

<Profiler 구현>

20230012 컴퓨터공학과 김지연



<목차>

1. 프로그램 개요	
2. 프로그램 수행절차 분석 및 실행 결과	
2.1. 디렉토리 구성 예시	5
2.2. 로컬 개발 환경 설정 (터미널 또는 Git Bash에서 진행)	5
2.3. 폴더 및 파일 구조 만들기	6
2.4. parser.js 작성	6
2.5. app.js 코드 (기본 서버 실행용)	7
2.6. cmd에서 notepad views\index.ejs 입력	8
2.7. app.js 수정	10
2.8. http://localhost:3000/ 에서의 실행 화면	11
2.9. index.ejs 수정	11
2.10. Multer 설정 & 업로드 라우트 추가	11
2.11. cmd에서 node app.js 실행 후 결과	13
2.12. inputFile.txt 파일 선택 → “업로드 및 분석” 클릭	13
2.13. 결과	14
3. 주요 기능 설명	15
3.1. 파일 업로드 기능 (Multer 사용)	15
3.2. Core/Task 기반 데이터 분석 기능	15
3.3. Chart.js를 이용한 실시간 결과 시각화 기능	15
3.4. EJS 기반 동적 렌더링	15
4. 소스 코드 분석	16
4.1. app.js - 서버 구성과 라우팅 처리	16
4.2. parser.js - 데이터 파싱 및 통계 분	16
4.3. index.ejs - 클라이언트 화면 구성석	16
4.4. 디렉토리 구조	17
5. 프론트엔드 시각화 및 로직	18
5.1. Chart.js를 이용한 시각화	18
5.2. EJS 템플릿을 통한 동적 데이터 삽입	18
5.3. 사용자 업로드 인터페이스	18
5.4. 향후 확장 가능 요소	19

<그림 목차>

그림 1 - 실행	8
그림 2 - JOSN 형식	8
그림 3 - 그래프_1	11
그림 4 - multer 설치	12
그림 5 - 그래프_2	13
그림 6 - 파일 선택	13
그림 7 - 그래프_3	14

1. 프로그램 개요

본 프로젝트는 Node.js 기반의 웹 프로파일러 프로그램으로, 사용자가 업로드한 성능 데이터 파일(inputFile.txt)을 분석하여 각 Core와 Task의 실행 시간에 대한 통계 정보를 제공하고, 이를 시각적으로 그래프로 표현하는 시스템이다.

웹 환경에서 사용자는 텍스트 파일을 업로드함으로써 별도의 설치 없이 간편하게 데이터 분석을 수행할 수 있으며, 분석 결과는 Core 및 Task 단위로 평균값(AVG), 최솟값(MIN), 최댓값(MAX)을 구하여 시각화된다.

시각화는 Chart.js를 활용한 막대그래프로 제공되며, 사용자는 각 지표를 직관적으로 확인할 수 있다. 이를 통해 다중 Core/Task 시스템의 성능 비교, 병목 분석 등에 활용할 수 있는 웹 기반 프로파일링 도구로 설계되었다.

해당 프로그램은 Express.js 웹 프레임워크, EJS 템플릿 엔진, multer 파일 업로드 처리기를 기반으로 구성되었으며, GitHub를 통해 코드가 관리된다.

2. 프로그램 수행 절차 분석

2.1. 디렉토리 구성 예시

```
/profiler-project
|
├── routes/
|   ├── index.js          ← 파일 업로드 및 데이터 처리 라우터
├── public/
|   ├── styles.css        ← 그래프 스타일링 (선택)
|   ├── script.js         ← Chart.js 이용한 시각화
├── views/
|   ├── index.ejs         ← 업로드 및 그래프 표시 페이지
├── uploads/
|   ├── inputFile.txt     ← 업로드된 파일 저장 경로
├── models/
|   ├── parser.js         ← MIN/MAX/AVG 계산 로직
├── app.js                ← 메인 서버
└── package.json
```

2.2. 로컬 개발 환경 설정 (터미널 또는 Git Bash에서 진행)

원하는 폴더로 이동

`cd ~/Desktop` # 또는 원하는 경로

새 폴더 생성 후 이동

`mkdir node-profiler-project`

`cd node-profiler-project`

Git 초기화

`git init`

GitHub 원격 저장소 연결

`git remote add origin https://github.com/jiYeon0730/node-profiler.git`

Node 프로젝트 초기화

`npm init -y`

필요한 모듈 설치

`npm install express multer ejs chart.js`

2.3. 폴더 및 파일 구조 만들기

cmd에서 다음을 입력

```
"routes", "views", "public", "uploads", "models" | ForEach-Object { mkdir $_ }  
notepad app.js  
notepad models\parser.js  
notepad routes\index.js
```

.gitignore 파일에 다음을 작성

```
node_modules/  
uploads/
```

2.4. parser.js 작성

```
// models/parser.js  
const fs = require('fs');  
const path = require('path');  
  
function parseProfilerData(filePath) {  
  const content = fs.readFileSync(filePath, 'utf-8');  
  const blocks = content.trim().split('\n\n');  
  
  const results = {  
    cores: {},  
    tasks: {},  
  };  
  
  blocks.forEach(block => {  
    const lines = block.trim().split('\n');  
    const taskLabels = lines[0].trim().split(/\s+/).slice(1);  
  
    lines.slice(1).forEach(line => {  
      const [coreLabel, ...values] = line.trim().split(/\s+/);  
      const numericValues = values.map(Number);  
  
      if (!results.cores[coreLabel]) results.cores[coreLabel] = [];  
      results.cores[coreLabel].push(numericValues);  
  
      taskLabels.forEach((task, idx) => {  
        if (!results.tasks[task]) results.tasks[task] = [];  
        results.tasks[task].push(numericValues[idx]);  
      });  
    });  
  });  
}
```

```

        });
    });
});

const stats = { coreStats: {}, taskStats: {} };

for (let [core, data] of Object.entries(results.cores)) {
    const flat = data.flat();
    stats.coreStats[core] = {
        min: Math.min(...flat),
        max: Math.max(...flat),
        avg: +(flat.reduce((a, b) => a + b, 0) / flat.length).toFixed(2),
    };
}

for (let [task, values] of Object.entries(results.tasks)) {
    stats.taskStats[task] = {
        min: Math.min(...values),
        max: Math.max(...values),
        avg: +(values.reduce((a, b) => a + b, 0) / values.length).toFixed(2),
    };
}

return stats;
}

module.exports = { parseProfilerData };

```

2.5. app.js 코드 (기본 서버 실행용)

```

// app.js
const express = require('express');
const path = require('path');
const { parseProfilerData } = require('./models/parser');

const app = express();
const PORT = 3000;

// 기본 라우트
app.get('/', (req, res) => {
    const filePath = path.join(__dirname, 'uploads', 'inputFile.txt');

```

```

    try {
      const stats = parseProfilerData(filePath);
      res.json(stats); // JSON 결과 브라우저에서 확인 가능
    } catch (error) {
      res.status(500).send('파일 파싱 중 오류 발생: ' + error.message);
    }
  });

// 서버 실행
app.listen(PORT, () => {
  console.log(`서버 실행 중 🚀 http://localhost:${PORT}`);
});

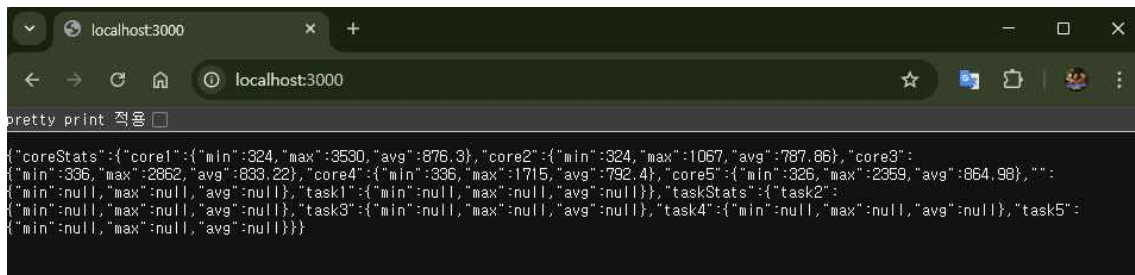
```

cmd에서 다음을 입력
 node app.js

http://localhost:3000 <-에서 coreStats,taskStats 결과가 JSON형식으로 보임.



<그림 1 - 실행>



<그림 2 - JSON형식>

cmd로 가서 notepad uploads\inputFile.txt 생성 후 입력

2.6. cmd에서 notepad views\index.ejs 입력

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Profiler 결과 시각화</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

```



```

</head>
<body>
  <h2>📊 Core별 평균 그래프</h2>
  <canvas id="coreChart" width="600" height="300"></canvas>

  <h2>📊 Task별 평균 그래프</h2>
  <canvas id="taskChart" width="600" height="300"></canvas>

  <script>
    const coreStats = <%- JSON.stringify(coreStats) %>;
    const taskStats = <%- JSON.stringify(taskStats) %>;

    const coreLabels = Object.keys(coreStats);
    const coreData = coreLabels.map(k => coreStats[k].avg);

    const taskLabels = Object.keys(taskStats);
    const taskData = taskLabels.map(k => taskStats[k].avg);

    const ctx1 = document.getElementById('coreChart').getContext('2d');
    new Chart(ctx1, {
      type: 'bar',
      data: {
        labels: coreLabels,
        datasets: [{
          label: 'Core 평균',
          data: coreData,
        }]
      }
    });

    const ctx2 = document.getElementById('taskChart').getContext('2d');
    new Chart(ctx2, {
      type: 'bar',
      data: {
        labels: taskLabels,
        datasets: [{
          label: 'Task 평균',
          data: taskData,
        }]
      }
    });
  </script>

```

```
</script>
</body>
</html>
```

2.7. app.js 수정

```
const express = require('express');
const path = require('path');
const { parseProfilerData } = require('./models/parser');

const app = express();
const PORT = 3000;

// 뷰 엔진 설정
app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));

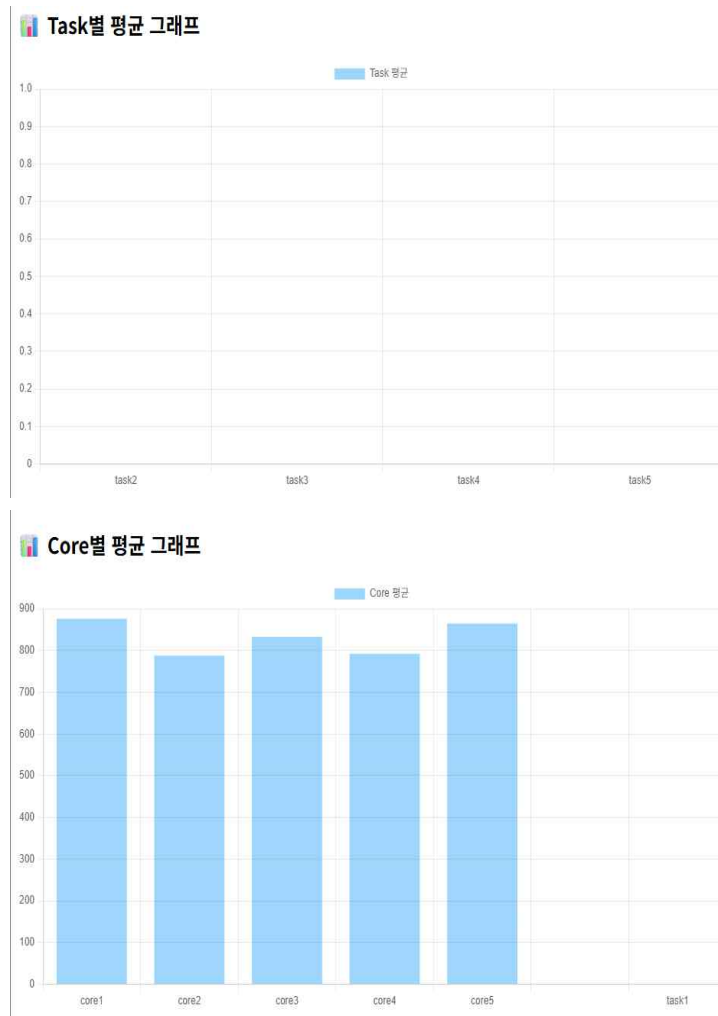
// 정적 파일 경로 (public 폴더 필요시)
app.use(express.static(path.join(__dirname, 'public')));

// 라우트
app.get('/', (req, res) => {
  const filePath = path.join(__dirname, 'uploads', 'inputFile.txt');
  try {
    const stats = parseProfilerData(filePath);
    res.render('index', {
      coreStats: stats.coreStats,
      taskStats: stats.taskStats
    });
  } catch (error) {
    res.status(500).send('파일 파싱 중 오류 발생: ' + error.message);
  }
});

app.listen(PORT, () => {
  console.log(`🔗 http://localhost:${PORT} 에서 실행 중`);
});
```

cmd에서 다시 node app.js 실행

2.8. http://localhost:3000/ 에서의 실행 화면



<그림 3 - 그래프_1>

2.9. index.ejs 수정

views/index.ejs 열고 <body> 태그 안에 아래 품을 가장 위에 추가

```
<h1>📁 inputFile.txt 업로드</h1>
```

```
<form action="/upload" method="POST" enctype="multipart/form-data">
```

```
  <input type="file" name="profilerFile" accept=".txt" required />
```

```
  <button type="submit">업로드 및 분석</button>
```

```
</form>
```

```
<hr>
```

2.10. Multer 설정 & 업로드 라우트 추가

- npm install multer 설치

```
C:\Users\PC\Desktop\node-profiler-project>npm install multer

up to date, audited 103 packages in 1s

17 packages are looking for funding
  run 'npm fund' for details

found 0 vulnerabilities
```

<그림 4 - multer 설치>

- app.js에 multer 설정 추가

```
const multer = require('multer');
```

- 파일 업로드를 위한 저장 위치와 설정 추가

```
// 업로드 경로 설정
```

```
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads'); // uploads 폴더에 저장
  },
  filename: function (req, file, cb) {
    cb(null, 'inputFile.txt'); // 항상 같은 이름으로 저장 (덮어쓰기)
  }
});
```

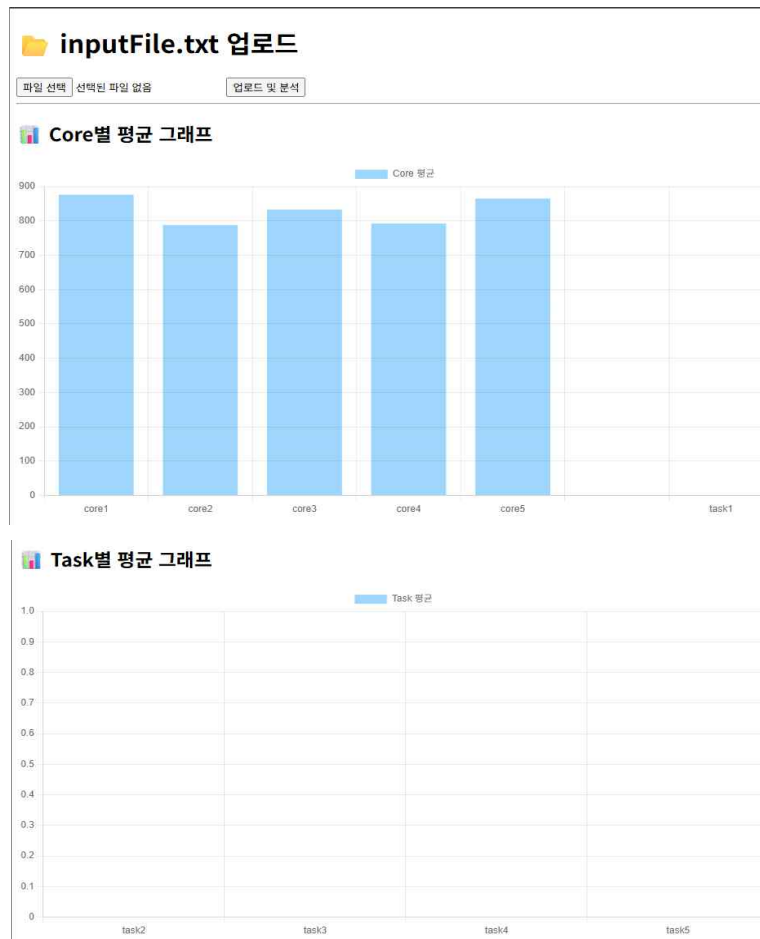
```
const upload = multer({ storage: storage });
```

- app.js 아래쪽 /upload 라우트 추가

```
// 업로드 처리 라우트
```

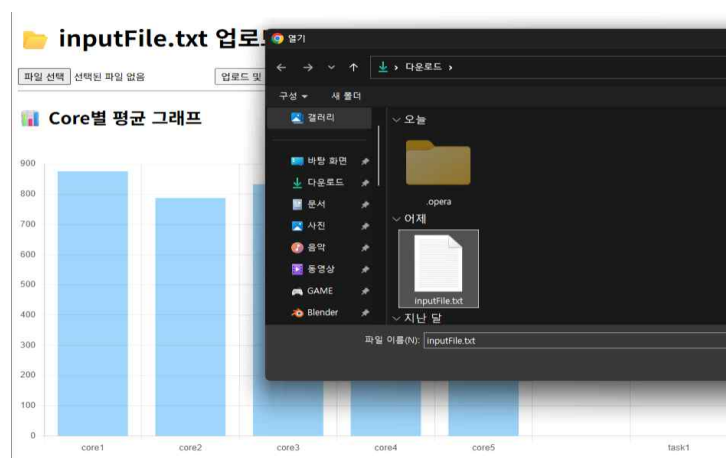
```
app.post('/upload', upload.single('profilerFile'), (req, res) => {
  const filePath = path.join(__dirname, 'uploads', 'inputFile.txt');
  try {
    const stats = parseProfilerData(filePath);
    res.render('index', {
      coreStats: stats.coreStats,
      taskStats: stats.taskStats
    });
  } catch (error) {
    res.status(500).send('파일 분석 중 오류 발생: ' + error.message);
  }
});
```

2.11. cmd에서 node app.js 실행 후 결과



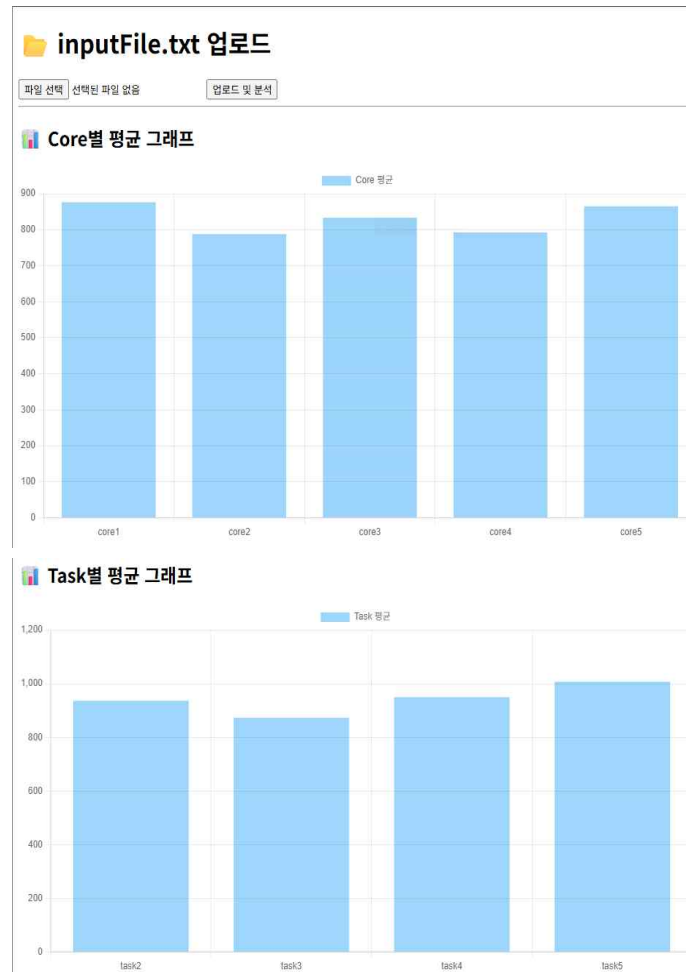
<그림 5 - 그래프_2>

2.12. inputFile.txt 파일 선택 → “업로드 및 분석” 클릭



<그림 6 - 파일 선택>

2.13. 결과



<그림 7 - 그래프_3>

3. 주요 기능 설명

이 프로젝트는 웹 환경에서 파일 업로드부터 데이터 분석, 시각화까지 가능한 간단하고 직관적인 성능 프로파일링 시스템을 구현한 것이다. 주요 기능은 다음과 같다.

3.1. 파일 업로드 기능 (Multer 사용)

사용자는 웹 브라우저에서 .txt 형식의 입력 파일을 선택하여 업로드할 수 있으며, 업로드된 파일은 서버의 uploads 폴더에 저장된다.

이 기능은 multer 모듈을 통해 구현되었으며, 사용자가 업로드한 파일명을 inputFile.txt로 고정하여 덮어쓰기 방식으로 처리한다.

3.2. Core/Task 기반 데이터 분석 기능

업로드된 파일을 parser.js에서 읽어들이고, 각 Core 및 Task에 대해 다음 통계값을 계산한다:

- 최솟값 (MIN)
- 최댓값 (MAX)
- 평균값 (AVG)

분석은 각 줄의 데이터를 정리하여 Core 기준과 Task 기준으로 분류 후, JavaScript의 내장 수학 함수를 이용하여 처리한다.

3.3. Chart.js를 이용한 실시간 결과 시각화 기능

분석된 결과는 웹 페이지에서 Chart.js 라이브러리를 활용해 막대그래프 형태로 시각화된다.

- Core별 평균값을 보여주는 그래프
 - Task별 평균값을 보여주는 그래프
- 두 개의 Canvas 요소에 각각 출력된다.

사용자는 업로드 즉시 분석 결과를 웹에서 시각적으로 확인할 수 있다.

3.4. EJS 기반 동적 렌더링

템플릿 엔진 EJS를 사용하여 업로드 후 분석된 데이터를 클라이언트로 전달하고, HTML 문서 안에 직접 삽입하여 렌더링함으로써 결과를 실시간으로 확인 가능하게 했다.

4. 소스 코드 분석

이 프로젝트는 Node.js와 Express 프레임워크를 기반으로 동작하며, 구조는 크게 다음과 같이 구성된다:

app.js (메인 서버), parser.js (데이터 분석), index.ejs (UI 템플릿), uploads/ (파일 저장), views/models/routes 폴더

4.1. app.js - 서버 구성과 라우팅 처리

```
const express = require('express');
const multer = require('multer');
const path = require('path');
const { parseProfilerData } = require('./models/parser');
- Express 서버를 구성하고, 뷰 엔진으로 EJS를 사용함
- GET /: 기본 경로로 접근 시 서버에 존재하는 inputFile.txt를 분석하여 그래프를 렌더링
- POST /upload: 사용자가 업로드한 텍스트 파일을 서버에 저장하고 분석 결과를 다시 렌더링
app.post('/upload', upload.single('profilerFile'), (req, res) => {
  const filePath = path.join(__dirname, 'uploads', 'inputFile.txt');
  const stats = parseProfilerData(filePath);
  res.render('index', {
    coreStats: stats.coreStats,
    taskStats: stats.taskStats
  });
});
```

4.2. parser.js - 데이터 파싱 및 통계 분석

```
const content = fs.readFileSync(filePath, 'utf-8');
const blocks = content.trim().split('\n\n');
- 업로드된 텍스트 파일을 읽고, Core별, Task별로 데이터를 분류함
- 평균(AVG), 최솟값(MIN), 최댓값(MAX)을 계산하여 JSON 형태로 반환
stats.coreStats[core] = {
  min: Math.min(...flat),
  max: Math.max(...flat),
  avg: +(flat.reduce((a, b) => a + b, 0) / flat.length).toFixed(2)
};
```

4.3. index.ejs - 클라이언트 화면 구성

```
<form action="/upload" method="POST" enctype="multipart/form-data">
  <input type="file" name="profilerFile" accept=".txt" required />
  <button type="submit">업로드 및 분석</button>
</form>
- 업로드 폼: 사용자가 .txt 파일을 선택해 서버로 업로드 가능
```


- 시각화: Chart.js를 사용해 Core/Task별 평균값을 막대그래프로 출력

```
const coreLabels = Object.keys(coreStats);  
const coreData = coreLabels.map(k => coreStats[k].avg);
```

4.4. 디렉토리 구조

uploads/ : 업로드된 `inputFile.txt` 저장 위치 (Git에 포함되지 않음)

models/ : 데이터 처리 로직 (`parser.js`)

views/ : 화면 렌더링 템플릿 (`index.ejs`)

public/ : 필요 시 CSS, JS 등 정적 파일 저장 (현재 Chart.js는 CDN으로 사용)

5. 프론트 엔드 시각화 및 로직

웹 브라우저에서 Core 및 Task별 성능 데이터를 직관적으로 확인할 수 있도록 Chart.js를 사용하여 막대그래프로 시각화하며, EJS 템플릿을 통해 HTML에 데이터를 삽입하는 방식으로 구성되어 있다.

5.1. Chart.js를 이용한 시각화

```
<canvas id="coreChart"></canvas>
```

```
<canvas id="taskChart"></canvas>
```

- HTML <canvas> 요소 두 개를 사용하여 Core별 평균값과 Task별 평균값을 각각 그래프로 출력
- Chart.js 라이브러리는 CDN 방식으로 불러오며, 사용법이 간단하고 시각적으로 세련된 차트를 제공

```
new Chart(ctx1, {  
  type: 'bar',  
  data: {  
    labels: coreLabels,  
    datasets: [{  
      label: 'Core 평균',  
      data: coreData,  
    }]  
  }  
});
```

- 서버에서 전달된 coreStats, taskStats 데이터를 JavaScript 객체로 받아 사용함
- labels: Core나 Task 이름, data: 평균값

5.2. EJS 템플릿을 통한 동적 데이터 삽입

```
<script>  
  const coreStats = <%- JSON.stringify(coreStats) %>;  
  const taskStats = <%- JSON.stringify(taskStats) %>;  
</script>
```

- EJS 문법 <%- %>를 사용하여 서버에서 계산된 통계 데이터를 JavaScript 변수로 삽입
- 이 방식으로 페이지를 새로고침하거나 업로드 후 바로 데이터가 시각화됨

5.3. 사용자 업로드 인터페이스

```
<form action="/upload" method="POST" enctype="multipart/form-data">  
  <input type="file" name="profilerFile" required />  
  <button type="submit">업로드 및 분석</button>  
</form>
```

- 사용자는 .txt 파일을 선택하여 업로드하면 자동으로 분석 결과가 페이지에 시각화됨
- 업로드 → 분석 → 시각화가 한 번에 동작하는 직관적 인터페이스 제공

5.4. 향후 확장 가능 요소

- Core/Task 외에 다른 기준(예: 날짜, 카테고리 등)으로도 필터링 가능
- Bootstrap 등을 사용해 그래프 스타일을 개선할 수 있음
- 버튼 동적 생성 기능을 추가하면 Core/Task 선택형 그래프로 확장 가능