Charge

The purpose of golos.charge smart contract

The `golos.charge` smart contract performs an applied function — it allows to limit user activity in creating posts, comments, and also in voting for posts. There is no direct user interaction with the `golos.charge` smart contact, recourse to actions occurs through `golos.publication` smart contract.

The operating principle of golos.charge smart contract

The mechanism regulating user activity in the system is implemented on the basis of batteries. The battery-based solution provides ability to keep track of both individual user operations and user's overall activity. The number of batteries allocated to a user can be one or more, each of which can be configured to account for one of the operations performed by this user for a certain time interval (for example, comments are recorded by one battery, sending votes in the form of «upvote» is another). When operation is executed, a part of the associated battery resource (charge) is consumed, taking into account the cost of this operation. Then this resource is restored in whole or in part. Amount of the restored resource is calculated according to the formula specified in the battery settings. Operation wil be blocked if the updated value of the spent resource exceeds the boundary value.

For each user operation being performed, a portion of the battery resource is consumed, taking into account the cost of this operation. When the battery reaches the limit value, the operation is blocked. Battery charge is fully restored after a time. The battery charge is fully restored after a time in accordance with selected algorithm, which is specified in the `setrestorer` operation.

Access to batteries is possible through call of special actions. The `golos.charge` smart contract implements the inline functions get_stored and get_current_value as well as the actions use, useandstore, removestored, setrestorer, usenotifygt and usenotifylt.

One of the required parameters used in these actions is `token_code`, which identifies a specific token. The account that calls any of these actions must have the token creater rights in

accordance with the authorization requirements.

Each battery is identified by the token code (`token_code`) and the battery identifier directly (`charge_id`).

Each type of token can be linked to more than one battery. The specific battery of a specific token is set by the `charge_id` parameter. A battery is a value that increases with the rise of it's use and decreases with it's expiry, i.e restoration.

---

use

The `use` action is used to control the battery charge (resources) when the user performs any kind of actions.
The action `use` has the form:

```
[[eosio::action]] void use(
    name user,
    symbol_code token_code,
    uint8_t charge_id,
    int64_t price,
    int64_t cutoff,
    int64_t vesting_price
);
```

Parameters:

- `user` — account name, used battery resource.
- `token_code` — the code of the token linked to the battery.
- `charge_id` — battery ID. This value, in conjunction with the `token_code` value, provides battery identification.
- `price` — amount of battery charge that the `user` consumes when the action is performed.
- `cutoff` — the threshold value of the battery charge, upon which the action is terminated.
- `vesting_price` — the amount of funds (in vesting) that the `user` must pay in case of insufficient battery charge to complete an operation. That is, if the battery charge exceeds the `cutoff` threshold before the operation completes. *(**Note:** this parameter is currently not used and should be equal to `0`).*

Any of the smart contracts can turn to the battery, but not directly, but through an action call. In case the battery charge does not meet the `cutoff` value conditions, the smart battery contract issues a corresponding message with a lock on the execution of the transaction, including the action that caused the access to the battery.

In order for the user to be able to pay for the missing battery charge (to complete the operation), it is necessary to provide the following:

- `golos.charge` must have access to user's staked vesting;
- `golos.charge` must have authority to «burn» the appropriate amount of the vesting;
- on user's balance must be the required amount of vesting in unlocked state.
  *(**Note:** this functionality is currently blocked)*

---

useandstore

*(**Note:** this action is currently disabled and has no effect)*
The `useandstore` action as well as `use` is used to account for battery charge when the user performs any actions. Unlike `use`, the `useandstore` action saves the values of the battery resources used in the tables inside the `golos.charge` smart contract and remains readable by other smart contracts.

The `useandstore` action has the following form:

```
[[eosio::action]] void useandstore(
    name user,
    symbol_code token_code,
    uint8_t charge_id,
    int64_t stamp_id,
    int64_t price
);
```

Parameters:

- `user` — account name, used battery resource.
- `token_code` — the code of the token linked to the battery.
- `charge_id` — battery ID. This value, in conjunction with the `token_code`, provides battery identification.

- `stamp_id` — identifier of the stored value of the battery `charge_id` used by the `user` account. The value is stored in the table of this battery reserved for the `user` account.
- `price` — the amount of funds (in conventional units of the battery), accrued to the `user` account for the used part of the battery resource.

---

removestored

*(**Note:** this action is currently disabled and has no effect)*
The `removestored` action is used to delete previously saved data on the use of battery resources in `golos.charge`. The key to finding deleted data is the set of values for the `charge_id`, `token_code`, and `stamp_id` parameters. The `removestored` action can be called from any smart contract.

The action `removestored` is:

```
[[eosio::action]] void removestored(
    name user,
    symbol_code token_code,
    uint8_t charge_id,
    int64_t stamp_id
);
```

Parameters:

- `user` — account name, used battery resource.
- `token_code` — the code of the token linked to the battery.
- `charge_id` — battery ID. This value, in conjunction with the `token_code` value, provides battery identification.
- `stamp_id` — the identifier of the value of the `charge_id` battery resource used by the `user` account, which is removed from the table of this battery.

---

setrestorer

The `setrestorer` action is used to set the function by which the battery charge is restored.
The `setrestorer` action operation has the form:

```
[[eosio::action]] void setrestorer(
    symbol_code token_code,
    uint8_t charge_id,
    std::string func_str,
    int64_t max_prev,
    int64_t max_vesting,
    int64_t max_elapsed
);
```

Parameters:

- `token_code` — token code, which is linked to the battery.
- `charge_id` — battery ID. This value, in conjunction with the `token_code` value, provides battery identification.
- `func_str` — a mathematical expression that defines the function of battery charge restoration.
- `max_prev` — the maximum value of the previous battery charge. This value is taken into account when restoring the charge.
- `max_vesting` — the maximum value of the vesting, which can be used by the function as an argument in the process of restoring battery charge.
- `max_elapsed` — the maximum period of time since the last call to the battery.

The `func_str` value parameter is calculated by a defined function with three variables, namely:

- `p` — sets the previous value of the battery charge.
- `v` — sets the value of the user's visitor. *(**Note:** this parametr is currently not used and should not be specified in the function).*
- `t` — sets the time period since the last setrestorer call.

For example, if a string is passed as the `func_str` parameter
`sqrt(v / 500000) × (t / 150)` ,
then the battery, identified by `token_code` and `charge_id` parameters, will be restored according to this function. If the user has an amount of funds equal to 500,000 (in vesting) during the period of 150 sec, the value of `func_str` will be the following:
`func_str = sqrt(500000 / 500000) × (150 / 150) = 1`
That means during 150 sec time the value of the battery will increase by one.

usenotifygt and usenotifylt

These actions are internal and unavailable to a user.

The `usenotifygt` name means «use battery resources and notify `publication` smart contract if battery consumption is greater than specified threshold value».
The `usenotifylt` name means «use battery resources and notify `publication` smart contract if the battery charge is less than specified threshold value».

Both operations are used to monitor users activity and notify the smart contract passed as a parameter (for example, `golos.publication`) about exceeding the permissible level of activity of these users. The `usenotifygt` action differs from `usenotifylt` by specified condition under which notification is sent.

The `usenotifygt` action has the form:

```
void usenotifygt(
    name user,
    symbol_code token_code,
    uint8_t charge_id,
    int64_t price_arg,
    int64_t id,
    name code,
    name action_name,
    int64_t cutoff
)
```

Parameters:

- `user` — battery owner account name.
- `token_code` — code of the token that is bound to the `charge_id` battery.
- `charge_id` — battery identifier. This value, together with `token_code`, uniquely identify a battery.
- `price_arg` — amount of battery charge that the `user` consumes when the action is performed.
- `id` — internal identifier of the contract `code`. This parameter is in notification along with the user name and the new battery value.
- `code` — contract code to which a notification is sent about the spent resources .
- `action_name` — action that issues the notification in the contract `code`.

- `cutoff` — threshold value of battery charge, upon reaching which a notification is sent about user's over-activity.

The action `usenotifylt` has a similar structure and parameters.

A user can publish several posts for a certain period and at the same time exceed the battery charge limit assigned to the user (user's over-activity). For posts published in the excess of battery charge, amount of reward will be reduced. In order for smart contract `golos.publication` had information about such posts, the actions `usenotifygt` and `usenotifylt` are used.

Before publishing a post, the smart contract `golos.publication` sends the action `usenotifygt` (or `usenotifylt`) to another smart contract `code` to use battery resources of `user` account by that smart contract. Smart contract `code` informs `golos.publication` about the current battery charge. After receiving the appropriate notification, the `golos.publication` smart contract logic calculates the post reward.

---

get_stored

The `get_stored` inline function is used to determine the battery charge of a user at a particular point in time.

The `get_stored` inline function has the form:

```
int64_t get_stored(
    name code,
    name user,
    symbol_code token_code,
    uint8_t charge_id,
    uint64_t stamp_id) {
    …
}
```

Parameters:

- `code` — account name hosting the charge contract.
- `user` — account name, the battery charge of which is about to be determined.

- `token_code` — code of the token, which is linked to the battery.
- `charge_id` — battery identifier. This value, together with the `token_code`, provides battery identification.
- `stamp_id` — identifier of the stored value of the `charge_id` battery resource used by the user account, which is required to be obtained from the battery table.

The value of battery charge is stored in the battery table when the `useandstore` is executed. Subsequently, to get the value from this table, call `get_stored`. Calling `get_stored` does not require additional authorization.

Data on stored values of battery resources are available for obtaining from any smart contract. The `get_stored` function does not process data, but only gets the saved values from the `golos.charge` smart contract tables.

The interaction of `golos.charge` with any of the smart contracts provides ability to keep track user's activities in these smart contracts. One battery can interact simultaneously with the actions of several smart contracts.

---

get_current_value

The inline function `get_current_value` determines the battery charge of a specific user. The `get_stored` inline function has the form:

```
int64_t get_current_value(
    name code,
    name user,
    symbol_code token_code,
    uint8_t charge_id = 0);
```

Parameters:

- `code` — account name hosting the charge contract.
- `user` — account name, the battery charge of which is about to be determined.
- `token_code` — code of the token, which is linked to the battery.
- `charge_id` — battery identifier. This value, together with the `token_code`, provides battery identification.