基于Substrate智能合约设计与实现(pallet-talent-contract)

1. Intro

定义如下场景:

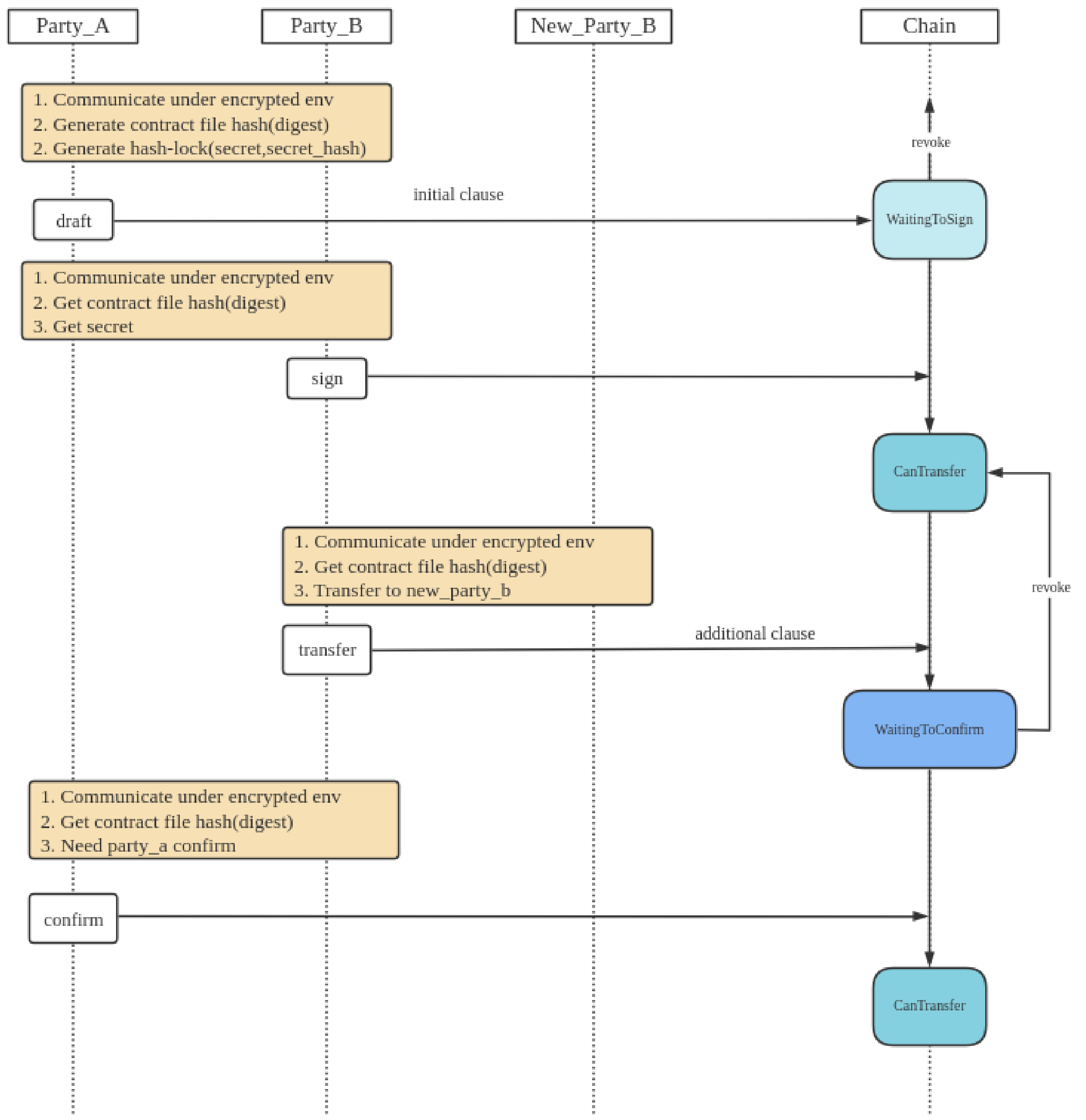
甲方(party_a)是签约人才,乙方(party_b)是签约公司,

新乙方(new_party_b)是合约转让中出现的角色.

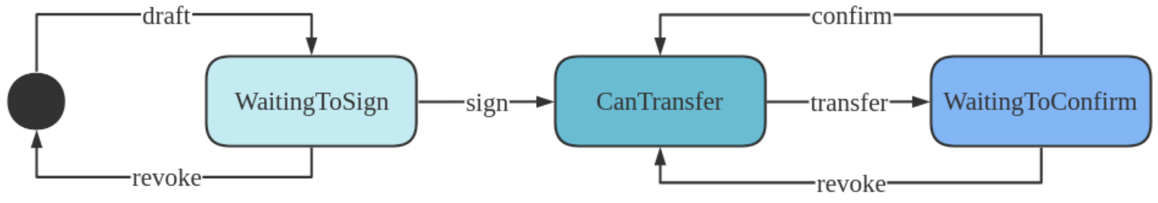
甲乙双方通过加密终端建立联系, 并在链上留下签约的关键信息.

链上信息是公开的,为了保证甲乙双方的签约过程不被第三方破坏且可以撤销,

我们使用了Hash时间锁技术.



Party_A
Party_B
New_Party_B
Chain
1. Communicate under encrypted env
2. Generate contract file hash(digest)
2. Generate hash-lock(secret,secret_hash)
revoke
initial clause
draft
WaitingToSign
1. Communicate under encrypted env
2. Get contract file hash(digest)
3. Get secret
sign
CanTransfer
1. Communicate under encrypted env
2. Get contract file hash(digest)
3. Transfer to new_party_b
revoke
additional clause
transfer
WaitingToConfirm
1. Communicate under encrypted env
2. Get contract file hash(digest)
3. Need party_a confirm
confirm
CanTransfer



2. 签约过程(合同起草人和合同签订人): draft, sign

(a) draft

甲乙双方任意一方都可在链上起草(draft)一份人才签约合同

前置条件:

- 合同起草人链下生成签约文件的32字节hash
- 合同起草人链下生成Hash锁(blake2_256 hash算法): (secret, secret_hash)

`draft(origin, digest, secret_hash, initial_clause, is_party_a)`

参数解析:

- origin: 合同起草人
- digest: 32字节文件Hash
- secret_hash: 合同起草人设置的hash锁
- initial_clause: 合同初始条款
- is_party_a: 合同起草人是甲方还是乙方.

链上合同状态: `WaitingToSign`

(b) sign

甲乙双方通过加密终端建立联系,合同起草人在加密环境下将secret传递给合同最后签订人

前置条件:

- 当前链上合同状态处于WaitingToSign
- 合同签订人获知secret

`sign(origin, digest, secret)`

参数解析:

- origin: 合同签订人
- digest: 32字节文件Hash
- secret: Hash锁原文

链上合同状态: `Cantransfer`

3. 合同转让过程(合同转让发起人, 合同转让确认人): transfer, confirm

链上合同的所有者属于甲乙双方,

甲乙双方都可以发起transfer, 来更改链上合同的乙方,然后由另一方确认并完成链上合同的转让.

甲方和新乙方可以签订新的附加条款.

(1) transfer

甲或乙方通过加密终端与新的乙方建立联系

前置条件:

- 链上合同处于CanTransfer状态

`transfer(origin, digest, new_party_b, additional_clause)`

参数解析:

- origin: 合同转让发起人
- digest: 32字节文件Hash
- new_party_b: 新的乙方
- additional_clause: 附加条款

链上合同状态: WaitingToConfirm

(2) confirm

链上合同的所有者属于甲乙双方,所以需要一方发起，另一方确认来完成链上合同的转让操作

前置条件:

- 链上合同处于WaitingToConfirm状态,且匹配请求的confirmer

`confirm(origin, digest)`

参数解析:

origin: 合同转让确认人

digest: 32字节文件Hash

4. 撤销签约过程或转让过程: revoke

有效期是1小时, 有效期内只能由链上合同的甲方或乙方撤销, 超过有效期可以由任意人撤销.

- draft之后链上合同的状态为 `WaitingToSign` , 撤销后, 删除该链上合同.
  链上合同状态: `无`
- transfer之后链上合同的状态为 `WaitingToConfirm` , 撤销后, 该链上合同的状态回退到 CanTransfer
  链上合同状态: `CanTransfer`

5. custom types

```
{
  "Address": "MultiAddress",
  "LookupSource": "MultiAddress",
  "BlockNumber": "u32",
  "Status": {
      "_enum": {
        "WaitingToSign": "H256",
        "WaitingToConfirm": "(AccountId, AccountId, Bytes)",
        "CanTransfer": null
      }
  },
  "Contract": {
      "digest": "H256",
      "status": "Status",
      "updated": "BlockNumber",
      "party_a": "Option<AccountId>",
      "party_b": "Option<AccountId>",
      "clause": "Bytes",
      "additional": "Bytes"
  }
}
```

6. source code

```
#![cfg_attr(not(feature = "std"), no_std)]

pub use pallet::*;

#[cfg(test)]
mod mock;
#[cfg(test)]
mod tests;

use codec::{Decode, Encode};
use frame_support::inherent::Vec;
use frame_support::pallet_prelude::*;
use sp_runtime::traits::{Hash, StaticLookup};
use sp_core::{H256, Bytes};
#[cfg(feature = "std")]
use serde::{Deserialize, Serialize};
```

```rust

#[derive(Clone, Eq, PartialEq, Encode, Decode)]
#[cfg_attr(feature = "std", derive(Debug, Serialize, Deserialize))]
#[cfg_attr(feature = "std", serde(rename_all = "camelCase"))]
pub enum Status<AccountId> {
    // (secret_hash)
    WaitingToSign(H256),
    // (confirmer, new_party_b, additional_clause)
    WaitingToConfirm(Option<AccountId>, AccountId, Bytes),
    CanTransfer,
}

#[derive(Clone, Eq, PartialEq, Encode, Decode)]
#[cfg_attr(feature = "std", derive(Debug, Serialize, Deserialize))]
#[cfg_attr(feature = "std", serde(rename_all = "camelCase"))]
pub struct Contract<AccountId, BlockNumber> {
    pub digest: H256,
    pub status: Status<AccountId>,
    pub updated: BlockNumber,
    pub party_a: Option<AccountId>,
    pub party_b: Option<AccountId>,
    pub clause: Bytes,
    pub additional: Bytes,
}

#[frame_support::pallet]
pub mod pallet {
    use super::*;
    use frame_support::dispatch::DispatchResult;
    use frame_system::pallet_prelude::*;

    /// Configure the pallet by specifying the parameters and types on which
it depends.
    #[pallet::config]
    pub trait Config: frame_system::Config {
        /// Because this pallet emits events, it depends on the runtime's
definition of an event.
        type Event: From<Event<Self>> + IsType<<Self as
frame_system::Config>::Event>;
        /// The period of validity about draft and transfer
        type Period: Get<<Self as frame_system::Config>::BlockNumber>;
    }

    #[pallet::pallet]
    #[pallet::generate_store(pub(super) trait Store)]
    pub struct Pallet<T>(_);

    #[pallet::storage]
    #[pallet::getter(fn contracts)]
    pub type Contracts<T: Config> =
    StorageMap<_, Blake2_128, H256, Contract<T::AccountId, T::BlockNumber>>;

    #[pallet::event]
    #[pallet::metadata(T::AccountId = "AccountId")]
    #[pallet::generate_deposit(pub(super) fn deposit_event)]
    pub enum Event<T: Config> {
        // (digest, drafter)
        Drafted(H256, T::AccountId),
        // (digest, signer)
        Signed(H256, T::AccountId),
        // (digest, new_party_b)
        Transferring(H256, T::AccountId),
        // (digest)
        Confirmed(H256),
        // (digest)
        Revoked(H256),
    }

    #[pallet::error]
    pub enum Error<T> {
        DigestIsExisted,
        DigestIsNotExisted,
        MismatchSecret,
        MismatchConfirmer,
        NotWaitingToSign,
        NotWaitingToConfirm,
        ExpiredToSign,
        ExpiredToConfirm,
        CannotTransfer,
        CannotRevoke,
        RequireTalentContractOwner,
    }

    #[pallet::hooks]
    impl<T: Config> Hooks<BlockNumberFor<T>> for Pallet<T> {}

    #[pallet::call]
    impl<T: Config> Pallet<T> {
        #[pallet::weight(1000)]
        pub fn draft(
            origin: OriginFor<T>,
            digest: H256,
            secret_hash: H256,
            clause: Bytes,
            is_party_a: bool
        ) -> DispatchResult {
            let drafter: T::AccountId = ensure_signed(origin)?;

            ensure!(!Contracts::<T>::contains_key(digest), Error::<T>::DigestIsExisted);

            Contracts::<T>::mutate(digest, |contract| {
                let party = if is_party_a {
                    (Some(drafter.clone()), None)
                } else {
                    (None, Some(drafter.clone()))
                };

                let new_contract = Contract{
                    digest: digest.clone(),
                    status: Status::WaitingToSign(secret_hash),
                    updated: <frame_system::Pallet<T>>::block_number(),
                    party_a: party.0,
                    party_b: party.1,
                    clause,
                    additional: Vec::new().into()
                };

                *contract = Some(new_contract);

                Self::deposit_event(Event::Drafted(digest, drafter));

                Ok(())
            })
        }

        #[pallet::weight(1000)]
        pub fn sign(
            origin: OriginFor<T>,
            digest: H256,
            secret: Vec<u8>,
        ) -> DispatchResult {
            let signer: T::AccountId = ensure_signed(origin)?;
            let secret_hash = T::Hashing::hash(&secret);

            Contracts::<T>::try_mutate_exists(digest.clone(), |contract| {
                let contract = contract
                    .as_mut()
                    .ok_or(Error::<T>::DigestIsNotExisted)?;

                match &contract.status {
                    Status::WaitingToSign(hash) => {
                        ensure!(
                            hash.as_ref() == secret_hash.as_ref(),
                            Error::<T>::MismatchSecret
                        )
                    },
                    _ => ensure!(false, Error::<T>::NotWaitingToSign),
                }

                let current = <frame_system::Pallet<T>>::block_number();
                ensure!(current <= contract.updated + T::Period::get(), Error::<T>::ExpiredToSign);

                contract.status = Status::CanTransfer;
                contract.updated = current;

                if contract.party_a.is_none() {
                    contract.party_a = Some(signer.clone());
                } else if contract.party_b.is_none() {
                    contract.party_b = Some(signer.clone());
                } else {
                    unreachable!("Never can reachable; qed");
                }

                Self::deposit_event(Event::Signed(digest, signer));

                Ok(())
            })
        }

        #[pallet::weight(1000)]
        pub fn transfer(
            origin: OriginFor<T>,
            digest: H256,
            new_party_b: <T::Lookup as StaticLookup>::Source,
            additional: Bytes,
        ) -> DispatchResult {
            let origin: T::AccountId = ensure_signed(origin)?;
            let new_party_b: T::AccountId = T::Lookup::lookup(new_party_b)?;

            Contracts::<T>::try_mutate_exists(digest.clone(), |contract| {
                let contract = contract
                    .as_mut()
                    .ok_or(Error::<T>::DigestIsNotExisted)?;

                let current = <frame_system::Pallet<T>>::block_number();

                match &contract.status {
                    Status::CanTransfer => {
                        ensure!(true, Error::<T>::CannotTransfer)
                    },
                    _ => ensure!(false, Error::<T>::CannotTransfer),
                }

                ensure!(
                    contract.party_a == Some(origin.clone())
                        || contract.party_b == Some(origin.clone()),
                    Error::<T>::RequireTalentContractOwner
                );

                let confirmer = if contract.party_a == Some(origin) {
                    contract.party_b.clone()
                } else {
                    contract.party_a.clone()
                };

                contract.status = Status::WaitingToConfirm(confirmer, new_party_b.clone(), additional);
                contract.updated = current;

                Self::deposit_event(Event::Transferring(digest, new_party_b));

                Ok(())
            })
        }

        #[pallet::weight(1000)]
        pub fn confirm(
            origin: OriginFor<T>,
            digest: H256,
        ) -> DispatchResult {
            let origin: T::AccountId = ensure_signed(origin)?;

            Contracts::<T>::try_mutate_exists(digest.clone(), |contract| {
                let contract = contract
                    .as_mut()
                    .ok_or(Error::<T>::DigestIsNotExisted)?;

                let current = <frame_system::Pallet<T>>::block_number();

                match &contract.status {
                    Status::WaitingToConfirm(Some(confirmer), new_party_b, additional) => {
                        ensure!(confirmer.clone() == origin, Error::<T>::MismatchConfirmer);
                        ensure!(current <= contract.updated + T::Period::get(), Error::<T>::ExpiredToConfirm);

                        contract.party_b = Some(new_party_b.clone());
                        contract.updated = current;
                        contract.additional = additional.clone();
                        contract.status = Status::CanTransfer;
                    },
                    _ => ensure!(false, Error::<T>::NotWaitingToConfirm),
                }

                Self::deposit_event(Event::Confirmed(digest));

                Ok(())
            })
        }

        #[pallet::weight(1000)]
        pub fn revoke(
            origin: OriginFor<T>,
            digest: H256,
        ) -> DispatchResult {
            let origin: T::AccountId = ensure_signed(origin)?;

            Contracts::<T>::try_mutate_exists(digest.clone(), |option_contract| {
                let contract = option_contract
                    .as_mut()
                    .ok_or(Error::<T>::DigestIsNotExisted)?;

                let current = <frame_system::Pallet<T>>::block_number();
                let is_expired = current <= contract.updated + T::Period::get();
                let is_owner = contract.party_a == Some(origin.clone())
                    || contract.party_b == Some(origin.clone());

                match &contract.status {
                    Status::WaitingToConfirm(_, _, _) if is_expired || is_owner => {
                        contract.status = Status::CanTransfer;
                        contract.updated = current;
                    },
                    Status::WaitingToSign(_) if is_expired || is_owner => {
                        *option_contract = None;
                    }
                    _ => ensure!(false, Error::<T>::CannotRevoke),
                }

                Self::deposit_event(Event::Revoked(digest));

                Ok(())
            })
        }
    }
}

impl<T: Config> Pallet<T> {
    pub fn get_contract(digest: H256) -> Option<Contract<T::AccountId, T::BlockNumber>> {
        Self::contracts(digest)
    }
}
```