AI Agent调用小模型预测股票收益率——进度汇报

1. 系统总体框架

业务目标

我们致力于开发一个基于深度学习和Java Web的股票收益率预测系统，满足以下业务目标：

1. **数据采集与处理**：
   - 采集多支股票的历史量价数据。
   - 使用深度学习模型预测指定股票未来3天的收益率。
2. **数据库构建与更新**：
   - 利用Tushare或Ricequant等数据源，构建一个实时更新的股票量价信息数据库。
3. **用户交互**：
   - 构建B/S模式信息系统。
   - 用户通过指令与大模型交互，完成数据查询、预测等任务。
   - 垂直领域任务（如收益率预测）由本地模型完成，结果通过前端反馈给用户。

系统模块划分及交互

1. 前端

- **页面设计**：
  - 使用课程提供的模板设计用户界面，确保系统美观、易用。
  - 提供动态交互功能，例如数据可视化、任务指令输入框。
- **大模型API和Agent接入**：
  - 与OpenAI、通义千问等大模型API集成，解析用户指令。
  - 动态调用本地深度学习模型完成垂直任务。

2. 中端

- **服务逻辑**：
  - 使用Spring Boot框架搭建服务端。

- 负责任务分发，包括解析用户指令、调用数据库或本地模型。
- 实现RESTful API，为前端提供统一的接口支持。

3. 后端

- **数据库**：
  - 数据源：使用Tushare采集10支高校持股的股票交易数据。
  - 构建MySQL数据库，存储历史量价数据和实时更新信息。
  - 提供查询接口，支持大模型任务和用户查询。
- **Agent模型**：
  - 模型选择：计划使用Transformer或LSTM建模，因其适合处理时间序列特征。
  - 数据：基于采集的历史交易数据进行模型训练和微调。
  - 功能：支持收益率预测任务，作为系统的垂直领域工具。

2. 当前进展汇报

已完成的任务

1. 系统设计：

- 完成系统模块划分，明确前中后端的功能与交互方式。
- 选定技术栈，包括Spring Boot框架、MySQL、深度学习框架（PyTorch/TensorFlow）和大模型API。

`Spring Boot` 项目分为以下层次：

层次	功能
Controller	接收用户请求，调用 Service 层，返回结果。
Service	负责业务逻辑，包括调用大语言模型、股票数据接口、深度学习模型和数据库。
Repository	负责与数据库交互，包括查询和更新股票数据。
Model	定义数据结构，包括用户请求、股票数据、预测结果等。

详细设计
参看流程图及以下文字说明。

1. Controller 层

- 功能：
  接收用户请求并将其转发给 Service 层。
  返回结果（如预测结果和分析报告）。
- 接口设计：
  接口名称：POST /api/analysis
  参数：JSON 格式的用户问题，例如：

```
{
  "query": "基于最近一年某上市企业的股价分析未来三个月的股价变动情况"
}
```

返回值：JSON 格式预测结果和分析报告，例如：

```
{
  "company": "某上市企业",
  "prediction": [120.5, 121.3, 123.8],
  "analysis": "未来三个月的股价预计呈缓慢上升趋势。"
}
```

示例代码：

```
@RestController
@RequestMapping("/api")
public class CommandController {
    @Autowired
    private AnalysisService analysisService;

    @PostMapping("/analysis")
    public ResponseEntity<?> analyze(@RequestBody CommandRequest command) {
        PredictionResult result = analysisService.processCommand(command.getQuery());
        return ResponseEntity.ok(result);
    }
}
```

2. Service 层

- 功能：
  解析用户请求：调用大语言模型提取企业名称、时间范围等。
  调用股票数据接口更新数据库。
  调用深度学习模型预测股票走势。
  整合预测结果，生成分析报告。

- 涉及接口：
  大语言模型接口：
  接口名称：POST /api/analyze-query
  参数：问题文本
  返回值：提取的关键信息（企业名称、时间范围等）。
  股票数据接口：
  接口名称：第三方平台 API，如 GET /stock/{symbol}
  参数：企业代码、时间范围。
  返回值：股票历史数据。
  深度学习模型接口：
  接口名称：POST /model/predict
  参数：JSON 格式股票数据。
  返回值：预测结果。
  示例代码：

```
@Service
public class AnalysisService {
    @Autowired
    private DatabaseService databaseService;
    @Autowired
    private AIModelService aiModelService;
    @Autowired
    private LLMService llmService;

    public PredictionResult processCommand(String query) {
        // 调用大语言模型接口解析问题
        CommandInfo info = llmService.parseQuery(query);

        // 检查数据库，若缺失数据则调用股票数据接口更新
        List<StockData> data = databaseService.getStockData(info.getCompany(), info.getTimeRange
        if (data.isEmpty()) {
            data = databaseService.updateStockData(info.getCompany(), info.getTimeRange());
        }

        // 调用深度学习模型预测
        List<Double> prediction = aiModelService.predict(data);

        // 整理结果并返回
        return new PredictionResult(info.getCompany(), prediction, "未来三个月股价有上升趋势。");
    }
}
```

3. Repository 层

- 功能：
  提供对数据库的操作，包括查询和插入。
- 涉及接口：
  查询接口：根据企业名称和时间范围查询数据。
  插入接口：插入新获取的股票数据。
  示例代码：

```
@Repository
public interface StockDataRepository extends JpaRepository<StockData, Long> {
    @Query("SELECT s FROM StockData s WHERE s.company = :company AND s.date BETWEEN :startDate A
    List<StockData> findStockData(
        @Param("company") String company,
        @Param("startDate") LocalDate startDate,
        @Param("endDate") LocalDate endDate
    );
}
```

4. 数据模型

涉及数据结构：

```
// 用户请求数据模型（CommandRequest）：
public class CommandRequest {
    private String query;
    // Getter & Setter
}
// 股票数据模型（StockData）：
@Entity
public class StockData {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String company;
    private LocalDate date;
    private double price;
    // Getter & Setter
}
// 预测结果数据模型（PredictionResult）：
public class PredictionResult {
    private String company;
    private List<Double> prediction;
    private String analysis;
    // Constructor, Getter & Setter
}
```

5. 层间对接设计

Controller → Service：

调用 processCommand 方法，传递用户请求内容。

Service → Repository：

通过 StockDataRepository 查询和更新数据库。
Service → 大语言模型接口：
使用 HTTP 客户端调用大语言模型接口。
Service → 股票数据接口：
使用 HTTP 客户端调用第三方平台 API，获取股票数据。
Service → 深度学习模型接口：
使用 HTTP 客户端将数据发送至 Python 模型服务。

2. 数据采集与处理：

- 已采集10支高校持股的股票过去一年的历史量价数据。
- 初步完成数据预处理，包括缺失值填充、特征工程（如移动平均线、收益率计算）。

一、调用股票数据源 API

1. 股票数据源 Python API 背景
   提供者的接口是基于 Python 开发的，通常通过 HTTP 提供 RESTful API 服务。我们需要在 Python 中运行一个服务，例如使用 Flask 或 FastAPI 搭建 API。
2. Python 端接口实现（示例）
   以下示例展示了如何用 Python 的 Flask 提供股票数据接口：

```python
from flask import Flask, request, jsonify
import yfinance as yf  # 示例：通过 yfinance 获取股票数据

app = Flask(__name__)

@app.route('/api/stock', methods=['GET'])
def get_stock_data():
    symbol = request.args.get('symbol')  # 企业代码
    start_date = request.args.get('start_date')  # 起始日期
    end_date = request.args.get('end_date')  # 结束日期

    if not symbol or not start_date or not end_date:
        return jsonify({"error": "Missing parameters"}), 400

    # 获取股票数据
    data = yf.download(symbol, start=start_date, end=end_date)
    if data.empty:
        return jsonify({"error": "No data found"}), 404

    result = data.reset_index().to_dict(orient='records')
    return jsonify(result)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

3. Spring Boot 调用 Python API

3.1 使用 RestTemplate 或 WebClient

示例代码（使用 RestTemplate 调用）：

```
@Service
public class StockDataService {
    private static final String STOCK_API_URL = "http://localhost:5000/api/stock";

    public List<Map<String, Object>> fetchStockData(String symbol, String startDate, String
        RestTemplate restTemplate = new RestTemplate();
        String url = String.format("%s?symbol=%s&start_date=%s&end_date=%s",
                STOCK_API_URL, symbol, startDate, endDate);

        try {
            ResponseEntity<List> response = restTemplate.getForEntity(url, List.class);
            return response.getBody();
        } catch (HttpClientErrorException e) {
            throw new RuntimeException("Error fetching stock data: " + e.getResponseBodyAsS
        }
    }
}
```

3.2 说明
依赖库：Spring 提供了 RestTemplate 和 WebClient 供我们与外部 API 通信。需要在 pom.xml 中添加：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

接口路径：根据 Python 服务的地址调整 STOCK_API_URL。

二、调用 Python 深度学习模型

1. 深度学习模型接口背景
   模型服务由 Python 提供，通过 Flask 或 FastAPI 等框架暴露 HTTP API。
2. Python 端接口实现
   以下代码演示一个接受数据并返回预测结果的接口：

```python
from flask import Flask, request, jsonify
import numpy as np
import tensorflow as tf

app = Flask(__name__)

# 加载预训练模型
model = tf.keras.models.load_model('stock_prediction_model.h5')

@app.route('/api/predict', methods=['POST'])
def predict():
    data = request.json.get('data')  # 获取输入数据
    if not data:
        return jsonify({"error": "No input data"}), 400

    # 转换数据格式并预测
    input_data = np.array(data).reshape(1, -1)  # 示例：输入为二维数组
    prediction = model.predict(input_data).tolist()

    return jsonify({"prediction": prediction})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=6000)
```

3. Spring Boot 调用 Python 模型接口

   3.1 使用 RestTemplate 或 WebClient

   示例代码：

```java
@Service
public class AIModelService {
    private static final String MODEL_API_URL = "http://localhost:6000/api/predict";

    public List<Double> predictStockTrend(List<Double> inputData) {
        RestTemplate restTemplate = new RestTemplate();

        // 构造请求数据
        Map<String, Object> request = new HashMap<>();
        request.put("data", inputData);

        try {
            // 发送 POST 请求
            ResponseEntity<Map> response = restTemplate.postForEntity(MODEL_API_URL, reques
            Map<String, Object> responseBody = response.getBody();

            if (responseBody.containsKey("prediction")) {
                return (List<Double>) responseBody.get("prediction");
            } else {
                throw new RuntimeException("Prediction error: " + responseBody);
            }
        } catch (HttpClientErrorException e) {
            throw new RuntimeException("Error calling prediction API: " + e.getResponseBody
        }
    }
}
```

3. 模型选择与初步实验：

- 已实现LSTM模型的初步训练代码，验证了其在样本数据上的表现。
- 探索了Transformer模型在时间序列数据上的应用，实验结果表明其具备较强的特征捕捉能力。

```
# !pip install tushare
import tushare as ts
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# 设置token
ts.set_token('c1977fccb6b193794feab07a8f6fbf03ca8e2a297613fa486d7fe9be')
pro = ts.pro_api()
# 获取股票列表
df = pro.stock_basic(exchange='', list_status='L', fields='ts_code,symbol,name,area,industr

# 获取股票日线数据
df = pro.daily(ts_code='600000.SH', start_date='20200101', end_date='20201231')

# 获取股票分钟线数据
df = pro.minute(ts_code='600000.SH', start_date='20200101', end_date='20201231', freq='1min
# 获取上证指数数据
df = ts.get_index()
print(df.head())

# 获取上证指数历史行情数据
df = ts.get_hist_data('000001', start='2018-01-01', end='2018-07-01')
print(df.head())

# 绘制上证指数历史行情数据
df['close'].plot()
plt.show()
LSTM模型
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler

# 假设df是包含股票数据的DataFrame，'Close'列包含我们想要预测的收盘价
# df = pd.read_csv('your_stock_data.csv')  # 如果你从CSV文件读取数据

# 为了简化，这里我们只使用收盘价
data = df.filter(['Close'])

# 将数据转换为数组
```

```
dataset = data.values

# 将数据集分为训练集和测试集
train_size = int(len(dataset) * 0.8)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size], dataset[train_size:len(dataset)]

# 数据归一化
scaler = MinMaxScaler(feature_range=(0, 1))
train_scaled = scaler.fit_transform(train)
test_scaled = scaler.transform(test)

# 创建数据集，其中X是特征，y是标签
def create_dataset(dataset, look_back=1):
  X, y = [], []
  for i in range(len(dataset) - look_back - 1):
      a = dataset[i:(i + look_back), 0]
      X.append(a)
      y.append(dataset[i + look_back, 0])
  return np.array(X), np.array(y)

look_back = 60  # 定义时间步长
X_train, y_train = create_dataset(train_scaled, look_back)
X_test, y_test = create_dataset(test_scaled, look_back)

# 转换为PyTorch张量
X_train = torch.tensor(X_train).float()
y_train = torch.tensor(y_train).float()
X_test = torch.tensor(X_test).float()
y_test = torch.tensor(y_test).float()

# 重塑输入数据的形状 [samples, time steps, features]
X_train = X_train.reshape(X_train.shape[0], 1, X_train.shape[1])
X_test = X_test.reshape(X_test.shape[0], 1, X_test.shape[1])

# 定义LSTM模型
class LSTMModel(nn.Module):
  def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
      super(LSTMModel, self).__init__()
      self.hidden_dim = hidden_dim
      self.layer_dim = layer_dim

      # LSTM层
```

```
        self.lstm = nn.LSTM(input_dim, hidden_dim, layer_dim, batch_first=True)

        # 线性层
        self.linear = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # 初始化隐藏状态和细胞状态
        h0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()
        c0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()

        # 前向传播
        out, (hn, cn) = self.lstm(x, (h0.detach(), c0.detach()))
        out = self.linear(out[:, -1, :])  # 只取最后一个时间步的输出
        return out

input_dim = 1  # 特征维度
hidden_dim = 50  # 隐藏层维度
layer_dim = 1  # LSTM层数
output_dim = 1  # 输出维度

model = LSTMModel(input_dim, hidden_dim, layer_dim, output_dim)

# 损失函数和优化器
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# 训练模型
num_epochs = 100
for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()

    # 前向传播
    outputs = model(X_train)

    # 计算损失
    loss = criterion(outputs, y_train)

    # 反向传播和优化
    loss.backward()
    optimizer.step()

    if epoch % 10 == 0:
```

```
        print(f'Epoch {epoch+1}/{num_epochs}, Loss: {loss.item()}')

    # 进行预测
    model.eval()
    with torch.no_grad():
      train_predict = model(X_train)
      test_predict = model(X_test)

    # 反归一化预测数据
    train_predict = scaler.inverse_transform(train_predict.detach().numpy())
    test_predict = scaler.inverse_transform(test_predict.detach().numpy())

    # 绘制结果
    plt.figure(figsize=(15, 6))
    plt.plot(scaler.inverse_transform(dataset), label
```

正在进行的工作

1. 前端开发：

- 集成课程模板，设计用户界面布局。
- 开发交互组件，如指令输入框和数据可视化面板。

2. Agent模块开发：

- 编写Agent逻辑，支持基于大模型API的用户指令解析。
- 搭建本地模型调用框架，将垂直任务结果反馈给用户。

3. Spring Boot服务搭建：

- 开发中端RESTful API，连接前端和后端。
- 实现数据查询和模型调用的接口。

4. 数据库搭建：

- 完成MySQL数据库的初始化，创建包含股票基本信息和交易数据的表结构。
- 使用Python编写数据更新脚本，可定期从Tushare拉取新数据并更新数据库。

遇到的问题

1. 数据实时性：

- 数据源（Tushare）的接口速率限制导致数据更新效率较低，正在尝试优化更新频率。

2. 模型训练难点：

- Transformer模型对小样本时间序列数据敏感，当前训练结果不够稳定。

3. 前后端交互：

- 大模型解析的指令多样性较高，可能需要进一步规范用户输入格式。

3. 实施进度安排

时间表

阶段	时间范围	任务内容
第一阶段	已完成	系统设计、数据采集、数据库搭建和模型选择。
第二阶段	1周内	完成前端页面布局，Agent模块初步开发。
第三阶段	2周内	完成Spring Boot服务开发，数据库与前端整合。
第四阶段	2周内	深度学习模型的优化训练及本地部署。
第五阶段	3周内	集成大模型API，完成整体测试和功能演示。

分工安排

1. **系统设计**：
   - 负责：系统模块划分、技术栈选型、接口设计、完成系统架构设计、数据库设计。
   - 成员：全体成员
2. **数据处理**：
   - 负责：数据预处理、实时更新脚本的优化。
   - 成员：李聪、丁健。
3. **前端开发**：
   - 负责：完成UI设计，确保功能组件与后台的接口对接。

- 成员：李佳、李聪、崔家伟、童梓航。

4. **后端与模型**：
   - 负责：模型训练、服务接口设计与实现。
   - 成员：李聪、丁健、崔家伟、童梓航。
5. **集成与测试**：
   - 负责：大模型API与本地服务的集成，系统整体测试。
   - 成员：李佳、丁健。

4. 总结

系统开发已完成整体框架设计和关键模块的搭建工作，接下来将重点优化前端交互、Agent逻辑及模型性能。我们团队将按计划推进各项任务，确保系统在预定时间内完成并展示所有功能。