课题申请书

课题名称：为 Arthas 实现 JFR 解析和前端UI库

课题主导师：Raymond

申请人：陈文骏

邮箱：m202473970@hust.edu.cn

1、课题背景

1.1 课题背景

Arthas 是一个Java 诊断工具，使得开发人员在无需修改应用程序代码的情况下，动态地监视和解决生产环境中Java 应用程序的问题。Java Flight Recorder（JFR）是 Oracle JDK 内置的性能采样工具，能够以极低的性能开销持续采集 JVM 运行过程中的关键事件数据，如 CPU 使用、内存分配、线程状态、文件 IO、异常抛出等，是进行 Java 性能分析和线上问题排查的重要基础。然而，JFR 默认生成的是二进制格式的 `.jfr` 文件，手动分析难度较大。且Arthas 目前尚未支持对 JFR 文件的直接解析和可视化展示。在当前微服务普及、服务持续运行、对系统可观测性要求日益提高的背景下，为 Arthas 增加 JFR 分析支持，不仅能提升其诊断覆盖面和深度，也能帮助开发者更方便地理解应用的运行状况，精准定位性能瓶颈。本课题旨在通过 JDK 的 JFR API 实现 `.jfr` 文件的后端解析，并以 Web 界面呈现可交互的火焰图和事件数据，打造一个轻量、友好、开源的性能分析工具，为 Arthas 用户提供生产环境下更强大的可观测能力。

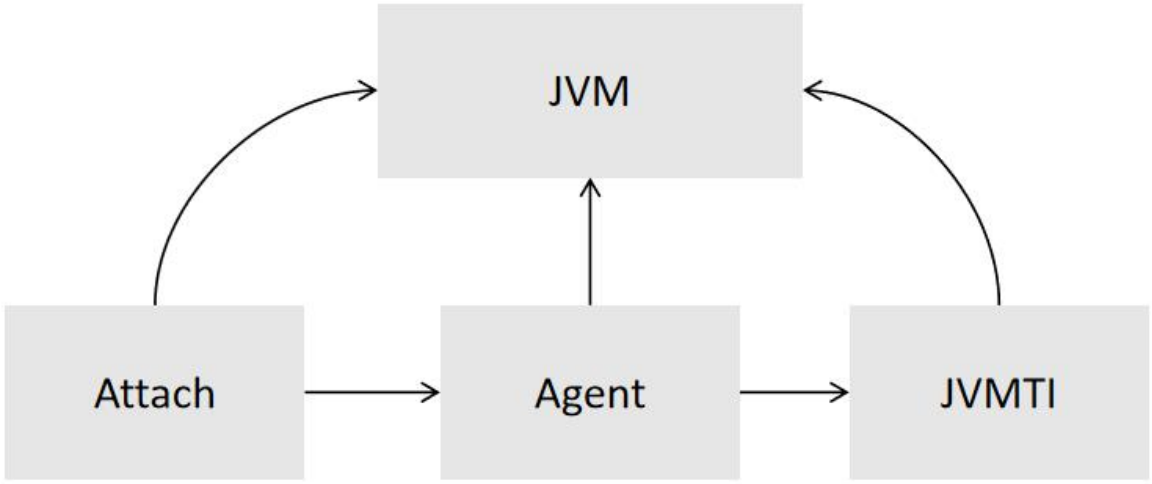
1.2 课题相关文档

- 基本需求：
  - `issue仓库地址：` https://github.com/alibaba/arthas/issues/3012
- 前端参考：
  - `Ant Design 网站：` https://ant.design/index-cn
  - `jifa火焰图前端实现：` https://github.com/eclipse-jifa/jifa/tree/main/frontend

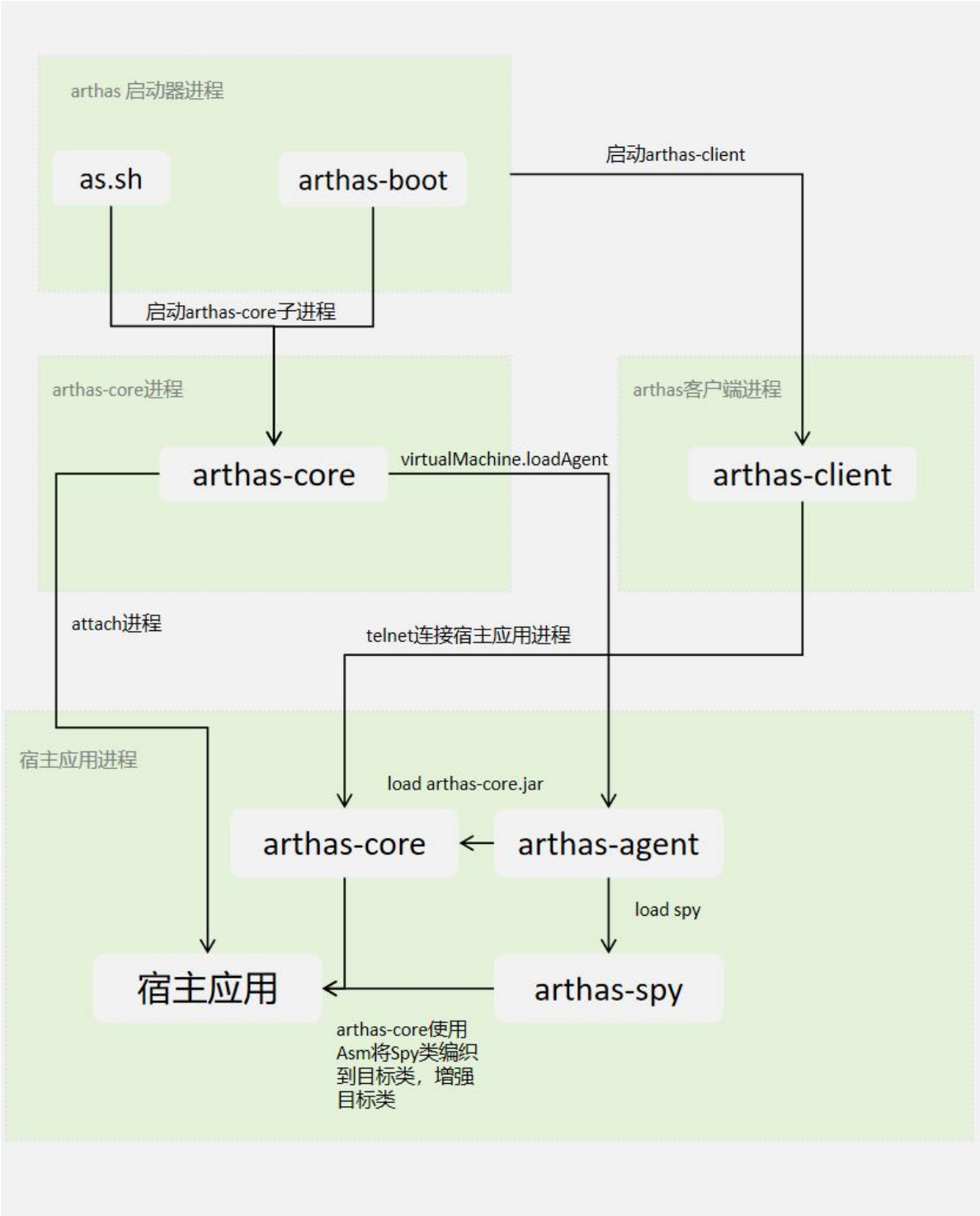
- 后端参考：
  - `jifa后端解析实现`： https://github.com/eclipse-jifa/jifa/tree/main/analysis/jfr
  - `arthas源码解析`：https://juejin.cn/post/7247428110164590629

2、课题可行性

2.1 Arthas架构



- Attach：连接运行JVM，加载Agent。
- Agent：内部实现agentmain和premain方法。
- JVMTI：JVM 提供的一个 **C/C++ native 接口**，可以做比 agent 更底层的事。



- `arthas-boot`：启动辅助工具，负责 attach 和初始化。
- `as.sh / as.bat`：封装 boot jar，提供命令行增强体验。
- `arthas-agent`：Java Agent，连接核心逻辑桥梁。
- `arthas-core`：核心执行模块，命令、控制逻辑等。
- `arthas-spy`：注入钩子，配合 `watch` 等命令使用。
- `arthas-client`：控制台客户端，Telnet 交互用。

2.2 课题概要设计

- 后端设计：
  - **文件解析**：使用 `jdk.jfr.consumer.RecordingFile` 遍历 `.jfr` 事件，提取关键事件类型（如 `jdk.ExecutionSample`、`jdk.FileRead`、`jdk.ExceptionThrown` 等）；
  - **数据聚合**：按事件类别、线程、时间段等维度统计；
  - **REST 接口**：基于 Spring Boot 提供文件上传和火焰图 JSON 数据接口；
  - **火焰图构建**：复用 JIFA 中 `generateCpuTime()` 与 `generate()` 生成 `FlameGraph` 结构。
- 前端设计：
  - 获取JSON数据后，使用火焰图可视化库绘制，拟采用`d3-flame-graph`；
  - 使用 Ant Design 的 `Table` 组件展示分组统计后的事件详情，例如按线程或类名汇总的 CPU 消耗、内存分配、IO 时长、异常次数等。

2.3 后端设计

.jfr文件解析及解析数据生成

- 参考JIFA实现
  - `createFlameGraph`(核心函数)：初始化数据容器 os（火焰图数据）、names（任务名-总时间映射）、symbolTable（符号表）。通过维度类型选择数据源（如 result.getCpuTime()），调用 generate(...) 或 generateCpuTime(...) 生成数据。构建 FlameGraph 对象并返回。
  - 数据生成方法：
    - `generate`：遍历 DimensionResult 的任务结果，根据 taskSet 和 include 过滤任务。调用 `doTaskResult` 转换堆栈跟踪为火焰图结构。
    - `doTaskResult`：将堆栈帧倒序排列，生成符号ID数组（通过 SymbolMap），记录样本数和任务名。
    - `generateCpuTime`：计算每个样本对应的实际时间（perSampleTime），生成基于CPU时间的火焰图数据。

```
public class JFRAnalyzerImpl implements JFRAnalyzer {

//.....................................................................
...

    private FlameGraph createFlameGraph(ProfileDimension dimension,
AnalysisResult result, boolean include,List<String> taskSet) {
        // 存储火焰图中每条栈的原始数据（如方法调用栈及其耗时）
        List<Object[]> os = new ArrayList<>();
        // 线程或任务名到 ID 的映射，用于图中区分不同线程
        Map<String, Long> names = new HashMap<>();
        // 符号表，用于方法名或类名的映射管理
        SymbolMap symbolTable = new SymbolMap();
        if (dimension == ProfileDimension.CPU) {
             // 获取 CPU 时间分析结果
            DimensionResult<TaskCPUTime> cpuTime = result.getCpuTime();
             // 生成 CPU 时间的火焰图数据
            generateCpuTime(cpuTime, os, names, symbolTable, include, taskSet);
```

```
        } else {
            // 根据不同维度提取相应的结果数据
            DimensionResult<? extends TaskResultBase> DimensionResult = switch
(dimension) {
                //...
            };
             // 生成非 CPU 时间维度的火焰图数据
            generate(DimensionResult, os, names, symbolTable, include, taskSet);
        }
        // 创建 FlameGraph 对象并设置其属性
        FlameGraph fg = new FlameGraph();
        fg.setData(os.toArray(new Object[0][]));
        fg.setThreadSplit(names);
        fg.setSymbolTable(symbolTable.getReverseMap());
        System.out.println(fg);
        System.out.println(fg.getSymbolTable());
        return fg;
    }
```

FlameGraph 数据结构

- `GraphBase` 抽象出多个图形分析通用的结构，比如线程信息、符号表等，可以被火焰图、调用图等复用。

```
//基类
public class GraphBase {
    private List<String> threads = new ArrayList<>();
    private Map<String, Long> threadSplit = new HashMap<>();
    private Map<Integer, String> symbolTable = new HashMap<>();
}
```

- `FlameGraph` 添加特有的数据结构 `Object[][] data`，用于具体展示火焰图堆栈帧。

```
public class FlameGraph extends GraphBase {
    private Object[][] data = new Object[0][];
}
```

- 返回的json数据示例。

```
{
  "threads": ["main", "GC Thread#1"],
  "threadSplit": {
    "main": 123456,
    "GC Thread#1": 45678
  },
  "symbolTable": {
    "1": "java.lang.Thread.run",
    "2": "com.example.MyService.process",
    "3": "java.util.HashMap.get"
  },
  "data": [
    [1, 0, 50, 1],  // [symbolId, startTime, duration, depth]
    [2, 10, 30, 2],
    [3, 20, 10, 3]
```

```
  ]
}
```

RESTful 服务

```
@PostMapping("/api/jfr/upload")
public ResponseEntity<String> uploadJfr(@RequestParam("file") MultipartFile
file) {
    Path tmp = Paths.get(System.getProperty("java.io.tmpdir"),
file.getOriginalFilename());
    Files.copy(file.getInputStream(), tmp, StandardCopyOption.REPLACE_EXISTING);
    try {
        jfrParser.parseJfrFile(tmp);
    } catch (IOException e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("解析
失败");
    }
    return ResponseEntity.ok("上传并解析成功");
}
```

Spring Boot 控制器接收文件上传并调用解析逻辑。此接口使用 `@PostMapping` 注解和 `MultipartFile` 实现文件接收。

```
@GetMapping("/api/jfr/flamegraph")
public ResponseEntity<FlameGraphData> getFlameGraph(
        @RequestParam(required = false) Long startTime,
        @RequestParam(required = false) Long endTime) {
    //过滤指定时间范围内的事件并构造层次化数据
    FlameGraphData data = flameGraphService.generate(startTime, endTime);
    return ResponseEntity.ok(data);
}
```

2.4 前端设计

- 数据获取--通过调用后端提供的 API 接口，获取 .jfr 解析后的元信息及调用栈原始数据。

```
useEffect(() => {
  fetch('...')
    .then(res => res.json())
    .then(data => setRawData(data));
}, []);
```

- 数据解析--将调用栈的原始数据（如扁平事件列表）转换为层级帧树结构，便于后续布局和渲染。

```
function buildFrameTree(events: Event[]): FrameNode {
  const root: FrameNode = { name: 'root', children: [], value: 0 };
  for (const event of events) {
    let current = root;
    for (const frame of event.stack) {
      let child = current.children.find(c => c.name === frame);
      if (!child) {
        child = { name: frame, children: [], value: 0 };
```

```
        current.children.push(child);
      }
      child.value += 1;
      current = child;
    }
  }
  return root;
}
```

- 布局计算--采用递归方式计算每个帧节点在火焰图中的位置和宽度，依据调用频率进行可视化比例设置。

```
function computeLayout(node: FrameNode, x: number, y: number, width: number) {
  node.x = x;
  node.y = y;
  node.width = width;
  const total = node.children.reduce((sum, c) => sum + c.value, 0);
  let offset = x;
  for (const child of node.children) {
    const w = (child.value / total) * width;
    computeLayout(child, offset, y + FRAME_HEIGHT, w);
    offset += w;
  }
}
```

3、规划

- **阶段一：研究 JFR 文件结构与解析 API（7.1 - 7.15）**
  - **第一周**：
    - 阅读 JDK 官方 JFR API（`jdk.jfr.consumer.RecordingFile` 等）文档与示例，理解 `.jfr` 文件的整体结构与事件模型；
    - 尝试使用 API 读取并打印 `jdk.ExecutionSample`、`jdk.ObjectAllocationInNewTLAB` 等核心事件。
  - **第二周**：
    - 实现初步的 JFR 文件事件提取工具，支持事件类型过滤、基本聚合（按事件名/线程/时间段）；
    - 调研 chrishantha/jfr-flame-graph、Eclipse Jifa 等开源工具解析方式。
- **阶段二：实现 JFR 后端解析服务（7.16 - 8.7）**
  - **第三周、第四周**：
    - 搭建 Spring Boot 后端服务，提供 `.jfr` 文件上传与解析 API；
    - 设计火焰图数据结构（调用栈聚合树）。
  - **第五周**：
    - 实现 `/api/jfr/flamegraph` 接口，返回 JSON 层级结构供前端可视化；
    - 拓展事件汇总接口：支持查询异常、内存、文件 IO 等统计信息。
- **阶段三：实现前端可视化页面（8.8 - 9.1）**
  - **第六周、第七周**：

        - 完成 `.jfr` 上传页面，接入后端文件上传接口；
        - 集成 d3-flame-graph（或 react-d3-flamegraph）库，完成火焰图渲染组件。
    - **第八周**：
        - 实现事件统计表格视图，展示 CPU/内存/IO/异常事件；
        - 实现用户交互：加载状态提示、时间范围过滤等。
- **阶段四：优化联调、撰写总结与提交（9.2 - 9.31）**
    - **第九周、第十周**：
        - 联调前后端数据传输逻辑，优化火焰图展示性能；
        - 处理大文件解析优化（如分段解析、异步处理）。
    - **第十一周、第十二周**：
        - 撰写 README、项目文档与使用说明；
        - 提交最终代码，整理总结报告与 PR/MR。

4、其他

这是我第一次参与大型开源项目，非常感谢 OSPP 提供的宝贵机会。我期待通过此次实践，为开源社区贡献力量，积累项目经验，提升技术能力。同时，我对 Arthas 项目非常感兴趣，平时在日常开发中也经常使用。如果有机会，我希望今后能持续参与开源社区的建设，为其发展贡献更多力量。