NNS 系列之 Registry

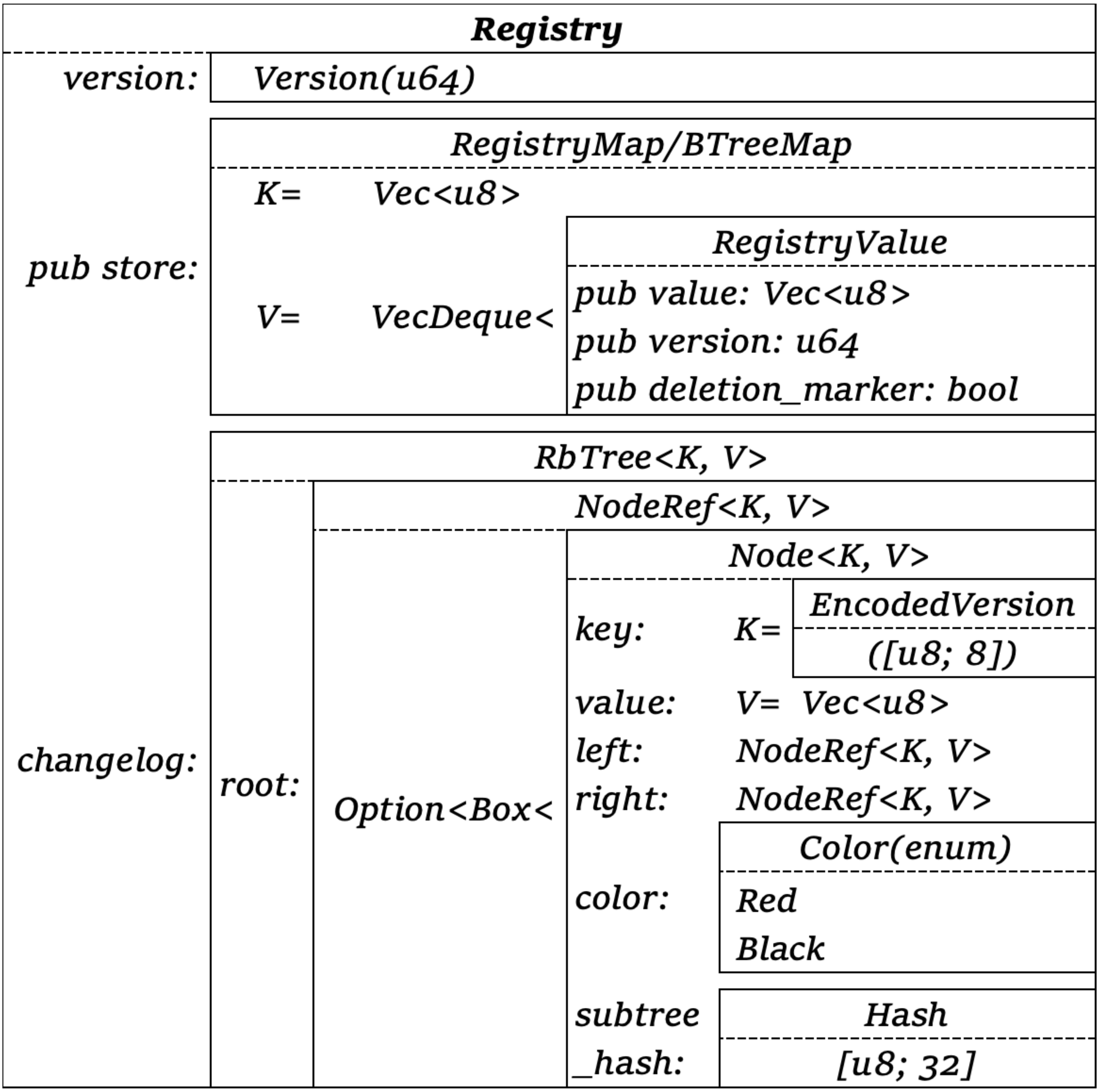
Contents

概述

Registry canister，功能上是执行。在 Governance 得到投票结果之后，Governance 就会调用 Registry 里面的 Functions 来具体执行。

数据结构

REGISTRY



- 核心的数据结构就是 Registry。它有三个字段:
  - version，一个全局的计数器，每次有 mutation 作用于 Registry，就会 +1。
  - store 是真正存数据的地方，一个 Map，它的 Key 和 Value 需要转化成 u8（bytes） 数组。
  - changelog 所有的 mutation 的历史记录。

接下来看一下怎么示例化的。

实现了一个全局唯一的静态变量，`REGISTRY`，来存储所有的状态。

并且通过两个函数，`registry()` 以及 `registry_mut()` 来获取到这个变量。它们的区别是上面的 `registry()` 获取到的 `REGISTRY` 只读，不能更改它的状态，下面这个 `registry_mut()` 还能更改其状态。

```
static mut REGISTRY: Option<Registry> = None;

fn registry() -> &'static Registry {
    registry_mut()
}

fn registry_mut() -> &'static mut Registry {
    unsafe {
        if let Some(g) = &mut REGISTRY {
            g
        } else {
            REGISTRY = Some(Registry::new());
            registry_mut()
        }
    }
}
```

接口函数

canister_init

在 `canister` 部署的时候，需要初始化一些状态，执行初始化函数，和 Solidity 的 constructor（构造函数）类似。比如部署一个 token 合约，需要初始化 token 的名字，符号，总量等信息。这就是 canister_init 函数的功能。

先执行了 `hook()` 函数，`hook()` 来自于 `dfn_core` 这个包里面，去看一下它的具体内容，它执行了两个函数，`set_stdout()` 和 `set_panic_hook()`。根据 `set_panic_hook()` 的具体内容，总的来说，`hook()` 它的功能是设置好标准输出，以及程序出现 Panic 时，把错误消息，以及所在文件，第几行，第几列的 log 打印出来。

```
//
https://github.com/dfinity/ic/blob/8fffeb4be1/rs/registry/canister/caniste
r/canister.rs#L118
dfn_core::printer::hook();

//
https://github.com/dfinity/ic/blob/8fffeb4be1/rs/rust_canisters/dfn_core/s
rc/printer.rs
/// Sets stdout, stderr, and a custom panic hook
pub fn hook() {
    set_stdout();
    set_panic_hook();
}

//
```

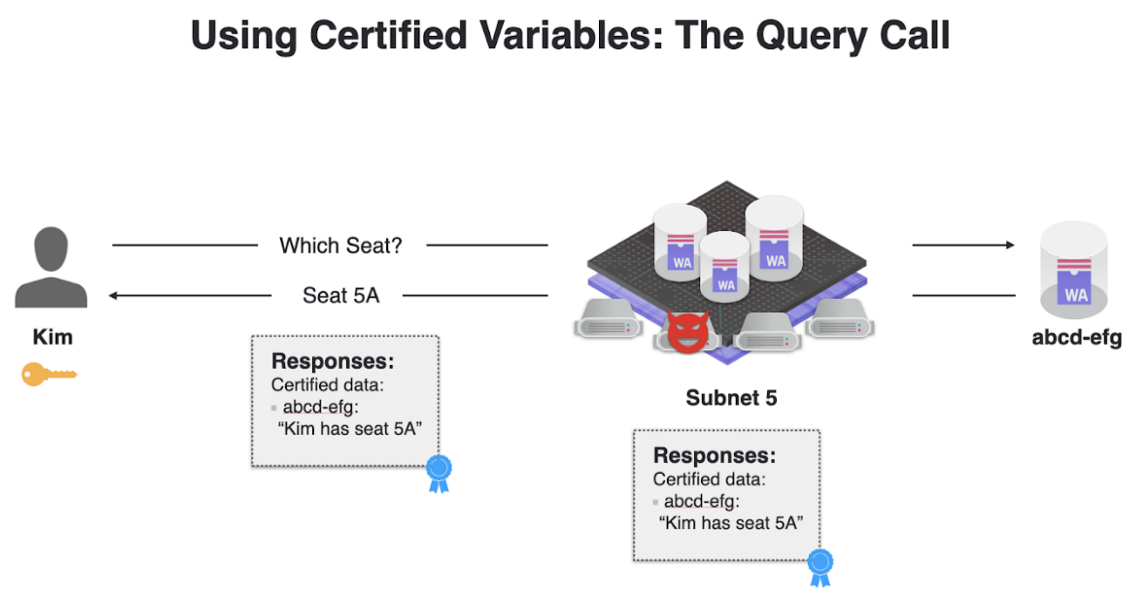
```
https://github.com/dfinity/ic/blob/8fffeb4be1/rs/rust_canisters/dfn_core/src/printer.rs#L118-L146
/// Sets a custom panic hook, uses debug.trace
pub fn set_panic_hook() {
    panic::set_hook(Box::new(|info| {
        let file = info.location().unwrap().file();
        let line = info.location().unwrap().line();
        let col = info.location().unwrap().column();

        let msg = match info.payload().downcast_ref::<&'static str>() {
            Some(s) => *s,
            None => match info.payload().downcast_ref::<String>() {
                Some(s) => &s[..],
                None => "Box<Any>",
            },
        };

        let err_info = format!("Panicked at '{}', {}:{}:{}", msg, file,
line, col);

        unsafe {
            log(&err_info);
        }
        crate::api::trap_with(&err_info);
    }));
}
```

接下来执行 `recertify_registry()` 这个函数。先整体说一下 certified_data。




```
recertify_registry();
```

```
fn recertify_registry() {
    use ic_certified_map::{fork_hash, labeled_hash};

    let root_hash = fork_hash(
        &current_version_tree(registry().latest_version()).reconstruct(),
        &labeled_hash(b"delta", &registry().changelog().root_hash()),
    );
    set_certified_data(&root_hash);
}
```

先从里面的 `registry()` 开始看，它返回了一个只读的那个全局的 Registry 实例。然后去拿它的 `latest_version()`，其实就是拿的它 version 字段的值，当前版本号，一个 u64 的值。

```
//
https://github.com/dfinity/ic/blob/8fffeb4be1/rs/registry/canister/src/reg
istry.rs#L177-L179
pub fn latest_version(&self) -> Version {
    self.version
}
```

然后把这个 Version 输入到 `current_version_tree()` 里面，看一下 `current_version_tree()`，它接受一个 Version 类型，我们之前在 struct 看过，其实就是 u64，然后返回一个 HashTree 里面的 Labeled 节点，把 current_version 存为数据标签，连接着一个 Leaf 叶子节点，里面存了传进去的 Version 的编码。(leb128 是 Little Endian Base 128 是编码算法，对于存储大整数有空间优势)

```
use ic_certified_map::{labeled, HashTree};

/// The maximum amount of bytes a 64-bit number can occupy when encoded in
/// LEB128.
const MAX_U64_ENCODING_BYTES: usize = 10;

pub fn current_version_tree(v: Version) -> HashTree<'static> {
    let mut buf = Vec::with_capacity(MAX_U64_ENCODING_BYTES);
    leb128::write::unsigned(&mut buf, v).unwrap();
    labeled(
        b"current_version",
        HashTree::Leaf(std::borrow::Cow::from(buf)),
    )
}
```

接下来需要看一下 HashTree 是怎么回事，以及包括 ic-certified-map crate（包） 的函数：fork_hash, labeled_hash。

下面定义了一个 Hash, Sha256 的输出，32 位的 u8 数组。还定义了一个 HashTree 的数据结构，一个树结构，里面有几类节点，比如空节点；下面有两个子节点，它本身没有任何值的 Fork 节点；下面有一个节点，并且本身带标签（数据）的 Labeled 节点；或者只有数据的 Leaf 节点，或者是 [u8; 32] 的 Pruned 节点。

```rust
// https://github.com/dfinity/cdk-rs/blob/319795e9b4/src/ic-certified-map/src/hashtree.rs#L9-L49

/// SHA-256 hash bytes.
pub type Hash = [u8; 32];

/// HashTree as defined in the interfaces spec.
/// https://sdk.dfinity.org/docs/interface-spec/index.html#_certificate
#[derive(Debug)]
pub enum HashTree<'a> {
    Empty,
    Fork(Box<(HashTree<'a>, HashTree<'a>)>),
    Labeled(&'a [u8], Box<HashTree<'a>>),
    Leaf(Cow<'a, [u8]>),
    Pruned(Hash),
}

pub fn fork<'a>(l: HashTree<'a>, r: HashTree<'a>) -> HashTree<'a> {
    HashTree::Fork(Box::new((l, r)))
}

pub fn labeled<'a>(l: &'a [u8], t: HashTree<'a>) -> HashTree<'a> {
    HashTree::Labeled(l, Box::new(t))
}

pub fn fork_hash(l: &Hash, r: &Hash) -> Hash {
    let mut h = domain_sep("ic-hashtree-fork");
    h.update(&l[..]);
    h.update(&r[..]);
    h.finalize().into()
}

pub fn leaf_hash(data: &[u8]) -> Hash {
    let mut h = domain_sep("ic-hashtree-leaf");
    h.update(data);
    h.finalize().into()
}

pub fn labeled_hash(label: &[u8], content_hash: &Hash) -> Hash {
    let mut h = domain_sep("ic-hashtree-labeled");
    h.update(label);
    h.update(&content_hash[..]);
    h.finalize().into()
}

fn domain_sep(s: &str) -> sha2::Sha256 {
    let buf: [u8; 1] = [s.len() as u8];
    let mut h = Sha256::new();
    h.update(&buf[..]);
    h.update(s.as_bytes());
    h
}
```

所以 fork_hash, labeled_hash 都是输入不可变量，通过标准的 Sha256，然后往 Hash 的 buf 里面，填充一些自定义字段，比如 ic-hashtree-fork 这样的字符串。最后输出 32 位的 bytes 数组。

HashTree 实现了一个方法，reconstruct() 其中输入一个 HashTree，输出一个 Hash。Hash 就是 32 位的 Bytes 数组。`reconstruct()` 这个函数，和前面的 fork_hash(), labeled_hash(), leaf_hash() 相呼应，总的来说，将一个 HashTree 这样的树，递归的 hash 了一遍，并最后输出一个 32 位的 u8 数组。

```
impl HashTree<'_> {
    pub fn reconstruct(&self) -> Hash {
        match self {
            Self::Empty => domain_sep("ic-hashtree-
empty").finalize().into(),
            Self::Fork(f) => fork_hash(&f.0.reconstruct(),
&f.1.reconstruct()),
            Self::Labeled(l, t) => {
                let thash = t.reconstruct();
                labeled_hash(l, &thash)
            }
            Self::Leaf(data) => leaf_hash(data),
            Self::Pruned(h) => *h,
        }
    }
}
```

可以再去我们之前的 Struct 图看一下，RbTree 是怎么定义的。这里给它实现了一个 root_hash() 的方法，其实就是拿的它 subtree_hash。

```
// https://github.com/dfinity/cdk-rs/blob/319795e9b4/src/ic-certified-
map/src/hashtree.rs#L57-L63
impl<K: 'static + AsRef<[u8]>, V: AsHashTree + 'static> AsHashTree for
RbTree<K, V> {
    fn root_hash(&self) -> Hash {
        match self.root.as_ref() {
            None => Empty.reconstruct(),
            Some(n) => n.subtree_hash,
        }
    }
}
```

然后 labeled_hash 输入一个 "delta" 的字符串的 bytes 形式，以及 changelog 的 subtree_hash，将结果和前面的到的 reconstruct() 的结果输入到 fork_hash，最后得到 root_hash。

```
    set_certified_data(&root_hash);

//
https://github.com/dfinity/ic/blob/8fffeb4be1/rs/rust_canisters/dfn_core/s
rc/api.rs#L840-L847
/// Sets the certified data of this canister.
```

```
///
/// # Panics
///
/// * This function traps if data.len() > 32.
pub fn set_certified_data(data: &[u8]) {
    // https://smartcontracts.org/docs/interface-spec/index.html#system-
api-certified-data
    unsafe { ic0::certified_data_set(data.as_ptr() as u32, data.len() as
u32) }
}
```

然后调用系统接口 certified_data_set 把这个 root_hash 设置成认证数据。认证数据的具体内容可以看 interface-spec 文档，也可以看官方视频，简单的说是你 query 一个数据的时候，没有经过全网共识，直接从一个节点拿的，可能该节点作恶。certified_data 是之前通过共识过放进去的，下次被 query 时还能携带一个全网的签名，就可以验证这个数据的安全性很高的。好像 Canister Signature 也和这个东西有关，期待有人更深入的分享 certified_data 相关。

所以总的来说，recertify_registry() 只用了当前 Registry 的 version，changelog 这两个字段，经过一系列的 hash，生成了 32 位的 Hash 结果，root_hash。 root_hash 是能够反映 Registry 在 version 和 changelog 上的变化的。最后，这个 root_hash 设置成 certified_data。Registry 里面的 store 字段并没有影响到 root_hash，后续可以看出每次 store 变化，都会往 changelog 里面存这次变化，所以可能 changelog 里面的数据已经能够反映 store 的信息了。

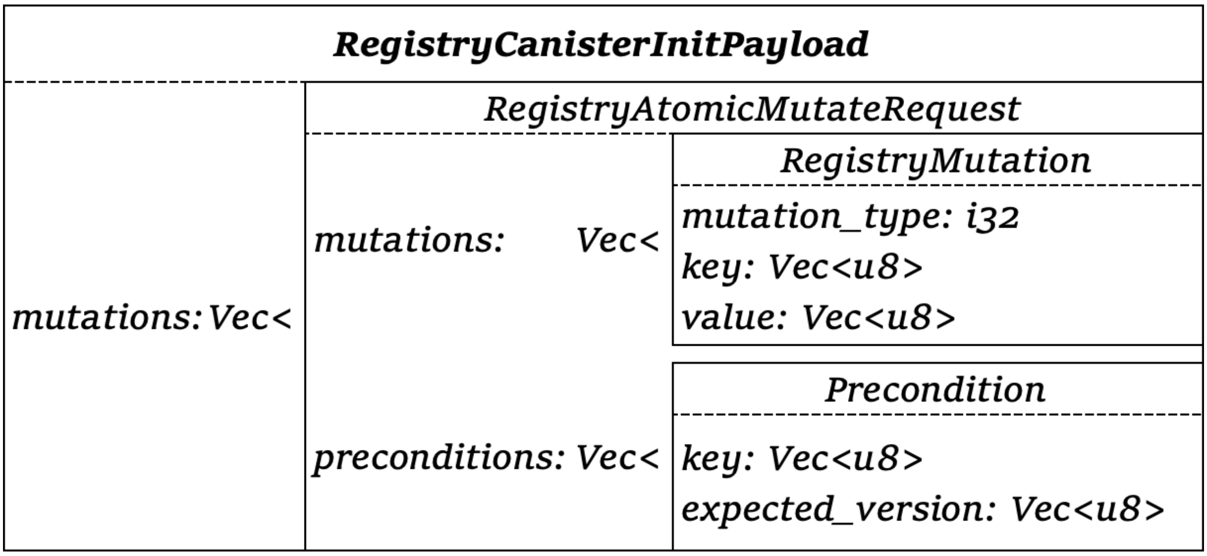
接下来：

```
    let init_payload =
        Decode!(&arg_data(), RegistryCanisterInitPayload)
            .expect("The init argument for the registry canister must be a
Candid-encoded RegistryCanisterInitPayload.");
    println!(
        "{}canister_init: Initializing with: {}",
        LOG_PREFIX, init_payload
    );

//
https://github.com/dfinity/ic/blob/8fffeb4be1/rs/rust_canisters/dfn_core/s
rc/api.rs#L656-L664
/// Returns the argument extracted from the message payload.
pub fn arg_data() -> Vec<u8> {
    let len: u32 = unsafe { ic0::msg_arg_data_size() };
    let mut bytes = vec![0; len as usize];
    unsafe {
        ic0::msg_arg_data_copy(bytes.as_mut_ptr() as u32, 0, len);
    }
    bytes
}
```

然后通过几个系统接口，msg_arg_data_size 和 msg_arg_data_copy，获取到这个函数的输入参数，因为是 canister_init，所以也是这个 canister 的安装参数。这个参数是 RegistryCanisterInitPayload 类型。

接着申明并初始化 init_payload 变量为 `Vec<u8>` 形式的参数解码成 RegistryCanisterInitPayload 类型的数据。



这个类型，首先有个唯一的字段 mutations，它是一个动态数组，而里面的 RegistryAtomicMutateRequest 有两个字段，其中一个也叫 mutations，另一个叫 preconditions。它们里面的字段都包含了一些 `Vec<u8>` 的值。

```
    let registry = registry_mut();

    init_payload
        .mutations
        .into_iter()
        .for_each(|mutation_request| {

registry.maybe_apply_mutation_internal(mutation_request.mutations)
        });
```

接着获取到了 Registry 示例，并且是可变的。接下来修改这个示例。

通过遍历 init_payload 的 mutations 这个数组里面的每一个 RegistryAtomicMutateRequest 类型的值，执行 maybe_apply_mutation_internal 方法，输入 RegistryAtomicMutateRequest 的 mutations 字段的数据（`Vec<RegistryMutation>`）来修改 Registry 示例。

这个方法执行了哪些功能?

```
//
https://github.com/dfinity/ic/blob/8fffeb4be1/rs/registry/canister/src/reg
istry.rs#L271-L294
/// Checks that invariants hold after applying mutations
pub fn maybe_apply_mutation_internal(&mut self, mutations:
Vec<RegistryMutation>) {
```

```
    println!(
        "{}Received a mutate call containing a list of {} mutations",
        LOG_PREFIX,
        mutations.len()
    );

    let errors = self.verify_mutation_type(mutations.as_slice());
    if !errors.is_empty() {
        panic!(
            "{}Transaction rejected because of the following errors:
[{}].",
            LOG_PREFIX,
            errors
                .iter()
                .map(|e| format!("{}", e))
                .collect::<Vec::<String>>()
                .join(", ")
        );
    }

    self.check_global_invariants(mutations.as_slice());
    self.apply_mutations(mutations);
}
```

在做完一些验证和检查之后，真正的状态修改发生在 apply_mutations()

```
//
https://github.com/dfinity/ic/blob/8fffeb4be1/rs/registry/canister/src/reg
istry.rs#L226-L240
/// Applies the given mutations, without any check corresponding
/// to the mutation_type.
///
/// This should be called only after having made sure that all
/// preconditions are satisfied.
fn apply_mutations(&mut self, mutations: Vec<RegistryMutation>) {
    if mutations.is_empty() {
        // We should not increment the version if there is no
        // mutation, so that we keep the invariant that the
        // global version is the max of all versions in the store.
        return;
    }
    self.increment_version();
    self.apply_mutations_as_version(mutations, self.version);
}

//
https://github.com/dfinity/ic/blob/8fffeb4be1/rs/registry/canister/src/reg
istry.rs#L171-L185
/// Increments the latest version of the registry.
fn increment_version(&mut self) -> Version {
    self.version += 1;
    self.version
```

```
}

fn apply_mutations_as_version(
    &mut self,
    mut mutations: Vec<RegistryMutation>,
    version: Version,
) {
    // We sort entries by key to eliminate the difference between
changelog
    // produced by the new version of the registry canister starting from
v1
    // and the changelog recovered from the stable representation of the
    // original version that didn't support certification.
    mutations.sort_by(|l, r| l.key.cmp(&r.key));
    for m in mutations.iter_mut() {
        // We normalize all the INSERT/UPDATE/UPSERT operations to be just
        // UPSERTs. This serves 2 purposes:
        //
        // 1. This significantly simplifies reconstruction of the
changelog
        //    when we deserialize the registry from the original stable
        //    representation.
        //
        // 2. This will play nicely with garbage collection: if an old
        //    INSERT entry is removed, the newly connected clients won't
        //    fail because of an UPDATE in the first survived entry with
the
        //    same key.
        m.mutation_type = match Type::from_i32(m.mutation_type).unwrap() {
            Type::Insert | Type::Update | Type::Upsert => Type::Upsert,
            Type::Delete => Type::Delete,
        } as i32;
    }

    let req = RegistryAtomicMutateRequest {
        mutations,
        preconditions: vec![],
    };
    let bytes = pb_encode(&req);

    // version 是之前已经 +1 的版本，并转换成 [u8; 8]
    self.changelog.insert(version.into(), bytes);

    for mutation in req.mutations {

(*self.store.entry(mutation.key).or_default()).push_back(RegistryValue {
            version,
            value: mutation.value,
            deletion_marker: mutation.mutation_type == Type::Delete as
i32,
        });
    }
}
```

```
pub enum Type {
    /// Key is expected to not exist in the registry at the current
version.
    /// (This includes the case of a key that has existed in the past and
    /// later got deleted).
    /// The mutation will fail otherwise.
    Insert = 0,
    /// Key is expected to exist in the registry at the current version.
    /// The mutation will fail otherwise.
    Update = 1,
    /// Key is expected to exist in the registry at the current version.
    /// The mutation will fail otherwise.
    Delete = 2,
    /// If the key does not exist at the current version, it will be
created.
    /// Otherwise, the value will be updated. The name is common in the
    /// database world, and means Update or Insert.
    Upsert = 4,
}

fn pb_encode(msg: &impl prost::Message) -> Vec<u8> {
    let mut buf = vec![];
    msg.encode(&mut buf).unwrap();
    buf
}
```

先对输入 mutations 进行了修改，然后对整个 RegistryAtomicMutateRequest 编码，并写入 changelog，然后把 RegistryMutation 按照 key 和 value 写入到 store。

总的来说是把参数里面的数据结构写入到 Registry，并且增加了 n 个版本号，插入了 n 条 changelog。在 store 里面增加了 n * m 个数据对。其中 n 是最外面 mutations 的数据长度，m 是里面的 mutations 的平均长度。

需要注意的是

1. changelog 里面，将整个最里面的 mutations 编码作为 value，以 version 作为 key，插入到 changelog 的红黑树里面，已经把版本和对应的数据都保存好了，store 里面存的可能是为了方便查询的冗余数据，因为使用了 Map 结构。
2. RegistryAtomicMutateRequest 里面的 preconditions 字段没有使用，代码里面也是通过构造一个空的数组来填充的：`preconditions: vec![]`

最后，又调用了 `recertify_registry()`，根据刚刚更新的 version 和 changelog 得到 root_hash，然后存到 certified_data 里面。

canister_pre_upgrade

升级前函数，将系统的状态先提前 stable 保存一下，然后在升级的时候 stable 保存的数据是不会丢失的，然后升级之后有一个升级后函数讲数据恢复出来。

```
#[export_name = "canister_pre_upgrade"]
fn canister_pre_upgrade() {
    println!("{}canister_pre_upgrade", LOG_PREFIX);
    let registry = registry_mut();
    let mut serialized = Vec::new();
    let ss = RegistryCanisterStableStorage {
        registry: Some(registry.serializable_form()),
    };
    ss.encode(&mut serialized)
        .expect("Error serializing to stable.");
    stable::set(&serialized);
}
```

这个函数先打印一些信息出来，然后获取 Registry 示例 registry，其实这里没必要用 mut，因为并不会修改状态（删掉 mut 可以编译通过）。然后根据 registry 的信息构造了一个 RegistryCanisterStableStorage 数据，并且把这个数据编码到一个 buffer serialized 里面，然后调用 stable 的 set 接口保存这个数据。

来看一下 stable 的 set 是怎么存的

```
/// The wasm page size is 64KiB
const PAGE_SIZE: f64 = 64.0 * 1024.0;

/// We store all the data prepended by the length of data in the first 4
bytes
/// 0        4              4 + length
/// +--------+-----------------+-------->
/// | length |     content     |  junk
/// +--------+-----------------+-------->

const LENGTH_BYTES: u32 = 4;

//
https://github.com/dfinity/ic/blob/8fffeb4be1/rs/rust_canisters/dfn_core/s
rc/stable.rs#L37-L45
/// Sets the contents of the stable memory
pub fn set(content: &[u8]) {
    let len: u32 = content.len() as u32;
    ensure_capacity(len);
    unsafe {
        ic0::stable_write(LENGTH_BYTES, content.as_ptr() as u32, len);
    }
    set_length(len);
}

fn ensure_capacity(capacity_bytes: u32) {
    let required_pages = (f64::from(capacity_bytes + LENGTH_BYTES) /
PAGE_SIZE).ceil() as u32;
    let current_pages = unsafe { ic0::stable_size() };

    if required_pages > current_pages {
```

```
            let difference = required_pages - current_pages;
            unsafe {
                ic0::stable_grow(difference);
            };
        }
    }

    pub fn set_length(len: u32) {
        ensure_capacity(LENGTH_BYTES);
        let len_bytes = len.to_le_bytes();
        unsafe { ic0::stable_write(0, len_bytes.as_ptr() as u32, LENGTH_BYTES)
    }
    }
```

有一个 wasm page 的概念，每个 page 64 KiB，ensure_capacity 函数将 page 扩展到足够的空间，然后调用系统接口 stable_write 从第 4 bytes 开始往里面写数据，最后往里面的前 4 个 bytes 存放数据的长度。

接口看一下 RegistryCanisterStableStorage 数据结构：

<table>
<tr><th colspan="4">RegistryCanisterStableStorage</th></tr>
<tr><td rowspan="11">pub registry: Option<</td><td colspan="3">RegistryStableStorage</td></tr>
<tr><td colspan="3">pub version: i32</td></tr>
<tr><td rowspan="6">pub deltas: Vec<</td><td colspan="2">RegistryDelta</td></tr>
<tr><td colspan="2">pub key: Vec<u8></td></tr>
<tr><td rowspan="4">pub values: Vec<</td><td>RegistryValue</td></tr>
<tr><td>pub value: Vec<u8></td></tr>
<tr><td>pub version: u64</td></tr>
<tr><td>pub deletion_marker: bool</td></tr>
<tr><td rowspan="3">pub changelog: Vec<</td><td colspan="2">ChangelogEntry</td></tr>
<tr><td colspan="2">pub version: u64</td></tr>
<tr><td colspan="2">pub encoded_mutation: Vec<u8></td></tr>
</table>

因为是要把 Registry 结构存下来，所以和 Registry 结构比较类似。

```
/// Serializes the registry contents using the specified version of stable
/// representation.
fn serializable_form_at(&self, repr_version: ReprVersion) ->
RegistryStableStorage {
    match repr_version {
        ReprVersion::Version1 => RegistryStableStorage {
            version: repr_version as i32,
            deltas: vec![],
            changelog: self
                .changelog
                .iter()
                .map(|(encoded_version, bytes)| ChangelogEntry {
                    version: encoded_version.as_version(),
                    encoded_mutation: bytes.clone(),
                })
                .collect(),
        },
```

```
            ReprVersion::Unspecified => RegistryStableStorage {
                version: repr_version as i32,
                deltas: self
                    .store
                    .iter()
                    .map(|(key, values)| RegistryDelta {
                        key: key.clone(),
                        values: values.iter().cloned().collect(),
                    })
                    .collect(),
                changelog: vec![],
            },
        }
    }

    pub fn serializable_form(&self) -> RegistryStableStorage {
        self.serializable_form_at(ReprVersion::Version1)
    }
```

升级的时候，RegistryStableStorage 的 deltas 字段里面并没有存数据，而且 version 也是指定的，因此原来状态变量 Registry 里面的两个字段，version，store 都没有保存。不过，之前得出，changelog 里面反正有所有的信息：有 n 个树的节点，当前 version 就是 n，store 里面的值也可以通过遍历 RbTree 的节点来得到。

canister_post_upgrade

升级后函数，同样的，执行 hook()，设置好标准输出以及 panic 时打印错误信息，发生 panic 所在的代码位置。

```
#[export_name = "canister_post_upgrade"]
fn canister_post_upgrade() {
    dfn_core::printer::hook();
    println!("{}canister_post_upgrade", LOG_PREFIX);
    // Purposefully fail the upgrade if we can't find authz information.
    // Best to have a broken canister, which we can reinstall, than a
    // canister without authz information.
    let ss =
RegistryCanisterStableStorage::decode(stable::get().as_slice())
        .expect("Error decoding from stable.");
    let registry = registry_mut();
    registry.from_serializable_form(ss.registry.expect("Error decoding
from stable"));

    registry.check_global_invariants(&[]);
    recertify_registry();
}
```

在打印一些升级消息之后，接着从 stable 存储里面读到对应的数据，有一个 stable_read 的系统接口，可以读到。先读前 4 字节，获取整个的长度，然后再根据长度获取存储的数据。

```
/// Gets the contents of the stable memory
pub fn get() -> Vec<u8> {
    let len = length();
    let mut out: Vec<u8> = vec![0; len as usize];
    unsafe {
        ic0::stable_read(out.as_mut_ptr() as u32, LENGTH_BYTES, len as
u32);
    }
    out
}

pub fn length() -> u32 {
    let mut len_bytes: [u8; 4] = [0; 4];
    unsafe {
        ic0::stable_read(len_bytes.as_mut_ptr() as u32, 0, LENGTH_BYTES);
    }
    u32::from_le_bytes(len_bytes)
}
```

看一下怎么从 RegistryStableStorage 恢复出 Registry 的。先检查 Registry 是不是所有的字段是不是空，或者为 0，然后得到 version。之前 version 是通过指定 ReprVersion::Version1，也就是执行 Version1 这个代码分支。先遍历 changelog 字段，获取到 entry 这个 ChangelogEntry 类型的实例，对 Registry 先填充无用信息，目的是把版本号对应起来，然后把 entry 里面的 encoded_mutation 解码成 RegistryAtomicMutateRequest 类型，并将获取到的 mutations 写入 Registry。

而且从前面可以看到，entry（ChangelogEntry 类型）的 encoded_mutation 是直接复制的 Registry 的节点（`Node<K, V>`）的 V，而这个 V 是直接从 RegistryAtomicMutateRequest encode 过来的。所以 canister_init 时所有的数据（包括 store 和 changelog）都会恢复过来。

```
/// Sets the content of the registry from its serialized representation.
///
/// Panics if not currently empty: this is only meant to be used in
/// canister_post_upgrade.
///
/// In post_upgrade, one should do as much verification as possible, and
/// panic for anything unexpected. Indeed, panicking here keeps the
/// pre-upgrade state unchanged, and gives the developer an opportunity
/// to try upgrading to a different wasm binary. As a corollary, any
/// lossy way of handling unexpected data must be banned in
/// post_upgrade.
pub fn from_serializable_form(&mut self, stable_repr:
RegistryStableStorage) {
    assert!(self.store.is_empty());
    assert!(self.changelog.is_empty());
    assert_eq!(self.version, 0);

    let repr_version =
ReprVersion::from_i32(stable_repr.version).unwrap_or_else(|| {
        panic!(
            "Version {} of stable registry representation is not supported
```

```
by this canister",
                stable_repr.version
            )
        });

        match repr_version {
            ReprVersion::Version1 => {
                let mut current_version = 0;
                for entry in stable_repr.changelog {
                    // Code to fix ICSUP-2589.
                    // This fills in missing versions with empty entries so
that clients see an
                    // unbroken sequence.
                    // If the current version is different from the previous
version + 1, we
                    // need to add empty records to fill out the missing
versions, to keep
                    // the invariants that are present in the
                    // client side.
                    for i in current_version + 1..entry.version {
                        let mutations = vec![RegistryMutation {
                            mutation_type: Type::Upsert as i32,
                            key: "_".into(),
                            value: "".into(),
                        }];
                        self.apply_mutations_as_version(mutations, i);
                        self.version = i;
                    }
                    // End code to fix ICSUP-2589

                    let req =
RegistryAtomicMutateRequest::decode(&entry.encoded_mutation[..])
                        .unwrap_or_else(|err| {
                            panic!("Failed to decode mutation@{}: {}",
entry.version, err)
                        });
                    self.apply_mutations_as_version(req.mutations,
entry.version);
                    self.version = entry.version;
                    current_version = self.version;
                }
            }
            ReprVersion::Unspecified => {
                ...
            }
        }
    }
```

总之执行升级后函数，里面会填充一些无用数据，但是之前的数据都恢复过来了。

update_authz

```
#[export_name = "canister_update update_authz"]
fn update_authz() {
    check_caller_is_root();
    over(candid_one, |_: Vec<MethodAuthzChange>| {
        println!(
            "{}update_authz was called. \
                 This does not do anything, since the registry canister no
longer has any \
                 function whose access is controlled using this mechanism.
\
                 TODO(NNS1-413): Remove this once we are sure that there
are no callers.",
            LOG_PREFIX,
        );
    })
}
```

根据打印的信息，执行这个函数应该不会修改任何状态。确实也是这样，它首先调用 check_caller_is_root，通过系统接口 msg_caller_copy 获取的调用者的 PrincipalId，然后检查该 PrincipalId 是否和 Root 的 Canister id 对得上。

```
pub fn check_caller_is_root() {
    if caller() != PrincipalId::from(ic_nns_constants::ROOT_CANISTER_ID) {
        panic!("Only the root canister is allowed to call this method.");
    }
}

/// Returns the caller of the current call.
pub fn caller() -> PrincipalId {
    let len: u32 = unsafe { ic0::msg_caller_size() };
    let mut bytes = vec![0; len as usize];
    unsafe {
        ic0::msg_caller_copy(bytes.as_mut_ptr() as u32, 0, len);
    }
    PrincipalId::try_from(bytes).unwrap()
}
```

然后接着调用了一个 over 函数，输入两个函数作为参数，具体功能还没理得很清楚（Todo），大概为了方便使用不同的编码，比如有的对外接口是基于 candid，有的是 protobuf，做了个封装。这种方式调用函数后续会有很多。

```
/// Over allows you to create canister endpoints easily
/// ```no_run
/// # use dfn_core::over;
/// #[export_name = "canister_query happy_birthday"]
/// fn hb() {
///     fn happy_birthday((name, age): (String, u16)) -> String {
///         format!("Happy Birthday {}", name)
```

```
///     }
///     over(dfn_json::json, happy_birthday)
/// }
/// ```
///
/// This function always call `reply` unless it traps. If you need a function
/// that may `reject` the call, use one of the _may_reject variant.
pub fn over<In, Out, F, Witness>(_: Witness, f: F)
where
    In: FromWire + NewType,
    Out: IntoWire + NewType,
    F: FnOnce(In::Inner) -> Out::Inner,
    Witness: FnOnce(Out, In::Inner) -> (Out::Inner, In),
{
    over_bytes(|inp| {
        // TODO(RPL-266) Rejecting instead of trapping seems more
        // natural for deserialization errors. Debate in Jira.
        let outer = In::from_bytes(inp).expect("Deserialization Failed");
        let input = outer.into_inner();
        let res = f(input);
        let output = Out::from_inner(res);
        Out::into_bytes(output).expect("Serialization Failed")
    })
}
```

current_authz

```
#[export_name = "canister_query current_authz"]
fn current_authz() {
    over(candid, |_: ()| {
        println!(
            "{}current_authz was called. \
                 This always returns the default value, since the registry canister's state no \
                 longer contains a CanisterAuthzInfo. \
                 TODO(NNS1-413): Remove this once we are sure that there are no callers.",
            LOG_PREFIX,
        );
        CanisterAuthzInfo::default()
    })
}
```

同样根据打印的信息，执行这个函数应该不会返回有用信息。只是返回 CanisterAuthzInfo 的初始化数据。

CanisterAuthzInfo	
pub methods_authz: Vec<	*MethodAuthzInfo*
	pub method_name: String
	pub principal_ids: Vec<Vec<u8>>

get_changes_since

get_certified_changes_since

get_value

get_latest_version

get_certified_latest_version

atomic_mutate

atomic_mutate()，这个函数只允许 Governance 和 Root 这两个 canister 来调用。然后，把调用参数反序列化，如果反序列化成功，就对 Registry 的可变实例执行 maybe_apply_mutation_internal() 来进行修改。然后调用 recertify_registry() 来更新 certified_data。最后把返回结果序列化返回给调用者。

```
#[export_name = "canister_update atomic_mutate"]
fn atomic_mutate() {
    let caller = dfn_core::api::caller();
    //
    // - The governance canister is always allowed to mutate the registry
    // - The root canister is also allowed, so that IDs of new NNS
canisters can be
    //   recorded.
    assert!(
        caller == GOVERNANCE_CANISTER_ID.get() || caller ==
ROOT_CANISTER_ID.get(),
        "{}Principal {} is not authorized to call 'atomic_mutate'.",
        LOG_PREFIX,
        caller
    );
    println!("{}call 'atomic_mutate' from {}", LOG_PREFIX, caller);

    let response_pb = match deserialize_atomic_mutate_request(arg_data())
{
        Ok(request_pb) => {

registry_mut().maybe_apply_mutation_internal(request_pb.mutations);
            RegistryAtomicMutateResponse {
                errors: vec![],
                version: registry().latest_version(),
            }
        }
        Err(error) => {
            println!(
                "{}Received a mutate call, but the request could not de
deserialized due to: {}",
                LOG_PREFIX, error
            );
            let mut response_pb = RegistryAtomicMutateResponse::default();
            let error_pb = RegistryError {
                code: Code::MalformedMessage as i32,
```

```
                reason: error.to_string(),
                ..Default::default()
            };
            response_pb.errors.push(error_pb);
            response_pb
        }
    };

    recertify_registry();

    let bytes =
serialize_atomic_mutate_response(response_pb).expect("Error serializing
response");
    reply(&bytes)
}
```

看一下反序列化函数入参是怎么做的，参数需要是 `Vec<u8>` 的类型，它能够解码成 RegistryAtomicMutateRequest 类型就行。然后序列号返回结果是把返回结果编码成 `Vec<u8>`。

```
/// Deserializes the arguments for a request to the atomic_mutate()
function in
/// the registry canister, from protobuf.
pub fn deserialize_atomic_mutate_request(
    request: Vec<u8>,
) -> Result<pb::v1::RegistryAtomicMutateRequest, Error> {
    match pb::v1::RegistryAtomicMutateRequest::decode(&request[..]) {
        Ok(request) => Ok(request),
        Err(error) => Err(Error::MalformedMessage(error.to_string())),
    }
}


/// Serializes a response for a atomic_mutate() request to the registry
/// canister.
//
// This uses the PB structs directly as this function is meant to
// be used in the registry canister only and thus there is no problem with
// leaking the PB structs to the rest of the code base.
pub fn serialize_atomic_mutate_response(
    response: pb::v1::RegistryAtomicMutateResponse,
) -> Result<Vec<u8>, Error> {
    let mut buf = Vec::new();
    match response.encode(&mut buf) {
        Ok(_) => Ok(buf),
        Err(error) => Err(Error::MalformedMessage(error.to_string())),
    }
}
```

这里能够直接 .encode() .decode() 就把这些工作完成了，是因为 RegistryAtomicMutateRequest 这个 struct 已经实现好 Message 这个 trait，里面包含 encode，decode 方法。

```
/// Message corresponding to a list of mutations to apply, atomically, to
the
/// registry canister. If any of the mutations fails, the whole operation
will fail.
#[derive(candid::CandidType, candid::Deserialize, Eq)]
#[derive(Clone, PartialEq, ::prost::Message)]
pub struct RegistryAtomicMutateRequest {
    /// The set of mutations to apply to the registry.
    #[prost(message, repeated, tag="1")]
    pub mutations: ::prost::alloc::vec::Vec<RegistryMutation>,
    /// Preconditions at the key level.
    #[prost(message, repeated, tag="5")]
    pub preconditions: ::prost::alloc::vec::Vec<Precondition>,
}
```

最后看一下 reply()，它调用系统接口 msg_reply_data_append() 和 msg_reply() 来返回数据。这样除了函数本身有一个返回值之外，还可以通过这种方式返回数据。

```
/// Replies with the given byte array.
/// Note, currently we do not support chunkwise assembling of the
response.
/// Warning if you use this with an endpoint it will cause a trap due to
the
/// message trying to return multiple responses
pub fn reply(payload: &[u8]) {
    unsafe {
        ic0::msg_reply_data_append(payload.as_ptr() as u32, payload.len()
as u32);
        ic0::msg_reply();
    }
}
```

bless_replica_version

```
#[export_name = "canister_update bless_replica_version"]
fn bless_replica_version() {
    check_caller_is_governance_and_log("bless_replica_version");
    over(candid_one, |payload: BlessReplicaVersionPayload| {
        bless_replica_version_(payload)
    });
}

#[candid_method(update, rename = "bless_replica_version")]
fn bless_replica_version_(payload: BlessReplicaVersionPayload) {
    registry_mut().do_bless_replica_version(payload);
    recertify_registry();
}
```

这个函数只允许 Governance Canister 来调用。

```
fn check_caller_is_governance_and_log(method_name: &str) {
    let caller = dfn_core::api::caller();
    println!("{}call: {} from: {}", LOG_PREFIX, method_name, caller);
    assert_eq!(
        caller,
        GOVERNANCE_CANISTER_ID.into(),
        "{}Principal: {} is not authorized to call this method: {}",
        LOG_PREFIX,
        caller,
        method_name
    );
}
```

```
/// Adds a new replica version to the registry and blesses it, i.e., adds
/// the version's ID to the list of blessed replica versions.
///
/// This method is called by the governance canister, after a proposal
/// for blessing a new replica version has been accepted.
pub fn do_bless_replica_version(&mut self, payload:
BlessReplicaVersionPayload) {
    println!("{}do_bless_replica_version: {:?}", LOG_PREFIX, payload);

    let version = self.latest_version();
    // Get the current list
    let blessed_key = make_blessed_replica_version_key();
    let before_append = match self.get(blessed_key.as_bytes(), version) {
        Some(old_blessed_replica_version) => {
            decode_registry_value::<BlessedReplicaVersions>(
                old_blessed_replica_version.value.clone(),
            )
            .blessed_version_ids
        }
        None => vec![],
    };

    let after_append = {
        let mut copy = before_append.clone();
        copy.push(payload.replica_version_id.clone());
        copy
    };
    println!(
        "{}Blessed version before: {:?} and after: {:?}",
        LOG_PREFIX, before_append, after_append
    );

    let mutations = vec![
        // Register the new version (that is, insert the new
```

```
ReplicaVersionRecord)
        RegistryMutation {
            mutation_type: registry_mutation::Type::Insert as i32,
            key: make_replica_version_key(&payload.replica_version_id)
                .as_bytes()
                .to_vec(),
            value: encode_or_panic(&ReplicaVersionRecord {
                release_package_url: payload.release_package_url.clone(),
                release_package_sha256_hex:
payload.release_package_sha256_hex,
            }),
        },
        // Bless the new version (that is, update the list of blessed
versions)
        RegistryMutation {
            mutation_type: registry_mutation::Type::Upsert as i32,
            key: blessed_key.as_bytes().to_vec(),
            value: encode_or_panic(&BlessedReplicaVersions {
                blessed_version_ids: after_append,
            }),
        },
    ];

    // Check invariants before applying mutations
    self.maybe_apply_mutation_internal(mutations);
}
```

BlessReplicaVersionPayload

update_subnet_replica_version

```
#[export_name = "canister_update update_subnet_replica_version"]
fn update_subnet_replica_version() {
    check_caller_is_governance_and_log("update_subnet_replica_version");
    over(candid_one, |payload: UpdateSubnetReplicaVersionPayload| {
        registry_mut().do_update_subnet_replica_version(payload);
        recertify_registry();
    });
}
```

UpdateSubnetPayload
pub subnet_id: PrincipalId
pub replica_version_id: String

```
pub fn do_update_subnet_replica_version(&mut self, payload:
UpdateSubnetReplicaVersionPayload) {
    println!(
```

```
        "{}do_update_subnet_replica_version: {:?}",
        LOG_PREFIX, payload
    );

    check_replica_version_is_blessed(self, &payload.replica_version_id);

    // Get the subnet record
    let subnet_key =
make_subnet_record_key(SubnetId::from(payload.subnet_id));
    let mutation = match self.get(subnet_key.as_bytes(),
self.latest_version()) {
        Some(RegistryValue {
            value: subnet_record_vec,
            version: _,
            deletion_marker: _,
        }) => {
            let mut subnet_record =
                decode_registry_value::<SubnetRecord>
(subnet_record_vec.clone());
            subnet_record.replica_version_id = payload.replica_version_id;
            RegistryMutation {
                mutation_type: registry_mutation::Type::Update as i32,
                key: subnet_key.as_bytes().to_vec(),
                value: encode_or_panic(&subnet_record),
            }
        }
        None => panic!("Error while fetching the subnet record"),
    };

    let mutations = vec![mutation];

    // Check invariants before applying mutations
    self.maybe_apply_mutation_internal(mutations)
}
```

update_icp_xdr_conversion_rate

```
#[export_name = "canister_update update_icp_xdr_conversion_rate"]
fn update_icp_xdr_conversion_rate() {
    check_caller_is_governance_and_log("update_icp_xdr_conversion_rate");
    over(candid_one, |payload: UpdateIcpXdrConversionRatePayload| {
        registry_mut().do_update_icp_xdr_conversion_rate(payload);
        recertify_registry();
    });
}
```

UpdateIcpXdrConversionRatePayload

pub data_source: String
pub timestamp_seconds: u64
pub xdr_permyriad_per_icp: u64

```
pub fn do_update_icp_xdr_conversion_rate(
    &mut self,
    payload: UpdateIcpXdrConversionRatePayload,
) {
    println!(
        "{}do_update_icp_xdr_conversion_rate: {:?}",
        LOG_PREFIX, payload
    );

    // If there is no ICP/XDR conversion rate, we have to Insert new one
    let mutations = vec![upsert(
        make_icp_xdr_conversion_rate_record_key()
            .as_bytes()
            .to_vec(),
        encode_or_panic::<IcpXdrConversionRateRecord>(&payload.into()),
    )];

    // Check invariants before applying mutations
    self.maybe_apply_mutation_internal(mutations);
}
```

add_node

add_node_operator

create_subnet

add_nodes_to_subnet

delete_subnet

recover_subnet

remove_nodes_from_subnet

remove_nodes

remove_node_directly

update_node_operator_config

update_subnet

clear_provisional_whitelist

set_firewall_config

update_node_rewards_table

add_or_remove_data_centers

update_unassigned_nodes_config

get_node_providers_monthly_xdr_rewards

Reference

- https://smartcontracts.org/docs/interface-spec/index.html#certification
- https://smartcontracts.org/docs/interface-spec/index.html#canister-signatures