Roving Lily Technical Report

Bohan Cui
ID: 221220089
NJU, Department of computer science and technology
Nanjing, China
bohan.cui@outlook.com

I. 背景

此部分请见 Proposal Talk:
https://jackcui.notion.site/rovinglily

II. 基本目标

此部分请见 Proposal Talk:
https://jackcui.notion.site/rovinglily

III. 宏观设计

本项目采用基于 **Rust** 的 **Substrate** 开发套件。

A. 系统类型的论证

在这一项目的开发过程中，实际上有三种可行的技术路线。

- 基于公链系统上智能合约的实现：好处是设计简单易行，但是性能低下，链上资源的利用率低。
- 采用原生的 Extrinsic 进行增量式数据结构的管理：这是 **Substrate** 建议的方式。
- 直接开发 *Msg as Transaction* 的公链系统：直接将逻辑集成到确认逻辑中，可以支持永久删除，和更灵活的社区管理和安全机制。坏处是需要从底层重新设计区块链系统。

在技术上，**Substrate** 框架均支持上述三种开发路径。由于 **Substrate** 原生的 **pallet_contracts** 组件仅支持 **Ink!** 而不支持 **Solidity**，添加 EVM 的支持由第三方公司 Frontier 提供，这会导致项目结构太过复杂。而第三种路线则需要从底层共识机制等重新构建整个区块链系统，工程量过大。综上考虑，采用第二种技术路线。

B. 基本逻辑补充

Substrate 将区块链系统视为一个状态机，构建的系统主要有两个部分 Node 和 Runtime，Node 提供系统运行需要的支持，而 Runtime 则是状态机的状态转移逻辑。由于打破了一切依托货币转帐的限制，Substrate 的几乎可以实现任何状态转移（比如最基本的增量式数据库管理）。Runtime 部分采用模块化（称为 pallet）设计，框架提供了一部分常用的模块。

C. 模块设计

我们需要实现的一个核心模块是 bbs pallet，这个模块需要实现以下内容：

1) 一个用户管理系统，每个用户可以注册自己的昵称，头像，以及公钥（额外的，稍候详述）。
2) 一个帖子存储结构。**Substrate** 的增量式数据库管理并不支持无限长的向量类型，想要实现无限长的数据结构，只能使用 Map 维护的指针映射 + 手动索引。（此处我们将（发帖人，发贴的 Unix 时间）作为指针，也就严格限制了发帖子频率的上理论上限是 1 帖/ms（实际当然是远低于这个值））
3) 一个帖子索引数据结构。因为 Map 是不可遍历的，为了对帖子进行索引，我们维护两个索引，分别是发贴时间和最后回复时间。
4) 一个监控系统，用来监控有多少活跃用户。
5) 回复，点赞机制。此处我们选择限制回复

IV. 微观设计

下面我们来从代码层面进行详述。开发环境配置过程略，采用 **Substrate**(现名 **Polkadot-SDK**) 1.10.0, **Rust** 语言，构建工具 **cargo**.

A. pallet 结构体宏构建

注意其中的 # 是 **Rust** 的参数宏，而不是注释，注释是//

```
use frame_support::pallet_prelude::*;
use frame_system::pallet_prelude::*;
use sp_std::marker::PhantomData;
use frame_support::traits::GenesisBuild;
use log::warn;

#[pallet::pallet]
pub struct Pallet<T>(_);

#[pallet::config]
pub trait Config: frame_system::Config +
↪  pallet_timestamp::Config + TypeInfo {
    type RuntimeEvent: From<Event<Self>> +
    ↪  IsType<<Self as
    ↪  frame_system::Config>::RuntimeEvent>;
}
```

由于我们的模块中需要调用现有的时间戳模块，所以 Config 需要引入 **pallet_timestamp::Config**

B. 用户信息注册

下面是一个 Map 用于存储用户注册的昵称信息等。包括下面的 **UserSum** 用来记录注册总人数。**UserSystemItem** 是注册项结构体，**Users** 即为存储注册信息的 Map。

结构体中的 **avatar** 字段是 **Gravatar** 的 Hash 值，用来渲染头像。rsaPublicKey 则是公钥字段。（此处特别说明，该公钥是用来发私信的。虽然区块链上的账户（**Substrate** 中是 AccountID 类型）就是一个公钥，但由于 Rust 中禁用显式的类型转换，同时这个公钥包含了其他信息，不能直接用于公私钥加密，因此添加一个额外的字段，允许用户添加自己的公钥，用于 RSA 加密的私信功能。

```
#[pallet::storage]
#[pallet::getter(fn posts)]
pub type UserSum <T: Config> = StorageValue<_, u128>;
↪  // the sum of all user.

#[derive(Encode, Decode, Clone, PartialEq, Eq,
↪  RuntimeDebug, Default, MaxEncodedLen, TypeInfo)]
pub struct UserSystemItem<T: Config> {
    ///**lastChange**: the last time the user changed
    ↪  their information (to limit the frequency of
    ↪  changes)
    pub lastChange: T::Moment,
    /// **nickname**: a string with a maximum length
    ↪  of 64 characters
    pub nickname: BoundedVec<u8, ConstU32<64>>,
    /// **avatar**: hash string of the Gravatar
    ↪  avatar
    pub avatar: BoundedVec<u8, ConstU32<32>>,
    /// **rsaPublicKey**: the public key of the
    ↪  user's RSA encryption algorithm, use for
    ↪  private message
    pub rsaPublicKey: BoundedVec<u8, ConstU32<256>>,
    // TODO: Regulation
    // pub bannedUntilTime: T::Moment,
}

#[pallet::storage]
#[pallet::getter(fn users)]
pub type Users<T: Config> = StorageMap<_,
↪  Blake2_128Concat, T::AccountId,
↪  UserSystemItem<T>>;
```

C. 核心：帖子保存和索引

我采用 < **发帖时间，发贴人** > 作为帖子的指针。

后两个索引的一致性并不由其保证，需要外部的函数来维护保证。两个索引，分别是发贴时间和最后回复时间。

```
/// **PostByTimeAndSender**: A map that stores all
↪  posts.
#[pallet::storage]
#[pallet::getter(fn post_database)]
pub type PostsByPointers<T: Config> = StorageNMap<
    -,
    (
        NMapKey<Blake2_128Concat, T::Moment>,
        NMapKey<Blake2_128Concat, T::AccountId>,
    ),
    Post<T>,
>;

/// **PostByPostDate**: an index that stores all
↪  posts by date.
#[pallet::storage]
#[pallet::getter(fn posts_index_post_date)]
pub type PostsByPostDate<T: Config> = StorageMap<
```

```
    _,
    Blake2_128Concat,
    u64, // Date precise to day
    BoundedVec<(T::Moment, T::AccountId),
    ↪ ConstU32<10000>>, // List of pointers to Post
    ↪ objects
>;

/// **PostByLastReplyDate**: an index that stores all
↪ posts by the last reply date.
#[pallet::storage]
#[pallet::getter(fn posts_index_reply_date)]
pub type PostsByLastReplyDate<T: Config> =
↪ StorageMap<
    _,
    Blake2_128Concat,
    u64, // Date precise to day
    BoundedVec<(T::Moment, T::AccountId),
    ↪ ConstU32<10000>>, // List of pointers to Post
    ↪ objects
>;
```

D. 核心：新帖子的创建

下面是最核心的函数调用 **new_post** 该函数尝试发送一个帖子，如果指定了 reply 的对象，则作为回复发出，如果没有指定对象，则开启一个新的 thread。另外该函数需要更新监控系统的 Monitoring 和 MonitoredDay，这两个变量用于统计当日的发贴量（活跃量）。同时其中还需要维护两个索引的内容，后面分情况更新索引的代码过于冗长，此处就不完全展示，源代码请查看 pallets/bbs/src/lib.rs

```
pub fn new_post(
    origin: OriginFor<T>,
    content: BoundedVec<u8, ConstU32<2048>>,
    reply_to: Option<(T::Moment, T::AccountId)>,
) -> DispatchResult {
    let who = ensure_signed(origin)?;
    // check if the caller is registered
    ensure!(Users::<T>::contains_key(& who),
    ↪ Error::<T>::UnregisteredUser);
    // check the length
    warn!("content.len() = {}", "content.len()");

    ensure!(content.len() <= 2048,
    ↪ Error::<T>::ContentTooLarge);
    let now = pallet_timestamp::Pallet::<T>::get();
    let post_id = (now, who.clone());

    ↪ ensure!(!PostsByPointers::<T>::contains_key(&post_id),
    ↪ Error::<T>::PostTooFrequent);
    //warn!("post_id = {:?}", post_id);

    let new_post = Post {
        content: content,
        owner: who.clone(),
        replies: BoundedVec::default(),
        likes: 0,
        dislikes: 0,
        attention: 0,
        postedTime: now,
        lastReplyTime: now,
    };

    //warn!("new_post = {:?}", new_post);
    // check the pointer is unique
    PostsByPointers::<T>::insert(post_id.clone(),
    ↪ new_post);

    let post_date =
    ↪ TryInto::<u64>::try_into(now).ok().unwrap()/
    ↪ 86400000;
    PostsByPostDate::<T>::try_mutate(post_date,
    ↪ |list| {
        if list.is_none() {
            *list = Some(BoundedVec::default());
        }
        list.as_mut()
            .unwrap()
            .try_push(post_id.clone())
            .map_err(|_|
            ↪ Error::<T>::TooManyPostsOnThisDate)
    })?;

    //warn!("post_date = {}", post_date);

    // add 1 to the monitoring system
    let mut monitored_day = MonitoredDay::<T>::get();
    if monitored_day == Some(post_date)  {
        Monitoring::<T>::mutate(|n| {
            if let Some(n) = n {
                *n = n.saturating_add(1);
            } else {
                *n = Some(1);
```

```
            }
        });
    } else {
        Monitoring::<T>::put(1);
        MonitoredDay::<T>::put(post_date);
    }
    .....
```

E. 点赞系统

提供了三个态度 API，分别是 Like，Dislike 和 Attention。非常接近，下面只展示一个。

```
#[pallet::call_index(2)]
#[pallet::weight({0})]
/// **fn like_post**: Like a post.
pub fn like_post(origin: OriginFor<T>, post_id:
↪  (T::Moment, T::AccountId)) -> DispatchResult {
    let who = ensure_signed(origin)?;
    // check if the caller is registered
    ensure!(Users::<T>::contains_key(& who),
    ↪  Error::<T>::UnregisteredUser);
    // check if the post exists

    ↪  ensure!(PostsByPointers::<T>::contains_key(&post_id),
    ↪  Error::<T>::PostNotFound);
    // check if the post is not liked by the owner
    let post =
    ↪  PostsByPointers::<T>::get(&post_id).unwrap();
    ensure!(post.owner != who,
    ↪  Error::<T>::LikeOwnPost);
    // update the post
    let post_id_clone = post_id.clone();
    PostsByPointers::<T>::mutate(post_id_clone, |p| {
        if let Some(post) = p {
            post.likes =
            ↪  post.likes.saturating_add(1);
        }
    });
    Self::deposit_event(Event::PostLiked(who,
    ↪  post_id));
    Ok(())
}
```

F. 初始化：创世区块配置

系统中的值存储项需要在创世区块中初始化，如下。

```
#[pallet::genesis_config]
pub struct GenesisConfig<T: Config> {
    pub user_sum: Option<u128>,
    pub monitoring: Option<u128>,
    pub monitored_day: Option<u64>,
    pub _phantom: PhantomData<T>,
}


impl<T: Config> Default for GenesisConfig<T> {
    fn default() -> Self {
        Self {
            user_sum: Some(0),
            monitoring: Some(0),
            monitored_day: Some(0),
            _phantom: PhantomData,
        }
    }
}

#[pallet::genesis_build]
impl<T: Config> BuildGenesisConfig for
↪  GenesisConfig<T> {
    fn build(&self) {
        if let Some(user_sum) = self.user_sum {
            UserSum::<T>::put(user_sum);
        }
        if let Some(monitoring) = self.monitoring {
            Monitoring::<T>::put(monitoring);
        }
        if let Some(monitored_day) =
        ↪  self.monitored_day {
            MonitoredDay::<T>::put(monitored_day);
        }
    }
}
```

G. 注册新的模块

下面需要在 runtime 中注册我们新加入的模块。因为我们的模块没有需要动态配置的类型，因此 Config 的实现非常简单。

```
impl pallet_bbs::Config for Runtime {
    type RuntimeEvent = RuntimeEvent;
}
```

V. 前端实现

由于时间有限以及该框架一些奇奇怪怪的兼容问题，目前只用 Typescript 实现了一个静态的 UI。

源代码请见/ui 文件夹。

```
const { ApiPromise, WsProvider } =
↪  require('@polkadot/api');
const fs = require('fs');
const path = require('path');
const local_url = "ws://127.0.0.1:9944";
const main = async () => {
    const wsProvider = new WsProvider(local_url);
    const api = await ApiPromise.create({
        provider: wsProvider,
        types: {
            PostByPointer: {
                id: 'u32',
                content: 'Text',
                author: 'AccountId',
                timestamp: 'Moment'
            }
        },
    });
    await api.isReady;

    const now_time = await api.query.timestamp.now();
    console.log(`Current time: ${now_time}`);
    const now_date = Math.floor(now_time.toNumber() /
    ↪  86400000);
    console.log(`Current date: ${now_date}`);
    const postVec = await
    ↪  api.query.bbs.postsByPostDate(now_date);
    console.log(`PostVec: ${postVec}`);
    console.log(`PostVec type: ${typeof postVec}`);

    const posts = [];

    for (const postPointer of postVec.unwrap()) {
        const post = await
        ↪  api.query.bbs.postsByPointers(postPointer[0],
        ↪  postPointer[1]);                          }
        if (post.isSome) {
            console.log(`Post: ${post.unwrap()}`);
            posts.push(post.unwrap());
        }
    }

    // Create HTML content
    let htmlContent = `
    <html>
    <head>
    <title>Roving Lily 漂流百合 DEMO</title>
    ...
    <h1>Roving Lily 漂流百合 (DEMO)</h1>
    <h2>Where roving lily travels, where it take
    ↪  roots.</h2>
    <ul>`;

    for (const post of posts) {
        const owner = post.owner;
        const user = await
        ↪  api.query.bbs.users(owner);
        const nickname = user.isSome ? new
        ↪  TextDecoder().decode(user.unwrap().nickname)
        ↪  : 'Unknown';
        const content = new
        ↪  TextDecoder().decode(post.content);
        htmlContent += `<li><h3>${nickname}
        ↪  Says</h3>`;
        htmlContent += `<p>${content}</p>`;
        htmlContent += `<p>  ${post.likes}
        ↪  ${post.dislikes}
        ↪  ${post.attention}</p>`;
        htmlContent += '<hr></li>';
    }

    htmlContent += `
        </ul>
    </body>
    </html>`;

    // Write HTML content to file
    const filePath = path.join(__dirname,
    ↪  'posts.html');
    fs.writeFileSync(filePath, htmlContent, 'utf8');
    console.log(`Posts have been written to
    ↪  ${filePath}`);
```

VI. 运行

请看图 1-11 及其文字注释。

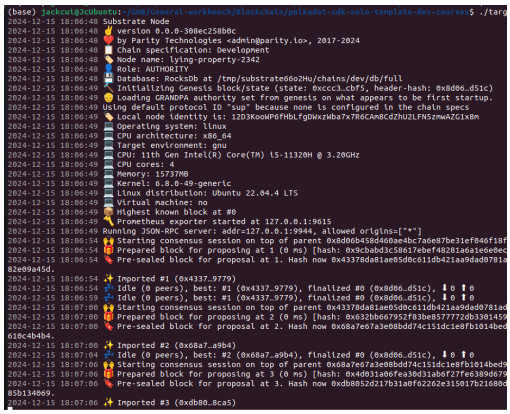

图 1. 我们经过 15 分钟的编译，开始运行我们的节点。可以看到下面已经开始生成一些区块。这是由于我们引入了 timestamp 模块，为了维护时间戳的正确性，每隔 6s 就会产生一个新区块。

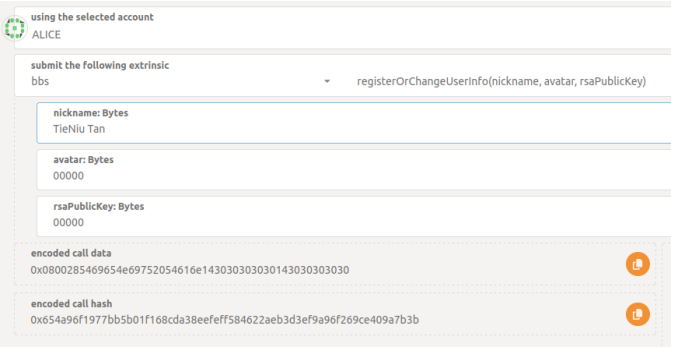

图 2. 我们使用 polkadot 提供的监视器来对链上状态进行控制。首先我们注册一个账户。用原生帐号 ALICE 注册昵称 “Tieniu Tan”

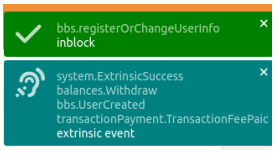

图 3. 可以看出已经正常出块，我们再类似注册几个昵称，注意我们其它参数给的是 0，因为我们还没有实现接入 Gravatar 头像。

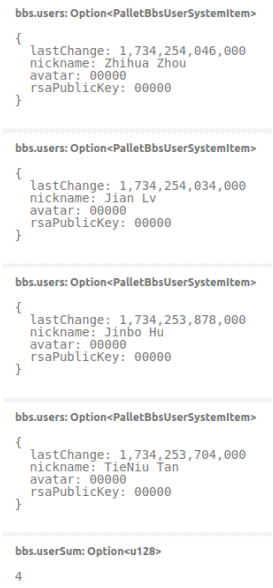

图 4. 用链状态的查看功能，我们可以看到都注册成功了

图 5. 我们用 Tantie Niu 发一个消息。

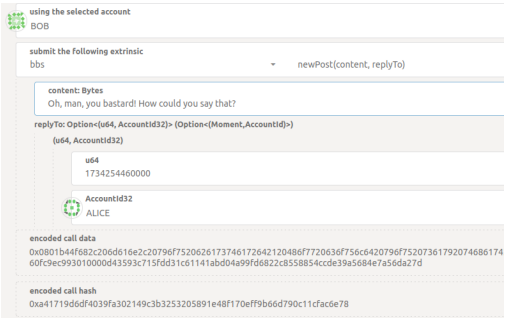

图 6. 我们使用 Jinbo Hu 回复这条消息，注意我们用 Pointer 的可选参数指定了要回复的帖子。

图 7. 我们检查检测系统和索引的链上状态发现都正常更新了。20072 是 Unix 历的日期数

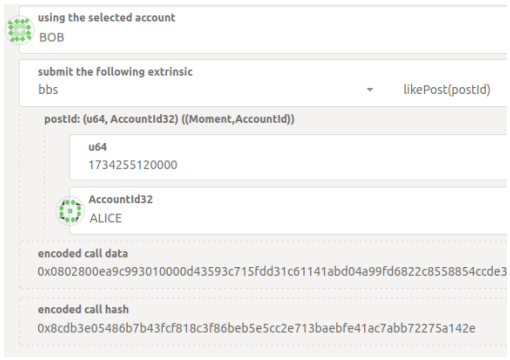

图 8. 我们再用 jimbo 给铁牛踏金波的帖子点个赞。

图 9. 我们再次检查第一条帖子的链上状态，发现已经更新了，replies 列表里面的一条指针正是 jimbo 给这条帖子的回复的指针。

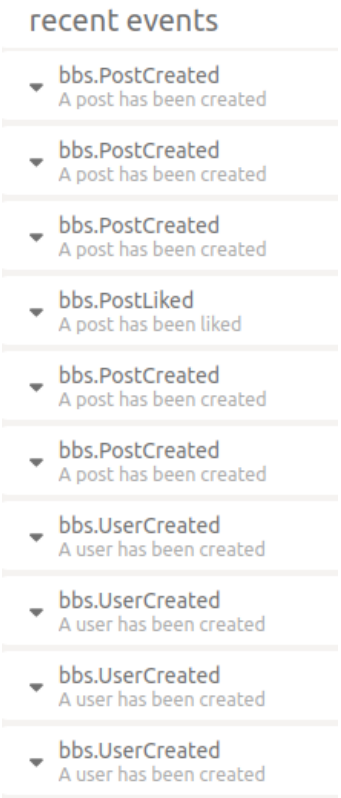

图 10. 再发几条，我们可以看到所有的事件都被正确释放了，这样我们就基本验证了我们的功能！（注意图 11 在后面）

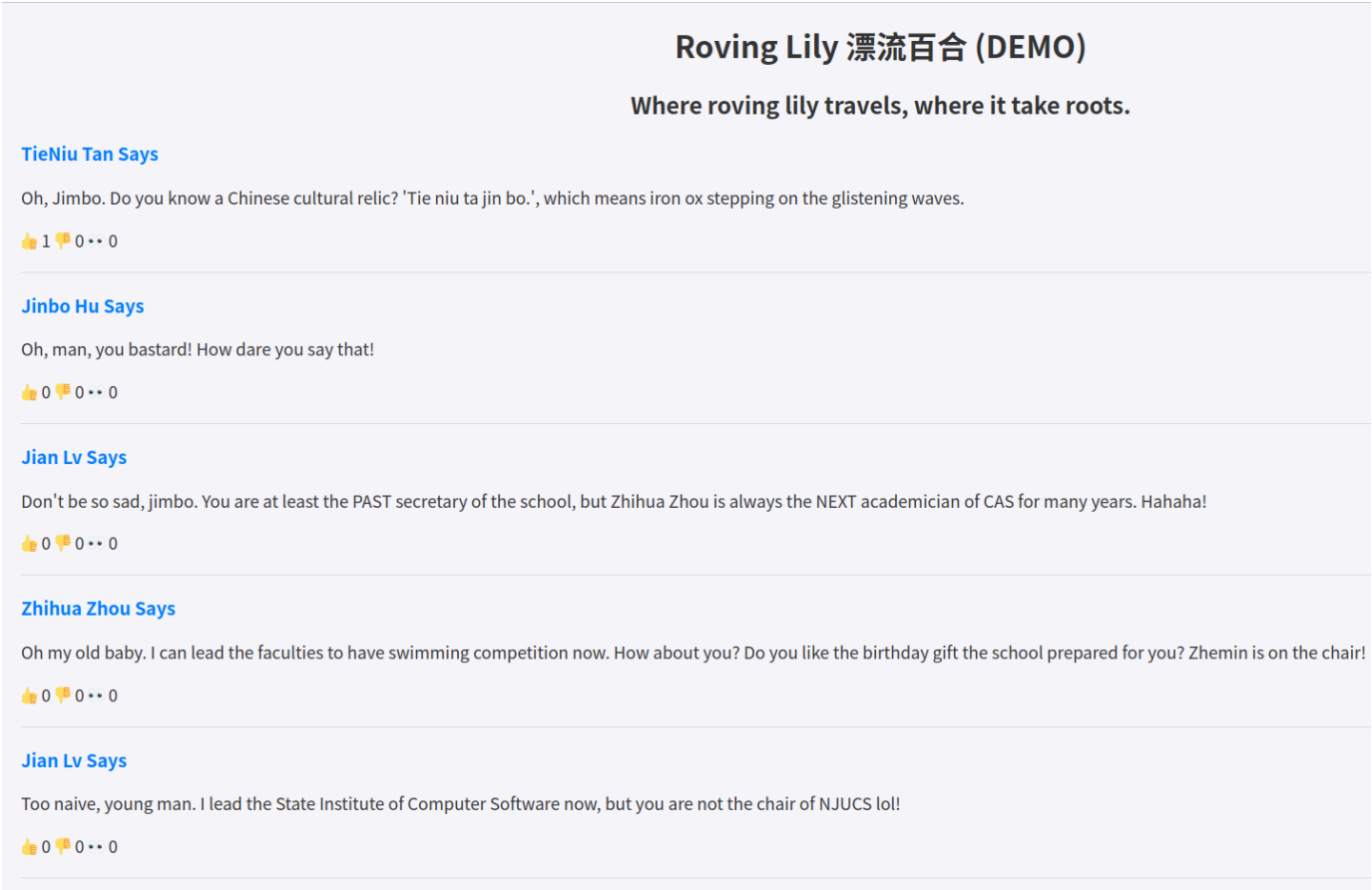

图 11. 这时候我们打开我们的 UI，可以看到完整版的对话了！细看有惊喜！

VII. Github 仓库

项目的 Github 仓库在这里。https://github.com/Jackcuii/RovingLily/

VIII. 未来工作

首先是 UI 需要改进一下，目前的还是太丑了。

其次是没有做安全相关的部分，基于低频硬分叉防止大量篡改和 DDoS 的设想需要从底层重构整个共识机制。短时间内还写不太出来。

另外是链上治理的部分，计划采用的折叠逻辑是，任何人都可以发出删帖动议，当删帖动议提出后，有 1/5 的活跃用户支持则删帖动议有效，但同时如果有超过 1/5 人反对则不会被删除。这样可以一定程度上保证垃圾信息会被清理，同时又不会出现多数人压制少数人的声音的民粹行为。但这种机制如果没有安全机制保证是没意义的。

最后是垃圾清理部分，因为帐号可以无限创建，应增加定时清理非活跃帐号的功能。

IX. 后记

由于 Substrate 框架的版本问题（所有的教程都是错的!），以及各种各样莫名其妙的工程问题，用了一个月的周末时间才做出一个雏形。再加上现学 js，可以说是时间非常赶。看来工程问题亦不好解决!

X. Acknowledgement

感谢黄老师一学期的教学和帮助!