Flush Code Hack 仕様書

変更履歴

日付	対象箇所	変更内容
2024/05/08	全体	新規作成
2024/05/20	プレイヤープログラム開発手順	変更可能な関数を具体的に列挙
2024/05/20	プレイヤープログラム開発手順	ログの確認方法を追加
2024/05/20	プレイヤープログラム関数	前提情報の節を追加
2024/05/20	プレイヤープログラム関数	新規の変数および関数の定義の節を追加
2024/05/20	プレイヤープログラム関数	ドローフェーズの誤字を修正
2024/05/20	ユニットテスト	問い合わせに関する注記を追加
2024/05/20	提出	提出先の情報を修正
2024/05/20	提出	提出後の動きを修正
2024/05/30	プレイヤープログラム開発手順	ラスに変数や関数を追加する意義を追加
2024/05/30	プレイヤープログラム関数	クラスに変数や関数を追加する意義を追加
2024/05/30	プレイヤープログラム関数	参考リンク追加

目次

環境構築

環境情報

- Node.js v22.1.0

構築手順

Node.js をインストールしていない方は、事前に Node.js をインストールしてください。
https://nodejs.org/en/download

1. 開発キットの zip ファイルを解凍します。

2. 開発キットのルートディレクトリで、以下のコマンドを実行し、依存パッケージのインストールを行います。

```
npm i
```

3. 起動

```
npm run dev
```

ブラウザで http://localhost:3000 にアクセスし、正常に起動できているか確認します。
※http://localhost:3000/games にリダイレクトします。

ディレクトリ構造

```
├── data
│   ├── {n}.json // 試合情報
│   └── sequence.txt // 生成した試合の通し番号管理
├── docs // 資料
├── node_modules // 依存パッケージ
├── public // 静的ファイル
└── src
    ├── __test__
    │   ├── developer
    │   │   ├── jsdom // UI系テスト
    │   │   └── node // Node系テスト
    │   └── player // プレイヤーサービステスト
    ├── apis // query, mutaion
    ├── clients // APIクライアント
    ├── components // UIコンポーネント
    ├── constants // 定数
    ├── libs // ライブラリ
    ├── pages // UIページ
    ├── schema // スキーマ
    ├── services
    │   ├── admin.ts // 試合操作サービス
    │   ├── file.ts // ファイル関連サービス
    │   ├── game.ts // 試合実行サービス
    │   ├── players
    │   │   ├── demo-player-js
    │   │   │   └── index.js // Javascript版デモプレイヤー
    │   │   ├── demo-player-ts
    │   │   │   └── index.ts // Typescript版デモプレイヤー
    │   │   └── index.tsx // プレイヤー定義ファイル
    │   └── round.ts // ラウンド処理サービス
    ├── styles // UIスタイル
    ├── templates // UIテンプレート
    └── utils // Util関数
```

試合を行う

「新規作成」ボタンをクリックすると、試合作成のポップアップが表示されます。
プレイヤーを 4 人選択して、「作成」ボタンを押すと試合が作成されます。
※まずはデモプレイヤーを 4 人選択してください。

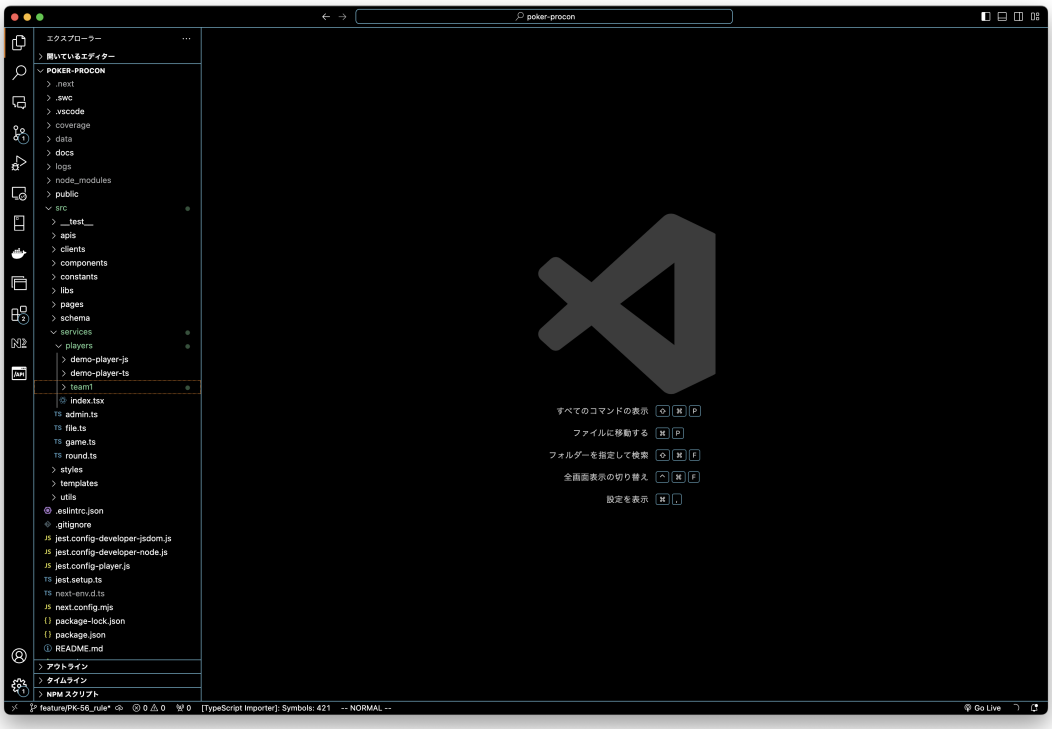
ポップアップが閉じ、試合のカードが追加されていることを確認してください。
「開始」ボタンを押すと、試合が始まります。
試合が終了すると自動的に、「開始」ボタンが「詳細」ボタンに変わります。

「詳細」ボタンをクリックすると、対戦ラウンドの一覧が表示されます。
※所持ポイントが参加フィーを下回った場合、試合から除外されます。参加可能数が 1 人になった時点で試合が終了するため、設定したラウンド数を完了しない場合があります。

表示されているラウンドを一つ選択すると、そのラウンドの盤面のログを見ることができます。

プレイヤープログラム開発手順

1. `src/services/players` にある、`demo-player-js` または `demo-player-ts` をコピーし、ディレクトリ名を **チーム名** に変更します。
    i. Javascript で開発する場合は、`demo-player-js` を、Typescript で開発する場合は、`demo-player-ts` を選択してください。



※他チームとモジュールが競合する場合、運営からバージョンの指定がはいる場合があります。

2. 必要な npm モジュールは追加していただいて構いません。必ず提出時にモジュールとバージョンを申告してください。

- npm モジュールとは
  npm モジュールは、Node.js のプロジェクトで使用されるパッケージのことです。これらのパッケージは、特定の機能を提供するコードの集まりで、プロジェクトに追加することで簡単にその機能を利用できるようになります。様々なパッケージが公開されており、必要に応じて追加しても問題ありません。
- バージョンとは
  バージョンは、npm モジュールの特定のリリースを識別する番号です。バージョン番号は、通常、セマンティックバージョニング（semver）という規則に従って付けられます。例えば、「1.0.0」のように、3 つの数字で構成され、それぞれがメジャー、マイナー、パッチバージョンを表します。
- 追加した npm モジュールとそのバージョンの確認方法

i. package.json ファイルを確認する: プロジェクトのルートディレクトリにある package.json ファイルには、インストールされているすべての npm モジュールとそのバージョンが記載されています。このファイルを開いて、dependencies や devDependencies セクションを確認してください。

```
{
  "dependencies": {
    "express": "^4.17.1",
    "lodash": "^4.17.21"
  },
  "devDependencies": {
    "jest": "^26.6.3"
  }
}
```

ii. npm コマンドを使用する: ターミナルやコマンドプロンプトで以下のコマンドを実行すると、インストールされているすべてのモジュールとそのバージョンを一覧表示できます。

```
npm list --depth=0
```

これにより、プロジェクトのルートディレクトリにインストールされているモジュールとそのバージョンが表示されます。

3. サンプルプレイヤープログラムの次の箇所はディーラープログラムから呼び出される関数になるため、**変更禁止**です。

i. start 関数
ii. bet 関数
iii. draw 関数
iv. end 関数

4. サンプルプレイヤープログラムのロジックを記述できる関数は以下です。引数の変更はできませんが、ロジックの内容については自由に書き換えてください。

    i. startRound 関数
    ii. decideBetPoint 関数
    iii. drawCard 関数
    iv. endRound 関数

    詳細は後述のプレイヤープログラム関数 を参照してください。

5. `src/services/players/index.tsx`を編集し、自身で作成したプログラムを import します。

```
import Team1 from './team1'; // ←追加 自身のチームの名称で定義します。

const Players: {
  [key: string]: any;
} = {
  DemoPlayer1: TsPlayer,
  DemoPlayer2: TsPlayer,
  DemoPlayer3: TsPlayer,
  DemoPlayer4: JsPlayer,

  Team1, // ←追加 importしたファイルを追加します。
};

export default Players;
```

6. 自身で開発したプレイヤープログラムを用いて、試合を行います。

7. ログを確認しながら設計通りの挙動をしているかを確認してください。ログは、ゲーム盤面の webUI またはログファイルから確認できます。

    ※1 ゲーム目の 1 ラウンド目の場合

    i. webUI: http://localhost:3000/1/1
    ii. ログファイル: `logs/1/game/1.log`

プレイヤープログラム関数

前提情報

関数には引数と戻り値があります。

- 引数: 関数に渡すデータで、関数名の右側のカッコ内に記述します。
- 戻り値: 関数が処理を終えた後に返す値で、return 文を使います。

クラス（メンバ）変数は追加、変更を行っても構いません。独自に変数や関数を追加する場合は、後述の新規の変数および関数の定義を参照してください。
ただし、ディーラから呼び出される関数は、変更しないでください。変更可能な関数は後述の変更可能な関数を参照してください。

クラスに変数や関数を追加する意義

- 状態の保持: 関数内で定義した変数は、関数が呼び出されるたびに初期化されてしまいます。そのため、ゲームを通して状態を維持したい値（当該プレイヤーのその時点までの勝ち数など）はクラス内で変数を定義することにより、ゲームの進行中ずっとその値を保持できます。
- 再利用性の向上: クラスに関数を追加することで、繰り返し何度も行う必要があるプレイヤーの動作を定義できます。例えば、プレイヤーが交換するカードを選ぶ動作を関数として定義することで、必要なタイミングでその関数を呼び出し、実行することができます。

新規の変数および関数の定義

変数を定義する場合

```
//  （中略）

class TsPlayer {
  private logger: winston.Logger | null | undefined; // player logger

  private id: string; // ゲームID

  private name: string; // プレイヤー名

  private round: number; // ラウンド

  private betUnit: number; // 賭けポイントを追加する単位

  private win: number; // 勝数

  private newVariable: string; // ←追加（適切な変数名、型を指定します。）

  //  （中略）
}
```

定義した変数を呼び出す時は、`this.newVariable`で呼び出しができます。

※上記は、Typescript のコードです。
※ Javascript で記載する場合は、`private` `public`等のアクセス修飾子や、`string` `number`等の型定義はありません。

関数を定義する場合

```
//  （中略）

class TsPlayer {
  //  （中略）

  private newFunction() {
    // 適切な関数名、引数を定義し、処理を記述します。
  }

  //  （中略）
}
```

新たに記述した関数を呼び出す時は、`this.newFunction()`で呼び出しができます。

※上記は、Typescript のコードです。
※ Javascript で記載する場合は、`private` `public`等のアクセス修飾子や、`string` `number`等の型定義はありません。

プログラム内で使用している型の定義は後述型定義を参照してください。
戻り値が必要となる関数については、別紙「戻り値サンプル（return-value-pattern.pdf）」を用意しているので参考にしてください。

参考

MDN Web Docs: クラス (日本語)
TypeScript Handbook: クラス (日本語)

変更可能な関数

開始フェーズ

各ラウンド開始時に呼び出される関数です。
ラウンド開始時に行いたい処理があれば、自由に記述してください。

引数

1. ゲーム情報（GameInfo）

戻り値

なし

対象関数

`startRound`

デモプレイヤープログラムでは以下の処理を行っています。
不要な処理を削除し、必要な処理があれば記述してください。

- ラウンドが開始したことをログに出力
- 各プレイヤーの情報をログに出力
- 当該ラウンドでベット・レイズを宣言する際に追加する賭けポイントを 200〜500 の間の値でランダムに設定する

```
/**
 * ラウンド開始時に行う処理
 * このプログラムではラウンド開始時にレイズ宣言時に追加するポイントを設定する
 * @param data
 * @returns
 */
private startRound(data: GameInfo): void {
  this.round = data.currentRound;
  this.logger?.info(this.formattedLog('Round start.'));

  // 各プレイヤーの情報をログに出力する
  Object.values(data.players).forEach((player) => {
    this.logger?.debug(
      this.formattedLog(
        `Round start. ${player.name} info. status: ${player.status},
point: ${player.point}`
      )
    );
  });

  this.betUnit = randomByNumber(300) + 200; // 1ターンごとに追加するポイント数
（このプログラムでは1ターンごとに追加するポイント数を規定しておく。値は200〜500までの間
のランダム値）
  this.logger?.debug(this.formattedLog(`bet unit: ${this.betUnit}.`));
}
```

※上記は Typescript 版のデモプレイヤープログラムです。

ベットフェーズ

ベットフェーズの自身のターンに呼び出される関数です。
賭けるポイントを指定するロジックを記述してください。

引数

1. ゲーム情報（GameInfo）

戻り値

number

数値によって宣言するコールが変わります。

戻り値	コール	備考

戻り値	コール	備考
0 未満	ドロップ	マイナスの値を指定します
0	チェック or コール	まだ誰もポイントを賭けていないときはチェック、それ以外はコール
0 ～ 所持ポイントまで	ベット or レイズ	まだ誰もポイントを賭けていないときはベット、それ以外はレイズ
所持ポイント以上	オール・イン	全ての所持ポイントを賭けます。場の最低賭けポイントは更新されません

対象関数

`decideBetPoint`

デモプレイヤープログラムでは以下の処理を行っています。
不要な処理を削除し、必要な処理があれば記述してください。

- 現在のラウンド情報をログに出力
- 各プレイヤーの情報をログに出力
- このプログラムでは最低賭けポイントが初期ポイントの 1/2 以上になった場合、ドロップを宣言する
- ドロップまたはオール・インを宣言ない場合に支払う必要のあるポイントを計算する
- レイズが宣言できるかをチェックする
- 宣言するアクションを決める
  - 1 回目のベットフェーズの場合、誰もポイントを賭けていなければベットを宣言する
  - 2 回目のベットフェーズの場合、最低賭けポイントが初期ポイントの 1/10 未満の時、レイズを宣言する
  - 上記に該当しない場合、1/1000 の確率でオール・インを宣言する
  - 上記に該当しない場合、コールを宣言する

```
/**
 * 場の最低賭けポイントに対して追加で賭けるポイントを決定する
 * @param data
 * @returns
 */
private decideBetPoint(data: GameInfo): number {
  this.logger?.info(
    this.formattedLog(
      `Phase ${data.phase}. pot: ${data.pot}, minBetPoint:
${data.minBetPoint}`
    )
  );
```

```
  // 各プレイヤーの情報をログに出力する
  Object.values(data.players).forEach((player) => {
    this.logger?.debug(
      this.formattedLog(
        `${player.name} info. point: ${player.point}, betPoint:
${player.round.betPoint}`
      )
    );
  });

  // ドロップ宣言をするかを決める（このプログラムでは最低賭けポイントが初期ポイントの半分を超えていたらドロップする）
  if (data.minBetPoint > data.initialPoint / 2) return -1;

  const self = data.players[this.name]; // 自身のデータ
  const diff = data.minBetPoint - (self?.round.betPoint ?? 0); // 現在の最低賭けポイントと既に賭けたポイントとの差額
  this.logger?.info(
    this.formattedLog(
      `my cards: ${JSON.stringify(self?.round.cards)}, diff: ${diff}`
    )
  );

  const point = self?.point ?? 0; // 所持ポイント
  const stack = point - diff; // 自由に使用できるポイント
  const canRaise = stack > 0; // 自由に使用できるポイントが1以上あればレイズが宣言できる

  if (canRaise) {
    // レイズが宣言できる場合
    if (data.phase === 'bet-1') {
      // 1回目のベットフェーズ
      // このプログラムでは1回目のベットフェーズで、誰も賭けていなければベットを行う
      if (!data.minBetPoint) return this.betUnit;
    } else if (data.phase === 'bet-2') {
      // 2回目のベットフェーズ
      // このプログラムでは2回目のベットフェーズで、初期ポイントの1/10以上の値が賭けられていなければレイズを宣言する
      if (data.minBetPoint < data.initialPoint / 10) return this.betUnit;
// stackがbetUnit賭けポイントを追加する単位より大きければレイズ、小さければオール・インとなる（このプログラムではレイズを宣言する時betPoint分のポイントを追加する）
    }
  }

  // レイズが宣言できない時 チェック/コール or オール・イン
  const declareAllIn = randomByNumber(1000) < 1; // オール・インを宣言するか（このプログラムでは1/1000の確率でオール・インを宣言する）
  return declareAllIn ? stack : 0; // オール・インまたはコール
}
```

※上記は Typescript 版のデモプレイヤープログラムです。

ドローフェーズ

ドローフェーズの自身のターンに呼び出される関数です。
交換するカード、キープするカードを判定するロジックを記述してください。
ディーラーから通知された手札の順番に従って、交換するカードは true, キープするカードは false となるように配列を生成してください。

引数

1. ゲーム情報（GameInfo）

戻り値

boolean[]

対象関数

`drawCard`

デモプレイヤープログラムでは以下の処理を行っています。
不要な処理を削除し、必要な処理があれば記述してください。

- 自身の手札の情報をログに出力
- カードを交換するかを決定する（このプログラムでは全てのカードに対し、1/2 の確率で交換を行う）

```
/**
 * 交換する手札を選択する
 * @param data
 * @returns
 */
private drawCard(data: GameInfo): boolean[] {
  const self = data.players[this.name]; // 自身のデータ
  const cards = self?.round.cards ?? [];
  this.logger?.info(
    this.formattedLog(
      `phase: ${data.phase}. my cards: ${JSON.stringify(cards)}`
    )
  );

  return [
    randomByNumber(2) < 1,
    randomByNumber(2) < 1,
    randomByNumber(2) < 1,
    randomByNumber(2) < 1,
    randomByNumber(2) < 1,
  ];
}
```

※上記は Typescript 版のデモプレイヤープログラムです。

終了フェーズ

各ラウンド終了時に呼び出される関数です。
ラウンド終了時に行いたい処理があれば、自由に記述してください。

引数

1. ゲーム情報（GameInfo）

戻り値

なし

対象関数

`endRound`

デモプレイヤープログラムでは以下の処理を行っています。
不要な処理を削除し、必要な処理があれば記述してください。

- 勝者の情報をログに出力する
- 各プレイヤーの情報をログに出力
- 勝者が自身の場合は、勝数を追加する

```
/**
 * ラウンド終了時に行う処理
 * @param data
 * @returns
 */
private endRound(data: GameInfo): void {
  this.logger?.info(
    this.formattedLog(
      `${data.currentRound}>: Round end. winner: ${data.winner}`
    )
  );

  // 各プレイヤーの情報をログに出力する
  Object.values(data.players).forEach((player) => {
    this.logger?.debug(
      `<Round: ${data.currentRound}>: Round end. ${
        player.name
      } info. status: ${player.status}, point: ${
        player.point
      }}, cards: ${JSON.stringify(player.round.cards)}, hand: ${
        player.round.hand
      }`
    );
  });

  if (data.winner === this.name) {
    this.win += 1;
```

```
    this.logger?.debug(this.formattedLog(`Win count: ${this.win}`));
  }
}
```

※上記は Typescript 版のデモプレイヤープログラムです。

型定義

Card

```
type Card = {
  number: CardNumber; // トランプの数字（1〜13の数値） ※カードの強さとは異なります
  suit: Suit; // トランプのスート（Spades, Hearts, Diamonds, Clubs）
};
```

GameInfo

```
type GameInfo = {
  totalRound: number; // 試合の総ラウンド数
  initialPoint: number; // 試合開始時のプレイヤー所持ポイント
  fee: number; // 参加フィー
  players: {
    [key: string]: {
      name: string; // プレイヤー名
      status: PlayerStatus; // プレイヤーステータス（active, out）
      point: number; // 現在の所持ポイント
      round: {
        betPoint: number; // 現在のラウンドで賭けた総ポイント（参加フィー除く）
        first: number; // 1回目のベットフェーズで賭けたポイント
        second: number; // 2回目のベットフェーズで賭けたポイント
        cards: CardSet<Card>; // 手札
        action: Action; // 現在のラウンドで最後に宣言したコール（check, bet,
call, raise, all-in, drop）
        hand?: Hand; // 手札の役 （Drop, HighCard, OnePair, TwoPair,
ThreeOfAKind, Straight, Flush, FullHouse, FourOfAKind, StraightFlush,
RoyalStraightFlush,）※2回目のベットフェーズが終了したあとに判定されます。
      };
    };
  };
  currentRound: number; // 現在のラウンド数
  phase: Phase; // 試合のフェーズ（start, join, bet-1, draw-1, bet-2, draw-
2, result, finished）
  order: string[]; // プレイ順 ※プレイヤー名の配列
  pot: number; // ポットの値
  minBetPoint: number; // 最低賭けポイントの値
  winner?: string; // 現在のラウンドの勝者のプレイヤー名
};
```

ユニットテスト

目的

自身で作成したプレイヤープログラムが正しく、ディーラープログラムから呼び出されるかを確認することを目的としています。
引数を自由に設定し、想定通りの戻り値を返しているかを確認することができます。
既存のテストコードでは、ディーラープログラムから呼び出され、戻り値を返す必要のある`bet`と`draw`のテストを行います。
既存のテストコードには変更を加えないでください。
テスト項目を追加することは可能です。自身で引数を自由に設定し、パターンごとのテストを行うことを推奨しています。

テストコードは`src/__test__`ディレクトリにファイルを配置します。
このシステムでは、ディーラープログラムや WebUI のテストも入っています。
プレイヤープログラムのテストは、`src/__test__/player/services/players`ディレクトリで管理されます。

手順

1. `src/__test__/player/services/players`にある、`demo-player-js.test.js`または`demo-player-ts.test.ts`をコピーします。
2. テスト対象のプログラムを自身で開発したプログラムに差し替えます。
    i. 最上位層の describe の直下にあるコードを変更します。

```
// Player1となっている箇所を src/services/players/index.tsxで定義した名称に変更します。
// const player = new Players.DemoPlayer1('1', 'Player1');
const player = new Players.Team1('1', 'Team1'); // 関数名と第二引数を変更します。
```

3. コンソールにて、以下のコマンドを実行します。

```
npm run test
```

4. 作成したプログラムのテストが全て通っているか確認します。
    i. 事前に準備しているテストケースに変更を加えないでください。
    ii. 自身でテストケースを追加しても構いません。

※技術的な不明点があれば、担当者にお問い合わせください。

提出

提出前に必ずユニットテストを行ってください。

提出先

Slack（各チームのチャンネル）
運営向けにメンションをつけて提出してください。

提出形式

作成したプレイヤープログラムのディレクトリを zip に圧縮してメッセージに添付

提出期限

6/13 17:00 厳守

メッセージフォーマット

```
【チーム名】


【プレイヤープログラムのファイル名】


【追加した外部ライブラリとバージョン】


【実装した戦略の概要】


```

サンプル

```
【チーム名】
Team1

【プレイヤープログラムのファイル名】
Team1

【追加した外部ライブラリとバージョン】
lodash: v4.17.15

【実装した戦略の概要】
{どのような戦略を採用したのか、採用した理由、期待している挙動などを記述します。}
```

提出後の注意事項
提出したコードが動作するかを確認します。動作しないことが確認された場合に運営から連絡を行います。
必ず **6/13 19:00 まで** は連絡を確認できるように待機していてください。