ViskaSAT

Tamego

第 1 章
アルゴリズム

1.1 基本設計

SAT ソルバのアルゴリズム部分だけを取り出してライブラリとして使えるようにする。

- **`Solver`** 様々なアルゴリズムのソルバの共通のインターフェース。このトレイトを持つものはロジックだけに集中し、Godot については何も触らない。
- **`SolverRunner`** `Solver` と Godot の橋渡し的役割をする。`Solver` を別スレッドで走らせ、ソルバの進捗を得たりソルバを制御したりと双方向のやりとりを可能にする。

可読性のためにソースを分割して記述する。

```
//| file: rust/viska-sat/src/lib.rs
```

第 2 章

Godot

2.1　エントリーポイント

godot-rust API の主要部分を読み込む。

```
//| id: l_godot-rust-api
use godot::prelude::*;
```

ファイル分割して記述したモジュールを読み込む。

```
//| id: l_modules
mod tests;
```

空の構造体 `ViskaSATExtension` を作って、GDExtension 用のエントリーポイントにする。Godot とやりとりをする部分だから `unsafe` になっている。

```
//| id: l_gdextension-entry-point
struct ViskaSATExtension;

#[gdextension]
unsafe impl ExtensionLibrary for ViskaSATExtension {}
```

```
//| file: rust/godot-rust/src/lib.rs
<<l_godot-rust-api>>
<<l_modules>>

<<l_gdextension-entry-point>>
```

第 3 章
テスト

3.1　テスト用モジュール

　様々な内容の動作確認のために書いたコードを雑多にまとめておく。

```
//| file: rust/godot-rust/src/tests.rs
pub mod thread_channel_communication;
```

3.2　スレッド間のチャネルの通信

　`Solver` と `SolverRunner` 間の通信の中核をなすチャネルについてテストしてみる。

　スレッドとチャネルのモジュールを読み込む。

```
//| id: tcc_modules
use std::thread;
use std::sync::mpsc;
use std::time::Duration;
```

3.2.1 `init`

フィールドの初期化をする。

```
//| id: tcc_init
fn init(base: Base<Control>) -> Self {
    Self {
        num_rx: None,
        ctrl_tx: None,
        is_pause: true,
        base
    }
}
```

3.2.2 `ready`

チャネルを立ててフィールドに代入したり、その他変数を用意する。

```
//| id: tcc_init_vars
let (num_tx, num_rx) = mpsc::channel::<u64>();
let (ctrl_tx, ctrl_rx) = mpsc::channel::<bool>();
self.num_rx = Some(num_rx);
self.ctrl_tx = Some(ctrl_tx);

let mut is_pause = self.is_pause;
```

そして、1 秒ごとに数字をカウントアップするスレッドを立てる。

```
//| id: tcc_thread
thread::spawn(move || {
    for val in 0..=100 {
        <<tcc_pause-handle>>
        num_tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
```

```
    }
});
```

ただし、一時停止のメッセージを受け取ったら再開のメッセージを受け取るまで停止する。

```
//| id: tcc_pause-handle
while let Ok(received) = ctrl_rx.try_recv() {
    is_pause = received;
}
while is_pause {
    let mut pause_flag = ctrl_rx.recv().unwrap();
    while let Ok(received) = ctrl_rx.try_recv() {
        pause_flag = received;
    }
    is_pause = pause_flag;
}
```

```
//| id: tcc_ready
fn ready(&mut self) {
    <<tcc_init_vars>>
    <<tcc_thread>>
}
```

3.2.3 `process`

そもそもチャネルが作られているか確認する。

```
//| id: tcc_check-channel
let (num_rx, ctrl_tx) = match (&self.num_rx, &self.ctrl_tx) {
    (Some(rx), Some(tx)) => (rx, tx),
    _ => return,
};
```

もしデータがあるなら受け取る。

```
//| id: tcc_receive
if let Ok(received) = num_rx.try_recv() {
    godot_print!("{}", received);
}
```

決定ボタンが押されたらカウントアップの一時停止・再開をする。

```
//| id: tcc_pause-stop
let input = Input::singleton();
if input.is_action_just_pressed("ui_accept") {
    self.is_pause = !self.is_pause;
    godot_print!("is_pause: {}", self.is_pause);
    ctrl_tx.send(self.is_pause).unwrap();
}
```

```
//| id: tcc_process
fn process(&mut self, _delta: f64) {
    <<tcc_check-channel>>
    <<tcc_receive>>
    <<tcc_pause-stop>>
}
```

```
//| file: rust/godot-rust/src/tests/
thread_channel_communication.rs
use godot::prelude::*;
use godot::classes::{Control, IControl};
use std::process;
<<tcc_modules>>

#[derive(GodotClass)]
#[class(base=Control)]
```

```
struct ThreadChannelCommunicatoin {
    num_rx: Option<mpsc::Receiver<u64>>,
    ctrl_tx: Option<mpsc::Sender<bool>>,
    is_pause: bool,
    base: Base<Control>
}

#[godot_api]
impl IControl for ThreadChannelCommunicatoin {
    <<tcc_init>>
    <<tcc_ready>>
    <<tcc_process>>
}
```