南京大学

本 科 毕 业 论 文

院　　系　计算机科学与技术系

专　　业　信息与计算科学

题　　目　基于大语言模型的方法注释生成工具的设计与实现

年　　级　2020　学　　号　201502006

学生姓名　周意

指导教师　潘敏学　职　　称　副教授

提交日期　2024 年 6 月 1 日

南京大学本科毕业论文（设计）
诚信承诺书

本人郑重承诺：所呈交的毕业论文（设计）（题目：基于大语言模型的方法注释生成工具的设计与实现）是在指导教师的指导下严格按照学校和院系有关规定由本人独立完成的。本毕业论文（设计）中引用他人观点及参考资源的内容均已标注引用，如出现侵犯他人知识产权的行为，由本人承担相应法律责任。本人承诺不存在抄袭、伪造、篡改、代写、买卖毕业论文（设计）等违纪行为。

作者签名：周意

学号：201502006

日期：2024.5.31

南京大学本科生毕业论文（设计、作品）中文摘要

题目：基于大语言模型的方法注释生成工具的设计与实现

院系：计算机科学与技术系

专业：信息与计算科学

本科生姓名：周意

指导教师（姓名、职称）：潘敏学　副教授

摘要：

随着人工智能和机器学习技术的快速发展，自动代码注释生成工具已经成为软件开发领域的热门研究话题。这些工具的目标是通过自动化的方式生成代码注释，以提高代码的可读性和可维护性，同时减轻开发者的工作负担。

基于大模型的方法注释生成工具是这一领域的重要研究方向。这类工具通常使用预训练的大型语言模型（如 GPT-3、CodeT5+ 等）作为基础，通过微调的方式适应特定的代码注释生成任务。这种方法的优点在于，预训练的大型语言模型已经学习了大量的语言知识和编程知识，因此可以生成高质量的代码注释。

本文通过微调已有的预训练大语言模型的方式，实现了一种可以根据用户的 java 代码内容，自动生成与语句相对应的代码注释工具。该工具使用 codeT5+ 等模型为基础，并以 apache-commons，guava，joda 等流行的 java libraries 中的代码片段为数据集进行微调工作，从而实现对代码注释实行自动生成。本文的主要工作包括：

1. 使用 eclipse JDT 对多种知名 java libraries 进行代码切片与收集，主要收集对象为：java 方法中抛出异常/处理异常的代码语句及其 javadoc 注释，返回值语句/处理返回值的语句及其 javadoc 注释，除去异常和返回值的语句的其余部分及其 javadoc 注释。每组数据以“代码-注释”的数据对格式加载，完成了对数据集的构建。

2. 通过已构建的数据集，利用 hugging face 的 transformers，datasets 等工具对 CodeT5+ 等大模型进行微调。

3. 对微调前后的大模型的代码注释生成能力进行比较和评估，根据实验数据选出最佳模型作为代码注释生成工具，并通过实例展示的方式说明微调前后的模型代码生成能力差距。

总的来说，微调工作对模型的代码注释工作有着显著的促进效果。微调后的模型相比之前，能减少关键词遗漏，并正确概括代码内容。

关键词：大语言模型；微调；代码注释

南京大学本科生毕业论文（设计、作品）英文摘要

THESIS: Design and implementation of a method comments generation tool based on large language models

DEPARTMENT: Computer Science and Technology

SPECIALIZATION: Information and Computational Science

UNDERGRADUATE: Zhou Yi

MENTOR: Pan Minxue ,Associate professor

ABSTRACT:

With the rapid development of artificial intelligence and machine learning technologies, automated code comments generation tools have become a popular research topic in the field of software development. The goal of these tools is to generate code comments in an automated way to improve the readability and maintainability of the code, as well as to reduce the workload of developers.

Large model-based method comments generation tools are an important research direction in this area. Such tools usually use pre-trained large language models (e.g., GPT-3, CodeT5+, etc.) as a base, which are fine-tuned to adapt to specific code comments generation tasks. The advantage of this approach is that the pre-trained large language models have already learned a large amount of language knowledge and programming knowledge, and thus can generate high-quality code comments.

In this thesis, by fine-tuning an existing pre-trained large language model, we implement a code comments tool that can automatically generate code comments corresponding to statements based on the content of the user's java code. The tool uses models such as codeT5+ as a base and works on fine-tuning the data set with code snippets from popular java libraries such as apache-commons, guava, joda, etc., so as to realize the implementation of automatic generation of code comments. The main work of this thesis includes:

1. use eclipse JDT on a variety of well-known java libraries for code slicing and collection, the main collection object for: java methods throw exceptions/handle exceptions in the code statements and their javadoc comments, return value statements/pro-

cessing return value statements and their javadoc comments, remove exceptions and return value of the statement of the rest of the part and its javadoc comments. Each set of data is loaded in a ”code-comment” data pair format, which completes the construction of the data set.

2. Using the constructed dataset, we fine-tuned CodeT5+ and other large language models using tools such as transformers and datasets of hugging face.

3. Compare and evaluate the code comments generation capability of large models before and after fine-tuning, select the best model as the code comments generation tool based on the experimental data, and illustrate the gap between the code generation capability of the models before and after fine-tuning by means of examples.

Overall, the fine-tuning work has a significant effect on the code-comments generation work of the model. The fine-tuned model reduces keyword omissions and correctly summarizes the code content compared to the previous one.

KEYWORDS: Large Language Models; fine-tune; code-comments

目　录

第一章　绪论

1.1　研究背景

在书写代码开发软件时，代码注释一直是人们重点关注的对象。代码注释可以帮助开发者和其团队快速阅读该代码片段。在大型项目中，同一段代码往往会被多个开发者阅读，此时代码的简单易读会带来更高的工作效率。目前对开发人员来说，对多个关键代码进行注释编写往往会花费不少时间和精力。同时不同开发者的注释风格不同，也会导致其他开发者的难以阅读。

自动代码注释生成工具的作用在于帮助开发人员快速生成高质量的代码注释，而不必手动编写每一行注释。自动生成工具可以大大缩短注释编写时间，从而提高开发效率。同时在团队协作中，代码注释的一致性非常重要。自动生成工具可以确保注释风格和格式保持一致，避免因个人习惯而导致的不一致另外，如果可以根据代码的实际功能和目的生成准确、清晰的注释，将有助于开发人员更好地理解代码，减少后续维护和调试的难度。

当谈到大型语言模型（LLMs）时，我们指的是具有大规模参数的神经网络模型，用于生成和理解自然语言。这些模型通过在大量无标记文本上进行训练，能够更好地理解和生成人类语言。而且大型语言模型可以快速学习大量数据，在学习之后对各种下游任务都有着较强的泛化能力，因此能够在各种任务上表现出色，包括自然语言处理、图像识别、语音识别。它们也能够更准确地预测下一个词，理解文本的语义，并生成更流畅、自然的文本、图像和音频内容。更重要的是，它可以大幅度减少人工劳动和成本，从而实现许多过程的自动化，例如情感分析、顾客服务、内容创作等。

市面上绝大多数的大语言模型并不能直接用于开发工具。预训练的大型语言模型（如 GPT-3、BERT 等）通过大规模的无标记文本数据学习了丰富的语言知识，但这些知识是通用的，不针对特定任务。所以要对大模型进行微调工作。微调的目的是在保持通用性的基础上，使模型能够更好地适应特定领域、任务或

应用。微调过程中，一般使用特定领域的标记数据（例如文本分类、命名实体识别、对话生成等）来训练模型。让模型可以学习到特定任务的相关知识，从而提高在该任务上的性能。而且从训练成本上来说，大型语言模型的训练成本非常高，对于一些规模较小的工作，从零开始训练大模型的性价比非常低。微调则是一种更经济、高效的方法，可以在已有模型的基础上进行定制，而无需重新训练整个模型。

随着大语言模型的众多优点被发掘，其对于开发自动代码注释生成工具也有许多推进作用：自动化，格式规范，表达清晰等，这些正是代码注释所需要的。因此，通过再次微调训练已有的大语言模型，使其能够针对注释生成这一点有着更好的表现，成为一种新兴的研究方式。

1.2 研究现状

大语言模型自 2018 年 GPT（Generative Pre-trained Transformer）[1]模型诞生以来，模型的规模不断扩大。从最初的 GPT-1 到 GPT2[2]如今的 GPT-3[3],InstructGPT[4]和 GPT-4[5]，模型参数从数百万增加到数十亿[6]。这种规模的增加使得模型能够更好地捕捉语言的复杂性和上下文信息。大语言模型主流上采用预训练和微调的方法，在预训练阶段，模型通过大规模的无监督学习从文本数据中学习语言知识；在微调阶段，模型在特定任务上进行有监督的训练，以适应具体应用场景。

目前，大语言模型在文本生成、对话系统、机器翻译等方面表现出了强大的生成能力。例如，GPT-3 可以生成连贯、有逻辑的文章段落，甚至可以写诗、编写代码。其强大的功能也在许多应用领域有所建树：首先是自然语言生成：自然语言生成是计算语言学的一个分支，旨在使计算机具有类似人类的表达和写作能力。主要根据一些关键信息和其在机器内部的表达形式，经过规划过程，自动生成高质量的自然语言文本。大语言模型在自然语言生成方面应用广泛。它们可以用于自动摘要、文章创作、广告文案等。

大语言模型在机器翻译领域也有显著的应用。机器翻译（Machine Translation，MT）是利用计算机将一种自然语言翻译成另一种自然语言的过程。通过使用语料库等技术，机器翻译可以实现更复杂的自动翻译，包括处理不同的文法结构、词汇识别、惯用语的对应等。例如，使用 GPT-4 进行跨语言翻译，用户可以获得

更准确的翻译结果。

代码注释生成工具是当前程序理解研究领域的一个研究热点。这种工具的目标是自动为代码生成注释，以减轻开发人员的负担，提高软件开发效率。现有的代码注释生成方法主要分为三类：基于模板的方法、基于信息检索的方法和基于深度学习的方法。

基于模板的方法通常使用预定义的模板来生成代码注释。模板中的占位符会被替换为从代码中提取的信息。这种方法的优点是生成的注释通常结构清晰，容易理解。但是它无法处理复杂的代码结构和语义。

信息检索则使用信息检索技术来从大量的代码和注释库中查找与目标代码相似的代码段，并将这些代码段的注释用作目标代码的注释。此类方法需要利用大量的已有代码和注释资源，生成的注释虽然通常较为准确，但是对代码的相似度度量方法有较高的要求。

深度学习的方法[7]则利用深度学习模型来学习代码和注释之间的映射关系，然后根据这种关系为新的代码生成注释。它可以处理复杂的代码结构和语义，生成的注释通常较为准确。深度学习模型需要大量的带注释的代码数据和大量计算资源进行训练。

在这三类方法中，基于深度学习的方法是最近几年的研究热点。随着深度学习技术的发展，这种方法的性能也在不断提高。目前，Wang 等人提出了 CodeT5[8]，这是一个统一的预训练编码器-解码器模型，用于代码理解和生成。它特别关注于代码中的标识符，以改善模型对代码结构和语义的理解。Wang 等人还提出了 CodeT5+[9]，这是 CodeT5 的增强版本，它通过更大的参数规模来提供更高的性能和更准确的代码理解能力。CodeT5+ 在多个代码相关的任务上展示了最先进的性能。Guo 等人提出了 UniXcoder[10]，这是一个统一的跨模态预训练模型，旨在通过深度学习和强化学习来掌握代码生成。它结合了编程语言和自然语言处理，以提高代码生成的质量和效率。Feng 等人提出了 CodeBERT[11]，这是一个针对编程和自然语言的预训练模型。CodeBERT 结合了编程语言的结构和自然语言的语义，以提升代码相关任务的性能。这些模型都在代码理解和生成领域发挥着重要作用，为软件开发和维护提供了强大的支持。

1.3 本文主要工作

本文的主要工作包括：

1. 利用 eclipseJDT 工具，设计了对市面上主要流行的 java library 进行分析的算法，以提取其中（1）处理异常的语句以及对应的 javadoc 注释（2）处理返回值的语句以及对应的 javadoc 注释（3）方法中其余语句以及对应的 javadoc 注释。并将这些语句-注释数据对作为数据集保存。

2. 使用数据集，使用 hugging face 平台以及其提供的 datasets，transformers 等 python 工具库对 codeT5+ 模型进行微调。

3. 对微调后的模型进行评估，通过对比找出最适合代码注释生成工具的模型。

图1-1为本文工作的整体框架

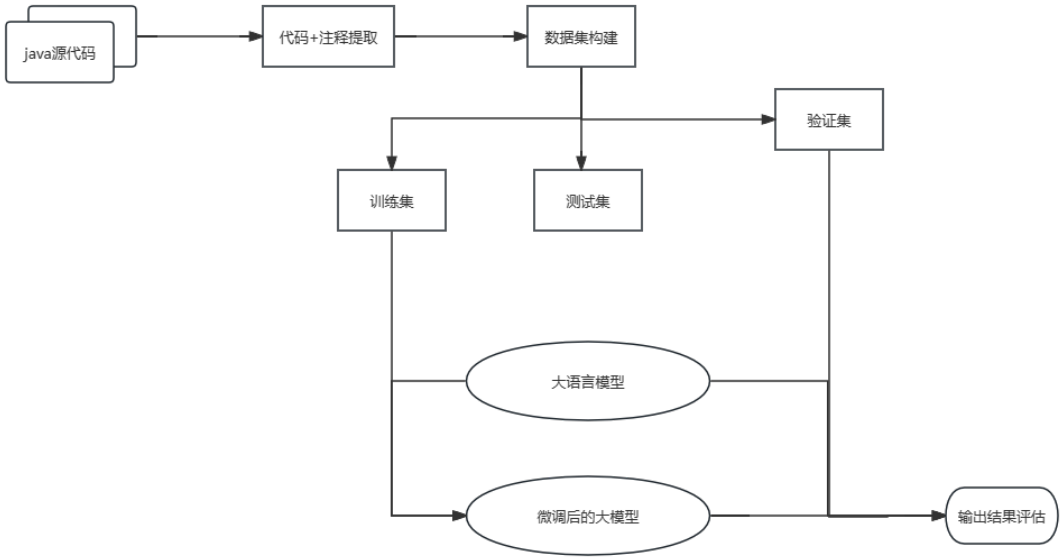

图 1-1　整体框架

1.4 本文组织结构

第一章：绪论部分，本章介绍了研究背景，总结了大语言模型以及代码注释生成工具的研究现状，展现了多种大模型在代码层面的功能和特征，选定了本文工具主要采用的对象 CodeT5+-220m-bimodal 模型和 java 语言。

第二章:相关理论和技术概述。本章首先介绍了 Eclipse 以及其提供的 Eclipse-

JDT 工具插件组成结构。之后介绍了 hugging face 平台以及其提供的 datasets 和 transformers 库的功能和任务。最后详细介绍 fine-tune，分析其在机器学习以及智能软件工程任务当中的应用，引出本文完成的新工作。

第三章：基于 EclipseJDT 的代码切割方法。本章详细介绍了切割代码以构建数据集的算法。首先介绍本文中切割特定代码要求的需求与意义，之后介绍采用的数据在实际应用场景中的功能，然后描述整个算法的整体框架，分语句的不同情况进行讨论。最后对所有切割出的数据进行整理，完成数据集的构建。

第四章：模型微调。本章详细介绍了微调模型的整个过程。首先介绍选择合适的大语言模型来完成代码注释生成工作的原因，简述模型的特征及功能。其次对用于微调模型的脚本进行简单介绍，简述其原理。最后，展现在输入第三章中准备的训练集后的微调结果。

第五章：实验与评估。本章详细地介绍了经过量化评估后各个大语言模型在代码注释生成任务的性能。同时通过实例展示直观地对微调前后的性能变化进行对比。

第六章：总结与展望。本章总结本文工作内容，对未来研究方向进行展望。

第二章　相关理论和技术概述

本章介绍本文提出的方法所涉及的背景知识。首先，介绍 Eclipse 其中的工具库 EclipseJDT 的相关组件，着重介绍其中的 AST 相关工具。然后介绍 hugging face 平台，对其在机器学习社区的贡献做出简要描述，并且介绍本文将要使用的 datasets 库和 transformers 库。最后介绍 fine-tune 微调技术的基本原理和基础技术。

2.1　Eclipse

Eclipse[1]是一个开源的、跨平台的集成开发环境（IDE）也是基于 Java 的可扩展开发平台，是目前最流行的 Java 语言开发工具之一。Eclipse 开发者提供的一系列工具库以及插件让该 IDE 具有强大的代码编排功能，大大提高了程序开发的效率。

2.1.1　Eclipse JDT

Eclipse JDT（Java Development Tools）[2]是 Eclipse 提供的一组工具插件，它实现了一个支持开发任何 Java 应用程序（包括 Eclipse 插件）的 Java IDE。它向 Eclipse 工作台添加了一系列的视图，编辑器，向导，构建器，以及代码合并和重构工具等。

其中，Eclipse JDT 中的 AST（Abstract Syntax Tree，抽象语法树）[3]是一个非常重要的组成部分。编译器在编译开发者的代码时，会将其解析为抽象语法树以便进一步分析，抽象语法树上的每个树节点代表一个代码元素。Eclipse JDT 中的 AST 工具提供了一组其中包括 AST、ASTParser、ASTNode、ASTVisitor 等类

1　https://www.eclipse.org

2　https://eclipse.dev/jdt/

3　https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/

来访问和操作 Java 源代码的 API。接下来简单介绍这些常用 AST 类的特点以及作用：

Class AST：抽象语法树（AST）是源代码的抽象语法结构的树状表示，树上的每个节点都表示源代码中的一种结构。

ASTParser：用于解析源代码，开发者可以以 Java Model 的形式或者以字符数组的形式来创建并设定 ASTParser。ASTParser 支持对以下四种内容的解析：

K_COMPILATION_UNIT：一个编译单元，可以输入 Java 文件。

K_STATEMENTS：Java statements，比如赋值语句，while 语句块，if 语句块等。此类型需要输入完整的 statements。

K_EXPRESSION: Java expressions。

K_CLASS_BODY_DECLARATIONS: Java class 中的元素。

ASTNode：AST 节点类，所有的节点类型都是 ASTNode 的子类。

ASTVisitor：ASTVisitor 是一个用于遍历抽象语法树的基类。可以通过继承 ASTVisitor 类来定义访问器。ASTVisitor 提供了一系列重载的 visit() 和 endvisit() 方法。这些重载的方法可以帮助开发者处理多种不同类型的节点。当访问节点时执行 visit() 方法，并决定是否继续访问子节点，结束访问后则使用 endvisit() 方法。

总的来说，Eclipse JDT 的 AST 提供了一套全面的工具，使得 Java 开发变得更加高效和便捷。

2.2 hugging face

Hugging Face[1]是一家美国公司，专门开发用于构建机器学习应用的工具。与 github 类似，它是一个流行的自然语言处理 (NLP) 模型库和社区，平台和用户提供了大量的预训练模型以及工具库等等，这让学习者和开发者都能方便快捷地获取符合自己需求任务的 NLP 模型，并跟随详细的教程进一步开发。Hugging Face 成立于 2016 年，由法国企业家克莱门特 • 德朗格（Clément Delangue）、朱利安 • 肖蒙（Julien Chaumond）和托马斯 • 沃尔夫（Thomas Wolf）创立。Hugging Face Hub 是一个集中式 Web 服务平台，用于托管以下内容：基于 Git 的代码仓

1 https://huggingface.co/

库，具有类似于 GitHub 的功能，包括项目的讨论（discussions）和拉取请求（pull requests）；具有基于 Git 版本控制的模型（models）；数据集（datasets），主要涵盖文本、图像和音频；Web 应用程序（“spaces”和“widgets”），用于机器学习应用的小规模演示。

Hugging Face 提供了大量的预训练模型，如 BERT、GPT-2 等，这些模型可以直接用于各种 NLP 任务，如文本分类、命名实体识别、情感分析等。同时它也是一个开放的社区，用户可以在这里分享自己的模型，也可以下载和使用其他人分享的模型。为了使更多人可以快速上手，Hugging Face 提供了一系列的工具和接口，使得用户可以快速地使用预训练模型进行 NLP 任务。例如，用户可以安装 Hugging Face 的 transformers 库。除了使用和下载模型，用户还可以在平台对预训练模型进行 fine-tuning，以适应特定任务或领域的需求。如果 Hugging Face 提供的现成模型无法满足需求，则可以通过继承 PreTrainedModel 和 PreTrainedTokenizer 类来创建自己的模型和分词器。

2.2.1 datasets

datasets[12]是一个 Python 库，它提供了一种简单易用的方式来加载和操作各种数据集，覆盖了机器学习和自然语言处理的多个领域。它包含了数百个预处理好的数据集，并允许用户自定义数据处理流程，使得实验设置标准化变得更加容易。Hugging Face 的 datasets 库支持以下 4 种数据格式的数据集，基本上覆盖了大部分数据格式：CSV & TSV，Text files，JSON & JSON Lines，Pickled DataFrames。datasets 库包含大量常用的数据集，如 GLUE、SQuAD、IMDB 等。此外，库还支持自定义数据源，方便用户导入自己的数据集。该库利用了 Apache Arrow 进行内存管理和序列化，能够快速地处理大规模数据，而且在多 GPU 环境中表现优越。

datasets 提供了一致且直观的 API，使数据读取、处理和分割变得轻松。例如使用 load_dataset 函数直接加载数据集，然后通过简单的操作进行切片、过滤或转换。通过 map、filter、concatenate 等函数，用户可以轻松地对数据集执行各种操作。这使得在不编写复杂自定义类的情况下实现数据预处理成为可能。库中每个数据集都有唯一的标识符和版本信息，确保实验的可重复性。

对于 NLP 研究人员和开发者，datasets 提供了一个统一的数据入口，便于比

较不同模型在标准数据集上的性能。

2.2.2 transformers

Hugging Face 的 Transformers 库[13]是一个开源项目，它提供了一系列接口和预训练模型，用于处理自然语言处理任务，如文本分类、信息抽取、问答系统等。Transformers 库是由 Hugging Face 团队开发的预训练模型库，这些模型可以直接用于各种 NLP 任务，如文本分类、命名实体识别、情感分析等。该库通过对底层 ML 框架（如 PyTorch、TensorFlow 和 JAX）进行抽象，简化了 Transformer 模型的实现，从而大大降低了 Transformer 模型训练或部署的复杂性。

Transformers 库可以通过直接使用 pipeline 去完成各种 NLP 任务，此外还提供了一系列的函数和接口，使得用户可以快速地使用预训练模型进行 NLP 任务。例如可以使用 from_pretrained() 函数来加载预训练模型。如果需要对预训练模型进行微调，可以使用 Trainer 类来对指定模型进行训练。Trainer 类中提供了一系列的函数，如 train()、evaluate() 等用于进行模型训练和评估。

继承 PreTrainedModel 类通常可以用于创建自己的模型，从而实现在自己的模型中定义模型的结构和前向传播过程。如果需要创建自己的分词器，可以在继承 PreTrainedTokenizer 之后自定义文本的编码和解码过程。

总的来说，Hugging Face 的 Transformers 库提供了一种简单而强大的方式来处理和操作数据集，使得用户可以更加高效地进行数据分析和模型训练。

2.3 fine-tune

Fine-tune（微调）[14]是一种常用的深度学习技术，它涉及到在特定领域或业务数据集上对预训练模型进行针对性优化，以提升其在特定任务上的性能。这种技术的应用涵盖了多个领域，从自然语言处理到计算机视觉，都可以通过 Fine-tuning 来提高模型的性能和适用性。

Fine-tune 的基本原理是利用已知的网络结构和已知的网络参数，修改 output 层为自己的层，微调最后一层前的若干层的参数，这样就有效利用了深度神经网络强大的泛化能力，又免去了设计复杂的模型以及耗时良久的训练，所以 fine tuning 是当数据量不足时的一个比较合适的选择。在大语言模型微调领域，有一

些值得关注的研究和论文。例如，LoRA（Low-Rank Adaptation）[15]是由 Edward Hu 等人提出的一种方法，它通过在预训练模型的权重矩阵中添加低秩矩阵来实现微调。这种方法可以显著减少微调所需的参数数量，同时保持模型性能。

另一个相关的研究是 Edward J. Hu 等人提出的 PEFT（Parameter-Efficient Fine-Tuning）[16]，它旨在减少微调大型语言模型时所需的资源。PEFT 方法包括 LoRA 在内的多种技术，如 Adapters 和 Prefix Tuning，这些技术都是为了在避免牺牲模型性能的前提下，减少微调过程中的计算成本。

Fine-tuning 的原理包括以下两个步骤：

1. 初始化：使用预训练模型的参数作为新任务模型的初始参数。

2. 微调：在新任务的数据上继续训练模型，对模型参数进行微调。

在 fine-tuning 过程中，首先保留预训练模型的大部分参数（也就是模型的“知识”），然后在新的数据集上对这些参数进行微小的调整。这样就可以利用预训练模型在大规模数据集上学习到的信息，同时避免了过拟合的问题。

具体来说，微调过程通常包括以下几个步骤：

1. 选择预训练模型：预训练模型是在大规模数据集上训练的深度神经网络模型。这个模型通常包含了大量的层，每一层都学习到了一些从输入数据中提取特征的能力。

2. 创建新的输出层：由于预训练模型的输出层是针对原始任务设计的，因此我们需要替换它以适应新的任务。新的输出层的大小通常与新任务的类别数相同。

3. 冻结预训练模型的参数：在训练新的输出层时，我们通常会冻结预训练模型的参数。这意味着在这个阶段，我们只更新新的输出层的参数，而保持预训练模型的其他参数不变。

4. 微调模型的参数：一旦新的输出层训练好了，我们就可以开始微调模型的参数。在这个阶段，我们会对模型的所有参数（包括预训练模型的参数和新的输出层的参数）进行更新。

通过这种方式，fine-tuning 可以有效地将预训练模型的知识迁移到新的任务上，从而提高模型在新任务上的性能。

Fine-tuning 的目的是将通用的大型语言模型转变为更具专业性的工具，以满足特定领域或应用的需求。通过微调，用户可以根据自身特有的数据集和要求，

使模型更好地适应特定的业务场景。要在个性化的服务中使用大模型的能力，此时针对每个用户的数据，训练一个轻量级的微调模型性价比更高。此外当数据因为保密性等安全要求不能传递给第三方大模型服务时，搭建自己的大模型非常必要。

总的来说，Fine-tuning 是一种在深度学习中常用的技术，它可以帮助我们更好地利用预训练模型，提高模型在特定任务上的性能。无论是在自然语言处理还是在计算机视觉领域，Fine-tuning 都是一个非常重要的步骤。

第三章　基于 EclipseJDT 的代码切割方法

本文的研究目标是对收集到的 java libraries 进行代码切割，要求切割出符合要求且格式规范的代码-注释对。本文使用 java 语言，借助 EclipseJDT 工具，通过对 java 方法中的 AST 树进行分析，从而提取代码中 return 语句，异常语句等等并与对应 javadoc 标签注释配对，完成规范形式数据集的构建。本章围绕基于 EclipseJDT 的代码切割返回发展开，首先给出方法的整体框架，然后对不同目标语句的情况下的策略进行详细介绍。

3.1　问题分析

目前在 java 编程中常常遇到以下问题：当阅读代码时无法直观地了解描述异常的具体类型和处理方式，或者是抛出异常的原因；当遇到具有返回值的语句时，找出处理返回值的整个流程需要耗费大量的时间和精力，并且在理解流程逻辑时也常常伴随困难，这就导致了编程工作效率的降低。

同时，javadoc 注释[1]作为一种业界广泛使用的注释工具，其核心功能在于：只要根据一定的代码注释书写规范来编写注释，开发者就可以用 javadoc 工具来生成有关源代码各 API 的 html 帮助文档。而对于单个 java 方法的 javadoc 注释，可以通过标签等方式来对特定元素进行说明。比如，@author 后描述作者信息，@version 后描述版本号信息等等，而 @throw 和 @return 标签分别描述的是方法抛出的异常和方法返回值信息。这两种标签作为最常用也是代码中最关键的功能注释，如果有自动化工具生成，会让代码理解工作事半功倍。

针对这些问题，有必要对 throw 语句进行注释，以帮助开发者理解在什么情况下会抛出异常，以及这个异常表示什么。try-catch 语句块的相关注释可以提示代码可能会遇到的异常情况，以及这些异常是如何被处理的。同时，了解返回值在整个 java 方法中的处理流程也可以帮助开发者快速了解整个方法的运行逻辑。

1　https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html

最后，对于除开以上的剩下代码语句，也仍然需要提取其数据，以保证在面对完整代码方法的情况下仍然可以精准给出注释。

因此，本文对以上几种常见的语句以及注释进行采样，在保证准确率的同时，尽可能覆盖这些常见难题以达到更好的效果。

3.2 方法框架

本节将对基于 EclipseJDT 的代码切割方法整体框架进行描述。图3-1流程图展示了算法的整体流程: 首先寻找适合的 java libraries 作为等待处理的数据集。之后对该数据集运行此方法：方法的输入为含有 java 文件的文件夹。首先对该文件夹下的所有文件进行遍历，当遇到 java 文件后，对该文件进行如下动作：

步骤 1：分析代码，这个方法首先读取指定的 Java 文件，并将其内容转换为一个字符串。然后，它使用 Eclipse JDT Core 的 ASTParser 来解析这个字符串，生成一个抽象语法树（AST）。接着，它创建一个新的 ASTVisitor，并让 AST 接受该 ASTVisitor。

步骤 2：首先获取方法的名称和方法体。然后，它遍历方法体中的所有语句，ASTVisitor 会访问每个 Statement 节点。如果发现 try 语句，就进一步处理，转到步骤 3。如果发现 try-catch 语句，则转到步骤 4。如果发现 return 语句则转到步骤 5。

步骤 3：首先获取包含当前 throw 语句的方法声明。然后，它检查这个方法声明的 Javadoc 注释，如果存在 @throws 标签，就将 throw 语句和对应的 @throws 注释写入到输出文件中。之后转到步骤 6。

步骤 4：遍历 try 语句的所有 catch 子句，并检查方法的 Javadoc 注释中是否存在 @throws 标签，如果存在，就将 catch 子句和对应的 @throws 注释写入到输出文件中。之后转到步骤 6。

步骤 5：检查该节点所在的方法是否有 @return 标签。如果有，它会将该方法的签名、返回值处理代码、以及 @return 标签的描述写入到输出文件中。之后转到步骤 6.

步骤 6; 对已提取代码和注释的 java 方法，对其剩余的语句和注释进行配对。

步骤 7：集合所有得到的代码-注释数据对，随机打乱，并按 8：1：1 的比例

划分训练集，测试集和验证集。

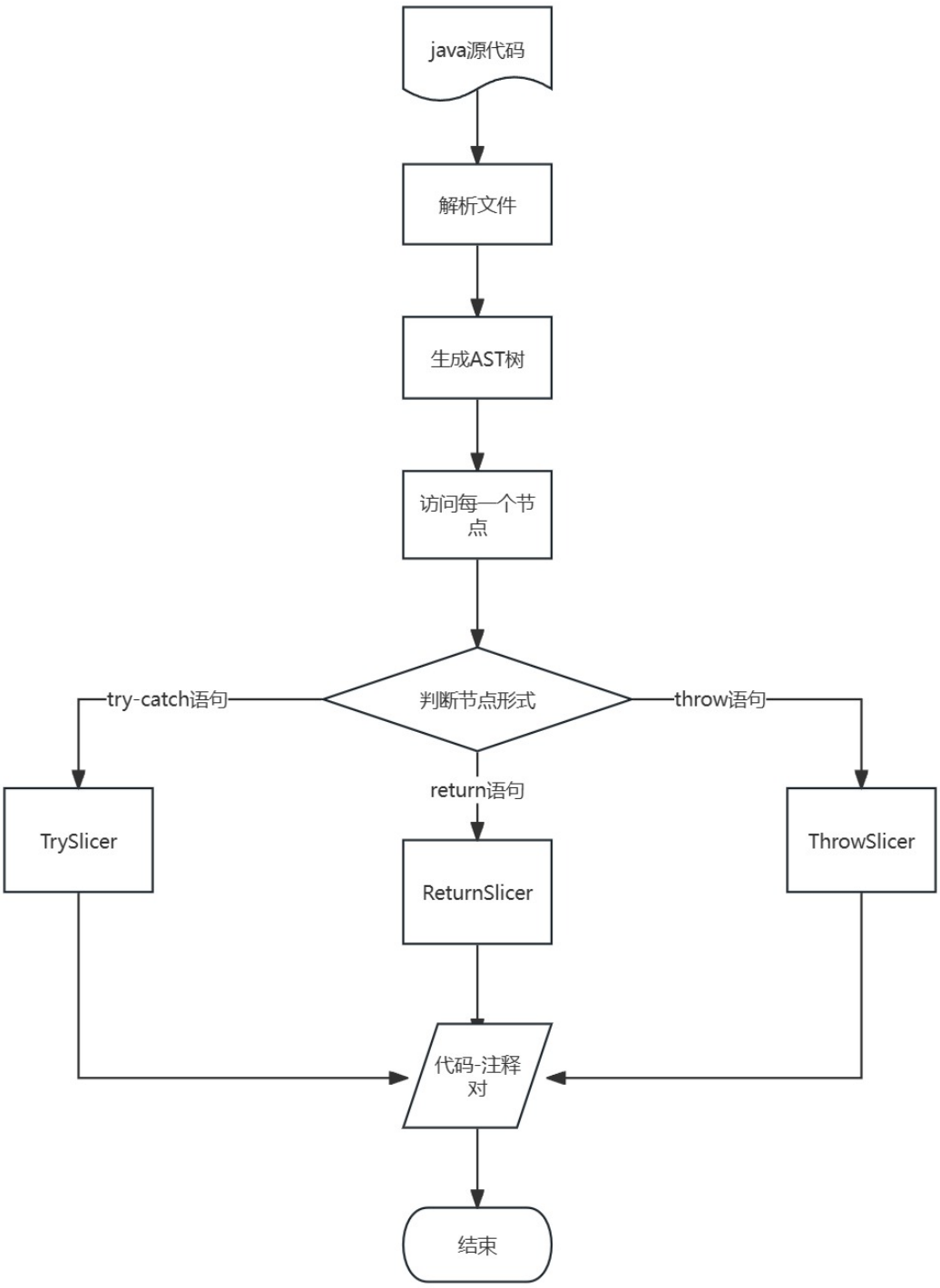

图 3-1　算法流程

3.3 收集数据

为了尽可能覆盖 java 开发者可能遇到的编程场景，以下对日常项目中使用频率较高的 java 第三方库进行收集。

1. 日志库，作为服务器端应用最重要的组成之一，几乎每个项目都需要用到它，日志被存放在可以阅读应用程序当前运行时情况的地方。尽管 JDK 附带了自己的日志库，但是许多第三方库功能更加强大，例如 Log4j，SLF4j 和 LogBack。

2.JSON 解析库，在当今的 Web 服务和物联网领域，为了在独立平台间传输信息，相对于之前的 XML，JSON 已成为将信息从客户端传送到服务器的首选协议。JDK 本身并不自带 JSON 库，有许多优秀的第三方库允许开发者解析和创建 JSON 消息，如 Jackson 和 Gson。

3. 单元测试库，单元测试可以帮助开发者在早期发现和修复代码中的错误，从而提高代码的质量和稳定性。通过自动化的单元测试，开发者可以快速地验证代码的正确性，而无需手动进行繁琐的测试工作，从而提升开发效率。在持续集成和持续部署的环境中，单元测试也是非常重要的一环，它可以在每次代码提交时自动运行，确保代码的质量和稳定性。常用的单元测试库包括 JUnit，Mockito 和 PowerMock。

4. 通用库，Java 开发人员常常使用通用第三方库，比如 Apache Commons[1]和 Google Guava[2]。因为它们简化了很多功能。

5. 网络库，如果需要编写需要执行底层网络任务的应用程序，Java 中有许多优秀的网络库，如 Netty, Apache MINA, Apache HttpClient 等。网络库通常支持多种网络协议（如 HTTP, FTP, SMTP 等）和服务（如 Web 服务，邮件服务，文件传输服务等），可以满足各种网络编程的需求。它们也提供异步和非阻塞的网络通信，可以帮助开发者编写出高性能的网络应用。网络库通常还提供了一些网络安全相关的功能，如 SSL/TLS 加密，身份验证等，可以帮助保护网络通信的安全。这些库提供了丰富的功能和良好的用户体验，可以帮助开发者更有效地进行网络编程。

6. 其他，如 HTTP 库，XML 解析库，Excel 库，字节码库等等。

1 https://commons.apache.org/

2 https://github.com/google/guava

3.4 代码切割算法

3.4.1 AST 树分析算法

要想对 java 文件中的方法的每个语句进行分析，最直接的方法则是使用 ElcipseJDT 工具对方法进行解析，解析的结果为抽象语法树 AST，通过遍历语法树的每个节点，我们可以通过检查每个节点是否符合目标，从而实施下一步分析措施。

算法1详细地描述了 AST 树分析算法，算法需要含有 java 源代码的文件路径，以及返回文件的存储路径作为输入，然后分析该 java 源代码的 AST 树。这里仅描述对于一个 java 文件的分析过程，多份 java 文件只需要重复此过程即可。

Algorithm 1 AST 分析算法

输入: java 文件路径 $filePath$，输出文件路径 $outputFilePath$
输出: *true*

```
1:  source ←readFileToString(filePath.toString())
2:  parser ←ASTParser.newParser(AST.JLS3)
3:  parser.setSource(source.toCharArray())
4:  parser.setKind(ASTParser.K_COMPILATION_UNIT)
5:  parser.setResolveBindings(true)
6:  cu ←parser.createAST(null)
7:  cu.accept(newASTVisitor(){
8:  procedure visit(MethodDeclaration method)
9:      methodName ←method.getName().getIdentifier()
10:     body ←method.getBody()
11:     if body is not null then
12:         for each statement stmt in body do
13:             if stmt is an instance of ThrowStatement then
14:                 Throw 语句及注释提取算法
15:             end if
16:             if stmt is an instance of TryStatement then
17:                 try-catch 异常处理语句及注释提取算法
18:             end if
19:             if stmt is an instance of ReturnStatement then
20:                 返回值语句及注释提取算法
21:             end if
22:         end for
23:     end if
24:     return true
25: end procedure})
```

具体地，代码首先读取源代码文件的内容，将其转换为字符串形式（第 2 行），然后创建一个新的 ASTParser 对象，该对象由 EclipseJDT 提供，可以用于解析 Java 源代码，同时设定 AST 的版本（第 3 行），此后设置解析器的源代码（第 4 行），并设置解析器的类型为 K_COMPILATION_UNIT, 这意味着解析器将把源代码视为一个完整的 java 单元（第 5 行）。启用绑定解析，这将允许解析器识别源代码中的变量和方法。（第 6 行）。使用解析器创建一个抽象语法树（AST），并记为 cu（第 7 行）。之后让让一个新的 ASTVisitor 对象访问抽象语法树。ASTVisitor 是一个可以遍历并处理 AST 中各种类型节点的对象（第 8 行）。（第 9 行）定义一个名为 VISIT 的过程，它接受一个 MethodDeclaration 对象作为参数。在 Java 中，MethodDeclaration 是 org.walkmod.javalang.ast.body 包的一部分。它代表了 Java 源代码中的一个方法声明。MethodDeclaration 对象包含了方法的所有信息，如方法名、参数列表、返回类型、访问修饰符等。这个过程将在 ASTVisitor 访问每个方法声明节点时被调用。然后获取当前方法的名称（第 10 行）获取当前方法的主体（第 11 行）并检查方法主体是否存在（第 12 行）。若存在，则遍历方法主体中的每个语句（第 13 行）。如果当前语句是一个抛出异常的语句，那么执行 ThrowSlicer 算法（第 14，15，16 行）. 如果当前语句是一个 try-catch 语句，那么执行 TrySlicer 算法 (第 17，18，19 行)。如果当前语句是一个 return 语句，那么执行 ReturnSlicer 算法 (第 20，21，22 行）。当算法执行的最后一步，说明此 java 文件中的所有方法都已解析完毕，此时结束进程。

3.4.2 throw 异常语句

算法2展示了处理 throw 异常语句的流程。总体来说，方法在读取 Java 文件内容，使用 ASTParser 创建抽象语法树（AST），然后遍历 AST 以查找到 ThrowStatement 节点后。对于每个 ThrowStatement 节点，找到包含该节点的方法声明，并检查该方法的 Javadoc 中是否有 @throws 标签。如果有，将抛出的异常信息和相关的 Javadoc 标签内容写入到输出文件中。

以下是当遇到 throw 语句时调用的切割语句方式：ThrowSlicer 接受一个 ThrowStatement 对象作为参数（第 1 行）。（第 2 行）尝试在 outputFilePath 指定的文件上打开 FileWriter、BufferedWriter 和 PrintWriter，以追加模式写入数据。这步是为了之后提取出的代码写入到指定的 txt 文件当中做准备。然后调用 ge-

tEnclosingMethod 函数，获取包含当前节点的方法声明，并将结果赋值给 method（第 3 行）。

Algorithm 2 throw 语句及注释提取算法

输入: *ThrowStatement* 类型节点 *node*
输出: *super.visit*(*node*)

```
1:  try Open FileWriter, BufferedWriter, and PrintWriteron outputFilePath in
    append mode
2:  method ← getEnclosingMethod(node)
3:  if methodisnotnull then
4:      javadoc ← method.getJavadoc()
5:      if javadoc is not null then
6:          tags ← javadoc.tags()
7:          for each tag in tags do
8:              if tag.getTagName() is not null and equals "@throws" then
9:                  Print "throw " + node.getExpression() to output file
10:                 Print tag.toString() to output file
11:                 Print a blank line to output file
12:             end if
13:         end for
14:     end if
15: end if
16: Catch IOExceptione
17: return super.visit(node)
```

getEnclosingMethod 函数的内容如下：这个函数接受一个 ASTNode 对象作为参数，然后通过循环检查该节点及其所有父节点，直到找到一个类型为 MethodDeclaration 的节点。如果找到了这样的节点，就返回它；否则，返回 null。这个函数的目的是找到包含给定节点的方法声明。赋值之后，检查 method 是否为 null。如果不是，执行后续的代码块（第 4 行）。从 method 中获取 Java 文档注释，并将结果赋值给 javadoc（第 5 行）。检查 javadoc 是否为 null。如果不是，执行后续的代码块（第 6 行）。从 javadoc 中获取所有的标签，并将结果赋值给 tags（第 7 行）。（第 8 行）遍历 tags 中的每一个标签，检查当前标签的名称是否不为 null 且等于"@throws"。如果是，执行后续的代码块（第 9 行）。将 throw 语句打印到输出文件中（第 10 行）。将当前标签的字符串表示形式打印到输出文件中（第 11 行）。最后在输出文件中打印一个空行，以分隔不同的输出（第 12 行）。（第 17 行）捕获可能抛出的 IOException，（第 18 行）super.visit(node) 是一个方法调用，它调用了当前对象的父类（或超类）中的 visit 方法。这里的 super 关键字是用来引用父类的。node 是传递给 visit 方法的参数，它是一个 ThrowStatement

对象，代表了 Java 源代码中的一个抛出异常的语句。visit 方法的返回值是一个布尔值，它通常表示遍历（访问）AST（抽象语法树）的过程是否应该继续。如果 visit 方法返回 true，那么遍历过程将继续访问 AST 中的其他节点；如果返回 false，那么遍历过程将停止。在本算法中，super.visit(node) 的调用发生在 try 块的最后，作为 visit 方法的返回值。这意味着，无论 try 块中的代码是否成功执行，visit 方法都将返回 super.visit(node) 的结果。

3.4.3 try-catch 异常语句和异常处理语句

Algorithm 3 try-catch 异常处理语句及注释提取算法

输入： *MethodDeclaration* 对象 *method*
输出： *true*

```
1:  methodName ← method.getName().getIdentifier()
2:  body← method.getBody()
3:  if body is not null then
4:      for each object obj in body.statements() do
5:          stmt← (Statement)obj
6:          if stmt is an instance of TryStatement then
7:              tryStmt ← (TryStatement)stmt
8:              catchClauses ← tryStmt.catchClauses()
9:              for each objectccObjin catchClauses do
10:                 cc← (CatchClause)ccObj
11:                 javadoc ←method.getJavadoc()
12:                 if javadoc is not null then
13:                     tags ← javadoc.tags()
14:                     for each TagElementtag in tags do
15:                         if tag.getTagName()is not null and equals ”@throws” then
16:                             for each statement in cc.getBody().statements() do
17:                                 writeToFile(statement.toString(),
    outputFilePath)
18:                             end for
19:                             writeToFile(tag.toString(), outputFilePath)
20:                             writeToFile("", outputFilePath)
21:                         end if
22:                     end for
23:                 end if
24:             end for
25:         end if
26:     end for
27: end if
28: return true
```

算法3是对 try-catch 异常语句的处理方式，简单来说，如果方法中的 try 块

包含 catch 子句，并且方法有 Javadoc 注释且包含 @throws 标签，则程序将 catch 中处理异常的代码，异常以及 @throw 注释写入指定的输出文件。

该算法接受一个 MethodDeclaration 对象作为参数。之后获取当前方法的名称（第 2 行），获取当前方法的主体（第 3 行），获取当前方法的主体（第 4 行）若存在，则遍历方法主体中的每个语句（第 5 行）。之后获取该语句（第 6 行）并检查当前语句是否是一个 try 语句（第 7 行）。如果是，则获取 try 语句的所有 catch 子句（第 8，9 行）。之后遍历所有的 catch 子句并获取子句（第 10，11 行）然后获取当前方法的 Java 文档注释（第 12 行），检查 Java 文档注释是否存在（第 13 行），若存在则获取 Java 文档注释的所有标签（第 14 行）。遍历所有的标签（第 15 行）检查当前标签的名称是否不为 null 且等于”@throws”（第 16 行）。若不为 null 且为 @throws 标签，则将当前 catch 所有子语句的字符串表示形式写入到输出文件中（第 17，18 行）。最后将将当前标签的字符串表示形式写入到输出文件中（第 20 行），且每组结果用空行隔开（第 21 行）。算法结束时，返回 true，表示 VISIT 过程已成功完成（第 29 行）。

3.4.4 返回值语句和处理返回值语句

本小节主要解释切割返回值以及处理返回值语句的算法，其下（算法4）为第一部分，该算法接受一个参数为 ReturnStatement 类型的节点作为输入（第 1 行），然后调用 getEnclosingMethod 函数获取包含该返回语句的方法声明（第 2 行）。如果找到了方法声明，则继续执行（第 3 行）并且获取该方法声明的 Java 文档注释（第 4 行）。如果 Java 文档注释存在且包含返回标签 @return，则继续执行（第 5 行）。下一步开始一个异常处理块（第 6 行），尝试在追加模式下打开指向 outputFilePath 的文件写入器、缓冲写入器和打印写入器（第 7 行），并将文件路径输出到文件中（第 8 行）。（第 9 行）获取方法的修饰符字符串（如 public, static），以及获取方法的返回类型（第 10 行），方法的名称（第 11 行），以及方法参数的字符串表示（第 12 行）, 以上这些是为了将方法签名输出到文件中（第 13 行）。接下来则是处理返回值以及处理返回值语句的部分：首先获取方法的返回语句中的返回表达式（第 14 行），如果返回表达式不为空，则继续（第 15 行）获取与返回表达式相关的语句列表（第 16 行）此处用到了 getRelatedStatements 函数，该函数的作用是获取返回值对应的所有处理该返回值的语句，在之后会详细讲解

该函数的流程。之后遍历所有相关语句（第 17 行）把每个相关语句的字符串表示输出到文件中（第 18 行）之后结束遍历。接下来则处理代码语句对应的 javadoc 注释部分：将 Java 文档注释中的返回描述输出到文件中（第 21 行），并且输出一个空行，用于分隔内容（第 22 行）。以上内容结束后，捕获并处理 IOException 异常（第 23 行），并打印异常的堆栈跟踪（第 24 行），最后，返回调用父类 visit 方法的结果（第 28 行），结束进程。上述总体代码中的 getRelatedStatements 函数，

Algorithm 4 返回值语句及注释提取算法

输入： *ReturnStatement* 对象 *node*
输出： *super.visit*(*node*)

```
1:  procedure VISIT(ReturnStatementnode)
2:      method← getEnclosingMethod(node)
3:      if method is not null then
4:          javadoc ←method.getJavadoc()
5:          if javadoc is notnulland hasReturnTag(javadoc) then
6:              Try
7:              Open FileWriter,BufferedWriter, and PrintWriter on outputFilePath in append mode
8:              Print filePath to output
9:              modifiers← getModifiersString(method.modifiers())
10:             returnType ← method.getReturnType()
11:             methodName← method.getName().getIdentifier()
12:             parameters ← getParametersString(method.parameters())
13:             Print method signature: modifiers + " " + returnType + " " + methodName+ "(" + parameters+ ")" to output
14:             returnExpression ← node.getExpression()
15:             if returnExpression is not null then
16:                 relatedStatements← getRelatedStatements
17:                 for each statementin relatedStatements do
18:                     Printstatement.toString() to output
19:                 end for
20:             end if
21:             Print getReturnDescription(javadoc) to output
22:             Print a blank line to output
23:             Catch IOExceptione
24:             Print stack trace of e
25:             End Try
26:         end if
27:     end if
28:     return super.visit(node)
29: end procedure
```

将在以下算法5作详细叙述：首先创建一个空列表 relatedStatements，用于存储与返回表达式相关的语句。（第 2 行）同时创建一个空集合 varNames，用于存储变

Algorithm 5 返回值相关语句提取算法

输入: 返回值语句 *returnExpression*，方法体 *methodBody*
输出: 相关语句 *relatedStatements*

```
1: relatedStatements←empty list
2: varNames ←empty set
3: outputtedStatements ←empty set
4: if isSimpleName(returnExpression) then
5:     add returnExpression to varNames
6: else if isInfixExpression(returnExpression) then
7:     if isSimpleName(leftOperand(returnExpression)) then
8:         add leftOperandtovarNames
9:     end if
10:    if isSimpleName(rightOperand(returnExpression)) then
11:        addrightOperand to varNames
12:    end if
13: else if isMethodInvocation(returnExpression) then
14:    for all argument in arguments(returnExpression) do
15:        if isSimpleName(argument) then
16:            add identifier of argument to varNames
17:        end if
18:    end for
19: end if
20: newVarFound←object with value false
21: repeat
22:    newVarFound.value←false
23:    newStatements ←empty list
24:    for all node in methodBody do
25:        if isSimpleName(node) and identifier of node is in varNames then
26:            parent ←parent of node
27:            while not isStatement(parent) do
28:                parent ←parent ofparent
29:            end while
30:            if not in relatedStatements and not in outputtedStatements then
31:                add parent to newStatements
32:                add parent to outputtedStatements
33:                newVarFound.value←true
34:            end if
35:        end if
36:    end for
37:    for all statement in newStatements do
38:        for all node in statement do
39:            if isSimpleName(node) then
40:                add identifier of node to varNames
41:            end if
42:        end for
43:    end for
44:    sort newStatements by their order in methodBody
45:    add all newStatements to relatedStatements
46: until not newVarFound.value
47: sort relatedStatements by their order in methodBody
48: return relatedStatements
```

量名。（第 3 行），再创建一个空集合 outputtedStatements，用于存储已经输出过的语句（第 4 行）。检查 returnExpression 是否是一个简单的名字（变量名）（第 5 行），如果是，将 returnExpression 添加到 varNames 集合中（第 6 行）。否则，检查 returnExpression 是否是一个中缀表达式（例如 a + b）（第 7 行）。如果是，检查中缀表达式的左操作数是否是一个简单的单变量（第 8 行）。如果是，则将左操作数添加到 varNames 集合中（第 9 行）并结束以上对左操作数的审查。然后开始检查中缀表达式的右操作数是否是一个简单的名字（第 11 行）。如果是，将右操作数添加到 varNames 集合中（第 12 行）。否则，检查 returnExpression 是否是一个方法调用（14 行）。之后的操作同上：如果是，遍历方法调用中的所有参数（第 15 行）。检查每个参数是否是一个简单的单变量（第 16 行）；如果是，将参数的标识符添加到 varNames 集合中（第 17 行）。结束以上判断和循环之后，创建一个名为 newVarFound 的对象，并将其值初始化为 false）（第 21 行）并开始一个 repeat 循环（第 22 行）。在每次循环开始时，将 newVarFound 的值设置为 false（第 23 行）。创建一个空列表 newStatements，用于存储新发现的语句（第 24 行）。然后遍历 methodBody 中的所有节点（第 25 行）。如果节点是一个简单的名字，并且其标识符在 varNames 集合中（第 26 行），则获取该节点的父节点（第 27 行）。如果父节点不是一个语句（第 28 行），就继续向上查找，直到找到一个语句（第 29 行）。如果这个父语句既不在 relatedStatements 列表中，也不在 outputtedStatements 集合中（第 31 行）。那么将这个父语句同时添加到这个列表和集合当中（第 32,33 行), 并且将 newVarFound 的值设置为 true（第 34 行）。在结束以上 if 语句和 for 循环后，遍历 newStatements 列表中的所有语句（第 38 行），再遍历每个语句中的所有节点（第 39 行），如果节点是一个简单的单变量，就将节点的标识符添加到 varNames 集合中（第 41，42 行）。最后对于所有的语句，根据它们在 methodBody 中的顺序对 newStatements 列表进行排序（第 45 行）然后将所有 newStatements 列表中的语句添加到 relatedStatements 列表中（第 46 行）。重复以上步骤，直到 newVarFound 的值为 false（第 47 行）。这一步是为了保证和 return 相关的所有变量都被记录，以及含有处理这些变量的语句也被记录。在根据它们在 methodBody 中的顺序对 relatedStatements 列表进行排序（第 48 行）。此处是保证所有的语句会按照原来的函数中的语句顺序输出。返回返回 relatedStatements 列表之后进程结束。该列表代表着函数中所有与 return 值相关

的语句。

3.4.5 其余语句

假设全集为“函数语句集合 F”，现在集合 Tr（throw 语句集合）和集合 Tc（try-catch 语句集合）以及集合 Re（return 语句集合）为 F 集合的子集，那么其余语句的集合可以用公式 3-1 表示：

$$F \setminus (Tr \cup Tc \cup Re) \tag{3-1}$$

因此，在之前对 throw 语句，try-catch 语句以及 return 语句处理的基础上，对提取出的语句取反即可、代码实现主体与以上算法相似度较高，此处不再更多叙述。对于 javadoc 注释的处理同理。

3.4.6 数据清洗

数据清洗是数据预处理的重要环节，它涉及检查、处理和修正数据集中的错误或不一致的记录。数据清洗的目的是确保数据的准确性、完整性、一致性和可靠性，从而提高数据分析的质量和准确性。

通过移除重复数据、纠正错误和填补缺失值，数据清洗有助于提升数据的质量和准确性，同时可以减少分析过程中的误差，使得结果更加可靠。此外，数据清洗有助于避免因数据错误或不一致性而导致的错误决策，从而降低决策风险。

由于 javadoc 的格式规范，在提取的注释数据当中，常常会有诸如”@code”,”@link”,” ”,” ”,”<code>”,”</code>” 等等原本用于 html 文档构建的标签。但是这些标签对于代码注释生成并没有积极作用，相反，它们的存在会给大模型的数据微调带来相当多的噪点，从而降低生成结果的可读性。因此，在最后数据集的构建中，我们需要把这些无用字段清除。

在之前代码片段切割过程中，所有的数据都以以下3-2格式保存到 txt 文件中：

```
public static void HelloWorld(){
    System.out.println("HelloWorld!");
}
print HelloWorld!
```

图 3-2　txt 文件中格式

每组数据的最后一行固定为注释行，除此之外皆为代码行。每组数据之间在 txt 文件中以一行空行间隔最后提取出一共 174708 组代码-注释对数据，按照 8：1：1 的比例划分训练集，测试集和验证集。得到训练数据 139766 对，测试集 17472 对，验证集 17470 对。

3.5　本章小结

本章主要介绍了通过 EclipseJDT 工具对 java 代码片段的各种切割算法，以构建后续用于微调的数据。集。首先介绍了算法的总体结构，然后介绍了相关数据的收集，对市面上广泛运用的 java libraries 进行采集。最后介绍了切割 throw 语句，try-catch 语句，return 语句以及其余语句及其各自对应的 javadoc 注释的算法。至此本文涉及到的算法部分全部介绍完毕，下一章介绍实验相关内容。

第四章　模型微调

本章的研究目标是选择合适的大模型进行微调，要求在代码注释生成的准确度和可读性较于未微调之前有一定的提升。本文给出一种基于大模型的微调方案，以之前准备好的训练集为输入，以 hugging face 提供的 datasets 和 transformers 库为辅助工具，在 pytorch 环境下进行模型微调。本章围绕整个微调过程展开，首先给出选择各种模型的原因和其模型特点，其次介绍微调过程中使用的脚本工具，最后展示微调结果。

4.1　模型选择

由于目前需要完成代码注释生成任务，在选择基础的预训练大语言模型时，也就要求其在代码-文本双模态数据上有良好的预训练。同时要求选择的大语言模型在 code-to-NL 下游任务上有良好的表现，比如代码总结任务。再者，由于我们在构建数据集时使用的是 java 代码，最后微调结果也希望能面向 java 编程工作进行代码注释的自动生成，因此要求大语言模型在 code-to-NL 任务中支持 java 语言。并且选择的模型需要是近几年发布的表现较好的大模型，且大语言模型之间的规模差距不能过大。基于以上几点基础需求，本文选取了以下几种大语言模型作为微调基础。

CodeT5: CodeT5 是一个统一的预训练 encoder-decoder Transformer 模型，它可以更好地利用从开发人员分配的标识符传达的代码语义。模型采用统一的框架来无缝支持代码理解和生成任务，并允许多任务学习。

此外，模型还利用用户编写的代码注释和双峰双生成任务进行更好的 NL-PL 对齐。实验表明，CodeT5 在理解代码缺陷检测和克隆检测等任务以及包括 PL-NL、NL-PL 和 PL-PL 在内的各个方向的生成任务方面明显优于先前的方法。

CodeT5 是一种新的编程语言预训练编码器-解码器模型，它在 8 种编程语言 (Python, Java, JavaScript, PHP, Ruby, Go, C 和 c#) 上进行了预训练。总的来说，它

在代码智能基准- CodeXGLUE 中的 14 个子任务上实现了先进的结果。

CodeT5+：来自 Salesforce 的研究者提出了 CodeT5+ 模型，CodeT5+ 采用编码器-解码器架构，可以以编码器-only、解码器-only 或编码器-解码器模式运行，可以用于广泛的代码理解和代码生成任务。在各种代码相关任务上，CodeT5+ 的性能达到了先进水平，例如代码生成和补全、数学编程以及文本到代码检索任务。

codeT5+ 的作者实现了一系列 CodeT5+ 模型，模型大小从 220M 到 16B 不等。此处选择了 codeT5+-220m-bimodal 的规格，该模型在单峰和双峰数据上通过两个阶段进行预训练。该模型可用于零采样方式的代码总结和代码检索，也可用于带微调的代码生成。

CodeBERT：CodeBERT 是一个由哈尔滨工业大学、中山大学和微软亚洲研究院共同开发的预训练模型，专门用于处理编程语言（PL）和自然语言（NL）。这个模型是 BERT 的扩展，BERT 是一种广泛用于自然语言处理任务的模型。CodeBERT 的创新之处在于它能够理解和生成与代码相关的自然语言文本，以及理解编程语言本身。

CodeBERT 采用了多层双向 Transformer 作为其核心架构，这与 BERT 和 RoBERTa 模型相似。具体来说，CodeBERT 的架构包括 12 个层，每层有 12 个自注意力头，每个头的大小为 64，隐藏维度为 768，前馈层的内部隐藏层大小为 3072，总共有大约 1.25 亿的模型参数。

CodeBERT 在预训练上使用了双模态数据，即自然语言-代码对平行数据，以及大量单模态代码数据。双模态数据点是具备函数级自然语言文档的代码，而单模态数据则是不具备平行自然语言文本的代码。这使它能够支持多种下游 NL-PL 应用，例如自然语言代码搜索、代码文档生成等。它可以捕捉自然语言和编程语言之间的语义连接，并输出可广泛支持 NL-PL 理解任务和生成任务的通用表示。

在实验中，CodeBERT 模型在自然语言代码搜索和代码文档生成两项任务上均取得在当时的最优性能。此外，研究者还构建了 NL-PL 探测数据集，并在 zero-shot 设置中测试了 CodeBERT，即不对 CodeBERT 进行调参。测试结果表明，CodeBERT 的性能在当时持续优于仅基于自然语言的预训练模型 RoBERTa。

UniXcoder: UniXcoder 是一个统一的跨模态预训练模型，专为编程语言设计。

它旨在支持与代码相关的理解和生成任务，通过利用多模态内容如抽象语法树（AST）和代码注释来增强代码表示。

UniXcoder 采用了一个统一的编码器-解码器框架，但与传统的编码器-解码器模型不同，它针对自回归任务（如代码补全）进行了优化，特别是在需要仅解码器方式进行有效推理的场景中。为了控制模型的行为，UniXcoder 利用带有前缀适配器的掩码注意力矩阵，并通过对比学习来学习代码片段的表示，然后使用跨模态生成任务在编程语言之间对齐表示。

UniXcoder 的输入包括代码文本、代码注释和 AST。为了并行编码以树形结构表示的 AST，研究者提出了一种一对一映射方法，将 AST 转换为保留树中所有结构信息的序列结构。输出方面，UniXcoder 能够为每个 token 生成上下文相关的表示，这些表示可以用于多种编程语言理解和生成任务。

UniXcoder 适用于多种代码相关的任务，如代码补全、代码生成、代码摘要、代码搜索等。它可以捕捉代码的结构信息和语义信息，从而提供更准确的代码表示。在五个代码相关任务的九个数据集上的评估显示，UniXcoder 在大多数任务上都取得了最佳性能。此外，研究者还构建了一个新任务——零样本代码到代码搜索的数据集，用于评估代码片段表示的性能。结果表明，UniXcoder 的性能优于其他模型，且代码注释和 AST 都能有效增强模型的性能。

代码总结任务和代码注释生成任务都是为了帮助理解和解释代码的功能和行为。它们之间的相同之处，让在总结任务上有优秀表现的大模型可以在些许调整之后完成代码注释工作：两者都需要理解代码的功能和操作，以便生成有用的信息。代码总结通常会提供一个高层次的概述，描述代码的主要功能和目标。而代码注释则会更详细，解释代码的每一部分是如何工作的，以及为什么要这样做。因此，要获得详细的代码注释，需要收集到更加细粒度的代码-文本信息，这也正是第三章的工作。在之后的微调工作中，也将基于以上介绍的几种大模型的 code-summarization 任务进行进一步训练，以获取更高的精准度。

4.2 微调脚本

以上所有的大模型在开源时都会给出具有高完成度的微调脚本，以供后续工作进行地更加方便。一个典型的微调脚本通常包括以下几个部分：

1. 数据准备：选择适合的数据集，并将其分为训练、验证和测试集。

2. 模型加载：加载预训练的大语言模型。

3. 微调设置：配置微调的参数，如学习率、批次大小等。

4. 训练过程：在训练集上运行微调，调整模型的权重。

5. 评估与保存：在验证集上评估模型性能，并保存经过微调的模型。

4.2.1 数据准备

由于之前准备的数据集为 txt 文件形式，其并不能直接被各种大语言模型直接读取作为训练数据。因此，针对不同输入需求的大语言模型，需要针对地进行数据预处理。

对于 codeT5 以及 codeT5+，以下是读取 txt 文件中数据作为数据集的一种简化算法形式:在这段代码中,传入 data-file 参数的文本文件是通过 load_tokenize_data 函数加载到数据集中的。这个函数首先检查是否存在缓存数据，如果存在，就直接从缓存加载。如果不存在，它会执行以下步骤来加载和处理文本文件：

1. 使用 open 函数以只读模式打开文本文件，并读取所有行到 lines 列表中。

2. 通过 groupby 函数将 lines 分组，每个非空行的组代表一个数据样本。

3. 对于每个非空的组，将最后一行作为注释（comment），其余部分合并为代码（code），并将它们作为字典对象添加到 data 列表中。

4. 使用 AutoTokenizer.from_pretrained 加载预训练的 tokenizer。

AutoTokenizer.from_pretrained 是一个方法，它允许从预训练模型的名称或路径自动加载相应的分词器（Tokenizer）。这个方法会下载并加载所需的预训练模型及其所有相关资源，使我们可以方便地使用这些预训练模型进行自然语言处理任务。例如，如果有一个 BERT 模型的预训练版本，可以使用这个方法来加载与之对应的分词器，它会处理文本数据，将其转换为模型可以接受的输入格式。

5. 定义 preprocess_function，它接受 examples 字典，使用 tokenizer 处理 input 和 target 字段，并处理标签，使得 padding 的部分不会对损失计算产生影响。

6. 将 data 列表转换为一个字典 data_dict，包含所有输入和目标文本。

7. 使用 Dataset.from_dict 创建一个 Dataset 对象，并使用 map 方法应用 preprocess_function，批量处理数据，并移除原始的 input 和 target 列。

Dataset.from_dict 是用于将 Python 字典转换为 Dataset 对象的方法。这个方

法非常有用，因为它允许直接从内存中的数据结构创建数据集，而不需要先将数据保存到文件中。例如，如果有一个包含多个键值对的字典，每个键对应一列数据，可以使用这个方法将其转换为一个 Dataset 对象，以便进行进一步的处理和分析。

map 方法是 Dataset 对象的一个方法，它允许对数据集中的每个元素应用一个函数，并返回一个新的 Dataset 对象。这个方法在数据预处理阶段非常有用，因为它可以用来清洗、转换或增强数据。例如，可以使用 map 方法来对文本数据进行分词、添加新的特征列或者过滤掉不符合条件的数据行。

8. 最后，使用 save_to_disk 方法将处理后的数据集保存到指定的缓存目录。

算法6这个过程确保了文本文件中的数据被正确加载和预处理，以便用于模型的微调训练。

Algorithm 6 LoadDatasets

输入: 训练参数 $args$
输出: 训练数据 $train\_data$

if EXISTS($args.cache\_data$) **then**
  $train\_data \leftarrow LoadFromDisk(args.cache\_data)$
else
  $file \leftarrow Open(args.data\_file, mode = `r', encoding = `utf-8')$
  $lines \leftarrow ReadLines(file)$
  $Close(file)$
  $data \leftarrow ProcessLines(lines)$
  $tokenizer \leftarrow AutoTokenizer.from\_pretrained(args.load)$
  $preprocess\_function \leftarrow DefinePreprocessFunction$
  $data\_dict \leftarrow CreateDictionary(data)$
  $train\_data \leftarrow Dataset.from\_dict(data\_dict)$
  $train\_data \leftarrow train\_data.map(preprocess\_function, batched = True,$ *remove_columns*=['input', 'target'], $num\_proc = 32, load\_from\_cache\_file = False$)
  $SaveToDisk(train\_data, args.cache\_data)$
end if
return $train\_data$

对于 codeBERT 和 UniXcoder 两种模型，所需要的数据集格式应该是一个 jsonl 文件, 每行为一个 json 对象, 包含以下字段4-1:

```
{
  "idx": 0, // 示例索引
  "code_tokens": ["def", "fun", "(", "a", ")", ":", ...]
  "docstring_tokens": ["This", "function", "does", ...]
}
```

图 4-1　jsonl 文件格式

其中: docstring_tokens 为注释的 token 序列。Token 序列是自然语言处理（NLP）中的一个基本概念，它指的是将文本内容分解为最小的语义单元，即 token。在英文中，一个 token 通常是一个单词，但也可以是标点符号或特殊字符。例如，句子”I love you” 可以被分割成三个 token：“I”、“love” 和”you”。Token 序列则是这些 token 按原文本顺序排列形成的序列。

code_tokens 为代码的 token 序列。代码的 token 序列是指在编程语言中，源代码被分解成一系列的最小语义单元，这些单元被称为 token。在编译原理中，这个过程是词法分析的一部分，其中编译器将字符流（源代码）转换为 token 序列。每个 token 代表了源代码中的一个基本元素，如关键字、操作符、标识符、字面值等。例如，一个简单的 C 语言代码段：

```
int x = 10;
```

经过词法分析后，可能会被分解成以下 token 序列：*int*(关键字) *x*(标识符) = (操作符) 10 (字面值) ; (分隔符)

这些 token 序列是编译器进一步进行语法分析和语义分析的基础，最终生成可执行的机器代码。在代码理解和生成的机器学习模型中，token 序列被用来训练模型，以便模型能够理解代码的结构和语义。

idx 可选, 如果没有提供则会使用行号作为索引。

该数据集格式与 CodeSearchNet[17]数据集类似。在读取数据时, 代码将连接 code_tokens 并剥离换行符, 连接 docstring_tokens 并剥离换行符, 构造 Example 对象的 source 和 target 字段。

因此，对于之前的 txt 格式数据，还需要把代码和注释都转换为对应的 token 序列。此处使用了 RoBERTa 分词器[18]。RoBERTa 分词器是一种用于将文本分

割成 token 序列的工具。它是基于字节对编码（Byte Pair Encoding，BPE）的方法，这种方法可以将输入文本切分成含有最小信息熵的子词。RoBERTa 分词器能够处理自然语言语料库中的大量常见词汇，且不依赖于完整的单词，而是依赖于子词（sub-word）单元。这些子词单元是通过对训练语料库进行统计分析而提取的，其词表大小通常在 1 万到 10 万之间。RoBERTa 分词器的优势在于它能够有效地处理各种语言和领域特定的术语。

RoBERTa 分词器的基本工作原理大概分为以下步骤：

1. 加载预训练模型：首先，加载一个预训练的 RoBERTa 模型，这个模型已经在大量文本数据上进行了训练，以学习语言的统计特性。

2. 文本输入：将需要处理的文本输入到分词器中。

3. 标准化：对输入文本进行标准化处理，例如将所有字符转换为小写，去除空格和特殊字符等。

4. 字节对编码（BPE）：使用字节对编码方法将文本切分成子词（subword）单元。这个过程包括将常见的字符或字符序列合并为更大的单元，从而减少整体的词汇量，并能够有效处理未知或罕见的词汇。

5. 生成 Token 序列：将标准化后的文本切分成 token 序列，每个 token 代表文本中的一个词、子词或字符。

6. 添加特殊 Token：在序列的开始和结束处添加特殊的 token。

7. 输出 Token 序列：最终输出处理后的 token 序列，这些 token 序列可以被用于 RoBERTa 模型的下游任务，如文本分类、情感分析等。

在获得预期的含有 token 序列的 jsonl 格式数据集之后，就可以让 codeBERT 和 UniXcoder 直接加载 jsonl 文件进行下一步微调工作。

4.2.2 模型加载

在 hugging face 平台的支持下，绝大多数大语言模型都在平台上开放下载，同时平台也提供了一些通用库和函数以方便用户进行模型的加载。以 codeT5+ 模型为例，可以使用以下代码4-2进行加载：

```
from transformers import AutoModel, AutoTokenizer

checkpoint = "Salesforce/codet5p-220m-bimodal"
device = "cuda" # for GPU usage or "cpu" for CPU usage

tokenizer = AutoTokenizer.from_pretrained(
                    checkpoint, trust_remote_code=True)
model = AutoModel.from_pretrained(
        checkpoint, trust_remote_code=True).to(device)
```

图 4-2　模型加载代码

本段代码是使用 transformers 库来加载一个预训练的模型和分词器（tokenizer），用于处理代码或文本。首先，从 transformers 库中导入了 AutoModel 和 AutoTokenizer 类。transformers 库是由 Hugging Face 提供的，它包含了大量预训练的模型，可以用于自然语言处理任务。

AutoModel 和 AutoTokenizer 是 Hugging Face 的 transformers 库中的两个非常重要的类，它们用于自动化地处理与预训练模型相关的任务。

AutoModel 类是一个通用模型类，它可以根据提供的预训练模型名称或路径自动推断出想要使用的架构，并实例化为相应的模型类。例如，使用 AutoModel.from_pretrained('bert-base-cased')，它将创建一个 BertModel 的实例。这个类不能通过 __init__() 方法直接实例化，而是通过 from_pretrained 或 from_config 类方法来实例化。

AutoTokenizer 类是一个通用的分词器类，它会根据提供的预训练模型名称或路径自动推断并创建相应的分词器类实例。分词器负责将文本输入转换为模型可以理解的格式，通常包括将文本分割成单词或子词单元、添加必要的特殊标记、对序列进行填充或截断等。AutoTokenizer.from_pretrained 方法用于从预训练的分词器配置中实例化分词器。

这两个类大大简化了使用不同预训练模型的过程，用户不需要知道每个模型的具体类名，只需提供模型的名称或路径即可。这对于快速实验和部署不同的模型非常有用。

变量 checkpoint 存储了预训练模型的路径。”Salesforcecodet5p-220m-bimodal” 是模型在 Hugging Face 模型库中的标识符，表示这是一个由 Salesforce 预训练的 CodeT5+ 模型，大小为 220M 参数。device 变量用于指定模型运行的设备。”cuda” 表示使用 NVIDIA 的 CUDA 技术在 GPU 上运行模型，而”cpu” 表示在 CPU 上运行。然后创建一个 tokenizer 对象，它会从 checkpoint 指定的路径加载预训练的分词器。trust_remote_code=True 参数允许加载远程代码，这通常用于加载自定义的分词器或模型组件。最后，model 加载了预训练的模型，并将其发送到指定的 device 上。AutoModel.from_pretrained 方法用于加载模型，to(device) 方法则是将模型移动到指定的设备上，以便进行计算。

总的来说，这段代码的目的是为了加载一个预训练的 CodeT5+ 模型，并准备好在指定的设备上进行代码或文本的处理任务。对于其他 hugging face 平台上分享的模型，只需要获取其在 Hugging Face 模型库中的标识符并赋值给 checkpoint 变量即可。

4.2.3 微调设置

微调脚本设置中，最重要的是设置合适的参数值。虽然各模型提供的微调脚本中的参数不尽相同，但是除了文件路径输入以外，仍然有一些重要的通用的参数，需要根据情况进行调整，主要参数列表如下：

batch_size: 批量大小。批量大小指的是每次训练迭代中用于更新模型的样本数量。

Batch Size 的大小设置受到硬件资源的限制。如果 GPU 或 CPU 的内存不足，则需要减小 Batch Size。通常，更大的 Batch Size 需要更多的内存。较大的 Batch Size 可以加快训练速度，但可能会导致模型精度下降；而较小的 Batch Size 可以提高模型精度，但会减慢训练速度。因此，需要在训练速度和精度之间找到一个平衡点。使用较小的 Batch Size 可能会导致训练过程中的噪声更多，有助于模型逃离局部最小值，但收敛速度可能较慢。较大的 Batch Size 可能会更快地收敛，但有时候会陷入局部最小值。

在实际应用中，通常使用一些常用的 Batch Size 值（如 32、64、128、256 等 32 的倍数）作为起点。

learning_rate: 学习率。学习率是一个超参数，它决定了在每次迭代中模型

权重更新的幅度。过高的学习率可能导致模型在最优解附近震荡，而过低的学习率则会导致训练过程缓慢，甚至停滞不前。选择合适的 Learning Rate（学习率）是深度学习中一个非常重要的步骤，因为它直接影响到模型训练的效率和最终性能。

通常，初始学习率设置得较小，这样可以防止模型在训练初期因为过大的更新步长而发散。如果要调整学习率，可以在训练过程中观察损失曲线，如果损失下降过快且出现震荡，可能意味着学习率过高；如果损失下降缓慢或不下降，可能意味着学习率过低。

num_of_train_epochs: 训练代数。代数是指整个训练数据集被遍历和用于训练模型的次数。更多的代数意味着模型有更多的机会学习数据，但也可能导致过拟合，即模型在训练数据上表现很好，但在未见过的数据上表现不佳。选择合适的 Epochs 对于确保模型能够充分学习数据集而不会过拟合至关重要。如果数据集较大或复杂，可能需要更多的 Epochs 来确保模型能够充分学习数据特征。相反，对于较小或较简单的数据集，可能需要较少的 Epochs。

在实验调整中，实验不同的 Epochs 数量，并观察模型在验证集上的表现。如果连续几个 Epoch 模型的性能没有显著提升，那么可以考虑停止训练。

max_source_length: 最大源长度。最大源长度是指模型处理的输入序列的最大长度。这个参数通常根据任务的需求和模型的能力来设置。较长的序列可能包含更多的信息，但也会增加计算负担。

max_target_length: 最大目标长度。最大目标长度是指模型生成的输出序列的最大长度。这个参数需要根据预期输出的长度来调整，以确保模型能够生成完整且有意义的输出。

Max Source Length（最大源长度）和 Max Target Length（最大目标长度）的作用是确保模型能够有效处理输入和输出。如果任务涉及到长文本，比如文档摘要，可能需要一个较长的 Max Source Length。对于翻译或文本生成任务，Max Target Length 也需要足够长，以生成完整的输出。而且也需要在保证输入指令完整性和生成内容多样性之间进行权衡。例如，对于简单的指令，可以设置较短的 Max Source Length；对于复杂的指令，可能需要较长的 Max Source Length。一般来说，分析数据集，检查输入和输出的典型长可以帮助设置一个合理的长度，既不会截断太多有用信息，也不会浪费计算资源在很少出现的长序列上。

由于现在需要完成的任务是代码注释自动生成工具，也就是 code to NL（代码到自然语言）任务，因此，在数据集中，代码作为 source，注释作为我们需要的 target。根据先前收集的数据集都为代码片段和注释语句。其中代码片段的平均长度明显少于完整代码，这里所有的模型在微调时的 max source length 可以设置为 256，注释语句多为一到两句对代码内容的简短的叙述，max target length 设置为 128 即可。batch size 方面，不同模型的规模和计算量会影响到显存使用量，本文使用的机器学习设备为一块 RTX 4090 24GB GPU，其显存会对 batch size 有一定限制。为了防止显存不足而无法进行微调过程，经测试，将 CodeT5 的 batch size 设置为 32，CodeT5+ 设置为 16，Unixcoder 设置为 48，CodeBERT 设置为 32。

损失函数（Loss Function）是一个核心概念，它用于衡量模型预测结果与真实结果之间的差异。损失函数的值越小，表示模型的预测结果与真实值越接近，模型的性能也就越好。在调整 learning rate 和 epoch 的过程中，需要用到 loss 函数。损失函数，也称为代价函数（Cost Function），是一个非负实值函数，通常使用

$$L(Y, f(x)) \tag{4-1}$$

来表示。在机器学习任务中，我们通常有一组训练数据，每个数据点包含一个特征向量 x 和一个真实标签 Y。模型的目标是学习一个函数 $f(x)$，使得对于新的输入数据，模型能够给出尽可能准确的预测结果。损失函数的主要作用是提供一个衡量标准，来评估模型的预测结果与真实结果之间的差异。在模型训练过程中，通过最小化损失函数，我们可以找到最优的模型参数。

在实际训练过程中，需要结合损失函数的变化趋势来调整学习率和训练轮数：

损失曲线分析：观察训练过程中损失曲线的变化。如果损失下降过快，可能意味着学习率过高；如果损失下降缓慢或者停滞不前，可能意味着学习率过低或者已经接近最优解。

调整策略：根据损失曲线的观察结果，我们可以采取以下策略：如果损失下降过快，可以适当减小学习率，避免震荡。如果损失下降缓慢，可以适当增加学习率，或者使用学习率预热（warm-up）策略，在训练初期使用较大的学习率。

动态调整：在训练过程中，可以动态调整学习率和 epoch。例如，可以动态

监控当前的损失函数，当损失在一定 epoch 内没有显著下降时，自动减小学习率或提前结束训练。图4-3展现了微调的过程，可以从中看到，在 epoch 大于 14

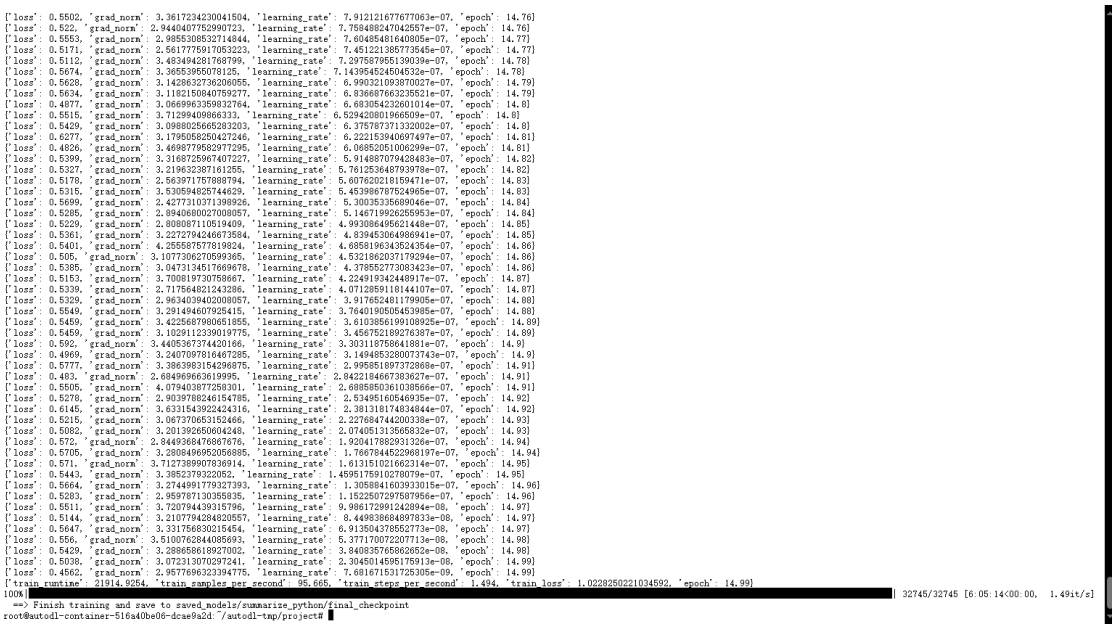

图 4-3　微调过程

时，训练过程中的 loss 值已经趋于平稳，如果继续训练会有过拟合的风险。因此 epoch 不适宜超过 14 太多。

经过实验调整，CodeT5,CodeT5+，Unixcoder 以及 CodeBERT 的初始 learning rate 都设置为 5e-5. 根据开源微调脚本的算法，不同模型会根据自身的微调情况动态调整 learning rate。CodeT5 以及 CodeT5+ 的 epoch 设置为 15，Unixcoder 设置为 10，CodeBERT 设置为 3。

4.2.4　微调结果

在模型微调结束，之后，如果想人工查看模型的输出以评估性能，可以将测试集输入到模型当中，让模型把输出结果保存。以模型 CodeT5+ 为例，图4-4展示了微调之后的模型在测试集上的一部分输出。而图4-5展示了原本人工编写的代码注释。通过对比可以看出，微调后的模型大多数情况下能保留注释原意，减少关键词丢失。

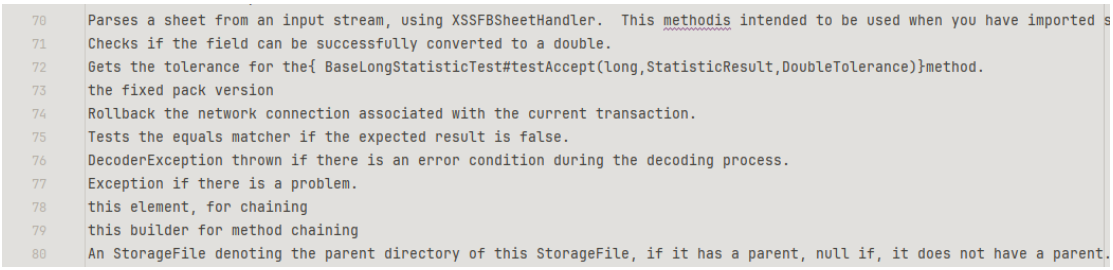

图 4-4　模型预测结果 preds

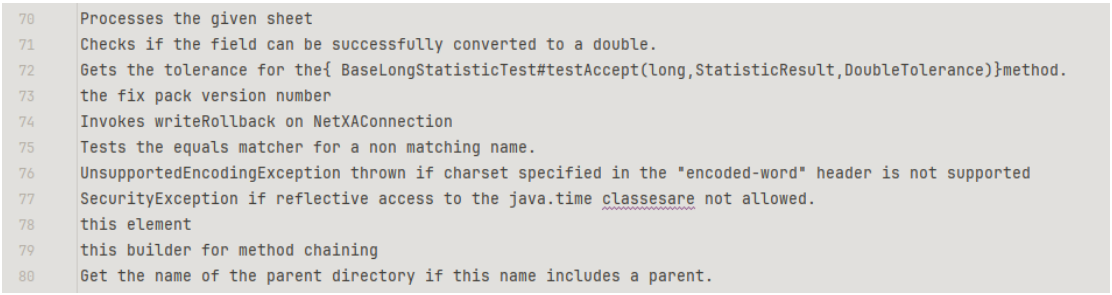

图 4-5　原注释 target

4.3　本章小结

本章介绍了模型微调过程，首先介绍了选择 codeT5 等大模型的理由，然后介绍了微调过程中需要注意的参数，并给出了相关实例。最后输入测试集到微调后的模型，对其输出进行人工考察。发现模型的代码注释生成效能相比原来有了不小的提升。至此本文涉及到的算法和技术就全部介绍完毕，下一章节将介绍评估相关内容。

第五章　实验与评估

实验部分主要关注微调模型的代码注释生成能力，因此提出以下两个研究问题：

RQ1: 微调后的模型和微调前的模型相比，能否取得更好的注释生成结果？

RQ2: 各个模型的注释生成效果如何？

5.1　评估对象

本文实验所需数据为一组包含异常语句及注释，返回值语句及注释，除去异常语句和返回值语句的其余语句及其注释的代码-注释对。这些数据从第三章的代码切割工作中获取，作为验证集的数据一共有 17470 对。

由于要评估模型的代码注释生成效果，首先将验证集的代码部分输入各模型，然后把模型预测输出结果与真实注释相比，最后通过一定的评估标准来检验各模型的效能。

5.2　评估标准

BLEU-4 分数[19]是一种用于评估机器翻译质量的指标，它通过比较机器翻译的文本和人类翻译的参考文本之间的相似度来计算。BLEU 代表双语评估替代品（Bilingual Evaluation Understudy），它是在 IBM 于 2001 年发明的，是最早声称与人类对翻译质量判断高度相关的指标之一，也是目前最流行的自动化和低成本的评估指标之一。本文需要比较机器生成代码注释的文本和人工标注的参考注释文本之间的相似度，所以用 BLEU-4 分数比较合适。

BLEU-4 分数特别关注 4-gram 的匹配程度，即连续四个词的序列。这种评分方法认为，如果机器翻译的文本中的 4-gram 与参考文本中的 4-gram 更接近，那么翻译质量就越高。BLEU-4 分数是 BLEU 评分方法中的一个变体，它专门考虑了 4-gram 的精确度。

BLEU-4 分数的计算方法包括以下几个步骤：

n-gram 匹配：首先，计算机器翻译文本中的 n-gram 与参考文本中的 n-gram 匹配的数量。n-gram 是指文本中任意连续的 n 个词组成的序列。

修正的 n-gram 精确度：为了防止机器翻译仅仅通过重复参考文本中的词汇来提高分数，BLEU-4 引入了修正的 n-gram 精确度。这意味着，如果机器翻译中的某个 n-gram 重复出现，超过了它在任何一个参考翻译中出现的次数，那么这个 n-gram 的计数将被限制在参考翻译中的最大出现次数。

简洁惩罚（Brevity Penalty, BP）：如果机器翻译的文本比参考文本短，那么它将受到简洁惩罚。这是因为过短的翻译可能忽略了原文的一些信息。简洁惩罚的计算公式 5-1 为：

$$BP = \exp(1 - \frac{r}{c}) \tag{5-1}$$

其中，r 是参考翻译的长度，c 是机器翻译的长度。

BLEU-4 分数的最终计算：最后，BLEU-4 分数是通过计算修正的 n-gram 精确度的几何平均数，并乘以简洁惩罚得到的。数学表达式 5-2 为：

$$BLEU-4 = BP \cdot \exp\left(\sum_{i=1}^{4} w_i \cdot \log p_i\right) \tag{5-2}$$

其中，w_i 是第 i 个 n-gram 精确度的权重，p_i 是修正的 n-gram 精确度。

BLEU-4 分数的优点在于可以快速自动计算，不需要人工干预。同时与人类对翻译质量的判断有较高的相关性。因此 BLEU-4 分数被广泛用于评估各种机器翻译系统和研究。

5.3 评估结果

如下图为 CodeT5-base 模型评估过程截图5-1展示：

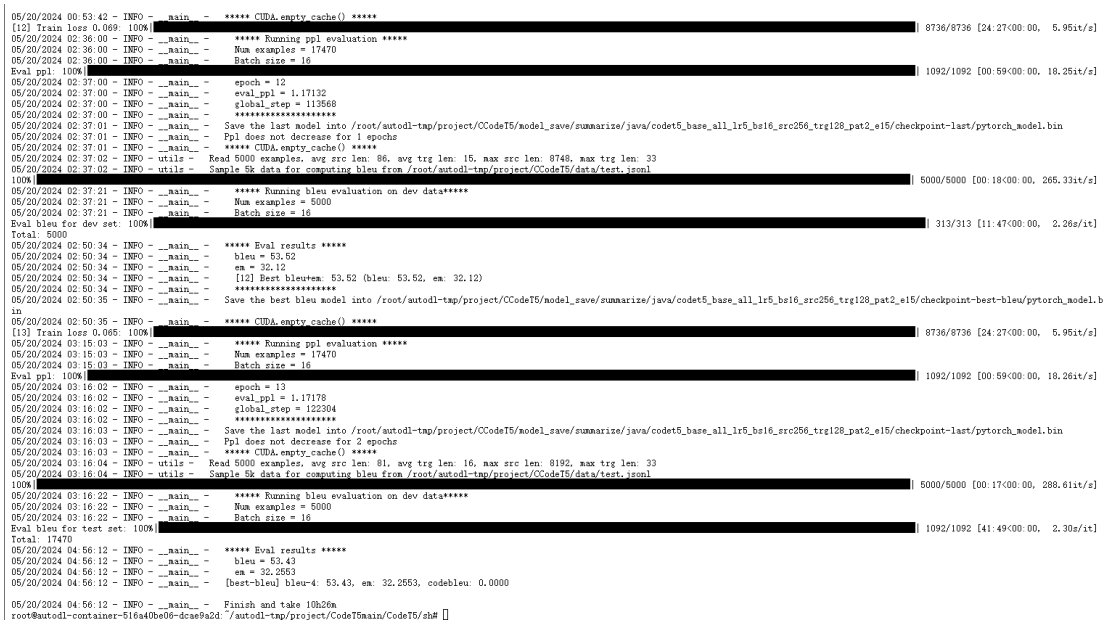

图 5-1　CodeT5-base 模型评估过程

如下图为 CodeBERT-base 模型评估过程截图5-2展示：

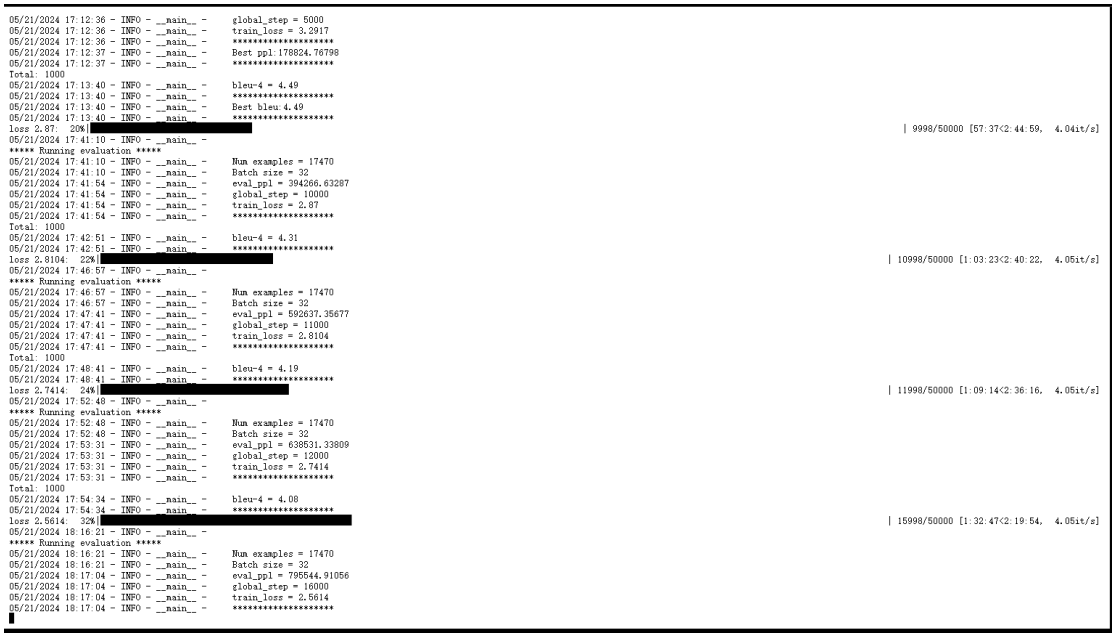

图 5-2　CodeBERT-base 模型评估过程

各个模型的对比表格如5-1：model 一栏展示了本文中所采取的模型的型号以及规模。其中，codeT5+ 模型的规模为 220m 个参数，且采用了 bimodal 型号，本型号在代码-文本对这种双峰数据有着更好的表现。UniXcoder，codeBERT，以及 CodeT5 都为 base 信号。此外为了更好地对比微调前后的差距，各模型还采取了原作者给出的已经根据不同下游任务微调完成的 checkpoints。由于本文需

要完成的是 java 方法注释生成工具，所以这些模型的下游任务需要选择在 java 语言上的 code-summarization 任务。code-summarization 任务的目的是，对于已知的较完整的代码，根据其逻辑和上下文，用自然语言总结其含义。这与我们的代码注释生成任务有相似之处。本表格的测试评估分数为 BLEU-4 分数，测试用的数据集为之前 javaslicer 切割出的 17470 对代码-注释对。由图表可以看出，

表 5-1　BLEU-4 分数对比

Model	fine-tuned	before
CodeT5+-220m-bimodal	37.34	0.71
UniXcoder-base	48.1	3.97
CodeBERT-base	48.44	4.49
CodeT5-base	53.52	4.07

在本文采集的数据集上微调前后的 BLEU-4 分数差距很大。经分析，原因可能是本文采用的数据集与原作者采用的数据集有较大的分歧。本文使用的微调前的对比模型为原模型作者发布的已经在一定数据集上微调后可以适应 java code-summarization 任务的模型，或者在原作者论文指导下，使用一定数据集尝试复现的可以适应 java code-summarization 任务的模型，本文微调时使用的模型为未被微调过的基础模型。其中，CodeT5+,CodeT5 以及 CodeBERT 模型作者使用的是 CodeSearchNet 数据集进行下游任务微调，Unixcoder 使用的是 CodeXGlue[20]数据集，二者与本文所采取的数据集不同在于：CodeSearchNet 和 CodeXGlue 中的代码-文本对中的代码皆为完整的 java 方法，文本也为 java 方法自带的文档注释；而本文将代码和文本拆分成多份代码语句小组，比先前数据集更加细粒度。更细粒度的数据集有效减少了代码和文本在模型训练过程中的信息遗忘，因此在生成注释时可以更好地概括代码中的关键信息。

总的来说，CodeT5 模型在微调后的效果是最好的。如果想使用 CodeT5 模型作为自动代码注释生成工具，可以使用 CodeT5 模型作者提供的脚本[1]，在脚本中输入微调后的模型路径来加载该 checkpoint，之后在特定的数据集上运行预测即可。

5.4　实例展示

对于以下示例代码5-3：

1　https://github.com/salesforce/CodeT5/tree/main/CodeT5

```
public void testConsistency(){
for (int i=1; i < cumulativeTestPoints.length; i++) {

 TestUtils.assertEquals(0d,distribution.probability(cumulativeTestPointsi,cumulativeTestPointsi),tolerance);
  double upper=JdkMath.max(cumulativeTestPointsi,cumulativeTestPointsi - 1);
  double lower=JdkMath.min(cumulativeTestPointsi,cumulativeTestPointsi - 1);
  double diff=distribution.cumulativeProbability(upper) - distribution.cumulativeProbability(lower);
  double direct=distribution.probability(lower,upper);
  TestUtils.assertEquals("Inconsistent probability for (" + lower + ","+ upper+ ")",diff,direct,tolerance);
}
}
```

图 5-3　实例展示

这段代码是一个测试方法，名为 testConsistency，用于验证概率分布的一致性。它执行两个主要的检查：验证累积分布函数（CDF）的性质以及验证概率的一致性。总的来说，这个方法通过对比直接计算的概率值和通过 CDF 计算得出的概率值，来确保概率分布的实现在数学上是一致的。这是统计学和概率论中常见的一种测试方法，用于验证概率模型的正确性。

原文注释如下5-4：本意为验证概率计算是否一致。

```
Verifies that probability computations are consistent
```

图 5-4　原文注释

CodeT5 模型微调前后的输出如下5-5：

微调前输出：一致概率。本输出和原注释相比缺乏了“验证”的关键词。

微调后输出：验证累积概率密度计算是否与期望值匹配。“match values”体现在两个主要的测试断言中：

```
CodeT5 before: In cons istent probability for

CodeT5 fine-tuned: Ver ifies that cumulative probability
density calculations match expected values .
```

图 5-5　CodeT5 模型输出

CDF 的性质验证：通过断言 TestUtils.assertEquals(0d, distribution.probability(x, x), tolerance)，期望对于任何给定的点 x，distribution.probability(x, x) 的计算结果应该为 0。这里的“match values”是指计算结果应该与预期的 0 值匹配。

概率的一致性验证：通过计算 distribution.cumulativeProbability(upper) - distribution.cumulativeProbability(lower) 与 distribution.probability(lower, upper) 并使用 TestUtils.assertEquals 进行比较，期望这两种不同的概率计算方法得出的结果是一致的。这里的“match values”是指两种计算方法得出的概率值应该相匹配。

总的来说，微调后的模型生成的注释没有缺少关键词，且确实的概括了代码中的关键信息。

CodeT5+ 模型微调前后的输出如下5-6：

微调前输出：意为 TestUtils.assertEquals 该断言与上下累计测试点的概率相等相关。这里只介绍了该断言的相关信息，却忽略了完整的代码信息。

微调后输出：意为使用当前测试实例数据验证概率计算是否与期望值匹配。与原文相差并不大。

```
CodeT5+ before:Test Ut. assert Equ als ( 0 d In cons istent
probability for lower and upper ulative test points are equal

CodeT5+ fine-tuned: Verifies that probability calculations
match expected valuesusing current test instance data
```

图 5-6　CodeT5+ 模型输出

Unixcoder 模型微调前后的输出如下5-7：

微调前输出：抽查。缺少信息较多。

微调后输出：验证累积概率密度计算是否与使用当前测试实例数据的期望值相匹配。与原文相差并不大。

```
Unixcoder before: test check

Unixcoder fine-tuned: Ver ifies that cumulative probability
density calculations match expected values using current
test instance data
```

图 5-7　Unixcoder 模型输出

CodeBERT 模型微调前后的输出如下5-8：

微调前输出：计算概率的概率。缺少了“验证”“一致性”关键词。

微调后输出：验证累积概率密度计算是否与期望值匹配。与原文相差并不大。

```
CodeBERT before: Computes the probability of the probabilities

CodeBERT fine-tuned:Ver ifies that cumulative probability
density  calculations match expected values .
```

图 5-8　CodeBERT 模型输出

总的来说，通过实例展示，微调工作对模型的代码注释工作有着显著的促进效果。微调后的模型相比之前，能减少关键词遗漏，并正确概括代码内容。

5.5　本章小结

在本章中我们首先说明了选择 BLEU-4 作为评估标准的理由以及优点，之后对比了 CodeT5+，CodeT5，coderBERT，Unixcoder 在相同训练集和验证集下的微调前后的 BLEU-4 分数，分析了与原模型存在差距的原因。并且选出了实验数据下表现最好的模型 CodeT5，简要说明了其作为代码注释生成工具的使用方法。最后通过实例展示，验证了微调工作的有效性。

第六章　总结与展望

6.1　本文工作总结

本文中，我们对基于大语言模型的代码方法注释生成工具进行了设计与实现。该方法运用从 java libraries 中收集的代码片段以及注释片段作为数据集，来指导大模型在代码注释生成上进行进一步微调。为了证明生成工具的效果，我们同时对多个大语言模型进行微调，并对比各自的性能，从中选出表现最优的模型。本文工作主要分为以下几个部分：

1. 对市面上广泛使用的 java libraries 进行收集和整理。主要集中在日志库，单元测试库，通用库等领域。

2. 对 java 文件中的特定代码片段和注释片段进行切割提取，组成代码-注释对，完成对数据集的构建。切割算法针对 throw 语句，try-catch 语句和 return 语句进行处理，采用 Elcipse JDT 工具对 java 方法进行细粒度分析，从而提取出需要的片段。

3. 选择合适的大模型进行微调。在选择时同时考虑了 code-to-NL 任务性能，模型规模，支持语言等特征，以提高微调效能，方便横向对比最终结果。微调时主要对微调脚本中的 learning rate，epochs 等参数进行分析和调整，以获取最佳的微调结果。

4. 对微调前后的大模型的代码注释生成性能进行对比和评估。通过对 BLEU-4 分数进行对比评估，选择出最佳的代码注释生成工具并简要说明使用方法。

6.2　未来展望

本文工作仍然存在许多不足，未来工作集中在以下几个方面：

1. 本文采用的数据集并不能完美包含所有代码编写场景。对于较为冷门小众的代码工作场景，或者较长且逻辑复杂的代码语句，模型的注释生成效果并不好。表现为有可能只生成一些短小的词句，此类词句缺乏对代码的描述信息。

未来我们将优化数据集的采集与构建，用于对长难代码进行更好的理解。

2. 本文选取的模型规模一般，可能会忽略性能较好但是规模巨大的大语言模型。两者的区别在于，同一种模型的小规模版本具有较基础的分析功能，可以解决大部分应用场景，而大规模版本对于一些边缘情况会有更好的表现。在未来将尝试微调更大规模的语言模型进行代码注释生成。

参考文献

[1] RADFORD A, NARASIMHAN K. Improving Language Understanding by Generative Pre-Training[C/OL]//. 2018. https://api.semanticscholar.org/CorpusID:49313245.

[2] RADFORD A, WU J, CHILD R, et al. Language Models are Unsupervised Multitask Learners[C/OL]//. 2019. https://api.semanticscholar.org/CorpusID:160025533.

[3] BROWN T B, MANN B, RYDER N, et al. Language Models are Few-Shot Learners[J]. ArXiv e-prints, 2020, arXiv:2005.14165: arXiv:2005.14165. arXiv: 2005.14165 [cs.CL]. DOI: 10.48550/arXiv.2005.14165.

[4] OUYANG L, WU J, JIANG X, et al. Training language models to follow instructions with human feedback[J]. ArXiv e-prints, 2022, arXiv:2203.02155: arXiv:2203.02155. arXiv: 2203.02155 [cs.CL]. DOI: 10.48550/arXiv.2203.02155.

[5] OpenAI, ACHIAM J, ADLER S, et al. GPT-4 Technical Report[J]. ArXiv e-prints, 2023, arXiv:2303.08774: arXiv:2303.08774. arXiv: 2303.08774 [cs.CL]. DOI: 10.48550/arXiv.2303.08774.

[6] ZHAO W X, ZHOU K, LI J, et al. A Survey of Large Language Models[J]. ArXiv e-prints, 2023, arXiv:2303.18223: arXiv:2303.18223. arXiv: 2303.18223 [cs.CL]. DOI: 10.48550/arXiv.2303.18223.

[7] HU X, LI G, XIA X, et al. Deep code comment generation[C]//. 2018: 200-210. DOI: 10.1145/3196321.3196334.

[8] WANG Y, WANG W, JOTY S, et al. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation[J].

ArXiv e-prints, 2021, arXiv:2109.00859: arXiv:2109.00859. arXiv: 2109.00859 [cs.CL]. DOI: 10.48550/arXiv.2109.00859.

[9] WANG Y, LE H, GOTMARE A D, et al. CodeT5+: Open Code Large Language Models for Code Understanding and Generation[J]. ArXiv preprint, 2023.

[10] GUO D, LU S, DUAN N, et al. UniXcoder: Unified Cross-Modal Pre-training for Code[EB/OL]. https://arxiv.org/abs/2203.03850.

[11] FENG Z, GUO D, TANG D, et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages[Z]. 2020. arXiv: 2002.08155 [cs.CL].

[12] LHOEST Q, VILLANOVA DEL MORAL A, JERNITE Y, et al. Datasets: A Community Library for Natural Language Processing[C/OL]//Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. Online: Association for Computational Linguistics, 2021: 175-184. arXiv: 2109.02846 [cs.CL]. https://aclanthology.org/2021.emnlp-demo.21.

[13] WOLF T, DEBUT L, SANH V, et al. Transformers: State-of-the-Art Natural Language Processing[C/OL]//Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. Online: Association for Computational Linguistics, 2020: 38-45. https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[14] HUANG K, MENG X, ZHANG J, et al. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair[C]//2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2023: 1162-1174. DOI: 10.1109/ASE56229.2023.00181.

[15] HU E J, SHEN Y, WALLIS P, et al. LoRA: Low-Rank Adaptation of Large Language Models[J]. ArXiv e-prints, 2021, arXiv:2106.09685: arXiv:2106.09685. arXiv: 2106.09685 [cs.CL]. DOI: 10.48550/arXiv.2106.09685.

[16] MANGRULKAR S, GUGGER S, DEBUT L, et al. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods[Z]. https://github.com/huggingface/peft. 2022.

[17] HUSAIN H, WU H H, GAZIT T, et al. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search[J]. ArXiv e-prints, 2019, arXiv:1909.09436: arXiv:1909.09436. arXiv: 1909.09436 [cs.LG]. DOI: 10.48550/arXiv.1909.09436.

[18] LIU Y, OTT M, GOYAL N, et al. RoBERTa: A Robustly Optimized BERT Pretraining Approach[J]. ArXiv e-prints, 2019, arXiv:1907.11692: arXiv:1907.11692. arXiv: 1907.11692 [cs.CL]. DOI: 10.48550/arXiv.1907.11692.

[19] PAPINENI K, ROUKOS S, WARD T, et al. Bleu: a Method for Automatic Evaluation of Machine Translation[C/OL]//ISABELLE P, CHARNIAK E, LIN D. Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, 2002: 311-318. https://aclanthology.org/P02-1040. DOI: 10.3115/1073083.1073135.

[20] LU S, GUO D, REN S, et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation[J]. ArXiv e-prints, 2021, arXiv:2102.04664: arXiv:2102.04664. arXiv: 2102.04664 [cs.SE]. DOI: 10.48550/arXiv.2102.04664.

致　　谢

感谢潘老师，翟老师在毕业设计过程中在学术上给予我的指导，在完成毕业设计期间我遇到过许多困难，两位老师的耐心教导让我能够顺利解决问题。同时感谢余学长对我的辅助，学长在我遇到技术难题时给予的提示和教导让我受益颇丰。

感谢 LUG@NJU提供的毕业论文 latex 模板。