

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 の橋渡しの役割をする。

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
mod lit;
mod clause;
mod cnf;
mod assignment;
mod solver;
```

1.2 基本的な型

1.2.1 リテラル

リテラルとは原子論理式それ自体もしくはその否定のどちらかである。とくに命題論理において原子論理式は命題変数だから、リテラルとは命題変数それ自体かその否定のいずれかである。

`var` が命題変数を表し、`negated` が否定されているかどうかを表す。

```
/// file: rust/viska-sat/src/lit.rs
pub struct Lit {
    pub var_id: usize,
    pub negated: bool
}
```

1.2.2 節

リテラルを OR で繋いだ論理式を**節**という。`Lit` の配列として表現する。CDCL ソルバのために必要ならメタ情報を付けることを可能にした。

```
/// file: rust/viska-sat/src/clause.rs
use crate::lit::Lit;
pub struct Clause<Meta=()> {
    pub lits: Vec<Lit>,
    pub meta: Meta,
}
```

1.2.3 CNF (連言標準形)

節を AND で繋いだ構造をしている論理式を**CNF**という。`Clause` の配列として表現する。`num_vars` は変数の個数 (最大の ID + 1) を表す。

```
//| file: rust/viska-sat/src/cnf.rs
use crate::clause::Clause;
pub struct Cnf {
    pub clauses: Vec<Clause>,
    pub num_vars: usize
}
```

1.2.4 割り当て

各変数に真偽値を対応付ける構造を**割り当て**という。真・偽・未割り当ての3つの値を取り得るので、`Option<bool>` 型を取ることで対応する。その配列として割り当てを表現する。

```
//| file: rust/viska-sat/src/assignment.rs
pub struct Assignment {
    pub values: Vec<Option<bool>>
}
```

1.3 ソルバのトレイト

様々なアルゴリズムに共通するインターフェースを与えるために `Solver` トレイトを定める。ソルバ側の主な関数は以下の通り：

`Solver::initialize()` 問題 (`Cnf`) を引数に取って SAT ソルバを初期化する。

`Solver::solve()` 問題を解き、充足可能 (`SatResult::Sat`) か充足不能 (`SatResult::Unsat`) かを返す。

`Solver::model()` 充足可能のときの割り当ての1つを返す。充足不能なら何も返さない。型は `Option<Assignment>`。

また、`SolverRunner` との連携に関わる関数は以下の通り：

`Solver::emit_event()` イベントを発生させる。 `SolverRunner` にイベントを通知し、そのタイミングで一時停止する必要があるなら通知があるまで停止する。

```
/// id: sol_solver-trait
```

ここで、`SolverRunner` との通信のために `SolverCommunicator` トraitを定める。これは以下の関数を持つ：

`SolverCommunicator::send_event()` `SolverRunner` にソルバのイベントを伝える。

`SolverCommunicator::recv_latest_control()` `SolverRunner` からの**最新の制御のみ**を受けとる。スレッドをブロックする。

`SolverCommunicator::try_recv_latest_control()` `SolverRunner` からの**最新の制御のみ**を受けとる。スレッドをブロックしない。

```
/// file: rust/viska-sat/src/solver.rs
<<sol_solver-trait>>
```

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: grl_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: grl_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: grl_gdextension-entry-point  
struct ViskaSATExtension;  
  
#[gdextension]  
unsafe impl ExtensionLibrary for ViskaSATExtension {}
```

```
/// file: rust/godot-rust/src/lib.rs  
<<grl_godot-rust-api>>  
<<grl_modules>>  
  
<<grl_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

チャン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

そもそもチャン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>>  
}
```