中国科学院大学

University of Chinese Academy of Sciences

高级软件工程组队项目
项目报告

CoderAssistant
面向软件开发领域的问答机器人

所 选 主 题 ：聊天机器人

项 目 名 称 ：面向软件开发领域的
问答机器人

指 导 教 师 ：罗铁坚 教授

团 队 成 员 一：祝 捷（202128015059003）

团 队 成 员 二：闫熠光（202128015059004）

团 队 成 员 三：徐泽杉（2021E8015082015）

二〇二一年十二月

目录

1 项目概述

1.1 项目背景概述

本项目 CoderAssistant 是一个面向软件开发领域的问答机器人，选题主要是关于聊天机器人在特定领域的一个具体应用。目前，软件开发领域方兴正艾，每年都有数以百万计的员工进入互联网或软件公司任职，同时也诞生了非常多的关于协助软件开发的程序和工具。比如目前比较常用的软件开发协助工具包括 IDE 集成开发环境，Git 代码托管平台等等。但是尽管有了很多专业的工具，以及 Google 和 Stackoverflow 这些可以迅速从网上获取专业知识的平台和论坛，许多软件开发人员依然在编码和协作上存在诸多问题，程序研发效率低下，代码质量不高等等，同时他们在遇到了问题的时候往往很难快速地找到一个准确的答案。尽管目前有很多插件和工具都致力于解决这个问题，比如 VSCode 上的 Tabnine、Idea 上的 StackInTheFlow 以及 Copilot 等等，但是他们大多关注在代码补全以及代码生成等问题，并不能真正地帮助开发者找到高质量的解决问题的答案。为了能更好地协助软件开发人员高效、快速、自动化地找到高质量技术类型问题的答案，我们设计了 CoderAssistant 这个面向软件开发领域的问答机器人来帮助他们更快地找到技术问题的高质量的回答和解决方案。

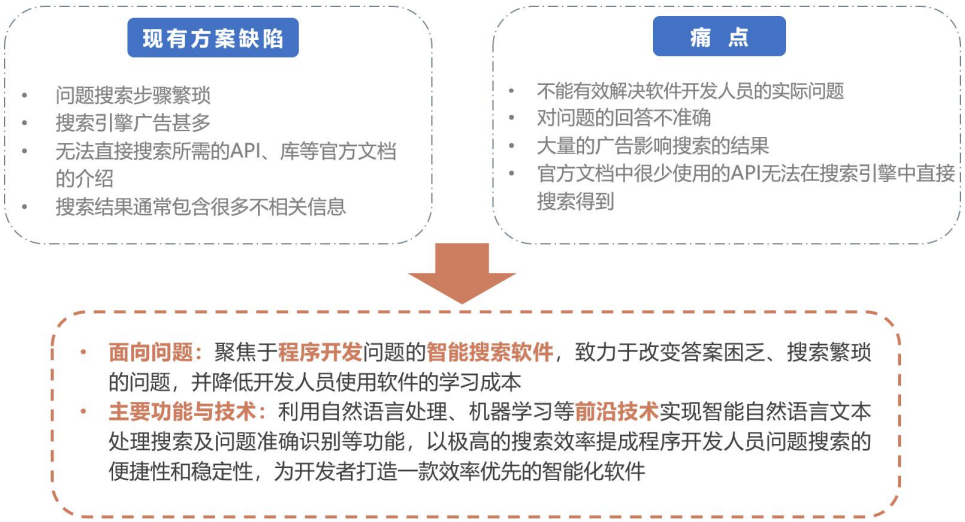


图 1.1.1 项目背景

本项目的一个具体情景是在平时写代码的过程中，我们经常会遇到一些不好解决的问题，我们就会手动复制一些关键的代码信息和报错信息在谷歌、StackOverflow 上搜索，但搜索后我们首先看到的很多时候是一大串的广告链接，其次才是真正我们需要的问题信息。而当我们点开链接后，又发现这个问题并不是我们遇到的问题，于是又需要不断地去一个一个点开看。

另一个情景是对于很多新手程序员或者开发者在使用一些平时不太常用的语言框架编写代码时也会遇到不少 bug，但开发者又因为对语言框架不够熟悉或者表达不够准确而不能写出一个很好的 query 语句到网络上搜索，因而难以找到高质量的回答和解决方案。

对于这样的一些场景，开发者需要花较多的时间去复制代码、错误信息以及写出必要的补充说明，才能到网络上进行搜索，而且他们的搜索结果也可能因为他们的查询输入不够准确而质量不高。这样的手动检索过程不仅消耗了宝贵的时间，使得开发者持续编程的状态受到打扰，还难以找到高质量的答案，给整体软件开发流程客观上构成了一个不小的阻碍。

图 1.1.2 目前存在的问题

Fibonacci sequence in Python3.2 [closed]

Asked 7 years, 6 months ago Active 7 years, 6 months ago Viewed 3k times

Google 已关闭此广告

-5

1

Closed. This question needs details or clarity. It is not currently accepting answers.

Want to improve this question? Add details and clarify the problem by editing this post.
Closed 7 years ago.

Improve this question

I really need your help. I know this question has been asked countless times already but I still cant find the answer...

I need to programm a fibonacci sequence recursively in a bla-bla.py file, this is what I've got so far:

```
print("Unendlicher Fibonacci-Generator Rekursiv")
def fib(n):

  if n == 0:
        return 0
  elif n == 1:
        return 1
  else:
        return fib(n-1) + fib(n-2)

for n in fib(n):
  print (str(n))
```

But I always get a NameError "name 'n' is not defined" which is driving me crazy... I just cannot understand how one can "print" the value of a variable in Python! Please help!

图 1.1.3 低质量提问/查询案例

1.2 项目简介

本项目 CoderAssistant 是一个面向软件开发领域的问答机器人，选题主要是关于聊天机器人在特定领域的一个具体应用，希望能像一个助手一样，帮助软件开发人员更高效快速地找到他们在编程过程中遇到的问题的解决方案。具体来说，CoderAssistant 是一个在 Java 语言最常用的集成开发环境 Idea 上的插件，可以帮助用户更方便地在 IDE 内部解决他们开发中遇到的问题，其主要核心特色在于：提供了自动化的基于错误信息和代码段的问答检索功能，以及 IDE 内的沉浸式体验。在传统的方式中，开发者遇到程序错误 Bug 时往往会经过如下这几个步骤：首先根据 bug 信息和代码进行分析，然后复制一些关键的错误信息以及代码段到 Google 或者 Stack Overflow 上，然后再肉眼过滤掉一些不相干的网页和广告，最后找到一个比较好的答案网页进行学习，直到解决 Bug。但这种方式的操作和步骤比较繁琐，切换的页面也比较多，而且不同水平的程序员他们 Debug 分析错误的能力也不同，所以 CoderAssistant 希望能够将这个过程自动化，帮助软件开发人员更快更准地找到高质量的问答和解决方案。

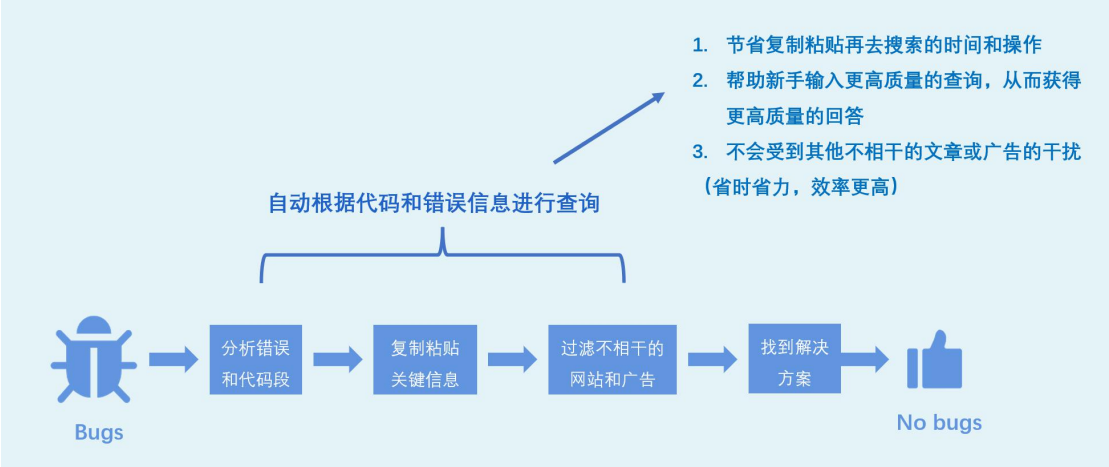


图 1.2 与传统的手工查询方式对比

1.3 应用场景与需求分析

1.3.1 应用场景分析

CoderAssistant 的一个具体的情景是在平时写代码的过程中，我们经常会遇到一些不好解决的问题，我们就需要手动复制一些关键的代码信息和报错信息在谷歌、StackOverflow 上搜索，但搜索后我们首先看到的很多时候是一大串的广告链接，其次才是真正我们需要的问题信息。而当我们点开链接后，又发现这个问题并不是我们遇到的问题，于是又需要不断地去一个一个点开看。这个过程是相当繁琐的，可以采用一些方式将其自动化的运行。

另一个情景是对于很多新手程序员或者开发者在使用一些平时不太常用的语言框架编写代码时也会遇到不少 bug，但开发者又因为对语言框架不够熟悉或者表达不够准确而不能写出一个很好的 query 语句到网络上搜索，因而难以找到高质量的回答和解决方案。这时我们也可以直接利用代码信息和错误信息，来帮助他们找到更好的答案。

对于这样的一些场景，开发者需要花较多的时间去复制代码、错误信息以及写出必要的补充说明，才能到网络上进行搜索，而且他们的搜索结果也可能因为他们的查询输入不够准确而质量不高。这样的手动检索过程不仅消耗了宝贵的时间，使得开发者持续编程的状态受到打扰，还难以找到高质量的答案，给整体软件开发流程客观上构成了一个不小的阻碍。

1.3.2 项目需求分析

本项目的主要面向用户为软件开发人员，项目从设计上可以支持各种编程语言，但是因为时间原因，在具体的开发上我们选择以 Java 作为主要的语言，选择 Intellij 公司的 Idea 集成开发环境作为我们的插件平台，如果有需要也可以拓展到其他平台和语言。在确定了目标用户群体后，我们对软件需求进行了分析建模，通过调研来保证整个软件的可用性和需求痛点更具体。这里以分层用例建模图的形式来描述我们项目的宏观目标、用户目标、以及具体的子目标，并将它们有机地结合联系在一起，从而更好的优化业务流程以实现软件价值。下图即产品的分层用例需求建模。

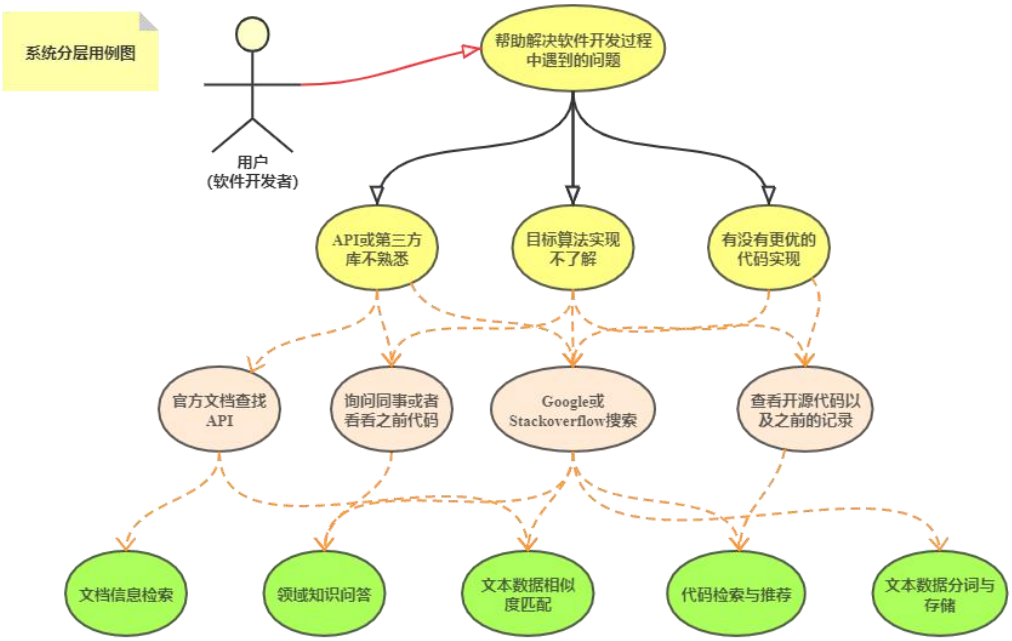


图 1.4 用例需求建模

1.3.3 项目定位

我们的项目并不是要做一个大而全的框架或者一个全过程的解决方案，而是针对软件开发中的一个具体问题，即软件开发问题答案的自动检索。

本项目拟开发一个基于 Idea 的插件软件，帮助软件工程师在开发过程中利用错误信息和代码语义信息自动进行 Stack Overflow 的问答检索，从而减少软件开发人员在外部搜集信息的时间，提升研发效率。

1.3.4 项目非功能性需求分析

项目的性能上能满足开发者的正常使用需求，相比于传统方式，CoderAssistant 可以帮助开发者更快的找到问题答案，在手动输入检索上可以减少一半的时间，在错误信息检索上可以提升 10 倍以上，使用代码语义生成的性能也在 1s 以内，符合实际性能要求。

项目可扩展性和可维护性也较高，如果需要开发其他语言或者其他框架和 IDE 上的插件软件，核心的后台模型算法只需要更新数据集训练即可，主要扩展成本在于重新开发其他语言 IDE 的插件。

1.4 研究现状与竞品分析

1.4.1 研究现状

目前，针对软件开发和软件工程领域的研究有一个很有趣的趋势，就是很多不同的任务都使用了机器学习和深度学习来改进软件开发的过程。例如代码搜索，抄袭检测，程序修复，文档(如 API 和问题/答案/标签)推荐等等。比如 CodeX（Copilot）就提供了基于自然语言描述生成代码的一个功能，但是因为 CodeX 的开发公司是 OpenAI，他们基于同样是他们开发的 GPT-3 预训练模型在 Github 的代码上进行 Fine-tuning 微调得到了 CodeX。不过这样的一些工作难以重复（运行训练成本比较高），而且对 API 试用接口申请也比较严格，一般人目前还无法使用这样的工具。另一类研究是比如 CodeBERT，他们同样使用大量代码文本和自然语言文本训练了 Code 上的预训练模型，并在一系列任务中取得了很好的成绩。还有一些工作是针对 Stack Overflow 上的社区内容质量改进的一些工作，比如 Xin Xia 提出的自动问题生成等等。

但是目前大部分研究工作并没有从根本上改变软件开发的流程，并不是说有了一个模型，就不需要程序员了。这个原因是目前软件开发中遇到的问题并不是在写代码本身这个过程上，而是集中在一些 Bug 错误调试或者问题分析上。CodeX/Copilot 生成的代码依然需要有经验的软件开发人员去调试，去修改，而不是说光靠模型就行了。所以本项目正是针对这样的一个问题和背景进行研究，并试图基于 Stack Overflow 等问答社区的资源进行改进。

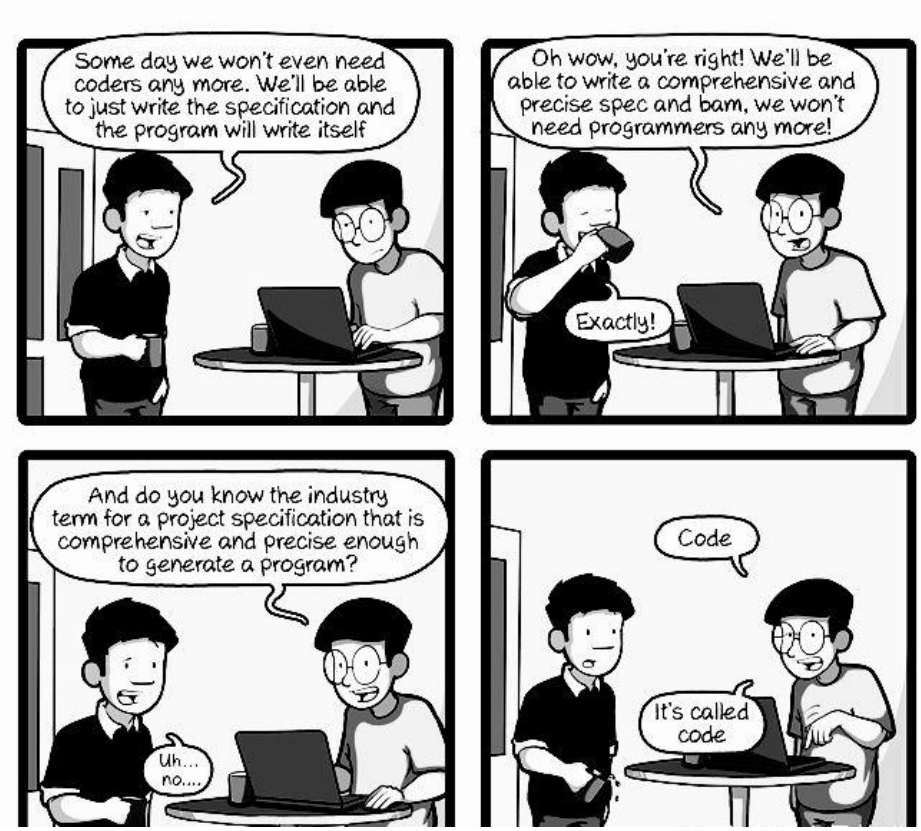


图 1.5 关于机器取代程序员的漫画（调侃）

1.4.2 行业和竞品分析

正如上文所述，目前软件开发智能助手还是一个比较新的行业领域。最知名的工作即 Codex/Copilot，是 OpenAI 用来支持 Github Copilot 的大规模预训练模型。正如 OpenAI 的联合创始人兼首席技术官 Greg Brockman 介绍的那样，“Codex 是 GPT-3 的后代”，Codex 基于 GPT-3 使用 Github 上的开源 code 数据进行了 Fine-Tuning，模型参数从 12M 到 12B 不等。具体模型结构与 GPT-3 类似，在代码生成任务（Text2Code）上取得了很好的效果。但是它也存在很多问题，比如学习效率很低，Codex 在训练过程中使用了大量代码，即使是经验丰富的开发人员，在整个职业生涯中也不会遇到这种数量级的代码，但 Codex 学习后的模型能力依然赶不上一个接受一两年计算机专业培养的学生。另一方面，Codex 显示出了一些失败或者反直觉的行为。比如 Codex 会生成语法错误或者未定义的代码，并且会调用未定义或超出范围的函数、变量和属性。此外，Codex 很难解析更长的、更抽象的、系统级的代码。从以上限制可以看出，Codex 还是

更倾向于“背代码”和做“代码组合”，而没有真正掌握多少编程知识，如果我们把编程能力分为 1. 编程语言知识（语法知识、API 功能等）；2. 需求和逻辑理解能力（理解需求和程序的逻辑）；3. 利用已有代码的能力（掌握一些常用实现）；4. 错误调试和信息收集能力（遇到错误 Bug 去网上进行检索找到解决方案）。Codex 只在第三点上展现出了强大的能力，其他的能力要么完全没有（比如错误调试），要么比较弱。因此，CodeX/Copilot 并没有从根本上改变软件开发的流程，并不是说有了这样一个模型，就不需要程序员了。目前软件开发中遇到的很多问题并不是说程序员写不出来代码或者写的很慢，而是写的代码能不能跑，能不能达到目标，怎么去做 Bug 错误调试和问题分析。所以 CodeX/Copilot 生成的代码依然需要有经验的软件开发人员去调试，去修改，而不是说光靠模型就行了。所以本项目正是针对这样的一个问题和背景进行研究，并试图基于 Stack Overflow 等问答社区的资源进行改进。CoderAssistant 与 CodeX/Copilot 的关系严格意义上并不是相互替代的竞品关系，而是一种互补和相互辅助的关系。

1.5 项目可行性分析

1.5.1 技术可行性

技术可行性分析是根据用户提出的系统功能、性能及实现系统的各项约束条件，从技术角度研究实现系统的可能性。本项目中对各项功能的具体实现是经过了仔细分析反复提炼的过程，对一系列的技术可行性进行分析，排除了一些难以实现的方案，关于具体的技术选型和一些方案设计，可以参考最后一章中关于具体问题的分析。在具体实现上，CoderAssistant 基于 Idea 进行插件开发，参考了 Stack-Intheflow 等类似插件样式，基于 CodeBERT 和 CodeXGlue 的数据集和算法，利用 StackExchange 的 API 开发了一个基于 Stack Overflow 的自动化程序开发问题答案检索插件。

1.5.2 法律可行性

在本项目中，平台的开发主要从外部数据源和平台的技术实现进行考察分析。CoderAssistant 的外部数据源主要来源为 StackOverflow 的开放接口 StackExchangeAPI，所涉及到的数据集均已开源，平台功能的实现所涉及到的软件和组件均为 MIT 开源协议，且本项目并不是盈利为目标，不存在商业风险。在用户隐私方面支持基于本地模型部署，可以规避法律层面的意外。

2 相关工作

2.1 代码语义信息提取和预训练模型

代码语义信息提取任务类似于文本摘要任务，只不过是将文本转化为了代码，思路上是类似的。但是不同点在于代码与文本间存在不小的 gap，直接将文本摘要的方法迁移过来使用效果非常差，所以我们首先调研了代码语义信息提取也就是代码摘要 Code Summarization 的相关工作。

在目前的软件开发的生命周期（例如，实施、测试和维护）中，近 90%的工作用于维护，这项工作的大部分时间都花在理解维护相关软件源代码。因此，如何更高效地理解源代码或者快速写出更高层次的代码描述方便其他人阅读对于软件维护非常重要。尽管目前已经开发了各种技术来方便程序员在软件开发和测试期间写注释和文档，但是代码摘要/语义信息提取仍然是一项很麻烦的任务。具体来说，代码摘要/语义信息提取是一项试图压缩一段代码并直接从中生成描述的任务。目前，大多数方法都遵循 Seq2Seq 也就是编码器-解码器框架，将代码编码为隐藏的空间，然后将其解码成自然语言空间，或者采用一些抽取式或者启发式的方案，比如采用 tf-idf 算法或者 LSI 等方法抽取一些关键代码和关键字来作为提取的信息，但是他们都没有利用到目前互联网上海量的代码数据。

Code
(Java)
	private void attachPlot (SVGPlot newplot) {
	this.plot = newplot;
	if (newplot == null) {
	super.setSVGDocument(null);
	return;
	}
	newplot.synchronizeWith(synchronizer);
	super.setSVGDocument(
	newplot.getDocument());
	super.setDisableInteractions(
	newplot.getDisableInteractions());
	}

Summ.	Attach to a new plot and display.

图 2.1.1 代码语义信息提取任务/代码摘要任务

而在自然语言处理研究中，目前常用的一个研究范式就是 Large Pretrained Model + Fine-tuning 也就是预训练+微调的一个过程。即首先基于线上海量的无监督无标签数据采用 MLM 等方式构建一个预训练模型，比如 BERT，然后再利用这个预训练模型去在一个具体的下游任务上采用具体的比较小的带标签数据集进行 Fine-tuning 微调，从而得到一个比较好的效果。这种思路有点类似迁移学习，目前的工作也非常多比如 T5 Text-To-Text Transfer Transformers 以及 BART 等模型，都在自然语言处理的各项任务中取得了 SOTA 的效果。

因此，在代码摘要/语义信息提取任务中，我们也可以引入这种思想，采用一个预训练模型作为基础，再用具体的代码-文本描述的数据集进行微调，从而取得比直接训练更好的效果。

2.2 面向软件开发的信息检索类辅助插件

我们对目前 Idea 中类似插件进行了调研，主要是关于下面几个：Browser Search、CodeSearch、searchcode

2.2.1 Browser Search

图 2.2.1 Browser Search

该插件提供了在浏览器中搜索的功能，包括 stackoverflow, duckduckgo, google, baidu, bing 五个网站，还提供了快捷键操作，在选中文本之后可以快速跳转到浏览器中进行查询。

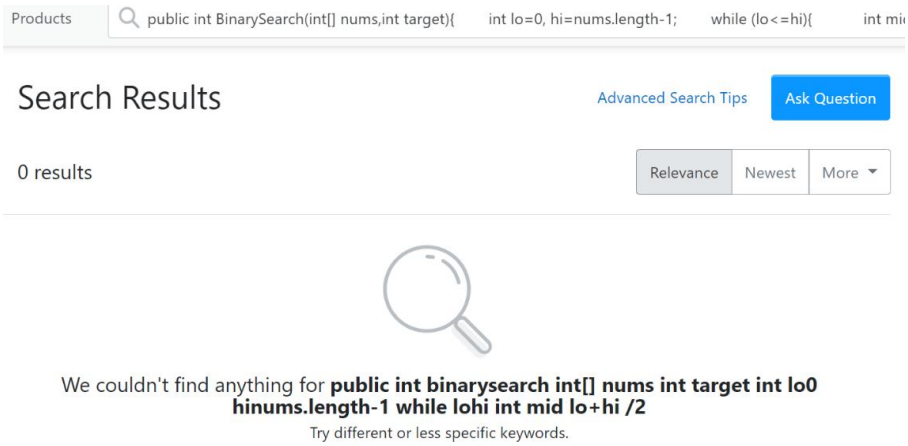


图 2.2.2 搜索 Stack Overflow

但是并没有对源代码进行处理，而是直接复制全部的文本并提交搜索。这样会导致如图搜索中有很多空行，且由于信息过长，信息密度低，出现查不到任何结果的情况。大部分的搜索引擎对于输入较短的查询或者关键词查询能得到较好的效果，而对于大段文本的支持效果不好。

图 2.2.3 百度大段检索

在百度等搜索引擎下由于搜索长度限制，会导致更多的信息丢失情况而导致搜索不到想要的结果。

另外，每次查询都会打开默认浏览器再开始搜索，这样会一定程度上打断开发的连贯性，需要在 IDE 和浏览器中反复切换导致用户体验下降。

2.2.2 CodeSearch

Code Search 同样是一个将代码直接复制到浏览器中搜索引擎进行查询的插件，会遇到和 Brower Search 中同样的问题。

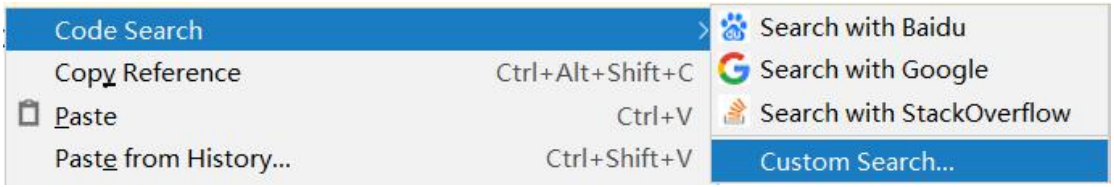


图 2.2.4 CodeSearch

不过该插件没有提供快捷键，而且需要在二级菜单中打开，进一步导致搜索效率下降。

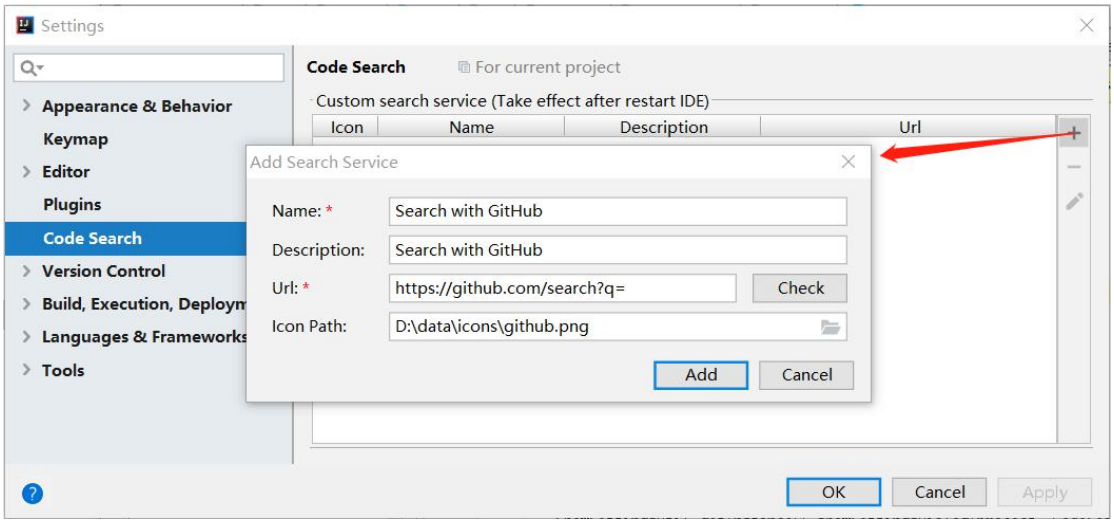


图 2.2.5 自定义搜索引擎

不过该插件一个比较大的优势是可以自定义搜索引擎，可以添加个人常用的搜索引擎，满足了用户个性化定制的需求。

2.2.3 searchcode

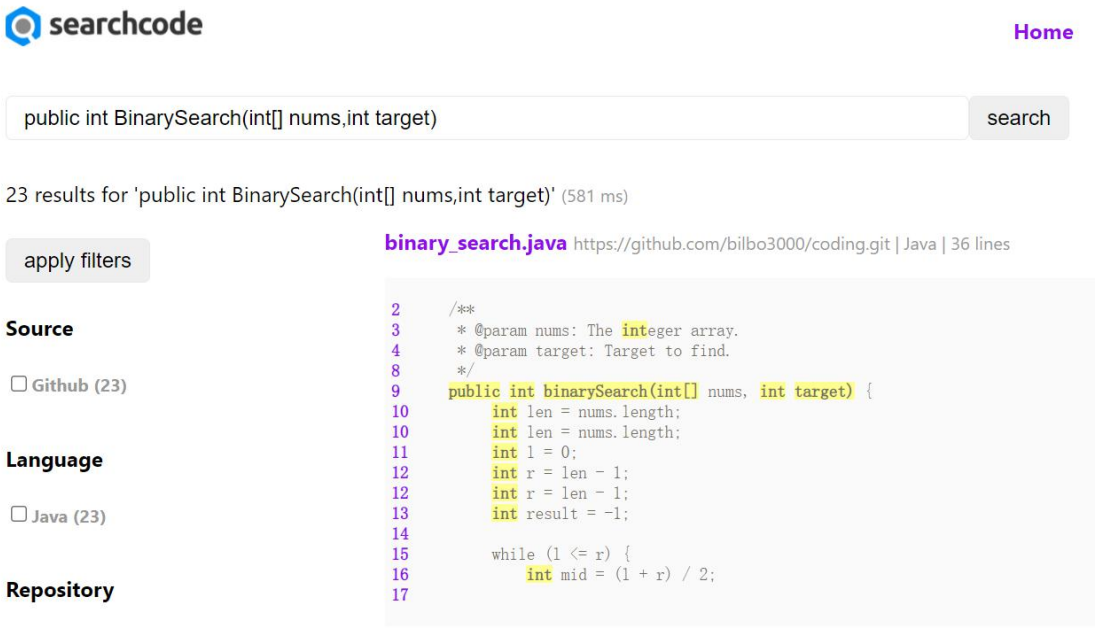
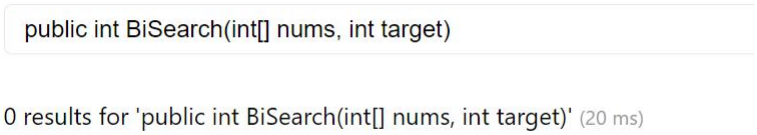


图 2.2.6 searchcode

该网站提供了对 github 网站源代码的搜索功能，可以直接根据文本找到相应的开源代码，可以达到不错的搜索效果。

存在的问题是严重依赖于变量命名，如果修改了一些变量名，可能导致搜索结果有较大差异且有可能搜索不到结果。



并且，对于 Github 代码的搜索缺少相应的解答说明，因为大部分情况下的搜索是由于遇到了代码 Bug，这样并不能解决一部分实际问题，而只是提供了一些代码切片供用户检索。

综上所述，目前用户的需求有以下几点：

(a) 可以集成在 IDE 中，不需要跳转到外部浏览器。

(b) 可以根据代码自动生成合适查询，而不是复制大段源代码。

(c) 可以自动收集错误信息并搜索，方便用户进行 Debug。

2.3 Stack Overflow问答

StackOverflow 是软件开发领域最常用的技术问答网站，其上包含了各类软件项目或技术主题的大量问答记录，这些数据组织结构清晰、包含较多有价值的结构化信息、覆盖大多数常用软件项目且易于获取。它面向的是编程人员群体，于 2008 年创建，发展至今，StackOverflow 已经成为最大的程序问答网站，拥有超过 800 万的用户以及超过 3800 万帖子，目前每天还有很多新用户的加入产生大量的博客和回答，以及浏览、点赞、收藏、评论等。

根据我们的调研，2. T. T. Nguyen, H. A. Nguyen, N. H. Pham and T. N. Nguyen, "Recurring bug fixes in object-oriented programs", *ICSE '102010*, pp. 315-324. 这篇文献表明，有很大一部分的 Bug 是经常出现的、会出现在不同项目上的，大概占总 Bug 数量的 17%-45%。一个重要原因是框架的使用越来越丰富，如 Spring、Hadoop，那么当违反框架的约束时，更加容易出现同一类 Bug。这使得在 Stack Overflow 中搜索错误信息将更加容易得到解决。另一篇文献 2015 年发表在 ASE 上的论文（Fixing Recurring Crash Bugs via Analyzing Q&A Sites）证明，根据程序的崩溃信息搜索 Stack Overflow 中的相关帖子，并使用讨论中的示例代码，可以很有效地修复这一类重复出现的 Bug。

另外，国内使用 Stack Overflow 少的一个重要原因是国内访问速度慢，或者根本无法访问，不过 Stack Overflow 开放了网站的 Api，只需要申请就可以获得大量的访问次数，通过整合 Api 就可以制作自己的 Stack Overflow 搜索引擎，访问速度比网站快且不会出现无法访问的问题。

最后，根据关于 Stack Overflow 的调研数据，我们可以发现 Stack Overflow 是大多数开发者遇到问题首先使用到的求助/寻找答案的工具。79.96%的受调查者表示他们会在遇到开发问题时使用 Stack Overflow 查找：

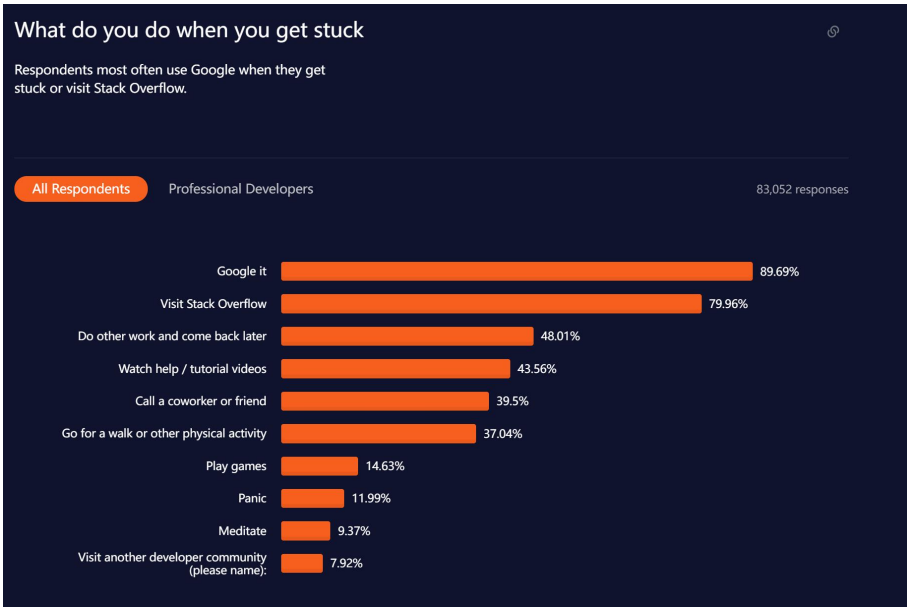


图 2.3.1 大多数人在遇到问题时用到 Stack Overflow

另外，调研数据也显示，Stack Overflow 上的用户大多数都是专业的开发者，其中全栈/后端开发占据了大多数，70%的人都是已经工作的软件工程师，学生用户比较少。由此可以反映出社区还是比较专业的，大多数人具有比较深的专业能力，社区内容质量也比较高。

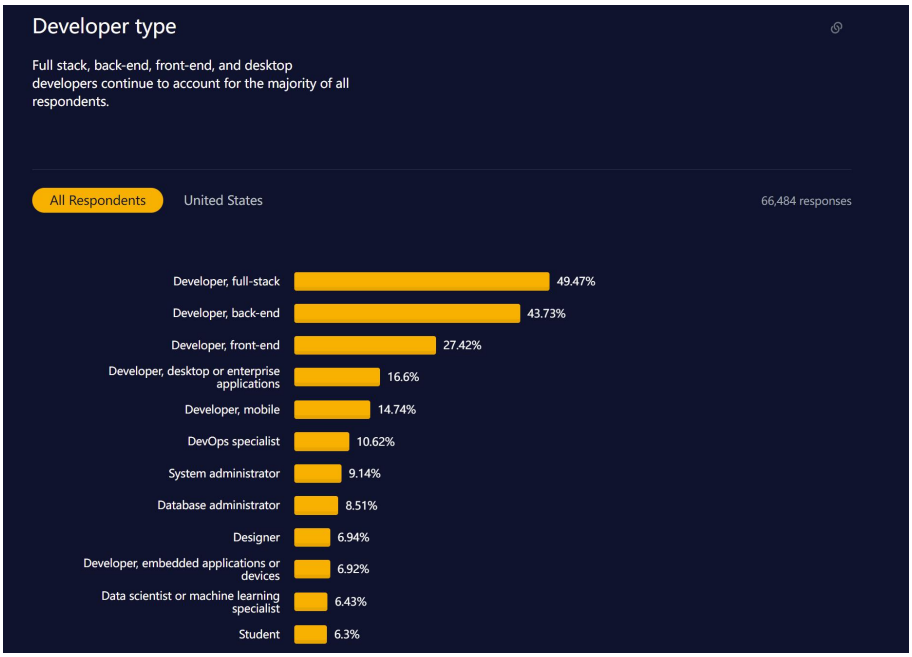


图 2.3.2 社区开发者方向调研

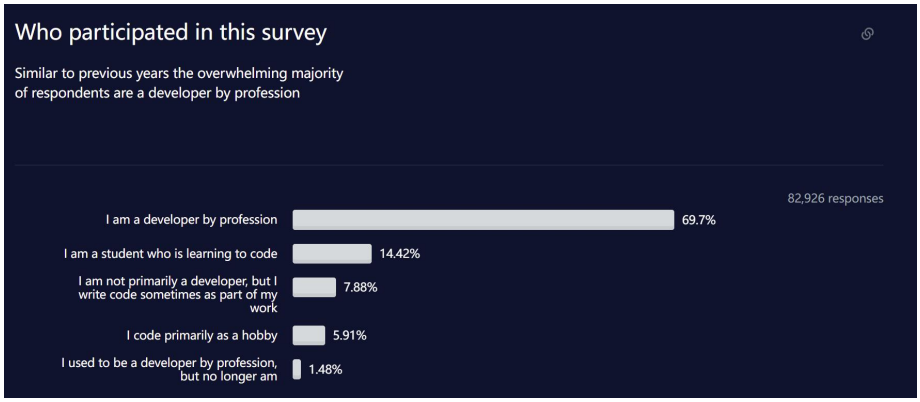


图 2.3.3 社区用户类型调研

3 系统设计与展示

3.1 功能设计与展示

基于前文对 CoderAssistant 的用例建模和需求分析以及相关工作的介绍，本节会进一步分析我们的系统应该如何为用户提供服务，创造价值，改善整体流程和效率，并以此为出发点，我们在我们开题 PPT 的基础上重新设计了产品的主要功能如下：

3.1.1 IDE 内检索

CoderAssistant 提供在 IDE 内直接检索的功能，这样可以让开发者直接在 Idea 内部进行检索。相比在 Google 等搜索引擎中检索，这样的形式可以有效地过滤掉不相干的广告；相比直接在 Stack Overflow 搜索，这样也可以避免一大堆页面切换的繁琐，同时直接的看到查询的结果，并且基于的 StackExchangeAPI 在国内要比直接访问 Stack Overflow 更加稳定。总的来说，相比于打开浏览器，再在 Google 或者 SO 上检索的方式，直接在 IDE 内进行检索会更加的方便高效，避免开发中的思路被干扰。

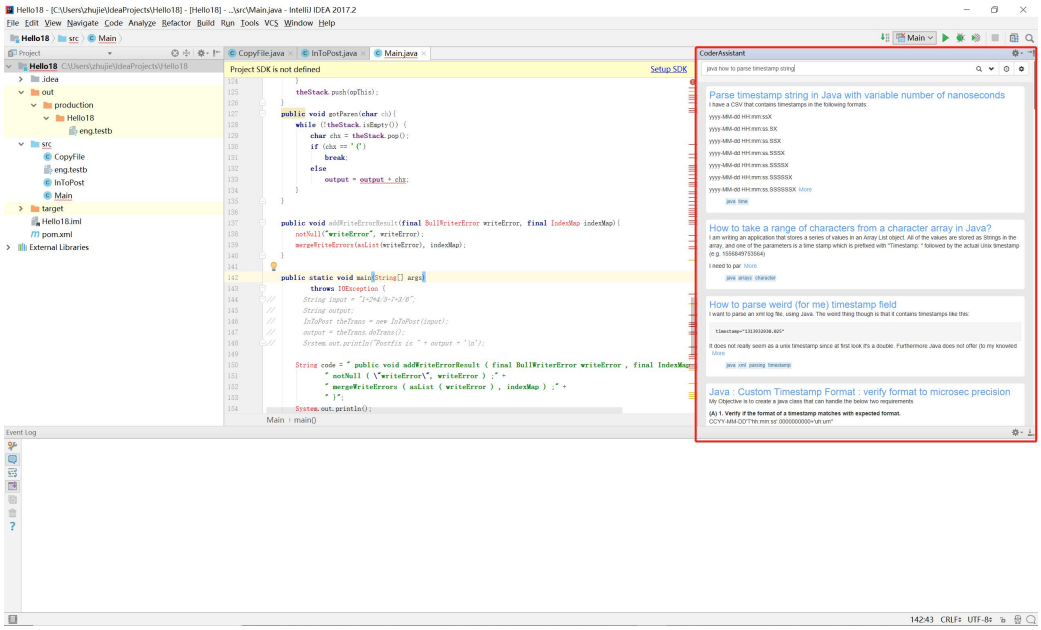


图 3.1.1 IDE 内输入查询检索

3.1.2 根据报错信息自动检索

CoderAssistant 还提供根据错误信息自动检索的功能。比如下图所示，在程序运行出现 Error 时，CoderAssistant 会自动截取第一行的错误信息作为查询 query 到 StackExchange 的 API 去检索，并将检索到的答案自动返回并展示。传统 Debug 或者检索答案的方法往往需要先复制错误信息，然后去粘贴到网站上，这样比较麻烦而且搜索多了之后打开的页面可能会很混乱。CoderAssistant 会自动根据报错信息去搜索，开发者只需要直接查看检索得到的问答结果即可。

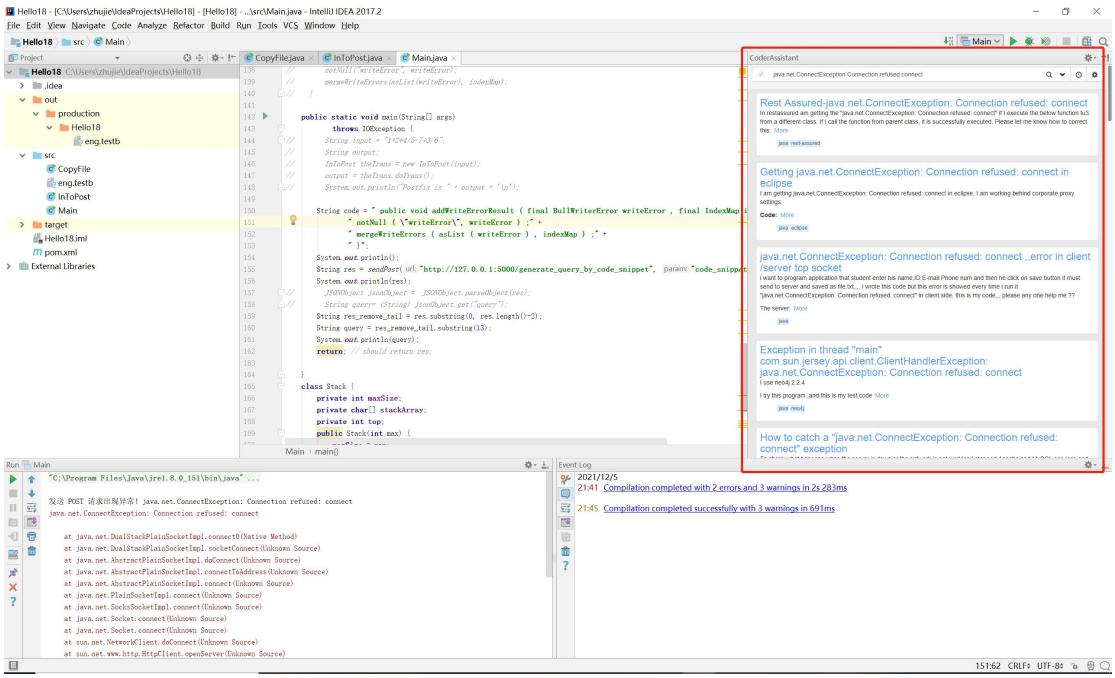


图 3.1.2 根据报错信息自动检索

3.1.3 提取代码语义信息进行检索

CoderAssistant 还可以代码语义信息进行自动检索。比如下图所示，开发者可以选取一段代码，然后 CoderAssistant 会自动对代码进行分词、解析，并将代码段字符串发送到模型中进行处理，获得代码语义信息的自然语言描述文本。然后将这一段文本作为查询 query 到 StackExchange 的 API 去检索，并将检索到的答案自动返回并展示。

传统 Debug 或者检索答案的方法往往对于开发者自身的技术水平有较高的要求，需要开发者能看懂或者搞明白代码到底在做什么事情，但是这一点对于一些新手程序员或者不太熟悉某种语言或框架的程序员来说比较困难。而我们的工具 CoderAssistant 可以帮助开发者直接生成针对某一段代码的 query 查询，从而帮助他们更好的找到高质量的问答结果，同时节省理解代码和写查询的时间。

具体是如何根据代码段生成查询语句的过程，会在模型算法实现部分进行具体的介绍。

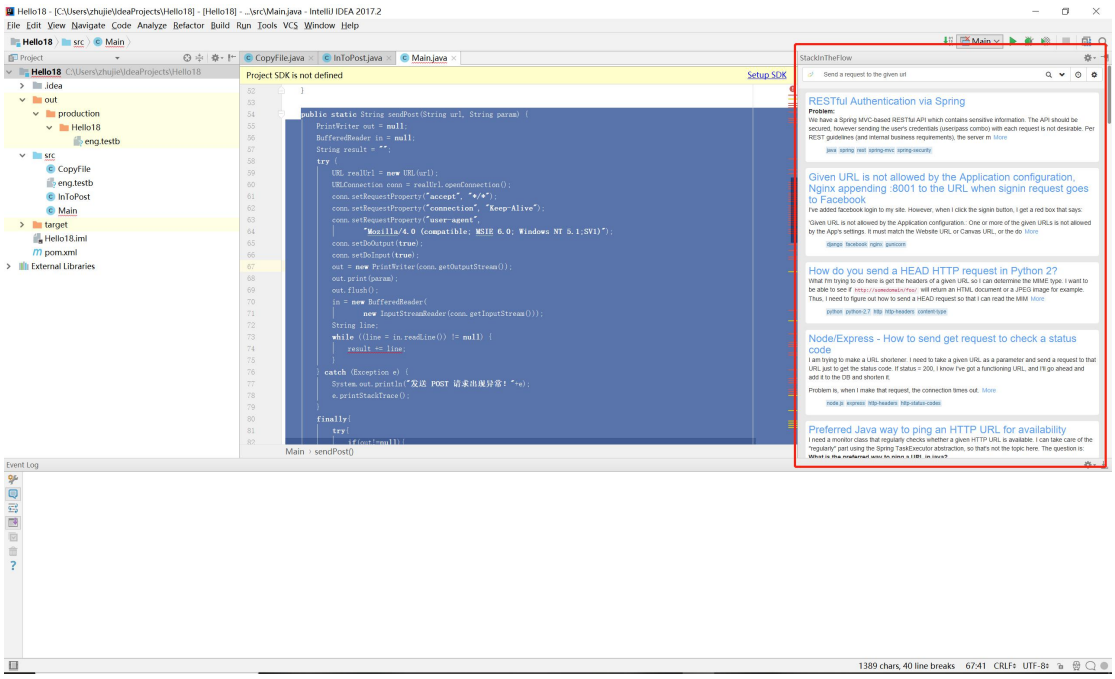


图 3.1.3 提取代码语义信息自动检索

3.2 系统分层架构设计

简要介绍系统的主要技术架构，包括插件架构和算法部分的架构等

3.2.1 插件架构概览

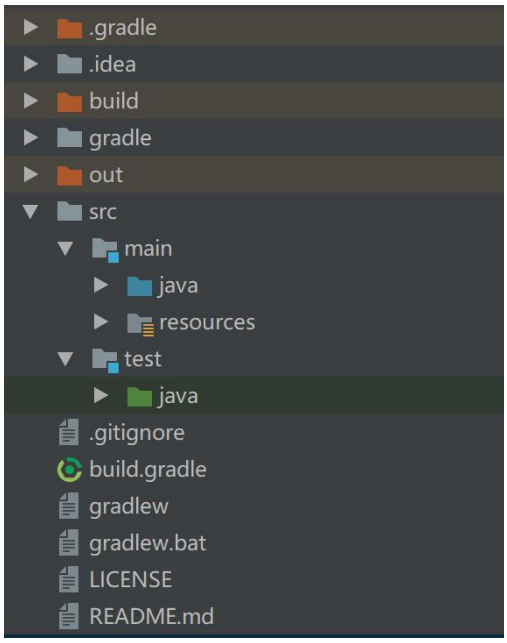


图 3.2.1 插件文件架构

Project Structure:

```
├──── .gradle/                Gradle 版本库
├──── .idea/                  idea 项目设置
├──── gradle
|     └──── wrapper/          Gradle Wrapper 目录
├──── build/                  build 输出目录
├──── src                     插件源文件
|     └──── main
|          ├──── java/        Java 项目源文件
|          └──── resources/   依赖资源，如前端页面，图标等等
|     └──── test              测试目录
|          └──── java/        java 测试目录
├──── .gitignore              Git ignoring 规则
├──── build.gradle            Gradle build 脚本
├──── gradlew                 Gradle Wrapper 脚本
├──── gradlew.bat             Windows Gradle Wrapper 脚本
├──── LICENSE                 License
└──── README.md               README
```

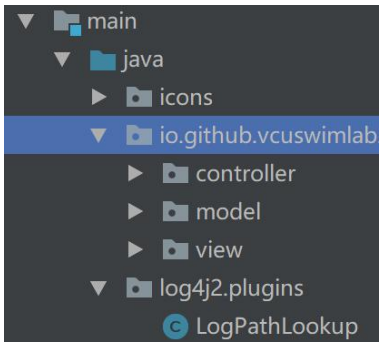


图 3.2.2 main 文件架构

icons 是项目图标存储位置

controller 项目业务逻辑层

model 对业务逻辑中需要使用的数据结构的定义与封装

view 视图层，定义了前端搜索页面

log 引入的 log 组件，记录日志

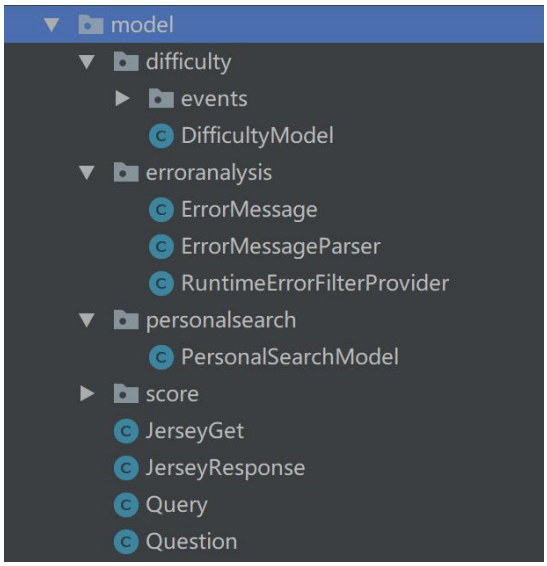


图 3.2.3 model 层架构

JerseyGet 使用 Jersery 框架完成 Web 请求，封装了 StackExchange Api 来发送请求

JerseyResponse 封装了 Jersey 服务响应的一些方法，供 JerseyGet 调用

Query　定义了请求的格式，方便 JerseyGet 调用

Question　定义了问题的格式，方便 JerseyResponse 调用。

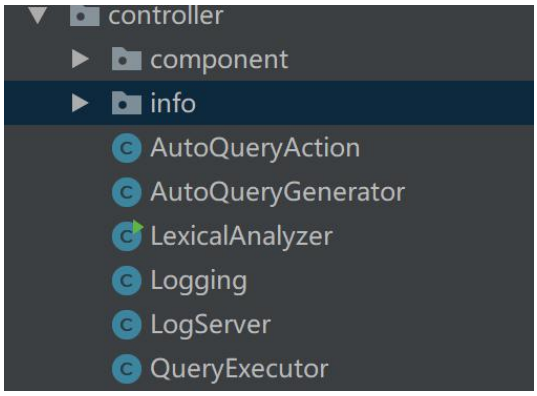


图 3.2.4 逻辑层架构

Component　定义了一些组件，方便访问

AutoQueryAction　定义了自动查询的业务逻辑，包括触发器，获得查询，显示 GUI 等等

AutoQueryGenerator　根据编辑器选中文本生成查询，应用了我们的模型

LexicalAnalyzer　简单的词法分析，输入字符串，输出 token 流方便导入模型

Logging/LogServer 生成日志记录相关代码

QueryExecutor 实际发送请求查询的代码,调用了 JerseyResponse 和 JerseyGet

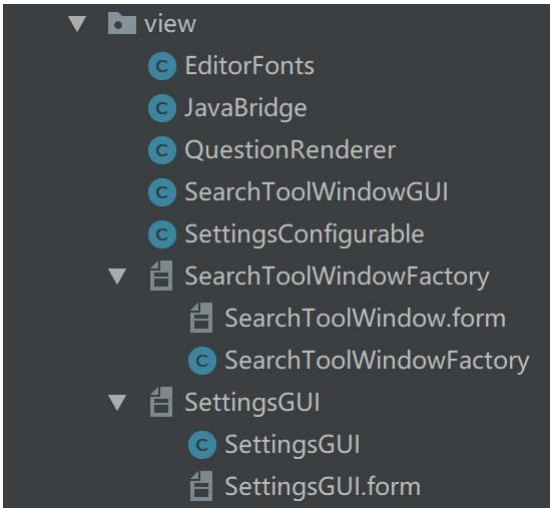


图 3.2.5 视图层架构

EditorFonts　定义了编辑区域的字体、背景等

JavaBridge 桥接类，方便 Javascript 调用 Java 代码

QuestionRenderer　问题渲染方法，如字体、背景、显示结构、分页等等视图

SearchToolWindowGUI　搜索视图的 GUI 界面编写，定义了整体结构和调用关系

SettingsConfigurable 定义了设置功能的 UI

SearchToolWindowFactory　创建搜索视图 GUI 的工厂方法

SettingsGUI　GUI 的实际创建设置

3.2.2 算法流程概览

算法架构流程概览如下

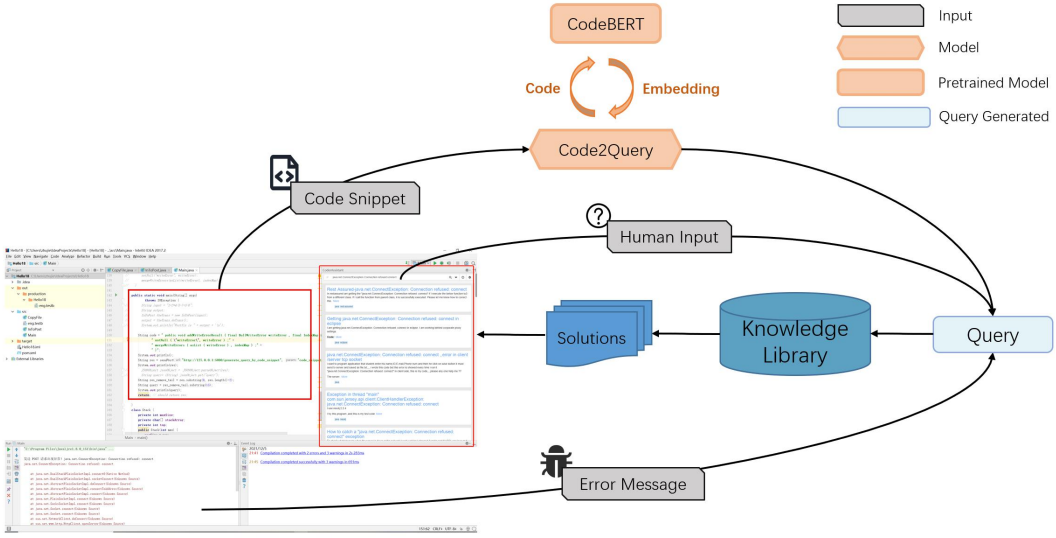


图 3.2.1 算法流程概览

算法基本流程即通过手动输入，错误信息，代码语义信息抽取等模块获取输入，然后请求发给 StackExchangeAPI，获得查询结果后展示到插件中。

代码语义信息抽取模型 Code2Query 基于 CodeBERT 预训练模型，负责将代码段转化为对应的文本描述 qeury，具体实现细节在第四章中介绍。

4 技术实现

4.1 插件系统

4.1.1 组件库和框架简介

Gradle 简介

Gradle 是一个构建自动化工具，以其构建软件的灵活性而闻名。生成自动化工具用于自动创建应用程序。生成过程包括编译、链接和打包代码。在构建自动化工具的帮助下，该过程变得更加一致。

它因其能够以 Java，Scala，Android，C / C++和 Groovy 等语言构建自动化而广受欢迎。该工具支持基于 groovy 的域特定语言（基于 XML）。Gradle 在多个平台上提供构建、测试和部署软件的功能。

该工具在构建任何软件和大型项目时很受欢迎。Gradle 包括了 Ant 和 Maven 的优点，并遏制了两者的缺点。

在 Idea 插件开发过程中使用 Gradle 作为包管理以及 Build 工具，如需要获取’jersey-client’等等外部包，我们可以方便地进行版本控制和引用。

插件算法流程

IDE 内检索：

View 层生成的 GUI 包含检索功能，用户可以自己键入信息再点击搜索按钮，即可进行查询。执行流程是：js 监听到搜索按键按下，通过 JavaBridge 返回到 Java 中执行，将搜索字符提交到 QueryExcutor 中执行后返回 jerseyResponse，获取响应的问题列表，返回视图层渲染结果。

错误信息自动检索：

通过 idea 开放的 console api,可以监听到程序发生 bug,生成 Error Message 实例，提取到有效错误信息后执行 QueryExcutor，返回视图层渲染结果。

根据代码语义信息检索：

可以根据选择的文本右键菜单中执行 Auto Query，如果不选则默认选择全部文本，通过 AutoQueryGenerator 生成查询，具体是输入 LexicalAnalyzer 生成

token 流，再发送到搭建了根据代码查询的 Flask Api 中，获取返回结果后返回查询字符串，之后执行 QueryExcutor，返回视图层渲染结果。

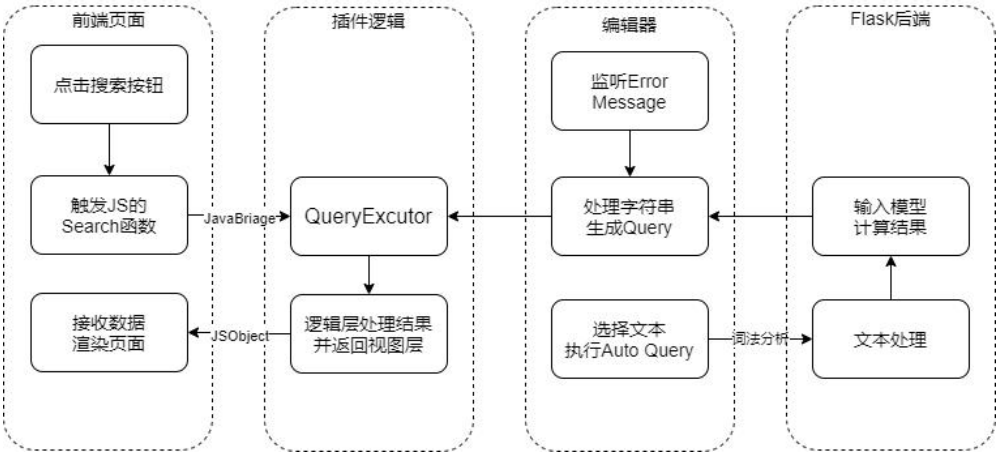


图 4.1.1 插件逻辑架构图

4.1.2 前端系统功能实现

前端系统主要由 javafx 提供的 webview 功能实现。WebView：使用 WebKit HTML 技术的 Web 组件，可以在 JavaFX 应用程序中嵌入 Web 页面。在 WebView 中运行的 JavaScript 可以调用 Java API，Java API 可以调用在 WebView 中运行的 JavaScript。JavaFX 中添加了对其他 HTML5 功能的支持，包括 Web 套接字，Web Worker 和 Web 字体以及打印功能。

WebView 可以说以一种嵌入式浏览器，它通过 JavaFX 应用程序中的 API 提供 Web 查看器和完整浏览功能，可以使 Java 应用程序中实现以下功能：

- 渲染 HTML，从本地或远程 URL 呈现 HTML 内容
- 支持历史记录并提供后退和前进导航
- 重新加载内容
- 将效果应用于 Web 组件
- 执行 JavaScript 命令
- 执行从 JavaScript 到 JavaFX 的上行调用
- 处理事件

WebView 封装了 WebEngine 对象，将 HTML 内容合并到应用程序的场景中，并提供应用效果和转换的属性和方法。

WebEngine 有如下特点：

1.是一个能够一次管理一个网页的非可视对象

2.通过其 API 提供基本网页功能。

3.它支持用户交互，例如导航链接和提交 HTML 表单，但它不直接与用户交互。

4.它加载网页，创建文档模型，根据需要应用样式，并在页面上运行 JavaScript。

5.它提供对当前页面的文档模型的访问，并允许 Java 应用程序和页面的 JavaScript 代码之间的双向通信。

6.它包装了一个 WebPage 对象，该对象提供与本机 Webkit 核心的交互。

利用 WebView 强大的 web 浏览器构建功能就可以实现前端 GUI 界面的编写。

首先编写静态的 HTML 页面和对应的 CSS 样式：

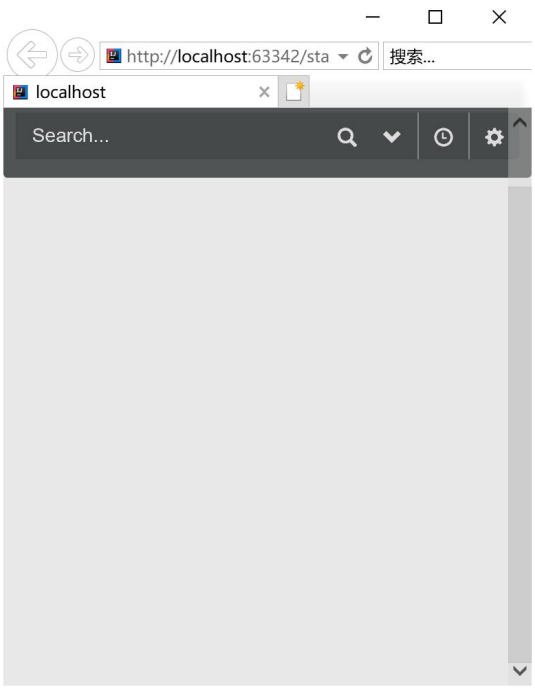


图 4.1.2 静态 HTML 页面

主要包括了搜索框、搜索按钮、排序按钮、历史记录按钮和设置按钮以及对应的搜索记录。

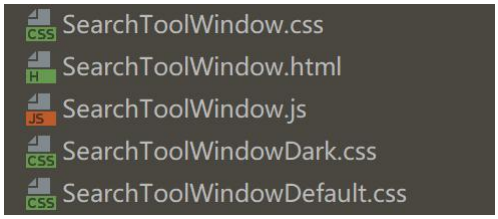


图 4.1.3 网页结构图

再编写对应的 JS 脚本与 Java 进行通信，将所有按钮映射为功能，接受 Java 获得返回值后进行 HTML 渲染。

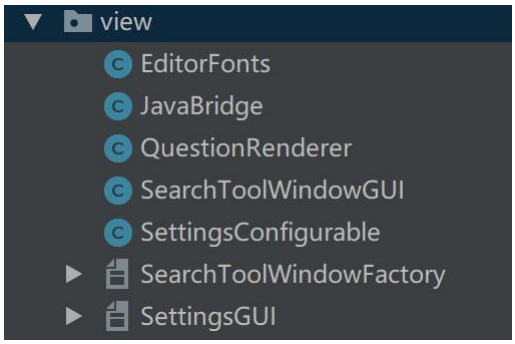


图 4.1.4 view 层结构图

View 层结构如图所示，GUI 的主要原理是通过 Javafx 提供的 Webview 组件，渲染 HTML 文件，并嵌入 idea 的视图中，来实现侧边栏功能。

```
private JSObject window; //Object to interact with JS.
private JavaBridge bridge; //Object to interact with JS.
```

图 4.1.5 Java 与 js 通信

在 Java 中，需要通过 JSObject 对象调用 JavaScript 函数来与前端页面通信，同时需要创建 JavaBridge 对象，实现在 JavaScript 中调用 Java 函数的功能，来实现插件和前端页面的双向通信，实现一个搜索功能的主要流程如下：

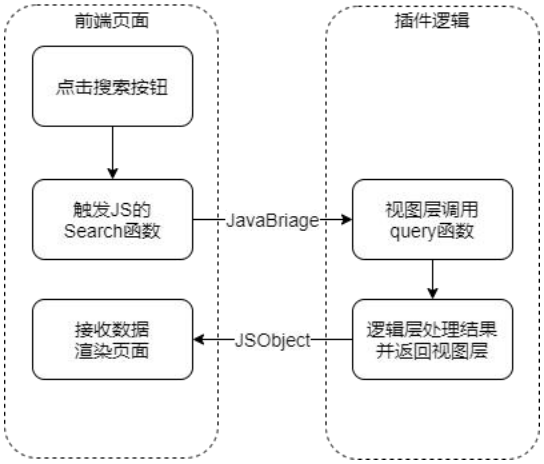


图 4.1.6 搜索功能 GUI 调用逻辑

同样的，实现对错误信息的 AutoQuery 功能只需，接受逻辑层传来的 ErrorMessage 再次通过 JSObject 对象传递给 Javascript 进行 HTML 页面渲染。

4.2 数据集与模型算法实现

4.2.1 数据集介绍

使用到的数据集主要来自 CodeXGlue 的数据集，一共有 500754 条 Java 代码和描述的数据对，这些数据的来源是 Github 上的开源代码。该数据集的原作者将他们收集起来，并且按照了一定规则进行筛选，获得了一些比较高质量的代码和描述的集合。具体的筛选规则如下：

1） 选择的项目至少被其他项目引用或使用了一次

2） 函数代码段的描述只截取第一行

3） 短于 3 个 token 单词的描述都过滤掉

4） 函数代码段短于三行的都过滤掉

5） 所有函数名包含 test 的都过滤掉

通过这样的一些规则设计，数据集的质量还是比较高的，因为时间还有我们机器性能等等原因，我们就直接使用了他们的数据集，具体数据集的样例如下：

{
"repo": "google/guava",
"path":"android/guava/src/com/google/common/io/ReaderInputStream.java",
"func_name": "ReaderInputStream.grow",

"original_string": "private static CharBuffer grow(CharBuffer buf) {\n char[] copy = Arrays.copyOf(buf.array(), buf.capacity() * 2);\n CharBuffer bigger = CharBuffer.wrap(copy);\n bigger.position(buf.position());\n bigger.limit(buf.limit());\n return bigger;\n }",
"language": "java",
"code": "private static CharBuffer grow(CharBuffer buf) {\n char[] copy = Arrays.copyOf(buf.array(), buf.capacity() * 2);\n CharBuffer bigger = CharBuffer.wrap(copy);\n bigger.position(buf.position());\n bigger.limit(buf.limit());\n return bigger;\n }",
"code_tokens": ["private", "static", "CharBuffer", "grow", "(", "CharBuffer", "buf", ")", "{", "char", "[", "]", "copy", "=", "Arrays", ".", "copyOf", "(", "buf", ".", "array", "(", ")", ",", "buf", ".", "capacity", "(", ")", "*", "2", ")", ";", "CharBuffer", "bigger", "=", "CharBuffer", ".", "wrap", "(", "copy", ")", ";", "bigger", ".", "position", "(", "buf", ".", "position", "(", ")", ")", ";", "bigger", ".", "limit", "(", "buf", ".", "limit", "(", ")", ")", ";", "return", "bigger", ";", "}"],
"docstring": "Returns a new CharBuffer identical to buf, except twice the capacity.",
"docstring_tokens": ["Returns", "a", "new", "CharBuffer", "identical", "to", "buf", "except", "twice", "the", "capacity", "."],
"sha": "7155d12b70a2406fa84d94d4b8b3bc108e89abfd",
"url":"https://github.com/google/guava/blob/7155d12b70a2406fa84d94d4b8b3bc108e89abfd/android/guava/src/com/google/common/io/ReaderInputStream.java#L189-L195",
"partition": "valid"
}

图 4.2.1 数据集的一个样例

如上图所示，数据集就是由一个一个的函数代码段 code，和对应的文本描述 docstring 组成，既可以进行正向的从代码到文本描述的任务训练，也可以进行反向的从文本到代码生成，即 text2code 和 code2text 任务。考虑到 CodeX 基于 GPT-3 模型在文本到代码的表现更好，我们因为种种原因（未开源、机器性能）又没办法去运行 CodeX，所以我们在我们的项目中主要是针对 code2text，也就是解决从代码段到文本查询以及自动化解决方案检索的功能。

具体数据集链接可参考 CodeXGLUE 提供的 Github 开源项目：https://github.com/microsoft/CodeXGLUE/tree/main/Code-Text/code-to-text

4.2.2 预训练模型 CodeBERT

模型我们采取的 CodeBERT 作为我们算法的预训练模型。CodeBERT 遵循 BERT 和 RoBERTa 的做法，并使用了多层双向 Transformer 作为 CodeBERT 的模型架构。具体而言，CodeBERT 的模型架构与 RoBERTa-base 基本一致，包括 12 层，每一层有 12 个自注意力头，每个自注意力头的大小为 64。隐藏维度为 768，前馈层的内部隐藏层大小为 3072。模型参数总量为 1.25 亿。

类似 BERT 的思路，CodeBERT 在预训练过程中将输入设置为两个片段和一个特殊分隔符的组合即 [CLS], w1, w2, ..wn, [SEP], c1, c2, ..., cm, [EOS]。其中一个片段是自然语言文本，另一个则是用某种编程语言写成的代码。而 CodeBERT 的输出则包括每个 token 的语境向量表示（适用于自然语言和代码）和[CLS] 的表示(聚合序列表示)。目标函数包括遮蔽语言建模(masked language modeling，MLM）和替换 token 检测（RTD），MLM 即预测被遮蔽的原始 toke，这种思路在自然语言处理的很多预训练模型比如 BERT 中都得到了广泛应用并取得了很好的效果，而替换 token 检测则利用单模态代码学得更好的生成器，从而输出更好的替换 token，综合采用这两个目标函数可以很好地在双模态数据（代码-文本）中实现对语言的建模。

我们在模型算法上选择 CodeBERT 作为我们算法的预训练模型的主要原因一方面在于它是目前该领域最新的、性能也是最好的一个开源“代码-文本预训练模型”，另一方面它也将源代码以及如何进行下游任务 fine-tuning 的文档完全开放。我们可以相对轻松的在它的基础上进行调整和对比实现，经过验证，相比 CoSQA 和 Code2Que 等同样发表在自然语言处理和软工领域的顶会上的文章，CodeBERT 在性能和易用性上都更好的满足了我们的需求。

4.2.3 基于 CodeBERT 进行下游代码描述任务微调

目前，自然语言处理领域内的一个基本研究范式就是对预训练语言模型(PLM）进行微调（fine-tuning）。这种方法已经取代了从预训练 embedding 做特征提取的方法，在几乎所有的领域任务上取得了更好的效果，比如 BERT，T5 等大模型取得的一系列突破。而这种方式的一个基本流程就是首先在大规模无监督数据上使用建模语言特征的 loss 对一个模型做预训练，然后在下游任务的有标签数

据上使用标准的 cross-entropy loss 对预训练模型做 fine-tuning，从而实现更好的效果同时又不需要自己去收集一个很大的语料库。

而我们的任务其实就跟这个方式比较接近，我们需要根据代码来自动生成对应的查询信息，而这其实是一个类似摘要生成的过程，即提取出代码中的关键语义信息来查询。所以整体流程大致是先学习代码在连续语义空间中的表示，再利用这种语义表示来生成自然语言描述文本。因此，我们选择了 CodeBERT 作为我们的预训练模型，基于其双模态的大规模语料数据学习到比较好的语义表示，然后再利用超过 50 万条的 java 代码文本标签数据对进行微调（fine-tuning），最后训练生成一个从代码到文本描述的模型。

Fine-tuning 微调的过程中采用了经典的 Seq2Seq 模型，由一个编码器和一个解码器组成，训练中采用交叉熵作为 loss 进行训练。

```
def forward(self, source_ids=None,source_mask=None,target_ids=None,target_mask=None,args=None):
    outputs = self.encoder(source_ids, attention_mask=source_mask)
    encoder_output = outputs[0].permute([1,0,2]).contiguous()
    if target_ids is not None:
        attn_mask=-1e4 *(1-self.bias[:target_ids.shape[1],:target_ids.shape[1]])
        tgt_embeddings = self.encoder.embeddings(target_ids).permute([1,0,2]).contiguous()
        out = self.decoder(tgt_embeddings,encoder_output,tgt_mask=attn_mask,memory_key_padding_mask=(1-source_mask).bool())
        hidden_states = torch.tanh(self.dense(out)).permute([1,0,2]).contiguous()
        lm_logits = self.lm_head(hidden_states)
        # Shift so that tokens < n predict n
        active_loss = target_mask[..., 1:].ne(0).view(-1) == 1
        shift_logits = lm_logits[..., :-1, :].contiguous()
        shift_labels = target_ids[..., 1:].contiguous()
        # Flatten the tokens
        loss_fct = nn.CrossEntropyLoss(ignore_index=-1)
        loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1))[active_loss],
                        shift_labels.view(-1)[active_loss])

        outputs = loss,loss*active_loss.sum(),active_loss.sum()
        return outputs
```

图 4.2.2 损失函数计算

```
for epoch in range(args.num_train_epochs):
    bar = tqdm(train_dataloader,total=len(train_dataloader))
    for batch in bar:
        batch = tuple(t.to(device) for t in batch)
        source_ids,source_mask,target_ids,target_mask = batch
        loss,_,_ = model(source_ids=source_ids,source_mask=source_mask,target_ids=target_ids,target_mask=target_mask)

        if args.n_gpu > 1:
            loss = loss.mean()
        if args.gradient_accumulation_steps > 1:
            loss = loss / args.gradient_accumulation_steps
        tr_loss += loss.item()
        train_loss=round(tr_loss*args.gradient_accumulation_steps/(nb_tr_steps+1),4)
        bar.set_description("epoch {} loss {}".format(epoch,train_loss))
        nb_tr_examples += source_ids.size(0)
        nb_tr_steps += 1
        loss.backward()

        if (nb_tr_steps + 1) % args.gradient_accumulation_steps == 0:
            optimizer.step()
            optimizer.zero_grad()
            scheduler.step()
            global_step += 1
```

图 4.2.3 训练过程

4.2.4 模型预测和部署

在模型训练结束后，我们得到了一个可以根据代码生成描述文本的模型文件 java_model.bin，接下来就可以 load 这个模型并进行预测，有关模型的评估效果我们会在后面进行介绍，这里是一个预测过程的代码：

```
all_source_ids = torch.tensor([source_ids], dtype=torch.long)
all_source_mask = torch.tensor([source_mask], dtype=torch.long)
eval_data = TensorDataset(all_source_ids,all_source_mask)

eval_sampler = SequentialSampler(eval_data)
eval_dataloader = DataLoader(eval_data, sampler=eval_sampler, batch_size=eval_batch_size)

# 模型生成query
p=[]
for batch in tqdm(eval_dataloader,total=len(eval_dataloader)):
    batch = tuple(t.to(device) for t in batch)
    source_ids,source_mask= batch
    with torch.no_grad():
        preds = model(source_ids=source_ids,source_mask=source_mask)
        for pred in preds:
            t=pred[0].cpu().numpy()
            t=list(t)
            if 0 in t:
                t=t[:t.index(0)]
            text = tokenizer.decode(t,clean_up_tokenization_spaces=False)
            p.append(text)
```

图 4.2.4 预测过程

部署的方式我们采用的 Flask 作为后端框架，以接口的形式获取插件代码请求中发过来的代码文本，并运行模型预测，将预测的结果即描述文本返回给插件。插件获取到描述文本后自动发请求到 StackExchange 去检索答案，并展示给用户。这个过程中，模型在后端中一直维持，而不是每次都需要重新 load，整体时间开销大约 1s 也是在可以接受的范围以内（具体数据在性能测试部分介绍）。

图 4.2.5 Flask 后端

目前因为时间有限，后端直接采用了 Flask 作为框架，这样的一个缺点在于可移植性不强。因为如果其他人需要用这个模型的时候，又没有 GPU 的时候如果采用 CPU 运行模型预测的话时间开销会比较大（增加了 5 倍以上，超过了 5 秒），将模型部署到服务器上也面临同样的问题（GPU 服务器成本高，CPU 服务器性能弱）。因此，一种可行的方案是提供一个 cuda 镜像，用户可以在本地运行 cuda 容器来进行预测，这种方式也是目前基于深度学习的插件常采用的方式（比如 DocString Generator 等）。另一种思路是设计一些基本的启发式算法或者不涉及模型预测的方式，来更高效快速地根据代码生成 Query 描述。

5 系统集成、测试和部署

5.1 持续集成与持续部署

持续集成 Continuous Integration 强调开发人员提交了新代码之后，立刻进行构建、（单元）测试。并根据测试结果，可以确定新代码和原有代码能否正确地集成在一起。确保符合预期以后，再将新代码"集成"到主干。而持续部署 Continuous Deployment 则是在持续交付的基础上，把部署到生产环境的过程自动化。持续集成和持续部署的好处在于，每次代码的小幅变更，都能看到运行结果，从而不断累积小的变更，而不是在开发周期结束时，一下子合并一大块代码。

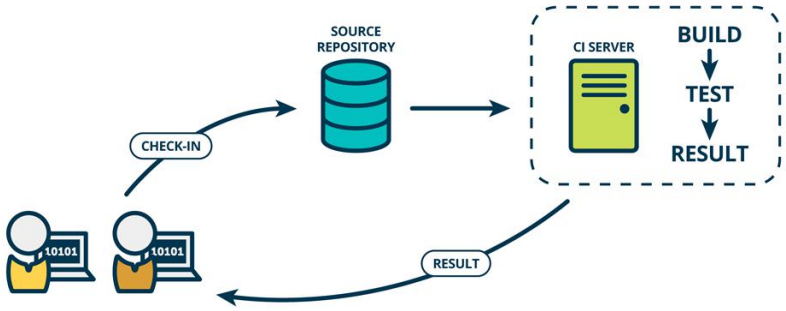


图 5.1.1 持续集成

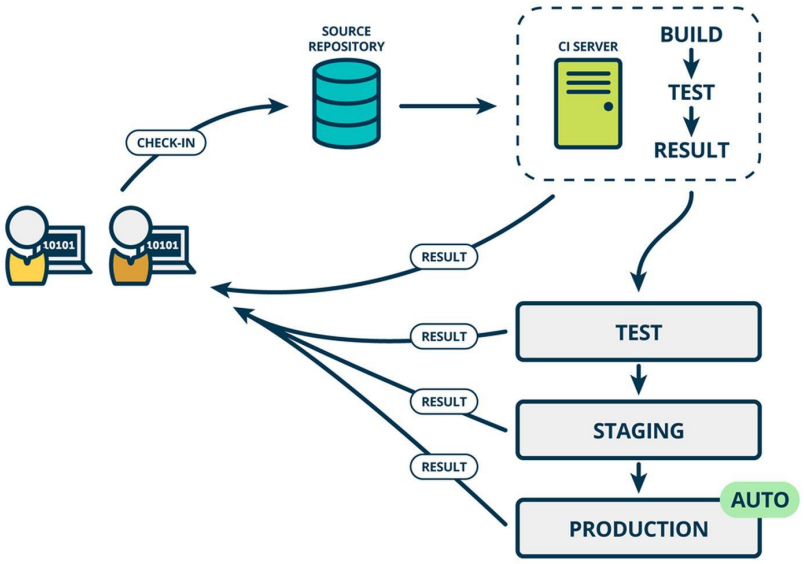


图 5.1.2 持续部署

在我们的项目中，我们采用了 github-actions 作为我们的 CI/CD 工具，每次在 git commit/push 提交后，会自动执行测试代码，检验是否可以将代码集成到项目中去，从而实现项目的持续集成。

5.2 自动化测试

自动化测试是让程序代替人去验证程序功能的过程，将大量的重复性劳动力交付给机器运行。相对于传统测试，自动化测试优势包括：减少失误率、节省时间和执行成本、提高效率等。

下面我们会基于单元测试，性能测试，自动化测试过程等等来进行介绍：

5.2.1 基于 JUnit 的单元测试

本项目使用的测试框架为 JUnit 单元测试框架。JUnit 在测试驱动的开发方面有很重要的发展，是起源于 JUnit 的一个统称为 xUnit 的单元测试框架之一。我们从 Error、Logging、Score 三个角度给代码主要部分编写了单元测试。51 个测试全部通过即完成。如下图所示：

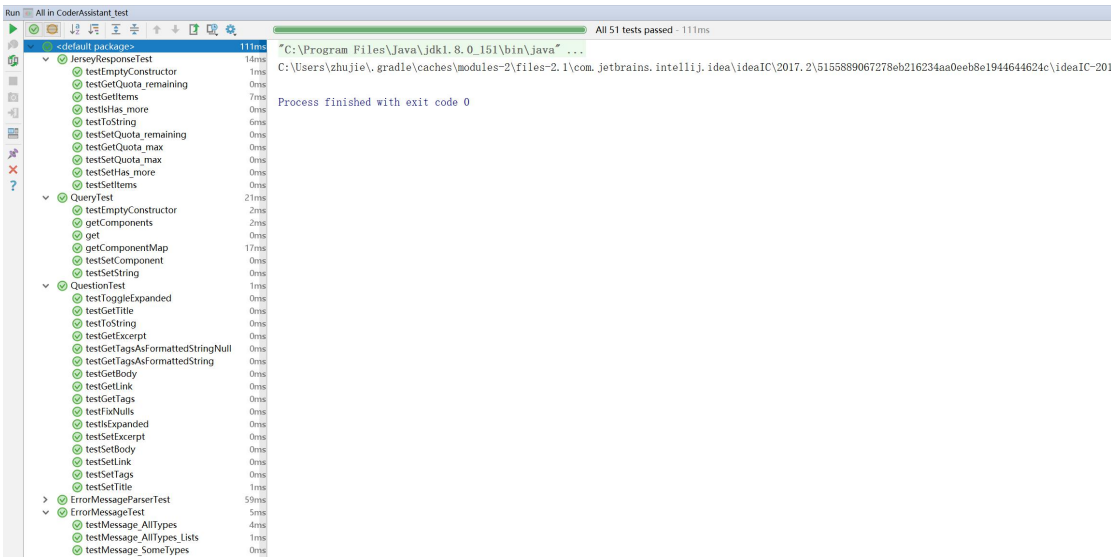


图 5.2.2 测试完成截图

Junit 中测试失败时：

- Failure：一般是由于测试结果和预期结果不一致引发的，表示测试的这

个点发现了问题

- error：是由代码异常引起的，它可以产生于测试代码本身的错误，也可以是被测试代码中隐藏的 bug

下面是一些常用注解：

- @Test:将一个普通方法修饰成一个测试方法 @Test(excepted=xx.class): xx.class 表示异常类，表示测试的方法抛出此异常时，认为是正常的测试通过的 @Test(timeout = 毫秒数) :测试方法执行时间是否符合预期
- @BeforeClass： 会在所有的方法执行前被执行，static 方法 （全局只会执行一次，而且是第一个运行）
- @AfterClass：会在所有的方法执行之后进行执行，static 方法 （全局只会执行一次，而且是最后一个运行）
- @Before：会在每一个测试方法被运行前执行一次
- @After：会在每一个测试方法运行后被执行一次
- @Ignore：所修饰的测试方法会被测试运行器忽略
- @RunWith：可以更改测试运行器 org.junit.runner.Runner
- Parameters：参数化注解

Logtest 中主要作用为两部分：Jasypt 加密器，用于生成加密信息和检查文件是否存在且包含上下文。如下图所示:

```
try{
Scanner scanner = new Scanner(file);
List<String> list=new ArrayList<>();
while(scanner.hasNextLine()){
    list.add(scanner.nextLine());
}

if(list.isEmpty()){

    assertTrue(false);
}else{
    assertTrue(true);
}
```

图 5.2.2 查询日志

总的来说，插件代码在除开 view 等 GUI 以及外部代码等部分外，核心代码的测试覆盖率达到了 100%。下图即测试覆盖率统计：

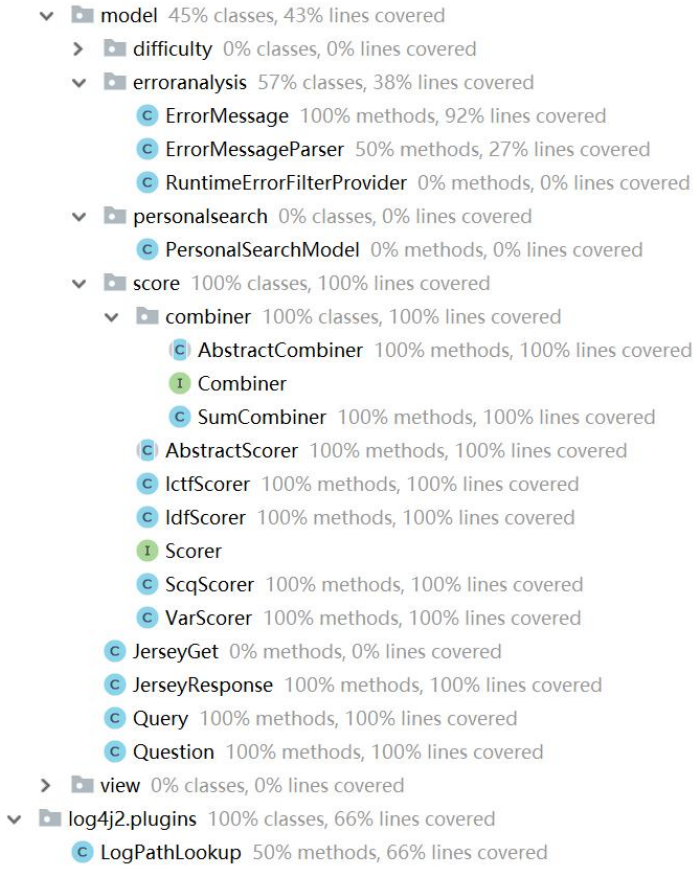


图 5.2.3 测试覆盖率

5.2.2 测试模块介绍

测试的主要模块和内容如表 5.1 所示

测试名称	测试内容
IctfScorerTest	Ictf 得分测试
IdfScorerTest	Idf 得分测试
ScqScorerTest	Scq 得分测试
VarScorerTest	Var 得分测试
SumCombinerTest	组合得分测试
JerseyResponseTest	Jersey 响应测试

QueryTest	Query 生成测试
QuestionTest	问题测试

表 5.1 测试内容

5.2.3 性能测试

本插件的主要性能瓶颈在于代码语义提取模型和请求 Stack Exchange API 返回结果的过程。在进行性能测试的时候综合考虑了这几个问题，并基于我们的三个主要核心功能进行了性能上的测试对比。这里的对比指标是功能任务用时：

功能任务	Baidu	Stack Overflow	CoderAssistant
手动输入检索	16.03	14.87	**10.52**
代码语义信息搜索	22.45	19.77	**3.67**
错误信息检索	14.79	13.22	**2.65**

表 5.2 任务用时对比（单位：秒）

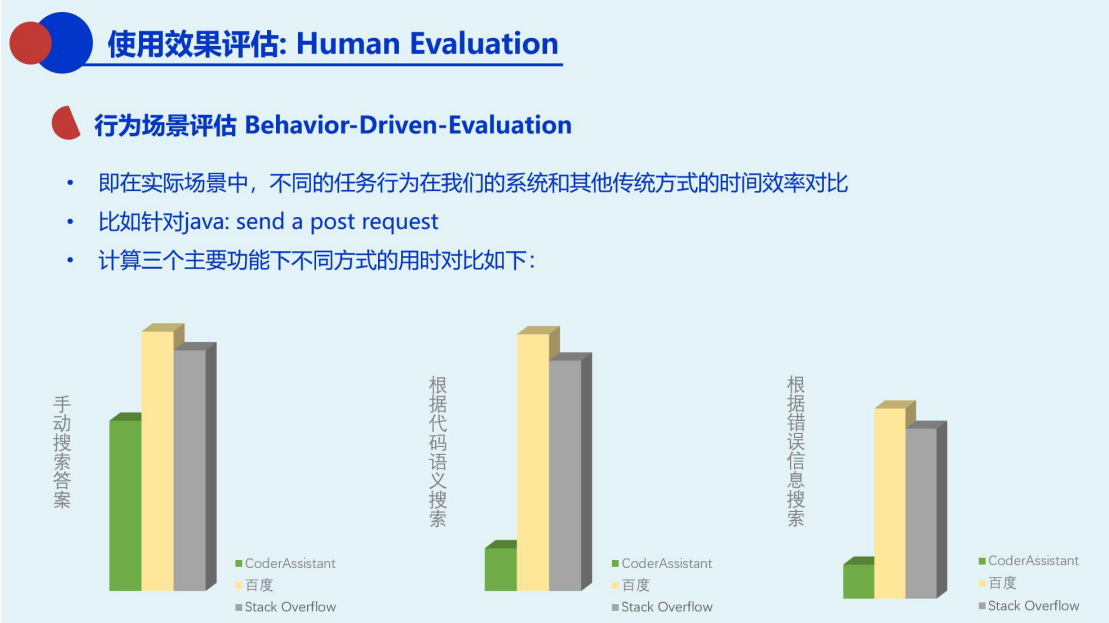


图 5.2.1 性能评估对比图

5.2.4 接口集成测试

使用 Postman 工具对本项目的接口进行接口集成测试：

接口集成测试主要检测 1.判断请求返回的结构体是否符合预期 2.判断请求返回的内容中是否包含预期的内容（关键字）

5.2.5 View 层测试

View 层的测试自然依赖于 Java 环境，同时需要模拟 UI 交互的能力，我们使用 JUnit 使用 Java Instarumentaion Tests 运行，代码存放于 ViewTest 中。

5.3 软件安全性测试

5.3.1 概述

软件安全性测试主要包括静态应用程序安全测试（SAST），动态应用程序安全测试（DAST），软件组成分析（SCA）和 API 接口安全性测试等等。

静态应用程序安全测试 （SAST）

运行静态代码扫描，SAST 工具可以被认为是白盒测试，包括架构图，对源代码的访问等。通过使用 SAST 工具检查静态源代码，以检测和报告可能导致安全漏洞的弱点。源代码分析器可以在未编译的代码上运行，以检查缺陷，例如数值错误、输入验证、争用条件、路径遍历、指针和引用等。这样可以发现潜在的漏洞，可能是由于程序员的疏忽或未考虑到边界情况造成的。

动态应用程序安全测试（DAST）

与 SAST 工具相比，DAST 工具可以被认为是黑盒测试，其中测试人员对系统没有先验知识。它们检测指示处于运行状态的应用程序中存在安全漏洞的情况。DAST 工具在操作代码上运行，以检测接口，请求，响应，脚本（即 JavaScript），数据注入，会话，身份验证等问题。DAST 工具采用模糊测试：将已知的无效和意外的测试用例抛向应用程序， 通常是大量测试，检验程序是否能应对所有这些意外情况。

软件组成分析 （SCA）

SCA 工具检查软件以确定软件中所有组件和库的来源。这些工具在识别和查找常见和流行组件（尤其是开源组件）中的漏洞方面非常有效。但是，它们不会检测内部自定义开发组件的漏洞。SCA 工具在查找常见和流行的库和组件（尤其是开源部分）方面最为有效。它们的工作原理是将代码中找到的已知模块与已知漏洞列表进行比较。SCA 工具查找具有已知和记录的漏洞的组件，并且通常会建议组件是否过期或是否有可用的修补程序。例如最近发现重大漏洞的日志组件 Log4j，经过 SCA 扫描可以发现该种依赖库问题，可以进行升级和替换进行修复

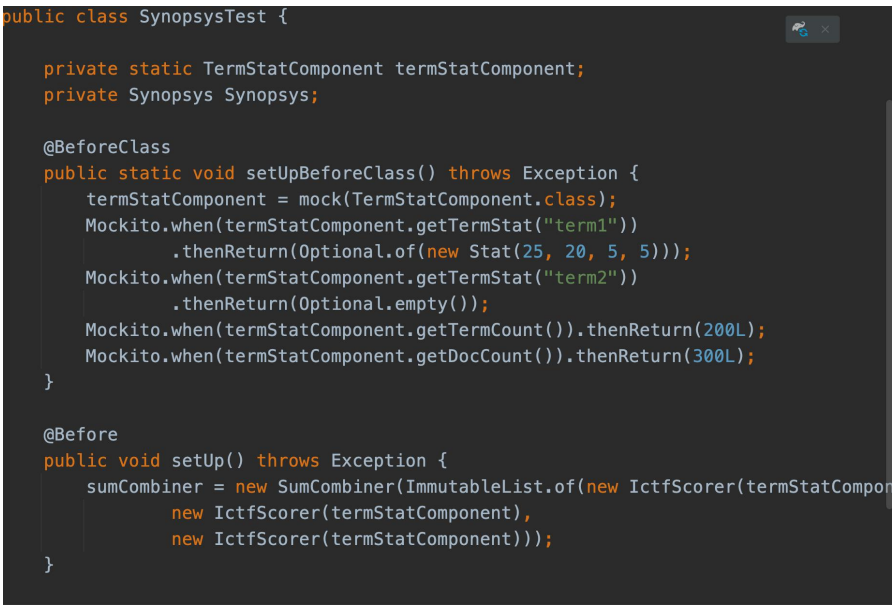
API 接口安全性测试：

1. 封闭性：接口封装在插件逻辑内部，访问需要鉴权，防止暴露在外部。
2. 流量限制：对同一用户短时间内访问次数进行限制，防止滥用现象。
3. 最小特权：运行的接口使用普通用户，同时与系统其他文件隔离。

5.3.2 基于 Coverity 进行软件安全性测试

具体来说，我们使用了 Synopsys（新思）工具辅助开发测试代码，主要使用到的工具是：Coverity® (SAST)：在编码开发过程中解决编码安全和质量缺陷。

Coverity 的智能静态分析引擎能够帮助开发者在工作流程中找出质量缺陷和安全漏洞，提供精确、可行的修复指导，在开发过程中识别关键质量缺陷，降低风险并减少项目成本。通过深刻理解行为和问题危急程度， Coverity SAVE 可以智能测试，精确找出那些潜在代码生成中 Java 代码库中的难以发现的能够引发崩溃的问题。



```
public class SynopsysTest {

    private static TermStatComponent termStatComponent;
    private Synopsys Synopsys;

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        termStatComponent = mock(TermStatComponent.class);
        Mockito.when(termStatComponent.getTermStat("term1"))
                .thenReturn(Optional.of(new Stat(25, 20, 5, 5)));
        Mockito.when(termStatComponent.getTermStat("term2"))
                .thenReturn(Optional.empty());
        Mockito.when(termStatComponent.getTermCount()).thenReturn(200L);
        Mockito.when(termStatComponent.getDocCount()).thenReturn(300L);
    }

    @Before
    public void setUp() throws Exception {
        sumCombiner = new SumCombiner(ImmutableList.of(new IctfScorer(termStatCompor
                new IctfScorer(termStatComponent),
                new IctfScorer(termStatComponent)));
    }
```

图 5.3.1 Synopsys 安全性测试

我们也使用了一些其他的工具进行测试如 Black Duck® (SCA)：在应用和容器中保护并管理开源风险、Seeker® (IAST)：在 DevOps 流水线中自动执行 Web 安全测试、Tinfoil: 将 DAST 和 API 安全测试集成到开发和 DevOps 工作流中。

5.4 持续反馈与改进

高效的反馈回路可以在规模较小，修复成本较低的情况下发现并且修复问题。并且让开发人员参与部署环节，通过接受用户真实的反馈和需求可以对产品进一步改进，降低部署风险。并且，通过不断吸收用户的信息和改进可以增加用户的参与度和转化率。

本项目计划通过 GitHub Issues 和 Intellij 的插件评论反馈等方式收集用户信息来做进一步地迭代改进。

6 实现效果和实验分析

6.1 实验环境和系统运行效果

6.1.1 实验环境和运行说明

实验环境上，在模型训练时我们采用的是实验室的 Nvidia 3090 显卡进行的训练，数据集使用的是 CodeXGlue（之前有介绍）。模型本地预测用时是在本机（笔记本电脑）进行测试的，本机配置为 R5-5800H 和 Nvidia 3060 的 GPU，操作系统 windows，使用的 Idea 版本为 2017-2。模型线上预测用时测试是在腾讯云 ecs 服务器上进行的，ecs 云服务器配置为 4 核 16GiB，操作系统为 CentOS 7.8，模型线上部署并返回预测。

本项目所有代码均已在 Github 平台上开源，插件代码库和模型代码库分别是 https://github.com/JasonZhu-WHU/CoderAssistant 和 https://github.com/JasonZhu-WHU/CoderAssistantModel。为了方便使用，我们也写出了安装和部署步骤，并全部放在了 Github 的 Readme 文件以及 Release 发布包中，同时相关运行环境和版本也均在文档中有说明，这里简单列举一下安装运行步骤：

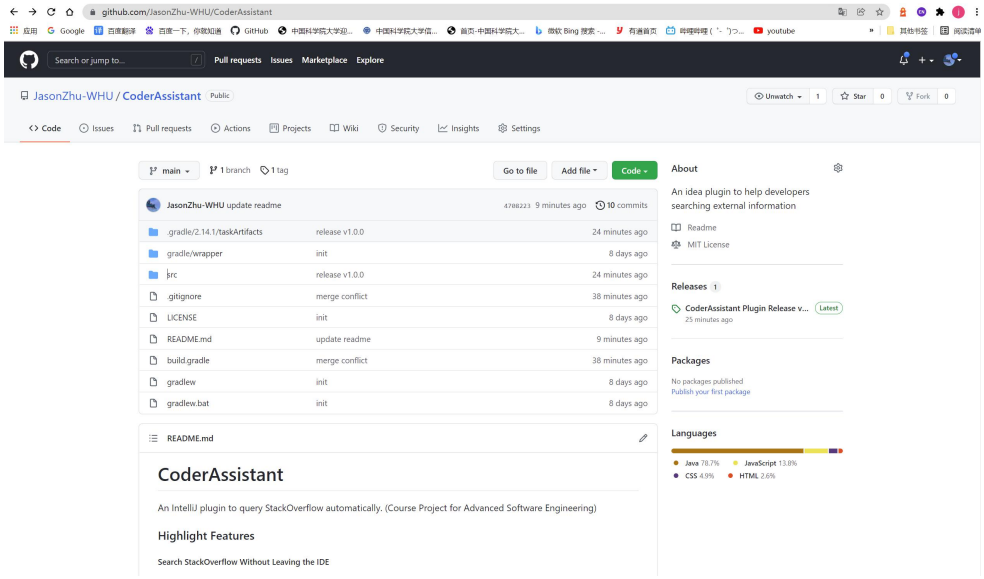
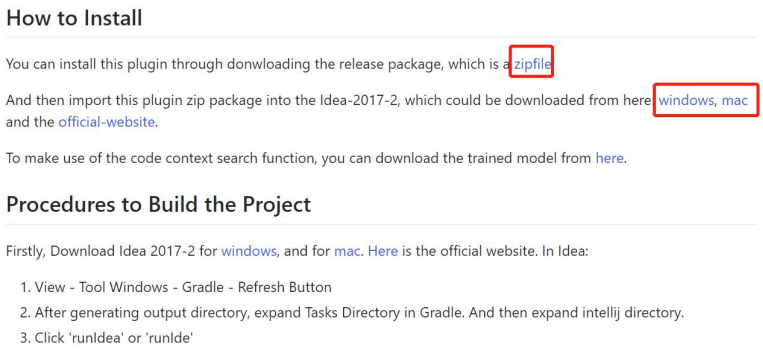


图 6.1.1.1 插件代码库（不需要全部下载）



How to Install

You can install this plugin through donwloading the release package, which is a zipfile

And then import this plugin zip package into the Idea-2017-2, which could be downloaded from here windows, mac and the official-website.

To make use of the code context search function, you can download the trained model from here.

Procedures to Build the Project

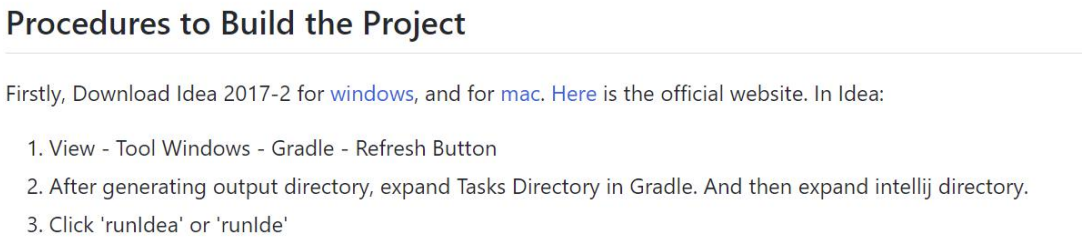
Firstly, Download Idea 2017-2 for windows, and for mac. Here is the official website. In Idea:

1. View - Tool Windows - Gradle - Refresh Button
2. After generating output directory, expand Tasks Directory in Gradle. And then expand intellij directory.
3. Click 'runIdea' or 'runIde'

图 6.1.1.2 插件包以及 Windows/Mac 版的 Idea 下载链接（README 文件中）

1. 首先在插件代码库下载所需要的 Idea 开发工具和 CoderAssistant-1.0.0.jar 插件包（如图 6.1.2 所示）
2. 然后再在下载的 Idea 开发工具中导入对应的插件包即可

如果希望从头构建部署的可以参考 Github 上的 Readme 说明，按说明运行下图中相应指令即可：



Procedures to Build the Project

Firstly, Download Idea 2017-2 for windows, and for mac. Here is the official website. In Idea:

1. View - Tool Windows - Gradle - Refresh Button
2. After generating output directory, expand Tasks Directory in Gradle. And then expand intellij directory.
3. Click 'runIdea' or 'runIde'

图 6.1.1.3 Github 上的构建步骤

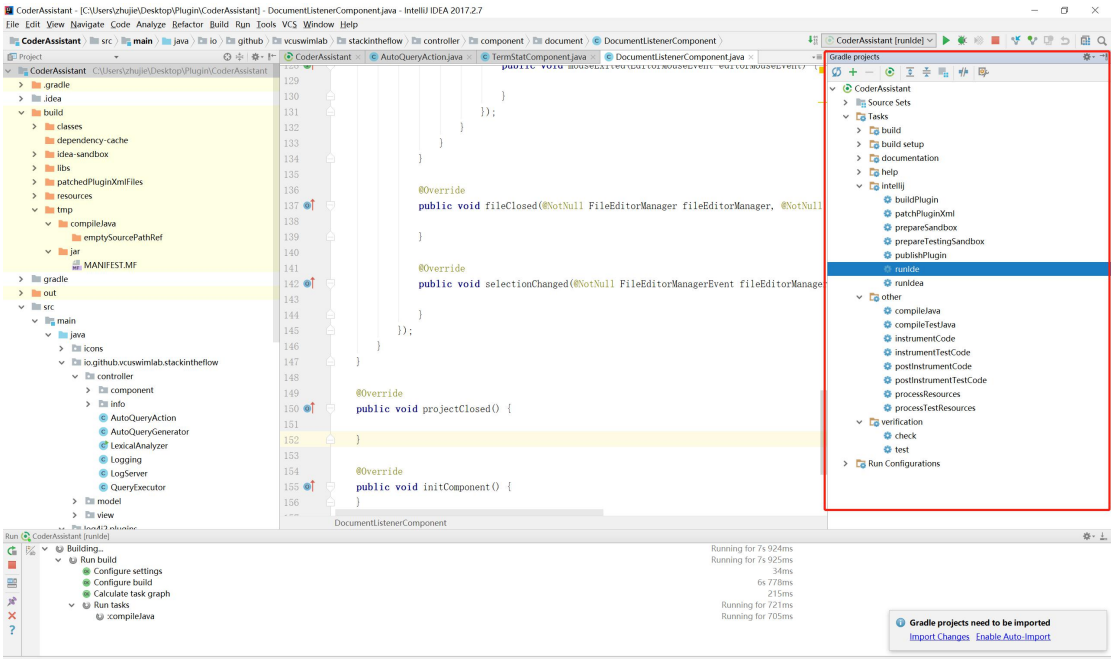


图 6.1.1.4 开发和构建的指令

最终，系统的运行实现效果如下：

6.1.2 IDE 内检索效果

CoderAssistant 提供在 IDE 内直接检索的功能，这样可以让开发者直接在 Idea 内部进行检索。相比在 Google 等搜索引擎中检索，这样的形式可以有效地过滤掉不相干的广告；相比直接在 Stack Overflow 搜索，这样也可以避免一大堆页面切换的繁琐，同时直接的看到查询的结果，并且基于的 StackExchangeAPI 在国内要比直接访问 Stack Overflow 更加稳定。总的来说，相比于打开浏览器，再在 Google 或者 SO 上检索的方式，直接在 IDE 内进行检索会更加的方便高效，避免开发中的思路被干扰。

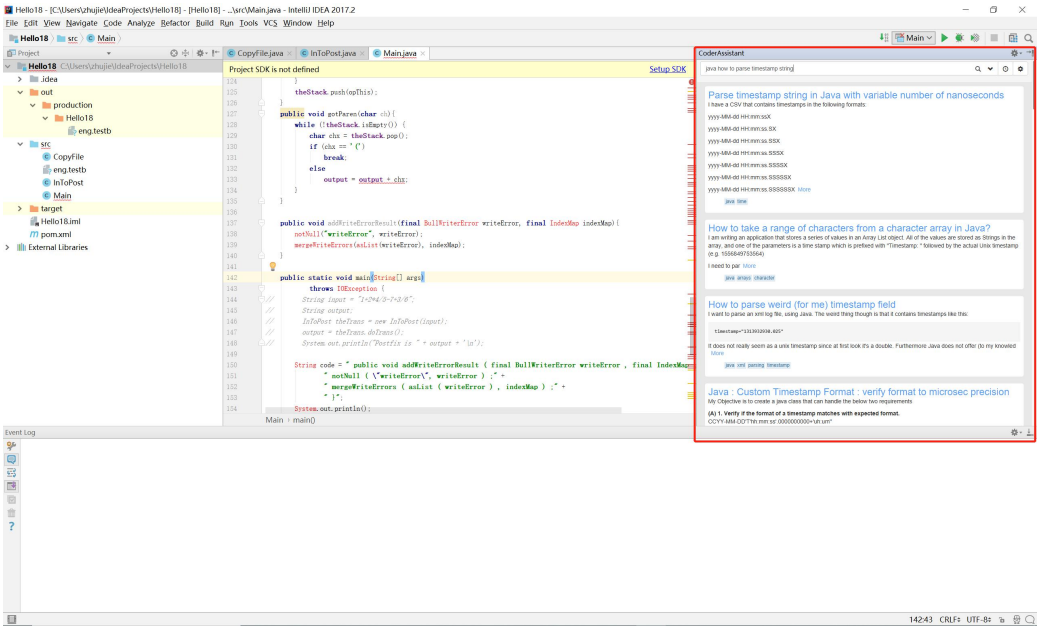


图 6.1.2 IDE 内输入查询检索

6.1.3 根据报错信息自动检索效果

CoderAssistant 还提供根据错误信息自动检索的功能。比如下图所示，在程序运行出现 Error 时，CoderAssistant 会自动截取第一行的错误信息作为查询 query 到 StackExchange 的 API 去检索，并将检索到的答案自动返回并展示。传统 Debug 或者检索答案的方法往往需要先复制错误信息，然后去粘贴到网站上，这样比较麻烦而且搜索多了之后打开的页面可能会很混乱。CoderAssistant 会自动根据报错信息去搜索，开发者只需要直接查看检索得到的问答结果即可。

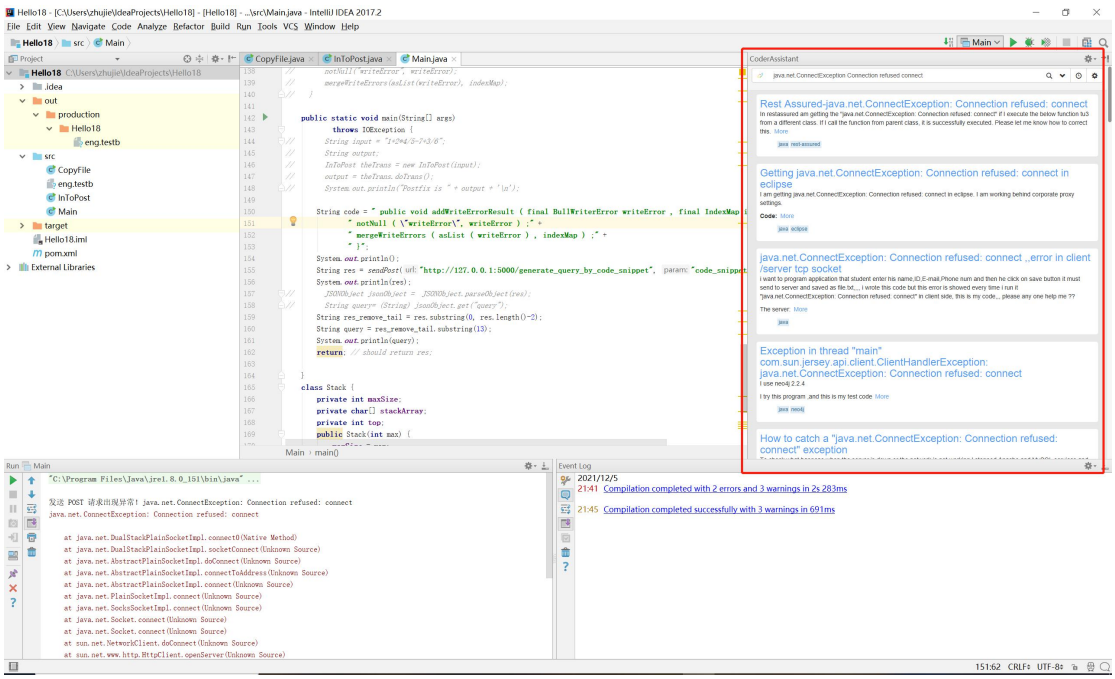


图 6.1.3 根据报错信息自动检索

6.1.4 提取代码语义信息进行检索效果

CoderAssistant 还可以代码语义信息进行自动检索。比如下图所示，开发者可以选取一段代码，然后 CoderAssistant 会自动对代码进行分词、解析，并将代码段字符串发送到模型中进行处理，获得代码语义信息的自然语言描述文本。然后将这一段文本作为查询 query 到 StackExchange 的 API 去检索，并将检索到的答案自动返回并展示。

传统 Debug 或者检索答案的方法往往对于开发者自身的技术水平有较高的要求，需要开发者能看懂或者搞明白代码到底在做什么事情，但是这一点对于一些新手程序员或者不太熟悉某种语言或框架的程序员来说比较困难。而我们的工具 CoderAssistant 可以帮助开发者直接生成针对某一段代码的 query 查询，从而帮助他们更好的找到高质量的问答结果，同时节省理解代码和写查询的时间。

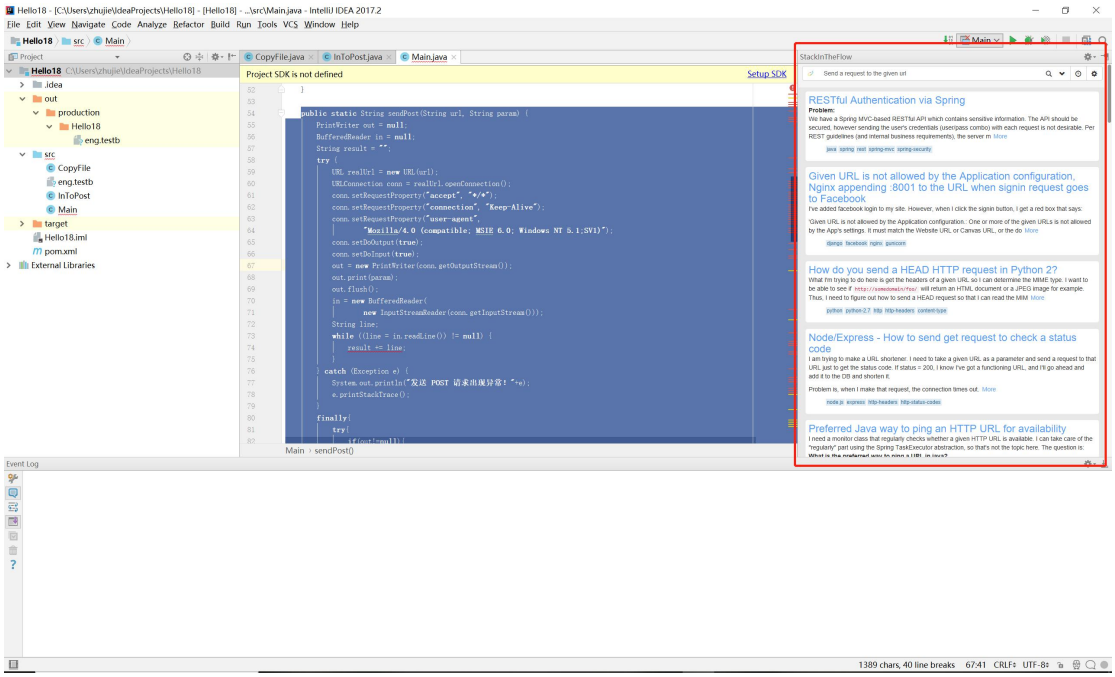


图 6.1.4 提取代码语义信息自动检索

可以看到，从代码语义信息进行自动检索的效果非常不错，上面这段代码其实就是一段给服务器发出 Post 请求的代码，可以看出后台的训练好的模型可以很准确的估计出代码的意思，并回传给插件端到 Stack Overflow 上检索答案。综合上述的介绍和分析，从实现效果上看我们基本实现了项目的预期功能。接下来一节会从各个方面对实现的效果进行具体评估。

6.2 实验评估

在我们项目中，我们对于实现效果评估主要是针对下面这些方面：插件代码提取拉选部分和错误信息的功能，自动化错误信息检索功能，代码语义提取功能的准确率等。

评估难点：根据代码生成的语义信息以及根据语义信息检索的相关内容是否合乎开发者的预期需求；如何测试插件代码是否自动化地获取到相关信息；检索性能是否符合要求。

评估测试目标：主要功能模块下单元测试数量不少于 10 个，测试覆盖率达到 80%以上；训练的代码语义提取模型能够提取出代码的语义信息，Rouge-L 达到 0.15 以上；插件查询信息的时间效率与传统方法相比能节省 2 倍以上的时间。

实验环境：软件环境即 Intellij Idea, Pytorch，硬件环境使用到的移动端笔记本 GPU 3060 AMD-5800H

测试评估工具：Junit Postman PyRouge 等

将 CoderAssistant 与其他传统方式（百度/Stack Overflow 搜索）在三个具体使用场景下进行对比：分别统计手动搜索用时对比，代码信息查询用时对比，错误信息搜索对比。

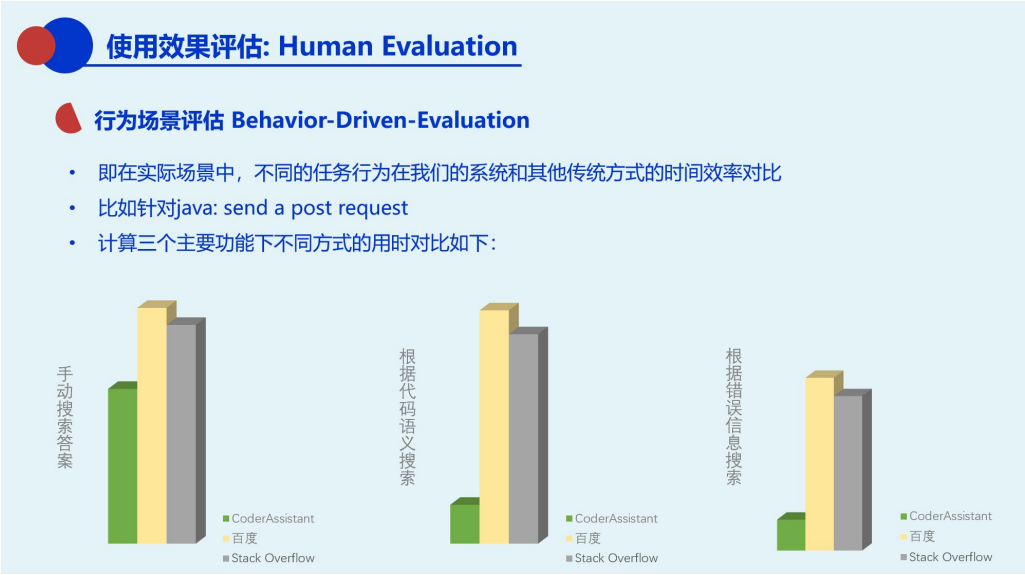


图 6.2.1 使用场景效果对比

关于对模型效果的评估如下，分别分为量化评估和案例评估：

首先是量化评估的标准，我们选择 Rouge-L 作为评估的指标。Rouge 是评估自动文摘以及机器翻译的一组经典指标，通过将自动生成的摘要或翻译与一组参考摘要（通常是人工生成的）进行比较计算，得出相应的分值，以衡量自动生成的摘要或翻译与参考摘要之间的“相似度”。在这里，我们使用 Rouge-L 作为代码语义信息提取的一个主要评价指标，Rouge-L 的 L 即是求 LCS(longest common subsequence，最长公共子序列)的长度，下面是 Rouge-L 的计算方法：

$$\mathrm{R}_{\mathrm{lcs}} = \frac{\mathrm{LCS}(X,Y)}{m}$$

$$\mathrm{P}_{lcS} = \frac{\mathrm{LCS}(X,Y)}{n}$$

$$\mathrm{F}_{lcs} = \frac{\left(1+\beta^2\right) R_{lcs} P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}}$$

图 6.2.2 Rouge-L 计算公式

——这里 X 是人工生成的描述文本(Gold Standard)，m 是其长度；Y 是自动生成的描述文本，n 为 Y 的长度

经过在测试集上对比，使用 pyrouge 库计算，我们的模型 Rouge-L 值达到了 0.17，能很好的提取出代码语义信息，达到了预期中大于 0.15 的目标。

为了更好的说明模型生成的效果，这里我们使用一个具体的案例进行说明：如下图是一个我们需要进行语义信息抽取的代码段，其主要功能就是给指定的 url 发出请求并携带参数 params：

```java
public static String sendPost(String url, String param) {
    PrintWriter out = null;
    BufferedReader in = null;
    String result = "";
    try {
        URL realUrl = new URL(url);
        URLConnection conn = realUrl.openConnection();
        conn.setRequestProperty("accept", "*/*");
        conn.setRequestProperty("connection", "Keep-Alive");
        conn.setRequestProperty("user-agent",
                "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;SV1)");
        conn.setDoOutput(true);
        conn.setDoInput(true);
        out = new PrintWriter(conn.getOutputStream());
        out.print(param);
        out.flush();
        in = new BufferedReader(
                new InputStreamReader(conn.getInputStream()));
        String line;
        while ((line = in.readLine()) != null) {
            result += line;
        }
    } catch (Exception e) {
        System.out.println("发送 POST 请求出现异常！"+e);
        e.printStackTrace();
    }
    finally{
        try{
            if(out!=null){
                out.close();
            }
            if(in!=null){
                in.close();
            }
        }
        catch(IOException ex){
            ex.printStackTrace();
        }
    }
    return result;
}
```

图 6.2.3 代码语义信息抽取例子

这里的标准的输出结果（Golden）应该是 Send a post request with params to the given url，模型的输出结果为 Send a request to the given url，可以看出我们模型提取的效果还是比较好的。

6.3 有效性风险

我们项目效果和结论的有效性风险 Threats to Validity 可以分为下面几个部分：

1. 内部有效性风险：项目内部有效性风险依赖于构成功能需求的插件，以及实现语义抽取的模型两部分。关于插件，我们对其进行了完整的测试，包括 50 多个一共 900 多行的单元测试代码，以及性能测试等等。针对模型，我们也对其抽取的效果进行了量化的评估，结果得到的 rouge 值大于 0.15 符合预期。

2. 外部有效性风险：外部有效性风险主要是关于项目的可扩展性和可移植性，项目扩展到其他平台和语言的主要成本在于开发对应插件的成本，即核心代码逻辑以及模型是可以很方便的进行迁移或者重新训练。主要困难在于开发对应平台插件的过程。
3. 意义和结论的有效性风险：意义和结论的有效性风险主要在于我们的项目能否达到预期的效果，即给用户提供在 IDE 内自动检索高质量问答的能力，从而节省到外部手动搜索的时间。这一点上，我们经过相关实验验证，从与传统方式对比，以及本身项目的实现效果上证明了我们相比之前的手工查询方式可以有效地减小数倍的时间，同时避免了对开发者注意力切换的干扰，达到了项目的预期目标，对于我们方法和工具的意义和结论不会有影响。

7 遇到的问题与解决方案

总结项目开发中遇到的问题，以及我们是如何解决的，对于无法解决的问题解释原因和局限性。

7.1 交互方案选择与设计

其实在开题报告中，我们最初的计划是设计一个简易的前后端系统，用户输入查询或者问题，然后 CoderAssistant 自动解答回答。这种方式有很多优点，比如技术实现方便（Web 技术比较成熟），用户试用方便（直接访问网页 URL）等等。但是**为什么我们最后选择了基于 IDE 内的插件式方案呢**，主要原因是我们经过实际调研和试用对比后发现，基于 Web 网页的工具体验在整体上是远远比不上在 IDE 内插件自动查询的体验的。

主要原因有下面这些点：

1）从与现有方式对比的角度看，在 **Web 端网页访问 CoderAssistant 的体验，与直接访问搜索 Stack Overflow 或者 Google 的体验差不多**。我们所能做的无非就是去除广告，提升搜索质量，但是所能带来的提升是很有限的。

2）从使用流程上看，在 IDE 内部的插件，可以直接自动获取代码或者是调试的错误信息，这样自动进行问答检索，在使用体验上更方便简单，不需要手动去复制或者输入很多东西。

3）从开发体验看，IDE 内的插件可以给用户带来沉浸式的开发过程体验，不需要到处切换，所有的信息都在 IDE 的开发面板中展示出来。

4）从时间开销等性能角度讲，IDE 内的插件直接生成查询并请求搜索，比经过 Web 端作为入口平台再去搜索，可以节约访问 Web 的时间，性能更好

5）从实际应用角度看，如果以后给更多的人使用，那么如果全部依靠 Web 端作为入口，当用户变多的时候，后台部署 GPU 等设备的运算成本较高；相比之下插件式的方案可以更方便地支持用户在本地 docker 容器中运行环境，用户量增加不影响使用性能和成本。

7.2 插件方案选择

在确定了插件方案后，我们遇到的下一个问题就是在什么平台上开发插件，最开始的计划是在 VSCode 平台上进行开发。VSCode 平台插件开发的优势在于 VSCode 本身就是一个基于 Electron 的跨平台代码编辑器，编写插件的过程就类似 Web 开发的一系列过程，相对简单方便。同时 VSCode 支持不同语言和框架的开发，既可以写 Web 的前端项目，也可以写各种后端项目，还可以开发模型代码或者编写脚本。但是我们在仔细调研了 VSCode 的 API 文档和一些其他参考资料后发现，VSCode 在错误信息调试上并不是直接去解析 AST 并进行语法语义分析，而是设计了一个 Debug Adapter Protocol 将编辑器和编程语言/调试服务的功能分离开。

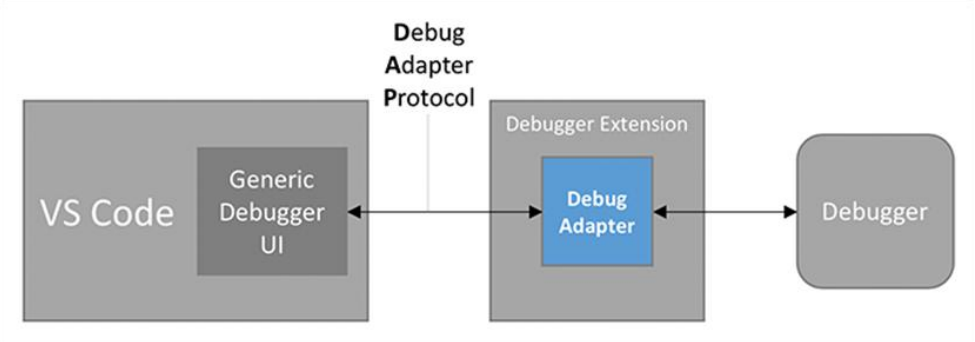


图 7.2.1 VS Code Debugger Architecture

因此，VS Code 开发程序中的具体的错误和调试信息并不是在 VSCode 内解析的，而是在外部 Debugger 运行，之后再传入显示到内部 GUI 中，这固然在支持多语言方面节省了很多操作和开销，但是也为插件在获取错误和调试信息时造成了很多麻烦。而目前来看，VS Code 也没有提供获取这些信息的接口和 API，因此我们认为构建基于 VS Code 的插件的方案对于我们来说比较麻烦，需要了解一系列底层机制，开发成本比较高，目前在获取错误信息等 API 时会有很多问题。但是问题整体的解决思路是类似的，所以我们计划先在其他平台上进行方法有效性验证后再在 VS Code 上进行探索，当然如果未来能解决这个问题也会很有意义。具体来说，我们选择在 Intellij 公司的 Idea 这个主要用于 Java 语言的集成环境上进行插件开发。同时 IntelliJ 在业界也被公认为最好的 java 开发工具之一，而且提供了比较成熟的插件开发接口供开发者使用。因此，我们最后提供

的插件是基于 Idea 进行设计开发的，迁移到其他 IDE 或者编辑器可能需要对其 API 和接口有更多的了解。

7.3 模型运行和部署问题

另外一个碰到的主要问题是在模型的运行部署上。众所周知，深度学习的模型在训练过程所需的资源是巨大的，而即使是训练完成后，做预测的时候，在输入文本比较大的情况下依然会面临性能的考验。我们的模型对于正常的比如十几行的代码段生成 query 所需时间大约在 0.2s 左右，但是当迁移到服务器的时候，服务器 CPU 在预测时大约需要 3-5s 的时间，这个等待时延对于体验的影响比较大。因此，我们认为这里存在一个取舍 tradeoff，即应该将这个选择交给用户，用户可以根据自己的需求去进行选择：用户如果选择部署在本地，那么模型预测效果更快但是需要下载模型并运行，这种方案我们可以提供一个 docker 的镜像，用户可以在本地运行 docker 容器来进行预测；另一种方式是可以让用户使用线上的模型进行预测，但是考虑到 GPU 服务器成本所以模型预测的耗时更久，好处就是不需要任何其他操作。另一种可行的方式是设计一些基本的启发式算法或者不涉及深度学习模型的方式，来更高效快速地根据代码生成 Query 描述。

8 总结与展望

8.1 项目总结

本项目 CoderAssistant 是一个面向软件开发领域的问答机器人，选题主要是关于聊天机器人在特定领域的一个具体应用，希望能像一个助手一样，帮助软件开发人员更高效快速地找到他们在编程过程中遇到的问题的解决方案。具体来说，CoderAssistant 是一个在 Java 语言最常用的集成开发环境 Idea 上的插件，可以帮助用户更方便地在 IDE 内部解决他们开发中遇到的问题，其主要核心特色在于：提供了自动化的基于错误信息和代码段的问答检索功能，以及完全在 IDE 内的沉浸式检索体验。在传统的方式中，开发者遇到程序错误 Bug 时往往会经过如下这几个步骤：首先根据 bug 信息和代码进行分析，然后复制一些关键的错误信息以及代码段到 Google 或者 Stack Overflow 上，然后再肉眼过滤掉一些不相干的网页和广告，最后找到一个比较好的答案网页进行学习，直到解决 Bug。但这种方式的操作和步骤比较繁琐，手动切换的页面和输入也比较多，而且不同水平的程序员他们 Debug 分析错误的能力也不同，所以 CoderAssistant 希望能够将这个过程自动化，帮助软件开发人员更快更准地找到高质量的问答和解决方案。

我们工作的主要亮点和贡献可以总结为下面几点：

1. CoderAssistant 是第一个将代码语义信息提取和 Stack Overflow 问答检索相结合的工作，并使用了目前最新的 CodeBERT 预训练模型和 CodeXGlue 数据集使得语义信息提取的效果较好
2. 开发了一个基于代码语义信息和报错信息的自动化问答检索工具，将之前的繁琐的人工手动查询外部信息过程自动化，为开发者节省了搜索时间
3. 通过实际例子验证了方法和工具的有效性，并基于数据集对语义提取效果进行了准确率的量化验证评估

对于项目的整体总结可以用三个亮点+一个目标+一个不同点来概括如下：

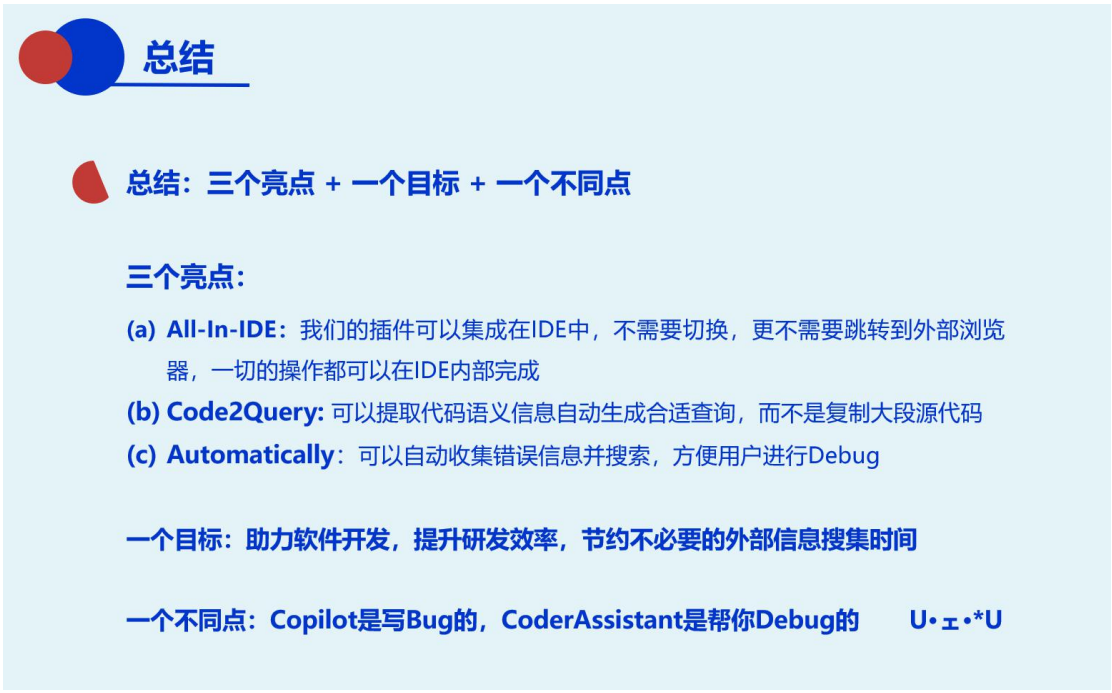

总结
总结：三个亮点 + 一个目标 + 一个不同点
三个亮点：
(a) All-In-IDE：我们的插件可以集成在IDE中，不需要切换，更不需要跳转到外部浏览器，一切的操作都可以在IDE内部完成
(b) Code2Query: 可以提取代码语义信息自动生成合适查询，而不是复制大段源代码
(c) Automatically：可以自动收集错误信息并搜索，方便用户进行Debug
一个目标：助力软件开发，提升研发效率，节约不必要的外部信息搜集时间
一个不同点：Copilot是写Bug的，CoderAssistant是帮你Debug的 U•ェ•*U

图 8.1 项目总结

8.2 未来展望

受限于客观条件因素，开发人员的时间和精力能力有限。项目也有一些不足和未来可以进一步开发的方向，这里将项目的展望和 Future Work 总结如下：

1. 目前项目是基于 Idea 的一个版本进行的尝试性实现，只支持 Java 语言，但是理论上我们的方法是可以推广到其他语言和平台上的。未来可以进一步推广到 VS Code 等跨语言的编辑器上，但是需要解决错误信息提取等问题，并重新编写对应的 UI。
2. 目前提供的三个功能中，代码语义信息检索需要显卡 GPU 的支持，在 CPU 或者云服务器上运行速度较慢。参考目前的类似插件基本上也都是采用的本地 docker 运行部署，这一块还需要进一步研究，或者也可以提供一种简单的启发式的信息提取算法作为备选。
3. 目前代码语义提取的研究还处于一个初始状态，CodeBERT 等模型目前在 Rouge 值上也只有 0.17 左右，距离文本摘要/信息提取的差距还比较大。如果有更好的模型和方法，检索效果也会有进一步提升的空间。

另外我们的指导老师罗铁坚老师也在答辩中为我们进一步改进提升提供了很多具体的指导建议，其中包含很多在功能和目标上的拓展。因为时间人力等客观因素，我们无法在本次课程中一次完成，但是我们认为这对于我们软件的进一步迭代发展是至关重要的。因此，我在这里一并进行分析总结，为我们未来的进一步开发确定方向：

1. **增加软件的“后端记忆”，即记录每个人的搜索历史并提供总结回顾的交互方式**。这是因为我们目前的插件主要是一个基于代码错误信息和语义信息的检索，这主要是**从客观上通过帮助程序员解决开发中的各种问题来实现研发效率的提升**。但是**人的写代码的效率提升，其实更多的是来自于自己在技术和思考上的沉淀，那么通过学习过去遇到的问题，无疑可以帮助程序员不断总结提升，从而构成一个学习成长的过程**。比如告诉用户，他/她今年遇到了哪些问题，哪些问题多次遇到，帮助用户学习回顾，形成一种从解决问题中沉淀总结的提升方式。

2. 另一个是构建一个整体的全体用户的搜索历史知识库，这个可以与个人的知识库相结合，能挖掘出每个时间段有哪些技术问题是比较受到关注或者比较重要比较常见的。并且可以提供给用户从整体的角度去思考问题的方式，了解目前流行的一些工具框架，更高效地从问题中学习。
3. 对于 Codex 的对比，罗老师建议我们不要拒绝这个从文本生成代码的 Text2Code 的功能，可以融入到系统中互为补充。这一块的主要挑战在于更进一步的明确功能需求场景，从行为和场景中建模需求，明确它的核心价值和意义在哪里。另一方面，对于模型代码生成的效果，能否在时间和机器性能都有限的情况下达到不错的准确率，依然是一个有挑战性的工作，未来也是值得探索的。

总的来说，提升研发效率，不是简简单单的一个软件或者插件就能解决的，里面涉及到的各种流程和能力是相当复杂的。如果我们将研发效率 E 定义为关于个人能力、搜集信息能力等参数的一个函数，即 E = f (个人能力, 搜集外部信息解决问题的效率…)。那么要提升研发效率，光提升“搜集外部信息解决问题的效率”是不够的。而个人能力可以从解决问题中不断学习、总结、沉淀得到。所提我们的项目只涉及到了简单的从外部搜集信息解决问题的效率，而没有考虑到更多的提升可能。老师则将提升开发者个人能力也引入了进来，所以未来我们的项目还具有非常广阔的可探索空间和应用价值。

附件

海报 Poster：

CoderAssistant

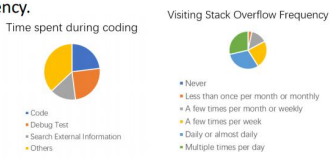
作者：祝捷，闫熠光，徐泽杉 指导老师：罗铁坚

中国科学院大学 University of Chinese Academy of Sciences

ISCAS 中国科学院软件研究所 Institute of Software Chinese Academy of Sciences

Motivation

Programmers spend lots of time on searching external information from sites such as Stack Overflow and Google for debugging, during which process they might even be distracted. Consequently, saving time and energy on searching information from external knowledge library such as Stack Overflow could significantly enhance R&D efficiency.

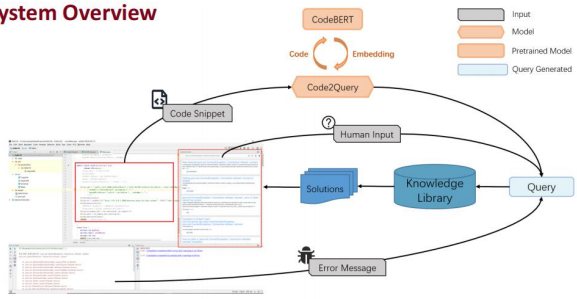


Key Features

CoderAssistant automates the process of searching external information for debugging and help beginners find high-quality solutions easier.

1) Help analyze code snippets by retrieving code context information
2) Searching external information with error messages automatically
3) All interacting actions could be done inside IDE

System Overview



Experiments

Dataset: CodeXGlue[2] (6 languages)　　Model Evaluation: Bleu-4

Programming Language	Training	Test	Valid	Languages	Ours	Seq2Seq
Java	**164923**	**5183**	**10955**	Java	**17.25**	15.09
Python	251280	13914	14918	Python	**19.06**	15.93
Go	167288	7325	8122	Go	**18.07**	13.98
...	...	...	...	...	...	...

Result of other 3 languages is hidden due to the space limit

Human Evaluation

Comparing time taken by different methods

Human Evaluation: Time saved by 7X times with CoderAssistant

Models and code are publicly available:

https://github.com/JasonZhu-WHU/CoderAssistant

https://github.com/JasonZhu-WHU/CoderAssistantModel

And our work is built based on open source software including: CodeBERT[1], CodeXGlue[2] and Stackintheflow[3]. Usage is only limited for the course(Advanced Software Engineering) practice.

[1] Feng Z, Guo D, Tang D, et al. Codebert: A pre-trained model for programming and natural languages[J]. arXiv preprint arXiv:2002.08155, 2020.
[2] Lu S, Guo D, Ren S, et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation[J]. arXiv preprint arXiv:2102.04664, 2021.
[3] Greco C, Haden T, Damevski K. StackInTheFlow: behavior-driven recommendation system for stack overflow posts[C]//Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings. 2018: 5-8.