Java Project Analyser

Course Project for Java EE, Tongji University

Team Leader: **Zhang Yao**
zhang@cinea.cc

Tongji University

Table of Contents

Overview 3
- Team Members 3
- Description 3
- Features 3

Usage 4
- Requirement 4
- Compile our project 4
- Simple usage 4
- Open a project 5
- Query method invocation relationships 5
- Query method parameter sources 6
- Automaticly fixing syntax errors 7

Design 7
- System architecture and components 7
- General workflow 7

Implementation 11
- Major data structures 11
- Efficiency considerations 14
- Key Techniques 17
- External libraries 18

Test 19
- Test Cases 19
- Functionality Test 19

Conclusion 22

Overview

Team Members

Name	Matr. No.	Contact number	Email address
Zhang Yao	2152955	+86 13518772062	zhang@cinea.cc
Tang Shuhan	2153877	+86 15300862015	2405353202@qq.com
Wang Shuyu	2151894	+86 17709584390	1491456253@qq.com
Tang Kexian	2152240	+86 18707016886	1135530278@qq.com

No specific order in the table.

Description

This project offers analysis functionality for Java projects. Users can conveniently obtain the call relationship of specified methods in a project using the software provided by this project. This includes a list of methods that a given method calls, as well as a list of methods that call the given method. The software can also trace call chains to a specified depth. Additionally, it provides tracking for the sources of function arguments within a method, assisting users in locating the origins of these arguments. If the arguments are provided in the form of literals, the software will also directly provide their values.

Features

Here is a feature list which is implemented by our team:

- REPL-style command-line interaction
- Import and organize Java projects according to package structure
- Import large-scale Java projects within an acceptable timeframe
- Friendly progress bar when loading large Java project
- Check for the presence of syntax errors when importing projects
- Parallelize reading, importing, and analyzing projects
- Provide user-friendly output for time-consuming operations
- List all files, classes and interfaces, and methods in the project
- List all instances of a specific class within the project
- Distinguish different function overloads and prompt users to make a selection in cases where overloads exist.
- Find all other methods called by a specified method
- Identify all instances where a specific method is called within the project
- Trace all methods that could potentially call a specified method
- Locate and trace all possible argument sources for a specified method and provide values for literal argument sources.

Usage

Requirement

A Java environment that is **greater than or equal to 17 is required**. We use **Amazon Corretto 17 LTS** in our development and test. But other JDK distribution is also compatiable. Learn more about Amazon Corretto: https://aws.amazon.com/corretto

Compile our project

We use **Maven** to manage our project and its dependencies. A **Maven Wrapper** instance is also included in our project repository. You can just using it to compile our project in most cases. Learn more about Maven Wrapper: https://maven.apache.org/wrapper/

We suggest checking your JDK version before compiling our project:

```
${JAVA_HOME}/bin/java -version    # Never continue if not 17 or higher.
```

To start the compilation process, please execute the following command:

```
./mvnw -B dependency:go-offline
./mvnw -B package
```

The first command will try to download all dependencies our project used from the Maven Central Repository. You may use Aliyun Maven Central Repository in China to accelerate the downloading process. Learn more about Aliyun Maven Central Repository: https://developer.aliyun.com/mvn/guide.

After finishing the compilation, you may find the artifact in `target` directory, which looks like `java-project-analyser-cli-0.0.1-SNAPSHOT.jar`. You may move the output to a convenient location and change its name as needed. Assuming we have renamed it to `analyser.jar` in convience in the following parts.

Simple usage

Just use `java -jar` command to launch our project and use it:

```
java -jar ./analyser.jar
```

Our project provides a REPL user interface, in which you can easily input commands, obtain results, and reuse the previous state instead of having to restart the program after each command execution. Our project utilizes the **Spring Shell** framework to provide you with an exceptional interactive command-line experience. Learn more about Spring Shell: https://spring.io/projects/spring-shell.

Enter `help` to get a list of functionalities provided by our project.

```
shell:>help
```

Open a project

Important

Our project does not support libraries that generate and process code at compile-time, such as **Lombok**.

If you want to analyse a project which uses Lombok, consider **Delombok** it manually before analysing. Learn more about the Delombok feature: https://projectlombok.org/features/delombok

Use `open` command to open, load and analyse a Java project:

```
shell:>open /usr/src/sample-project/src/main/java
```

Tips:

- The path should contain the root package, but not the root package self.
- On Windows, please double the backslashes in the path or change them to forward slashes. For example, both `C:/commons-lang/src/main/java` and `C:\\commons-lang\\src\\main\\java` are correct.
- The process of opening the project should be completed within one second, and it should not exceed ten seconds at most. In our test, open a huge project with 200 classes and 3000 methods usually cost about 2 seconds.

Example output:

```
shell:>open D:\\IdeaProjects\\java-project-analyser-sample\\src\\main\\java
Resolving 100% [============================] 1/1 (0:00:00 / 0:00:00)
Indexing 100% [============================] 3/3 (0:00:00 / 0:00:00) \

Open project success:
                      1 packages found.
                      1 files found.
                      1 classes and interfaces found.
                      3 methods found.
                      213ms used.
```

Query method invocation relationships

Use `func` command to query the method invocation relationships of a specified method:

```
shell:>func <class reference> <method name> <optional: searching deepth>
```

For example:

```
shell:>func main.Test sayHello
```

Example output:

```
shell:> func main.Test sayHello
Please wait.
******************** Invokes *******************
  -> java.io.PrintStream.println(java.lang.String)
************************************************

****************** Invoked by ******************
  <- main.Test.introduction(java.lang.String)
      <- main.Test.main(java.lang.String[])
  <- main.Test.main(java.lang.String[])
************************************************



Time cost:
    Get Invokes: 3 ms
    Get Invoked: 1 ms
shell:>
```

Some times you may want to specify a method from its overloads. The program will distinguish this situation and ask you to choose the method you need:

```
shell:>func org.apache.commons.lang3.StringUtils split
***************
 1) org.apache.commons.lang3.StringUtils.split(String)
 2) org.apache.commons.lang3.StringUtils.split(String, String)
 3) org.apache.commons.lang3.StringUtils.split(String, String, int)
 4) org.apache.commons.lang3.StringUtils.split(String, char)
***************
Please choose one:
```

Query method parameter sources

Use `param` command to query the parameter sources of a specified method:

```
shell:>param <class reference> <method name> <optional: searching deepth>
```

For example:

```
shell:>func main.Test sayHello
```

Example output:

```
shell:>param main.Test sayHello
Please wait.
********************Parameter Origin*******************
  String name:
        name1: (main.Test) introduction
         <- "Odie": (main.Test) main
        "Garfield": (main.Test) introduction
        "Jon": (main.Test) sayHello
*******************************************************

Time cost: 4 ms
shell:>
```

Automaticly fixing syntax errors

to be implemented

Design

System architecture and components

Our system is built on top of the Spring Boot framework. We utilize the **Spring Framework** to organize and manage the lifecycle of Beans and dependency injection. We employ Spring Shell to create an easy-to-use interactive command-line interface.

Specifically, our system is mainly divided into the following parts:

- **Service interfaces and their implementations**, which are responsible for providing the core functions of the project, such as importing the project, parsing the method call relationships, and tracing back the sources of parameters.
- **Model classes**, which are the data structures used by the core business of the project. They abstract and model business information and store this information.
- **Helper classes**, which are tools used during the execution of the project's core business. They operate without any state, provide only static services, and are not instantiated into class instances.
- DTOs (**Data Transfer Objects**), which are the data structures used for interaction between the core business classes and helper classes. These data structures don't store business-related models but are responsible solely for pure information transfer.

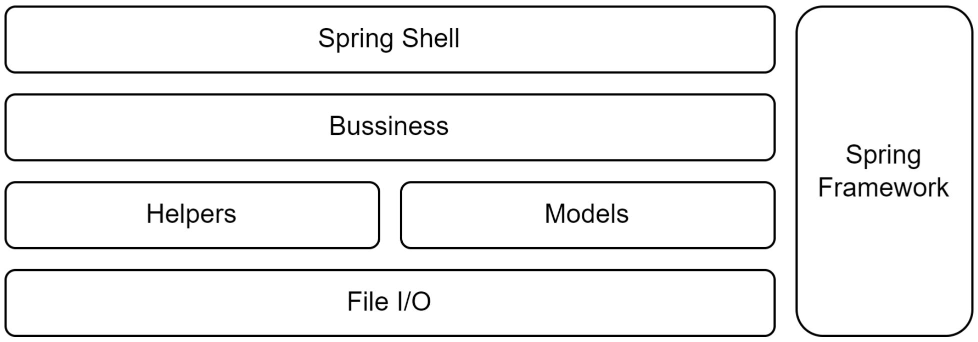


Figure 1: System architecture

General workflow

We have the control and data flow as below:

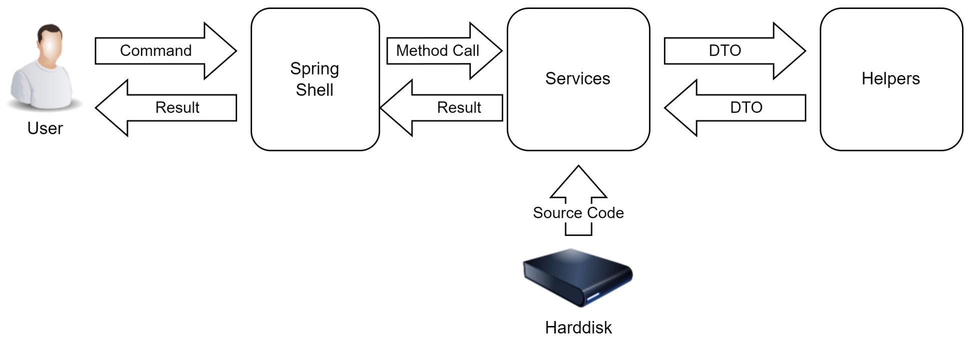
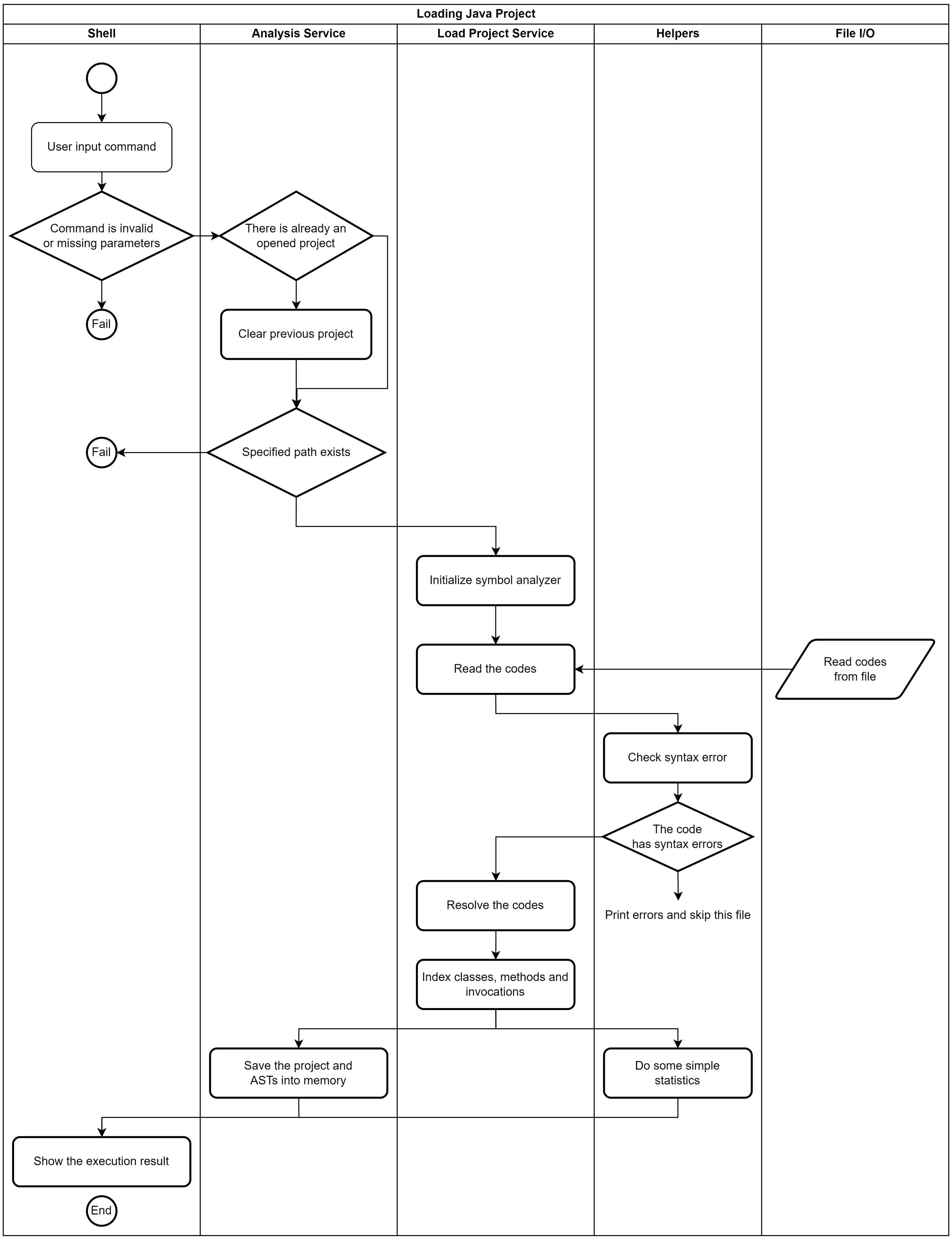


Figure 2: System control and data flow

Project loading workflow

Our project offers a systematic approach to analyze Java projects by reading, resolving, indexing, and performing statistical operations on the code while providing feedback on syntax errors.



Loading Java Project
Shell
Analysis Service
Load Project Service
Helpers
File I/O
User input command
Command is invalid or missing parameters
Fail
There is already an opened project
Clear previous project
Specified path exists
Fail
Initialize symbol analyzer
Read the codes
Read codes from file
Check syntax error
The code has syntax errors
Print errors and skip this file
Resolve the codes
Index classes, methods and invocations
Save the project and ASTs into memory
Do some simple statistics
Show the execution result
End

Figure 3: Project loading workflow

Methods invocation relationship querying workflow

This workflow provides a systematic method for users to query relationships between methods, identifying both the invoking methods and those that are invoked. The results are then organized and presented in a user-friendly manner.

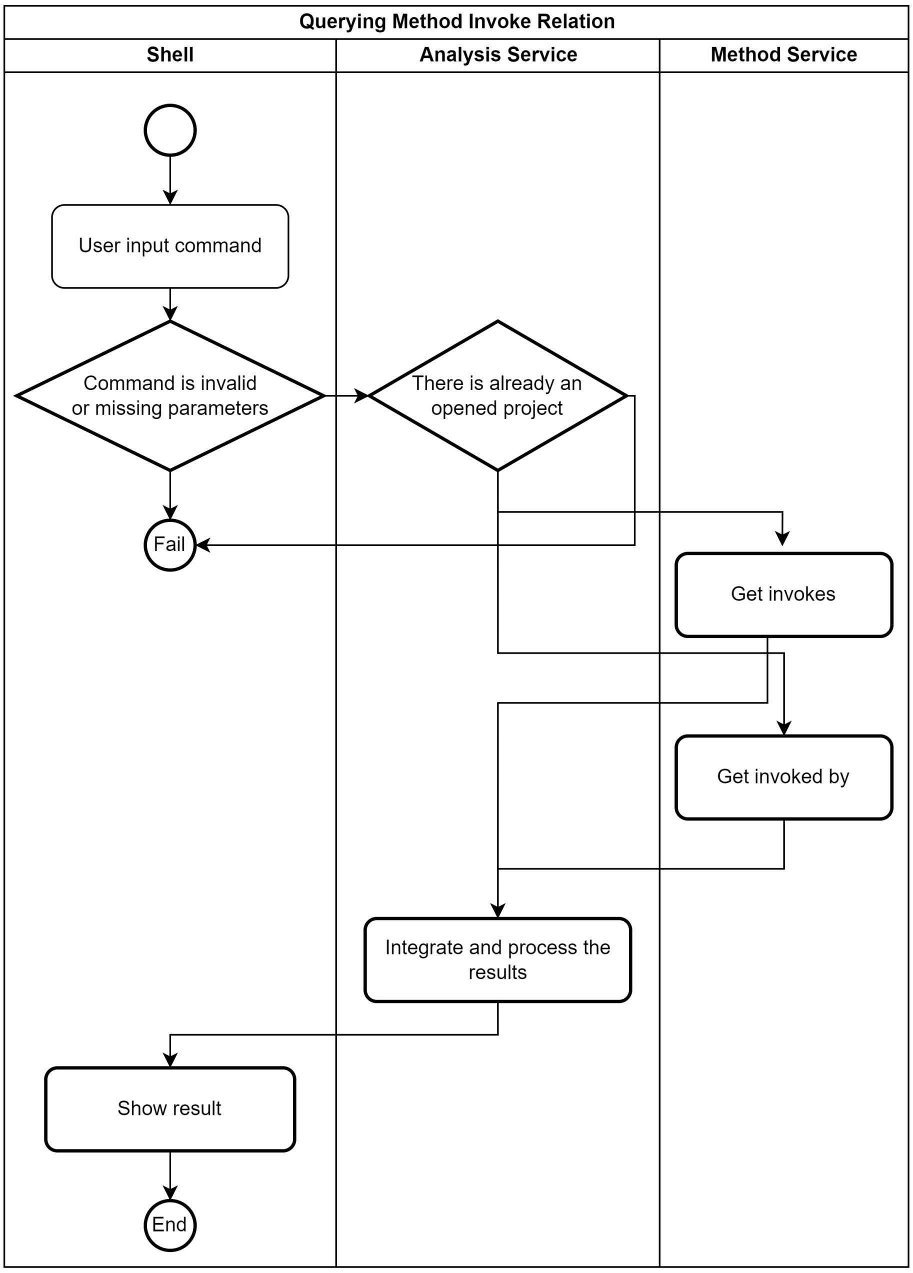

Querying Method Invoke Relation
Shell
Analysis Service
Method Service
User input command
Command is invalid or missing parameters
There is already an opened project
Fail
Get invokes
Get invoked by
Integrate and process the results
Show result
End

Figure 4: Methods invocation relationship querying workflow

Methods parameters origin querying workflow

This workflow furnishes users with a structured means of querying the origins of method parameters, elucidating both the parameters themselves and their respective sources. These findings are then systematically arranged and exhibited for user comprehension.

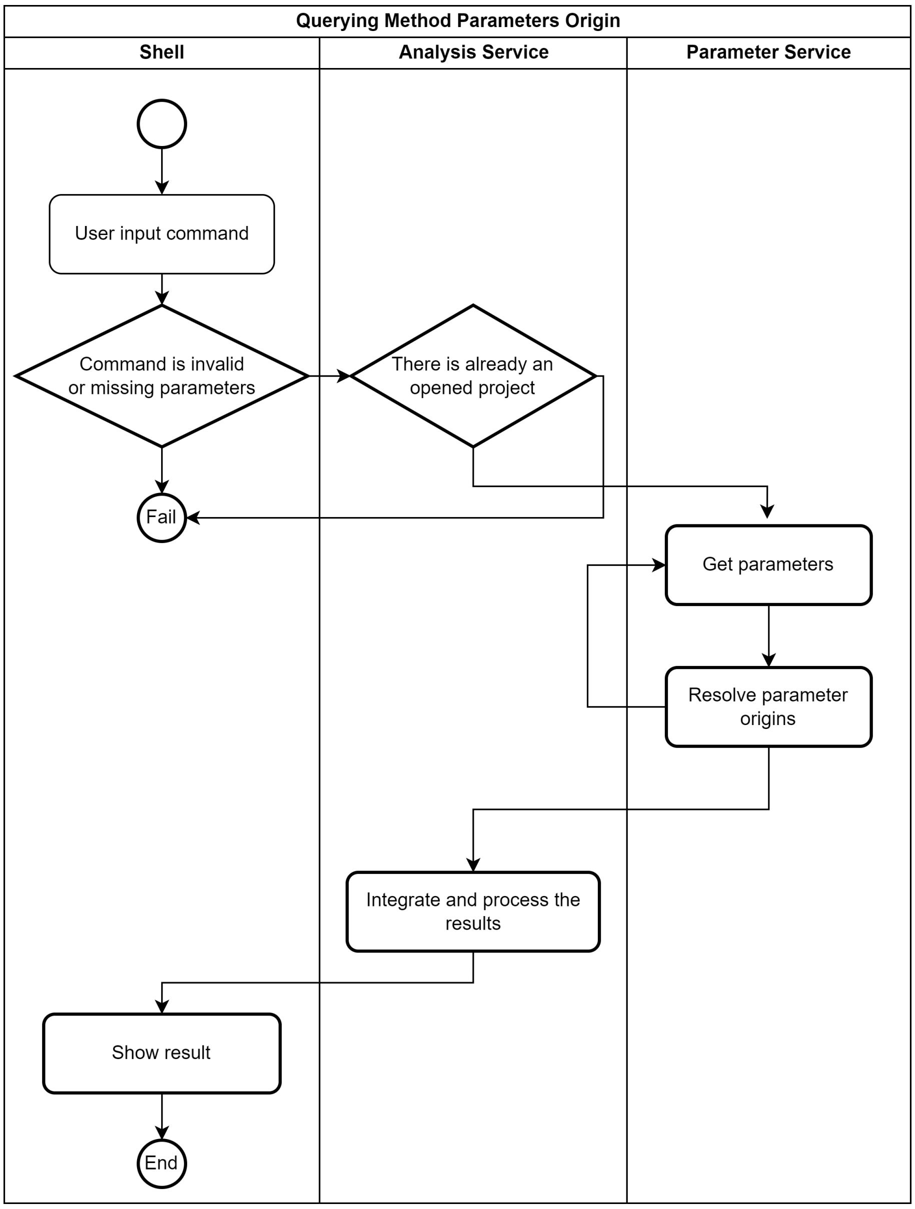

Querying Method Parameters Origin
Shell
Analysis Service
Parameter Service
User input command
Command is invalid or missing parameters
There is already an opened project
Fail
Get parameters
Resolve parameter origins
Integrate and process the results
Show result
End

Figure 5: Methods parameters origin querying workflow

Implementation

Major data structures

Abstracting a Java project is a complex task because Java projects are hierarchically composed of packages, files, classes and interfaces, methods, and numerous nested expressions and statements. Moreover, to enhance the efficiency of the program's execution, we need to store and index commonly used metadata, such as the classes and methods within the project.

This is because if every time we need to access a specific class and have to search for it and its corresponding file level by level according to the package name, the program's execution efficiency would be terribly poor.

Based on our practical requirements and for the sake of utility and efficiency, we divide a Java project into five levels. Each level is represented and stored by a unique Java class and holds the data for the next level. They are: Java Project, Java Package, Java File, Java Class and Java Method. Among them, the levels of Project, Class, and Method are the most commonly used, while the other two levels mainly serve to maintain the tree structure of the project.

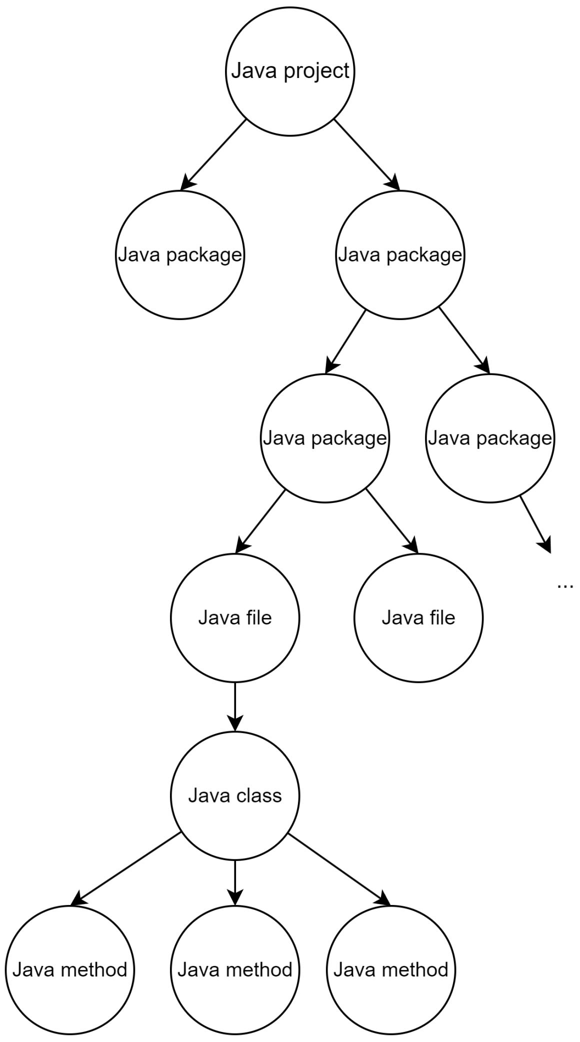


Figure 6: System models

Storing the Java project

While storing the project, we also saved some pre-loaded information to enhance the program's performance. This information includes: Java classes indexed with hash tables, Java methods indexed with compressed tries, and function call graphs indexed with hash tables. We will discuss about this later.

```
@Data
public class JavaProject {
    private List<JavaPackage> packages;

    private Map<String, JavaClass> classMap;

    private PatriciaTrie<JavaMethod> methodTrie;
    private Map<String,List<MethodCallExpr>> invokedRelation;
}
```

Children models

We save the AST node generated by JavaParser, and commonly used metadata in our models.

Java Method

```
@Data
public class JavaMethod {
    /**
     * AST node
     */
    private MethodDeclaration declaration;

    private String name;

    private String qualifiedSignature;

    private String classReference;

    private JavaClass javaClass;
}
```

Java Class

```
@Data
public class JavaClass {
    /**
     * AST node
     */
    private ClassOrInterfaceDeclaration declaration;

    private String classReference;

    private List<JavaMethod> methods;

    private JavaFile javaFile;
}
```

Java File

```
@Data
public class JavaFile {
    /**
     * AST root
     */
    private CompilationUnit compilationUnit;

    private String path;

    private String fileName;

    private String packageName;

    private List<JavaClass> classes;
}
```

Java Package

```
@Data
public class JavaPackage {
    private List<JavaFile> files;

    private List<JavaPackage> childrenPackages;

    private JavaPackage parentPackage;
}
```

Efficiency considerations

Parallelized file system operations

File system I/O operations are often performance bottlenecks in system execution. They frequently require waiting for the operating system to process and load files. During this waiting period, the current process remains idle. To enhance the program's runtime efficiency, we can leverage multithreading technology, allowing other threads of our program to continue running while one is waiting, thus fully tapping into potential performance.

Thread pooling is the solution we've chosen. Before starting to read a Java project, we prepare a thread pool of an appropriate size. When the program encounters a Java source code file, it loads the reading task into a new thread and places it in the thread pool. The JVM will properly schedule the execution of these threads, handing over the CPU to other threads when one awaits I/O.

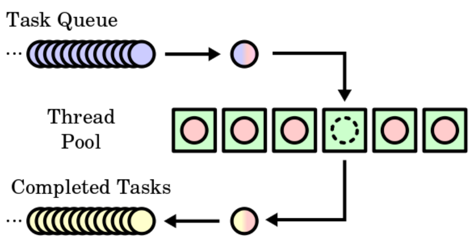


Figure 7: Thread pool[1]

[1]https://en.wikipedia.org/wiki/Thread_pool

Executor service is a java API that simplifies of running tasks in asynchronous fashion. ExecutorService provide API to supply tasks and numbers of threads, threads picks up task from the Queue and execute them. Additionally, compared to regular multithreading, threads in a Java thread pool are reused, thereby avoiding the overhead of creating and destroying threads. This reuse allows for more efficient use of system resources.

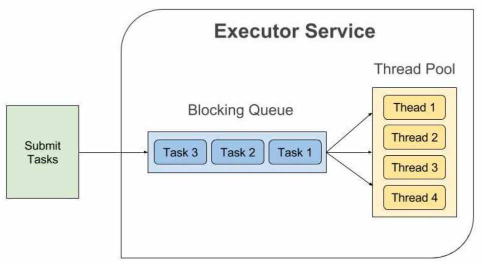


Figure 8: Executor service[2]

Code excerpt from the project:

```
ExecutorService executor = Executors.newFixedThreadPool(4);

var latch = new CountDownLatch(fileNum.get());

FileSystemHelper.traverseDirectories(startDir, file -> {
    if (file.getName().matches(".*\\.java")) {
        executor.submit(() -> {
            try {
                var javaFile = loadFile(file);
            } catch (Exception e) {
                throw new RuntimeException(e);
            } finally {
                latch.countDown();
            }
        });
    }
});

try {
    latch.await();
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
```

Using Compressed Trie to Accelerate Method Overload Matching

Our project supports matching overloaded methods. This means that if a user inputs an identifier for a class and the name of a method within it, our project should match the method name to the full method signature, and prompt the user to make a choice when multiple method signatures (i.e., overloads) are present. This means we need to match the prefix of a key from the index. Regular HashMaps and TreeMaps do not support such operations; to

[2]https://medium.com/@cse.soumya/executor-service-java-b3dc91853140

achieve this functionality, it would be necessary to traverse all keys in the tree. Fortunately, we have the **trie** algorithm, which allows us to match all key-value pairs starting with a certain prefix in linear time.

However, a regular trie still has some limitations. For instance, the method signatures processed in our project might share common prefixes, such as org.apache.commons-lang. This means that when searching for a method in this package using the trie, one would needlessly recurse 24 times first – an approach that is neither elegant nor efficient. Put in writing, the tree degenerates into a linked list, which doesn't leverage the performance advantages of tree structures. Therefore, we opted for a **compressed trie** to circumvent this issue. The compressed trie compresses single chains in the path into one node, eliminating single chains in the tree structure and enhancing traversal efficiency.

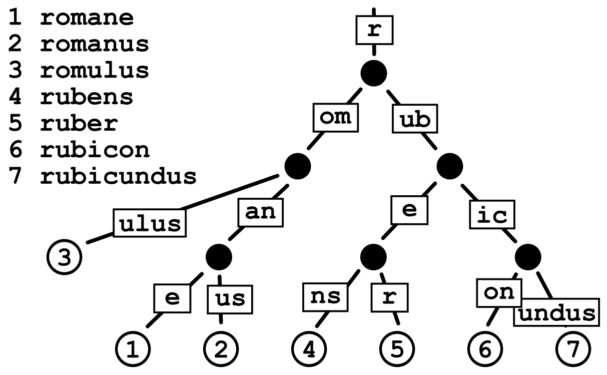


Figure 9: An example of a radix tree[3]

Patricia Trie is an implementation of the compressed trie. Our project uses the Patricia Trie implementation provided by Apache Commons Collections as our trie implementation.

Code excerpt from the project:

```java
/**
 * Find all overloads of a specified method.
 */
@Override
public List<String> getAllOverloads(JavaProject project, String
methodReference) {
    var searchString = methodReference + '(';
    var searchResult = project.getMethodTrie().prefixMap(searchString);

    return new ArrayList<>(searchResult.keySet());
}
```

Using Pre-Cached Method Call Graphs to Accelerate Querying

During the previous project importing process, we stored the AST built by JavaParser into our data structures. However, if there's a frequent need to query the relationships between methods, which is exactly what our project is doing, traversing the entire AST every time to determine where and how a particular method is called by which functions becomes inefficient. Therefore, pre-caching a method call graph becomes essential.

[3]https://en.wikipedia.org/wiki/Radix_tree

Designing the data structure for the method call graph is also a matter of careful consideration. Clearly, there's no need to cache which methods a particular method calls, as this can be determined in linear time by examining a small amount of code; what we really need to cache is where the method is called from. What type of data to cache also requires thought. The cached information needs to represent the call relationship comprehensively – that includes the caller, calling parameters, code location, and call content – while occupying as little space as possible. In our view, storing a reference to the AST node of the method call expression is an excellent choice. On one hand, this tree node is already stored in the project tree structure, so adding a reference to it doesn't introduce significant storage overhead to the system. On the other hand, this tree node can provide a wealth of information, and by traversing upward, it can reveal the method and class in which it resides. In conclusion, we use the AST node of the call expression as part of our method call graph.

In our tests, introducing this optimization significantly improved the runtime of the program. Below is a comparison of the time taken before and after implementing this optimization. The project used for testing was Apache Commons Lang. The command we used for testing was `func org.apache.commons.lang3.StringUtils isBlank` and `param org.apache.commons.lang3.StringUtils isBlank`.

Operation	**Before caching**	**After caching**
Project Loading	1806 ms	9676 ms
Methods Invocation Querying	10141 ms	2 ms
Parameters Origin Querying	13883 ms	37 ms

Table 1: Test result for the pre-cached graph's acceleration

Key Techniques

Abstract Syntax Tree (AST) Parsing

An Abstract Syntax Tree (AST) serves as a pivotal concept in the realm of programming and compiler design. Essentially, an AST is a hierarchical tree structure that encapsulates the abstract syntactic structure of a given source code written in a programming language. This tree doesn't just hold the code, but rather interprets its deeper meaning, breaking it down into its core components.

At each level of the AST, nodes are present that denote specific constructs encountered within the source code. For instance, in a mathematical operation like 'a + b', the '+' operation would be the parent node, with 'a' and 'b' as its child nodes. The term "abstract" in its naming suggests that this tree doesn't mimic the code verbatim. Instead, it filters out any redundant or non-essential information, focusing exclusively on the structural or content-centric details that are paramount for interpretation and analysis.

In our project, the AST parsing is facilitated by JavaParser. JavaParser isn't just another tool, but rather a comprehensive suite of open-source utilities meticulously crafted for the analysis, transformation, and generation of Java code. One of its standout features is its capability to produce an in-depth AST of any Java code fed into it. This not only aids in understanding the code structure but also in pinpointing specific patterns and nuances within the code.

Furthermore, JavaParser isn't limited to just parsing. It brings to the table an advanced capability to resolve symbols present in the code. This feature is instrumental for developers as it paves the way for a clearer understanding of the code's semantics. Resolving symbols can be likened to deciphering a puzzle, where each piece (symbol) is put in its rightful place, making the entire picture (code) clear and coherent. By leveraging JavaParser's symbol resolution, we can substantially diminish the complexities developers face, allowing them to focus on the more intricate aspects of code development and optimization.

External libraries

In our project, numerous outstanding third-party Java open-source libraries are utilized to enhance the system's functionality, improve its stability, and reduce repetitive tasks during development. Their specific details are shown in the table below:

Name	Role
spring Spring Framework	Provides reliable **inversion of control (IoC)** and **aspect-oriented programming (AOP)** support to our project.
spring boot	Simplify the process of building production-ready project, and provides **rapid development** support to our project.
Spring Shell	Facilitate the building of **interactive command-line** applications like our project.
JavaParser	Provides **source code parsing** and **symbols resolving** support to our project.

Project Lombok	Provides annotations to **reduce boilerplate code** in our project and generate data classes, builder pattern and logging.
GUAVA™	Provides utility methods and classes to ease common programming tasks and reduce boilerplate code. We mainly use the following components of this library: **Concurrent hash set**, **Immuteable list**.
Apache Commons http://commons.apache.org/	Provides utility methods and classes to ease common programming tasks and reduce boilerplate code. We mainly use the following components of this library: **Patricia trie**, **I/O utils**.

Test

Test Cases

We have two test cases with different role and feature in our project testing.

- **Case A: The most basic case**

We have chosen the sample code from the assignment outline as our foundational functionality test project. This project is concise and comprehensive, containing a complete call relationship chain, making it ideal for assessing code functionality and correctness.

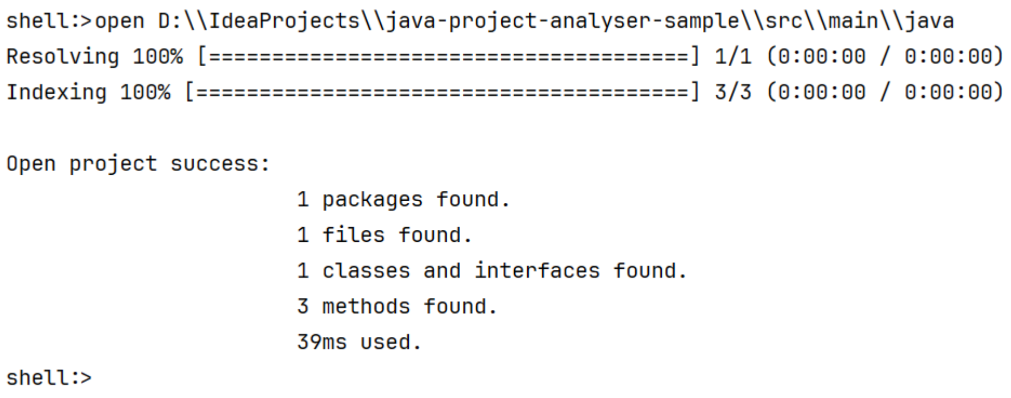
- **Case B: Large-scale Java library case**

We have chosen the commons-lang library, open-sourced by the Apache Foundation, as our test project. Apache Commons is an excellent project within the Java ecosystem, offering high-quality and reliable components for almost every aspect of Java. This library consists of over two hundred classes and more than three thousand methods, totaling 175,000 lines of code. Choosing such a vast project to test our own is a significant challenge. The version we selected is located at commit `gbe417ff07`.

Functionality Test

The following is the test result in our functionality test.

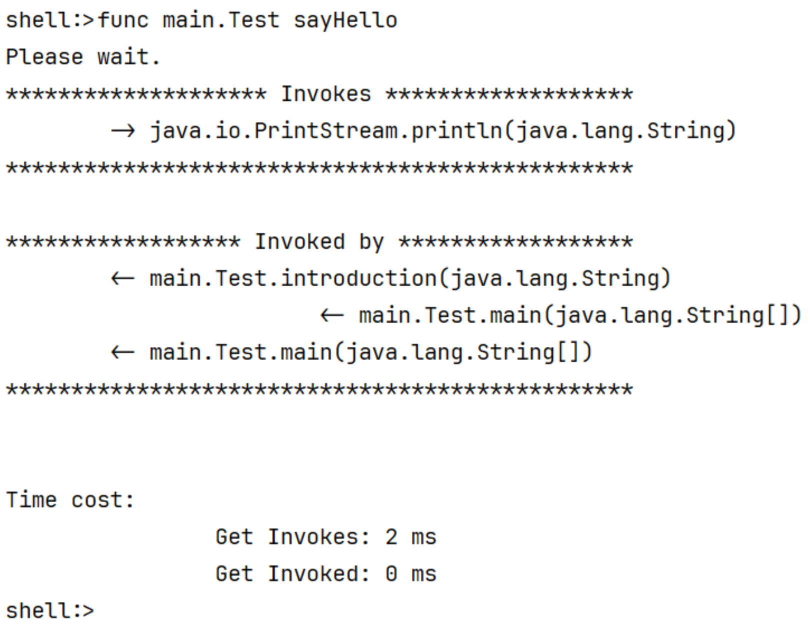
- **Case A**
  - Load the project



```
shell:>open D:\\IdeaProjects\\java-project-analyser-sample\\src\\main\\java
Resolving 100% [======================================] 1/1 (0:00:00 / 0:00:00)
Indexing 100% [=======================================] 3/3 (0:00:00 / 0:00:00)

Open project success:
                       1 packages found.
                       1 files found.
                       1 classes and interfaces found.
                       3 methods found.
                       39ms used.
shell:>
```

- Query method invocation relationships

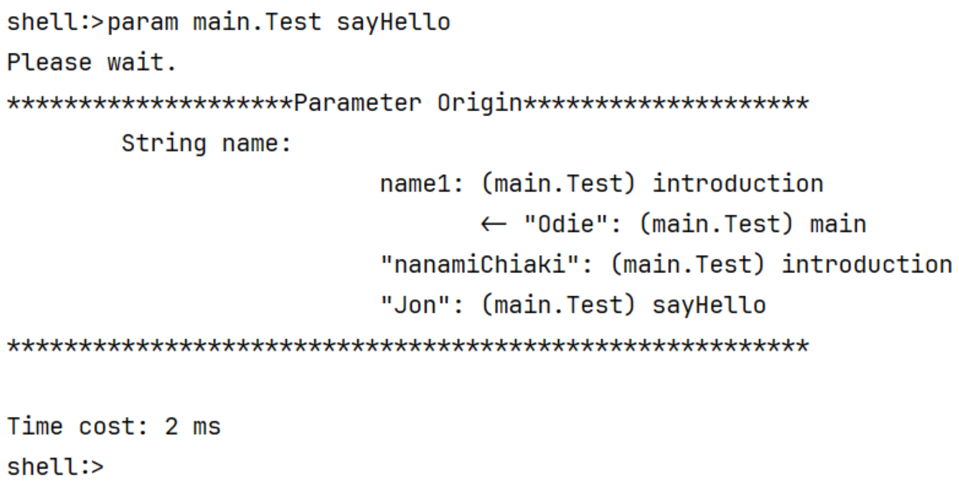


```
shell:>func main.Test sayHello
Please wait.
******************** Invokes *******************
        → java.io.PrintStream.println(java.lang.String)
************************************************

****************** Invoked by ******************
        ← main.Test.introduction(java.lang.String)
                        ← main.Test.main(java.lang.String[])
        ← main.Test.main(java.lang.String[])
************************************************


Time cost:
                Get Invokes: 2 ms
                Get Invoked: 0 ms
shell:>
```

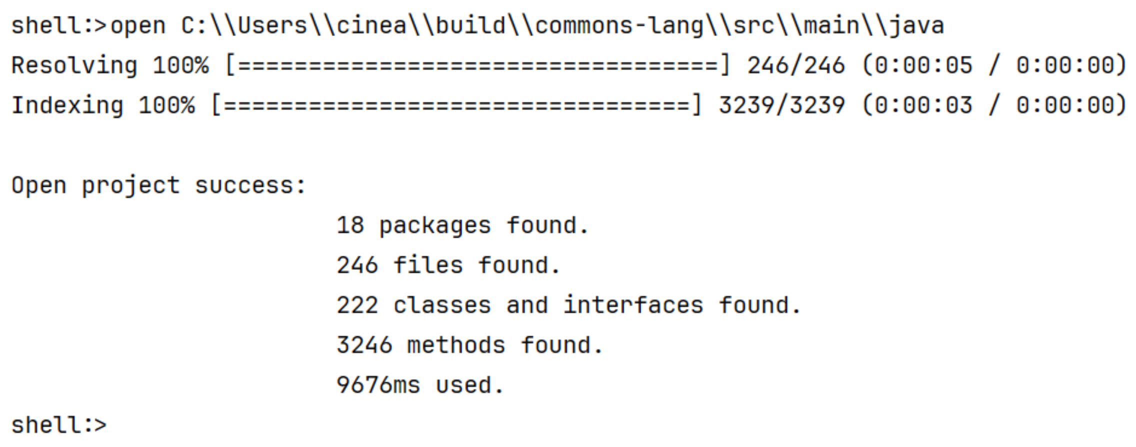
- Query method parameters origin



```
shell:>param main.Test sayHello
Please wait.
******************Parameter Origin******************
        String name:
                          name1: (main.Test) introduction
                                 ← "Odie": (main.Test) main
                          "nanamiChiaki": (main.Test) introduction
                          "Jon": (main.Test) sayHello
****************************************************

Time cost: 2 ms
shell:>
```

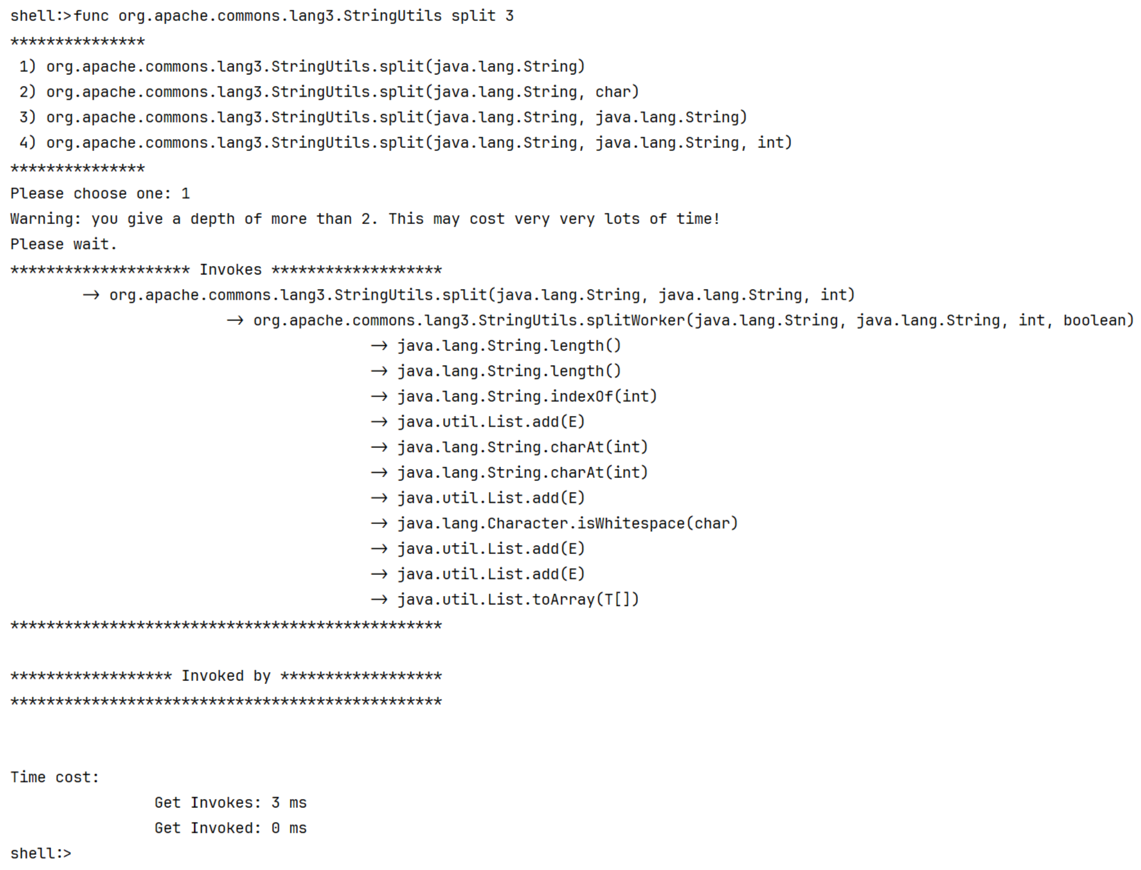
- **Case B**
  - Load the project



```
shell:>open C:\\Users\\cinea\\build\\commons-lang\\src\\main\\java
Resolving 100% [==================================] 246/246 (0:00:05 / 0:00:00)
Indexing 100% [=================================] 3239/3239 (0:00:03 / 0:00:00)

Open project success:
                       18 packages found.
                       246 files found.
                       222 classes and interfaces found.
                       3246 methods found.
                       9676ms used.
shell:>
```

- Query method invocation relationships



```
shell:>func org.apache.commons.lang3.StringUtils split 3
***************
 1) org.apache.commons.lang3.StringUtils.split(java.lang.String)
 2) org.apache.commons.lang3.StringUtils.split(java.lang.String, char)
 3) org.apache.commons.lang3.StringUtils.split(java.lang.String, java.lang.String)
 4) org.apache.commons.lang3.StringUtils.split(java.lang.String, java.lang.String, int)
***************
Please choose one: 1
Warning: you give a depth of more than 2. This may cost very very lots of time!
Please wait.
******************** Invokes ******************
        → org.apache.commons.lang3.StringUtils.split(java.lang.String, java.lang.String, int)
                        → org.apache.commons.lang3.StringUtils.splitWorker(java.lang.String, java.lang.String, int, boolean)
                                        → java.lang.String.length()
                                        → java.lang.String.length()
                                        → java.lang.String.indexOf(int)
                                        → java.util.List.add(E)
                                        → java.lang.String.charAt(int)
                                        → java.lang.String.charAt(int)
                                        → java.util.List.add(E)
                                        → java.lang.Character.isWhitespace(char)
                                        → java.util.List.add(E)
                                        → java.util.List.add(E)
                                        → java.util.List.toArray(T[])
*************************************************

****************** Invoked by ******************
*************************************************


Time cost:
                Get Invokes: 3 ms
                Get Invoked: 0 ms
shell:>
```

```
shell:>func org.apache.commons.lang3.StringUtils split 3
***************
 1) org.apache.commons.lang3.StringUtils.split(java.lang.String)
 2) org.apache.commons.lang3.StringUtils.split(java.lang.String, char)
 3) org.apache.commons.lang3.StringUtils.split(java.lang.String, java.lang.String)
 4) org.apache.commons.lang3.StringUtils.split(java.lang.String, java.lang.String, int)
***************
Please choose one: 4
Warning: you give a depth of more than 2. This may cost very very lots of time!
Please wait.
******************** Invokes ******************
        → org.apache.commons.lang3.StringUtils.splitWorker(java.lang.String, java.lang.String, int, boolean)
                        → java.lang.String.length()
                        → java.lang.String.length()
                        → java.lang.String.indexOf(int)
                        → java.util.List.add(E)
                        → java.lang.String.charAt(int)
                        → java.lang.String.charAt(int)
                        → java.util.List.add(E)
                        → java.lang.Character.isWhitespace(char)
                        → java.util.List.add(E)
                        → java.util.List.add(E)
                        → java.util.List.toArray(T[])
************************************************

****************** Invoked by ******************
        ← org.apache.commons.lang3.StringUtils.split(java.lang.String)
************************************************


Time cost:
                Get Invokes: 2 ms
                Get Invoked: 0 ms
shell:>
```

- Query method parameters origin

```
shell:>param org.apache.commons.lang3.StringUtils split 3
***************
 1) org.apache.commons.lang3.StringUtils.split(java.lang.String)
 2) org.apache.commons.lang3.StringUtils.split(java.lang.String, char)
 3) org.apache.commons.lang3.StringUtils.split(java.lang.String, java.lang.String)
 4) org.apache.commons.lang3.StringUtils.split(java.lang.String, java.lang.String, int)
***************
Please choose one: 4
Please wait.
********************Parameter Origin********************
        String str:
                          str: (org.apache.commons.lang3.StringUtils) split
        String separatorChars:
                          null: (org.apache.commons.lang3.StringUtils) split
        int max:
                          -1: (org.apache.commons.lang3.StringUtils) split
********************************************************

Time cost: 4 ms
shell:>
```

Conclusion