

Java 源代码分析工具 —— 实验报告

注：本此实验项目生成的分析报告为本目录下面的 Report-cli.txt 和 Report-math.txt 分别对应考核包中提供的两个 Java Maven 项目。

✧ 一、实验目的

随着软件规模不断扩大，自动化程序分析技术（如 AST 解析、字节码分析、CFG/PDG 构建）在理解代码、提取度量、发现缺陷等方面发挥着重要作用。

本实验旨在开发一个 **基于 JavaParser 的 Java 源代码分析工具**，通过对 Java 项目进行全面的静态分析，提取关键的代码结构和依赖关系信息，并计算代码质量指标，帮助开发者深入了解项目的质量状况，发现潜在的缺陷和优化点，提高代码的可维护性和可扩展性。

✧ 二、实验环境

2.1 硬件环境

- 处理器：AMD Ryzen 7 5800H
- 内存：16GB DDR4
- 硬盘：512GB SSD

2.2 软件环境

- 操作系统：Windows 11
- 开发工具：IntelliJ IDEA 2022.1.2
- JDK 版本：Java 8
- Maven 版本：Maven 3.26.4

✧ 三、技术栈

3.1 JavaParser

JavaParser 是一个用于解析和分析 Java 源代码的开源库，它可以将 Java 源文件转换为 **抽象语法树 (AST)**，方便后续对代码进行深入的分析和处理。通过 JavaParser，我们可以轻松地遍历 AST 节点，提取类、方法、字段等信息，为后续的代码分析提供基础。

3.2 Java 8

Java 8 是 Java 语言中应用广泛的长期支持版本，以其稳定性、成熟生态和 **Lambda** 表达式等新特性著称。本次选择 Java 8 作为开发语言，充分利用其语法糖和类库优势，实现代码分析与报告生成功能。

3.3 Maven

Maven 是一个强大的项目管理和构建工具，它可以自动处理项目的依赖管理、编译、测试和打包等任务。在本项目中，使用 Maven 来管理项目的依赖，确保项目的可重复性和可维护性。

✧ 四、实现思路

4.1 整体架构

项目采用模块化设计，主要由以下几个核心组件组成：

- 1 核心分析器 (**JavaSourceAnalyzer**)
 - 负责对指定目录下的 Java 源代码文件进行解析和分析。
 - 使用 JavaParser 生成 AST，并通过遍历 AST 节点收集类、方法、字段等信息。
 - 实现代码质量度量算法，计算圈复杂度、注释率等指标。
- 2 报告生成器 (**ReportGenerator**)
 - 整合核心分析器的分析结果，生成结构化的分析报告。
 - 报告内容包括类 / 接口汇总、方法统计、类依赖关系和代码度量汇总。
- 3 数据模型

- **ClassInfo**：用于 存储类的信息，包括包名、类名、字段列表、方法列表、注释率。
- **MethodInfo**：用于 存储方法的信息，包括方法名、参数数量、代码行数、圈复杂度。

4.2 关键算法实现

4.2.1 圈复杂度计算

圈复杂度是衡量代码复杂度的一个重要指标，它反映了代码中条件分支的数量。通过统计方法中的条件分支语句（**if**、**for**、**while**、**switch** 等）和 **try-catch** 块的数量来计算圈复杂度。具体实现代码如下：

```
private int calculateCyclomaticComplexity(MethodDeclaration method) {
    AtomicInteger complexity = new AtomicInteger(1); // 基础复杂度为1
    complexity.addAndGet(method.findAll(IfStmt.class).size());
    complexity.addAndGet(method.findAll(ForStmt.class).size());
    complexity.addAndGet(method.findAll(WhileStmt.class).size());
    complexity.addAndGet(method.findAll(DoStmt.class).size());
    complexity.addAndGet(method.findAll(SwitchEntry.class).size());
    complexity.addAndGet(method.findAll(ConditionalExpr.class).size());
    method.getBody().ifPresent(body → {
        complexity.addAndGet(body.findAll(TryStmt.class).stream()
            .mapToInt(tryStmt → tryStmt.getCatchClauses().size())
            .sum());
    });
    return complexity.get();
}
```

4.2.2 注释率计算

注释率是指代码中 **注释行数占总行数的比例**，它反映了代码的可读性和可维护性。通过遍历类中的所有注释，计算注释的行数，并除以类的总行数得到注释率。具体实现代码如下：

```
private void calculateCommentRatio(TypeDeclaration<?> type, ClassInfo
classInfo) {
    List<Comment> comments = type.getAllContainedComments();
    int commentLines = comments.stream()
        .mapToInt(comment -> comment.getEnd().get().line -
comment.getBegin().get().line + 1)
        .sum();
    int totalLines = type.getEnd().get().line - type.getBegin().get().line +
1;
    if (totalLines > 0) {
        classInfo.setCommentRatio((int) Math.round((commentLines * 100.0) /
totalLines));
    }
}
```

4.2.3 依赖关系分析

依赖关系分析是指 **分析类之间的调用关系**，通过遍历方法中的方法调用表达式，解析被调用方法所属的类，并将其添加到依赖关系列表中。具体实现代码如下：

```
private void analyzeMethodCalls(MethodDeclaration method, String
callerFullName, ClassInfo classInfo) {
    method.accept(new VoidVisitorAdapter<Void>() {
        @Override
        public void visit(MethodCallExpr call, Void arg) {
            super.visit(call, arg);
            call.getScope().ifPresent(scope -> {
                String calleeClass = scope.toString();
                classInfo.addDependency(calleeClass);
            });
        }
    }, null);
}
```

4.3 整体功能

4.3.1 源代码解析

使用 JavaParser 解析 Java 源文件，生成 AST，并配置 **ParserConfiguration** 并且保留注释信息。通过访问者模式遍历 AST 节点，收集类、方法、字段等信息。

4.3.2 数据收集

- 类级别：收集包名、类名、接口标识、字段列表等信息。
- 方法级别：收集方法名、参数个数、代码行数、圈复杂度等信息。
- 依赖关系：构建方法调用关系图，记录类之间的依赖关系。

4.3.3 指标计算

- 圈复杂度：统计方法中的条件分支语句数量，计算方法的圈复杂度。
- 注释率：计算类中注释行数占总行数的比例，得到注释率。
- 方法行数：通过 AST 节点的位置信息，计算方法的代码行数。

4.3.4 报告生成

将分析结果以文本格式输出到指定的报告中，分类展示各类统计信息，并可视化展示类调用关系。

✧ 五、实验过程

5.1 项目搭建

使用 Maven 创建项目，并在pom.xml中添加 JavaParser 依赖：

```
<dependencies>
  <dependency>
    <groupId>com.github.javaparser</groupId>
    <artifactId>javaparser-core</artifactId>
    <version>3.26.4</version>
  </dependency>
</dependencies>
```

5.2 项目核心结构

```
analyzer/
├── Application.java           # 应用程序入口
├── core/                     # 核心功能模块
│   ├── JavaSourceAnalyzer.java # 源代码分析器
│   └── ReportGenerator.java    # 报告生成器
├── model/                   # 数据模型
│   ├── ClassInfo.java        # 类信息模型
│   └── MethodInfo.java       # 方法信息模型
└── util/                   # 工具类
    └── FileUtils.java        # 文件操作工具类
```

5.3 核心功能实现

5.3.1 JavaSourceAnalyzer

实现源代码的解析和分析功能，包括计算注释率、收集类的字段和方法信息、分析方法调用关系等。具体实现步骤如下：

- 1 遍历指定目录下的所有 Java 源文件。
- 2 使用 `JavaParser` 解析每个 Java 源文件，生成 AST。
- 3 遍历 AST 节点，收集类、方法、字段等信息。
- 4 计算注释率、圈复杂度等指标。
- 5 分析方法调用关系，构建类的依赖关系图。

5.3.2 ReportGenerator

整合分析结果，生成分析报告。报告内容包括类 / 接口汇总、方法统计、类依赖关系和代码度量汇总等。具体实现步骤如下：

- 1 遍历类信息列表，生成类 / 接口汇总部分。
- 2 遍历类信息列表，生成方法统计部分。
- 3 遍历类信息列表，生成类依赖关系部分。
- 4 计算总类数、总方法数、平均圈复杂度和平均注释率，生成代码度量汇总部分。
- 5 将报告内容写入指定的文件中。

5.4 测试与调试

使用考核包提供的 commons-math/ 和 commons-cli/ 两个 Java Maven 项目对工具进行测试，检查输出报告的准确性和完整性。在测试过程中，发现了一些问题，如注释率计算不准确、依赖关系分析不完整等。通过调试代码，修复了这些问题，确保了工具的正确性和稳定性。

❖ 六、实验结果

6.1 报告内容

生成的报告包含类 / 接口汇总、方法统计、类调用关系和代码度量汇总部分，详细展示了项目的结构、依赖关系和代码质量指标。以下是报告的部分内容：（以 commons-cli 项目为例）

6.1.1 类 / 接口汇总

```
类 org.apache.commons.cli.AlreadySelectedException    [3字段, 2方法, 注释率48%]
类 org.apache.commons.cli.AmbiguousOptionException    [2字段, 2方法, 注释率40%]
接口 org.apache.commons.cli.CommandLineParser         [0字段, 2方法, 注释率87%]
接口 org.apache.commons.cli.Converter                 [8字段, 1方法, 注释率62%]
```

6.1.2 方法统计

```
org.apache.commons.cli.AlreadySelectedException (共2个方法):
该类的圈复杂度: 2          注释率: 48%
  方法: getOption
    参数个数: 0
    代码行数: 3
    圈复杂度: 1

  方法: getOptionGroup
    参数个数: 0
    代码行数: 3
    圈复杂度: 1
```

6.1.3 类调用关系

```
org.apache.commons.cli.AlreadySelectedException 依赖于:
→ org.apache.commons.cli.ParseException
→ org.apache.commons.cli.OptionGroup
→ org.apache.commons.cli.Option
```

6.1.4 代码度量汇总

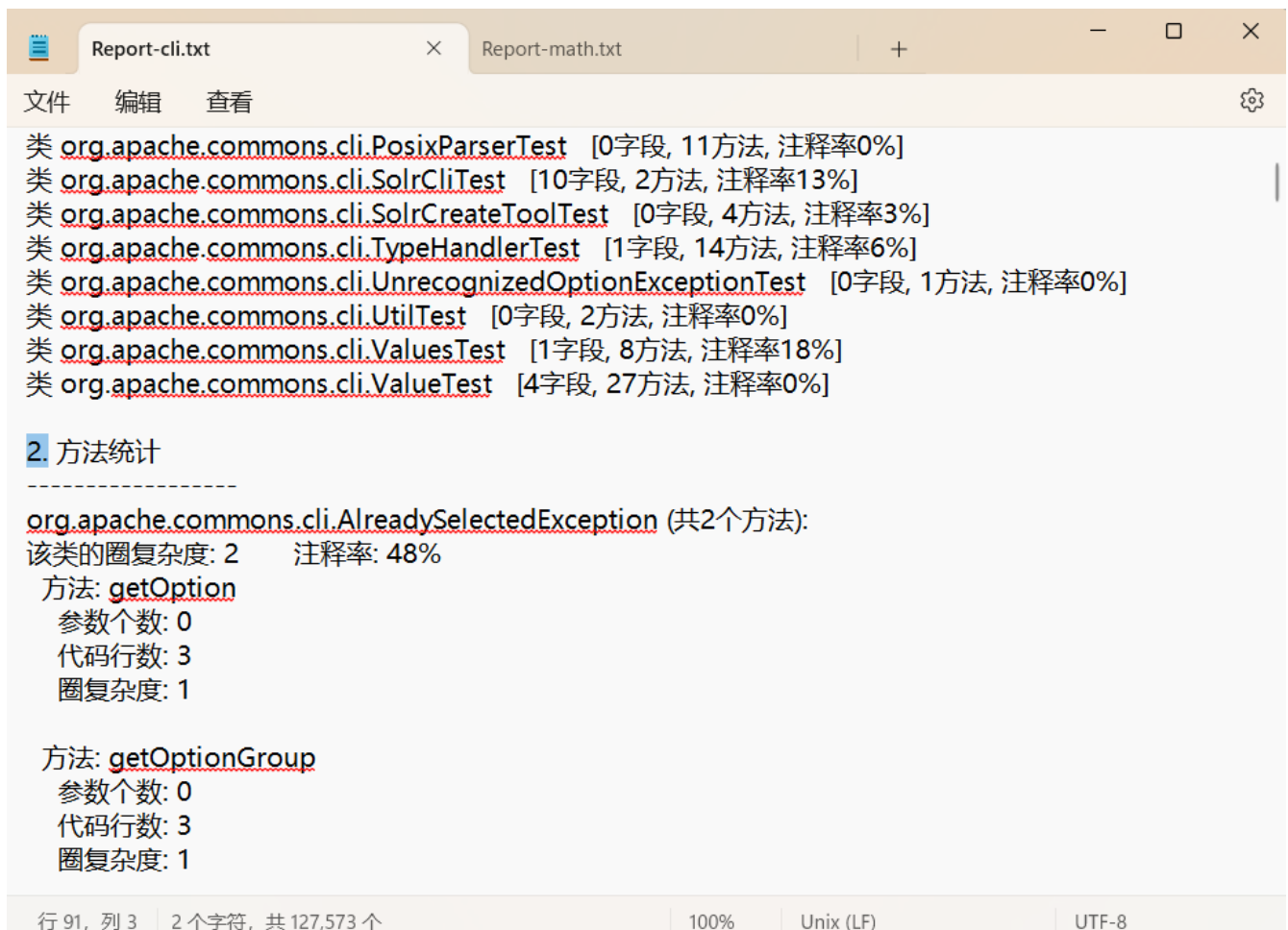
H5 commons-cli 项目:

总类数: 83
总方法数: 865
平均圈复杂度: 1.5
平均注释率: 23.2%

H5 commons-math 项目:

总类数: 1046
总方法数: 8162
平均圈复杂度: 2.1
平均注释率: 28.9%

生成的各项结果: (部分)



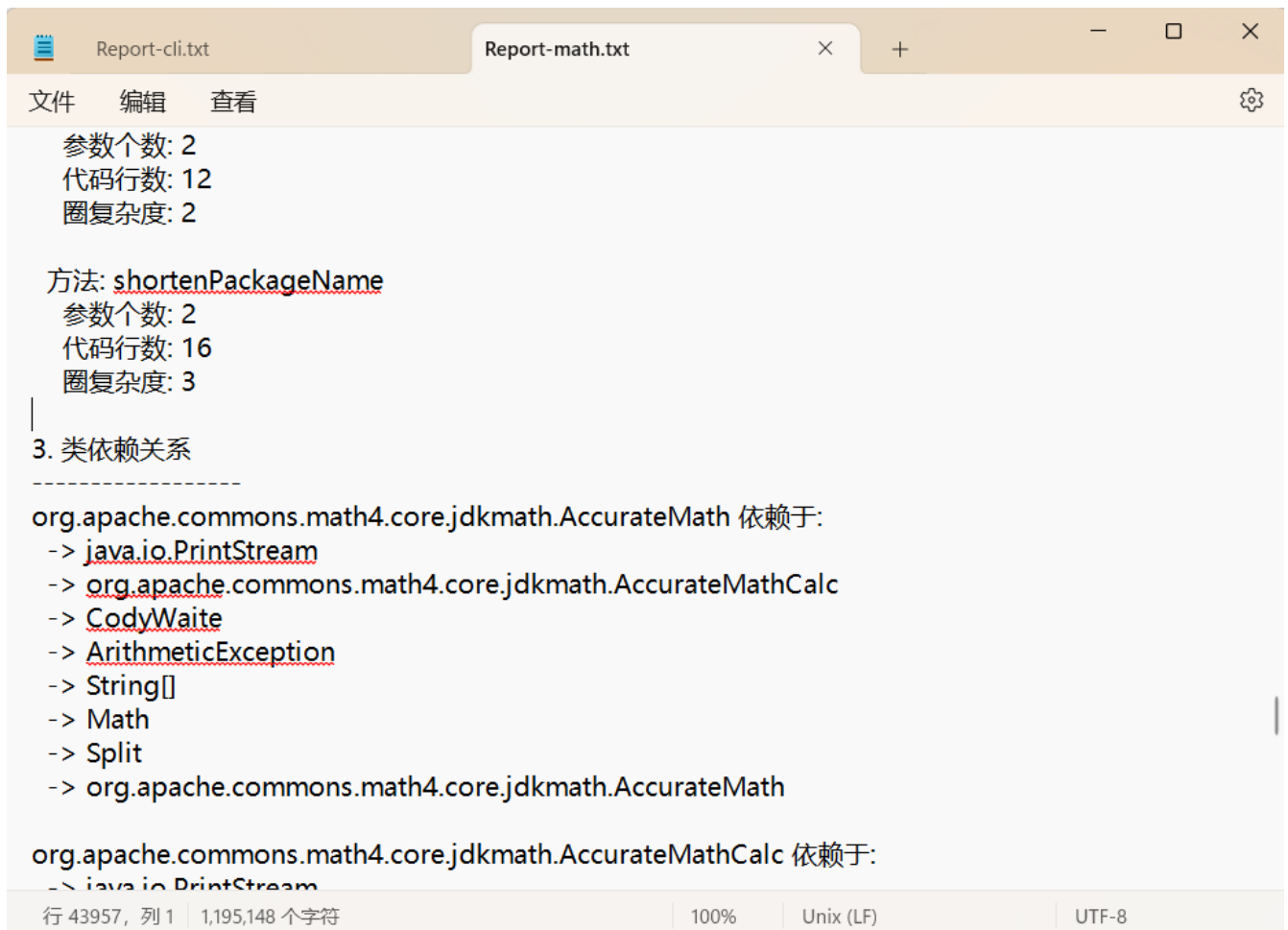
The screenshot shows a code analysis report window with two tabs: 'Report-cli.txt' and 'Report-math.txt'. The 'Report-cli.txt' tab is active, displaying a list of classes and their associated statistics. The classes listed are: `org.apache.commons.cli.PosixParserTest`, `org.apache.commons.cli.SolrCliTest`, `org.apache.commons.cli.SolrCreateToolTest`, `org.apache.commons.cli.TypeHandlerTest`, `org.apache.commons.cli.UnrecognizedOptionExceptionTest`, `org.apache.commons.cli.UtilTest`, `org.apache.commons.cli.ValuesTest`, and `org.apache.commons.cli.ValueTest`. Each class entry includes the number of fields, methods, and the comment rate. Below the class list, there is a section titled '2. 方法统计' (Method Statistics) which details the statistics for the `org.apache.commons.cli.AlreadySelectedException` class, including its circle complexity and comment rate, and then lists the `getOption` and `getOptionGroup` methods with their respective parameters, code lines, and circle complexities. The bottom status bar indicates the current position in the file: '行 91, 列 3 | 2 个字符, 共 127,573 个', along with '100%' zoom, 'Unix (LF)' line endings, and 'UTF-8' encoding.

```
Report-cli.txt × Report-math.txt + - □ ×
文件 编辑 查看 ⚙
类 org.apache.commons.cli.PosixParserTest [0字段, 11方法, 注释率0%]
类 org.apache.commons.cli.SolrCliTest [10字段, 2方法, 注释率13%]
类 org.apache.commons.cli.SolrCreateToolTest [0字段, 4方法, 注释率3%]
类 org.apache.commons.cli.TypeHandlerTest [1字段, 14方法, 注释率6%]
类 org.apache.commons.cli.UnrecognizedOptionExceptionTest [0字段, 1方法, 注释率0%]
类 org.apache.commons.cli.UtilTest [0字段, 2方法, 注释率0%]
类 org.apache.commons.cli.ValuesTest [1字段, 8方法, 注释率18%]
类 org.apache.commons.cli.ValueTest [4字段, 27方法, 注释率0%]

2. 方法统计
-----
org.apache.commons.cli.AlreadySelectedException (共2个方法):
该类的圈复杂度: 2 注释率: 48%
方法: getOption
  参数个数: 0
  代码行数: 3
  圈复杂度: 1

方法: getOptionGroup
  参数个数: 0
  代码行数: 3
  圈复杂度: 1

行 91, 列 3 | 2 个字符, 共 127,573 个 | 100% | Unix (LF) | UTF-8
```

```
Report-cli.txt | Report-math.txt
文件 编辑 查看
参数个数: 2
代码行数: 12
圈复杂度: 2

方法: shortenPackageName
参数个数: 2
代码行数: 16
圈复杂度: 3

3. 类依赖关系
-----
org.apache.commons.math4.core.jdkmath.AccurateMath 依赖于:
-> java.io.PrintStream
-> org.apache.commons.math4.core.jdkmath.AccurateMathCalc
-> CodyWaite
-> ArithmeticException
-> String[]
-> Math
-> Split
-> org.apache.commons.math4.core.jdkmath.AccurateMath

org.apache.commons.math4.core.jdkmath.AccurateMathCalc 依赖于:
-> java.io.PrintStream
行 43957, 列 1 | 1,195,148 个字符 | 100% | Unix (LF) | UTF-8
```

6.2 结果分析

6.2.1 commons-cli 项目分析:

1 项目整体结构与规模:

commons-cli 项目 包含 **83** 个类 和 **865** 个方法，表明该项目有一定的规模，功能模块划分相对细致。平均每个类约 10.4 个方法，反映出部分类承担了过多职责，需关注单一职责原则的遵守情况。

2 代码复杂度分析:

平均圈复杂度为 **1.5**，整体处于较低水平，说明大部分方法逻辑简单，结构清晰。

3 代码可读性与可维护性:

- 平均注释率 23.2%，略高于行业基准（10% - 20%），可能存在少量过度注释。
- 可能是因为代码质量偏低，从而需要增加注释弥补可读性，也可能是因为冗余注释较多而导致的。

6.2.2 commons-math 项目分析:

1 项目整体结构与规模

- **commons-math 项目** 包含 1046 个类 和 8162 个方法，表明这是一个中大型 Java 项目，功能模块复杂且代码量较大。
- 平均每个类约 7.8 个方法，模块划分相对合理，但需验证是否满足单一职责原则，保证项目的维护成本。

2 代码复杂度分析：

平均圈复杂度 **2.1**，整体复杂度较低，说明大部分方法逻辑简单，符合“小方法”设计原则。

3 代码可读性与可维护性：

- 平均注释率 **28.9%**，明显偏高。
- 可能存在的问题：
 - 可能 IDE 自动生成的模板注释未精简。
 - 可能是因为代码质量低从而需要大量注释弥补可读性。
 - 也可能是冗余注释多，重复解释单一逻辑。

✧ 七、实现难点与解决方案

7.1 AST 解析与遍历

难点

JavaParser 生成的 **AST** 结构复杂，节点类型比较多，需要准确识别各类 Java 语法结构，且保留和统计注释信息。

解决方案

使用访问者模式遍历 **AST**，针对不同类型的节点实现专门的访问方法。同时，配置 **ParserConfiguration** 保留注释信息，保证注释率的准确计算。

7.2 圈复杂度计算

难点

需要识别所有可能增加复杂度的控制结构，包括嵌套结构和异常处理块的统计，同时要考虑不同 Java 语法的处理。

解决方案

实现专门的复杂度计算方法，统计 `if`、`for`、`while`、`switch` 等控制结构，并考虑 `try - catch` 块的影响。通过遍历 AST 节点，准确计算方法的圈复杂度。

7.3 类名解析

难点

在Java代码，通常使用的是简单类名，导致很难从源代码中识别出一个类引用了哪些其他类。

解决方案

- 1 创建两级映射：一个是从简单类名到包名的映射（`classToPackageMap`），另一个是从 `import` 语句中提取的映射（`importedClasses`）
- 2 实现 `resolveFullClassName` 方法，先查找 `import` 语句中的映射，再查找全局类名映射表
- 3 采用两轮扫描策略：第一轮建立映射关系，第二轮进行具体分析

```
private String resolveFullClassName(String simpleName, Map<String, String>
importedClasses) {
    // 从import语句中找到完整类名
    if (importedClasses.containsKey(simpleName)) {
        return importedClasses.get(simpleName);
    }

    // 从全局类名映射表中找到包名
    if (classToPackageMap.containsKey(simpleName)) {
        String packageName = classToPackageMap.get(simpleName);
        return packageName.isEmpty() ? simpleName : packageName + "." +
simpleName;
    }
}
```

✧ 八、总结与展望

8.1 总结

通过本次实验，我站在软件安全的角度思考，学到了很多知识。静态分析工具为我们提供了一种有效的手段来发现软件中的安全漏洞，深刻认识到自动化程序分析技术（如 AST 解析）在软件安全中的重要性。静态分析工具可以在代码编写阶段或代码审查过程中，快速地对大量代码进行扫描，发现潜在的安全漏洞。在分析 commons - cli 和 commons - math 项目时，工具能够对代码进行全面的检查，通过分析生成的各项指标，可以帮助我们提前发现可能存在的安全隐患，避免在软件发布后才发现问题，从而降低修复成本。

在未来的学习中，我将继续加强对这方面的学习，提高自己在网络安全方面的技能，为开发更加安全可靠的软件贡献自己的力量。我也希望这个项目在此基础上能够不断优化和完善，加入更多的代码质量指标和可视化展现形式，为软件开发提供更强大的保障。

8.2 展望

未来，该工具可以继续优化，具体方向如下：

- 1 提高工具的处理速度，特别是对于大型项目分析。
- 2 支持更多的代码质量指标，并实现依赖关系可视化报告的生成。
- 3 支持更多的编程语言，可针对不同语言的项目进行分析。
- 4 添加图形化界面，提高与用户之间的交互性。

✧ 附录：部分学习笔记

- 1 配置 ParserConfiguration:

```
ParserConfiguration config = new ParserConfiguration();
config.setAttributeComments(true); // 保留注释信息
JavaParser javaParser = new JavaParser(config);
```

- 2 遍历 AST 节点

```
for (TypeDeclaration<?> type : cu.getTypes()) { }
```

JavaParser常用的解析方法

- 1 解析整个 Java 文件（Compilation Unit）：

```
ParseResult<CompilationUnit> result = StaticJavaParser.parse(file);
```

2 解析代码块 (**Block**) :

```
ParseResult<BlockStmt> result = StaticJavaParser.parseBlock(code);
```

3 解析表达式 (**Expression**) :

```
ParseResult<Statement> result =  
StaticJavaParser.parseStatement(statement);
```

4 解析语句 (**Statement**) :

```
ParseResult<Statement> result =  
StaticJavaParser.parseStatement(statement);
```

5 解析类型:

```
ParseResult<Type> result = StaticJavaParser.parseType(type);
```

6 解析名称:

```
ParseResult<Name> result = StaticJavaParser.parseName(name);
```

7 解析 **import** 语句:

```
ParseResult<ImportDeclaration> result =  
StaticJavaParser.parseImport(importDeclaration);
```

8 解析注释 (**Comment**) :

```
ParseResult<Comment> result = StaticJavaParser.parseComment(comment);
```

可以通过 `staticJavaParser` 这个类的静态方法直接调用以上函数:

```
ParseResult<T> result = // 执行解析操作, 获取解析结果  
  
if (result.isSuccessful()) {  
    // 解析成功  
    T parsedObject = result.getResult();  
    // 对解析得到的对象进行操作  
} else {  
    // 解析失败  
    List<Problem> problems = result.getProblems();  
    // 处理解析过程中遇到的问题  
}
```

