

Interface Description:

```
function getDaoTreasuryOutputRatio(bytes32 daoId) external view returns (uint256
  );
```

This function is responsible for retrieving the OutputToken ratio of the Treasury for Block Reward Issuance of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

An array of type ‘uint256’, representing the OutputToken ratio of the Treasury.

getDaoTreasuryInputRatio

Interface Description:

```
function getDaoTreasuryOutputRatio(bytes32 daoId) external view returns (uint256
  );
```

This function is responsible for retrieving the InputToken ratio of the Treasury for Block Reward Issuance of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

An array of type ‘uint256’, representing the InputToken ratio of the Treasury.

getDaoSelfRewardOutputRatio

Interface Description:

```
function getDaoSelfRewardOutputRatio(bytes32 daoId) external view returns (
  uint256);
```

This function is responsible for retrieving the OutputToken ratio of Internal Distribution for Block Reward Issuance of the specific SubNodes . The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

An array of type ‘uint256’, representing the OutputToken ratio of Internal Distribution.

getDaoSelfRewardInputRatio

Interface Description:

```
function getDaoSelfRewardInputRatio(bytes32 daoId) external view returns (
  uint256);
```

This function is responsible for retrieving the InputToken ratio of Internal Distribution for Block Reward Issuance of the specific SubNodes . The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

An array of type 'uint256', representing the InputToken ratio of Internal Distribution.

getDaoTopUpMode

Interface Description:

```
function getDaoTopUpMode(bytes32 daoId) external view returns (bool);
```

This function is responsible for checking whether the specific SubNodes is in the TopUp Mode. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

A boolean value representing whether the specific SubNodes is in the TopUp Mode.

getDaoIsThirdPartyToken

Interface Description:

```
function getDaoIsThirdPartyToken(bytes32 daoId) external view returns (bool);
```

This function is responsible for checking whether the specific SubNodes is in the Import OutputToken Mode. The function takes the following parameters:

Interface Parameters:

daoId	The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

A boolean value representing whether the specific SubNodes is in the Import OutputToken Mode.

getRoundOutputReward

Interface Description:

```
function getRoundOutputReward(bytes32 daoId, uint256 round) external view
  returns (uint256);
```

This function is responsible for retrieving the OutputToken amount of Internal Distribution for the specific SubNodes and Block. For Inactive Blocks this value is zero. The function takes the following parameters:

Interface Parameters:

daoId	The ID of the specific SubNodes, represented by 'bytes32'.
round	The specific Block number, represented by 'uint256'.

Interface Return Value:

An array of type 'uint256', representing the OutputToken amount of Internal Distribution for the specific SubNodes and Block.

getRoundInputReward

Interface Description:

```
function getRoundInputReward(bytes32 daoId, uint256 round) external view returns
    (uint256);
```

This function is responsible for retrieving the InputToken amount of Internal Distribution for the specific SubNodes and Block. For Inactive Blocks this value is zero. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

round The specific Block number, represented by 'uint256'.

Interface Return Value:

An array of type 'uint256', representing the InputToken amount of Internal Distribution for the specific SubNodes and Block.

getOutputRewardTillRound

Interface Description:

```
function getOutputRewardTillRound(bytes32 daoId, uint256 round) external view
  returns (uint256);
```

This function is responsible for retrieving the total OutputToken amount of Internal Distribution from the start till the specific Block for the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

round The specific Block number, represented by 'uint256'.

Interface Return Value:

An array of type 'uint256', representing the total OutputToken amount of Internal Distribution till the specific Block number for the specific SubNodes.

getInputRewardTillRound

Interface Description:

```
function getInputRewardTillRound(bytes32 daoId, uint256 round) external view
  returns (uint256);
```

This function is responsible for retrieving the total InputToken amount of Internal Distribution from the start till the specific Block for the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

round The specific Block number, represented by 'uint256'.

Interface Return Value:

An array of type 'uint256', representing the total InputToken amount of Internal Distribution till the specific Block number for the specific SubNodes.

royaltySplitters

Interface Description:

```
function royaltySplitters(bytes32 daoId) external view returns (address);
```

This function is responsible for retrieving the address of the Royalty Splitter contract for the specific SubNodes. The Royalty Splitter contract is responsible for distributing the RoyaltyFee to the respective recipients. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'address', representing the address of the Royalty Splitter contract.

getCanvasNextPrice

Interface Description:

```
function getCanvasNextPrice(bytes32 daoId, bytes32 canvasId) external view
  returns (uint256);
```

This function is similar to the *getCanvasNextPrice(bytes32 canvasId)* interface, but it applies to Builders that are not created. That is, it can obtain the price of the first Work of the next newly created Builder under the specified SubNodes.

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

canvasId The ID of the specific NFT, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the next System Price of the specific SubNodes and builder.

getDaoCirculateTokenAmount

Interface Description:

```
function getDaoCirculateTokenAmount(bytes32 daoId) external view returns (
  uint256);
```

This function is responsible for retrieving the total amount of circulate OutputToken , associated with the sepecific SubNodes (See **Claim Reward** section for detail). The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the total amount of circulate OutputToken.

getDaoInfiniteMode

Interface Description:

```
function getDaoInfiniteMode(bytes32 daoId) external view returns (bool);
```

This function is responsible for checking whether the specific SubNodes is in the Infinite Mode. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

A boolean value representing whether the specific SubNodes is in the Infinite Mode.

getDaoOutputPaymentMode

Interface Description:

```
function getDaoOutputPaymentMode(bytes32 daoId) external view returns (bool);
```

This function is responsible for checking whether the specific SubNodes is in the OutputToken Payment Mode. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

A boolean value representing whether the specific SubNodes is in the OutputToken Payment Mode.

getDaoRoundDistributeAmount

Interface Description:

```
function getDaoRoundDistributeAmount(bytes32 daoId, address token, uint256
  currentRound, uint256 remainingRound) external view returns (uint256);
```

This function is responsible for retrieving the amount of the specified token for Block Reward that will be issued if someone mints an NFT under the specified SubNodes. It is essential that the current Block is inactive and the parameters are correctly input to obtain the accurate value. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

token The address of the token, represented by 'address'. If zero address is provided, then ETH is considered.

currentRound The current Block number, represented by 'uint256'.

remainingRound The remaining Block number, represented by 'uint256'.

Interface Return Value:

The return value is of type 'uint256', representing the amount of OutputToken to be distributed.

getDaoTopUpInputToRedeemPoolRatio

Interface Description:

```
function getDaoTopUpInputToRedeemPoolRatio(bytes32 daoId) external view returns
  (uint256);
```

This function is responsible for retrieving the ratio that unlocked InputToken from TopUp Accounts transferred to the NodesRedeemPool of the sepecific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the InputToken ratio for TopUp Account Unlock Distribution.

getDaoTopUpOutputToTreasuryRatio

Interface Description:

```
function getDaoTopUpOutputToTreasuryRatio(bytes32 daoId) external view returns (
  uint256);
```

This function is responsible for retrieving the ratio that unlocked OutputToken from TopUp Accounts transferred to the Treasury of the sepecific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the OutputToken ratio for TopUp Account Unlock Distribution.

getDaoDefaultTopUpInputToRedeemPoolRatio

Interface Description:

```
function getDaoDefaultTopUpInputToRedeemPoolRatio(bytes32 daoId) external view
  returns (uint256);
```

This function is responsible for retrieving the default InputToken ratio for TopUp Account Unlock Distribution for a specific SeedNodes, that any newly created SubNodes in the SeedNodes will follow that ratio. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the default InputToken ratio for TopUp Account Unlock Distribution.

getDaoDefaultTopUpOutputToTreasuryRatio

Interface Description:

```
function getDaoDefaultTopUpOutputToTreasuryRatio(bytes32 daoId) external view
  returns (uint256);
```

This function is responsible for retrieving the default OutputToken ratio for TopUp Account Unlock Distribution for a specific SeedNodes, that any newly created SubNodes in the SeedNodes will follow that ratio. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'uint256', representing the default OutputToken ratio for TopUp Account Unlock Distribution.

getDaoGrantAssetPoolNft

Interface Description:

```
function getDaoGrantAssetPoolNft(bytes32 daoId) external view returns (address);
```

This function is responsible for retrieving the address of the GrantAssetPoolNFT contract for the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'address', representing the address of the GrantAssetPoolNFT contract.

getDaoTreasury

Interface Description:

```
function getDaoTreasury(bytes32 daoId) external view returns (address);
```

This function is responsible for retrieving the address of the Treasury contract for the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'address', representing the address of the Treasury contract.

getDaoEditInformationPermissionNft

Interface Description:

```
function getDaoEditInformationPermissionNft(bytes32 daoId) external view returns
   (address, uint256);
```

This function is responsible for retrieving the NFT that binds to the "Edit Information" right of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'address', representing the address of the accociated ERC721 contract of the NFT.

The return value is of type 'uint256', representing the token ID of the NFT.

getDaoEditParameterPermissionNft

Interface Description:

```
function getDaoEditParameterPermissionNft(bytes32 daoId) external view returns (
  address, uint256);
```

This function is responsible for retrieving the NFT that binds to the "Edit On-chain Parameter" right of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'address', representing the address of the accociated ERC721 contract of the NFT.

The return value is of type 'uint256', representing the token ID of the NFT.

getDaoEditStrategyPermissionNft

Interface Description:

```
function getDaoEditStrategyPermissionNft(bytes32 daoId) external view returns (
  address, uint256);
```

This function is responsible for retrieving the NFT that binds to the "Edit Strategies" right of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'address', representing the address of the accociated ERC721 contract of the NFT.

The return value is of type 'uint256', representing the token ID of the NFT.

getDaoRewardPermissionNft

Interface Description:

```
function getDaoRewardPermissionNft(bytes32 daoId) external view returns (address
  , uint256);
```

This function is responsible for retrieving the NFT that ties to the Starter reward right of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

Interface Return Value:

The return value is of type ‘address’, representing the address of the accociated ERC721 contract of the NFT.

The return value is of type ‘uint256’, representing the token ID of the NFT.

getTreasuryTransferAssetPermissionNft

Interface Description:

```
function getTreasuryTransferAssetPermissionNft(bytes32 daoId) external view
  returns (address, uint256);
```

This function is responsible for retrieving the NFT that binds to the “Transfer Treasury Asset" right the specific SeedNodes. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

Interface Return Value:

The return value is of type ‘address’, representing the address of the accociated ERC721 contract of the NFT.

The return value is of type ‘uint256’, representing the token ID of the NFT.

getTreasurySetTopUpRatioPermissionNft

Interface Description:

```
function getTreasurySetTopUpRatioPermissionNft(bytes32 daoId) external view
  returns (address, uint256);
```

This function is responsible for retrieving the NFT that binds to the “TopUp Unlock Distribution Ratio" right of the specific SeedNodes. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'address', representing the address of the accociated ERC721 contract of the NFT.

The return value is of type 'uint256', representing the token ID of the NFT.

getTreasuryEditInformationPermissionNft

Interface Description:

```
function getTreasuryEditInformationPermissionNft(bytes32 daoId) external view
  returns (address, uint256);
```

This function is responsible for retrieving the NFT that binds to the "Edit SeedNodes Information" right of the specific SeedNodes. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by 'bytes32'.

Interface Return Value:

The return value is of type 'address', representing the address of the accociated ERC721 contract of the NFT.

The return value is of type 'uint256', representing the token ID of the NFT.

getDaoEditInformationPermission

Interface Description:

```
function getDaoEditInformationPermission(bytes32 daoId, address account)
  external view returns (bool);
```

This function is responsible for checking whether the specific account has the "Edit Information" right of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

account The address of the account, represented by ‘address’.

Interface Return Value:

A boolean value representing whether the specific account has the “Edit Information" right.

getDaoEditParameterPermission

Interface Description:

```
function getDaoEditParameterPermission(bytes32 daoId, address account) external
  view returns (bool);
```

This function is responsible for checking whether the specific account has the “Edit On-chain Parameter" right of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

account The address of the account, represented by ‘address’.

Interface Return Value:

A boolean value representing whether the specific account has the “Edit On-chain Parameter" right.

getDaoEditStrategyPermission

Interface Description:

```
function getDaoEditStrategyPermission(bytes32 daoId, address account) external
  view returns (bool);
```

This function is responsible for checking whether the specific account has the “Edit Strategies" right of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

account The address of the account, represented by ‘address’.

Interface Return Value:

A boolean value representing whether the specific account has the “Edit Strategies" right.

getDaoRewardPermission

Interface Description:

```
function getDaoRewardPermission(bytes32 daoId, address account) external view
  returns (bool);
```

This function is responsible for checking whether the specific account has the Starter reward right of the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

account The address of the account, represented by ‘address’.

Interface Return Value:

A boolean value representing whether the specific account has the Starter reward right.

getTreasuryTransferAssetPermission

Interface Description:

```
function getTreasuryTransferAssetPermission(bytes32 daoId, address account)
  external view returns (bool);
```

This function is responsible for checking whether the specific account has the “Transfer Treasury Asset" right the specific SeedNodes. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

account The address of the account, represented by ‘address’.

Interface Return Value:

A boolean value representing whether the specific account has the “Transfer Treasury Asset" right.

getTreasurySetTopUpRatioPermission

Interface Description:

```
function getTreasurySetTopUpRatioPermission(bytes32 daoId, address account)
  external view returns (bool);
```

This function is responsible for checking whether the specific account has the “TopUp Unlock Distribution Ratio" right of the specific SeedNodes. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

account The address of the account, represented by ‘address’.

Interface Return Value:

A boolean value representing whether the specific account has the “TopUp Unlock Distribution Ratio" right.

getTreasuryEditInformationPermission

Interface Description:

```
function getTreasuryEditInformationPermission(bytes32 daoId, address account)
  external view returns (bool);
```

This function is responsible for checking whether the specific account has the “Edit SeedNodes Information" right of the specific SeedNodes. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

account The address of the account, represented by ‘address’.

Interface Return Value:

A boolean value representing whether the specific account has the “Edit SeedNodes Information" right.

getDaoNeedMintableWork

Interface Description:

```
function getDaoNeedMintableWork(bytes32 daoId) external view returns (bool);
```

This function is responsible for checking whether Pass Cards are reserved for the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

Interface Return Value:

A boolean value representing whether Pass Cards are reserved for the specific SubNodes.

getDaoInputToken

Interface Description:

```
function getDaoInputToken(bytes32 daoId) external view returns (address);
```

This function is responsible for retrieving the InputToken address for the specific SubNodes. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

Interface Return Value:

The return value is of type ‘address’, representing the InputToken address of the specific SubNodes. If zero address is returned, it means that the SubNodes uses ETH as InputToken.

4.10 IPDProtocolSetter

This interface includes writable methods for setting SubNodes attributes.

setMintCapAndPermission

Interface Description:

```
function setMintCapAndPermission(bytes32 daoId,uint32 daoMintCap,
  UserMintCapParam[] calldata userMintCapParams, NftMinterCapInfo[] calldata
  nftMinterCapInfo,NftMinterCapIdInfo[] calldata nftMinterCapIdInfo, Whitelist
  memory whitelist, Blacklist memory blacklist, Blacklist memory unblacklist)
  external;
```

This function is responsible for setting the minting strategies of the specific SubNodes. It is required the caller has the “Edit Strategies" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

daoMintCap See *IPDCreate* interface for detail, represented by ‘uint32’.

userMintCapParams See *IPDCreate* interface for detail, represented by ‘UserMintCapParam[]’ struct array.

nftMinterCapInfo See *IPDCreate* interface for detail, represented by ‘NftMinterCapInfo[]’ struct array.

nftMinterCapIdInfo See *IPDCreate* interface for detail, represented by ‘NftMinterCapIdInfo[]’ struct array.

whitelist See *IPDCreate* interface for detail, represented by ‘Whitelist’ struct array.

blacklist See *IPDCreate* interface for detail, represented by ‘Blacklist’ struct array.

unblacklist The set of Minters and Creators addresses that to be removed from the blacklist (to be not prohibited), represented by ‘Blacklist’ struct array.

Interface Return Value:

This function does not return any value.

setDaoPriceTemplate

Interface Description:

```
function setDaoPriceTemplate(bytes32 daoId, PriceTemplateType priceTemplateType,
   uint256 priceFactor) external;
```

This function is responsible for setting the type of System Pricing template for the specific SubNodes. It is required the caller has the “Edit On-chain Parameter" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

priceTemplateType See *IPDCreate* interface for detail, represented by ‘PriceTemplateType’ enum.

priceFactor The price factor of the price template, See *IPDCreate* interface for detail, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setDaoNftMaxSupply

Interface Description:

```
function setDaoNftMaxSupply(bytes32 daoId, uint256 newMaxSupply) external;
```

This function is responsible for setting the total amount of NFTs that can be minted in the specific SubNodes. It is required the caller has the “Edit On-chain Parameter" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

newMaxSupply The new total mintable NFT amount for the specific SubNodes, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setDaoFloorPrice

Interface Description:

```
function setDaoFloorPrice(bytes32 daoId, uint256 newFloorPrice) external;
```

This function is responsible for setting the Floor Price of the specific SubNodes. It is required the caller has the “Edit On-chain Parameter" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

newFloorPrice The new Floor Price of the specific SubNodes, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setRoundMintCap

Interface Description:

```
function setRoundMintCap(bytes32 daoId, uint256 roundMintCap) external;
```

This function is responsible for setting the maximum number of NFTs that can be minted in the specific SubNodes for each (Minter) address. It is required the caller has the “Edit On-chain Parameter" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

roundMintCap The new maximum number of NFTs that can be minted in the specific SubNodes for each (Minter) address, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setWhitelistMintCap

Interface Description:

```
function setWhitelistMintCap(bytes32 daoId, address whitelistUser, uint32
  whitelistUserMintCap) external;
```

This function is responsible for adding a sepecific address with a specific mint limit to the Minter WhiteList of specific SubNodes . It is required the caller has the “Edit Strategies" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

whitelistUser The address of the specific (Minter) address, represented by ‘address’.

whitelistUserMintCap The new maximum number of NFTs that can be minted in the specific SubNodes for the specific (Minter) address, represented by ‘uint32’.

Interface Return Value:

This function does not return any value.

setDaoUnifiedPrice

Interface Description:

```
function setDaoUnifiedPrice(bytes32 daoId, uint256 newUnifiedPrice) external;
```

This function is responsible for setting the Unified Price of the specific SubNodeswith Unified Fixed Price Mode. It is required the caller has the “Edit On-chain Parameter" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

newUnifiedPrice The new Unified Price of the specific SubNodes, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setChildren

Interface Description:

```
function setChildren(bytes32 daoId, SetChildrenParam calldata vars) external;
```

This function is responsible for setting the IDs of the children SubNodes (recipient SubNodesfor Block Reward issuance) of the specific SubNodes. It is required the caller has the “Edit On-chain Parameter" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

newUnifiedPrice The new ID list of the children SubNodes, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setRatio

Interface Description:

```
function setRatio(bytes32 daoId, AllRatioParam calldata vars) external;
```

This function is responsible for setting the MintFee distribution ratios and role coefficients for Internal Distribution of the specific SubNodes. It is required the caller has the “Edit On-chain Parameter" right. The function takes the following parameters:

Interface Parameters:

vars The *AllRatioParam* struct, see *IPDCreate* interface for detail.

Interface Return Value:

This function does not return any value.

setDaoRemainingRound

Interface Description:

```
function setDaoRemainingRound(bytes32 daoId, uint256 newRemainingRound) external
  ;
```

This function is responsible for setting the remaining number of Blocks for the specific SubNodes. If the SubNodes has already ended, calling this function can restart the SubNodes.It is required the caller has the “Edit On-chain Parameter" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

newRemainingRound The new remaining number of Blocks for the specific SubNodes, represented by ‘uint256’. If zero value is provided or the SubNodesis in Infinite Mode, it just returns.

Interface Return Value:

This function does not return any value.

changeDaoInfiniteMode

Interface Description:

```
function changeDaoInfiniteMode(bytes32 daoId, uint256 remainingRound) external;
```

This function is responsible for changing the SubNodes’ Infinite Mode state. If this operation is to turn off the Infinite Mode (which means it was on before the call), the *remainingRound* must indicate the number of remaining blocks for this SubNodes; if it turns on the Infinite Mode for an ended SubNodes, then the SubNodesis restarted. It is required the caller has the “Edit On-chain Parameter" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

remainingRound The new remaining number of Blocks for the specific SubNodes, represented by ‘uint256’. For turning off, if zero value is provided, it reverts; for turning on, it has no effect.

Interface Return Value:

This function does not return any value.

setDaoEditInformationPermission

Interface Description:

```
function setDaoEditInformationPermission(bytes32 daoId, address daoNftAddress,
  uint256 tokenId) external;
```

This function is responsible for setting the NFT that binds to the “Edit Information" right of the specific SubNodes. It is required the caller has the “Edit Information" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

daoNftAddress The address of the ERC721 contract of the NFT, represented by ‘address’.

tokenId The token ID of the NFT, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setDaoEditParamPermission

Interface Description:

```
function setDaoEditParamPermission(bytes32 daoId, address daoNftAddress, uint256
   tokenId) external;
```

This function is responsible for setting the NFT that binds to the “Edit On-chain Parameter" right of the specific SubNodes. It is required the caller has the “Edit On-chain Parameter" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

daoNftAddress The address of the ERC721 contract of the NFT, represented by 'address'.

tokenId The token ID of the NFT, represented by 'uint256'.

Interface Return Value:

This function does not return any value.

setDaoEditStrategyPermission

Interface Description:

```
function setDaoEditStrategyPermission(bytes32 daoId, address daoNftAddress,
  uint256 tokenId) external;
```

This function is responsible for setting the NFT that binds to the "Edit Strategies" right of the specific SubNodes. It is required the caller has the "Edit Strategies" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

daoNftAddress The address of the ERC721 contract of the NFT, represented by 'address'.

tokenId The token ID of the NFT, represented by 'uint256'.

Interface Return Value:

This function does not return any value.

setDaoRewardPermission

Interface Description:

```
function setDaoRewardPermission(bytes32 daoId, address daoNftAddress, uint256
  tokenId) external;
```

This function is responsible for setting the NFT that binds to the Starter reward right of the specific SubNodes. It is required the caller has the Starter reward right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

daoNftAddress The address of the ERC721 contract of the NFT, represented by 'address'.

tokenId The token ID of the NFT, represented by 'uint256'.

Interface Return Value:

This function does not return any value.

setDaoControlPermission

Interface Description:

```
function setDaoControlPermission(bytes32 daoId, address daoNftAddress, uint256
  tokenId) external;
```

This function is responsible for setting the (one) NFT that binds to above four rights collectively of the specific SubNodes. It is required the caller has all above four rights. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by 'bytes32'.

daoNftAddress The address of the ERC721 contract of the NFT, represented by 'address'.

tokenId The token ID of the NFT, represented by 'uint256'.

Interface Return Value:

This function does not return any value.

setTreasuryEditInformationPermission

Interface Description:

```
function setTreasuryEditInformationPermission(bytes32 daoId, address
  daoNftAddress, uint256 tokenId) external;
```

This function is responsible for setting the NFT that binds to the “Edit SeedNodes Information" right of the specific SeedNodes. It is required the caller has the “Edit SeedNodes Information" right. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

daoNftAddress The address of the ERC721 contract of the NFT, represented by ‘address’.

tokenId The token ID of the NFT, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setTreasuryTransferAssetPermission

Interface Description:

```
function setTreasuryTransferAssetPermission(bytes32 daoId, address daoNftAddress
  , uint256 tokenId) external;
```

This function is responsible for setting the NFT that binds to the “Transfer Treasury Asset" right of the specific SeedNodes. It is required the caller has the “Transfer Treasury Asset" right. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

daoNftAddress The address of the ERC721 contract of the NFT, represented by ‘address’.

tokenId The token ID of the NFT, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setTreasurySetTopUpRatioPermission

Interface Description:

```
function setTreasurySetTopUpRatioPermission(bytes32 daoId, address daoNftAddress
  , uint256 tokenId) external;
```

This function is responsible for setting the NFT that binds to the “TopUp Unlock Distribution Ratio" right of the specific SeedNodes. It is required the caller has the “TopUp Unlock Distribution Ratio" right. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

daoNftAddress The address of the ERC721 contract of the NFT, represented by ‘address’.

tokenId The token ID of the NFT, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setTreasuryControlPermission

Interface Description:

```
function setTreasuryControlPermission(bytes32 daoId, address daoNftAddress,
  uint256 tokenId) external;
```

This function is responsible for setting the (one) NFT that binds to above three rights collectively of the specific SeedNodes. It is required the caller has all above three rights. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

daoNftAddress The address of the ERC721 contract of the NFT, represented by ‘address’.

tokenId The token ID of the NFT, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setDaoTopUpInputToRedeemPoolRatio

Interface Description:

```
function setDaoTopUpInputToRedeemPoolRatio(bytes32 daoId, uint256
  inputToRedeemPoolRatio) external;
```

This function is responsible for setting the ratio that unlocked InputToken from TopUp Accounts transferred to the NodesRedeemPool of the sepecific SubNodes. It is required the caller has the “TopUp Unlock Distribution Ratio" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

inputToRedeemPoolRatio The ratio that unlocked InputToken from TopUp Accounts transferred to the NodesRedeemPool of the sepecific SubNodes, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setDaoTopUpOutputToTreasuryRatio

Interface Description:

```
function setDaoTopUpOutputToTreasuryRatio(bytes32 daoId, uint256
  outputToTreasuryRatio) external;
```

This function is responsible for setting the ratio that unlocked OutputToken from TopUp Accounts transferred to the Treasury of the sepecific SubNodes. It is required the caller has the “TopUp Unlock Distribution Ratio" right. The function takes the following parameters:

Interface Parameters:

daoId The ID of the specific SubNodes, represented by ‘bytes32’.

outputToTreasuryRatio The ratio that unlocked OutputToken from TopUp Accounts transferred to the Treasury of the sepecific SubNodes, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setDefaultTopUpInputToRedeemPoolRatio

Interface Description:

```
function setDefaultTopUpInputToRedeemPoolRatio(bytes32 daoId, uint256
  inputToRedeemPoolRatio) external;
```

This function is responsible for setting the default InputToken ratio for TopUp Account Unlock Distribution for a specific SeedNodes, that any newly created SubNodes in the SeedNodes will follow that ratio. It is required the caller has the “TopUp Unlock Distribution Ratio" right. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

inputToRedeemPoolRatio The default InputToken ratio for TopUp Account Unlock Distribution for the specific SeedNodes, represented by ‘uint256’.

Interface Return Value:

This function does not return any value.

setDefaultTopUpOutputToTreasuryRatio

Interface Description:

```
function setDefaultTopUpOutputToTreasuryRatio(bytes32 daoId, uint256
  outputToTreasuryRatio) external;
```

This function is responsible for setting the default OutputToken ratio for TopUp Account Unlock Distribution for a specific SeedNodes, that any newly created SubNodes in the SeedNodes will follow that ratio. It is required the caller has the “TopUp Unlock Distribution Ratio" right. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by 'bytes32'.

outputToTreasuryRatio The default OutputToken ratio for TopUp Account Unlock Distribution for the specific SeedNodes, represented by 'uint256'.

Interface Return Value:

This function does not return any value.

setTopUpInputSplitRatio

Interface Description:

```
function setTopUpInputSplitRatio(bytes32 daoId, uint256 defaultInputRatio,
  bytes32[] calldata subDaoIds, uint256[] calldata inputRatios) external;
```

This function is responsible for collectively setting the default InputToken ratio for a specific SeedNodes and InputToken ratio for a list of SubNodes, for TopUp Account Unlock Distribution. It is required the caller has the "TopUp Unlock Distribution Ratio" right. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by 'bytes32'.

defaultInputRatio The default InputToken ratio for TopUp Account Unlock Distribution for the specific SeedNodes, represented by 'uint256'.

subDaoIds The list of the SubNodes IDs, represented by 'bytes32[]'.

inputRatios The list of the ratios of the InputToken ratio for TopUp Account Unlock Distribution for the list of SubNodes, represented by 'uint256[]'. The length of this array must be equal to the length of the *subDaoIds* array.

Interface Return Value:

This function does not return any value.

setTopUpOutputSplitRatio

Interface Description:

```
function setTopUpOutputSplitRatio(bytes32 daoId, uint256 defaultOutputRatio,
  bytes32[] calldata subDaoIds, uint256[] calldata outputRatios) external;
```

This function is responsible for collectively setting the default OutputToken ratio for a specific SeedNodes and OutputToken ratio for a list of SubNodes, for TopUp Account Unlock Distribution. It is required the caller has the “TopUp Unlock Distribution Ratio" right. The function takes the following parameters:

Interface Parameters:

daoId Any SubNodes ID in the specific SeedNodes, represented by ‘bytes32’.

defaultOutputRatio The default OutputToken ratio for TopUp Account Unlock Distribution for the specific SeedNodes, represented by ‘uint256’.

subDaoIds The list of the SubNodes IDs, represented by ‘bytes32[]’.

outputRatios The list of the ratios of the OutputToken ratio for TopUp Account Unlock Distribution for the list of SubNodes, represented by ‘uint256[]’. The length of this array must be equal to the length of the *subDaoIds* array.

Interface Return Value:

This function does not return any value.

setDaoParams

Interface Description:

```
function setDaoParams(SetDaoParam calldata vars) external;
```

This function is responsible for setting the parameters of the specific SubNodes. It is required the caller has the “Edit On-chain Parameter" right. The function takes the following parameters:

Interface Parameters:

vars

```
struct SetDaoParam {
  bytes32 daoId;
  uint256 nftMaxSupplyRank;
  uint256 remainingRound;
```

```
            uint256 daoFloorPrice;
            PriceTemplateType priceTemplateType;
            uint256 nftPriceFactor;
            uint256 dailyMintCap;
            uint256 unifiedPrice;
            bool changeInfiniteMode;
            SetChildrenParam setChildrenParam;
            AllRatioParam allRatioParam;
          }
```

This structure contains all the necessary parameters for setting SubNodes parameters.

daoId The ID of the specific SubNodes, represented by 'bytes32'.

nftMaxSupplyRank The maximum rank of the NFTs that can be minted in the specific SubNodes, with the current setup including five ranks corresponding to the following quantities: [1000, 5000, 10,000, 50,000, 100,000], represented by 'uint256'.

remainingRound The remaining number of Blocks for the specific SubNodes, represented by 'uint256'.

daoFloorPrice The Floor Price of the specific SubNodes, represented by 'uint256'.

priceTemplateType The Price Template Type of the specific SubNodes, represented by 'PriceTemplateType'.

nftPriceFactor The Price Factor for System Pricing of the specific SubNodes, represented by 'uint256'.

dailyMintCap The minting limit in each Block of the SubNodes, represented by 'uint256'.

unifiedPrice The Unified Price of the specific SubNodes, represented by 'uint256'.

changeInfiniteMode The flag whether to change the Infinite Mode state of the specific SubNodes, represented by 'bool'.

setChildrenParam The *SetChildrenParam* struct, see *IPDCreate* interface for detail.

allRatioParam The *AllRatioParam* struct, see *IPDCreate* interface for detail.

Interface Return Value:

This function does not return any value.