シナリオテスター草案

シナリオについてはこちら, テストの流れについてはこちら, 主な機能についてはこちら, シナリオファイルの記述方法については
こちらを参照してください。

簡単な使い方

1. 当プラグインを導入する
2. テスト対象のプラグインの jar にシナリオファイルを埋め込み, ビルドする
3. テスト対象のプラグインを導入する
4. シナリオが発火し, テストが開始される(構成による <- 後述)
5. 開発者が, コンソールでテストの結果を確認する
6. 開発者が, テストの結果を修正する

概要

シナリオテストプラグイン（名前候補：Scenamatica, 以下 当プラグイン）は, PaperMC
プラグインのシナリオテストを自動化して高度な支援を提供します。
当プラグインを導入したサーバを用意し, テスト対象プラグインにシナリオファイルを置くことで,
自動化したテストを作成します。

シナリオファイルの配置場所どうしましょう..?

シナリオファイルは, 現時点で2つの配置場所を考えています：

1. plugins/Scenamatica/scenarios/<プラグイン名>/ に配置する
   この場合, テスト対象のプラグインにシナリオファイルを埋め込む必要がなく, jarの軽量化に繋がります。
   ですが, シナリオファイルの管理やアップデートがめんどうくさくなるでしょう。
2. テスト対象のプラグインの jar にシナリオファイルを埋め込む (ぺやんはコチラを推します)
   この場合, テスト対象のプラグインにシナリオファイルを埋め込む必要があり, jarが肥大化する必要があります(Profile
   で切り替えができると思います). メリットとしては, シナリオファイルがプラグインに同梱されるため,
   アップデートが簡単になります。
   また, プラグインと同じリポジトリで管理できるため,
   プラグインの開発者はシナリオファイルの管理を気にする必要がありません。

シナリオとは

シナリオとは, テスト対象のプラグインの機能をテストするための流れの集合です。

シナリオは, テスト対象のプラグインの機能を発火させるための動作 (Action),
その動作が正しく実行されたかを検証するための動作 (Assertion), およびそれらの動作を実行するための環境の定義 (Context)
からなります。

シナリオファイルの具体的な記述方法およびプロパティの一覧については, シナリオファイルの記述方法 を参照してください。

テストの流れ

シナリオテストは以下の流れで進行します。

1. テスト対象のプラグインを導入する
2. レビュワーがテスト開始をトリガする
3. 当プラグインが, シナリオに基づいてテスト対象のプラグインの特定の機能を発火させる
4. テスト対象のプラグインが, その機能を実行 (Execution) する（eg. プレイヤーにメッセージを送信, アイテムを与える,
   ブロックを設置する ...）
5. 当プラグインが, シナリオに基づいて, その機能が正しく実行されたかを検証 (Assertion) する
6. ステップ 5 で検証に失敗した場合, テストは失敗となりテスト対象のプラグインの開発者に通知される
7. 4-6 を繰り返す。
8. すべてのシナリオが成功した場合, テストは成功となりテスト対象のプラグインの開発者に通知される

テストが成功する条件

- すべての Assert が成功する

テストが失敗する条件

- 1つ以上のシナリオの Assertion が失敗する
- 現在待ち受けている動作よりも先に, 後述された動作が発生する
- シナリオの実行中にエラーが発生する
- シナリオの実行中にタイムアウトが発生する
- シナリオの実行中にプレイヤーがログアウトする

主な機能

疑似プレイヤ (PseudoPlayer) 機能

当プラグインは, テスト対象のプラグインの機能を発火させるために, 任意数の疑似プレイヤを生成します。

Bukkit の Player インターフェースを実装した疑似プレイヤは, 通常のプレイヤとして認識されます

mineflyer **との相違点**

疑似プレイヤ及び BOT を追加するツールとしてmineflyerが挙げられます。

ですが, 当プラグインは以下の点で mineflyer と異なります。

- 外部との通信を行わない
  PaperMC をインジェクションしサーバ内のみで完結するため, 外部とのパケットの送受信を行う必要がありません。

- 必要数のクライアントを準備しなくてよい
  mineflyerは, 人数分クライアントで準備する必要があります。
  当プラグインは, 上記の通り内部で完結するためその必要はありません。

動作実行 (Execution) 機能

疑似プレイヤ及び通常のプレイヤに対して, 任意の動作を行わせられます。

例えば, プレイヤに対して, メッセージを送信させたり, アイテムを使用させたり, ブロックを設置させたりできます。

対応する予定の動作については, こちらを参照してください。

動作検証 (Assertion) 機能

疑似プレイヤ及び通常のプレイヤに対して, 任意の動作が行われるのを待ち受けます。

タイムアウトを設け, その時間内に動作が行われなかった場合は, そのテストは失敗になります。

また, 後に定義された動作が先行して起きた場合にも, そのテストは失敗になります。(予定)

対応する予定の動作については, こちらを参照してください。

原状回復機能 (できたら)

テストが終了した際に, ワールドの状態や, エンティティの状態を元に戻します。
コスト無茶苦茶高いです。(ブロックやエンティティの状態を記録する必要があるので...)

シナリオファイルの記述方法

シナリオファイルは, YAML 形式で記述され(Java と迷ってます), 拡張子.yml または .yaml を持つ必要があります。

シンタックスシュガー重視で開発する予定です。

シナリオファイルの例

（未完成・各プロパティの説明は後述されたこちらを参照してください）

以下の例では，「プレイヤが死んだときに豪華にするプラグイン」のシナリオ記述例です。

テスト項目：

- 発火：PseudoPlayer1 が, PseudoPlayer2 によって殺される
- 検証：PseudoPlayer1 のスコア「kill」がインクリメントされる
- 検証：PseudoPlayer2 に個人メッセージ「PseudoPlayer1 によって殺されました！」が送信される
- 検証：サーバ全体にブロードキャストメッセージ「PseudoPlayer2 は PseudoPlayer1 によって殺されました！」が表示される
- 実行：PseudoPlayer2 がリスポーンする
- 検証：PseudoPlayer2 がスペクテイターモードになる

```
#  シナリオの名前
name: "プレイヤが死んだときの正常動作シナリオ"

#  シナリオが発火する条件
on:
#  コマンド実行。コマンドは正規表現で記述できるとよい
- type: command_dispatch
  with:
    command: ^/test$
  #  本シナリオを実行する前に必要なシナリオを実行する
  before:
  - $ref: killPlayer
#  プラグイン.jar　が変更されたとき
- type: plugin_modify
  before:
    - $ref: killPlayer
#  なんらかの手段で手動実行されたとき
- type: trigger_manual
  before:
    - $ref: killPlayer

#  使いまわしするスキーマを定義する(OpenAPI　のような感じ)
schemas:
  #  JsonSchema　の仕様的に，　schemas　内の入力補完は厳しい
  killPlayer:
     type: execute
     action: entity_kill
     with:
       damager: PseudoPlayer1
       damagee: PseudoPlayer2
  defaultSword:
    type: DIAMOND_SWORD
    amount: 1
    slot: HOTBAR_0
  defaultChestPlate:
    type: DIAMOND_CHESTPLATE
    amount: 1
    slot: ARMOR_CHEST
    metadata:
      name: "§b§lプレイヤーの胸当て"
      lores: ["§7プレイヤーが死んだときに，"§7自動的に装備される]"
      enchantments:
        PROTECTION_ENVIRONMENTAL: 4
        DURABILITY: 3

#  シナリオの実行に必要なプレイヤーや，　ワールドの状態を定義する
context:
  #  シナリオに実行な疑似プレイヤー
```

```
  pseudo_players:
  - name: PseudoPlayer1
    location:
      world: world
      x: 0
      y: 0
      z: 0
      yaw: 0
      pitch: 0
    uuid: 00000000-0000-0000-0000-000000000000
    inventory:
      contents:
      #  $ref  でスキーマを参照する
      - $ref: defaultSword
      - $ref: defaultChestPlate
  - name: PseudoPlayer2
    inventory:
      contents:
      - $ref: defaultSword
      - $ref: defaultChestPlate

#  シナリオを定義する
scenario:
#  スコアがインクリメントされるか検証する.  起きるまで待つ(要タイムアウト)
- type: expect
  action: score_change
  with:
    name: kill
    type: objective
    target: PseudoPlayer2
    action: increment
#  プレイヤーに個人メッセージが送信されるか検証する.
- type: expect
  action: message_private
  with:
    content: "PseudoPlayer1 によって殺されました！"
    recipient: PseudoPlayer2
#  サーバ全体にブロードキャストメッセージが表示されるか検証する.
- type: expect
  action: message_broadcast
  with:
    content: "PseudoPlayer2 は PseudoPlayer1 によって殺されました！"
#  プレイヤーをリスポーンさせる
- type: execute
  action: respawn
  with:
    target: PseudoPlayer2
#  プレイヤーがスペクテイターモードになるか検証する.
- type: expect
  action: gamemode_change
  with:
    mode: SPECTATOR
```

シナリオファイルのプロパティ

name: string - 必須

シナリオの人間が読みやすい名前です。

---

on: On[] - 必須

シナリオが発火する条件を定義します。
すべて省略する場合でも, 実行するには type: trigger_manual を指定する必要があります。

---

schemas: Schema[] - (省略可)

シナリオの中で使いまわすスキーマを定義します。
スキーマは, どこでも $ref で参照できます。

---

context: Context - (省略可)

シナリオを実行するために必要なプレイヤーや, ワールドの状態を定義します。

---

scenario: Scenario[] - 必須

シナリオの実行内容を定義します。

On: object

シナリオが発火する条件を定義するための構造体です。

type: string - 必須

シナリオが発火する条件の種類を指定します。対応する動作はシナリオの定義と同じですので, こちら を参照してください。

with: object - (省略可)

条件の詳細を指定します。動作によっては必須です。

before: Scenario[] - (省略可)

シナリオが発火する前に実行するシナリオを定義します。

Schema: object

シナリオの中で使いまわすスキーマを定義するための構造体です。基本的になんでも入れられます。

Context: object

シナリオを実行するために必要なプレイヤーや, ワールドの状態を定義するための構造体です。

pseudo_players: Player[] ¦ int - (省略可)

シナリオに実行な疑似プレイヤーの数または, それらの定義を指定します。

worlds: World[] ¦ string - (省略可)

シナリオに必要なワールドの定義を指定します。

Scenario: object

シナリオの実行内容を定義するための構造体です。

type: execute ¦ expect - 必須

シナリオの動作を指定します。

action: string - 必須

シナリオの動作の種類を指定します。対応する動作は, こちら を参照してください。

with: object - (省略可)

動作の詳細を指定します。動作によって異なり, 必須の場合もあります。

対応する動作の一覧

各動作のプロパティは, with に指定します。

event

Bukkit のイベントを発火させたり, 発生を検証したりします。
一応, すべてのイベントに対応するつもりです。

基本的には, これだけですべて検証できるはずですが, シンタックスシュガーとして, もっと詳細な動作を定義します。

event

イベントの完全修飾クラス名を指定します。 Bukkit 標準のイベントの場合は, パッケージ名は省略できます。

---

command_dispatch

コマンドを実行したり, 実行されたことを検証したりします。

command

実行するコマンドを**正規表現で**指定します。

sender

コマンドを実行するプレイヤーを指定します。省略した場合は, コンソールから実行されます。

---

message_broadcast

サーバ全体にブロードキャストメッセージを表示したり, 表示されたことを検証したりします。

content

表示されるメッセージを指定します。

---

message_private

プレイヤーに個人メッセージを送信したり, 送信されたことを検証したりします。

content

送信されるメッセージを指定します。

---

message_chat

プレイヤーにチャットメッセージを送信したり, 送信されたことを検証したりします。

content

送信されるメッセージを指定します。

(ry)