****



**本 科 毕 业 设 计**

院 系 软件学院

专 业 软件工程

题 目 LogHelper线程相关和数据库相关和重要类以及方法检查模块的设计与实现

年 级 2014 学 号 141250014

学生姓名 陈天歌

指导教师 荣国平 职 称 讲师

提交日期 2018-05-16

**南京大学本科生毕业论文（设计）中文摘要**

毕业论文题目： LogHelper线程相关和数据库相关和重要类以及方法检查模块的设计与实现

软件学院 院系 软件工程 专业 2014级本科生姓名： 陈天歌

指导教师（姓名、职称）： 荣国平 讲师

摘要：

随着软件工程的发展，企业以及个人对于软件质量的要求越来越高，日志记录是其中重要的一个部分。而如今对于日志分析的研究相对较多，却对于日志的质量，日志的产出过程关注较少。日志的质量极大地影响着后期日志的分析工作，所以本次要完成的插件工作的目标是为了填补当前IntelliJ IDEA上对于Java语言的代码日志提示插件的缺乏，希望能帮助有关开发人员提高日志的质量。

通过大量文章以及博客的梳理，本团队总结出了八条通用的代码检查规则，对这每条规则的检查与提示都进行了实现，以免开发人员遗漏需要记录日志的地方。为了提高日志的质量，本团队也制定了日志语句的模板，动态生成与当前代码环境相关的日志语句。在上述实现过程中，主要使用了IntellJ插件开发中的Inspection技术、PSI技术以及Quickfix技术。

上述八条检查规则分别为：assert检查、exception检查、重要分支语句检查、线程相关检查、文件相关检查、服务器相关检查、数据库相关检查、重要类以及方法检查。本人负责其中的线程相关检查、数据库相关检查以及重要类以及方法检查。

关键词：IntelliJ插件，日志检查，智能插入，Inspection，PSI，Java

**南京大学本科生毕业论文（设计）英文摘要**

THESIS： The design and implementation of LogHelper’s thread-related, database-related, critical class and method check modules

DEPARTMENT： Software Institute

SPECIALIZATION: Software Engineering

UNDERGRADUATE: Tiange Chen

MENTOR: Guoping Rong

ABSTRACT：

With the development of software engineering, companies and individuals are increasingly demanding software quality, and logging is playing an important part in guaranteeing this. Nowadays, there are relatively more studies on log analysis, but there is less concern about the quality of the log and the production period of log. The quality of the log greatly influences the analysis of post-logs. Therefore, the goal of the plug-in work to be completed this time is to fill in the lack of code log prompts for the Java language on IntelliJ IDEA. I hope to help the developers improve the quality of their logs.  
 Through reading a large number of articles and blogs, our team has summarized eight common code inspection rules, and implemented checks and prompts for each of these rules, so as to prevent developers from missing out on the need to record logs. In order to improve the quality of the log, our team also developed a template for the log statement and dynamically generated log statements related to the current code environment. In the above implementation process, the Inspection technology, PSI technology, and Quickfix technology in IntelJ plug-in development are mainly used.  
 The above eight inspection rules are: assert check, exception check, important branch statement check, thread-related check, file-related check, server-related check, database-related check, critical class and method check. I am responsible for thread-related checks, database-related checks, and critical class and method checks.

KEY WORDS: IntelliJ Plugin, Logging check, Intelligent insertion, Inspection, PSI, Java

**目 录**

[第一章 引言 1](#_Toc514328868)

[1.1 项目背景 1](#_Toc514328869)

[1.2 国内(外)日志记录研究现状 1](#_Toc514328870)

[1.2.1 何处进行日志记录研究现状 2](#_Toc514328871)

[1.2.2 日志记录内容研究现状 4](#_Toc514328872)

[1.3 论文的主要工作和组织结构 6](#_Toc514328873)

[第二章 技术概述 8](#_Toc514328874)

[2.1 IntelliJ Inspection技术 8](#_Toc514328875)

[2.2 PSI技术 9](#_Toc514328876)

[2.3 Quickfix技术 11](#_Toc514328877)

[2.4 logging技术 11](#_Toc514328878)

[2.4.1 Log4J 11](#_Toc514328879)

[2.4.2 Apache Commons Logging 12](#_Toc514328880)

[2.4.3 Logback 12](#_Toc514328881)

[2.4.4 SLF4J 12](#_Toc514328882)

[2.4 本章小结 12](#_Toc514328883)

[第三章 系统需求分析与概要设计 14](#_Toc514328884)

[3.1 LogHeler项目整体概述 14](#_Toc514328885)

[3.1.1 Inspection模块概述 14](#_Toc514328886)

[3.1.2 Quickfix模块概述 15](#_Toc514328887)

[3.2 LogHelper的需求分析 15](#_Toc514328888)

[3.2.1 用例图 15](#_Toc514328889)

[3.2.2 用例描述 16](#_Toc514328890)

[3.2.3 系统顺序图 19](#_Toc514328891)

[3.2.4 非功能性需求描述 19](#_Toc514328892)

[3.3 LogHelper概要设计 20](#_Toc514328893)

[3.3.1 系统框架结构 20](#_Toc514328894)

[3.3.2 系统模块设计 20](#_Toc514328895)

[3.4 本章小结 22](#_Toc514328896)

[第四章 详细设计与实现 23](#_Toc514328897)

[4.1 项目子模块概述 23](#_Toc514328898)

[4.2项目的详细设计 23](#_Toc514328899)

[4.2.1 重要类以及方法检查模块详细设计 23](#_Toc514328900)

[4.2.2 线程相关检查模块 25](#_Toc514328901)

[4.2.3 数据库相关检查模块 27](#_Toc514328902)

[4.3 项目的实现 29](#_Toc514328903)

[4.3.1 线程相关检查模块详细实现 29](#_Toc514328904)

[4.3.2数据库相关检查模块详细实现 32](#_Toc514328905)

[4.3.3用户自定义配置模块详细实现 35](#_Toc514328906)

[4.4 插件运行效果截图【以重要类以及方法为例】 39](#_Toc514328907)

[4.5 本章小结 40](#_Toc514328908)

[第五章 总结与展望 41](#_Toc514328909)

[5.1 总结 41](#_Toc514328910)

[5.2 展望 41](#_Toc514328911)

[参考文献 43](#_Toc514328912)

[致谢 44](#_Toc514328913)

# 图目录

[图1.1 assertion-check logging代码图 2](#_Toc514329176)

[图1.2 return-value-check logging代码图 3](#_Toc514329177)

[图1.3 exception logging代码图 3](#_Toc514329178)

[图1.4 logic-branch logging代码图 3](#_Toc514329179)

[图1.5 observing-point logging代码图 3](#_Toc514329180)

[图3.1 LogHelper系统用例图 16](#_Toc514329181)

[图3.2 LogHelper系统顺序图 19](#_Toc514329182)

[图4.1 重要类以及方法检查模块类图 23](#_Toc514329183)

[图4.2 重要类以及方法检查模块顺序图 25](#_Toc514329184)

[图4.3 线程相关检查模块类图 25](#_Toc514329185)

[图4.4 线程相关检查模块顺序图 26](#_Toc514329186)

[图4.5 数据库相关检查模块类图 27](#_Toc514329187)

[图4.6 数据库相关检查模块顺序图 28](#_Toc514329188)

[图4.7 ThreadInspection类代码 30](#_Toc514329189)

[图4.8 ThreadLoggingUtil类代码 31](#_Toc514329190)

[图4.9 JDBCInspection类代码 32](#_Toc514329191)

[图4.10 JDBCInspection类代码 34](#_Toc514329192)

[图4.11 JDBCExecuteLoggingUtil类代码 35](#_Toc514329193)

[图4.12 CriticalInspectionConfigAction类代码 36](#_Toc514329194)

[图4.13 plugin.xml代码 36](#_Toc514329195)

[图4.14 CriticalOpeInspection类代码 37](#_Toc514329196)

[图4.15 CriticalOpeLoggingUtil类代码 38](#_Toc514329197)

[图4.16 IntelliJ Inspection截图 39](#_Toc514329198)

[图4.17 重要类以及方法配置截图 39](#_Toc514329199)

[图4.18 重要方法检查与高亮提示截图 40](#_Toc514329200)

[图4.19 重要方法日志记录语句选择截图 40](#_Toc514329201)

[图4.20 重要方法已添加日志截图 40](#_Toc514329202)

# 

# 表目录

[表1.1 日志记录场景分类表 2](#_Toc514328995)

[表1.2 五类记录触发标准 4](#_Toc514328996)

[表3.1 UC1用例描述表 16](#_Toc514328997)

[表3.2 UC2用例描述表 16](#_Toc514328998)

[表3.3 UC3用例描述表 17](#_Toc514328999)

[表3.4 UC4用例描述表 18](#_Toc514329000)

[表3.5 UC5用例描述表 18](#_Toc514329001)

[表3.6 系统非功能性需求表 19](#_Toc514329002)

[表3.7 系统按规则划分模块设计表 21](#_Toc514329003)

[表3.8 系统按流程划分模块设计表 22](#_Toc514329004)

# 第一章 引言

## 1.1 项目背景

随着近几十年软件工程的发展，软件产品日益渗透人们的日常生活，如今我们生活在一个衣食住行都与软件相关的环境中。正是因为软件产品与人们生活的密切相关性，软件产品的质量很大程度上影响着人们的生产与生活质量，企业以及个人对于软件质量的要求也越来越高。日志记录是保证软件质量的一个重要环节。通过有效日志的分析，可以便于了解软件产品的运行情况，分析以往用户操作行为、类型喜好、地域分布等统计情况，更重要的是可以及时发现问题并快速定位问题以便开发人员解决问题，这对于软件质量提供了保证。如今，对于日志内容分析的研究与工具相对较多，比较常见的有：ELK(Elasticsearch[[1]](#footnote-1)1+Logstash[[2]](#footnote-2)2+Kibana[[3]](#footnote-3)3)，甚至还有直接提供日志分析的云平台，如：百度云的大数据分析产品提供日志分析托管服务[[4]](#footnote-4)4。

日志分析工具都是对已有日志内容进行操作，这很大程度依赖于其所基于的日志内容的可靠性。现在对于日志的规范以及产出过程的工具相对较少，这有部分原因是基于在何处记录日志的复杂性以及不同开发人员的日志习惯。由于软件逻辑与日志记录的相关性，软件的复杂度与个性化形成了日志记录的复杂与差异性。因此，为所有的项目实现一套统一的使得日志记录不多也不少的规则在理论上是很难以实现的。然而，日志记录有一些适用于所有项目的通用规则，这些规则能为软件质量提供一定的保证。因此，我们的插件提供了一些通用的日志记录规则的提示来帮助软件开发人员进行有效的日志记录。

## 1.2 国内(外)日志记录研究现状

日志记录要解决两个问题：在何处应该进行日志记录使得日志记录既不多也不少，过少的日志记录提供的信息不够充分，过多的日志记录不仅影响系统的性能也使得寻找有效日志变得困难；在该记录日志的地方需要记录哪些信息。这两个问题都很难准确回答，现如今对于这两个问题的研究也比较粗糙不够充分，个人有个人的见解，既不够全面、细节也不够统一。

### 1.2.1 何处进行日志记录研究现状

日志记录的地方可以按照场景分为两大类：unexpected situations用以记录非正常执行状态；execution points用以记录正常执行过程中的关键执行点。也可以按照关键词细分为五大类：Assertion-check logging; Return-value-check logging; Exception logging; Logic-branch logging; Observing-point logging。两种分类关系如下表1.1所示：

表1.1 日志记录场景分类表

|  |  |
| --- | --- |
| Unexpected situations | Assertion-check logging |
| Return-value-check logging |
| Exception logging |
| Execution points | Logic-branch logging |
| Observing-point logging |

Assertion-check logging

开发人员用assert或类似方法在源代码中执行断言检查来识别错误。失败的断言语句会在执行终止前自动记录失败消息。以下图1.1为Assertion-check logging在实际开发过程中的一个例子。

|  |
| --- |
| ULS.AssertTag(site != null, “site cannot be null”); |

图1.1 assertion-check logging代码图

Return-value-check logging

日志语句用于在执行返回值检查之后记录潜在的函数返回错误。函数调用的不正确的返回值是发现潜在错误的指标，因此，可以通过检查函数调用的返回值来显示检查与记录意外事件。以下图1.2为Return-value-check logging在实际开发过程中的一个例子。

|  |
| --- |
| if (String.IsNullOrEmpty(tokenReference)){  ULS.SendTraceTag(ULSTraceLevel.Unexpected, “Missing token reference value.”);} |

图1.2 return-value-check logging代码图

Exception logging

开发人员在异常发生后记录上下文。现代编程语言通常具有异常机制，以java为例，记录通常在catch块中或在throw语句之前。以下图1.3为Exception logging在实际开发过程中的一个例子。

|  |
| --- |
| try{  CheckAccount(account);  }catch(Exception e){  Logger.Log( “CheckAccount:”+account+” failed with exception: ”+e);} |

图1.3 exception logging代码图

Logic-branch logging

日志语句通常在逻辑分支点记录运行时执行信息。源代码中通常通过使用类似if或switch之类的分支语句来导致不同的代码逻辑路径，在关键分至点的日志信息可以帮助识别与因果关系相关的代码执行路径。以下图1.4为Logic-branch logging在实际开发过程中的一个例子。

|  |
| --- |
| if(instanceName.IsSqlExpressInstalled){  Tracer.TraceLogInfo(“Detect sql express instance. No need to install.”);  }else{  Tracer.TraceLogInfo(“No sql express instance. Do fresh install.”);  res = SqlCleanInstall();} |

图1.4 logic-branch logging代码图

Observing-point logging

以上四类日志记录以外的日志记录，用以观察和了解系统的运行时状态，这类记录有多种不同的记录方案，例如：记录一个函数的入口、出口，记录一个重要的事务并报告关键事件（如心跳）。以下图1.5为Observing-point logging在实际开发过程中的一个例子。

|  |
| --- |
| Tracer.TraceLogInfo(“Creating the tab order for form {0}”,base.Name); |

图1.5 observing-point logging代码图

从上面的描述以及代码例子可以总结出，以上五类记录触发标准如下表1.2：

表1.2 五类记录触发标准

|  |  |
| --- | --- |
| 类别 | 标准 |
| Assertion-check logging | 日志语句由一个Assert语句触发 |
| Return-value-check logging | 日志语句包含在一个分支条件中检查一个或多个函数返回值的分支语句之后的一个字句中。另外，日志语句不在子句中的任何catch块中 |
| Exception logging | 日志语句要么在一个catch块中，要么刚好在一个throw语句之后 |
| Logic-branch logging | 日志语句包含在一个分支条件中不包含任何对于函数返回值检查的分支语句后的一个子句中 |
| Observing-points logging | 所有除了以上场景的日志语句记录 |

然而，并不是所有满足以上触发标准的地方都需要进行日志记录。事实上，在现实工程中，只有一小部分重点代码片段被记录。

对于exception出现的场景，尽管许多异常类型与开发人员在其catch代码块中的logging decision有很高的相关性，对应于这种异常类型的catch块只占一小部分。也就是说，大多数catch块对应的异常类型与开发人员在其catch块中的logging decision不高度相关。大多数catch模块有以下三点主要的不进行日志记录的情况：将logging decision传递到下一个操作中；异常可恢复；不重要的异常。

### 1.2.2 日志记录内容研究现状

程序执行的某些状态更改或动作比其他更重要。为了满足日志的分类需求，所有良好的日志记录框架为日志消息提供了简单的分类系统，例如：Log4j[[5]](#footnote-5)1, LOGBack[[6]](#footnote-6)2, java.util.Logging[[7]](#footnote-7)3,, commons-logging[[8]](#footnote-8)4等，还有作为门面的SLF4J[[9]](#footnote-9)5也是如此。

典型的分类系统将每个项目和日志级别相关联。这些日志级别通常分为三类：第一类是WARN, ERROR, FATAL，这三种描述程序中发生的事件或是出现的问题，严重程度逐渐增加；第二类是INFO，用于提供一般信息；第三类是DEBUG，用于调试。

日志记录框架会以结构化的方法写入日志，使得在日志中搜索与某个级别相关的条目变得轻松。此外，某些框架允许进行消息过滤，可能会抑制具有特定级别的条目，或是将它们转移到不同的接口，例如，FATAL消息可能通过电子邮件发送给支持团队，DEBUG消息可能会被完全抑制。

有以上信息可知，日志的级别很重要，开发团队应该确定这些级别的使用情况保持开发人员之间的一致性。以下是各类日志级别的建议使用情况：

INFO

此级别单纯用于信息化的通用信息，不应该用于指示程序中的故障或是错误状态。为了有效使用这类日志级别，应当尝试考虑当主接口无法访问的时候，通用信息对于诊断程序错误应当有用。一些可能有用的信息例子如：与产品版本相关的信息（在程序启动时将版本信息输出到日志）、使用信息（软件的使用者，使用者的当前操作）、与产品有关的外部服务（数据库，web服务等）。

WARN

此级别是表示某种形式的程序失败的第一个级别。WARN级别的消息用来指示程序所面对的潜在的问题，此时，问题并没有发生，不对用户的使用造成影响。例如：WARN消息可能适用于一个外部服务无法使用但是有另一个能够提供相同功能的服务的情况。另外，WARN消息也适用于需要重复尝试访问给定资源的场景。

ERROR

此级别是表示某种形式的程序失败的第二个级别。ERROR级别用来指示程序面临比WARN更严重的问题，对用户的使用造成影响。例如：数据库连接失败可能导致部分程序呈现不可用状态。

FATAL

此级别是表示某种形式的程序失败的第三个级别。FATAL级别用来指示程序出现致命错误，不仅对用户体验造成影响，而且会导致程序停止。例如：一个对于程序运行至关重要的组件失败，使其处于不稳定的状态，唯一可能的行为是完全终止程序。

DEBUG

此级别用于指示调试信息的日志信息，直接针对开发人员而且其可用性与正在开发的程序性质有很高的相关性。因为通过调试器可以解决很多问题从而不需要DEBUG消息，但是，在以下情况中DEBUG消息很有用：

在调试图形渲染桌面程序时，引发调试器的过程可能会干扰渲染过程本身。在这种情况下，DEBUG消息可以以一种比较不突出的方式“观察”渲染过程。

开发过程中的某些程序类型或者阶段需要特殊的程序执行环境。在这种情况下，使用调试工具可能不适合，而可以使用DEBUG消息来寻找错误解决方案。

## 1.3 论文的主要工作和组织结构

本文主要介绍了在代码开发过程中开发人员应该在何处记录日志以及日志应该如何记录的相关理论知识，以及一个基于Java的用于进行智能日志提示的IntelliJ插件LogHelper的设计和实现。主要是对Java代码中应该在何处记录日志以及如何记录日志的理论知识进行描述，同时描述一个基于Java的用于进行智能日志提示的IntelliJ插件的设计和实现过程以及这个过程中出现的一些问题。

第一章：概述和前言部分，主要介绍了项目背景，当前工开发人员应该在何处记录日志以及日志应该如何记录的研究及现状 ，并描述了本论文的主要工作和组织结构。

第二章：主要介绍基于Java的用于进行智能日志提示的IntelliJ插件LogHelper的开发和实现过程中用到的相关理论知识和理论研究。

第三章：从项目概述和需求分析两个方面，描述基于Java的用于进行智能日志提示的IntelliJ插件LogHelper的提出背景和开发者信息，同时分析和总结出功能性和非功能性需求。

第四章：从项目概要设计方面，描述项目整体的设计框架以及通过功能划分对项目进行包设计和接口设计。

第五章：对系统不同模块的详细设计和实现进行描述，其中包括代码中日志记录检测规则的标准的制定和各个模块的类设计图。

第六章：总结该项目已实现的功能，探讨项目的缺点和不足，并指出该项目未来的扩展和发展方向。

# 第二章 技术概述

## 2.1 IntelliJ Inspection技术

IntelliJ IDEA[[10]](#footnote-10)1有自带的强大、快速以及灵活的静态代码分析能力，可以检测语言和运行时错误，在编译之前建议更正与改进。IntelliJ IDEA通过对代码进行检查来执行代码分析，而且代码分析可灵活配置并且可以启用或禁用每个代码检查并更改其严重性，使用自定义检查集创建配置文件可以实现在不同范围内用不同方式进行检查或抑制特定代码中的检查等。

检查严重程度表示检查发现的代码问题对项目的影响的严重性，并且会决定在编辑器中会如何突出所检测到的问题。默认情况下，检查有以下严重级的分类：Server problem、Typo、Info、Weak Warning、Warning、Error。

用户可以自行增加或减少每次检查的严重性级别，比如：将IntelliJ IDEA的某些警告显示为错误或者弱警告，这样会使得原本被当作弱警告的内容可以显示为警告或者错误，甚至仅仅显示为信息。用户还可以配置用语突出显示每个严重性级别的颜色或是字体的样式，还可以创建自定义的严重性级别并将其设置成特定的检查条目，并且可以为不同范围内的相同的检查设置成为不同的严重性级别。

默认情况下，所有被启用的代码检查作用范围是所有的项目文件，但是，用户可以在不同的作用域配置不同的代码检查，操作包含启用与禁用或是更改严重性级别和选项，这种配置将作为特定配置文件的一部分进行保存和应用。

存在着一个检查在不同的作用域中有不同的配置这样的复杂情况。当这样的一个检查在一个部分或全部属于这些作用域的文件中执行时，最高优先级的配置将会被执行。由检查设置中检查特定作用域配置的相对位置定义优先级：最高配置具有最高的优先级，Everywhere else配置具有最低的优先级。

在某些场景下，用户可能希望禁用特定的检查，也就是说，用户希望忽视一些代码问题而继续执行其他的检查，例如：某条检查判定某些代码是死的，但是用户可能希望稍后执行这些代码，所以不希望在当前的编辑器或是问题报告中突出显示这个问题。在这种情况下，可以为特定的语句、函数、标记或是文件禁止这些检查。抑制和禁用代码检查之间有所区别：抑制检查时，IntelliJ IDEA会插入一个特殊的注释去告知代码分析器忽视特定代码检查中发现的问题而禁用检查时，就是关闭了这项检查所以代码分析器会忽视代码检查中发现的问题，类似错误检测的某些代码检查不能被抑制。

### 2.2 PSI技术

PSI[[11]](#footnote-11)1全称为Program Structure Interface，是IntelliJ平台中负责解析文件并创建支持多平台特性的语法和语义代码模型的一个层。在IntelliJ IDEA所管理的项目中，每个目录，包，源代码以及资源文件都会被抽象成对应的PSI对象。PSI文件是表示用一个特定编程语言以分层元素来表示文件内容的结构的根。PsiFile类是所有PSI文件的通用基类，特定语言的文件通常由其子类表示，例如：PsiJavaFile类表示一个Java文件，XmlFile类表示一个XML文件，不同于VirtualFile和Document具有程序范围（即使是打开了多个项目，每个文件都有一个共同的VirtualFile实例表示），PSI具有项目范围（如果一个文件属于多个同时打开的项目，则该文件由多个PsiFile实例表示）。

FileViewProvider类是IntelliJ IDEA 6.0中引入的，为了管理对单个文件中多个PSI树的访问，例如：一个JSPX页面有一个Java代码的PSI树（PsiJavaFile），有一个XML代码的PSI树（XmlFile），还有一个JSP对应的PSI树。每个PSI树覆盖文件的全部内容，并且可以在找到不同语言内容的地方包含特殊的外部语言元素。FileViewProvider实例相当于一个单独的VirtualFile，一个单独的Document，而且可以检索多个PsiFile实例。

PSI文件是表示PSI元素（又称PSI树）的层次结构。一个PSI文件可能包含几种不同的编程语言的PSI树，一个PSI元素可以有子PSI元素，因此，一个PSI文件也是一个PSI元素。单个PSI元素级别的操作和PSI元素是用来分析IntelliJ平台解释的源代码内部结构的工具，所以可以利用PSI元素进行代码分析，例如本项目的代码检查以及修改操作。

PSI有三种主要的导航方式：自上而下，自下而上以及使用references。如果有一个PSI文件或是类似一个方法的另一个级别更高的元素，并且需要查找羽制定条件相配的所有元素，则适合使用第一种导航方式。如果PSI树中有类似插入符号处的元素的特定点，并且需要找出与其有关的上下文的内容，则适合使用第二种导航方式。如果需要从某个元素的使用到元素的声明并返回，则适合使用

第三种导航方式。

1. 自上而下导航

通常使用visitor实现自上而下导航。创建一个继承visitor基类的类，重写处理想处理的元素的方法，然后将visitor实例传递给PsiElement.accept()。Visitor基类是语言相关的，例如如果是处理一个Java文件的元素，所对应集成的基类应该是JavaRecursiveElementVisitor类。然而，在很多情况下也可以使用更具体的API实现自上而下的导航，例如调用PsiClass.getMethods()相比于使用visitor可以更简单地得到Java类中所有方法的列表。而且，PsiTreeUtil类里含有很多通用的独立于语言的对PSI树进行自上而下导航的方法。

1. 自下而上导航

可以使用PSI树中特定的元素或者偏移量来作为自下而上导航的起点。通过调用PsiFile.findElementAt()以及偏移量可以得到相应的PSI元素，获取的PSI元素是树中最低级别的元素，如果要获得更广的上下文，可以通过PsiTreeUtil.getParentOfType()来进行自下而上的导航，从而在树上找到所指定类型的元素，有时，也可以通过特定的导航方式实现。

1. 使用references导航

PSI树中的一个reference是表示代码中的特定元素的使用到相应声明的链接的一个对象。解析一个reference意味着定位一个特定的使用的声明。通过调用PsiReference.resolve()即代表解析一个reference，也就是说可以从reference获取被引用的声明。

### 2.3 Quickfix技术

Quickfix可视化显示为突出显示在代码左侧的操作指示器，允许通过显示的操作对代码进行自动更改。红色的小灯泡代表所检查代码的问题处于Error级别，黄色的小灯泡代表所检查代码的问题处于Warning、Suggestion或Hint级别。

大多数Quickfix立即应用，还有一部分Quickfix通过与用户的交互来选择如何处理代码。一个Inspection可以绑定多个Quickfix以实现对于一种检查规则的多种处理方法，提供给用户多种建议的解决方案。

## 2.4 logging技术

### 2.4.1 Log4J

Log4J是Apache的一个开源项目，是E.U.SEMPER项目编写的日志API，是一种流行的Java日志包。Log4J可以让开发人员用不同的粒度输出日志信息到多种目的地，可以控制每一条日志信息的输出格式以及定义每一条日志信息的级别。Log4J不仅可靠、快速和具有可扩展性，而且Log4J API易于理解与使用，学习曲线温和。

Log4J有三大组件：loggers，appenders，layouts。这三个组件分别使得开发人员可以使用不同的日志级别和日志类型输出信息、指定日志信息所输出的目的地以及日志信息输出的格式。

1. Loggers

Loggers可以设置日志级别，分别有TRACE、DEBUG、INFO、WARN、ERROR和FATAL这六种被定义在org.apache.log4j.Level类中的级别，也可以通过继承Level类自己定义级别，如果一个logger没有被指定level，这个logger会从父亲那里继承level，保证了所有的logger都会有level。

1. Appenders

Appender是日志输出的目的地，包括命令行、文件、GUI组件、远程socket服务器、JMS、NT事件日志和远程UNIX Syslog后台进程。一个logger实例可以拥有多个appender。每一个启用的日志输出请求的logger实例都会将请求转发到自己以及祖先上的所有appender上。

1. Layouts

Layout决定日志信息输出的格式，一个appender可以与一个layout相关联用以按照一定的格式输出日志信息。

### 2.4.2 Apache Commons Logging

Commons Logging提供统一的日志接口，是为需要建立在不同环境下使用不同日志架构的组件或是API库的开发人员创建的。一个使用Commons Logging API的库可以在运行时通过动态查找的机制自动找出真正使用的日志库。

### 2.4.3 Logback

Logback是由Log4J创始人设计的一个开源日志工具。目前，Logback分为logback-core、logback-classic、logback-access这三个模块。其中，logback-core模块是其他两个模块的基础；logback-classic模块扩展了logback-core模块，logback-classic模块本身实现了SLF4J API，因此可以切换不同的日志框架；logback-access模块与servlet容器集成以提供HTTP访问日志的功能。

### 2.4.4 SLF4J

SLF4J全称为Simple Logging Façade for JAVA，JAVA简单日志门面。与Apache Commons Logging类似，支持不同日志框架，可以在部署时不修改任何配置接入一种日志实现方案。SLF4J在编译时静态绑定具体的日志框架，支持参数化的log字符串。

## 2.4 本章小结

本章主要介绍了进行智能日志提示的IntelliJ插件过程中所需要了解和学习的技术。一条确定的规则可以定义为一个Inspection，所以需要了解IntelliJ Inspection有关的理论知识。一个Inspection需要通过Visitor去访问某一个起始的节点下的各个子PsiElement，所以需要了解关于PSI的理论知识。可以为一个Inspection内置QuickFix去实现代码更改（在此插件中体现为logging代码的插入），因此需要了解关于Quickfix的内容，与此同时，需要了解多种logging格式来完善提示的logging代码。

# 第三章 系统需求分析与概要设计

## 3.1 LogHeler项目整体概述

JetBrains公司开发的系列IDE产品都非常方便使用，也受到越来越多人的喜爱。Java语言庞大的用户群体也使得IntelliJ IDEA成为了其中用户使用最多的一款产品。在我们日常使用IntelliJ IDEA进行项目开发以及维护时，为了保证软件质量以及降低维护成本，经常会使用日志来记录软件中的特定操作。日志的记录地点以及记录内容的选择是日志记录中的重点。日常我们对于日志的使用是人工地按照自己或是团队的日志习惯手动编写加入的，为了便于在开发过程中可以得到更好的智能日志支持，我们开发了基于Java的智能日志提示与插入的IntelliJ插件。在开发人员进行代码编写时，插件会自动检测其中需要进行日志记录的部分，对需要进行日志记录的重要操作进行代码高亮提示。由于在软件开发中开发人员可能会使用不同的日志框架以及对于同一个操作所定位的日志level不同，所以我们提供给开发人员多种代码框架的选择以及多种日志level的选择。开发人员只需要在我们提供的代码插入语句中进行选择，插件就会在当前语句之后自动插入一条对应的日志记录语句。

本项目核心功能点主要有两个部分：创建inspection按照日志记录规则进行代码实时检查并提示，创建quickfix对所应该进行日志记录的地方提供多种日志记录插入方案。本小组成员工作按照日志的规则进行划分，本人负责实现代码中线程相关操作、数据库相关操作以及系统中某些关键操作的识别以及日志记录。以下将对这两个重要部分进行概要描述。

### 3.1.1 Inspection模块概述

Inspection模块用于实现按照编写的规则实时检查代码中应该进行日志记录的地方并对该地点进行代码高亮提示的功能。也就是说，在这个模块中进行了日志记录规则判定的实现，是将语言版的代码规则制定转化成为详细具体的代码的过程。Inspection模块具体流程是：插件持续扫描开发人员当前正在编写的代码，对已编写的所有内容进行所有日志规范的检查；对其中应该进行日志记录的部分进行代码高亮提示。

### 3.1.2 Quickfix模块概述

Quickfix模块用于实现对满足某个特定日志记录规则的语句的多种日志记录语句的提示以及自动插入开发人员所选定的日志记录语句的功能。也就是说，在这个模块中进行了日志记录自动插入的实现。Quickfix模块具体流程是：在开发人员选择提示后，提供给开发人员多种日志框架和多种日志记录level的日志记录语句的选择；当开发人员选择了其中某种特定的框架类型和日志记录level的语句后，在当前高亮提示的代码后自动插入所选择的日志记录语句。

## 3.2 LogHelper的需求分析

### 3.2.1 用例图

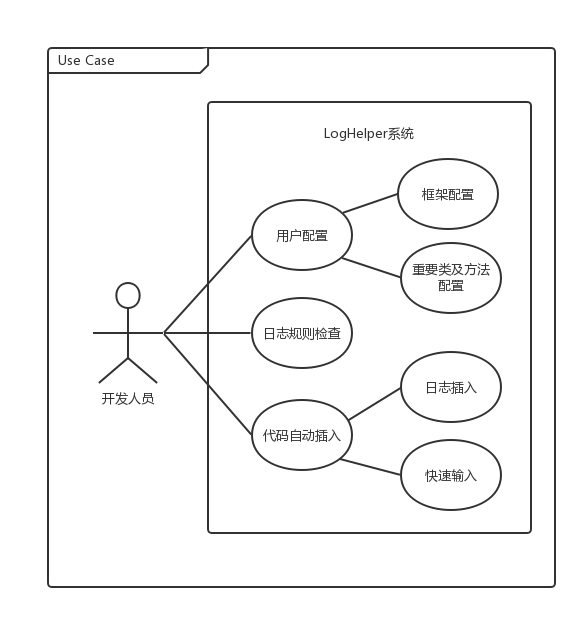


图3.1 LogHelper系统用例图

以上图3.1为系统用例图，基于Java的智能日志提示与插入的IntelliJ插件LogHelper作为系统，它的参与者是使用IntellJ进行Java开发的工作人员，用例主要是用户配置、对用户代码中日志规则的代码实时检查以及代码语句的自动插入。

### 3.2.2 用例描述

* 1. **UC1 框架配置**

框架配置用例的详细用例描述参见下表3.1。

表3.1 UC1用例描述表

|  |  |
| --- | --- |
| 用例编号 | UC1 |
| 参与者 | 开发人员 |
| 目标 | 配置项目所需要使用的日志框架 |
| 优先级 | 高 |
| 触发条件 | 开发人员想要配置本项目所使用的日志框架 |
| 前置条件 | 开发人员使用IntelliJ进行Java项目开发 |
| 后置条件 | 项目所使用日志框架被确定并在后续插入时按照此框架语句插入 |
| 正常流程 | 1. 开发人员选择配置开发所用日志框架 2. 系统显示多种日志框架 3. 开发人员选择其中某个框架 4. 系统记录开发人员所选择框架 |
| 扩展流程 | 无 |
| 特殊需求 | 无 |

* 1. **UC2 重要类以及方法配置**

重要类以及方法配置用例的详细用例描述参见下表3.2。

表3.2 UC2用例描述表

|  |  |
| --- | --- |
| 用例编号 | UC2 |
| 参与者 | 开发人员 |
| 目标 | 允许开发人员配置需要进行日志记录的重要的类或方法 |
| 优先级 | 高 |
| 触发条件 | 开发人员希望配置自己想要进行日志记录的类或方法 |
| 前置条件 | 开发人员使用IntelliJ进行Java项目开发 |
| 后置条件 | 系统保存开发人员配置的类与方法 |
| 正常流程 | 1. 开发人员选择配置按钮 2. 系统显示当前已配置的重要类以及方法并可供修改 3. 开发人员修改已配置的内容并确认 4. 系统保存开发人员的新配置并退出配置界面 |
| 扩展流程 | 3a. 开发人员选择退出配置  3a.1系统退出配置界面 |
| 特殊需求 | 配置类格式按照”包名.类名”格式（类名支持正则表达式）  配置方法格式按照”包名.类名:方法名“格式（类名以及方法名支持正则表达式） |

* 1. **UC3 日志规则检查**

日志规则检查用例的详细用例描述参见下表3.3。

表3.3 UC3用例描述表

|  |  |
| --- | --- |
| 用例编号 | UC3 |
| 参与者 | 开发人员 |
| 目标 | 在进行符合日志规则检查项的代码编写时获得日志记录提示 |
| 优先级 | 高 |
| 触发条件 | 开发人员所编写的代码中有符合某条日志规则检查项的代码 |
| 前置条件 | 开发人员使用IntelliJ进行Java项目开发 |
| 后置条件 | 关键代码高亮并获得提示 |
| 正常流程 | 1. 开发人员编写与某条日志规则检查项相符的关键 2. 系统检查到当前检查项还未进行日志记录，将相关代码高亮显示并显示提示按钮 3. 开发人员选择提示按钮 4. 系统显示代码插入提示项 |
| 扩展流程 | 2a.系统检查到当前检查项已进行日志记录 |
| 特殊需求 | 无 |

* 1. **UC4 日志插入**

日志插入用例的详细用例描述参见下表3.4。

表3.4 UC4用例描述表

|  |  |
| --- | --- |
| 用例编号 | UC4 |
| 参与者 | 开发人员 |
| 目标 | 在高亮的检查项相关代码后插入指定日志记录语句 |
| 优先级 | 高 |
| 触发条件 | 开发人员希望插入日志记录语句 |
| 前置条件 | 开发人员选择提示按钮 |
| 后置条件 | 高亮的检查项相关代码后被插入一条语句 |
| 正常流程 | 1. 开发人员选择日志记录提示 2. 系统在缺少日志记录的地方插入对应日志记录提示的日志记录语句 |
| 扩展流程 | 无 |
| 特殊需求 | 无 |

* 1. **UC5 快速输入**

快速输入用例的详细用例描述参见下表3.5。

表3.5 UC5用例描述表

|  |  |
| --- | --- |
| 用例编号 | UC5 |
| 参与者 | 开发人员 |
| 目标 | 允许开发人员通过关键词插入完整语句 |
| 优先级 | 高 |
| 触发条件 | 开发人员希望快速输入某个语句 |
| 前置条件 | 开发人员使用IntelliJ进行Java项目开发 |
| 后置条件 | 无 |
| 正常流程 | 1. 开发人员编写关键词 2. 系统提示与该关键词对应的语句 3. 开发人员确认语句 4. 系统将当前编写的关键词替换成对应的完整语句 |
| 扩展流程 | 无 |
| 特殊需求 | 无 |

### 3.2.3 系统顺序图

系统顺序图可以通过图形化地表示了系统用例图中参与者与系统之间详细的交互过程，本项目中较为复杂的交互过程是从开发人员进行重要类和方法配置直到插入语句的完整过程，此过程由下图3.1所示。将用户修改后的新配置更新入文件中，日志规则检查部分从文件中读取文件中的新配置进行代码实时检查，将需要进行日志记录的地方进行注册，通过工具类实现日志记录语句的插入。

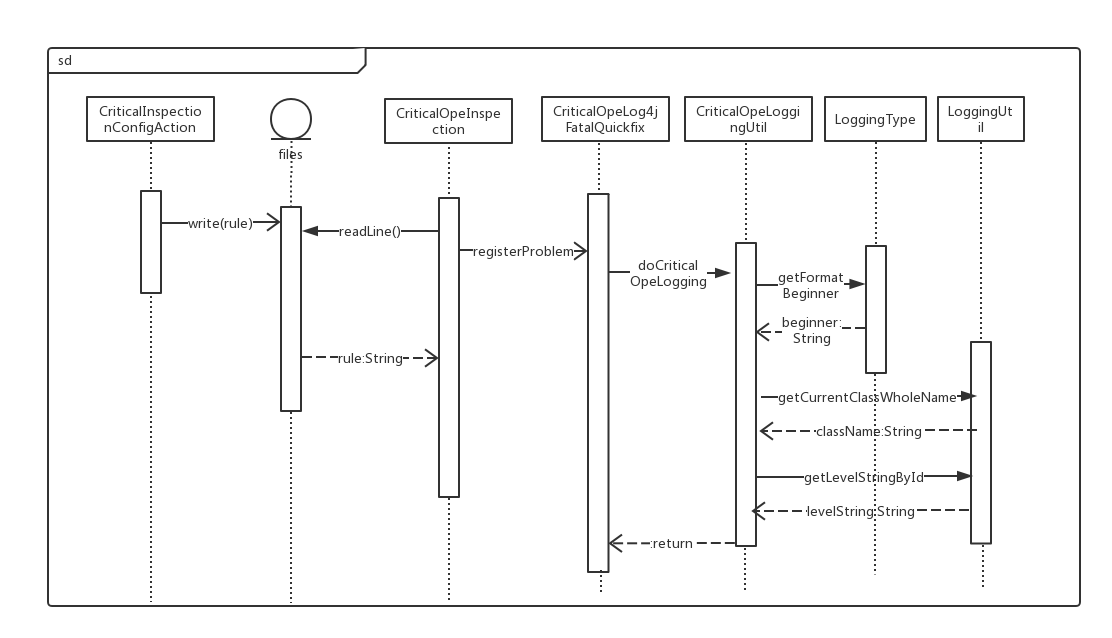


图3.2 LogHelper系统顺序图

### 3.2.4 非功能性需求描述

软件系统需求不仅包含以上的对功能需求的描述，还包含若干非功能性需求。非功能性需求不同于定义特定行为或功能的功能需求，非功能需求往往与产品的质量有关，是满足客户业务需求而不在功能需求内的需求。本插件系统的非功能性需求描述如下表3.6所示。

表3.6 系统非功能性需求表

|  |  |  |  |
| --- | --- | --- | --- |
| 需求ID | 非功能性需求 | 优先级 | 需求描述 |
| R1 | 易学性 | 8 | 用户应该可以在30分钟内了解插件的使用方法 |
| R2 | 可维护性 | 8 | 系统设计和模块划分应该尽可能合理，当增加新的需求时，新模块的扩展应该在一人两天内完成 |
| R3 | 迭代式开发 | 5 | 开发日起第一个月完成初始版本，两个月完成最终版本 |
| R4 | 可移植性 | 8 | 系统可以在Windows8及以上操作系统、MacOS、Linux等操作系统上正常运行 |
| R5 | 可靠性 | 10 | 不产生会使IntellJ崩溃的缺陷，在IntellJ自己无故崩溃后，不影响重启后本插件的使用且不丢失用户的配置信息 |
| R6 | 反应时间 | 8 | 在开发人员进行操作时，系统在0.5s内做出响应 |

## 3.3 LogHelper概要设计

### 3.3.1 系统框架结构

系统框架结构如下图3.3所示，其中与代码规则有关的核心部分是，每个Inspection都会关注一个日志规则检查点，在plugin.xml可以注册provider，通过provider可以注册inspection，一个inspection可以与多个quickfixes相关联。插件将自动扫描当前代码，inspection通过调用visitor访问某个PsiElement的各个子PsiElement，在访问PsiElement时，如果发现当前代码片段与某个日志规则相符合，则可以提供给开发人员与这个inspection相关联的多个quickfixes，当开发人员选择其中某个quickfix时，会触发该quickfix的执行完成日志记录语句的插入。

### 3.3.2 系统模块设计

通过大量的资料的梳理以及代码阅读，本插件确认了八条代码规则检查项，分别为：assert检查项、exception检查项、重要分支语句检查项、线程相关检查项、文件相关检查项、服务器相关检查项、数据库相关检查项、用户自定义配置重要类和方法检查项。因此按照这八条代码规则检查项对应设计了系统按规则划分的八个系统模块，这八个模块名称及职责如下表3.7所示。

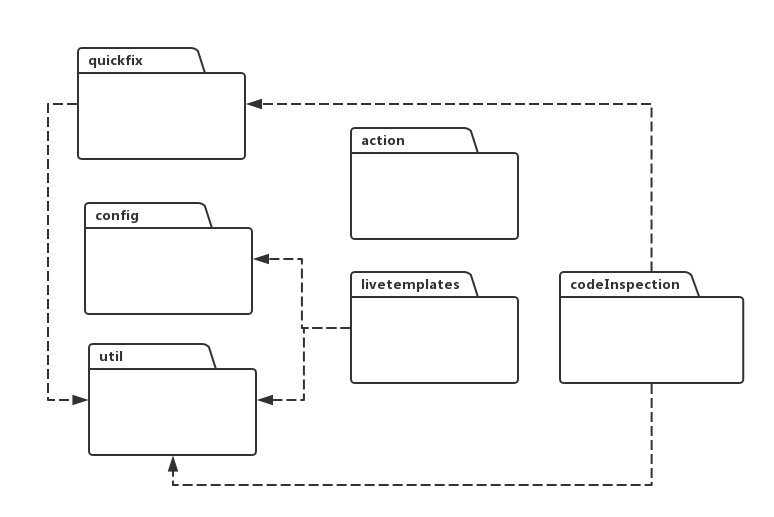


图3.3 系统框架结构图

表3.7 系统按规则划分模块设计表

|  |  |
| --- | --- |
| 模块名 | 主要职责 |
| assert检查模块 | 对开发人员当前开发内容完成assert检查项的实时代码检查、高亮提示以及日志插入 |
| exception检查模块 | 对开发人员当前开发内容完成exception检查项的实时代码检查、高亮提示以及日志插入 |
| 重要分支语句检查模块 | 对开发人员当前开发内容完成重要分支语句检查项的实时代码检查、提示以及日志插入 |
| 线程相关检查模块 | 对开发人员当前开发内容完成线程相关检查项的实时代码检查、高亮提示以及日志插入 |
| 文件相关检查模块 | 对开发人员当前开发内容完成文件相关检查项的实时代码检查、高亮提示以及日志插入 |
| 服务器相关检查模块 | 对开发人员当前开发内容完成服务器相关检查项的实时代码检查、高亮提示以及日志插入 |
| 数据库相关检查模块 | 对开发人员当前开发内容完成数据库相关检查项的实时代码检查、高亮提示以及日志插入 |
| 重要类及方法检查模块 | 对开发人员当前开发内容完成重要类以及方法检查项的实时代码检查、高亮提示以及日志插入 |

从图3.2系统顺序图可知，本系统可以按照流程进行如下表3.8所示的模块的划分。

表3.8 系统按流程划分模块设计表

|  |  |
| --- | --- |
| 模块名 | 主要职责 |
| actions | 提供开发人员进行自定义配置的界面，并将开发人员的配置保存 |
| providers | 完成inspection的注册，需要通过将provider在plugin.xml文件中完成 |
| inspections | 完成各个日志记录规则所对应的实时代码检查与高亮提示 |
| quickfixes | 实现多种日志语句的自动插入 |
| loggingutil | 定义各种日志框架类型并提供日志记录工具类 |

## 3.4 本章小结

本章内容主要由以下三部分构成：本插件系统的项目整体概述部分、需求分析部分以及系统的概要分析部分。第一部分描述了本插件系统的整体介绍以及核心模块简要介绍。第二部分通过用例图、用例描述、系统顺序图对功能需求进行了描述并描述了系统的非功能性需求。第三部分对本插件系统的框架结构以及系统按日志检查规则以及按流程划分的模块的描述以及模块职责的描述。

# 第四章 详细设计与实现

## 4.1 项目子模块概述

由第三章可知，本插件系统主要按功能点可以分为八大模块，分别为：assert检查模块、exception检查模块、重要分支语句检查模块、线程相关检查模块、文件相关检查模块、服务器相关检查模块、数据库相关检查模块、重要类及方法检查模块。本人负责其中的线程相关检查模块、数据库相关检查模块以及重要类及方法检查模块。本章将对本人负责的这三个模块的详细设计和实现进行介绍。

## 4.2项目的详细设计

### 4.2.1 重要类以及方法检查模块详细设计

**重要类以及方法检查模块类图及设计描述**

../../Downloads/criticalope.pdf

图4.1 重要类以及方法检查模块类图

由以上类图4.1可知，首先由CriticalInspectionConfigAction类提供给开发人员添加自定义配置的按钮以及界面，然后将配置储存在系统的文件夹中，然后是线程相关操作的代码检查、提示以及代码自动插入部分。CriticalOpeProvider类实现InspectionToolProvider接口，可以将继承了BaseJavaLocalInspectionTool的CriticalOpeInspection类绑定给CriticalOpeProvider类，然后通过将CriticalOpeProvider类配置在plugin.xml中完成CriticalOpeInspection的配置。CriticalOpeInspection类是完成对开发人员当前开发内容进行自定义配置的规则检查以及提示的部分，需要通过读取CriticalInspectionConfigAction类保存在系统文件夹中的日志规则来进行检查，然后对于需要进行日志记录语句插入的部分通过类中包含的多个Quickfix来完成，系统中的每个Quickfix都实现LocalQuickfix，并实现其最重要的applyFix方法，这样就能在开发人员选择某种提示时对开发人员当前开发内容进行更改。一个Quickfix对应某个特定的日志框架类型以及日志级别的日志语句插入方案，上图中分别以java.util.logging、Log4j以及Slf4j中的三种框架中的三种日志级别为例子。每个Quickfix对当前开发内容进行修改的真正实现通过CriticalOpeLoggingUtil类来完成，CriticalOpeLoggingUtil类需要利用LoggingType枚举类型来得到各个日志框架类型的语句模板，也需要LoggingUtil来获取当前系统时间、得到类名以及日志等级对应字符串的通用日志语句构成所需要的内容。

**重要类以及方法检查顺序图**

**../../Downloads/critical%20sd.pdf**

图4.2 重要类以及方法检查模块顺序图

以上图4.2中仅以Log4j框架Fatal级别的满足重要类以及方法检查规则的日志语句插入为例。

### 4.2.2 线程相关检查模块

**线程相关检查模块类图及设计描述**

**../../Downloads/thread.pdf**

图4.3 线程相关检查模块类图

由以上类图4.3可知，ThreadProvider类实现InspectionToolProvider接口（可以绑定类），ThreadInspection类继承BaseJavaLocalInspectionTool用于进行代码实时检查，所以通过将ThreadProvider类配置在plugin.xml中可以完成ThreadInspection的配置。ThreadInspection类是完成对开发人员当前开发内容进行实时线程相关操作的检查以及提示的部分，对于符合检查项的需要进行日志记录语句插入的部分通过ThreadInspection类中包含的多个不同的Quickfix来完成，系统中的每个Quickfix都实现LocalQuickfix，通过实现其最重要的applyFix方法来当开发人员选择某种提示时完成对开发人员当前开发内容进行更改。一个Quickfix对应某个特定的日志框架类型以及日志级别的与线程相关的日志语句插入方案，上图中分别以java.util.logging、Log4j以及Slf4j中的三种框架中的三种日志级别为例子。每个Quickfix对当前开发内容进行修改的真正实现通过ThreadLoggingUtil类来完成，ThreadLoggingUtil类需要利用LoggingType枚举类型来得到各个日志框架类型的语句模板，也需要LoggingUtil来获取当前系统时间、得到类名以及日志等级对应字符串这些日志记录语句所必需且通用的信息。

**线程相关检查模块顺序图**

**../../Downloads/thread%20sd.pdf**

图4.4 线程相关检查模块顺序图

以上图4.4中仅以Log4j框架Fatal级别的与线程有关的日志语句插入为例。

### 4.2.3 数据库相关检查模块

**数据库相关检查模块类图及设计描述**

**../../Downloads/jdbc.pdf**

图4.5 数据库相关检查模块类图

由以上类图4.3可知，JDBCProvider类实现InspectionToolProvider接口用来绑定类，JDBCInspection类继承BaseJavaLocalInspectionTool用于进行代码实时检查，通过将JDBCProvider类配置在plugin.xml中则完成JDBCInspection的配置。JDBCInspection类是完成对开发人员当前开发内容进行实时数据库相关操作的检查以及提示的部分，对于JDBC的连接以及执行SQL语句这些需要进行日志记录语句插入的部分通过JDBCInspection类中包含的多个不同的Quickfix来完成，系统中的每个Quickfix都实现LocalQuickfix，通过实现其中核心的applyFix方法来实现党开发人员选择某种提示时完成对开发人员当前开发内容完成更改。一个Quickfix对应某个特定的日志框架类型以及日志级别的与数据库相关的日志语句插入方案，上图中分别以java.util.logging、Log4j以及Slf4j中的三种框架中的三种日志级别中与数据库连接有关日志记录语句为例子。每个与数据库连接有关的Quickfix对当前开发内容进行修改的真正实现通过JDBCConnectionLoggingUtil类来完成，每个与数据库SQL语句执行有关的Quickfix对当前开发内容进行修改的真正实现通过JDBCExecuteLoggingUtil类来完成。JDBCConnectionLoggingUtil类以及JDBCExecuteLoggingUtil类都需要利用LoggingType枚举类型来得到各个日志框架类型的语句模板，也需要LoggingUtil来获取当前系统时间、得到类名以及日志等级对应字符串这些日志记录语句所必需且通用的信息。

**数据库相关检查模块顺序图**

../../Downloads/jdbc%20sd.pdf

图4.6 数据库相关检查模块顺序图

以上图4.6中仅以Log4j框架Fatal级别的与数据库有关的日志语句插入为例。

## 4.3 项目的实现

### 4.3.1 线程相关检查模块详细实现

线程相关检查模块最核心的代码片段是ThreadInspection类中对开发人员编写的Java代码的与线程有关的检查部分和ThreadLoggingUtil类中具体插入日志记录语句的部分。

**线程相关代码检查部分**

|  |
| --- |
| public void visitMethodCallExpression(PsiMethodCallExpression expression) {  super.visitCallExpression(expression);  if(isNullLiteral(expression)){  return;  }  final PsiReferenceExpression methodExpression = expression.getMethodExpression();  final String methodName = methodExpression.getReferenceName();  //check method name  if (methodName != null && (methodName.equals("start")||methodName.equals("join"))){  //check call class type  if (isThread(expression)){  holder.registerProblem(expression,  DESCRIPTION\_TEMPLATE, threadJavaInfoQuickfix,threadJavaFineQuickfix,threadJavaFinerQuickfix,threadJavaFinestQuickfix,threadJavaConfigQuickfix,threadJavaWarningQuickfix,threadJavaSevereQuickfix);  }  }else if (methodName != null && methodName.equals("run")){  final PsiMethod method = expression.resolveMethod();  if (method == null){  return ;  }  final PsiClass psiClass = method.getContainingClass();  final PsiClass father = psiClass.getSuperClass();  if (father.getName().equals("Thread")){  holder.registerProblem(expression,  DESCRIPTION\_TEMPLATE, threadJavaInfoQuickfix,threadJavaFineQuickfix,threadJavaFinerQuickfix,threadJavaFinestQuickfix,threadJavaConfigQuickfix,threadJavaWarningQuickfix,threadJavaSevereQuickfix);  }}} |

图4.7 ThreadInspection类代码

由于对线程的start、join和run方法调用语句进行检查，因此重写visitMethodCallExpression方法。以”thread1.start();”语句为例子，PsiMethodCallExpression对应的部分是“thread1.start()”，首先得到PsiReferenceExpression，对应的部分是”thread1.start”，通过PsiReferenceExpression可以得到当前语句方法名”start”。此时可以先判断当前检查方法调用语句的方法名是不是start或是join，如果判断为真，那么接下来只要判断这个方法的调用者类型是不是java.lang.Thread，用ThreadInspection中的私有方法isThread进行判断。如果方法的调用类型是java.lang.Thread类且调用的方法名是start或join，则进行registerProblem，传入的多个quickfix实例对应多个日志插入方案。同理，else语句中是对于“thread1.run();”为例子的语句进行检查，首先判断方法名字是不是run，然后将PsiMethodCallExpression这个方法表达式解析成方法，得到方法调用者的类，可以通过调用者的类得到它的父类，判断父类是不是java.lang.Thread类，如果都满足，则同上进行registerProblem。

**线程相关日志语句插入部分**

|  |
| --- |
| PsiMethodCallExpression methodExpression = (PsiMethodCallExpression) problemDescriptor.getPsiElement();  final PsiElement semicolon = methodExpression.getNextSibling();  final CodeStyleManager codeStyleManager = CodeStyleManager.getInstance(project);   PsiElementFactory factory = JavaPsiFacade.getInstance(project).getElementFactory();  //get thread type  final PsiMethod method = methodExpression.resolveMethod();  final String methodName = method.getName();  final PsiExpressionList argumentList = methodExpression.getArgumentList();  String arguments = "without arguments";  if (argumentList != null){  PsiExpression[] expressions = argumentList.getExpressions();  if (expressions.length > 0){  arguments = " with argument: ";  for (PsiExpression expression:expressions){  arguments = arguments + "\"+"+expression.getText()+"+\", ";  }  arguments = arguments.substring(0,arguments.length()-4);  }  }   final PsiClass containingClass = method.getContainingClass();  final String className = containingClass.getQualifiedName();  final PsiClass currentFileClass = PsiTreeUtil.getParentOfType(methodExpression, PsiClass.class);  final String currentFileClassName = currentFileClass.getName();  String logContent = String.format("%s%s%s %s Thread: \"+\"%s\"+Thread.currentThread().getId()+\"%s %s)",type.getFormatBeginner(type), currentFileClassName,LoggingUtil.getLevelStringById(level,typeId), LoggingUtil.getCurrentTimeStr(), className,methodName,arguments);  PsiMethodCallExpression logCall =  (PsiMethodCallExpression) factory.createExpressionFromText(logContent, null);  logCall.addAfter(semicolon,logCall);  methodExpression.addAfter(logCall,methodExpression.getParent()); |

图4.8 ThreadLoggingUtil类代码

由图4.7可知，我们在registerProblem处传入的是PsiMethodCallExpression，因此可以先进行强制类型转换得到这个PsiMethodCallExpression，由图4.7下面的描述可知，PsiMethodCallExpression并不包含代码语句末尾的逗号，因此，可以通过getNextSibling方法得到逗号。创建一个PsiElementFactory为以后的插入语句构建做准备。通过解析方法调用表达式得到PsiMethod，因此就可以得到这个方法的方法名、参数以及调用者的类。判断方法传入的参数是否为空，如果为空，则在日志语句中输入调用方法不含参数，如果传入的参数不为空，则将参数都加入到待插入的日志语句中。通过方法调用者的类可以得到方法调用者的类名全称，通过工具类PsiTreeUtil得到当前检查语句所在的类名然后形式化构建日志输出语句字符串。下一步，将日志语句字符串通过前面创建的PsiElementFactory构建为一个方法调用表达式，将前面得到的逗号插入到当前新构建的PsiMethodCallExpression之后，然后将这个整体加入到当前正在检查的语句之后。

### 4.3.2数据库相关检查模块详细实现

数据库相关检查模块最核心的代码片段是JDBCInspection类中对开发人员编写的Java代码的与数据库相关的检查部分和JDBCConnectionLoggingUtil和JDBCExecuteLoggingUtil类中具体插入日志记录语句的部分。

**数据库相关代码检查部分**

**1.检查数据库连接部分**

|  |
| --- |
| public void visitAssignmentExpression(PsiAssignmentExpression expression) {  super.visitAssignmentExpression(expression);  if(isNullLiteral(expression)){  return;  }  final IElementType opSign = expression.getOperationTokenType();  if (opSign == JavaTokenType.EQ){  final PsiExpression lConn = expression.getLExpression();  if (expression.getRExpression().textContains('.')) {  final PsiMethodCallExpression rGetConn = (PsiMethodCallExpression) expression.getRExpression();  //check left  if (lConn != null && lConn.getType().toString().equals("PsiType:Connection")) {  final PsiReferenceExpression methodExpression = rGetConn.getMethodExpression();  final String methodName = methodExpression.getReferenceName();  //check right  //check method name  if (methodName != null && methodName.equals("getConnection")) {  //check call class  final PsiMethod method = rGetConn.resolveMethod();  final PsiClass containingClass = method.getContainingClass();  final String className = containingClass.getQualifiedName();  if (className != null && className.equals("java.sql.DriverManager")) {  holder.registerProblem(expression, DESCRIPTION\_TEMPLATE, jdbcJavaInfoQuickfix);  }  }}}}} |

图4.9 JDBCInspection类代码

由于对数据库的连接方法调用语句进行检查，因此重写visitAssignmentExpression方法。以” conn = DriverManager.getConnection(url, "user", "pw");”语句为例子，visitAssignmentExpression对应的部分是“conn = DriverManager.getConnection(url, "user", "pw")”，首先确认这个赋值表达式的操作符是不是等号，然后分别得到赋值表达式的左边表达式” conn”以及右边表达式” DriverManager.getConnection(url, "user", "pw")“，接下来对右边表达式进行判断。首先判断右边表达式是否含有调用符号，如果含有调用符号那么就说明右边表达式是一个方法调用表达式，这样就可以将其进行强制类型转换得到对应的方法调用表达式，类似4.3.1章中的描述，由方法调用表达式可以得到被调用方法的名字以及调用者的类，通过简单的过程可以进一步得到方法调用者的类名字符串”java.sql.DriverManager“，然后直接通过调用者类名和被调用方法名“getConnection”的比对就可以知道右边表达式是否满足数据库方法连接的语法。接下来，判断左边表达式的类型是不是“PsiType:Connection”可以得出当前正在检查的代码语句是不是数据库的连接方法调用语句，如果满足，则进行registerProblem，传入的多个quickfix实例对应多个日志插入方案。

**2.检查数据库SQL执行部分【以executeQuery方法为例】**

|  |
| --- |
| public void visitDeclarationStatement(PsiDeclarationStatement statement) {  super.visitDeclarationStatement(statement);  if (statement == null){  return;  }  for (PsiElement declaredElement:statement.getDeclaredElements()){  if (declaredElement instanceof PsiVariable){  PsiVariable variable = (PsiVariable) declaredElement;  PsiExpression initializer = variable.getInitializer();  if (initializer != null) {  //1. execute query  if (initializer.getType().toString().equals("PsiType:ResultSet") && initializer instanceof PsiMethodCallExpression) {  final PsiMethodCallExpression methodCallExpression = (PsiMethodCallExpression) initializer;  final PsiReferenceExpression queryMethodExpression = methodCallExpression.getMethodExpression();  final String queryMethodName = queryMethodExpression.getReferenceName();  if (queryMethodName != null && queryMethodName.equals("executeQuery")) {  final PsiMethod queryMethod = methodCallExpression.resolveMethod();  final PsiClass queryClass = queryMethod.getContainingClass();  final String fullClassName = queryClass.getQualifiedName();  if (fullClassName != null && fullClassName.equals("java.sql.Statement")) {  holder.registerProblem(statement, DESCRIPTION\_TEMPLATE, jdbcQueryJavaInfoQuickfix);  }  } |

图4.10 JDBCInspection类代码

由于对数据库的SQL执行语句进行检查，因此重写visitDeclarationStatement方法。以” ResultSet rs = stmt.executeQuery(sql);”语句为例子，visitDeclarationStatement对应的部分是“ResultSet rs = stmt.executeQuery(sql);”，首先得到声明语句左边的部分，即声明的元素“rs”，类型是变量，将声明的元素进行强制类型转换为PsiVariable后可以通过变量得到将这个变量初始化的语句“stmt.executeQuery(sql)”，通过初始化表达式可以得到所声明变量的类型，判断初始化表达式是否是方法调用表达式，如果为方法调用表达式，用以上所描述过的步骤可以得到方法调用者的类名以及被调用的方法名，通过字符串比对即可判断此声明语句是否为SQL语句的执行语句，如果为SQL执行语句，则进行registerProblem，传入的多个quickfix实例对应多个日志插入方案。

**数据库相关日志语句插入部分【以SQL执行语句相关插入为例】**

|  |
| --- |
| PsiDeclarationStatement declarationStatement = (PsiDeclarationStatement) problemDescriptor.getPsiElement();  PsiElementFactory factory = JavaPsiFacade.getInstance(project).getElementFactory(); //get current class final PsiClass currentFileClass = PsiTreeUtil.getParentOfType(declarationStatement, PsiClass.class); final String currentFileClassName = currentFileClass.getName(); PsiElement[] elements = declarationStatement.getDeclaredElements(); if (elements.length!= 0){  PsiVariable variable = (PsiVariable) elements[0];  PsiExpression initializer = variable.getInitializer();  if (initializer != null) {  final PsiMethodCallExpression queryExpression = (PsiMethodCallExpression) initializer;  final PsiExpressionList argumentList = queryExpression.getArgumentList();  final PsiReferenceExpression methodExpression = queryExpression.getMethodExpression();  final String methodName = methodExpression.getReferenceName();  String sql = "null";  if (argumentList != null){  //…将方法调用中的参数插入日志输出语句中，类似图4.8中相关代码  }  //…代码插入部分，类似图4.8中相关代码 } |

图4.11 JDBCExecuteLoggingUtil类代码

由图4.10可知，我们在registerProblem处传入的是PsiDeclarationStatement，因此可以先进行强制类型转换得到这个PsiDeclarationStatement，由图4.10下面的描述可知，PsiDeclarationStatement并不包含代码语句末尾的逗号，因此，可以通过getNextSibling方法得到逗号。创建一个PsiElementFactory为以后的插入语句构建做准备。首先通过工具类PsiTreeUtil得到当前检查语句所在的类名，然后同前小段描述可知，可以逐步由声明语句得到初始化语句中调用的方法名以及传入的参数，由4.3.1小节描述可知，可由以上得到的值构建日志插入语句以及完成在正确位置的插入。

### 4.3.3用户自定义配置模块详细实现

**用户自定义配置部分**

|  |
| --- |
| public class CriticalInspectionConfigAction extends AnAction {  final String fileNameForMethod = "/Users/chentiange/Desktop/okplugin/resources/criticalMethod";  final String fileNameForClass = "/Users/chentiange/Desktop/okplugin/resources/criticalClass";  @Override  public void actionPerformed(AnActionEvent anActionEvent) {  Project project = anActionEvent.getProject();  String newRule = Messages.showInputDialog(project,"Please input your critical class or method","Critical rule config",Messages.getQuestionIcon());  newRule += "\n";   if (newRule != null){  try {  //add new rule in file  FileOutputStream fileOutputStream = new FileOutputStream(new File(fileNameForClass),true);  if (newRule.contains(":")){  fileOutputStream = new FileOutputStream(new File(fileNameForMethod),true);  }  fileOutputStream.write(newRule.getBytes());  fileOutputStream.flush();  fileOutputStream.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  } } |

图4.12 CriticalInspectionConfigAction类代码

|  |
| --- |
| <actions>  Add config  <action class="com.intellij.codeInspection.actions.CriticalInspectionConfigAction" id="plugintest" text="My plugin" description="a plugin" icon="/icons/icons8-plus-40.png">  <add-to-group group-id="MainToolBar" anchor="last"/> </action> |

图4.13 plugin.xml代码

在plugin.xml的配置中，icon是与CriticalInspectionConfigAction对应提供给用户的图标，group-id设置成MainToolBar是将图标设置在IntellJ代码编辑部分上方的主工具栏中，anchor设置成last是指将图标放在主工具栏的最后。CriticalInspectionConfigAction的图标在开发人员点击时设置了一个输入框，获取输入到的值放入变量newRule中，如果所输入的newRule不为空，根据输入的格式区别输入的是重要的类还是重要的方法，按类别存入不同的文件中。

**用户自定义配置检查部分**

|  |
| --- |
| line = bufferedReader.readLine(); while (line != null){  String currentClassWholeName = LoggingUtil.getCurrentClassWholeName(aClass);  if (Pattern.matches(line,currentClassWholeName)){  //get all method  PsiMethod[] methods = aClass.getMethods();  for (PsiMethod method:methods){  if (!method.getName().equals("main")) {  visitMethodInClass(method);  }  }  }  line = bufferedReader.readLine(); }  // ………  private void visitMethodInClass(PsiMethod method) {  super.visitMethod(method);  holder.registerProblem(method,DESCRIPTION\_TEMPLATE,criticalOpeJavaInfoQuickfix); } |

图4.14 CriticalOpeInspection类代码

由于要支持检查自定义的重要的类并且将自定义的重要的类中所有除main函数以外的方法调用都记录，因此，需要重写visitClass方法。因为还要支持检查自定义的重要的方法，因此也需要重写visitMethod方法。这两者的检查判断过程相似，此处重点描述visitClass方法的内容。首先读保存了自定义类的文件，逐行比对每一条自定义的类，得到当前正在扫描的语句所在的类的全名，如果当前插件正在扫描的类是自定义中重要的类，则获取这个类所有的方法PsiMethod，逐个方法判断是否为main函数，将所有除main函数的方法传入visitMethodInClass中进行registerProblem，传入的多个quickfix实例对应多个日志插入方案。

**用户自定义配置相关日志语句插入部分**

|  |
| --- |
| PsiMethod psiMethod = (PsiMethod) problemDescriptor.getPsiElement();  //…通过Psimethod获取方法信息  String levelString = LoggingUtil.getLevelStringById(level, typeId); String logContent = String.format("%s%s%s %s %s\")",type.getFormatBeginner(type), currentClassWholeName,levelString, LoggingUtil.getCurrentTimeStr(),stmt); System.out.println(logContent); PsiMethodCallExpression logCall =  (PsiMethodCallExpression) factory.createExpressionFromText(logContent, null);  PsiCodeBlock methodBody = psiMethod.getBody(); if (methodBody != null){  final PsiStatement[] oldStatements = methodBody.getStatements();  if (oldStatements.length >= 0){  PsiStatement firstStatement = oldStatements[0];  final PsiElement semicolon = firstStatement.getLastChild();  logCall.addAfter(semicolon,logCall);  psiMethod.addBefore(logCall,firstStatement);  //add return  PsiType returnType = psiMethod.getReturnType();  if (returnType != PsiType.VOID){  PsiReturnStatement[] returnStatements = PsiUtil.findReturnStatements(psiMethod);  for (PsiReturnStatement returnStatement:returnStatements){  String returnstr = returnStatement.getText();  returnstr = returnstr.replace("return ","");  returnstr = returnstr.replace(";","");  String returnstmt = methodWholeName+" return \"+"+returnstr+"+\"";  String returnlogContent = String.format("%s%s%s %s %s\")",type.getFormatBeginner(type), currentClassWholeName,levelString, LoggingUtil.getCurrentTimeStr(),returnstmt};  PsiMethodCallExpression returnlogCall =  (PsiMethodCallExpression) factory.createExpressionFromText(returnlogContent, null);  returnlogCall.addAfter(semicolon,returnlogCall);  psiMethod.addBefore(returnlogCall,returnStatement);  }  }}} |

图4.15 CriticalOpeLoggingUtil类代码

由图4.14可知，我们在registerProblem处传入的是PsiMethod，因此可以先进行强制类型转换得到这个PsiMethod，由前面的描述可知通过PsiMethod可以获得被调用方法的名称和方法所有的参数，因此可以构建应当在当前方法内部代码段中插入的第一句字符串，同样通过factory构建可以插入代码中的方法调用表达式。然后通过PsiMethod得到方法的内部语句PsiStatement，如果内部语句为空，这个方法调用没有意义，因此不需要进行日志记录。如果内部语句不为空，则得到内部语句的第一条，将新构建的日志记录语句插入到内部第一条语句之前。通过PsiMethod可以得到方法的返回值类型，如果返回值类型为void，则此方法没有任何添加记录返回值的日志语句的必要，如果当前方法有返回值，由于一个方法内部可能会有多个返回点，因此利用PsiMethod得到所有的返回语句PsiReturnStatement，然后去掉语句中的逗号等非返回值成分得到返回值，利用返回值构建新的返回值日志记录语句，并分别将这些返回值日志语句插入到其对应的PsiReturnStatement之前。

## 4.4 插件运行效果截图【以重要类以及方法为例】

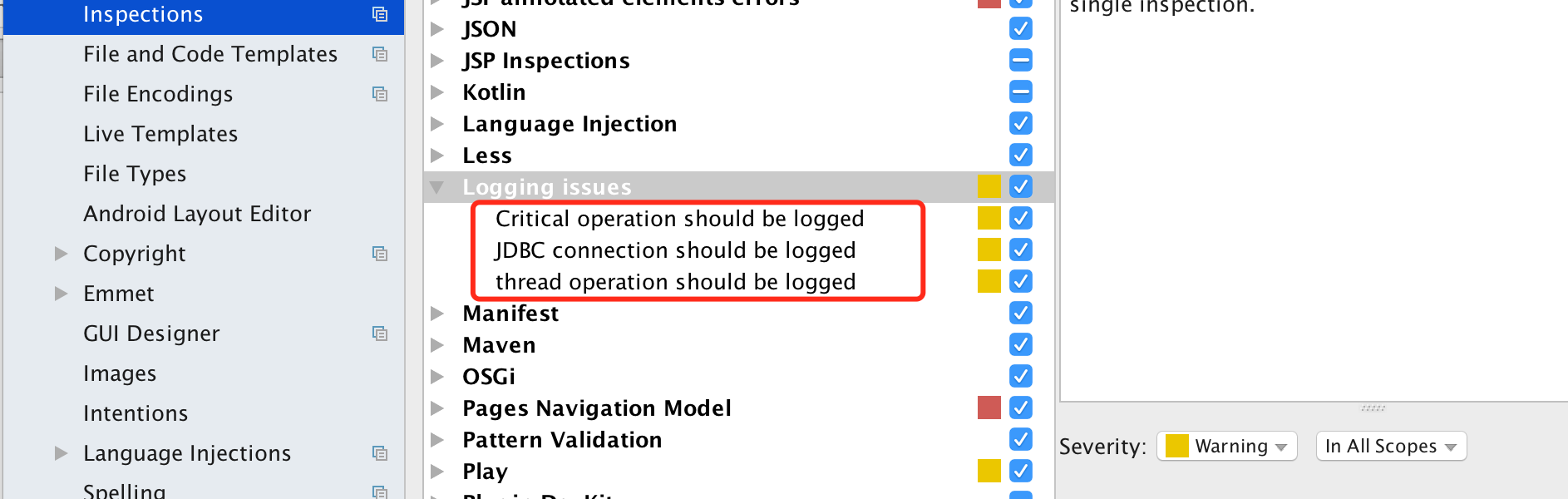


图4.16 IntelliJ Inspection截图

如上图4.16所示，以上三个模块的实现会使得以上三个模块对应的日志规则检查项注册到IntelliJ的检查项中。

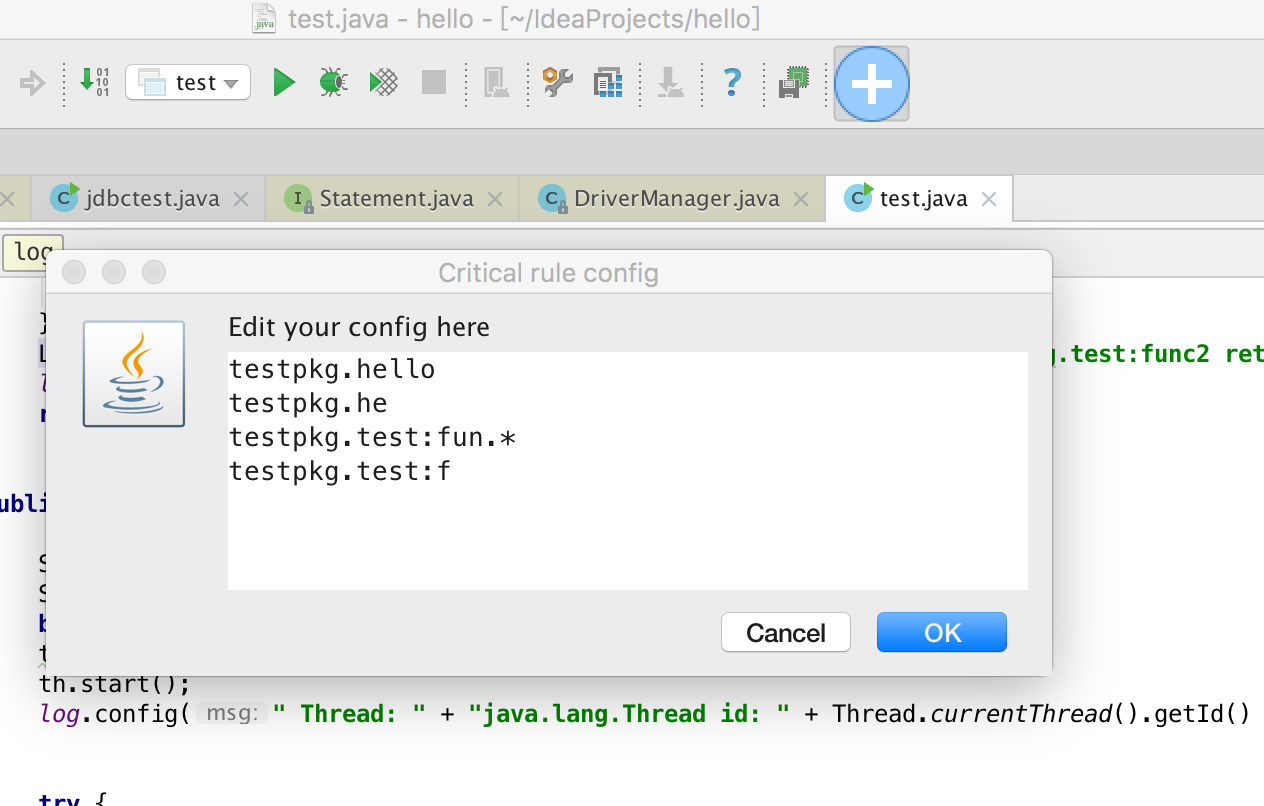


图4.17 重要类以及方法配置截图

如上图4.17所示，IntelliJ主工具栏末尾的蓝色加号图标为重要类以及方法配置的按钮，点击后即显示下面的配置输入框。

如下图4.18所示，所配置的重要方法即使只在某一个方法返回处缺少日记记录语句本插件也会给出高亮提示。

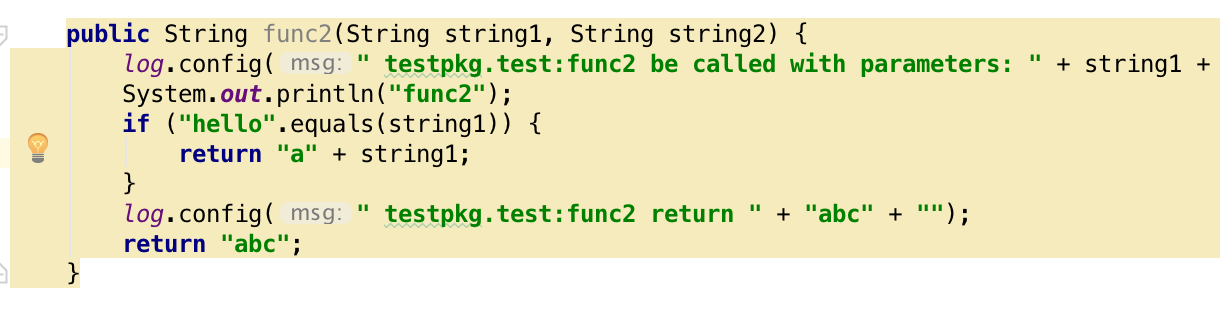


图4.18 重要方法检查与高亮提示截图

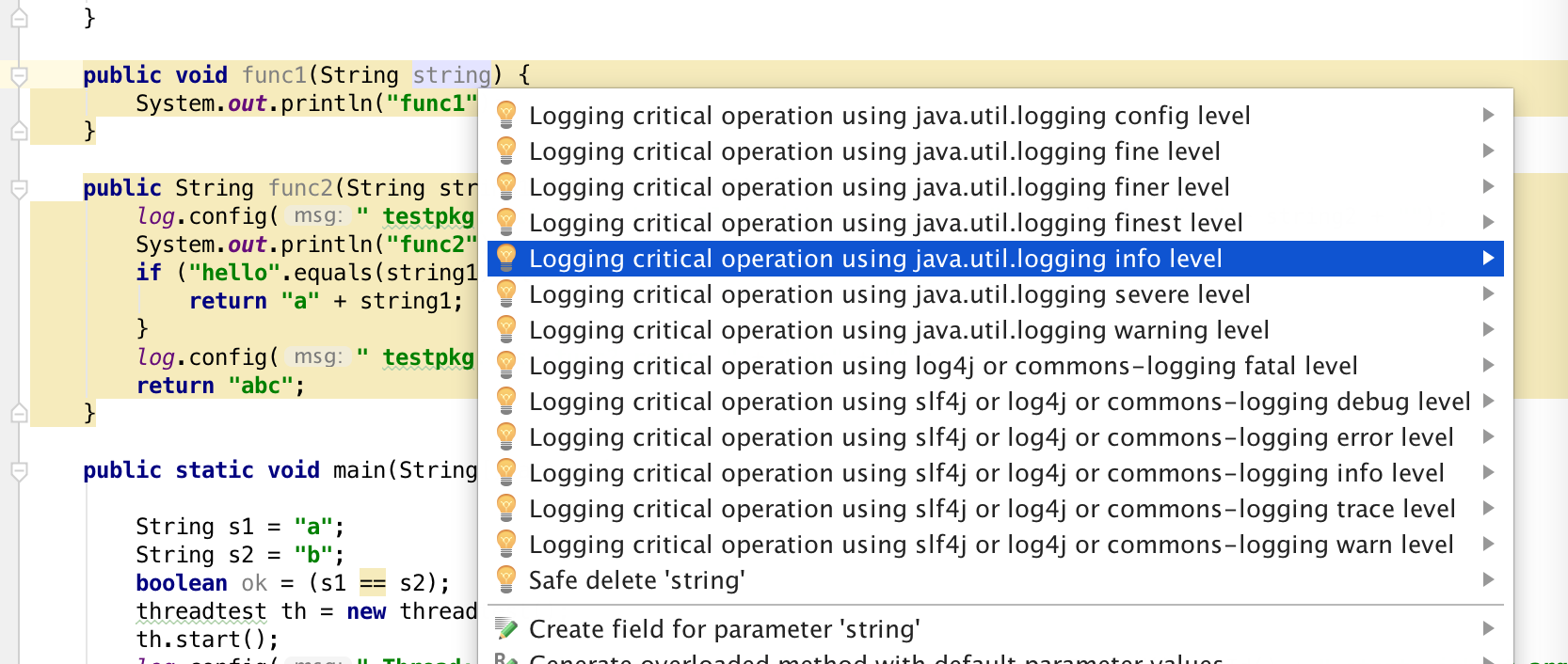


图4.19 重要方法日志记录语句选择截图

如上图4.19所示，在高亮提示处会给出应该插入的日志记录语句的选择。

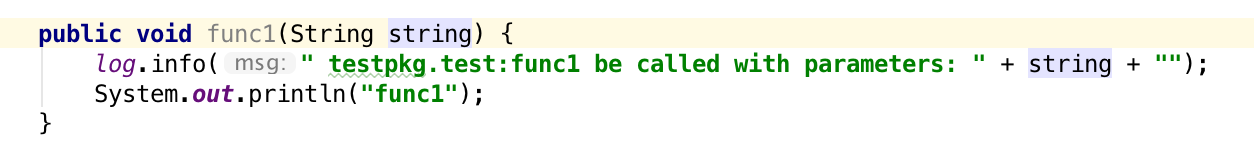


图4.20 重要方法已添加日志截图

如上图4.20所示，在已经添加日志的代码处高亮提示关闭。

## 4.5 本章小结

本章简短地整体介绍了整个插件项目按功能点的模块划分，然后针对本人负责的线程相关检查模块、数据库相关检查模块以及重要类以及方法检查模块进行了详细设计与实现的描述。在详细设计部分，分别对每一个模块用类图和顺序图进行了描述，并提供了文字说明。在设计实现的部分，分别对每一个模块的核心代码进行了划分、概括与代码逐行解释。

# 第五章 总结与展望

## 5.1 总结

如今，随着社会的发展与技术的进步，软件产品的规模日益扩大，人们对于软件产品的质量要求也越来越高。日志记录可以跟踪系统的状态，记录开发人员需要得到的运行信息等，因此，日志记录是保证软件产品质量的有效手段。为了能够帮助开发人员在使用IntellJ IDEA进行Java语言开发时能提高日志记录稳定性，本团队开发了基于Java语言的IntellJ日志记录智能提示与自动插入的插件。本团队的所有成员都没有经历过插件的开发，IntellJ平台的插件相关的文档和资料非常缺乏。由于平时在学校开发的项目较小且对软件质量的要求不是特别高，团队成员在以往的项目开发过程中都极少使用日志记录语句。在项目初期，本团队成员并不了解应该在何处记录日志以及日志记录中应该有哪些信息。

因此，我们遇到了不知道日志应该如何记录这个理论问题以及如何进行IntellJ插件开发这个技术问题。首先我们通过大量论文阅读以及上网搜索，对于日志记录有了一定的了解，大家一起制定了一些基础的通用的日志记录检查的规则，这些规则构成了我们实现的功能目标。下一步遇到的就是IntellJ插件开发教程缺乏的问题，由于找不到相关的教程与技术文章，IntellJ插件官方文档也不够详细与完整，因此我们只能通过阅读官方给出的示例代码以及IntellJ插件API的源码进行插件开发的学习，由于缺乏对于整个API源码PSI相关类的了解，在插件开发的过程中也遇到了许多技术上的困难。

## 5.2 展望

本项目中只定义了一些基本的日志记录检查规则，这些规则对于许多项目来说不够完整，因为不同的项目不同的开发人员有自己特定的日志记录决策点，因此本插件中记录的可能只是这些开发人员需要进行日志记录的一部分，而且对于某一个具体的日志记录检查规则而言，在不同的项目中会有多种细小的规则的划分，按不同的场景会有多种不同过的处理方法，因此，在日志记录检查规则上，可以以本插件为基础，提供给不同的开发团队进行满足自己团队日志记录决策的实现，这样就能保证插件中的日志记录检查规则与当前开发团队所需要的日志记录决策相一致。

总的来说，本项目是对制作IntellJ日志记录有关插件的一次初步尝试，可以在本项目的基础上就日志记录检查规则的制定，多种编程语言的扩充以及日志插入语句的完善方面做出改进。

# 参考文献

[1] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? an empirical study on logging practices in industry. In *Proc. of ACM/IEEE ICSE*, 2014, <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/ICSE-2014-SEIP-Where-Do-Developers-Log-An-Empirical-Study-on-Logging-Practices-in-Industry.pdf>

[2]The Art of Logging, <https://www.codeproject.com/Articles/42354/The-Art%20-%20of-%20Logging>

[3] C Gülcü, The complete log4j Manual,Switzerland:QOS.ch, November 20th, 2002 ,1-12

[4]Code Inspection, https://www.jetbrains.com/help/idea/code-inspection.html

[5] Suppressing Inspection, <https://www.jetbrains.com/help/idea/2016.1/suppressing-inspections.html>

[6]操作PSI对象，http://ju.outofmemory.cn/entry/322651

[7] PSI Files, https://www.jetbrains.org/intellij/sdk/docs/basics/architectural\_overview/psi\_files.html

[8]File View Providers, <https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/file_view_providers.html>

[9]PSI Elements，<https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi_elements.html>

[10]Navigating the PSI，<https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/navigating_psi.html>

[11]PSI References，<https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi_references.html>

[12]Quick-Fixes for Code Issues，<https://www.jetbrains.com/help/resharper/Code_Analysis__Quick-Fixes.html>

[13]Commons Logging，<http://commons.apache.org/proper/commons-logging/>

[14]Logback Project，https://logback.qos.ch/index.html

# 致谢

这是大学四年的最后一个项目的结尾，非常感谢这近四年来所有教导过我的老师和帮助过我的同学，在他们的帮助下，让我从对于软件工程一无所知的少年学习成长成为了对软件开发过程有所了解、能通过自身学习来进行项目开发的少年。南京大学良好的学习氛围、校园环境以及多样化的课程也让我有了更多自学的兴趣，了解到了许多其他专业的知识，黄蕾辅导员对我的学习与生活也提供了许多帮助和指导。

对于最后的这个日志插件开发项目，非常感谢我们团队的指导老师荣国平老师，这个项目的选题是荣国平老师提出的建议，让我们了解到了当前市场环境中对于自动化日志记录提示工具的缺乏，因此给了我们一个有着重要意义和实践价值的项目研究方向。在选择自动化日志记录提示工具的表现形式时，也是荣国平老师提出了用IDE的插件的形式来进行开发，这样可以提高我们研究的这个项目的实用性。非常感谢荣国平老师在项目的整体方向上给我们团队的这些具有创造性和启发性的建议，也非常感谢荣国平老师和顾胜辉学长提供给我们团队的日志记录规范有关的论文资料。

最后，非常感谢我们团队的另外两名队友，邢程和熊定波。感谢队友们一起阅读与讨论日志规则有关的论文，互相提供信息和讲解。在项目的开发过程中，是队友们在一起查阅文档，阅读源码，辛劳地学习新技术。在遇到难题的时候，感谢队友们及时开会讨论解决问题，提高了团队的开发效率。

1. 1关于Elasticsearch帮助logging的信息，参见：https://www.elastic.co/solutions/logging [↑](#footnote-ref-1)
2. 2 关于Logstash的信息，参见：https://www.elastic.co/products/logstash [↑](#footnote-ref-2)
3. 3 关于Kibana的信息，参见：https://www.elastic.co/products/kibana [↑](#footnote-ref-3)
4. 4 关于百度云日志分析托管服务的信息，参见：https://cloud.baidu.com/solution/log\_analysis.html [↑](#footnote-ref-4)
5. 1 关于Log4j的信息，参见：https://logging.apache.org/log4j/2.x/ [↑](#footnote-ref-5)
6. 2 关于LOGBack的信息，参见：https://logback.qos.ch/ [↑](#footnote-ref-6)
7. 3关于java.util.Logging的信息，参见：https://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html [↑](#footnote-ref-7)
8. 4 关于commons-logging的信息，参见：https://commons.apache.org/proper/commons-logging/ [↑](#footnote-ref-8)
9. 5 关于SLF4J的信息，参见：https://www.slf4j.org/ [↑](#footnote-ref-9)
10. 1 关于IntelliJ IDEA的信息，参见：https://www.jetbrains.com/idea [↑](#footnote-ref-10)
11. 1 关于PSI的信息，参见：https://www.jetbrains.org/intellij/sdk/docs/basics/architectural\_overview/psi.html [↑](#footnote-ref-11)