主専攻実験 K-4-2025 最終レポート

202310970 五十嵐尊人

目次

0. 要旨

本プロジェクトでは、リアルタイムで株式取引を模擬し、投資家のポートフォリオ分析を行うWebアプリケーションを開発した。システムは取引データの生成、株価の動的変動、ポートフォリオの管理、および統計分析機能を備えている。

バックエンドはJavaによるマイクロサービスアーキテクチャを採用し、取引生成、価格管理、データ分析の3つの独立したサービスで構成されている。フロントエンドはReact+TypeScriptで実装し、WebSocketによるリアルタイム通信を実現した。

特に動的な株価変動、価格保証システムの実装、地域別ポートフォリオ分析、性別・年代別統計などの機能を通じて、実際の株式市場に近い挙動を実現することができた。

1. はじめに

1.1 背景と動機

本レポートでは、主専攻実験の一環として、株式分析システムの実装を行った。この課題は標準課題であり、元々株取引に興味があったことと、課題1~5で学んだ内容を活かして実装を行うことができると考えたため、取り組むことにした。また、この課題は実装の拡張性を持ち、将来的な機能追加や改善が容易である点も魅力であった。

1.2 目的と課題設定

本プロジェクトの主な目的は以下の通りである：

1. **リアルタイムデータ処理技術の習得**：WebSocketやスライディングウィンドウなどを用いたリアルタイムデータ処理の実践
2. **マイクロサービスアーキテクチャの実装**：複数の独立したサービス間の連携による分散システム設計

3. **動的なデータ可視化の実現**：React+Chart.jsを用いた直感的なデータ表現
4. **現実的な市場シミュレーションの構築**：実際の株式市場に近い挙動を再現する仕組みの構築

課題設定としては、標準課題をベースとしながらも、特に以下の拡張・改善に焦点を当てた：

- 取引に応じた動的株価変動システムの構築
- 株取引、ポートフォリオの正しい株価参照の保証
- レスポンシブUIによるマルチデバイス対応

1.3 本レポートの構成

本レポートは以下の構成で、システムの設計から実装、評価まで詳細に説明する：

- 第2章では、システムの全体像と主要機能について概説する
- 第3章では、アーキテクチャとデータモデルの設計について詳細に説明する
- 第4章では、バックエンドとフロントエンドの具体的な実装について解説する
- 第5章では、技術的な工夫点と課題解決のアプローチについて述べる
- 第6章から11章では、UIの設計、開発課題、評価、関連技術、今後の展望とまとめを記す

2. システム概要

2.1 システムの全体像

本システムは「リアルタイム株式取引分析システム」として、株式取引データの生成からユーザーへの可視化までを一貫して行うWebアプリケーションである。システムは大きく分けて以下の3つのバックエンドコンポーネントと1つのフロントエンドで構成されている：

1. **取引生成サービス**（Transaction.java）：投資家による現実的な売買取引を自動生成する
2. **価格管理サービス**（PriceManager.java）：取引量に応じて株価を動的に変動させ、価格整合性を保証する
3. **分析処理サービス**（StockProcessor.java）：取引データを集計・分析し、ポートフォリオや統計情報を生成する
4. **Webフロントエンド**（React）：分析結果をリアルタイムで可視化し、ユーザーに提供する

これらのコンポーネント間はSocket通信とWebSocketを用いて連携し、リアルタイムなデータフローを実現している。

2.2 主要機能

本システムの主要機能は以下の通りである：

リアルタイム取引システム

- 保有株数に基づく確率的な売買生成
- 取引量に応じた動的な株価変動
- 空売り防止機能(未完成)

ポートフォリオ管理

- 株主別のポートフォリオ表示と評価損益計算

- 地域別（日本株・米国株・欧州株）ポートフォリオ分析
- リアルタイムな評価額・損益率の算出

統計分析・可視化

- スライディングウィンドウによる直近取引履歴の表示
- 性別統計（男女別の投資額・損益）
- 年代別統計（20代〜70代以上の投資傾向）

レスポンシブUI

- 画面サイズに応じた最適なレイアウト（3列/2列表示）
- 直感的なグラフ表示（円グラフ、統計表）

3. システム設計

3.1 アーキテクチャ設計

3.1.1 マイクロサービス構成

本システムは、3つの独立したJavaサービスとReactフロントエンドからなるマイクロサービスアーキテクチャを採用した：

1. **Transaction.java** - 取引生成エンジン
    - 役割：株式取引データの自動生成
    - 入力：なし（自律的に生成）
    - 出力：Socket通信によるJSONフォーマット取引データ
    - 特徴：メタデータから読み込んだ株主と多数の銘柄を管理し、現実的な取引を生成
2. **PriceManager.java** - 株価管理システム
    - 役割：取引に応じた価格計算と価格保証
    - 入力：Transactionからの取引データ
    - 出力：価格情報を付与した取引データ
    - 特徴：取引と価格の整合性を保証し、一貫した価格でStockProcessorに送信
3. **StockProcessor.java** - 分析処理エンジン
    - 役割：データ集計・分析とWebSocket配信
    - 入力：PriceManagerからの価格保証済み取引データ
    - 出力：WebSocketによるJSON形式分析結果
    - 特徴：ポートフォリオ管理、統計計算、スライディングウィンドウ処理を実装
4. **Reactフロントエンド**
    - 役割：データ可視化と操作インターフェース提供
    - 入力：WebSocketからのJSONデータ
    - 出力：ブラウザ上の可視化UI
    - 特徴：TypeScriptによる型安全性、Chart.jsによるグラフ表示

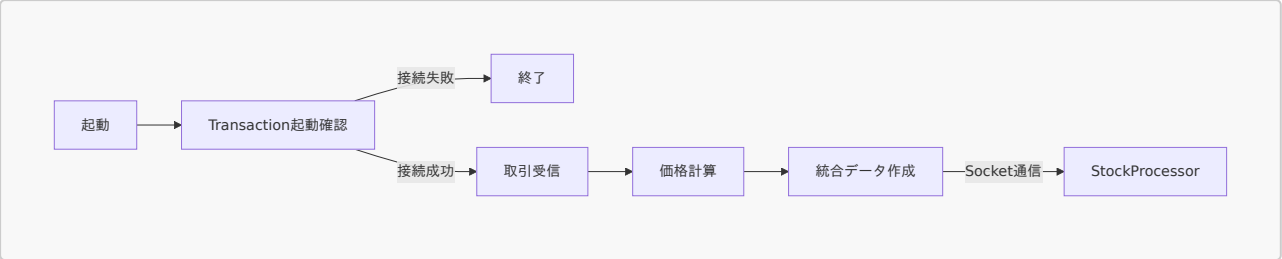
これらのサービスは独立して動作し、Socket通信・WebSocket通信によってデータを送受信する。

3.1.2 データフロー設計

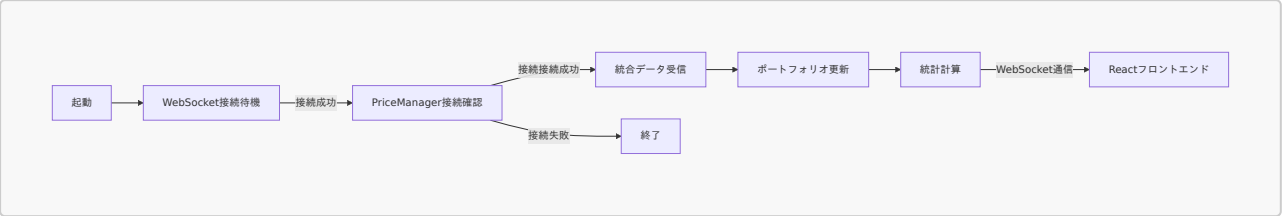
システム内のデータフローは以下の順序で進行する：

1. **取引生成フロー**(Transaction.java)

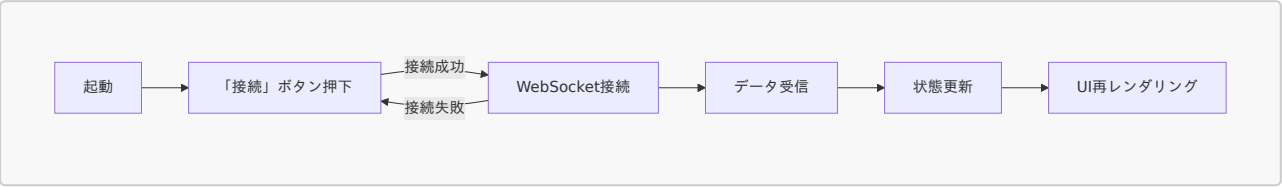
2. **価格計算フロー**(PriceManager.java)



3. **分析処理フロー**(StockProcessor.java)



4. **フロントエンド処理フロー**(Reactフロントエンド)



各サービス間のデータ形式は以下のように定義されている：

- Transaction → PriceManager:

```
{
  "shareholderId": 11,
  "stockId": 2,
  "quantity": 10,
  "timestamp": "12:34:56.78"
}
```

- PriceManager → StockProcessor:

```
{
  "transaction": {
    "shareholderId": 11,
    "stockId": 2,
    "quantity": 10,
    "timestamp": "12:34:56.78"
  },
  "currentPrice": 1250,
  "priceUpdateTimestamp": "12:34:56.789123"
}
```

- StockProcessor → Frontend:

```
{
  "type": "portfolio_summary",
  "shareholderId": 1001,
  "totalAsset": 1500000,
  "totalProfit": 50000,
  "profitRate": 0.034,
  "stocks": [...],
  "regionSummary": {
    "Japan": { "asset": 800000, "profit": 30000, "profitRate": 0.039,
"assetRatio": 0.533 },
    "US": { "asset": 500000, "profit": 15000, "profitRate": 0.031,
"assetRatio": 0.333 },
    "Europe": { "asset": 200000, "profit": 5000, "profitRate": 0.026,
"assetRatio": 0.133 }
  }
}
```

3.1.3 標準課題からの変更点と改善理由

標準課題から以下の主な変更点を導入した：

1. **アーキテクチャの変更**

   旧構成:

   ```
   StockPrice.java (独立)  ↘
                             StockProcessor.java
   Transaction.java (独立) ↗
   ```

   新構成:

   ```
   Transaction.java → PriceManager.java → StockProcessor.java
   ```

改善理由:

- 取引と株価の整合性保証
- 現実的な株価変動の実現（取引に応じた価格調整）
- データフローの一貫性向上

2. **価格変動ロジックの導入**

- 標準課題：ランダムな株価生成
- 改善版：取引量に応じた動的価格変動
- 理由：より現実的な市場動向の再現

3. **統計分析機能の拡充**

- 標準課題：基本的なポートフォリオ表示のみ
- 改善版：性別・年代別統計、地域別ポートフォリオ分析
- 理由：多角的な分析視点の提供

4. **レスポンシブUIの実装**

- 標準課題：固定レイアウト
- 改善版：画面サイズに応じた動的レイアウト
- 理由：マルチデバイス対応とユーザビリティ向上

3.2 データモデル設計

3.2.1 銘柄情報・株主情報

銘柄情報（StockInfo）

銘柄情報は`StockInfo`クラスで表現され、以下の主要属性を持つ：

```
public class StockInfo {
    private int stockId;                // 銘柄ID
    private String stockName;           // 銘柄名
    private String companyName;         // 会社名
    private CompanyType companyType;    // 企業種別（IT、製造業など）
    private MarketType marketType;      // 市場種別（日本、米国、欧州）
    private long basePrice;             // 基準価格
    // ...その他の属性とメソッド
}
```

市場種別は以下のEnumで定義されている：

```
public enum MarketType {
    JAPAN,    // 日本市場
    USA,      // 米国市場
    EUROPE,   // 欧州市場
```

```
    OTHER    // その他市場
}
```

この情報は地域別ポートフォリオ分析で重要な役割を果たす。

株主情報（ShareholderInfo）

株主情報はShareholderInfoクラスで表現され、以下の属性を持つ：

```
public class ShareholderInfo {
    private int shareholderId;          // 株主ID
    private String shareholderName;     // 株主名
    private Gender gender;              // 性別（MALE/FEMALE）
    private int age;                    // 年齢
    private String occupation;          // 職業
    private String address;             // 住所
    // ...その他の属性とメソッド
}
```

性別は以下のEnumで定義されている：

```
public enum Gender {
    MALE,    // 男性
    FEMALE   // 女性
}
```

これらの情報は、性別・年代別統計分析に活用されている。

3.2.2 取引データ構造

取引データはTransactionクラスとTransactionDataクラスで表現され、以下の構造を持つ：

```
public class TransactionData {
    private int shareholderId;      // 株主ID
    private int stockId;            // 銘柄ID
    private int quantity;           // 取引数量（正=買い、負=売り）
    private String timestamp;       // 取引時刻（HH:MM:SS.mmm形式）
    // ...getter/setterメソッド
}
```

また、PriceManagerにて価格情報が付与された状態は以下のクラスで表現される：

```
public class TransactionWithPrice {
    private TransactionData transaction; // 取引データ
    private int currentPrice;           // 取引時点の株価
```

```
    // ...getter/setterメソッド
}
```

StockProcessorが受け取る取引データはさらに拡張され、バッファに以下の情報が追加される：

```
Map<String, Object> tx = new HashMap<>();
tx.put("shareholderId", transaction.getShareholderId());
tx.put("shareholderName", shareholderName);
tx.put("stockId", transaction.getStockId());
tx.put("stockName", stockName);
tx.put("quantity", transaction.getQuantity());
tx.put("timestamp", transaction.getTimestamp());
tx.put("currentPrice", (double) guaranteedPrice); // 現在の価格
tx.put("previousPrice", (double) previousPrice); // 前回の価格
tx.put("acquisitionPrice", (double) guaranteedPrice); // 取得価格
```

これにより、取引データに関連するすべての情報が処理できるようになる。

3.2.3 ポートフォリオ管理構造

ポートフォリオはPortfolioクラスとPortfolio.Entryクラスによって表現されている：

```
public class Portfolio implements Serializable {
    private int shareholderId;                              // 株主ID
    private Map<Integer, Entry> holdings = new HashMap<>(); // 保有株式（銘柄
ID -> エントリ）

    // ...（省略）...

    public class Entry implements Serializable {
        private int stockId;                                // 銘柄ID
        private String stockName;                           // 銘柄名
        private int totalQuantity;                          // 合計保有数
        private List<Acquisition> acquisitions = new ArrayList<>(); // 取得
履歴

        // 平均取得単価の計算など
        public double getAverageCost() {
            // ...（計算ロジック）...
        }
    }
}
```

取得履歴はAcquisitionクラスで表現される：

```
public static class Acquisition implements Serializable {
    public double price;    // 取得価格
```

```
    public int quantity;    // 取得数量

    public Acquisition(double price, int quantity) {
        this.price = price;
        this.quantity = quantity;
    }
}
```

また、地域別の集計データは`RegionSummary`クラスで管理される：

```
private static class RegionSummary {
    int totalAsset = 0;     // 地域の総資産額
    int totalCost = 0;      // 地域の総コスト
    int totalProfit = 0;    // 地域の総利益

    void addStock(int asset, int cost, int profit) {
        this.totalAsset += asset;
        this.totalCost += cost;
        this.totalProfit += profit;
    }
}
```

これらの構造により、複雑なポートフォリオ管理と分析が可能になっている。

3.3 通信プロトコル設計

3.3.1 Socket通信（サービス間）

バックエンド内のサービス間通信には、TCP Socketを使用している。各サービスの通信設計は以下の通りである：

Transaction → PriceManager

- **プロトコル**: TCP Socket
- **ポート**: `Config.TRANSACTION_PORT` (設定ファイルで定義)
- **データ形式**: JSON形式のテキストデータ

- **通信フロー**:
    1. PriceManagerがTCP Serverとして起動し、指定ポートでListen
    2. TransactionがClientとしてPriceManagerに接続
    3. Transaction側から取引データを1行ずつJSON形式で送信
    4. PriceManagerが各取引データを受信・処理

PriceManager → StockProcessor

- **プロトコル**: TCP Socket
- **ポート**: `Config.PRICE_MANAGER_PORT` (設定ファイルで定義)
- **データ形式**: JSON形式のテキストデータ

- **通信フロー**:
    1. PriceManagerがTCP Serverとして起動し、指定ポートでListen
    2. StockProcessorがClientとしてPriceManagerに接続
    3. PriceManager側から価格付き取引データを1行ずつJSON形式で送信
    4. StockProcessorが各データを受信・処理

これらのSocket通信は`BufferedReader`と`InputStreamReader`を使用してデータ転送している。

3.3.2 WebSocket通信（クライアント-サーバー間）

StockProcessorとWebフロントエンド間の通信にはWebSocketを使用している：

- **プロトコル**: WebSocket (ws://)
- **ポート**: WebSocketサーバーが自動選択 (コンソールに表示)
- **データ形式**: JSON形式のテキストデータ
- **メッセージ構造**: 複数種類のメッセージタイプを定義
    - ポートフォリオ情報:

```
{
  "type": "portfolio_summary",
  "shareholderId": 1001,
  "totalAsset": 1500000,
  "totalProfit": 50000,
  "profitRate": 0.034,
  "stocks": [...],
  "regionSummary": {...}
}
```

    - 取引履歴:

```
{
  "type": "transaction_history",
  "windowStart": "12:34:56.00",
  "windowEnd": "12:35:01.00",
  "transactions": [...]
}
```

    - 性別統計:

```
{
  "type": "gender_stats",
  "male": {...},
```

```
    "female": {...}
  }
```

- 年代別統計:

```
{
  "type": "generation_stats",
  "generations": {
    "20s": {...},
    "30s": {...},
    // ...
  }
}
```

- **通信フロー**:

1. StockProcessorがWebSocketサーバーを起動
2. Reactフロントエンドが接続
3. 株主IDと株主名の対応関係を最初に送信(**ポートフォリオの株主選択用**)
4. その後、定期的に各種データをリアルタイム送信
5. フロントエンドは株主選択時にメッセージを送信
6. 選択された株主のデータを重点的に送信

WebSocket通信には自作の`WebsocketServer`クラスを使用し、JSON形式化には`Gson`ライブラリを活用している。送信前にはJSON構文チェックを実施し、無効なJSONの送信を防止している。

4. 実装詳細

4.1 バックエンド実装

4.1.1 Transaction.java - 取引生成エンジン

Transaction.javaは株式取引データを自動生成するコンポーネントである。主な機能と実装詳細は以下の通りである。

```
public class Transaction {
    // === インスタンス変数 ===
    private int shareholderId;     // 株主ID
    private int stockId;           // 銘柄ID
    private int quantity;          // 取引数量（正数=買い、負数=売り）
    private LocalTime timestamp;   // 取引発生時刻

    // === 静的フィールド（クラス全体で共有） ===

    // スケジューラー（定期的な取引生成制御）
    private static ScheduledExecutorService scheduler =
Executors.newSingleThreadScheduledExecutor();
    private static ScheduledFuture<?> updateTask;
```

```
    private static boolean isUpdateRunning = false;

    // 通信関連
    private static ServerSocket serverSocket; // フロントエンド接続用
    private static List<PrintWriter> clientWriters = new
CopyOnWriteArrayList<>(); // フロントエンド送信用

    // PriceManager通信用
    private static Socket priceManagerSocket;
    private static PrintWriter priceManagerWriter;

    // 保有株数管理（重要な機能）
    private static ConcurrentHashMap<String, Integer>
shareholderStockHoldings = new ConcurrentHashMap<>();
}
```

取引生成ロジック

取引生成は以下の主要メソッドで実装されている。

```
private static void updateTransactions() {
    LocalTime baseTime = LocalTime.now();
    List<Transaction> transactions = new ArrayList<>();

    for (int i = 0; i < Config.getCurrentTradesPerUpdate(); i++) {
        // 取引する株主と株IDをランダムで決定
        int shareholderId =
random.nextInt(Config.getCurrentShareholderCount()) + 1;
        int stockId = random.nextInt(Config.getCurrentStockCount()) + 1;

        // 保有株数に応じた取引量のランダム決定
        int quantity = generateSmartQuantity(shareholderId, stockId);

        // タイムスタンプ生成
        long nanoOffset = random.nextInt(Config.PRICE_UPDATE_INTERVAL_MS *
1_000_000);
        LocalTime timestamp = baseTime.plusNanos(nanoOffset);

        // 取引データをJSON形式化
        Transaction transaction = new Transaction(shareholderId, stockId,
quantity, timestamp);
        transactions.add(transaction);

        // StockProcessor接続後のみ保有株数を更新
        updateHoldings(shareholderId, stockId, quantity);

        // PriceManagerに送信
        sendToPriceManager(transaction);
    }

    transactions.sort((t1, t2) ->
```

```
t1.getTimestamp().compareTo(t2.getTimestamp()));

    // クライアント（フロントエンド）にも送信
    sendDataToClients(transactions);

    // リスナーに更新を通知
    for (StockTransactionUpdateListener listener : listeners) {
        listener.onTransactionUpdate(new ArrayList<>(transactions));
    }
}
```

売買数量決定ロジック

取引数量の決定は以下の方針で行われる。

```
/**
 * 保有株数を考慮したスマートな売買量生成（空売りなし版）
 */
private static int generateSmartQuantity(int shareholderId, int stockId) {
    String key = shareholderId + "-" + stockId;
    int currentHoldings = shareholderStockHoldings.getOrDefault(key, 0);

    // デバッグ用ログ（10秒に1回程度表示）
    if (System.currentTimeMillis() % 10000 < 100 && random.nextInt(100) <
1) {
        System.out.println("株主" + shareholderId + "の株" + stockId + "保有
数: " + currentHoldings + "株");
    }

    // 保有状況に応じた売買ロジック
    if (currentHoldings == 0) {
        // 保有なし → 買いのみ（初期投資）
        return generateBuyOnlyQuantity();

    } else if (currentHoldings <= 10) {
        // 少量保有 → 買い優勢（積み立て傾向）
        return generateBuyBiasedQuantity(currentHoldings);

    } else if (currentHoldings <= 50) {
        // 中量保有 → バランス良く（通常取引）
        return generateBalancedQuantityWithHoldings(currentHoldings);

    } else if (currentHoldings <= 100) {
        // 大量保有 → 売り優勢（利確傾向）
        return generateSellBiasedQuantity(currentHoldings);

    } else {
        // 超大量保有 → 強い売り傾向（リスク管理）
        return generateHeavySellQuantity(currentHoldings);
```

```
    }
}
```

4.1.2 PriceManager.java - 株価管理システム

PriceManager.javaは、取引に応じて株価を変動させ、価格保証を行うコンポーネントである。主な機能と実装詳細は以下の通り：

```
public class PriceManager {
    // Socket通信用
    private static ServerSocket priceManagerServerSocket;
    private static Socket transactionClientSocket;
    private static BufferedReader transactionReader;
    private static List<PrintWriter> stockProcessorWriters = new
CopyOnWriteArrayList<>();
}
```

価格変動ロジック

株価変動は取引量と方向（買い/売り）に応じて計算される：

```
private static void processTransactionAndUpdatePrice(Transaction
transaction) { // 取引データを受け取る
  int stockId = transaction.getStockId();
  StockPrice currentPrice = currentPrices.get(stockId);

  if (currentPrice == null) {
      System.err.println("Stock ID " + stockId + " not found in current
prices");
      return;
  }

  // 取引量に基づく価格変動計算(±1-5%程度)
  int quantity = transaction.getQuantity();
  double changeRate = calculatePriceChange(quantity, stockId);

  int basePrice = currentPrice.getPrice();
  int newPrice = (int) Math.max(50, basePrice * (1 + changeRate));

  // 新しい価格で更新（LocalTimeも正しく処理）
  StockPrice updatedPrice = new StockPrice(stockId, newPrice,
transaction.getTimestamp());
  currentPrices.put(stockId, updatedPrice);
}

/**
 * 取引量に基づく価格変動計算
 */
private static double calculatePriceChange(int quantity, int stockId) {
```

```
    // 買い注文（正の数量）は価格上昇、売り注文（負の数量）は価格下落
    double baseChange = quantity > 0 ? 0.01 : -0.01; // 1%の基本変動
    double volumeMultiplier = Math.min(Math.abs(quantity) / 100.0, 2.0); //
最大2倍
    return baseChange * volumeMultiplier * (0.5 + random.nextDouble()); //
ランダム要素
}
```

取引処理と価格保証

```
/**
* 取引データと価格データを統合してStockProcessorに送信
*/
private static void sendTransactionWithPriceToStockProcessors(Transaction
transaction) {
  int stockId = transaction.getStockId();
  StockPrice currentPrice = currentPrices.get(stockId);

  if (currentPrice == null) {
      System.err.println("株価が見つかりません: Stock ID " + stockId);
      return;
  }

  // StockProcessorが接続されているかチェック
  if (stockProcessorWriters.isEmpty()) {
      // 接続されていない場合は単純にスキップ（バッファリングしない）
      System.out.println("StockProcessor未接続 - 取引データをスキップ (株ID=" +
stockId +
              ", 数量=" + transaction.getQuantity() + ")");
      return;
  }

  // 統合データを作成
  TransactionWithPrice txWithPrice = new TransactionWithPrice(
          transaction,
          currentPrice.getPrice(),
          LocalTime.now());

  // カスタムGsonを使用してシリアライズ
  String json = gson.toJson(txWithPrice);

  // 全StockProcessorに送信
  List<PrintWriter> writersToRemove = new ArrayList<>();
  int successCount = 0;

  for (PrintWriter writer : stockProcessorWriters) {
      try {
          writer.println(json);
          writer.flush();
          successCount++;
```

```
            } catch (Exception e) {
                System.err.println("StockProcessorへの送信エラー: " +
e.getMessage());
                writersToRemove.add(writer);
            }
        }

        if (successCount > 0) {
            System.out.println("→ 取引データ送信: 株ID=" + stockId +
                    ", 株主ID=" + transaction.getShareholderId() +
                    ", 数量=" + transaction.getQuantity() +
                    ", 価格=" + currentPrice.getPrice() +
                    " (" + successCount + "接続)");
        }
    }
```

初期株価の設定

株メタデータから得た配当利回りと資本金の情報を元にある程度のランダム性をもたせて株の初期価格を決定するようにした。

```
    public static long calculateBasePrice(StockInfo stockInfo) {
        int dividendPerShare = stockInfo.getDividendPerShare();
        long capitalStock = stockInfo.getCapitalStock();
        StockInfo.CompanyType companyType = stockInfo.getCompanyType();

        // 1. 配当利回りベースの価格計算
        double dividendYield = getDividendYieldByCompanyType(companyType);
        double priceFromDividend = dividendPerShare / (dividendYield / 100.0);

        // 2. 資本金ベースの価格計算
        double marketCapMultiplier = getMarketCapMultiplier(companyType);
        long estimatedMarketCap = (long)(capitalStock * marketCapMultiplier);

        // 発行済み株式数を資本金から推定（1株あたり50000円と仮定）
        long estimatedShares = capitalStock / 50000;
        double priceFromMarketCap = (double)estimatedMarketCap /
    estimatedShares;

        // 3. 両方の価格を重み付き平均
        double basePrice = (priceFromDividend * 0.4) + (priceFromMarketCap *
    0.6);

        // 4. ランダム性を追加（±20%）
        double randomFactor = 0.8 + (random.nextDouble() * 0.4); // 0.8-1.2の範
    囲

        // 5. 最終価格（100円以上になるよう調整）
        double finalPrice = Math.max(100, basePrice * randomFactor);

        return Math.round(finalPrice);
```

```
    }

    private static double getDividendYieldByCompanyType(StockInfo.CompanyType
    type) {
        // 配当利回り（%）
        switch (type) {
            case LARGE:  return 1.5 + random.nextGaussian() * 0.5; // 1-2%程度
            case MEDIUM: return 2.5 + random.nextGaussian() * 0.7; // 1.8-3.2%
    程度
            case SMALL:  return 3.5 + random.nextGaussian() * 1.0; // 2.5-4.5%
    程度
            default:     return 2.5;
        }
    }

    private static double getMarketCapMultiplier(StockInfo.CompanyType type) {
        // 時価総額 = 資本金 × この倍率
        switch (type) {
            case LARGE:  return 3.0 + random.nextGaussian() * 1.0;  // 2-4倍
            case MEDIUM: return 2.0 + random.nextGaussian() * 0.7;  // 1.3-2.7
    倍
            case SMALL:  return 1.5 + random.nextGaussian() * 0.5;  // 1-2倍
            default:     return 2.0;
        }
    }
```

4.1.3 StockProcessor.java - 分析処理エンジン

StockProcessor.javaは取引データの分析、ポートフォリオ管理、統計計算を行い、WebSocketでクライアントに結果を送信する。主な機能と実装詳細は以下の通り：

```
    public class StockProcessor {
        // 統計情報用
        private static final ConcurrentHashMap<Integer, Integer> stockPriceMap
    = new ConcurrentHashMap<>();

        // メタデータ・管理構造
        private static final ConcurrentHashMap<Integer, StockInfo>
    StockMetadata = new ConcurrentHashMap<>();
        private static final ConcurrentHashMap<Integer, ShareholderInfo>
    ShareholderMetadata = new ConcurrentHashMap<>();
        private static final ConcurrentHashMap<Integer, Portfolio>
    PortfolioManager = new ConcurrentHashMap<>();

        // ウィンドウ管理用
        private static final int WINDOW_SIZE_SECONDS = 5;
        private static final int SLIDE_SIZE_SECONDS = 1;
        private static final Object windowLock = new Object();
        private static final Object bufferLock = new Object();
        private static final
    java.util.concurrent.CopyOnWriteArrayList<Map<String, Object>>
```

```
transactionBuffer =
        new java.util.concurrent.CopyOnWriteArrayList<>();
    private static final AtomicReference<LocalTime[]> windowRef = new
AtomicReference<>(null);

    // 選択中株主ID
    private static final AtomicReference<Integer> selectedShareholderId =
new AtomicReference<>(null);

    // WebSocketサーバーとJSONパーサー
    private static WebsocketServer wsServer;
    private static final Gson gson = new GsonBuilder()
        .registerTypeAdapter(LocalTime.class, new LocalTimeTypeAdapter())
        .create();
}
```

ポートフォリオ管理(実装)

```
private static void updatePortfolio(int shareholderId, int stockId, String
stockName,
                                    int quantity, int acquisitionPrice) {
    // 株主IDに対応するポートフォリオを取得または新規作成
    Portfolio portfolio = PortfolioManager.computeIfAbsent(
        shareholderId, k -> new Portfolio(shareholderId));

    // 取引を追加（保有株数と取得単価を更新）
    portfolio.addTransaction(stockId, stockName, quantity,
acquisitionPrice);
}
```

スライディングウィンドウ処理

```
private static void
processTransactionWithGuaranteedPrice(PriceManager.TransactionWithPrice
txWithPrice) {
    TransactionData transaction = txWithPrice.getTransaction();
    int guaranteedPrice = txWithPrice.getCurrentPrice();

    // ...（取引処理、バッファに追加など）...

    // ウィンドウ処理
    synchronized (windowLock) {
        LocalTime[] window = windowRef.get();
        if (window == null) {
            // 初回実行時にウィンドウを初期化
            LocalTime start = parseTimestamp(transaction.getTimestamp());
            LocalTime end = start.plusSeconds(WINDOW_SIZE_SECONDS);
            window = new LocalTime[] { start, end };
```

```
            windowRef.set(window);
        }
        window = windowRef.get();

        LocalTime transactionTime =
parseTimestamp(transaction.getTimestamp());

        if (transactionTime.isAfter(window[1])) {
            // ウィンドウ終了時刻を過ぎたら、ウィンドウをスライド
            LocalTime newStart = window[0].plusSeconds(SLIDE_SIZE_SECONDS);
            LocalTime newEnd = window[1].plusSeconds(SLIDE_SIZE_SECONDS);
            window[0] = newStart;
            window[1] = newEnd;
            windowRef.set(window);

            // 集計・送信・クリーンアップ
            aggregateAndSendWithCleanup(newStart);
        }
    }
}
```

統計計算 - 性別統計

```
private static void calculateAndSendGenderStats() {
    Map<String, Object> maleStats = new HashMap<>();
    Map<String, Object> femaleStats = new HashMap<>();

    // 初期化
    maleStats.put("investorCount", 0);
    femaleStats.put("investorCount", 0);
    // ... (他の初期値) ...

    int maleInvestorCount = 0;
    int femaleInvestorCount = 0;
    int maleTotalProfit = 0;
    int femaleTotalProfit = 0;
    int maleTotalCost = 0;
    int femaleTotalCost = 0;

    // ポートフォリオから統計計算
    for (Map.Entry<Integer, Portfolio> entry : PortfolioManager.entrySet())
{
        int shareholderId = entry.getKey();
        Portfolio portfolio = entry.getValue();

        if (portfolio.isEmpty()) continue;

        ShareholderInfo shareholder =
ShareholderMetadata.get(shareholderId);
        if (shareholder == null) continue;
```

```
        boolean isMale = (shareholder.getGender() ==
ShareholderInfo.Gender.MALE);

        int portfolioProfit = 0;
        int portfolioCost = 0;

        // ポートフォリオの評価損益計算
        for (Portfolio.Entry stockEntry : portfolio.getHoldings().values())
{
            int stockId = stockEntry.getStockId();
            int quantity = stockEntry.getTotalQuantity();
            int avgCost = (int) Math.round(stockEntry.getAverageCost());

            Integer currentPriceInt = stockPriceMap.get(stockId);
            int currentPrice = (currentPriceInt != null) ? currentPriceInt
: 0;

            int asset = quantity * currentPrice;
            int cost = quantity * avgCost;
            int profit = asset - cost;

            portfolioProfit += profit;
            portfolioCost += cost;
        }

        // 性別ごとに集計
        if (isMale) {
            maleInvestorCount++;
            maleTotalProfit += portfolioProfit;
            maleTotalCost += portfolioCost;
        } else {
            femaleInvestorCount++;
            femaleTotalProfit += portfolioProfit;
            femaleTotalCost += portfolioCost;
        }
    }

    // 結果を設定してJSON送信
    // ... (省略) ...
}
```

地域別ポートフォリオ分析

```
private static String getStockRegion(int stockId) {
    StockInfo stockInfo = StockMetadata.get(stockId);
    if (stockInfo != null && stockInfo.getMarketType() != null) {
        switch (stockInfo.getMarketType()) {
            case JAPAN:
                return "Japan";
            case USA:
                return "US";
```

```
                case EUROPE:
                    return "Europe";
                default:
                    return "Other";
            }
        }
        return "Unknown";
    }

    private static class RegionSummary {
        int totalAsset = 0;
        int totalCost = 0;
        int totalProfit = 0;

        void addStock(int asset, int cost, int profit) {
            this.totalAsset += asset;
            this.totalCost += cost;
            this.totalProfit += profit;
        }
    }

    public static String getPortfolioSummaryJson(int shareholderId) {
        Portfolio portfolio = PortfolioManager.get(shareholderId);
        if (portfolio == null || portfolio.isEmpty()) {
            // 空のポートフォリオを返す
            // ... (省略) ...
        }

        final int[] totals = new int[3]; // [totalAsset, totalCost,
totalProfit]
        List<Map<String, Object>> stockList = new ArrayList<>();

        // 地域別集計用
        Map<String, RegionSummary> regionMap = new HashMap<>();

        portfolio.getHoldings().entrySet().stream()
            .sorted(Map.Entry.comparingByKey())
            .forEach(entrySet -> {
                // ... (株ごとの計算) ...

                // 地域判定と地域別集計
                String region = getStockRegion(stockId);
                RegionSummary regionSummary = regionMap.computeIfAbsent(region,
k -> new RegionSummary());
                regionSummary.addStock(asset, cost, profit);
            });

        // 結果のJSONを作成して返す
        // ... (省略) ...
    }
```

4.1.4 WebSocketサーバー実装

WebSocketサーバーは独自のWebsocketServerクラスとして実装：

```
public class WebsocketServer {
    private static final int DEFAULT_PORT = 8887;
    private final int port;
    private final InetSocketAddress address;
    private WebSocketServer server;
    private final AtomicInteger connectionCount = new AtomicInteger(0);

    public WebsocketServer() {
        this(DEFAULT_PORT);
    }

    public WebsocketServer(int port) {
        this.port = port;
        this.address = new InetSocketAddress(port);

        server = new WebSocketServer(address) {
            @Override
            public void onOpen(WebSocket conn, ClientHandshake handshake) {
                connectionCount.incrementAndGet();
                System.out.println("New WebSocket connection: " +
conn.getRemoteSocketAddress());
            }

            @Override
            public void onClose(WebSocket conn, int code, String reason,
boolean remote) {
                connectionCount.decrementAndGet();
                System.out.println("WebSocket connection closed: " +
conn.getRemoteSocketAddress());
            }

            @Override
            public void onMessage(WebSocket conn, String message) {
                handleIncomingMessage(conn, message);
            }

            // ... (その他のメソッド実装) ...
        };
    }

    private void handleIncomingMessage(WebSocket conn, String message) {
        try {
            JsonElement jsonElement = JsonParser.parseString(message);
            if (!jsonElement.isJsonObject()) return;

            JsonObject json = jsonElement.getAsJsonObject();
            if (json.has("type")) {
                String type = json.get("type").getAsString();

                if ("selectShareholder".equals(type) &&
json.has("shareholderId")) {
```

```
                    int shareholderId =
json.get("shareholderId").getAsInt();
                    StockProcessor.setSelectedShareholderId(shareholderId);
                }
            }
        } catch (Exception e) {
            System.err.println("Error handling message: " +
e.getMessage());
        }
    }

    public void broadcast(String message) {
        if (server != null) {
            server.broadcast(message);
        }
    }

    // ... (その他のメソッド) ...
}
```

この実装により、WebSocketプロトコルを使用してブラウザクライアントとの双方向通信が可能になる。サーバーはJSON形式のメッセージを送受信し、クライアント選択に応じたデータ送信を実現する。

4.2 フロントエンド実装

4.2.1 React + TypeScript構成

フロントエンドはReact + TypeScriptで実装され、以下の基本構成を持つ：

```
// App.tsx - メインコンポーネント
import { useEffect, useRef, useState } from "react";
import { Col, Container, Row } from "react-bootstrap";
import ToggleButton from "react-bootstrap/esm/ToggleButton";
import "./App.css";
import GenderStatsSection from "./components/GenderStatsSection";
import GenerationStatsSection from "./components/GenerationStatsSection";
import PortfolioSection from "./components/PortfolioSection";
import TransactionHistorySection from
"./components/TransactionHistorySection";
import {
  type GenderStats,
  type GenerationStats,
  type PortfolioSummary,
  type ServerMessage,
  type ShareholderIdNameMap,
  type TransactionHistory
} from "./DataType";

function App() {

  const [is_trying_connect, setIsTryingConnect] = useState(false); // 接続状
```

```
況
  const [shareholderIdNameMap, setShareholderIdNameMap] =
useState<ShareholderIdNameMap>(); // ポートフォリオの株主選択用
  const [transactionHistory, setTransactionHistory] =
useState<TransactionHistory | null>(null); // 取引履歴
  const [portfolioSummary, setPortfolioSummary] = useState<PortfolioSummary
| null>(null); // ポートフォリオ
  const [genderStats, setGenderStats] = useState<GenderStats | null>(null);
// 性別統計
  const [generationStats, setGenerationStats] = useState<GenerationStats |
null>(null); // 年代別統計

  // ウィンドウサイズ管理
  const [windowWidth, setWindowWidth] = useState(window.innerWidth);

  const wsRef = useRef<WebSocket | null>(null);

  // ウィンドウサイズ監視
  useEffect(() => {
    const handleResize = () => {
      setWindowWidth(window.innerWidth);
    };

    window.addEventListener('resize', handleResize);
    return () => window.removeEventListener('resize', handleResize);
  }, []);

  // WebSocket接続処理
  useEffect(() => {
    let connection: WebSocket | null = null;

    const connectWebSocket = () => {
      connection = new WebSocket("ws://localhost:3000");
      wsRef.current = connection;

      connection.onopen = () => {
        console.log("WebSocket connected");
      };

      connection.onmessage = (event) => {
        try {
          const msg: ServerMessage = JSON.parse(event.data);
          setRawData(JSON.stringify(msg, null, 2));
          switch (msg.type) {
            case "portfolio_summary":
              setPortfolioSummary(msg); break;
            case "transaction_history":
              setTransactionHistory(msg); break;
            case "ShareholderIdNameMap":
              setShareholderIdNameMap(msg.ShareholderIdNameMap); break;
            case "gender_stats":
              setGenderStats(msg); break;
            case "generation_stats":
              setGenerationStats(msg); break;
```

```
            default:  break;
          }
          console.log("msg data:", msg);
        } catch {
          if (event.data) console.log("msg non-JSON data:", event.data);
        }
      };

      connection.onerror = (error) => {
        console.error("WebSocket error:", error);
        alert("WebSocket接続に失敗しました。");
        setIsTryingConnect(false);
      };

      connection.onclose = () => {
        console.log("WebSocket disconnected");
        setIsTryingConnect(false);
      };
    };

    if (is_trying_connect) {
      connectWebSocket();
    }

    return () => {
      if (connection) connection.close();
      wsRef.current = null;
    };
  }, [is_trying_connect]);

  // ブレークポイント定義
  const MOBILE_BREAKPOINT = 768;  // md未満: 1列表示
  const TABLET_BREAKPOINT = 992;  // lg未満: 2列表示


  // レイアウト判定
  const isMobile = windowWidth < MOBILE_BREAKPOINT;
  const isTablet = windowWidth >= MOBILE_BREAKPOINT && windowWidth <
TABLET_BREAKPOINT;


  return (
    <div className="App">
      <h1 id="title">課題 6</h1>
      <div id="connection-button-section">
        <ToggleButton
          id="toggle-connection"
          type="checkbox"
          variant="outline-primary"
          checked={is_trying_connect}
          value="1"
          onChange={(e)=>setIsConnected(e.currentTarget.checked)}
          >
        </ToggleButton>
```

```
      </div>

      <Container fluid className="p-3">
        {isMobile ? (
          // モバイル: 1列表示
          <div className="d-flex flex-column gap-3">
            <PortfolioSection
              shareholderIdNameMap={shareholderIdNameMap ?? {} as
ShareholderIdNameMap}
              ws={wsRef.current}
              portfolioSummary={portfolioSummary}
            />
            <TransactionHistorySection
              transactionHistory={transactionHistory}
              isTryingConnect={is_trying_connect}
              setIsTryingConnect={setIsTryingConnect}
            />
            <GenderStatsSection genderStats={genderStats} />
            <GenerationStatsSection generationStats={generationStats} />
          </div>
        ) : isTablet ? (
          // タブレット: 2列表示
          <Row>
            <Col md={6} className="mb-4">
              <div className="d-flex flex-column gap-3">
                <PortfolioSection
                  shareholderIdNameMap={shareholderIdNameMap ?? {} as
ShareholderIdNameMap}
                  ws={wsRef.current}
                  portfolioSummary={portfolioSummary}
                />
                <GenderStatsSection genderStats={genderStats} />
              </div>
            </Col>
            <Col md={6} className="mb-4">
              <div className="d-flex flex-column gap-3">
                <TransactionHistorySection
                  transactionHistory={transactionHistory}
                  isTryingConnect={is_trying_connect}
                  setIsTryingConnect={setIsTryingConnect}
                />
                <GenerationStatsSection generationStats={generationStats}
/>
              </div>
            </Col>
          </Row>
        ) : (
          // デスクトップ: 3列表示
          <Row>
            <Col lg={4} className="mb-4">
              <div className="d-flex flex-column gap-3">
                <GenderStatsSection genderStats={genderStats} />
                <GenerationStatsSection generationStats={generationStats}
/>
```

```
              </div>
            </Col>
            <Col lg={4} className="mb-4">
              <div className="d-flex flex-column gap-3">
                <PortfolioSection
                  shareholderIdNameMap={shareholderIdNameMap ?? {} as
ShareholderIdNameMap}
                  ws={wsRef.current}
                  portfolioSummary={portfolioSummary}
                />
              </div>
            </Col>
            <Col lg={4} className="mb-4">
              <TransactionHistorySection
                transactionHistory={transactionHistory}
                isTryingConnect={is_trying_connect}
                setIsTryingConnect={setIsTryingConnect}
              />
            </Col>
          </Row>
        )
        }
      </Container>
    </div>
  );
}

export default App;
```

4.2.2 リアルタイムデータ処理

リアルタイムデータ処理のためのカスタムフックを実装：

```
// hooks/useWebSocket.ts
import { useCallback, useEffect, useState } from 'react';

type WebSocketHook = [
  boolean,                  // isConnected
  () => void,               // connect
  () => void                // disconnect
];

const useWebSocket = (
  onMessage: (event: MessageEvent) => void,
  url: string
): WebSocketHook => {
  const [socket, setSocket] = useState<WebSocket | null>(null);
  const [isConnected, setIsConnected] = useState(false);

  // 接続関数
  const connect = useCallback(() => {
```

```
    if (socket !== null) return;

    const newSocket = new WebSocket(url);

    newSocket.onopen = () => {
      console.log('WebSocket接続成功');
      setIsConnected(true);
    };

    newSocket.onmessage = onMessage;

    newSocket.onclose = () => {
      console.log('WebSocket切断');
      setIsConnected(false);
      setSocket(null);
    };

    newSocket.onerror = (error) => {
      console.error('WebSocketエラー:', error);
      setIsConnected(false);
    };

    setSocket(newSocket);
  }, [url, onMessage, socket]);

  // 切断関数
  const disconnect = useCallback(() => {
    if (socket) {
      socket.close();
      setSocket(null);
      setIsConnected(false);
    }
  }, [socket]);

  // クリーンアップ
  useEffect(() => {
    return () => {
      if (socket) {
        socket.close();
      }
    };
  }, [socket]);

  return [isConnected, connect, disconnect];
};

export default useWebSocket;
```

4.2.3 Chart.jsによるデータ可視化

Chart.jsを使用して、さまざまなグラフを実装：

```tsx
// components/PortfolioSection.tsx - 地域別ポートフォリオ円グラフ
import { Pie } from 'react-chartjs-2';
import { Chart as ChartJS, ArcElement, Tooltip, Legend } from 'chart.js';

ChartJS.register(ArcElement, Tooltip, Legend);

// 地域別ポートフォリオ円グラフの作成
const RegionChart = ({ portfolioSummary }) => {
  // 資産価値が0より大きい地域のみ表示
  const filteredRegions = Object.entries(portfolioSummary.regionSummary)
    .filter(([_, regionData]) => regionData.asset > 0);

  if (filteredRegions.length === 0) {
    return null;
  }

  const data = filteredRegions.map(([_, regionData]) => regionData.asset);
  const labels = filteredRegions.map(([region, regionData]) => {
    const regionName = getRegionDisplayName(region);
    const ratio = (regionData.assetRatio * 100).toFixed(1);
    return `${regionName} (${ratio}%)`;
  });

  // 色の配列
  const colors = [
    '#FF6384', // 赤系 - 日本株
    '#36A2EB', // 青系 - 米国株
    '#FFCE56', // 黄系 - 欧州株
    '#4BC0C0', // 緑系 - その他
  ];

  const chartData = {
    labels: labels,
    datasets: [
      {
        data: data,
        backgroundColor: colors.slice(0, data.length),
        borderColor: colors.slice(0, data.length),
        borderWidth: 1,
      },
    ],
  };

  const options = {
    responsive: true,
    plugins: {
      legend: {
        position: 'right' as const,
      },
      title: {
        display: true,
        text: '地域別ポートフォリオ',
      },
```

```
    },
  };

  return (
    <div className="chart-container" style={{ height: '300px', marginTop:
'20px' }}>
      <Pie data={chartData} options={options} />
    </div>
  );
};
```

4.2.4 レスポンシブUI実装

React Bootstrapを活用して、画面サイズに応じて３段階のレイアウトを動的に切り替えるレスポンシブデザインを実装した。
発表時はTailwindCSSを用いたと述べたが、コードを見たときの可読性を考慮した結果、以下のような実装にした。

ブレークポイント設計

```
// ブレークポイント定義
const MOBILE_BREAKPOINT = 768;   // md未満: 1列表示
const TABLET_BREAKPOINT = 992;   // lg未満: 2列表示

// レイアウト判定
const isMobile = windowWidth < MOBILE_BREAKPOINT;
const isTablet = windowWidth >= MOBILE_BREAKPOINT && windowWidth <
TABLET_BREAKPOINT;
```

3段階レイアウト切り替え

モバイル(768px未満)	タブレット(769px-992px)	デスクトップ(992px以上)
1列縦並び	2列表示	3列表示

モバイル(768px未満)　　タブレット(769px-992px)　　デスクトップ(992px以上)

5. 技術的工夫と課題解決

5.1 データ整合性の保証

5.1.1 価格保証システム

株式取引システムにおいて最も重要な課題の一つは、取引時の株価参照と実際の約定価格の整合性である。本システムでは、以下の価格保証システムを実装した。

問題点:

- 取引処理時に参照する株価が古くなる可能性
- 複数取引の同時処理による価格の不整合
- 取引と価格更新のタイミングずれ

解決策:
PriceManagerによる一元管理システムを実装した。このシステムでは、取引データを受信した際に即座に株価を計算し、その価格で取引を確定させる。

```
public class PriceManager {
    // 株価管理用マップ
    private static Map<Integer, Integer> stockPriceMap = new
ConcurrentHashMap<>();

    // 取引と価格の統合処理
    public void processTransactionAndUpdatePrice(TransactionData
transaction) {
        // 1. 現在の株価を取得
        int stockId = transaction.getStockId();
        int quantity = transaction.getQuantity();
        int currentPrice = stockPriceMap.getOrDefault(stockId, 1000); // デ
```

```
フォルト価格

        // 2. 取引に基づいて新しい株価を計算
        int newPrice = calculateNewPrice(stockId, quantity, currentPrice);

        // 3. 株価を更新
        stockPriceMap.put(stockId, newPrice);

        // 4. 価格保証付き取引データを作成
        TransactionWithPrice txWithPrice = new
TransactionWithPrice(transaction, newPrice);

        // 5. 価格保証済みデータを送信
        sendToStockProcessor(txWithPrice);
    }

    // その他のメソッド...
}
```

このアプローチにより、以下の利点が得られた：

1. **データ整合性の完全保証**: 取引と株価更新が原子的に処理
2. **参照タイミング問題の解消**: 取引処理時の価格が常に最新
3. **アーキテクチャの単純化**: StockProcessor側でのデータ同期の懸念が不要

5.1.2 並行処理対応

リアルタイムシステムでは並行処理による競合状態の管理が重要である。。本システムでは以下の対策を実装した。

並行処理の課題:

- 複数スレッドからの同時データアクセス
- 処理順序の非決定性
- 更新の競合とデータ不整合

実装した対策:

1. **ConcurrentHashMapの活用**:

```
private static final ConcurrentHashMap<Integer, Integer> stockPriceMap
= new ConcurrentHashMap<>();
private static final ConcurrentHashMap<Integer, StockInfo>
StockMetadata = new ConcurrentHashMap<>();
private static final ConcurrentHashMap<Integer, Portfolio>
PortfolioManager = new ConcurrentHashMap<>();
```

2. **同期ブロックによる保護**:

```
synchronized (windowLock) {
    // ウィンドウ更新処理
}

synchronized (bufferLock) {
    // バッファ操作処理
}
```

3. **アトミック参照の使用**:

```
private static final AtomicReference<LocalTime[]> windowRef = new
AtomicReference<>(null);
private static final AtomicReference<Integer> selectedShareholderId =
new AtomicReference<>(null);
```

4. **スレッドセーフなコレクションの使用**:

```
private static final
java.util.concurrent.CopyOnWriteArrayList<Map<String, Object>>
transactionBuffer =
    new java.util.concurrent.CopyOnWriteArrayList<>();
```

これらの技術を組み合わせることで、高い並行性を維持しながらデータの整合性を保証している。特に、複数サービス間での通信において、データの不整合やタイミング問題を最小化することに成功した。

5.1.3 空売り防止機能(ロジック不十分)

投資システムでは、保有株数以上の売却（空売り）を防ぐ機能が重要である。本システムでは以下の空売り防止機能を実装した。

課題:

- 保有株数を超える売却注文の可能性
- Transaction.javaとStockProcessorでの保有株数管理の不整合
- フロントエンドでの正確な保有数表示

実装アプローチ:

1. **Transaction側での保有株数管理**:

```
// 株主と銘柄ごとの保有株数を管理
private static Map<String, Integer> shareholderStockHoldings = new
HashMap<>();

// 保有株数に基づく売買量決定
private static int determineQuantity(int shareholderId, int stockId) {
```

```
        String key = shareholderId + "_" + stockId;
        int currentHoldings = shareholderStockHoldings.getOrDefault(key,
    0);

        if (currentHoldings <= 0) {
            // 保有なしの場合は買いのみ
            return generatePositiveQuantity();
        } else {
            // 保有ありの場合、売りは保有数以内に制限
            if (random.nextBoolean()) {
                return generatePositiveQuantity(); // 買い注文
            } else {
                // 売り注文：保有数を超えない範囲で
                int maxSell = Math.min(currentHoldings, 50);
                return -random.nextInt(maxSell + 1);
            }
        }
    }
```

2. **フロントエンド側での対応**:
   時間的制約から完全な解決ができなかったため、フロントエンド側で負の保有数を表示しないよう対応した。

```
{portfolioSummary.stocks
  .filter(stock => stock.quantity > 0) // マイナス保有を除外
  .map(stock => (
    <tr key={stock.stockId}>
      <td>{stock.stockId}</td>
      <td>{stock.stockName}</td>
      <td>{stock.quantity.toLocaleString()}</td>
      {/* ... その他の表示処理 ... */}
    </tr>
  ))
}
```

今後の課題としては、Transaction.javaとStockProcessor間での保有株数の同期を完全に実装することが挙げられる。具体的には、StockProcessor接続時に過去の取引履歴を送信するか、StockProcessor接続時点から新たにポートフォリオ管理を開始する機能が必要である。

5.2 取引に応じた動的価格変動

5.2.1 価格変動アルゴリズム

実際の市場では、取引量や取引方向（買いか売りか）に応じて株価が変動する。本システムでは、よりリアルな市場シミュレーションを実現するために、取引に応じた動的価格変動アルゴリズムを実装した。価格変動のコード

このアルゴリズムは以下の要素を考慮している：

1. **取引方向**: 買い注文は価格上昇、売り注文は価格下落を引き起こす

2. **取引量**: 大量の取引ほど価格変動が大きくなる
3. **市場ノイズ**: ランダム要素を加えることで予測不可能性を実現
    1. 改めて考えると、取引量に基づく更新価格の決定は一意に定まるべきであると考えられるため、ランダム要素は持たせないほうが妥当かもしれない。

5.3 現実的な取引生成ロジック

5.3.1 保有株数ベースの売買判断

現実の投資家行動をシミュレートするため、保有株数に基づいた売買判断ロジックを実装した。保有株数ベースの売買量決定コード

このロジックにより、投資家の行動がより現実的になり、以下のような特徴が再現される：

- 新規銘柄は買いからスタート
- 少量保有では買い増し傾向が強い
- 中量保有ではバランスの取れた売買
- 大量保有では利益確定の売り傾向が強くなる

6. 開発過程で直面した課題

6.1 技術的課題

6.1.1 データ同期問題

リアルタイム株式取引システムを実装する中で、最も深刻な課題となったのがデータ同期問題である。具体的には、以下の問題が発生した：

1. **起動タイミングの非同期性**:
   Transaction.javaは起動時から取引を生成し、株主ごとの保有株数を内部で管理しているが、StockProcessorは後から接続するため、Transaction.javaが既に管理している保有株数情報とStockProcessorが構築するポートフォリオ情報の間に不整合が生じた。

2. **保有株数の不整合**:

```
// Transaction.javaの保有株数管理
private static Map<String, Integer> shareholderStockHoldings = new
HashMap<>();

// StockProcessor.javaのポートフォリオ管理
private static final ConcurrentHashMap<Integer, Portfolio>
PortfolioManager = new ConcurrentHashMap<>();
```

   この2つの独立したデータ構造間で同期が取れず、Transaction.javaが「株主Aは銘柄Bを10株保有している」と認識している一方で、StockProcessor.javaでは「株主Aは銘柄Bを保有していない」という状態が発生していた。

3. **マイナス保有株数の発生**:
   Transaction.javaが保有株数に基づいて売却取引を生成し、StockProcessorがそれを処理すると、StockProcessor側では保有株数がマイナスになる状況が発生していた：

```
// 例: StockProcessorでの処理
portfolio.addTransaction(stockId, stockName, -10, currentPrice); // 10株の売却
// portfolio内の該当銘柄の保有数が0なら、結果として-10株になる
```

6.1.2 マイクロサービス間通信の複雑性

マイクロサービスアーキテクチャを採用したことで、サービス間通信の複雑性が大きな課題となった：

1. **双方向通信の難しさ**:

```
Transaction.java ↔ PriceManager.java ↔ StockProcessor.java
```

   この構造で、StockProcessorの接続状態をTransactionに伝えるためには、PriceManagerを介した双方向通信が必要であった。しかし、この実装は、〆切直前の仕様変更だったこともあり、時間的制約の中で完全には実装できなかった。

2. **接続状態管理**:
   各サービスの接続状態を追跡し、他サービスに通知する仕組みが不十分だった。例えば：

```
// PriceManager.java内の一部
private static boolean isStockProcessorConnected = false;

// 接続検知コード（完全には実装できなかった）
public void onClientConnect(Socket client) {
  // クライアントがStockProcessorかどうかの識別が必要
  // ...
  isStockProcessorConnected = true;
  // Transactionへの通知機能も必要
}
```

3. **メッセージング形式の統一**:
   サービス間で異なるメッセージ形式を使用していたため、変換のオーバーヘッドが発生していた：

```
// Transaction -> PriceManager: シンプルなJSON
{
  "shareholderId": 1001,
  "stockId": 8301,
  "quantity": 10,
  "timestamp": "12:34:56.789"
}
```

```
// PriceManager -> StockProcessor: 拡張JSON
{
  "transaction": {
    "shareholderId": 1001,
    "stockId": 8301,
    "quantity": 10,
    "timestamp": "12:34:56.789"
  },
  "currentPrice": 1250,
  "previousPrice": 1230
}
```

このような複雑な通信要件は、シンプルなSocket通信では実装が困難で、より高度なメッセージングシステム（例：Apache Kafka、RabbitMQ）の導入が望ましかったと考えられる。

6.1.3 リアルタイム性能の確保

リアルタイムデータ処理におけるパフォーマンスの確保も大きな課題であった。

データ量と更新頻度のトレードオフ:
リアルタイム性を高めるために更新頻度を上げると、データ量が増加し、クライアント側の処理負荷が増大する問題があった。
クライアント側での接続完了から、表示更新までかなり時間を要していることが確認できた。
後述の7.1.2 パフォーマンス測定で詳しく検証する。

6.2 課題への対処法

6.2.1 実装した解決策

開発過程で直面した課題に対して、以下の解決策を実装した：

1. **データ整合性問題への対応**:
   - **フロントエンドフィルタリング**: マイナス保有株を表示しないフィルタリングを実装

```
{portfolioSummary.stocks
  .filter(stock => stock.quantity > 0) //  マイナス保有を除外
  .map(stock => (
    <tr key={stock.stockId}>
      {/* 表示処理 */}
    </tr>
  ))
}
```

     - 本来はTransaction.javaとStockProcessor.java間で保有株数を同期すべきだったが、時間的制約により、フロントエンドでマイナス保有を非表示にする応急対応を取った。
   - **PriceManager統合型アーキテクチャ**: 取引と株価の整合性を保証するため、PriceManagerを中心としたアーキテクチャに変更

```
// PriceManager.java内の価格保証処理
public void processTransactionAndUpdatePrice(TransactionData
transaction) {
  // 1. 現在の株価を取得
  int stockId = transaction.getStockId();
  int quantity = transaction.getQuantity();
  int currentPrice = stockPriceMap.getOrDefault(stockId, 1000);

  // 2. 取引に基づいて新しい株価を計算
  int newPrice = calculateNewPrice(stockId, quantity,
currentPrice);

  // 3. 株価を更新
  stockPriceMap.put(stockId, newPrice);

  // 4. 価格保証付き取引データを作成・送信
  TransactionWithPrice txWithPrice = new
TransactionWithPrice(transaction, newPrice);
  sendToStockProcessor(txWithPrice);
}
```

6.3 未解決の課題

実装完了時点で以下の課題が未解決のままとなっている：

1. **完全なデータ同期**:
   - Transaction.javaとStockProcessor.java間での完全なデータ同期は未実装。
   - 理想的には、StockProcessor接続時に過去の取引履歴を送信するか、接続時点から新たにポートフォリオ管理を開始する仕組みが必要。
2. **空売り機能の実装**:
   - 現実の市場では空売りが可能ですが、現状のシステムでは実装されていない。
   - 信用取引の概念も含めた拡張が今後の課題である。
3. **取引と株価の時系列データ管理**:
   - 現状はリアルタイムデータのみを扱い、過去データの永続化や分析は未実装。
   - データベースとの連携により、時系列分析や傾向予測機能の追加が望まれる。
4. **エラーハンドリングの強化**:
   - ネットワーク切断、サービスクラッシュなどの異常時の動作が十分に考慮されていない。
   - より堅牢なエラーリカバリーメカニズムの実装が必要。
5. **パフォーマンス最適化**:
   - 現時点でもフロントエンドでの表示までに目に見えた遅延が見られ、取引量を増やしたり大量のデータ処理時のパフォーマンスがさらに低下することが予想される。
   - より実用的なシステムにするには原因の解明・解決が必要。

7. 評価と検証・考察

7.1 機能評価

7.1.1 実装した機能の動作確認

開発した主要機能について、以下の観点から動作確認を行った：

1. **取引生成機能**:
    - ✓取引履歴が正しく生成され、表示されることを確認
    - ✓取引の種類（買い・売り）が正しく識別されることを確認
2. **株価変動システム**:
    - ✓取引量に応じた株価変動が適切に発生することを確認
    - ✓買い注文による株価上昇、売り注文による株価下落の挙動を検証
    - ✓株価変動の幅が現実的な範囲内であることを確認
3. **ポートフォリオ管理**:
    - ✗取引発生時のポートフォリオ更新にはラグがあることを確認
    - ✓評価損益、含み損益の計算が正確であることを確認
    - ✓地域別資産配分が正しく計算・表示されることを確認
4. **統計情報**:
    - ✓性別・年代別統計情報が正確に集計されることを確認
    - ✓グラフ表示が直感的でデータを正確に反映していることを確認
5. **WebSocket通信**:
    - ✓サーバー・クライアント間のリアルタイム通信が安定していることを確認
    - ✓切断時の再接続処理が動作することを確認
6. **レスポンシブUI**:
    - ✓異なる画面サイズでの表示が最適化されていることを確認
    - ✓モバイルデバイスでの操作性を検証

これらの確認により、基本的な機能要件は満たされていることが確認できたが、長時間稼働時の安定性やWeb側での更新速度については、さらなる検証が必要である。

7.1.2 パフォーマンス測定

システムのパフォーマンスを評価するため、以下のような測定を行った。

1. **接続後、取引履歴が表示されるまでの時間**
    - **約5~6秒**（100msごとに10取引の場合）
    - **約19~20秒**（100msごとに100取引の場合）
    - ブラウザのDeveloper Toolsのコンソールを確認したところ、初回データの送信はほぼラグなく行われていたため、React側の初期レンダリングに時間がかかっている可能性がある。

2. **取引履歴の更新速度**
    - 取引履歴の更新は約100msごとに行われる。
    - **約4~5秒**(100msごとに100取引の場合、一度目の表示から次の表示に切り替わるまでの時間)
    - ここから回数を経るごとに、早くなっていき、一定速度の更新に落ち着く。
3. **ポートフォリオの更新速度**
    - 株主を選択してから、ポートフォリオ情報が表示されるまでの時間は約3秒。
    - コンソールを確認したところ、プルダウンボタンで選択してから結果が送られてくるまでのは1~2秒程度で、表示にもそこまで時間がかかっていないと考えられる。
4. **Interaction to Next Paint (INP)**
    - ブラウザのDeveloper ToolsのPerformanceタブで確認できるINPは、ユーザーの操作から次の描画までの時間を測定する指標。
    - 200ms秒未満でページの応答性は良好
    - 200ms~500msで改善が必要
    - 500ms~以でページの応答性が低い
    - 参考: Web.dev - Interaction to Next Paint (INP)
    - 今回実装ページを確認したところ、**8512ms**と非常に高い値が出ており、ページの応答性が低いことがINPからも確認できた。

上記の測定結果から、主に取引履歴の更新について、大きく時間がかかっていることがわかった。これは、取引履歴の量が増えると、Reactの再レンダリングに時間がかかるためと考えられる。特に、100件以上の取引がある場合、初回の表示に時間がかかる傾向がある。
また、INPの値が非常に高いことから、ページの応答性が低いことも確認できた。これは、取引履歴の更新やポートフォリオの更新に時間がかかるため、ユーザーの操作に対する応答が遅くなっていることが原因と考えられる。

この点については、大きく改善の必要があり、以下のような対策が考えられる：

- **取引履歴の表示方法の見直し**: 大量の取引履歴を一度に表示するのではなく、ページネーションや無限スクロールを導入し、表示する取引履歴の量を制限する。
- **Reactのパフォーマンス最適化**: Reactの再レンダリングを最小限に抑えるため、`useMemo`や`useCallback`を活用し、必要な部分のみを更新するようにする。
    - `useMemo`は一度使用して修正を試みたが、更新がされているはずが、表示される取引の内容が変更されなくなってしまったりするなどの問題が発生したため、使用を中止し、シンプルに受け取ったデータをそのまま毎回描画する形に戻した。

7.3 システムの限界と制約

現状のシステムには、以下の限界と制約が存在する：

1. **データの永続性**:
    - メモリ内処理のみで、永続的なデータストレージが実装されていない
    - サービス再起動時にすべてのデータが初期化される
2. **スケーラビリティの制約**:
    - 単一サーバーでの稼働を前提としており、分散処理が実装されていない
    - 同時接続クライアント数に上限がある
3. **取引の現実性**:

- 市場の深さ、板情報などの概念が実装されていない
- 取引の時間帯による変動パターンが考慮されていない
- 企業の決算発表などのイベントに基づく変動が実装されていない

4. **セキュリティ上の制約**:
   - WebSocketの通信が暗号化されていない
   - ユーザー認証機能がない
   - 入力検証やサニタイズ処理が最小限

5. **機能の限界**:
   - 空売りや信用取引などの高度な取引タイプが実装されていない
   - 指値注文、成行注文などの注文タイプが区別されていない
   - 配当、株式分割などのコーポレートアクションが考慮されていない

これらの制約は、簡素化した実験用のシステムと考えると必要ないかもしれないが、機能拡張としては実装の優位性があると考えられる。

7.4 リアルタイムデータ処理技術の比較

本システムで使用したリアルタイムデータ処理技術と、他の一般的な技術を比較する：

1. **自作スライディングウィンドウ vs. Apache Flink**
   - **自作スライディングウィンドウ (採用)**:
     - 軽量でオーバーヘッドが少ない
     - プロジェクトに必要な機能のみ実装
     - 学習目的として理解しやすい
   - **Apache Flink**:
     - より高度なストリーム処理機能
     - 高いスケーラビリティ
     - 障害耐性と状態管理が優れている

本システムでは、教育目的と実装の単純さを優先して、より軽量でシンプルな技術選択を行った。本番環境で同様のシステムを構築する場合は、Apache Flinkを使用することが適していると考えられる。

8. 今後の展望

8.1 短期的改善案

8.1.1 データ同期問題の根本解決

現状のデータ同期問題を根本的に解決するために、以下の改善を検討している：

1. **初期状態同期メカニズム**:
   StockProcessor.javaが接続時に過去の取引履歴と現在の株価データを要求し、PriceManager.javaからTransaction.javaが保有している現在の保有状況を受け取る仕組みを実装する。

2. **イベントソーシング**:
   全ての取引を時系列順に保存し、任意の時点の状態を再構築できる仕組み。

ただし、長時間稼働により、メモリ使用量が増大化していくため、データベースとの連携による取引履歴の永続保存との両用が必須である。

これらの改善により、各サービス間でのデータ整合性が確保され、マイナス保有株数などの問題が解消されると考えられる。
StockProcessor.javaで保有株数管理を一元化する方法もあるが、現実的な株式取引システムでは、株取引は常に行われているため、Transaction.javaが生成開始時からのポートフォリオ管理を行うのが自然であると考えられる。

8.1.2 投資家行動の高度化

現状の単純な確率モデルをより現実的なものにするため、投資家タイプを導入し、各タイプに応じた行動パターンの変更を実装する事が考えられる。

例:

- 長期投資家: 淡々と買い増しを行う傾向
- デイトレーダー: 短期的な価格変動に敏感に反応
- 割安株投資家: 低PER銘柄を好む
- 成長株投資家: 高成長セクター
- 配当狙い: 高配当利回り銘柄を選択

この改善により、より現実的かつ多様な投資家行動のシミュレーションが可能になり、システム全体のリアリティが向上すると考えられる。

8.2 長期的発展構想

8.2.1 機能拡張（信用取引、配当システム等）

長期的に以下の機能拡張を検討している：

1. **信用取引システム**:
    - 空売りや信用取引の実装
    - 株主ごとの初期資産の設定(現金残高の管理)
2. **配当システム**:
    - 時間の進みを早めて、配当日に配当の分配を実装
    - 配当情報の管理と配当処理の実装
    - 配当利回りの計算
    - 各株主のポートフォリオを確認して配当分配
3. **通知システム**:
    - 取引や配当、大きな株価変動の通知機能
    - WebSocketを利用したリアルタイム通知

これらの機能拡張により、より本格的かつ教育的価値の高い株式市場シミュレーションが実現できると考えられる。

9. まとめ

9.1 達成した成果

本プロジェクトでは、以下の主要な成果を達成した：

1. **リアルタイム株式取引分析システム**の実装：
    - 株式取引データの自動生成
    - 取引量に応じた動的株価変動
    - ポートフォリオのリアルタイム更新と分析
    - WebSocketによるリアルタイムデータ配信
2. **マイクロサービスアーキテクチャ**の構築：
    - Transaction・PriceManager・StockProcessorの連携
    - Socket通信による効率的なデータ伝送
    - サービス間の責任分担と疎結合設計
3. **先進的なデータ処理技術**の実装：
    - スライディングウィンドウによる時間枠データ管理
    - ConcurrentHashMapによる並行データアクセス
    - WebSocketによるブラウザへのリアルタイム配信
4. **より良いユーザーインターフェース**の開発：
    - Reactコンポーネントによるモジュラー設計
    - Chart.jsを用いた直感的なデータ可視化
    - レスポンシブ設計によるマルチデバイス対応
5. **統計分析機能**の実装：
    - 性別・年代別の投資統計
    - 地域別ポートフォリオ分析
    - リアルタイム更新される取引履歴

これらの成果により、教育・実験目的として十分な機能を持つ株式取引シミュレーションシステムを実現できた。特に、マイクロサービス間の連携とリアルタイムデータ処理の実装は、大規模システム開発の学習として有意義な経験となった。

9.2 技術的な学び

本プロジェクトを通じて、以下の技術的な学びを得ることができた：

1. **リアルタイムデータ処理**:
    - WebSocketを使用したリアルタイム双方向通信の実装方法
    - スライディングウィンドウによる時間枠ベースのデータ処理手法
    - 非同期処理とイベント駆動アーキテクチャの設計

2. **分散システム設計**:

    - マイクロサービス間の効率的な通信方法
    - データ整合性の確保と同期問題への対応
    - サービス間の責任分担と疎結合設計

3. **並行処理**:

    - ConcurrentHashMapなどの並行データ構造の活用
    - スレッドセーフな実装テクニック
    - ロック戦略と競合回避

4. **フロントエンド開発**:

    - React + TypeScriptによる型安全な開発
    - Chart.jsを使った動的データ可視化
    - WebSocketクライアントの実装とイベント処理

9.3 今後の課題

本プロジェクトを通じて明らかになった主な課題と、今後の改善方針は以下の通り：

1. **データ整合性の完全な保証**:

    - Transaction.javaとStockProcessor間の保有株数同期を改善する
    - サービス間の双方向通信メカニズムの確立
    - スタートアップシーケンスの最適化（接続順序の制御）

2. **投資家行動の高度化**:

    - 年齢別の投資傾向をより詳細にモデル化
    - 個人ごとのリスク選好度や投資スタイルを設定
    - 市場状況に応じた動的な投資行動の実装

3. **機能拡張**:

    - 信用取引（空売り）システムの実装
    - 配当システムの追加（時間進行の加速と配当日設定）

4. **パフォーマンス最適化**:

    - Web側での表示更新速度の問題解決
    - データベースを用いた永続化と履歴保存
    - 統計計算の並列処理化
    - クライアント側キャッシュ機能の強化

これらの課題を段階的に解決することで、より現実的かつ教育的価値の高いシミュレーションシステムに発展させることができる。特に、データ整合性の問題とパフォーマンス最適化は優先度が高く、次のステップとして取り組むべき重要な課題である。

10. 参考文献・使用技術

1. React公式ドキュメント: https://react.dev/
2. Chart.js: https://www.chartjs.org/docs/latest/
3. Java WebSocket: https://www.oracle.com/technical-resources/articles/java/jsr356.html
4. TailwindCSS: https://tailwindcss.com/docs
5. React Bootstrap: https://react-bootstrap.netlify.app/
6. Gson: https://github.com/google/gson
7. Java Concurrency: https://docs.oracle.com/javase/tutorial/essential/concurrency/
8. TypeScript: https://www.typescriptlang.org/docs/

付録

A. ソースコード構成

本システムのソースコード構成は以下の通り：

```
work-06/
├── server-app/                 # Javaバックエンド
│   ├── src/main/java/io/github/vgnri/
│   │   ├── StockProcessor.java          # メイン分析エンジン
│   │   ├── RunConfiguration.java        # システム起動管理
│   │   ├── model/
│   │   │   ├── Transaction.java         # 取引生成サービス
│   │   │   ├── PriceManager.java        # 株価管理サービス
│   │   │   ├── Portfolio.java           # ポートフォリオ管理
│   │   │   ├── StockInfo.java           # 銘柄情報
│   │   │   ├── StockMetadata.java       # 銘柄メタデータ型定義クラス
│   │   │   ├── StockPrice.java          # 株価型定義クラス
│   │   │   ├── StockPriceCalculator.java # 株価決定計算クラス
│   │   │   └── ShareholderInfo.java     # 投資家情報
│   │   ├── server/
│   │   │   └── WebsocketServer.java     # WebSocket通信
│   │   ├── util/
│   │   │   ├── LocalTimeTypeAdapter.java # Gson用
│   │   └── loader/
│   │       └── MetadataLoader.java      # CSVデータ読み込み
│   └── src/main/resources/              # 設定・データファイル
│       ├── initial_price_data.csv
│       ├── shareholder_metadata.csv
│       ├── stock_metadata.csv
│       ├── stock_price_data.csv
│       └── tsukuba_metadata.csv
└── client-app/                # React フロントエンド
    ├── src/
    │   ├── App.tsx                      # メインアプリケーション
    │   ├── components/
    │   │   ├── PortfolioSection.tsx    # ポートフォリオ表示セクション
    │   │   ├── TransactionHistorySection.tsx    # 取引履歴表示セクション
    │   │   ├── TransactionTable.tsx    # 取引履歴テーブル
    │   │   ├── GenderStatsSection.tsx  # 性別統計セクション
    │   │   └── GenerationStatsSection.tsx # 年代別統計セクション
```

```
│   └── DataType.tsx                # TypeScript型定義
├── package.json
└── vite.config.ts
```

主要ファイル機能説明

バックエンド

1. **StockProcessor.java**: システムの中核部分。取引データの処理、ポートフォリオ管理、統計計算、WebSocket通信を担当
2. **RunConfiguration.java**: 全サービスの起動と管理を行う設定クラス
3. **Transaction.java**: 現実的な取引データを自動生成するサービス
4. **PriceManager.java**: 取引に応じて株価を計算・管理するサービス
5. **Portfolio.java**: 各投資家のポートフォリオデータを管理するクラス
6. **WebsocketServer.java**: フロントエンドとのリアルタイム通信を実現するサーバー

フロントエンド

1. **App.tsx**: Reactアプリケーションのエントリポイント、レイアウト管理
2. **PortfolioSection.tsx**: ポートフォリオ表示コンポーネント
3. **TransactionHistorySection.tsx**: 取引履歴表示コンポーネント
4. **GenderStatsSection.tsx**: 性別統計グラフコンポーネント
5. **GenerationStatsSection.tsx**: 年代別統計グラフコンポーネント

B. 開発環境・ツール

本プロジェクトの開発に使用した環境とツールは以下の通り：

開発環境

- **OS**: Windows 11 + WSL2 (Ubuntu 24.04)
- **JDK**: OpenJDK 21.0.7
- **Node.js**: v18.16.0
- **npm**: 9.5.1

開発ツール

- **エディタ**: Visual Studio Code
- **ビルドツール**: Maven 3.9.5
- **バージョン管理**: Git 2.43.0
- **フロントエンドビルド**: Vite 5.0.0

ライブラリ・フレームワーク（主要なもの）

- **バックエンド**:
    - Java WebSocket: 1.5.3
    - Gson: 2.10.1
    - log4j: 2.20.0

- **フロントエンド**:
    - React: 18.2.0
    - TypeScript: 5.3.3
    - Chart.js: 4.4.0
    - TailwindCSS: 3.3.2
    - React Bootstrap: 2.8.0

C. 動作確認手順

本システムの動作確認は以下の手順で行う。

前提条件

以下の環境が必要：

- **Java 17以上**: バックエンド実行環境
- **Node.js 18以上**: フロントエンド実行環境
- **npm 9.5.1以上**: パッケージ管理ツール

1. プロジェクトの取得

```
git clone https://github.com/Vigener/RealTime-Data.git
cd work-06
```

2. バックエンドの起動

サーバーアプリケーションを起動：

```
cd server-app

# 初回実行時は依存関係のコンパイルが必要
javac -cp "lib/*:src" src/main/java/io/github/vgnri/*.java

# システム一括起動（推奨）
java -cp "lib/*:src/main/java" io.github.vgnri.RunConfiguration
```

以下のように個別に起動することも可能（起動順序厳守）：

```
# 1. Transaction サービス起動
java -cp "lib/*:src/main/java" io.github.vgnri.model.Transaction

# 2. PriceManager サービス起動
java -cp "lib/*:src/main/java" io.github.vgnri.model.PriceManager
```

```
# 3. StockProcessor サービス起動
java -cp "lib/*:src/main/java" io.github.vgnri.StockProcessor
```

3. フロントエンドの起動

クライアントアプリケーションを起動：

```
cd client-app

# 依存関係のインストール（初回のみ）
npm install

# 開発サーバーの起動
npm run dev
```

4. システムへのアクセス

1. ウェブブラウザで `http://localhost:5173` にアクセス
2. 画面上部の「接続」ボタンをクリックしてWebSocket接続を確立
3. リアルタイムデータの表示が開始されることを確認

5. システムの操作方法

ポートフォリオ閲覧

1. 画面中央の「株主選択」ドロップダウンから投資家を選択
2. 選択した投資家のポートフォリオデータが表示される
3. 保有株式一覧と地域別の資産配分が表示される

統計情報の確認

- **性別統計**: 男女別の投資額・損益が円グラフとテーブルで表示
- **年代別統計**: 20代～70代以上の投資傾向がグラフで表示
- **取引履歴**: 直近5秒間の取引データがリアルタイムに更新

レスポンシブ表示の確認

- ブラウザのサイズを変更することで表示レイアウトが切り替わる
  - **992px以上**: 3列表示（統計 | ポートフォリオ | 取引履歴）
  - **768px以上992px未満**: 2列表示（ポートフォリオ+統計 | 取引履歴+統計）
  - **768px未満**: 1列表示

D. 発表スライド

2025年7月30日に行われた最終課題の発表で使用したスライドのリンクです。

OneDrive 共有リンク